Universidad Politécnica de Madrid
Master Universitario en Software de Sistemas Distribuidos y Empotrados

Diseño, Implementación y Validación del Software Embarcado y Estación de Tierra para la Misión HERCCULES Lanzada en el Globo Estratosférico BEXUS-32

PROYECTO FIN DE MASTER

Ángel Grover Pérez Muñoz

Enero – 2023

Universidad Politécnica de Madrid
Master Universitario en Software de Sistemas Distribuidos y Empotrados

Diseño, Implementación y Validación del Software Embarcado y Estación de Tierra para la Misión HERCCULES Lanzada en el Globo Estratosférico BEXUS-32

PROYECTO FIN DE MASTER

Autor: Ángel Grover Pérez Muñoz
Director: José Carlos Gamazo Real

Enero – 2023

Dedicatoria

A mis queridos padres Ángel y Maribel, por haberme apoyado en esta etapa tan importante.
A mis abuelos Juan y Marcela, por sus palabras de aliento.
A Joel, porque te quiero mucho hermano.
Por su amor y apoyo incondicional, les dedico todo mi esfuerzo.

Ángel Grover Pérez Muñoz

Agradecimientos

En primer lugar quiero agradecer al Prof. José Carlos Gamazo, por su tiempo, recomendaciones y ayuda brindada en la escritura de esta tesis de fin de máster. Al Prof. Juan Zamorano y al Dr. David González por haber depositado su confianza en mí y darme la oportunidad de seguir colaborando en el proyecto HERCCULES.

También quiero agradecer al grupo STRAST por toda la ayuda y recursos que me han prestado.

Muchas gracias a todos.

Índice

Índice de Figuras

Índice de Tablas

Glosario de Términos

AADL Lenguaje de Análisis y Diseño de Arquitectura ii, v, 7, 8, 18, 26–32, 49, 52, 68

ACS Sistema de Control de Actitud 52

ADC Conversor Analógico Digital 41–44, 46, 74, 83, 91, 104, 105, 107, 110, 113

API Interfaz de Programación de Aplicaciones 49

ASN.1 Notación Sintáctica Abstracta 1 xvii, 29–32, 50, 92–95, 99

ASSERT Automated proof based System and Software Engineering for Real-Time applications v, 28, 29, 31, 71, 74, 89, 115

ATL Laboratorio de Actitud vi, 7, 9, 38, 42–44, 46, 67, 72, 82, 83, 87, 88, 94, 105

BEXUS Balloon-borne EXperiments For University Students xvii, 2–4, 7, 54, 61, 64, 115

C&DH Comando y Gestión de Datos 37

CASE Ingeniería De Software Asistida Por Computadora xvii, 48, 49

CBD Desarrollo Basado en Componentes 12, 16–18, 28, 31, 32, 34, 71, 95, 115

CDPI Infraestructura Combinada de Gestión de Datos y Energía v, 37, 38, 40

cFS Core Flight System ii, v, 33–35, 115

CI Integración Continua 48

CONOPS Concepto de Operaciones ii, vi, 56, 61–63, 78, 79

COTS Componentes Comerciales Tomados del Estante 7, 40, 44

CV Vista de Concurrencia 30, 31

CVS Sistema de Control de Versiones 47

DEL Laboratorio Medioambiental Inferior 7–9, 46, 82, 113

DFD Diagrama de Flujo de Datos 19, 20

DLR Centro Aeroespacial Alemán 2, 3

DPV Vista de Despliegue 30, 31

DV Vista de Datos vi, 29, 31, 92

E-Box Caja Electrónica 38–40

E/S Entrada/Salida vi, 14, 37, 38, 45, 47, 105

ECSS Cooperación Europea Para La Normalización Espacial 115

EL Laboratorio Mediambiental vi, vii, 7, 9, 38, 42, 43, 62, 67, 72, 77, 79, 82, 83, 87, 88, 105, 130

ESA Agencia Espacial Europea xvii, 2, 3, 6, 23, 28, 30, 48, 50, 54, 55, 65, 68, 95, 115

FDIR Detección de Fallos y Recuperación del Aislamiento 15, 33, 37, 52, 68

FLP Future Low-cost Platform 52, 74, 78, 115

FNT Función TASTE vi, 32, 95–99

FPP F Prime Prime 30

GPIO Entrada/Salida de Propósito General vii, 41, 45–47, 50, 58, 74, 104, 127

GPS Sistema De Posicionamiento Global 7, 9, 42, 43, 50, 59, 66, 67, 74, 76, 82, 86, 104

GS Estación Terrestre vii, ix, xvii, 7, 9, 37, 45, 51, 64, 67, 68, 71, 73, 74, 79, 82, 83, 109, 110, 113, 117

GSW Software Terrestre iii, vi, 2, 64, 67–69, 71, 88, 94

GUI Interfaz Gráfica de Usuario vii, 64, 67, 68, 95, 105, 110, 117

HERCCULES Heat-transfer and Environment Radiative and Convective Characterization in a University Laboratory for Experimentation in the Stratosphere i, v–vii, xvii, 2, 3, 6–12, 37–39, 44, 46–48, 50, 53–57, 61–64, 71, 72, 74, 78, 79, 89, 91, 103, 105, 109, 113, 115, 117, 127, 129, 134

HK Housekeeping 9, 37, 82, 83, 94

HOOD Hierarchical Object Oriented Design 23

HRT-HOOD Hard Real-Time Hierarchical Object Oriented Design ii, v, 18, 23–26, 30, 32, 71, 74, 115

HRT-UML Hard Real-Time Unified Modeling Language ii, v, 26, 30, 68, 95

HTL Laboratorio de Transferencia De Calor iii, vi, vii, 6, 7, 38, 41, 62, 66, 67, 72, 77, 79, 82, 84–86, 105, 110, 131

I/F Interfaz 40, 41, 43, 46, 47, 50

I2C Inter-Integrated Circuit 37, 40, 41, 43, 45–47, 50, 62, 74, 82, 83, 103, 104

IBD Diagrama Interno de Bloques 39–43

IDE Entorno De Desarrollo Integrado 31, 48, 95

IDR Instituto de Microgravedad “Ignacio da Riva” xvii, xviii, 2, 6, 54, 55

IMU Unidad de Medición Inercial 7, 9, 42, 43, 59, 66, 67, 74, 82, 86, 104

IR Infrarroja 8, 9, 43

IV Vista de Interfaz vi, vii, 30–32, 95, 98, 99, 105, 130, 131

MBD Desarrollo Basado en Modelos 12, 17, 18, 28, 29, 31, 34, 71, 115

MDA Arquitectura Dirigida por Modelos 28

MIAT Tiempo Mínimo Entre Llamadas Consecutivas 28, 32, 78

MORABA Base Móvil de Cohetes 2, 3

NADS Sistema de Navegación y Determinación de Actitud iii, vi, vii, 7, 9, 38, 42, 43, 72, 82, 86–88, 95, 104–106, 130

NASA Administración Nacional De Aeronáutica Y El Espacio 33, 34, 69, 115

NMEA National Marine Electronics Association 43, 50, 66

OBC Computador de a Bordo ii, iv, vi, vii, xvii, xviii, 1, 2, 7, 37, 38, 40–42, 44–47, 51, 53, 56, 58, 62, 68, 69, 71, 74, 103, 104, 107, 108, 127, 128, 133, 134

OBDH Gestión de los Datos de a Bordo v, ix, 1, 24, 27, 28, 37, 44, 57–59, 64, 74, 83, 84

OBSW Software de a Bordo iii, v, vi, xvii, 1, 2, 10, 12, 20, 21, 33, 41, 45, 50–54, 56, 61, 62, 64, 66–69, 71–74, 76, 78–80, 82, 84–86, 88, 94–96, 103, 104, 107, 109, 115, 117

OOP Programación Orientada a Objetos 16, 19, 21

PCU Unidad de Control de Potencia iii, vi, ix, 9, 38, 40, 41, 67, 72, 76, 82, 86, 88, 97, 99, 101, 103, 105, 107, 109, 136, 137

PDR Revisión de Diseño Preliminar 55

PI Interfaz Provista 17, 18, 30, 32, 76, 78, 88–90, 95, 96, 99, 100, 107

POSIX Portable Operating System Interface X 49

PWM Modulación por Ancho de Impulsos 7, 40, 45–47, 50, 58, 74, 75, 104, 109, 110

REXUS Rocket EXperiments For University Students 2

RI Interfaz Requerida 16, 18, 30, 32, 77, 79, 82, 85, 86, 88, 89, 95, 97, 99

RIU Unidad de Interfaz Remota 37

RPi Raspberry Pi ix, 38, 44–47, 50, 58, 73, 104, 134

RTES Sistema de Tiempo Real Embebido i, ii, vi, 2, 10–12, 14–20, 23–28, 30, 35, 49, 51–53, 61, 64, 115

RTU Unidad Terminal Remota ii, vi, 37–42, 45, 72, 108

SBC Computador monoplaca 44, 49

SDL Lenguaje de Especificación y Descripción vi, 32, 48, 50, 97, 98

SDPU Unidad de Procesamiento de Datos Sensoriales iii, vi, vii, ix, 38, 40, 42–44, 46, 72, 83, 87, 104, 105, 107, 112, 113, 130, 135

SE Sistema Empotrado 13, 14

SNSA Agencia Espacial Nacional Sueca 2, 3, 55

SO Sistema Operativo 14, 49, 73, 74

SoC Sistema en Chip 44, 47

SPI Serial Peripheral Interface 37, 45

SSC Corporación Espacial Sueca 2, 3

STRAST Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos xvii, 2, 6, 30

SysML Lenguaje de Modelado de Sistemas xvii, 30, 39, 48

TASTE El Conjunto de Herramientas ASSERT para la Ingeniería ii–vii, xvii, 30–34, 48, 50, 51, 71, 74, 89, 92, 95–100, 104–107, 110, 115, 117, 130, 132

TC Telecomando ix, 1, 9, 24, 28, 33, 37, 62, 64, 67–69, 71–73, 77–79, 82–84, 86, 88, 93, 94, 97–99, 117

TM Telemetría ix, 1, 9, 24, 33, 37, 45, 62, 64, 67, 68, 71, 73, 79, 81–84, 88, 89, 93, 94, 103, 109, 117

TMU Unidad de Medición Térmica iii, vi, vii, 38, 40–42, 72, 90, 91, 105–107, 110–112

TT&C Telemetría, Seguimiento y Orden iii, vi, 4, 9, 73, 74, 81, 83

UART Universal Asynchronous Receiver-Transmitter 15, 37, 43, 45–47, 50, 74, 104

UEL Laboratorio Medioambiental Superior 7, 8, 46, 82, 113

UML Lenguaje de Modelado Unificado v–vii, xvii, 16–19, 21, 26, 30, 48, 52, 68, 71–73, 75–78, 80, 81, 83–88, 96, 97, 128

UPM Universidad Politécnica de Madrid xvii, 2, 6, 30, 51, 57

WCET Tiempo de Ejecución en el Peor Caso 25, 32

ZARM Centro de Tecnología Espacial Aplicada y Microgravedad 2, 3

Resumen en Español

El programa REXUS/BEXUS permite que estudiantes europeos desarrollen experimentos embarcados en cohetes sonda (REXUS) y globos estratosféricos (BEXUS). El experimento HERCCULES es uno de los seleccionados por la Agencia Espacial Europea (ESA) para el programa Balloon-borne EXperiments for University Students (BEXUS) que volará a 30 km de altitud para analizar la transferencia de calor por convección en la estratosfera y caracterizar el entorno térmico y la dinámica de misiones similares. HERCCULES ha sido desarrollado en su totalidad por estudiantes del Instituto de Microgravedad "Ignacio da Riva" (IDR) y el grupo Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos (STRAST) de la Universidad Politécnica de Madrid (UPM). Asimismo, como parte de BEXUS, HERCCULES ha recibido soporte técnico de expertos del DLR, SNSA, y la ESA.

El presente documento recoge el trabajo efectuado por el autor durante el ciclo de vida software del sistema HERCCULES. Esto incluye las actividades de requisitos, análisis, diseño, implementación, verificación y validación del software de los segmentos de vuelo y tierra. Para ello se ha realizado un estudio y selección de tecnologías, metodologías, y patrones arquitectónicos adecuados para el desarrollo de software empotrado de tiempo real. Asimismo, este trabajo incluye las tareas de integración del software sobre los diversos subsistemas, para así poder verificar y validar el software de vuelo basado en estándares del sector espacial. El entorno de desarrollo incluye una Raspberry Pi modelo 4B (RPi) para el OBC, y una plataforma Linux para la estación terrestre (GS).

HERCCULES cuenta con dispositivos analógicos y digitales como barómetros, radiómetros, termómetros, y calefactores que son controlados por el OBSW. El software del sistema HERCCULES se ha desarrollado con TASTE de la ESA. Dicho conjunto de herramientas impulsa la práctica de metodologías como el desarrollo basado en modelos y componentes. Asimismo, TASTE se empleó para el modelado y diseño del software embarcado que incluye tecnologías como Notación Sintáctica Abstracta 1 (ASN.1) para la (de-)serialización y (des-)codificación de datos; AADL para los aspectos de la arquitectura y concurrencia; SDL para el modelado de máquinas de estado; entre otros. También se emplearon herramientas de ingeniería de software asistida por computadora (CASE) como Visual Paradigm y OSATE para el modelado del software y sistema en el UML, Lenguaje de Modelado de Sistemas (SysML), y AADL. En lo referente al control de versiones y CI/CD, se usaron las herramientas *git* y *GitLab*.

En conjunto, el software y computador de a bordo serán responsables del control de la plataforma, el procesado de telecomandos, y la gestión de los datos. Se trata de un sistema distribuido,

por la comunicación de los segmentos de vuelo y tierra; con una arquitectura maestro-trabajador para el segmento de vuelo, ya que los sensores y actuadores responden a las órdenes del OBC. La validación del sistema precisó de pruebas en cámaras ambientales (como de vacío) del IDR de la UPM. Asimismo, las pruebas unitarias y de integración se realizaron sobre placas electrónicas elaboradas específicamente para el proyecto. Los resultados de este trabajo suponen la completitud e involucración por parte del autor en un proyecto multidisciplinar, además de la obtención de patrones arquitectónicos, aplicación de metodologías y tecnologías idóneas en aplicaciones del sector espacial.

Palabras clave en Español: *TASTE, software de a bordo, patrones de diseño, desarrollo basado en modelos, desarrollo basado en componentes.*

Resumen en Inglés

The REXUS/BEXUS program allows European students to develop experiments onboard scientific rockets (REXUS) and stratospheric balloons (BEXUS). The *Heat-transfer and Environment Radiative and Convective Characterization in a University Laboratory for Experimentation in the Stratosphere (HERCCULES)* is one of the experiments selected by the European Space Agency (ESA) for the Balloon-borne EXperiments for University Students (BEXUS) program that will fly at 30 km altitude to analyse convective heat transfer in the stratosphere and characterize the thermal environment and dynamics of similar missions.

This paper presents the work developed by the author during the software life-cycle process of the HERCCULES system. This covers the activities of requirements, analysis, design, implementation, verification and validation of the flight and ground segment software. To that end, the author performed an study and selection of technologies, methodologies, and architectural patterns suitable for the development of real-time embedded software. In addition, this work includes the software integration activities of all subsystems, to verify and validate the flight software based on space industry standards, ECCS. The development environment includes a Raspberry Pi model 4B (RPi) as the on-board computer (OBC), and a Linux execution platform for the ground station (GS).

HERCCULES is equipped with analogue and digital devices such as barometers, radiometers, thermometers, and heaters that are controlled by the on-board software (OBSW). The HERCCULES system software has been developed with the TASTE (The ASSERT Set of Tools for Engineering) tool-chain from ESA. TASTE encourages the practice of methodologies such as model and component-based development. Furthermore, TASTE was used to model and design the OBSW, including technologies such as ASN.1 for data (de-)serialization; AADL for architecture and concurrency aspects; SDL for state machine modelling; among others. Computer Aided Software Engineering (CASE) tools such as Visual Paradigm and OSATE were used for the software and system modelling in the Unified Modeling Language (UML), Systems Modeling Language (SysML), and AADL. Regarding version control management and CI/CD, *git* and *GitLab* were used.

In brief, the OBSW and OBC will be responsible for platform control, telecommand processing, telemetry generation, and data management. HERCCULES is a distributed system, the flight and ground segments communicate via TCP/IP sockets. The flight segment follows a master-worker architecture since the sensors and actuators respond to the commands sent by the OBC.

The validation of the system required was carried out in environmental chambers (such as vacuum, TVAC) of the “Ignacio da Riva” Microgravity Institute (IDR) from UPM. Unit and integration tests were also performed on electronic boards developed specifically for the project. The results of this work represent the completeness and involvement of the author in a multidisciplinary project conformed mainly by aerospace engineers. In addition, this work includes the obtention of architectural patterns, application of methodologies and technologies suitable for space applications.

Palabras clave en Inglés: *TASTE, on-board software, design patterns, model-based development, component-based development.*

Capítulo 1

Introducción

A inicios de la década de los 80 surge la denominada sociedad de la información. Esta trae consigo el advenimiento de la era digital, que se ha visto impulsada por el uso práctico de los sistemas informáticos para resolver problemas. La progresiva miniaturización de componentes electrónicos impulsó el uso de computadores empotrados en todo tipo de objetos y dio origen a los denominados *sistemas embebidos* o *sistemas empotrados*. Los sistemas embebidos son un tipo especial de sistema informático que, a diferencia de los sistemas de propósito general, forman parte de otros sistemas y están diseñados para realizar una función específica [1]. Tienen un gran abanico de aplicaciones y están cada vez más presentes en nuestras vidas. Se pueden encontrar en lavadoras para la apertura y cierre de sus válvulas, en dispositivos médicos como marcapasos, y en sistemas más complejos como vehículos para la supervisión y el control de sus subsistemas.

Los sistemas espaciales no son una excepción. Desde su primer uso a inicios de la década de los 60 en las misiones del programa Gemini de la NASA, los computadores y software empotrados han evolucionado y son cada vez más complejos. Prueba de ello es que el control de la *plataforma* y *carga útil* de un vehículo espacial depende de la funcionalidad y características del OBSW que, a su vez, está sujeto a las prestaciones ofrecidas por el OBC. Por tanto, si bien la carga de pago y plataforma determinan el rendimiento de las misiones espaciales, el software y computador de control son fundamentales para el éxito de la misión. Eickhoff identifica en [2] que los dispositivos de la carga útil, el control de la plataforma, y el OBSW desplegado en el OBC conforman la cadena de elementos clave para el funcionamiento correcto de una misión espacial.

Principalmente, las funciones del OBSW para con la plataforma incluyen el manejo de sus dispositivos, buses, e interfaces; gestión de los datos de a bordo (OBDH); y comunicación con el segmento de tierra para el intercambio de telemetría (TM) y TCs. En cuanto a funciones de la carga de pago, el OBSW incluye la gestión de datos y tareas específicas que precisen sus instrumentos [3]. La carga de pago, también denominada carga útil, es la parte más importante de un vehículo espacial ya que se encarga del desempeño de las funciones para las que se diseñó. Estas incluyen diversos instrumentos científicos como dispositivos óptico-electrónicos, radiómetros, ruedas de reacción, magnetopares, y demás sensores/actuadores dependientes de la misión. Por ejemplo, los satélites de comunicación tienen como carga útil las antenas de

comunicaciones, los receptores, y transmisores. Por otro lado, la plataforma constituye la parte restante del vehículo que da soporte a la carga de pago, proporcionando una infraestructura que incluye la distribución de potencia, control térmico, servicios de comando y control, estructura mecánica, entre otros.

El presente documento recoge el trabajo realizado por el autor en el proyecto HERCCULES [4], uno de los tres experimentos seleccionado por la ESA para volar a unos 30 km de altitud a bordo del globo estratosférico BEXUS-32 [5, 6]. El objetivo principal de HERCCULES es caracterizar la transferencia de calor por convección en plataformas similares y realizar un estudio de la dinámica y el entorno térmico de la misión durante las fases de ascenso y flote. HERCCULES está liderado por el IDR [7] y ha sido desarrollado en colaboración con el grupo de investigación de STRAST [8], del cual el autor es miembro. El proyecto ha sido desarrollado íntegramente por estudiantes universitarios bajo la supervisión de profesores de la UPM y expertos de la ESA.

El trabajo del autor incluye las actividades de requisitos, análisis, diseño, implementación, verificación y validación del OBSW y software terrestre (GSW), así como la elección de componentes hardware para dicha misión, tanto del OBC como de algunos sensores digitales. Durante la (i) *especificación de requisitos* se crean los modelos de análisis que implica la comprensión y el estudio del proyecto HERCCULES para su futura implementación y validación. El (ii) *análisis* cubre el estudio de metodologías, tecnologías, patrones arquitectónicos y de diseño para el desarrollo de software en aplicaciones espaciales. Todo ello en base al estudio de misiones satélites previas. Por otro lado, el (iii) *diseño* se efectuó en base a los resultados del análisis, y la (iv) *implementación* ciñéndose a recomendaciones y directrices adecuadas para software de Sistema de Tiempo Real Embebido (RTES). Finalmente, la (v) *verificación* y (vi) *validación* del sistema se llevaron a cabo siguiendo una estrategia de pruebas incrementales, ambas definidas por la metodología de desarrollo.

1.1. Contexto del Proyecto HERCCULES

1.1.1. Programa REXUS/BEXUS

REXUS/BEXUS [5] es un programa alemán-sueco que permite a estudiantes de universidades europeas realizar experimentos científicos y tecnológicos a bordo de cohetes y globos sonda a través de los programas Rocket EXperiments for University Students (REXUS) y BEXUS, respectivamente. Ambos se realizan en virtud de un acuerdo bilateral entre el Centro Aeroespacial Alemán (DLR) y la Agencia Espacial Nacional Sueca (SNSA). La mitad de la carga útil está reservada para estudiantes alemanes, mientras que la mitad sueca se pone a disposición de estudiantes de otros países europeos mediante una colaboración con la Oficina de Educación de la ESA. Por otro lado, EuroLaunch –una cooperación entre la Corporación Espacial Sueca (SSC) y la Base Móvil de Cohetes (MORABA) del DLR– es responsable de la gestión de la campaña y las operaciones de los vehículos de lanzamiento. Los experimentos se realizan en su totalidad por los estudiantes y a lo largo del proyecto reciben apoyo técnico de expertos del DLR, la ESA, la SSC, y el Centro de Tecnología Espacial Aplicada y Microgravedad (ZARM). Los experimentos se lanzan desde el Centro Espacial de Esrange en Kiruna, situado al norte de Suecia.

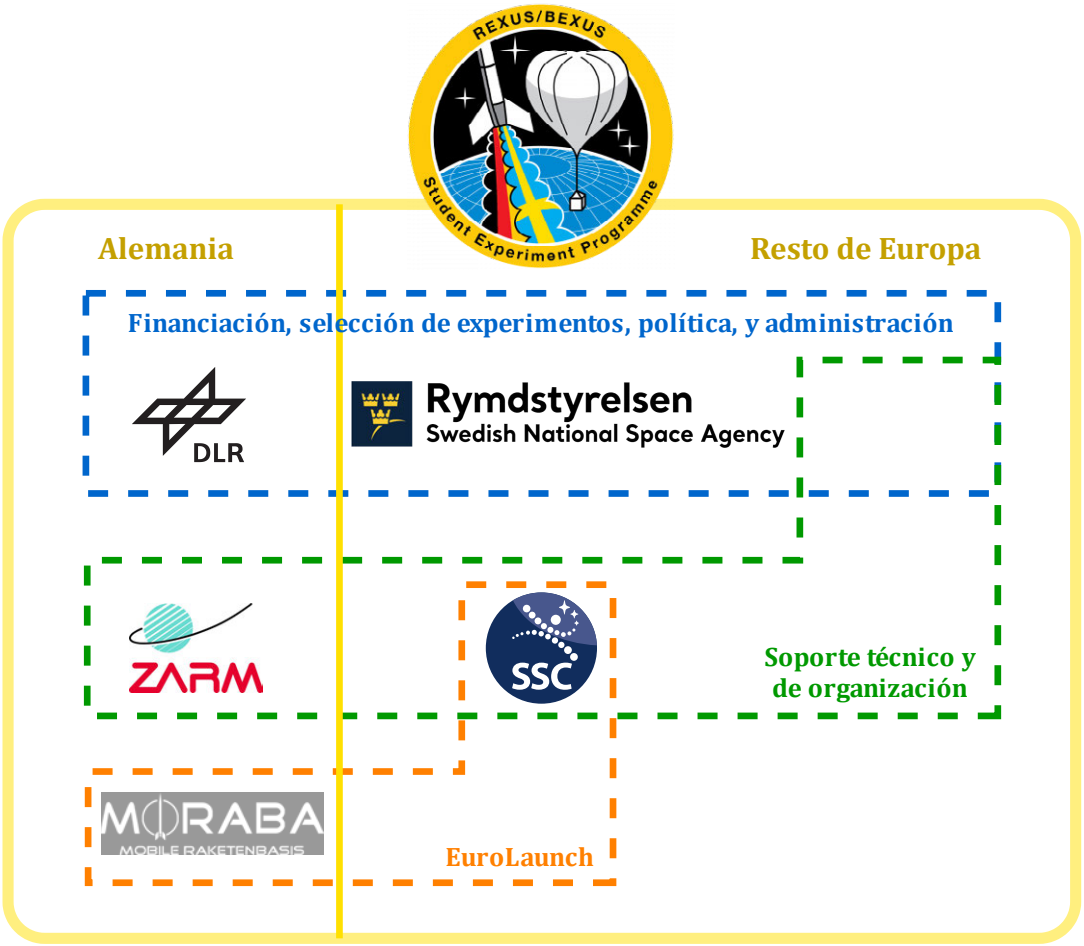


Figura 1.1: Entidades que participan en el programa REXUS/BEXUS [9].

La figura 1.1 muestra las entidades que posibilitan y gestionan el programa REXUS/BEXUS. Los patrocinadores principales del programa son el DLR y la SNSA; mientras que la ESA participa mediante un acuerdo con la SNSA. Los tres proveen soporte financiero, administrativo, y técnico a los estudiantes. Por otro lado, el ZARM (contratado por el DLR) y el centro SSC (contratado por la ESA) proporcionan soporte a los estudiantes, y se encargan de la integración de los experimentos. En síntesis, los experimentos alemanes están dirigidos por el DLR, ZARM, y MORABA; mientras que el resto de experimentos europeos por el SNSA, la ESA, y el SSC como se presenta en la figura 1.1.

1.1.2. Misión BEXUS-32

El presente año (2023) se lanzarán dos globos sonda del programa BEXUS en las campañas **BEXUS-32** y BEXUS-33 en septiembre/octubre de 2023 desde el centro SSC al norte de Suecia. En conjunto transportarán como carga de pago un total de seis experimentos, tres en cada campaña [10]: HERCCULES volará junto a los experimentos ROMULUS y HERMES en la campaña BEXUS-32; mientras que ALMA, SPACIS, y TOTORO en BEXUS-33.

Los experimentos se alojan en una góndola de dimensiones 1.16 m x 1.16 m x 0.84 m que puede elevar un máximmo de 300 kg aproximadamente, con cargas útiles que pesan entre 30

y 112 kg. El globo es un 35 SF de "Zodiac aerospace" que puede ocupar un volumen de hasta 12 000 m^3 para alcanzar los 30 km de altitud. En la figura 1.2 se puede apreciar un esquema general del vehículo de vuelo usado en las campañas BEXUS. Se pueden apreciar diversos subsistemas incluidos en él para el seguimiento, adquisición de datos, y aterrizaje. Asimismo, los operadores ponen a disposición de los experimentos el sistema de telemetría, seguimiento y orden (TT&C) E-Link que abstrae la comunicación con la estación de tierra mediante el protocolo TCP/IP. El consumo de energía también es responsabilidad de los operadores.

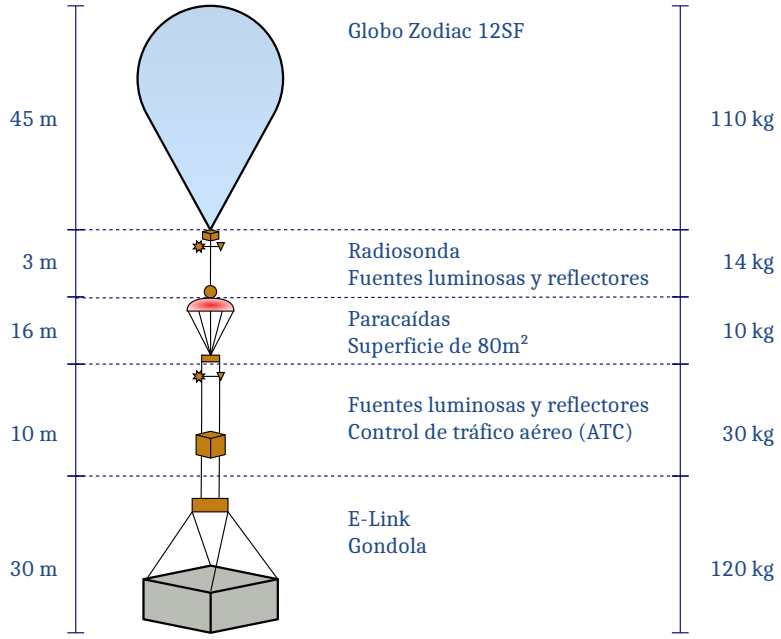


Figura 1.2: Esquema explicativo del globo estratosférico empleado en misiones BEXUS [11, pág. 19].

La duración del vuelo es de 2 a 5 horas y, a lo largo de su operación, los experimentos pasan por seis etapas. Estas definen el comportamiento de los experimentos (también llamado *concepto de operaciones*) y están definidas por la trayectoria recorrida por el vehículo:

1. **Lanzamiento**. En esta fase, la carga útil, alojada en la góndola, se sostiene mediante el vehículo Hercules[1] y se libera cuando se completa el inflado del globo (con Helio).

2. **Ascenso**. Tras el lanzamiento, el globo entra en la fase de ascenso y asciende con una velocidad media de 5 m/s. Esta fase dura aproximadamente 1.5 horas, dependiendo de la altitud requerida por la misión.

3. **Flote**. Esta es la fase más longeva y alcanza un rango de 25-33 km de altitud con una variación de $\pm$ 200 m. El tiempo medio en esta fase varía en el rango de 1 a 5 horas, dependiendo de la velocidad del viento.

4. **Descenso**. En esta fase se activa el "cutter" que separa el sistema de aterrizaje (paracaídas) del globo. Tras un breve tiempo de caída libre, el paracaídas se infla, lo que produce

[1]No confundir con HER**CC**ULES, el acrónimo del experimento que incluye doble **C**.

una fuerza de tracción por la desaceleración brusca. Al principio, la velocidad de descenso es alta, mientras que poco antes de llegar a tierra se estabiliza a unos 7-8 m/s.

5. **Aterrizaje**. El aterrizaje se produce con una velocidad de aproximadamente 7m/s (100g). La góndola cuenta con amortiguadores (denominados *crash-pads*) para evitar daños en los experimentos.

6. **Recuperación**. En la última etapa se recuperan los experimentos y datos registrados por cada experimento para el futuro pos-análisis.

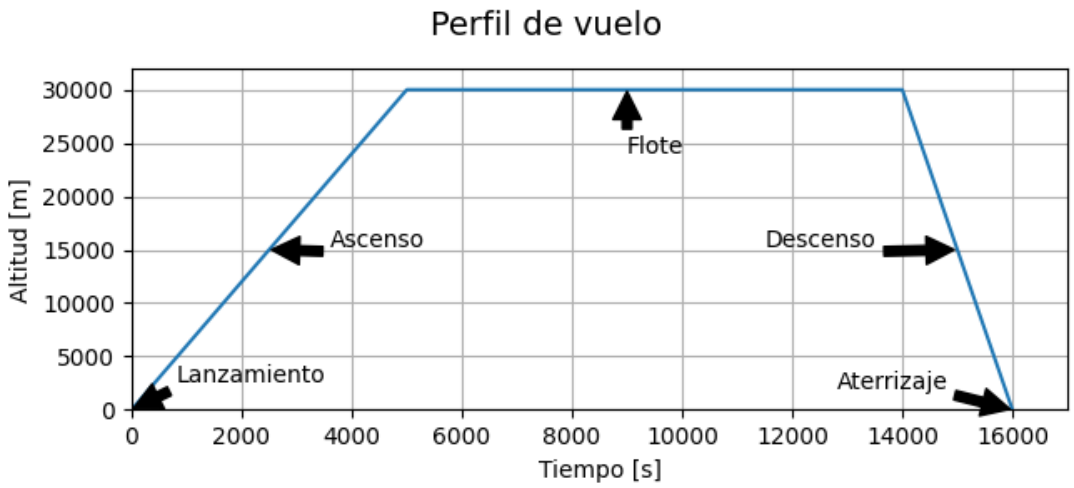


Figura 1.3: Perfil de vuelo de BEXUS [11, pág. 22].

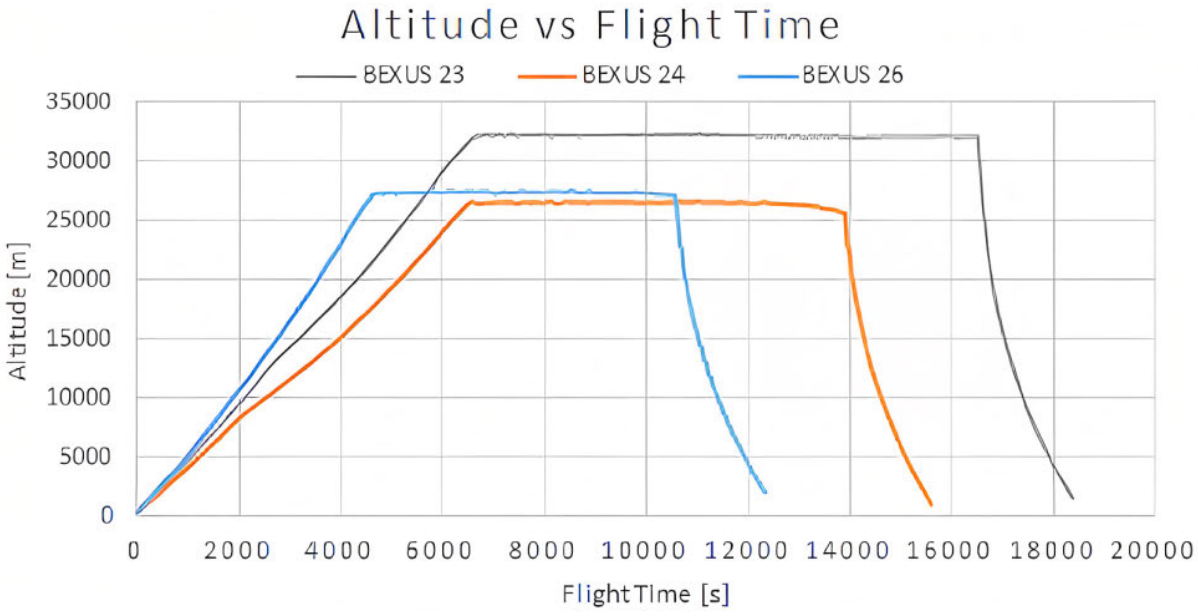


Figura 1.4: Altitud v/s tiempo de vuelo de las campañas BEXUS 23, 24, y 26 [11, pág. 25].

La figura 1.3 ilustra el perfil de vuelo típico de campañas BEXUS. Esta muestra la evolución de la altitud del vehículo a lo largo del tiempo e identifica gran parte de las fases descritas previamente, desde el lanzamiento (1) hasta el aterrizaje (5). Este perfil de vuelo se ve reflejado

en la trayectoria de vuelos previos, concretamente de las campañas BEXUS 23, 24 y 26 (véase figura 1.4). Se aprecia que el vehículo alcanza una altitud media de 26 km durante la fase de flote. Mientras que las fases de ascenso y descenso, se pueden identificar en los puntos de inflexión de la trayectoria, al inicio y final de la gráfica, respectivamente.

1.2. Proyecto HERCCULES

El proyecto HERCCULES surge de la experiencia del IDR en el control térmico de misiones de globos estratosféricos con el propósito de caracterizar la transferencia de calor por convección, el entorno térmico, y la dinámica en plataformas similares [12, 13]. La ESA seleccionó el proyecto HERCCULES propuesto por los grupos IDR y STRAST de la UPM para la campaña BEXUS-32. El proyecto ha sido desarrollado por un equipo multidisciplinar de estudiantes de grado, máster, y doctorado de la UPM, con el apoyo económico y técnico del personal docente e investigador de ambos grupos. Las siguientes subsecciones discuten los objetivos del proyecto HERCCULES y los subsistemas que lo componen.

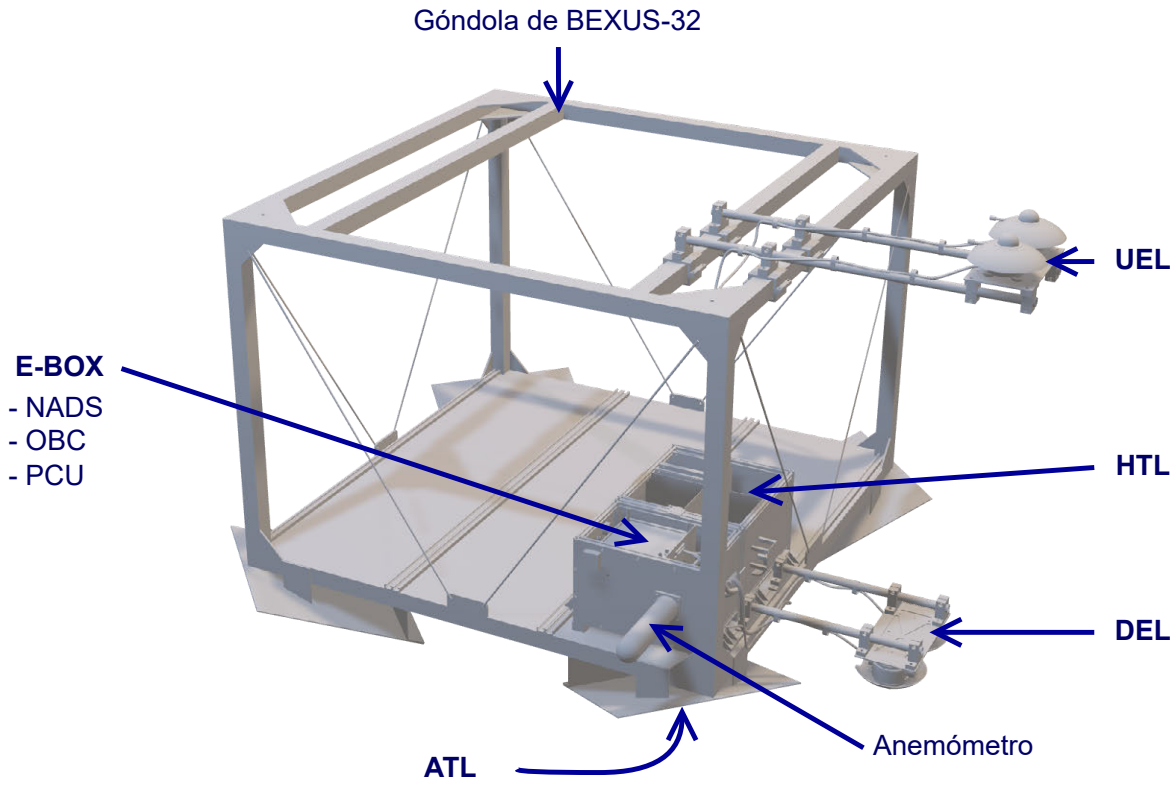


Figura 1.5: Vista general de HERCCULES dentro de la góndola.

1.2.1. Objetivos de HERCCULES

El **objetivo principal** de HERCCULES es caracterizar el entorno térmico y radiativo en la estratosfera para mejorar el modelado y análisis térmico de los sistemas estratosféricos. Para ello, HERCCULES cuenta con el *HTL*, que a su vez incluye varios experimentos de equipados con placas de aluminio configuradas en distintas posiciones, controladas por calefactores

y 28 sensores de temperatura analógicos (termistores PT1000). Por otro lado, la medición entorno radiativo es posible gracias al *EL*, que cuenta con dos piranómetros y pirgeómetros para medir el albedo, la radiación infrarroja terrestre, la radiación solar directa y la radiación del cielo. Por ello, EL se subdivide en dos subsistemas que apuntan al nadir y cenit, denominados Laboratorio Medioambiental Inferior (DEL) y Laboratorio Medioambiental Superior (UEL), respectivamente.

Un **objetivo secundario** durante el vuelo es probar las prestaciones de un sensor nadir adaptado para cargas útiles estratosféricas, incluidos en el *ATL*. Asimismo, se utilizan diferentes sensores para caracterizar el entorno, el vuelo, el rendimiento y estado de los instrumentos. Concretamente, la dinámica del vuelo se obtiene del NADS equipado con una Unidad de Medición Inercial (IMU) y un sensor Sistema de Posicionamiento Global (GPS). La figura 1.5 muestra una perspectiva general del experimento situado dentro de la góndola BEXUS-32, donde se pueden apreciar los subsistemas antes mencionados y la estructura mecánica de HERCCULES.

1.2.2. Subsistemas y Laboratorios

Para cumplir con los objetivos de la misión, HERCCULES está dividido funcionalmente en cinco subsistemas que interactúan con el entorno físico. Estos subsistemas se han diseñado empleando Componentes Comerciales Tomados del Estante (COTS) y se encargan de medir datos relevantes para el análisis térmico como la radiación, la temperatura o la presión, así como información relevante para la determinación de la actitud durante las fases de ascenso y flote. Además de sensores, estos subsistemas están equipados con actuadores para controlar la temperatura de placas de aluminio y de los instrumentos de la carga útil. El diagrama de contexto del sistema HERCCULES modelado en Lenguaje de Análisis y Diseño de Arquitectura (AADL) (véase figura 1.6) muestra la relación entre el OBC central y los subsistemas situados en la periferia. La dirección de las flechas muestra el flujo de los datos y los subsistemas muestran los tipos y cantidad de sensores y actuadores que contienen.

A continuación se ofrece una descripción general de todos estos subsistemas. Puede encontrar documentación más detallada sobre cada subsistema y sus dispositivos en [14].

- El **laboratorio HTL** tiene como objetivo cuantificar la transferencia de calor a través del aire durante las fases de ascenso y flote de la misión. Por ello, el experimento está equipado con veintiocho termistores analógicos PT1000 y cuatro calentadores de silicona controlados mediante Modulación por Ancho de Impulsos (PWM). Este subsistema se divide a su vez en cuatro experimentos compuestos por varias placas de aluminio, cuya temperatura es monitorizada por los veintiocho termistores y controlada individualmente por los calefactores. El controlador del experimento se configura en diferentes modos de operación en función de las temperaturas de las placas y la fase actual de la misión (inferida de la presión barométrica absoluta). Los datos sobre temperatura y potencia disipada son recogidos por el software de vuelo para el futuro análisis y se envían periódicamente a la GS, de modo que los operadores de HERCCULES puedan inspeccionar el estado del experimento en tiempo real.
- El **laboratorio medioambiental** –EL por sus siglas en Inglés– tiene por objeto caracterizar el entorno térmico midiendo la radiación térmica, la presión atmosférica absoluta y la velocidad relativa del viento. Para ello, este subsistema está equipado con dos tipos de

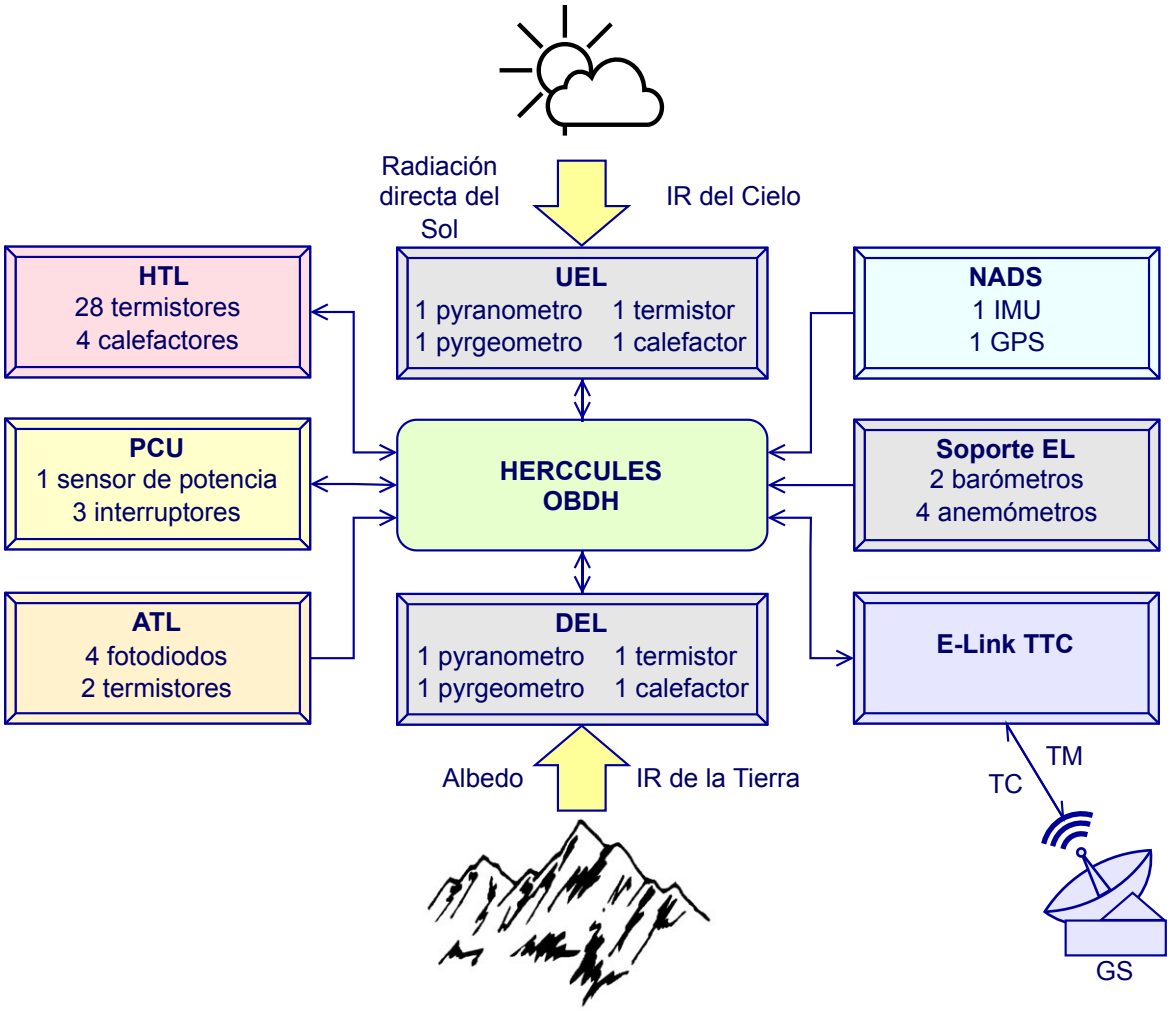


Figura 1.6: Diagrama de contexto del sistema HERCCULES modelado en AADL.

radiómetros: piranómetro y pirgeómetro. Estos son componentes COTS utilizados habitualmente en estaciones meteorológicas. Además de los radiómetros, hay dos barómetros absolutos que sirven para medir la presión atmosférica, y cuatro barómetros diferenciales empleados para medir la velocidad relativa del viento y cuantificar la fuerza de convección. En concreto, este laboratorio organiza todos estos equipos en tres compartimentos que están separados en función de su posición respecto a la estructura mecánica de la góndola:

- Laboratorio Medioambiental Superior o UEL: está situado en la parte superior de la góndola apuntando al cielo. Contiene un piranómetro, un pirgeómetro para medir la radiación solar directa y radiación infrarroja (IR) del cielo. Adicionalmente, incluye un termistor PT1000 y un calefactor de silicona para controlar la temperatura tanto del piranómetro como del pirgeómetro.

- Laboratorio Medioambiental Inferior o DEL: se compone de la misma cantidad y tipo de equipos que el UEL, pero está situado en la parte inferior de la góndola apuntando a la tierra. Los radiómetros miden el albedo y la radiación IR de la Tierra.

  - Soporte EL: está ubicado en el interior de la góndola y contiene los dos barómetros y los cuatro sensores de velocidad del viento (anemómetros).

- El **laboratorio ATL** consiste de un sensor Nadir, basado en cuatro sensores IR COTS, que proporcionan mediciones de la radiación infrarroja procedente de su Campo de Visión apuntando hacia la Tierra. Por lo tanto, el experimento se coloca cerca del subsistema DEL. Todas las mediciones se adquieren durante las fases de ascenso y flote del vuelo. Además de los fotodiodos IR, el experimento contiene dos termistores para corregir las medidas durante el posprocesado. Los resultados del experimento y el diseño del sensor Nadir se validan comparando la dirección Nadir esperada y la real. Para ello, la dinámica del globo se estima utilizando mediciones adicionales del subsistema de Navegación y Determinación de Actitud, que se explica a continuación.

- El **subsistema NADS** da soporte para validar los resultados del ATL. Proporciona datos de actitud del experimento durante las fases de ascenso y flote. Dichos datos se adquieren a partir de un sensor GPS y una IMU que, a su vez, se compone de un acelerómetro, un giroscopio y un magnetómetro. El sensor GPS proporciona información de tiempo, posición, velocidad y aceleración cada segundo. Mientras que la IMU mide la aceleración lineal del experimento, la intensidad del campo magnético y los vectores de velocidad angular.

- La **PCU** es responsable de la distribución, regulación y control de la fuente de alimentación. La PCU no está equipada con una batería interna, sino que es alimentada por un sistema externo proporcionado por los operadores de la campaña BEXUS-32. Durante el vuelo, dicha interfaz está conectada a un paquete de baterías externas de 28.8v y 1mA (13A h), por lo que los requisitos de voltaje de HERCCULES y el consumo de energía están limitados por las características de la batería BEXUS-32. La PCU consta de tres convertidores DC-DC que distribuyen los 28.8v de entrada en tres niveles distintos: líneas de 12v, 5v y 3.3v. Además, la PCU contiene un sensor digital de temperatura y otro de tensión/corriente, cuyas mediciones se recogen como datos de Housekeeping (HK).

- El sistema de TT&C **E-Link** no forma parte de HERCCULES, sino que es un subsistema externo proporcionado por los operadores de BEXUS-32 y que da soporte a la comunicación entre los segmentos de tierra y vuelo. El segmento de vuelo de HERCCULES se comunica con la GS a través de E-Link. Durante el vuelo, se transmite periódicamente TM a la GS. Esta incluye datos científicos obtenidos de los sensores, datos complementarios o TM de *HK*, el estado de los actuadores, los modos del sistema y los experimentos, entre otros. Asimismo, HERCCULES cuenta con un modo de funcionamiento autónomo que permite controlar los dispositivos sin intervención del operador. Alternativamente, existe un modo manual que permite a los operadores configurar el funcionamiento de los experimentos mediante el envío de TCs.

1.3. Estructura del la Tesis de Máster

Obviando este capítulo introductorio, los apéndices, y bibliografía; la tesis está estructurada en diez capítulos que se resumen a continuación:

- El capítulo 2 resume tanto los objetivos generales como específicos de esta tesis.

- El capítulo 3 presenta los conceptos teóricos cruciales para el correcto entendimiento de esta tesis.
- El capítulo 4 se corresponde con el estado del arte y presenta las metodologías y tecnologías apropiadas para el desarrollo de software de RTES.
- El capítulo 5 describe los materiales empleados para el desarrollo del proyecto HERCCULES. Esto incluye materiales hardware, software, y referencias bibliográficas adaptadas como base del conocimiento para el desarrollo del proyecto.
- El capítulo 6 incluye la gestión del proyecto HERCCULES desde el punto de vista del desarrollo de software. Este capítulo incluye la metodología o ciclo de vida empleado, la planificación en base a hitos, y el presupuesto de los materiales hardware y software.
- El capítulo 7 se centra en el análisis del sistema e incluye una descripción de la funcionalidad a nivel de sistema y los requisitos elicitados para el software de vuelo y tierra.
- El capítulo 8 aborda el sistema desarrollado, es decir, el núcleo de la presente tesis. Este capítulo resume la arquitectura software de alto y bajo nivel poniendo énfasis en el desarrollo del OBSW.
- El capítulo 9 describe el proceso de validación que se ha seguido para comprobar que el software del sistema HERCCULES cumple con la especificación y requisitos del sistema.
- El capítulo 10 presenta los resultados experimentales obtenidos tras desarrollar el sistema. Dichos resultados se obtienen de los procesos de verificación y validación e incluye medidas de rendimiento y otras métricas que permiten analizar los resultados obtenidos.
- El capítulo 11 incluye las conclusiones y limitaciones de este trabajo a partir de lo indicado en los resultados, el capítulo anterior. Asimismo, las conclusiones están en relación y justifican los objetivos que se han planteado en esta tesis.

Capítulo 2

Objetivos

A continuación se presentan los objetivos de la tesis de máster. Primero, se define el objetivo general que establece el marco de trabajo. Seguidamente, se describen los objetivos específicos que definen los resultados que se pretenden conseguir para la realización del objetivo general.

2.1. Objetivo General

El objetivo general de la presente tesis es **implementar el software de vuelo y tierra del sistema HERCCULES**. Para ello, se debe efectuar una serie de procesos fundamentales que incluyen el análisis, diseño, desarrollo, verificación y validación del sistema. Estos, a su vez, incluyen actividades más complejas como por ejemplo: especificación de requisitos, diseño arquitectónico, diseño de bajo nivel, pruebas de integración, entre otros.

2.2. Objetivos Específicos

OE1 Analizar tecnologías apropiadas para el desarrollo de software en el sector espacial.
Este objetivo específico implica actividades propias del proceso de "vigilancia tecnológica", esta aportará una base de conocimientos para la disminución de riesgos e incertidumbre en la toma de decisiones durante la elección de una (o varias) tecnología(s) para el desarrollo de HERCCULES. En este contexto, el análisis se trata de un proceso sistemático de búsqueda que tiene como resultado la obtención de una serie de herramientas, bibliotecas, o frameworks (empleados en RTES en el caso del segmento de vuelo) con mayor énfasis en aplicaciones del sector aeroespacial.

OE2 Analizar y seleccionar patrones arquitectónicos y de diseño apropiados para software RTES.
Los patrones arquitectónicos y de diseño son ampliamente utilizados en aplicaciones informáticas ya que proporcionan soluciones a problemas existentes. Existen abundantes publicaciones científicas sobre el tema orientadas a aplicaciones genéricas. Sin embargo, en su mayoría no son aplicables a los RTES ya que se tratan de un tipo específico de sistema informático con mayores restricciones, por lo tanto, los patrones que se emplean

en este tipo de sistemas difieren y deben ser investigados. El alcance de este objetivo específico contribuye en los procesos de diseño y desarrollo del OBSW de HERCCULES aprovechando la reusabilidad de soluciones existentes y reduciendo el tiempo de desarrollo.

OE3 Estudiar la aplicabilidad de metodologías Desarrollo Basado en Modelos (MBD) y Desarrollo Basado en Componentes (CBD) para el sector espacial.
Las metodologías de MBD y CBD son ampliamente usados en sistemas RTES debido a las ventajas y buenas prácticas que ofrecen en ese contexto. Este objetivo específico pretende investigar la aplicabilidad de estos paradigmas de desarrollo en el sector aeroespacial, concretamente para el desarrollo del software de dichos sistemas.

OE4 Verificar y validar componentes software con orientación a estándares espaciales.
Las actividades de verificación y validación del sistema son las más importantes, después de la especificación de requisitos, en el ciclo de vida software. Por ello, se ha visto por conveniente dedicar un objetivo específico a dichas actividades. La consecución de la verificación y validación implica que el sistema se ha desarrollado acorde a sus especificaciones y que cumple los objetivos del proyecto. Por lo tanto, el alcance de este objetivo es condición necesaria para el funcionamiento correcto de tanto el segmento de vuelo como el de tierra de HERCCULES.

OE5 Integrar los componentes software sobre las tarjetas a nivel de subsistema y de plataforma.
El segmento de vuelo de HERCCULES incluye una serie de tarjetas electrónicas controladas por el software de a bordo (OBSW). El despliegue del software sobre la plataforma real y los subsistemas revela problemas que difícilmente se detectan durante la verificación unitaria de los componentes software (cubierta por el **OE4**). Aquí surgen aspectos como concurrencia, sincronización, temporización, entre otros. La superación de este objetivo es clave para la validez del segmento de vuelo de HERCCULES.

Capítulo 3

Marco Teórico y Conceptual

Este capítulo tiene como objetivo definir los términos básicos y conceptos teóricos que son clave para la descripción, comprensión e interpretación del desarrollo y los resultados de esta tesis.

3.1. Sistemas Empotrados

Un Sistema Empotrado (SE) es un sistema informático que está físicamente incluido en otros sistema, tiene un propósito específico y dispone de múltiples interfaces para medir, control y manipular el entorno [15]. La medición se realiza a través de sensores y las funciones de control mediante actuadores, véase figura 3.1. Los SEs están presentes en diversos ámbitos y contextos como la industria médica, sistemas de telecomunicación, robótica y domótica, sistemas de transporte, electrónica de consumo, sistemas ciberfísicos, entre otros. A diferencia de los sistemas de propósito general, los SEs están sujetos a limitaciones hardware ya que se encuentran desplegados en procesadores con pocas prestaciones: baja frecuencia, consumo de energía restringido y escasa memoria [16]. Por lo tanto, la filosofía para el desarrollo del software de estos sistemas tiene requisitos especiales y deben ser diseñados para conseguir su objetivo de la manera más simple y directa posible [17].

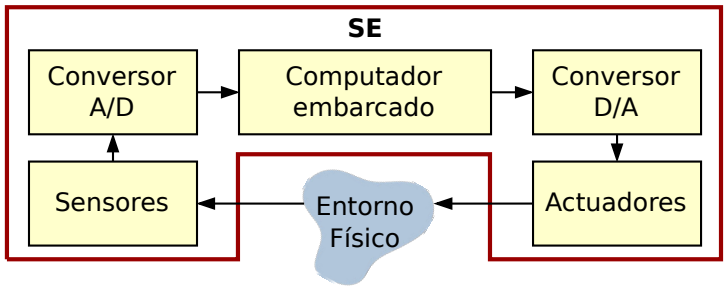
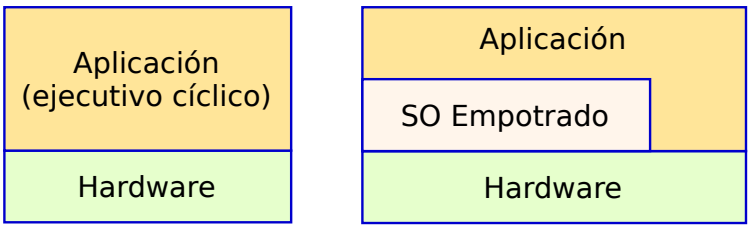


Figura 3.1: Esquema genérico de un sistema empotrado.

Los SEs están presentes en sistemas de alta integridad como satélites artificiales, dispositivos médicos, automóviles o aviones donde los requisitos de seguridad y fiabilidad son más estrictos

que los de otros sistemas informáticos. En estos casos, las condiciones de fallo deben ser tratadas y detectadas a tiempo. Además de las limitaciones físicas y de seguridad, el software de los SEs debe interactuar con los dispositivos Entrada/Salida (E/S). Los sistemas informáticos convencionales también acceden a una serie de periféricos, pero esta tarea es abstraída por el sistema operativo (SO). Esta abstracción no es posible en los SEs debido a la constante proliferación de dispositivos hardware que trae consigue una gran variedad de interfaces y métodos para la E/S.



(a) Máquina desnuda (b) Sistema operativo empotrado

Figura 3.2: Arquitecturas software típicas de un sistema empotrado.

Los SEs más sencillos prescinden de un SO —también denominados máquina desnuda (*bare-metal* o *bare-board* en Inglés)— y siguen una arquitectura basada en "ejecutivo cíclico", compuesta de un bucle de control y funciones auxiliares. Por el contrario, los SEs más complejos cuentan con los denominados "SOs empotrados" que, a diferencia de los convencionales, tienen un diseño compacto y eficiente, ofreciendo solo los servicios imprescindibles. La figura 3.2 muestra un esquema general para estos dos tipos de arquitectura. Se aprecia que incluso contando con un SO empotrado, la aplicación embarcada interactúa con el hardware del sistema. En ciertos sistemas, esta interacción con el entorno está sujeta a restricciones temporales dando lugar a los denominados sistemas de tiempo real, que se discuten en la siguiente sección.

3.2. Sistemas de Tiempo Real

Burns y Wellings [18] definen un sistema de tiempo real como un *tipo específico de SE cuya validez no sólo depende del resultado lógico de la computación, sino también del tiempo en el que se producen los resultados*[1]. Otros autores [16, 19] consideran que los sistemas de tiempo real también entran en la categoría de *sistemas reactivos* ya que deben responder a los estímulos del entorno físico. Desde el punto de vista software, las actividades computacionales, también denominadas tareas, hilos o hebras, son las encargadas de procesar estos estímulos en la forma de eventos, señales, y/o interrupciones, dependiendo del nivel de abstracción. Los estímulos, y por tanto las tareas, siguen diversos patrones de ejecución [16]:

- **Periódicos**. Las tareas periódicas o cíclicas suceden en intervalos de tiempo regulares. La ejecución de estas tareas es provocada por eventos generados por un reloj, perteneciente al SE (*time-triggered*). Por ejemplo, una estación climática debe leer un termómetro cada 50 milisegundos y controlar un calefactor dependiendo del estado del sensor.

[1] A lo largo del documento se emplean indistintamente los términos RTES y sistema de tiempo real.

- **Aperiódicos**. A diferencia de las actividades periódicas, las tareas aperiódicas se ejecutan en intervalos regulares ya que son provocadas por estímulos generados, a su vez, por el entorno físico (*event-triggered*). Dichos eventos pueden ocurrir en cualquier instante de tiempo. Un ejemplo de estímulo aperiódico son las interrupciones provocadas por un controlador Universal Asynchronous Receiver-Transmitter (UART) para el envío/recepción de caracteres.

- **Esporádicos**. Al igual que las tareas aperiódicas, las esporádicas están regidas por el entorno físico (*time-triggered*). La diferencia reside en la ocurrencia del evento, ya que se encuentra acotada por un retardo mínimo.

De estas dos definiciones se infiere el concepto de *plazo* (o *deadline* en Inglés) que define el tiempo en el que debe completarse una tarea. El plazo es relativo al instante en el que dicha actividad se libera para su ejecución. Los requisitos temporales de un RTES dependen de la criticidad de los plazos y se pueden clasificar en [18]:

- **RTES estrictos**. Los plazos de las tareas son estrictos (*hard*), es decir, su incumplimiento es inadmisible, produce fallos en el sistema y trae consecuencias devastadoras como grandes daños medioambientales, perdidas económicas o de vidas humanas. Los RTES con requisitos temporales estrictos, por tanto, deben ser tolerantes a fallos. En cuyo caso incluyen la Detección de Fallos y Recuperación del Aislamiento (FDIR), de manera que el incumplimiento de un plazo produce la activación de una rutina de recuperación o de seguridad ante fallos. Un ejemplo de sistema con plazos estrictos se encuentra en las funciones de control de actitud de un satélite artificial.

- **RTES flexibles**. Los plazos de las tareas son flexibles (*soft*) y su incumplimiento es tolerable siempre que no se produzca frecuentemente. Por ejemplo, el típico sistema de adquisición de datos es un RTES flexible ya que contiene tareas que muestrean sensores a intervalos regulares, pero puede tolerar retrasos eventuales. en este caso, una posible estrategia para la tolerancia a fallos consiste en mantener un recuento del incumplimiento de plazos, de modo que, cuando se supere un determinado umbral se pueda informar al operador.

- **RTES firmes**. Los plazos de las tareas son firmes (*firm*), es decir, la ejecución de una tarea fuera del plazo establecido carece de valor para el sistema. Por ejemplo, los sistemas multimedia como radios, televisores, consolas de videojuego son RTES firmes ya que sus requisitos temporales dependen de la percepción de los usuarios. Por tanto, el incumplimiento de los plazos en estos sistemas solo comprometerá la experiencia del usuario.

Es típico que en un mismo sistema se puedan encontrar subsistemas tanto de tiempo real estricto como no estricto (con plazos flexibles o firmes). A su vez, una misma tarea puede contar con plazos estrictos y no estrictos. Dentro de un sistema las tareas ejecutan en paralelo (o cuasi paralelamente en sistemas mono-núcleo) y raras veces son independientes. Por ello, es necesario realizar operaciones de comunicación y sincronización entre tareas. Se dice que dos tareas se comunican cuando comparten datos o recursos comunes, en este caso es necesario que se realicen en exclusión mutua para evitar condiciones de carrera. En cambio, se dice que dos tareas están sincronizadas cuando respetan un orden de ejecución. Existen diversas formas de abordar estas cuestiones que son el objeto de la teoría básica de concurrencia.

3.3. Paradigmas y Metodologías para el Desarrollo de Software RTES

El software de los RTES requiere una metodología y paradigma de desarrollo que (i) garantice la validez funcional y temporal del sistema, y (ii) que los problemas se puedan detectar los más pronto posible en el ciclo de vida [20]. El problema es que gran parte de las metodologías y paradigmas usados en sistemas informáticos convencionales, no son adecuados para las aplicaciones RTES. Esta sección compendia las más relevantes para este tipo de sistemas, haciendo hincapié en aplicaciones espaciales.

3.3.1. Desarrollo Basado en Componentes

El Desarrollo Basado en Componentes —CBD por sus siglas en Inglés— es un metodología de desarrollo software que surgió a fines de la década de los 90s como alternativa al paradigma de desarrollo orientado a objetos [16]. Su principal motivación es la *reusabilidad* de sistemas existentes para aumentar su rentabilidad y disminuir los tiempos y costes de desarrollo de las futuras aplicaciones software. Un sistema desarrollado con esta metodología está conformado por dos tipos de elementos: *componentes* e *interfaces*.

El estándar UML 2.0 [21] define un componente software como *parte reemplazable de un sistema que se ajusta y proporciona una realización de un conjunto de interfaces. Está pensado para que sea fácilmente sustituible por otros componentes que cumplan con la misma especificación*. Por otro lado, una interfaz se define como *la descripción del comportamiento de los componentes sin proporcionar su implementación o estado*. En la práctica esta definición depende del contexto y se manifiestan de diversas maneras [22]:

- Componentes en la forma de envoltorio de código, clases en Programación Orientada a Objetos (OOP) y módulos en una metodología estructurada (e.g.: paquetes en Ada, espacios de nombre en C++ o unidades de traducción en C).
- Componentes en la forma de capas o subsistemas en la arquitectura software. Por ejemplo, la pila del sistema operativo constituye varios componentes software: gestor de ficheros, manejadores de dispositivos, pila de red, etc.
- Componentes como una unidad desplegable en nodos físicos. Por ejemplo: librerías estáticas (`.so`) y dinámicas (`.a`) en entornos *NIX, ficheros `.jar` en Java, *crates* en Rust, `.dll` en .NET, o `.gem` en Ruby.
- Interfaces desplegadas a nivel de proceso como llamadas a subprogramas, métodos, o funciones.
- Interfaces distribuida mediante protocolos de red como TCP/IP o formatos de alto nivel como REST, colas de mensajes, entro otros.

En todos estos casos se respetan las definiciones de CBD, una interfaz es un conjunto de funciones/métodos implementados o requeridos por un componente. La figura 3.3 muestra la representación gráfica de un componente software y sus dos tipos de interfaces según el estándar UML 2.0. Se aprecian dos tipos de interfaces: requeridas y provistas (también llamadas proporcionadas). Por un lado, las Interfaces Requeridas (RIs) representan los servicios que emplea

el componente. En cambio, las Interfaces Provistas (PIs) se corresponden con los servicios ofrecidos a otros componentes.

Figura 3.3: Representación gráfica en UML 2.0 de un componente y sus interfaces. Adaptado de [16]

CBD en el ámbito de los RTES

Esta metodología ofrece ventajas para el desarrollo de RTES. Por ejemplo, en el caso de los RTES más críticos, el proceso de certificación será más asequible si se reutilizan componentes software que ya han sido certificados. Incluso si solo se desarrollan nuevos componentes, esta metodología trae consigo varios beneficios debido a los principios sobre los que se asienta: (i) bajo acoplamiento, ya que la relación entre componentes está bien definida, por tanto, el nivel de acoplamiento es fácilmente identificable; (ii) consistencia, es decir, siempre que se respeten las interfaces requeridas y provistas, un componente puede ser fácilmente reemplazado; y (iii) encapsulamiento, ya que los detalles de implementación se ocultan en la parte interna del componente.

3.3.2. Desarrollo Basado en Modelos

En la literatura existe una amplia variedad de acrónimos que incluyen la frase "basado en modelos". Brambilla et al. acuñó el término "jungla MD*" para referirse a dichos acrónimos y los resume a modo de prontuario en [23]. En aras de la claridad, a lo largo de la tesis se emplea el término MBD como super-conjunto a todos esos acrónimos, como se ilustra en la figura 3.4. El Desarrollo Basado en Modelos o MBD es una metodología de desarrollo que emplea los modelos como unidad básica para la construcción del sistema [24]. El estándar UML 2.0 [21] define un modelo como *una representación o abstracción que captura los aspectos más importantes de lo que se esté modelando, y que oculta los detalles irrelevantes para el diseño del sistema.*

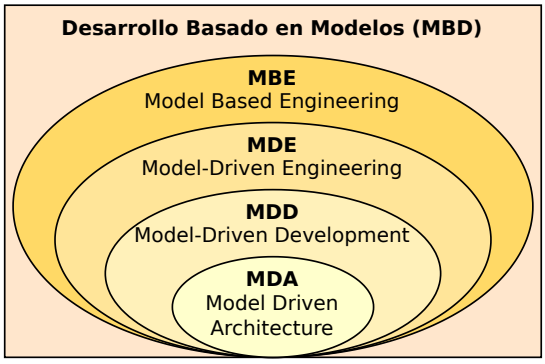


Figura 3.4: Jungla de términos MD. Adaptado de [23]*

Esta metodología es beneficiosa en la Ingeniería de Sistemas ya que ayuda a los desarrolladores a comprender las capacidades y características arquitectónicas (también denominadas "requisitos no funcionales" o "atributos de calidad" en [22]) del sistema, facilitando la detección de problemas en una fase temprana del desarrollo.

MBD en el ámbito de los RTES

Como toda aplicación informática, la etapa más importante para la construcción de un RTES es la generación de un diseño que satisfaga la especificación de sus requisitos. Por el contrario, los sistemas RTES difieren de los tradicionales ya que están limitados por requisitos temporales y no funcionales como seguridad y fiabilidad. Por ello, los modelos empleados para representar sistemas convencionales, no son aplicables para el diseño de un RTES. Burns identificó tres tipos de notaciones para el modelado de un sistema [20]:

- *Informal*. Consta de diagramas imprecisos y sin reglas, cuyo significado depende de la interpretación del lector. Por ejemplo, el significado de las flechas en los diagramas de bloques informales puede representar el flujo de datos, pero también el de control u otro tipo de relaciones como parte-todo o padre-hijo.
- *Estructurado*. Ofrecen representación gráfica con elementos bien definidos. Suelen ser difíciles de analizar. Por ejemplo, los diagramas de contexto o de flujo de datos. Tiene elementos que representan datos, funciones, y conexiones pero carecen de representación textual.
- *Formal*. Son modelos especificados con una notación estructurada pero con base matemática para su representación. Sus elementos están definidos por un *modelo de componentes*. La comprensión de estos modelos requiere de un buen conocimiento de la notación. Los modelos UML, Hard Real-Time Hierarchical Object Oriented Design (HRT-HOOD) o AADL son claros ejemplos de lenguajes con una notación formal, ya que presentan tanto representación gráfica como textual.

Las características de un RTES sugieren que estos deben ser modelado con un lenguaje de modelado formal que soporte las abstracciones de tiempo real (vistas en la sección 3.2): tareas periódicas, esporádicas y recursos compartidos. Asimismo, el uso de lenguajes de modelado formales trae consigo una gran ventaja: la *generación automática de código*. Gracias a esta, el tiempo de desarrollo se ve drásticamente reducido, ya que se evitan tareas repetitivas y el desarrollo se concentra en la parte funcional del sistema. El segundo beneficios de esta metodología para los RTES de alta integridad es que ayuda a predecir las capacidades, comprender las características arquitectónicas e identificar los problemas del sistema en las primeras fases del desarrollo [24].

Modelo de Componentes

Existe una fuerte relación entre las metodologías MBD y CBD. Como se vio en la figura 3.3, los componentes se pueden representar gráficamente, en este caso concreto mediante el lenguaje de modelado UML. El conjunto de reglas que rigen la representación y modelado en la metodología CBD se denomina **modelo de componente**. Esta también incluye una definición formal de los componentes software y los patrones de interacción/comunicación entre ellos. Por ejemplo, el estándar UML define que la relación entre una RI y una PI es de tipo N:1, de modo que una interfaz requerida solo puede estar conectada a una interfaz provista. En caso contrario, se trataría un modelo inválido.

Los lenguajes de modelado idóneos para el desarrollo de un RTES, precisan que su modelo de componentes incluya las abstracciones de concurrencia. En el caso de lenguajes de

modelado genéricos como UML se puede conseguir a través de *perfiles* y *anotaciones*, que permiten restringir y definir extensiones para el lenguaje original, siendo los más destacados HRT-UML [25], UML-RT [26] y MARTE [27].

Como se manifestó previamente, un modelo de componente sigue una notación formal; y esta trae consigo varios beneficios, siendo el más relevante la generación automática de código. En el caso concreto de los sistemas RTES, el generador de código se encargará de los aspectos dependientes de la plataforma, incrementando la portabilidad del sistema. Por ejemplo, en un entorno de ejecución Linux se emplearían las llamadas al sistema para la creación de las tareas y el acceso a los recursos compartidos. En cambio, en un entorno Ada, se emplearán las propias abstracciones del lenguaje: tareas y objetos protegidos.

3.3.3. Desarrollo Basado en Patrones

El desarrollo basado en patrones no dista mucho de las dos metodologías discutidas hasta ahora, en vez de enfocarse en la reutilización de código, se centra en la reutilización de conocimiento abstracto (ideas, estructuras, arquitecturas, diseños, etc.). Los patrones están presenten en una amplia variedad de profesiones y se encuentran a distintos niveles: en la forma de componentes pequeños o estructuras más grandes [28]. En esta tesis solo se tratan los patrones a nivel arquitectónico y de diseño.

El término *patrón de diseño* fue acuñado por Christophe Alexander en arquitectura de edificios y popularizado en la Ingeniería de Software por "La Banda de los Cuatro" en su libro "Patrones de diseño: Elementos de Software Orientado a Objetos Reutilizable" a fines de la década de los noventa [29]. En dicho libro se definió a los patrones de diseño como soluciones reutilizables a problemas de diseño recurrentes que han sido aplicadas con éxito en otros sistemas. Paralelamente, surgieron los *patrones arquitectónicos* bajo el nombre de "estilos arquitectónicos" y, a diferencia de los patrones de diseño, están orientados a los aspectos de alto nivel del sistema. Los patrones arquitectónicos son un conjunto de reglas que definen las características y comportamiento del sistema que lo aplique. Por eso, en sistemas complejos y heterogéneos, es común combinar varios estilos arquitectónicos.

Actualmente, hay una avalancha de patrones dependientes e independientes al dominio de aplicación. De modo que los patrones de diseño/arquitectónicos aplicables al software de contabilidad, banca o inteligencia artificial no serán aplicables a los RTES. Por ejemplo, todos los patrones de diseño definidos en [29] por Gamma et al., están enfocados en el paradigma OOP y se sustentan en la base del *polimorfismo* o *despacho dinámico*. Sin embargo, los RTES están restringidos a instancias estáticas y prescinden de elementos con tiempo de ejecución imprevisibles, por ende, el uso del polimorfismo se desaconseja en estos sistemas.

3.3.4. Análisis y Diseño Estructurado

El diseño estructurado es un paradigma clásico para el diseño de software que surgió a finales de la década de los setenta y llegó a su punto álgido gracias a las contribuciones de Myers, Constantine, DeMarco, Yourdon, entre otros pioneros en ese campo [30]. Están enfocadas en una metodología de arriba hacia abajo, definiendo en primera instancia los límites e interfaces del sistema con el mundo exterior (diagrama de contexto). Tras ello, se desciende en la jerarquía del sistema para definir los Diagrama de Flujo de Datos (DFD) constituidos por bloques

funcionales (procesos) encargados de transformar los datos de entrada en salidas conectadas a otros procesos. Posteriormente, se refinan los procesos en varios subprocesos y almacenes de datos compartidos; el proceso continua, ajustándose a la granularidad deseada, hasta obtener los módulos de software del sistema.

En los sistemas actuales, es considerada como un paradigma de desarrollo tradicional y anticuado. Aún así, sus aportes siguen vigentes, pues los sistemas modernos suelen emplear diagramas de contexto en las primeras etapas de análisis, previas al desarrollo [31, 28]. En el caso de los RTES, el diseño estructurado es ampliamente usado, debido a que se trata de un paradigma *orientado a flujo de datos* [32, 33, 34, 35]. Y como se vio en la sección 3.2, el comportamiento de un RTES está dirigido por los estímulos del entorno (datos de entrada de los sistemas externos), que deben ser analizados por tareas concurrentes (procesos), para producir una actuación sobre el entorno (salidas hacia los sistemas externos). Por ejemplo, la figura 3.5 ilustra el DFD de primer nivel para el microsatélite artificial UPMSat-2 cuyo OBSW entra en la categoría de RTES crítico.

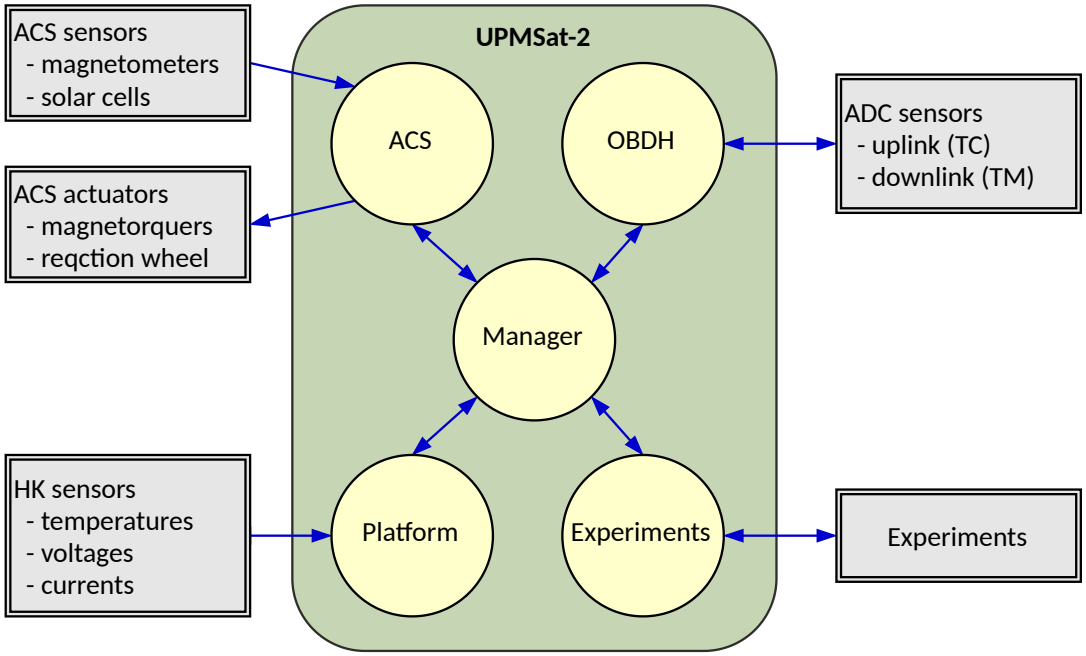


Figura 3.5: DFD de nivel 1 para el OBSW del UPMSat-2. Adaptado de [36, Fig. 2]

3.3.5. Análisis y Diseño Orientado a Objetos

Este paradigma surgió a finales de los ochenta como alternativa al paradigma estructurado, y al contrario de este, el análisis y diseño orientado a objetos se basa en una forma totalmente distinta de pensar el software. En lugar de enfocarse en datos y funciones, organiza el sistema en objetos que interactúan entre sí a través de mensajes. Dichos objectos encapsulan los datos y las funciones, ocultándolo a los demás [28]. Este paradigma aúna tanto el análisis como el diseño. Por un lado, el análisis orientado a objetos comprende una serie de modelos de requisitos (también denominado modelos de análisis) donde se representan los conceptos del dominio, sin entrar en detalles de implementación. Por otro lado, el diseño orientado a objetos refina los modelos de análisis para obtener un modelo del sistema que satisfaga los requisitos.

El uso extensivo de la OOP dio origen a un alud de métodos individuales para el análisis y desarrollo orientado a objetos, entre los que destacan los métodos de James Rumbaugh, Grady Booch e Ivar Jacobson [31]. Estos tres trabajaron en conjunto para unificar sus metodologías, dando lugar a la creación del lenguaje de modelado UML y, posteriormente, al proceso unificado. Estos dos se han convertido en el estándar de facto para todo lo relacionado al OOP. Existe una basta literatura sobre sistemas espaciales desarrollados con paradigmas OOP. Por ejemplo, Booch, en su libro dedicado al análisis y diseño orientado a objetos, presenta un caso de estudio enfocado a la arquitectura de un sistema de navegación por satélite [28, cap. 8]. También han surgido diversos *frameworks* orientado a objetos para el desarrollo del OBSW de sistemas espaciales [37]. A modo de ejemplo, en la figura 3.6 se ilustra el diagrama de clases empleado durante el proceso de diseño del segmento de vuelo del satélite artificial ULg.

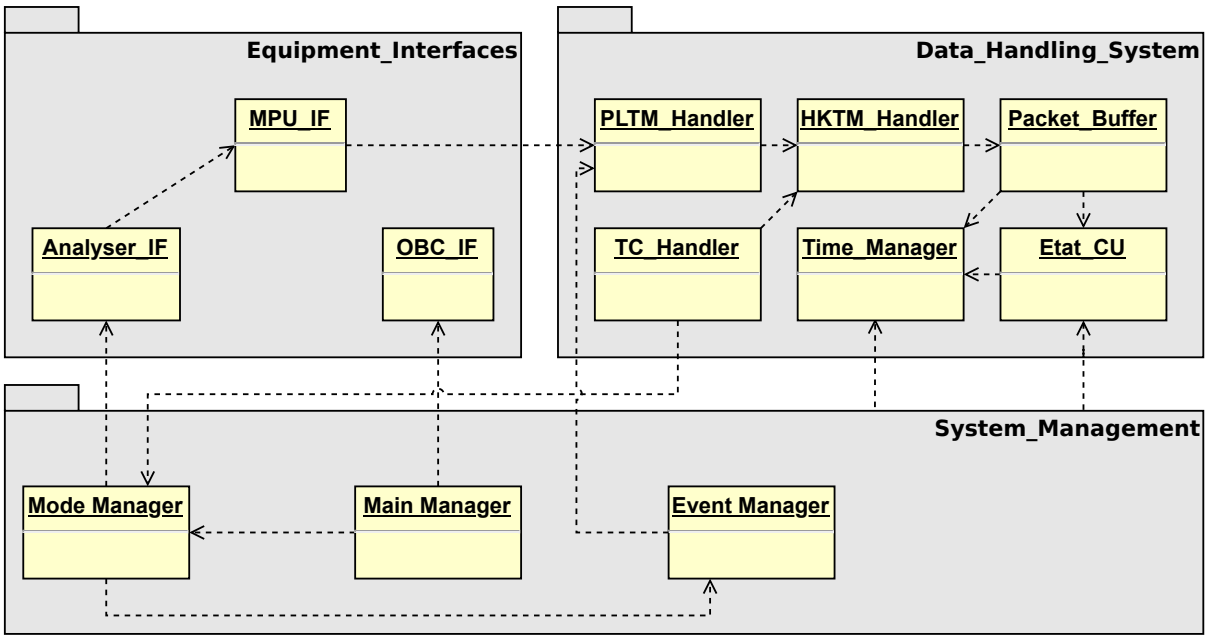


Figura 3.6: Diagrama de clases UML para el OBSW de ULg. Adaptado de [38, pág. 73]

Capítulo 4

Estado del Arte

Este capítulo tiene como objetivo presentar las tecnologías y metodologías más relevantes para el desarrollo software en aplicaciones espaciales. El contenido de este capítulo es el resultado de la investigación obtenido de diversos medios bibliográficos resumidos en el capítulo 5.

4.1. Lenguajes de Modelado para RTES

4.1.1. HRT-HOOD

HRT-HOOD es un método de diseño y análisis estructurado que fue diseñado por Burns y Wellings en 1994 [20]. Fue estandarizado por la ESA y surgió con el objetivo de definir una nueva metodología para el desarrollo de RTES estrictos. Incluye un lenguaje de modelado formal basado en la metodología Hierarchical Object Oriented Design (HOOD) y también incorpora un modelo de ciclo de vida que tiene en consideración las características temporales del sistema. Aunque es una metodología antigua, HRT-HOOD es la precursora y asienta las bases de los futuros métodos de diseño y lenguajes de modelado para los RTES. Las siguientes subsecciones tratan el modelo de componentes y de ciclo de vida introducidas por esta metodología.

Modelo de Componentes

Como su nombre indica, HRT-HOOD sigue un método de desarrollo basado en una serie de objetos estáticos organizados de forma jerárquica (relación padre-hijo), de modo que ofrecen un alto nivel de abstracción y ocultación de la información. Los objetos son los elementos de construcción más básicos del sistema que proporcionan una serie de operaciones y dependen de los servicios ofrecidos por otros objetos. Existen tres tipos de objetos HRT-HOOD que reflejan las abstracciones comunes en sistemas de tiempo real [39]:

- **Activos (A)**. Cada uno tiene asociado un hilo de ejecución exclusivo. De modo que las operaciones implementadas por este objeto, no solo se ejecutan en su contexto, sino que también en el del llamador. En el primer caso, las operaciones pueden bloquearse por restricciones de invocación o funcionales. Los objetos activos, a su vez, se clasifican en dos tipos según su patrón de activación: pueden ejecutar siguiendo patrón, o bien *cíclico*, o bien *esporádico*.

- **Cíclicos o periódicos (`C`)**, implementan una sola operación ejecutada a intervalos regulares (el periodo).
- **Esporádicos (`S`)** ejecutan una única operación asíncrona que está ligada a un suceso externo como una interrupción hardware o evento generado por otro objeto.

- **Pasivos (`Pa`)**. A diferencia de los activos, no tienen asociados ningún hilo de ejecución ni control sobre cuándo se ejecutan sus operaciones. Luego, todas sus operaciones se ejecutan sin esperas en el contexto del objeto que las invoque.
- **Protegidos (`Pr`)**. Estos objetos permiten representar datos compartidos por varias tareas (i.e.: objetos activos). De modo que sus operaciones se ejecutan en exclusión mutua.

El modelo de componentes no solo define los tipos de objetos, también incluye reglas de descomposición y uso que facilitan el análisis temporal del sistema. Esta sección se ha limitado a los tipos de objetos, puede encontrar un compendio de las demás reglas en [39]. A modo de ejemplo, en la figura 4.1 se presenta un modelo sencillo en HRT-HOOD. Dicho modelo se corresponde con la arquitectura software estática de un RTES ficticio inspirado en una versión reducida del OBDH de UPMSat-2 [40].

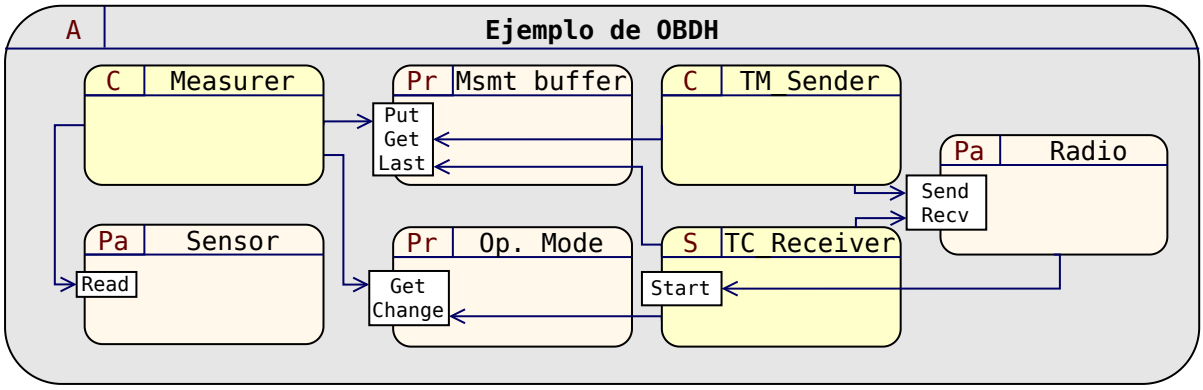


Figura 4.1: Ejemplo de un sistema OBDH modelado en HRT-HOOD. Adaptado de [40].

Se pueden apreciar los tres objetos activos `Measurer`, `TM_Sender` y `TC_Receiver`, representados como bloques de color amarillo. Estos ejecutan paralelamente y se comunican mediante dos objetos protegidos (bloques de color melón): `Msmt Buffer` para los datos de TM y `Op Mode` para el modo operativo del sistema. Asimismo, se puede identificar las operaciones ofrecidas por cada uno (recuadro blanco) y la relación entre ellos, representado por las flechas siguiendo la dirección de la dependencia, i.e.: objeto dependiente → independiente.

En primer lugar, el objeto esporádico `TC_Receiver` —encargado de procesar los TCs— ofrece una sola operación asíncrona (`Start`) que es invocada por el objeto pasivo `Radio`. Este también es responsable de el envío de TM y actualización del modo de operación del sistema, dependiendo del TC recibido. Por otro lado, el objeto cíclico `Measurer` se encarga de medir periódicamente los datos de los sensores, por eso está conectado al objeto protegido `Msmt Buffer` y al objeto pasivo `Sensor`, que abstrae el acceso al hardware mediante la operación `Read`. Finalmente, el objeto cíclico *TM_Sender* se encarga de enviar periódicamente la TM leída del objeto protegido `Msmt Buffer` accediendo a la operación `Get`.

Modelo de Ciclo de Vida

HRT-HOOD no solo es en lenguaje de modelado formal, sino que también incorpora un modelo de ciclo de vida para el desarrollo de RTES basado en un proceso iterativo y centrado en la verificación de los requisitos temporales.

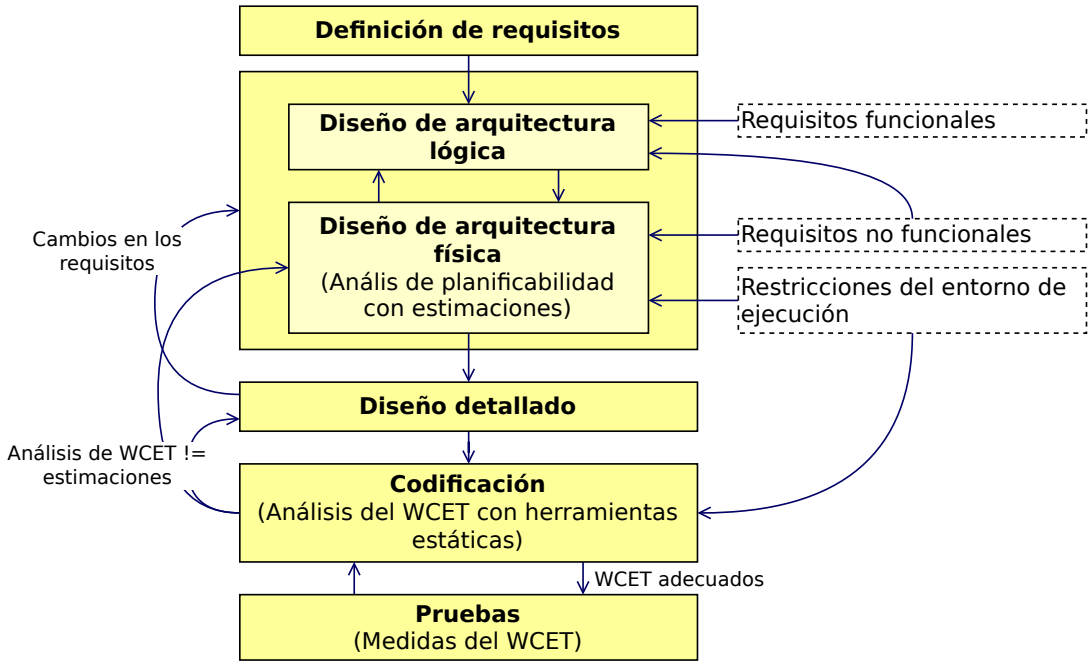


Figura 4.2: Modelo de ciclo de vida de HRT-HOOD. Adaptado de [20].

Este ciclo de vida se presenta en la figura 4.2 y a continuación se resumen las actividades realizadas en cada una de sus fases [20]:

1. **Definición de requisitos**. En esta fase, se especifican los requisitos funcionales y no funcionales. Algunos de los requisitos no funcionales más relevantes en RTES incluyen seguridad física (*safety*) y tecnológica (*security*), confiabilidad, disponibilidad, y mantenibilidad (RMA, por su siglas en Inglés) y requisitos temporales.

2. **Diseño arquitectónico**. Durante el cual se hace una descripción de alto nivel del sistema a desarrollar. Está subdividida en dos actividades:

   a) **Diseño de la arquitectura lógica**, que se centra en los requisitos funcionales y omite las restricciones impuestas por el entorno de ejecución; y

   b) **Diseño de la arquitectura física**. que abarca los requisitos funcionales y no funcionales; incorpora un análisis de planificabilidad preliminar en base a las características del entorno de ejecución y estimaciones, en su defecto.

3. **Diseño detallado**. Durante el cual se especifica el diseño completo del sistema. Si se precisan funcionalidades nuevas o modificaciones debido a cambios en los requisitos, por ejemplo, habrá que modificar tanto el diseño arquitectónico y detallado.

4. **Codificación**. Durante el cual se implementa el sistema. Aquí se deben emplear herramientas de análisis de Tiempo de Ejecución en el Peor Caso (WCET) (e.g.: Bound-T o

Rapi-Time) para comprobar la precisión de los tiempos estimados durante el diseño de la arquitectura física. Si los tiempos distan de los estimados, el diseño de la arquitectura y, en casos extremos, el diseño detallado deben refinarse para asegurar los requisitos temporales.

5. **Pruebas**. Durante el cual se comprueba la validez del sistema. Incluye actividades para medir el tiempo de ejecución del código (empleando, por ejemplo, perfiladores de código).

4.1.2. HRT-UML

Hard Real-Time Unified Modeling Language (HRT-UML) es un método de diseño estructurado derivado de HRT-HOOD que surgió en 2002 como sustituto del mismo. Tuvo como objetivo primario, extrapolar la metodología original a UML ya que este estaba soportado por tecnologías más atractivas y avanzadas de la época. UML ganó buena aceptación en la industria software, sin embargo, su notación y uso no eran adecuado para RTES. Esta adaptación se realizó en base a un perfil que consiste de extensiones sobre el estándar UML 2.0 e incluyó las abstracciones propias de los RTES [41]. La figura 4.3 ilustra los estereotipos del perfil HRT-HOOD, donde se pueden apreciar todos los tipos de objetos definidos en el modelo de componentes de HRT-HOOD (véase el apartado 4.1.1).

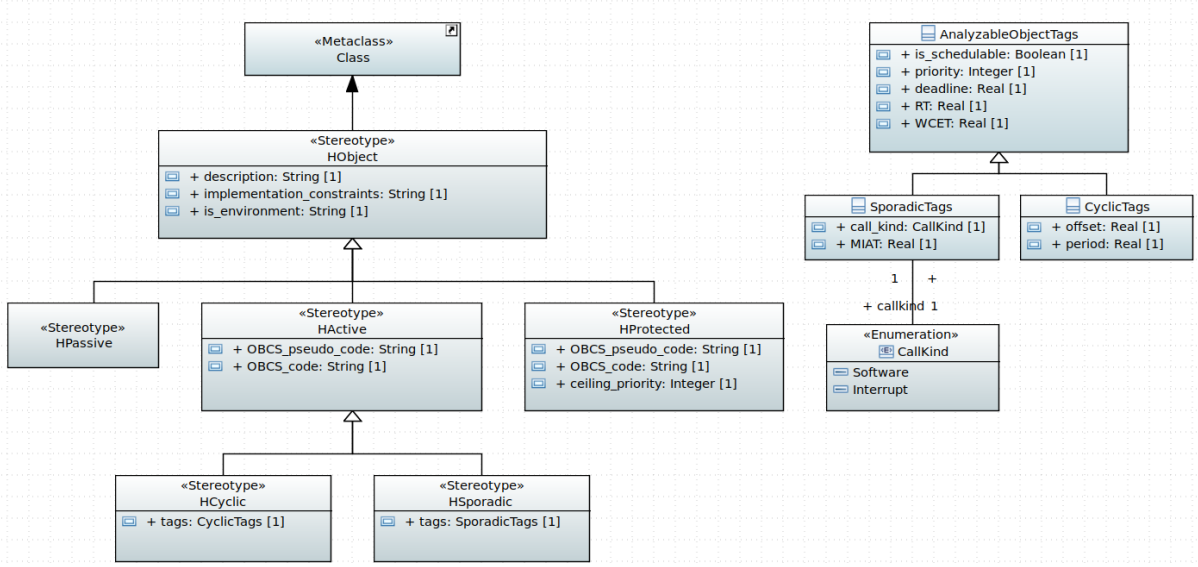


Figura 4.3: Estereotipos del perfil HRT-UML. Adaptado de [41].

4.1.3. AADL

El Lenguaje de Análisis y Diseño de Arquitectura o AADL, por sus siglas en Inglés, es un lenguaje de modelado formal estandarizado por la *Instituto de Ingeniería de Software (SEI)* de la Universidad Carnegie Mellon [24]. El lenguaje soporta representación tanto gráfica como textual y para el desarrollo de RTES siguiendo un enfoque basado en modelos y componentes. Como se ilustra en la figura 4.4, AADL soporta el modelado de diversos aspectos del sistema, entre las que destacan:

- *Arquitectura estática y dinámica del software*: La parte estática del sistema se corresponde con elementos lógicos como bloques funcionales, procesos, tareas (hilos o hebras) y datos compartidos. Asimismo, AADL soporta la especificación del dinamismo o comportamiento de los componentes estáticos a través de modos de operación, flujo de control y de datos.
- *Arquitectura de la plataforma de ejecución*: AADL permite representar los componentes hardware que dan soporte a la ejecución de los componentes software. Por ejemplo, incluye elementos físicos como procesadores, memorias, dispositivos, y buses de comunicación.
- *Entorno físico*: Hace referencia al entorno operacional que interactúa con el hardware y software del sistema RTES. No se modelan explícitamente, pero permite asociar a los componentes hardware del sistema diversas propiedades como frecuencia de funcionamiento, ancho de banda, capacidad de memoria, consumo energético, entre otros.

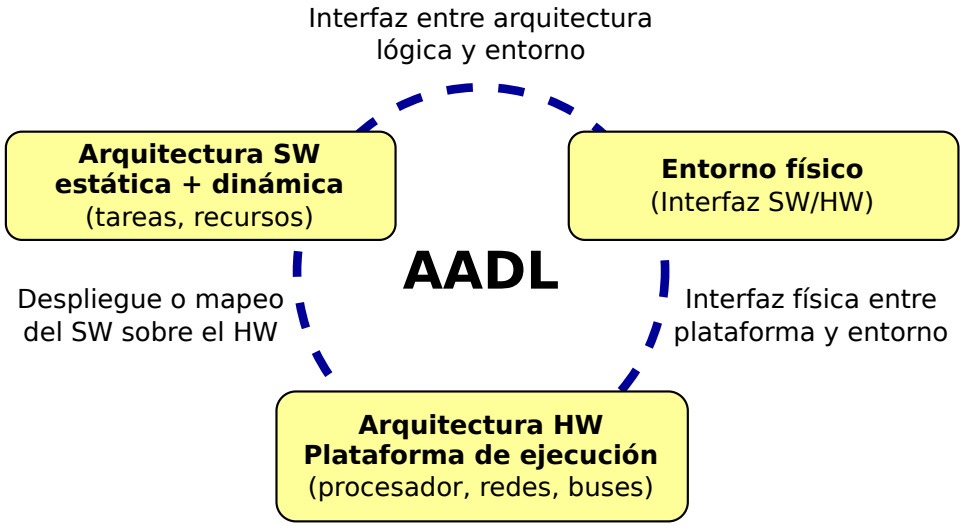


Figura 4.4: Vistas de AADL y sus relaciones. Adaptado de [24].

Modelo de componentes

A modo de resumen, la figura 4.5 muestra la representación gráfica de gran parte de los componentes de modelado de AADL. El prontuario se ha limitado a elementos clave para la comprensión de futuros diagramas presentados a lo largo de este documento. Se puede observar, que AADL permite modelar las abstracciones de tiempo real, como recursos compartidos, tareas (esporádicas y periódicas) y puertos para la intercomunicación de las tareas a través de eventos y datos. También soporta el modelado de la parte secuencial del sistema a través de bloques de sistema y subprogramas, o de la arquitectura hardware mediante dispositivos, procesadores y buses de datos.

Un ejemplo sencillo de modelo AADL se presenta en la figura 4.6. Dicho modelo se corresponde con el mismo sistema OBDH modelado a lo largo de la sección. A diferencia de otros lenguajes de modelado, con AADL se pueden identificar fácilmente el mapa de recursos y procesos del sistema, así como el flujo de control y datos. En este modelo se demuestra que AADL

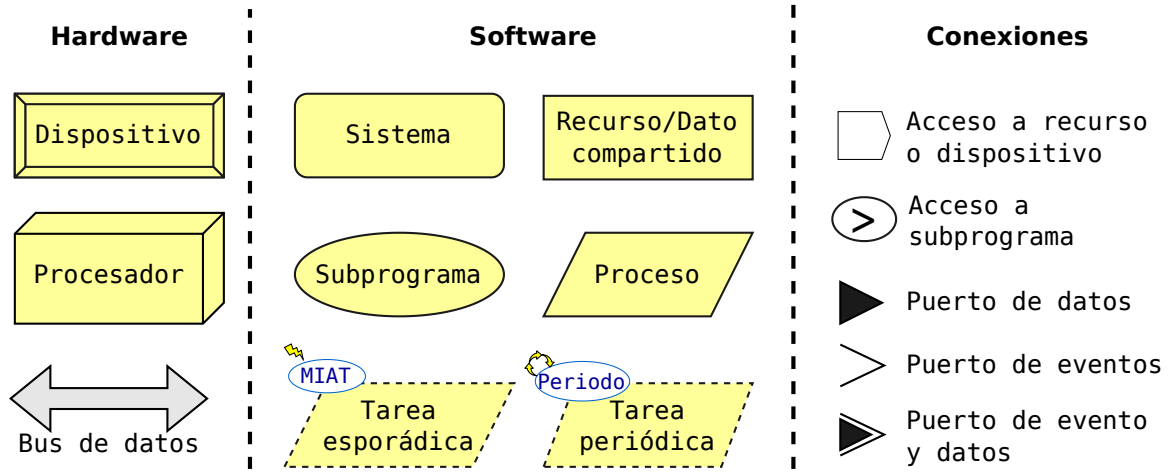


Figura 4.5: Elementos de modelado básicos de AADL.

permite representar la relación entre los componentes software y los dispositivos; por ejemplo, la tarea esporádica `TC_Receiver`, cuyo Tiempo Mínimo Entre Llamadas Consecutivas (MIAT) es de dos segundos, recibe los TCs a través del dispositivo *Radio*. Por otro lado, la tarea cíclica `Measurer` se encarga de medir periódicamente los datos de los sensores, por eso está conectada al dispositivo `Sensor`. En AADL, por lo contrario, es más complicado representar las parte secuencial del sistema.

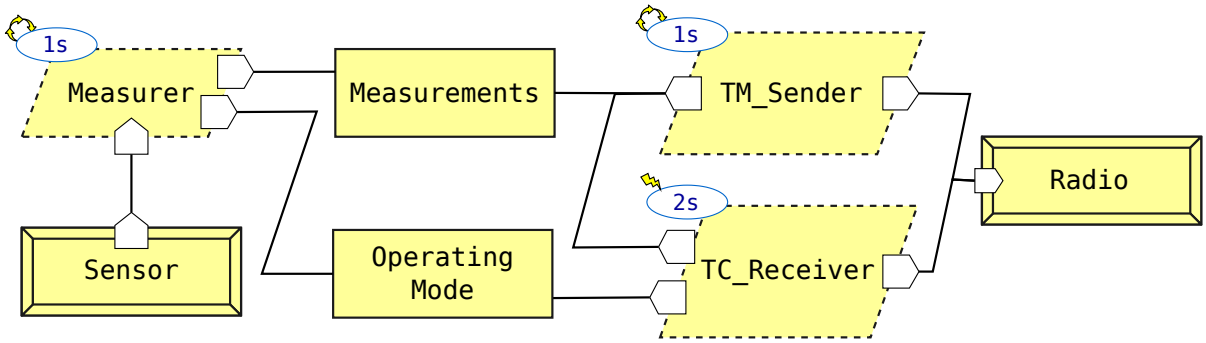


Figura 4.6: Ejemplo de un sistema OBDH modelado en AADL. Adaptado de [40]

4.2. Desarrollo del Software de Sistemas Espaciales

4.2.1. El proceso ASSERT

Automated proof based System and Software Engineering for Real-Time applications (ASSERT) fue un proyecto financiado por la Comisión Europea y coordinado por la ESA que tuvo como objeto de estudio la exploración de diversas tecnologías y metodologías (como el MBD y CBD) para mejorar el proceso de desarrollo software de RTES en el sector aeroespacial [42].

Uno de sus resultados fue la definición del proceso de desarrollo ASSERT [44], que se definió sobre la base de la Arquitectura Dirigida por Modelos (MDA) —el proceso de MBD propuesto por la OMG (Object Management Group) [45]— y está compuesto de una serie de modelos

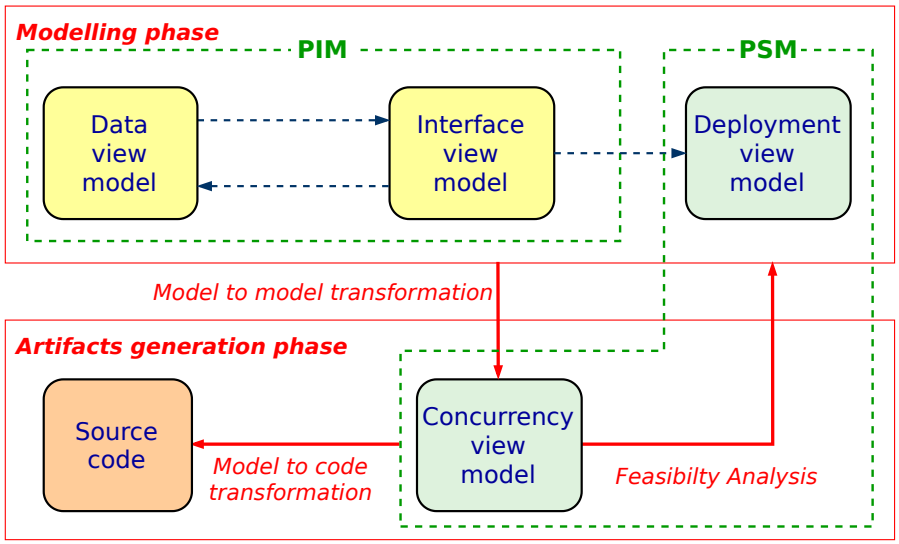


Figura 4.7: Proceso de MBD de la iniciativa ASSERT [43, Fig. 1].

utilizados para describir el sistema en distintas perspectivas, haciendo hincapié en los aspectos que son dependientes e independientes de la plataforma de ejecución (figura 4.7).

El proceso ASSERT divide los modelos en función de su nivel de dependencia de la plataforma. Estos modelos se definen progresivamente hasta la generación automática de los ejecutables del sistema:

- **Modelo Independiente de Cómputo (CIM)**: Los modelos CIM se desarrollan en las primeras fases del proyecto, concretamente, durante la obtención de requisitos. Estos modelos representan las abstracciones de dominio presentes en el sistema, también se denominan modelos de negocio. Se utilizan diagramas de contexto simples, diagramas de actividad y de casos de uso para describir los límites del sistema, los procesos funcionales, y el flujo de control y de los datos. Los modelos CIM no se representan en la figura 4.7 porque su traducción a modelos PIM no está automatizada. Algunos estudios, como [46], han propuesto una metodología para la traducción automática de CIM a PIM, pero es una linea de investigación que sigue abierta.
- **Modelo Independiente de la Plataforma (PIM)**: Estos modelos representan el comportamiento funcional del sistema sin ninguna referencia a su plataforma de ejecución. Así pues, los modelos PIM no tienen en cuenta ni el sistema de soporte de ejecución [1] (e.g.: Open Ravenscar real-time Kernel (ORK) [47], RTEMS, VxWorks, Linux y sistemas en máquina desnuda) ni el hardware del computador de destino, ni las particiones del sistema, ni las interfaces de comunicación del hardware. ASSERT promueve el bajo acoplamiento entre los tipos de datos y la lógica, por ello, se definen por separado en:
  - *Vista de Datos (DV)*: Estos modelos definen los tipos de datos utilizados en el sistema. Los tipos de datos no dependen de la plataforma y las propiedades de des/-codificación se pueden especificar de forma genérica. Este modelo se puede definir formalmente en lenguajes como XML, JSON, ASN.1 [48], Protocol Buffers o AADL [24].

[1] *Runtime system* en Inglés.

- *Vista de Interfaz (IV)*: Este modelo representa los elementos funcionales del sistema y la comunicación entre ellos. El modelo está orientado a componentes, luego, sus relaciones se llevan a cabo mediante el par PI-RI. Esta vista puede modelarse formalmente en lenguajes como AADL, UML, SysML, F Prime Prime (FPP), o HRT-HOOD [20].

- **Modelo Específico de la Plataforma (PSM)** Estos modelos son el producto de una transformación de los modelos PIM. y se obtienen automáticamente de traductores modelo-a-modelo. Los modelos PSM comprenden las características de la plataforma de ejecución y especifican los aspectos no funcionales del sistema, como la concurrencia y el comportamiento en tiempo real. Se producen los siguientes modelos:

  - *Vista de Despliegue (DPV)*: Este modelo define la plataforma física de ejecución, es decir, los dispositivos sobre la que se despliegan los elementos funcionales de la IV. Sus elementos incluyen procesadores, memoria, buses hardware para interconectar procesadores y el soporte de ejecución. Por ejemplo, en el UML, la IV se puede modelar con un Diagrama de Componentes, y los componentes definidos en él se mapearían después en los nodos de un Diagrama de Despliegue, correspondiente a la Vista de Despliegue.

  - *Vista de Concurrencia (CV)*: Este es un modelo complementario que contiene los atributos de tiempo real y concurrencia del sistema. En esta vista, el desarrollador de software especifica el tiempo de liberación de las tareas, sus prioridades, su tamaño de pila, el protocolo de despacho, y otros aspectos que dependen del soporte de ejecución. Esta vista puede especiricarse en lenguajes de modelado como AADL o UML. En el segundo caso serán necesarios perfiles específicos para RTES como MARTE o HRT-UML.

Como se puede apreciar en la figura 4.7, al final del proceso, los modelos PSM se transforman en código fuente para cada plataforma, sin intervención manual. La principal ventaja es que este código se genera de tal forma que los requisitos temporales de los sistemas son especificados por el usuario de forma oculta. Además, los modelos PIM pueden adaptarse fácilmente a distintas plataformas sin que ello afecte a las características funcionales del sistema.

4.2.2. TASTE: El conjunto de herramientas ASSERT para la ingeniería

TASTE [49] es un conjunto de herramientas desarrollada y mantenida por la ESA con el apoyo de empresas de la industria aeroespacial e instituciones académicas como el grupo STRAST de la UPM. TASTE está compuesto de software de código abierto que, en conjunto, forman una cadena de herramientas (de ahí el nombre) para el desarrollo de sistemas RTES distribuidos y heterogéneos. Por un lado la heterogeneidad es una característica subyacente de todo RTES ya que están conformados por equipos multidisciplinares. Por ejemplo, el diseño de un satélite artificial no solo precisa de ingenieros aeronáuticos, sino también de ingenieros software, de control, electrónicos, entre otros. A tal fin, TASTE permite implementar los componentes software en diversos lenguajes de programación. Por otro lado, TASTE abstrae la comunicación remota entre componentes distribuidos mediante ASN.1, que es adoptado como lenguaje universal para la comunicación entre componentes software, independiente de la plataforma en la que se desplieguen y del lenguaje en el que se implementen.

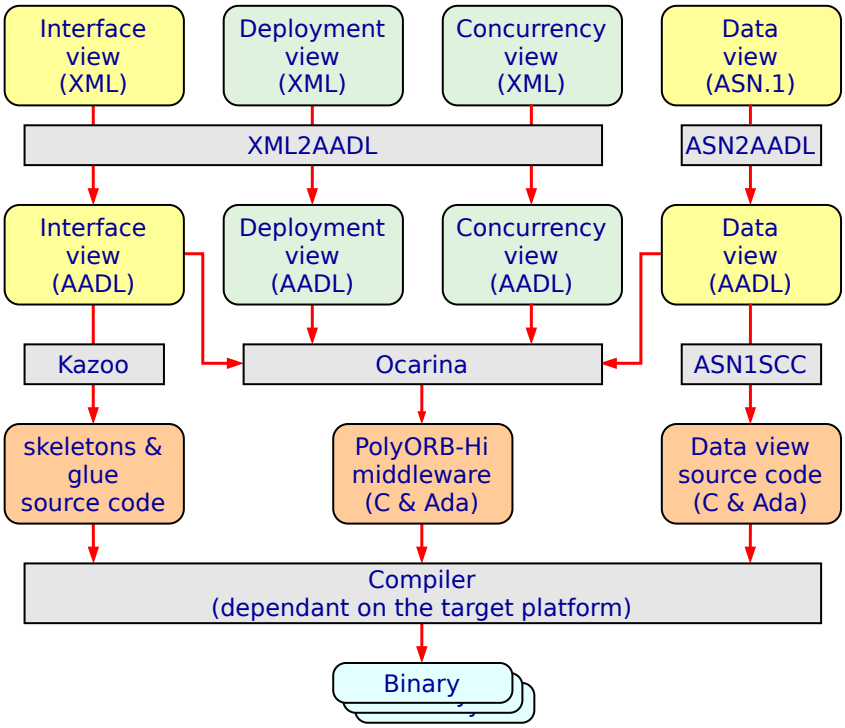


Figura 4.8: Proceso de desarrollo de TASTE para la generación de los binarios.

TASTE es el resultado más relevante del proyecto ASSERT (c.f. sección 4.2.1) y como tal, da soporte a su plan de desarrollo siguiendo los paradigmas MBD y CBD. La adopción de dichas metodologías fomenta buenas prácticas como la reutilización del software, encapsulamiento, ocultación de información y comunicación mediante interfaces bien definidas [16]. Esta sección ofrece una visión general del proceso y las tecnologías empleadas por TASTE para la generación automática de código. La figura 4.8 muestra las herramientas y tecnologías utilizadas por TASTE para aplicar el plan del proceso ASSERT. Se respeta el código de colores de la figura 4.7, de manera que los bloques amarillos representan modelos PIM, los verdes indican modelos PSM y los naranjas son código fuente producido por los generadores automáticos de código.

En el nivel más alto se encuentran las cuatro vistas definidas en el proceso ASSERT en formato XML, a excepción de la DV que se representa en ASN.1. Estas, pueden ser editadas por el desarrollador de software mediante la interfaz gráfica *Space Creator*, que está desarrollada como una extensión del Entorno de Desarrollo Integrado (IDE) *Qt Creator*. Tras ello, las vistas representadas en XML se transforman a un formato común, AADL. Seguidamente, TASTE invoca la herramienta *ASN1SCC* para transformar la DV de los modelos AADL en código fuente C y Ada. Los aspectos funcionales del sistema se modelan independientemente en la IV y sus modelos AADL se transforman automáticamente con la herramienta *Kazoo* [50] en "esqueletos de código" —la generación depende del lenguaje de implementación— para ser completados manualmente por el usuario con la lógica funcional. Posteriormente, se utiliza la herramienta *Ocarina* [51] para construir automáticamente el código fuente, tomando como entradas la DPV y la CV de los modelos AADL. Ocarina utiliza el *middleware PolyORB-Hi* [52] para abstraer funciones y los servicios como la gestión de archivos, las tareas o las operaciones de concurrencia de diferentes entornos de ejecución como Linux, FreeRTOS, RTEMS o Ada. Finalmente,

el código fuente se compila utilizando las herramientas de construcción *GPRbuild* y *Makefile*.

Modelo de Componentes

El modelo de componentes de TASTE está basado en AADL y HRT-HOOD. Por ello, su modelo de componentes guarda similitudes con los elementos de modelado de AADL. Los elementos básicos especificados en la IV se denominan **Función TASTE (FNT)** y desempeñan el papel de los componentes software de la metodología CBD (sección 3.3.1) o de los objetos en el método de diseño HRT-HOOD. Por lo tanto, estas funciones están compuestas de (i) al menos una interfaz proporcionada (denominada PI) mediante la cual se ofrecen los servicios que implementan, (ii) cero o más interfaces requeridas (denominadas RIs) que le permite acceder a los servicios proporcionados por componentes externos, (iii) un conjunto *parámetros de contexto* modificables en tiempo de compilación pero inalterables durante la ejecución, y (iv) al menos una implementación que respete el conjunto de PIs, para ello, TASTE ofrece diversos lenguajes de programación y modelado como C++, C, Ada, Micro Python, VHDL, QGen, Simulink o el Lenguaje de Especificación y Descripción (SDL).

Al más bajo nivel, las interfaces de TASTE, requeridas o provistas, representan funciones/procedimientos convencionales. Por lo tanto, tienen un nombre que las identifica y un conjunto de parámetros formales incluyendo su tipo de dato en ASN.1 y dirección (i.e.: parámetro de entrada o salida). Las PIs son el punto de acceso de las funciones, y, de forma similar a HRT-HOOD, tienen asociados el WCET y soporta los siguientes patrones de ejecución:

- **Interfaz cíclica**: Se tratan de operaciones que ejecutan periódicamente en un hilo de ejecución dedicado. La activación periódica es provocada por un reloj, de modo que no pueden ser invocadas por otras FNTs ni tienen asociados argumentos. El periodo de activación se especifica en el modelo de la IV.

- **Interfaz esporádica**: Se tratan de operaciones activadas esporádicamente, por tanto, tienen asociado un MIAT que se define durante el modelado de la IV. Se trata de una operación asíncrona que se ejecuta en un hilo de ejecución dedicado y es activada por otras funciones. El comportamiento asíncrono implica que la FNT llamadora continúe con su ejecución sin esperas, por lo tanto, no se admiten parámetros de salida.

- **Interfaz protegida**: Se corresponden con operaciones que ejecutan en el contexto de la FNT que la invoca. Sin embargo, se ejecutan en exclusión mutua con las interfaces cíclicas y esporádicas pertenecientes a la función que las implementa. Esta interfaz admite todo tipo de parámetros.

- **Interfaz no protegida**: Se trata de una función convencional que se ejecuta en cuanto se invoca. Al igual que las interfaces protegidas, no tienen restricciones en cuanto a cantidad y tipo de parámetros.

La figura 4.9 muestra en la parte derecha la representación gráfica de una FNT con cuatro PIs, una de cada tipo. A la derecha se puede observar la misma función vista como una caja blanca, de manera que se puedan identificar los elementos internos que la componen empleando la notación gráfica de AADL.

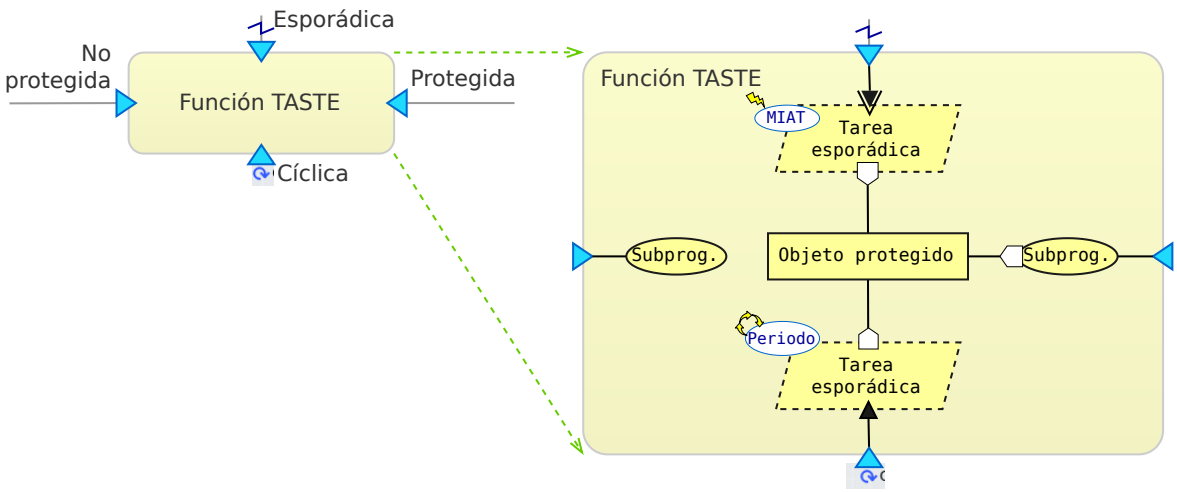


Figura 4.9: Función TASTE vista como una caja negra (izquierda) y blanca (derecha).

4.2.3. Core Flight System

Core Flight System (cFS) es una marco desarrollo de código abierto creado por la Administración Nacional de Aeronáutica y el Espacio (NASA) que tiene como objetivo brindar una herramienta normalizada para la implementación de software aeroespacial. Esta herramienta nace de la necesidad de reducir los costes, los riesgos y el tiempo de desarrollo para el OBSW de las misiones de la NASA, aunque también ha sido adoptada por otras agencias espaciales y empresas comerciales del sector. Para cumplir con su objetivo, la herramienta sigue un modelo de desarrollo enfocándose en la reusabilidad, es decir, se pone a disposición de los desarrolladores un catálogo de componentes software (aplicaciones en el contexto de cFS) que sirven como bloques básicos para la construcción del sistema. La arquitectura subyacente de la herramienta sigue una estructura de capas jerárquicas flexibles (figura 4.10), de modo que la parte dependiente de la plataforma y la misión (como el entorno de ejecución, acceso al hardware, o la parte funcional) son independientes de la parte reusable de cFS [53].

La arquitectura de una aplicación desarrollada en cFS se muestra en la figura 4.10. Todas estas capas trabajan en conjunto para proveer una plataforma genérica [54]. Se puede observar que la capa *RTOS/BOOT* se encuentra al más bajo nivel y encapsula el entorno de ejecución del sistema, que es dependiente de la misión. Incluye el software necesario para el arranque del sistema, y el sistema operativo de tiempo real como RTEMS, VxWorks, Linux, Linux-RT (Xenomai), entre otros. Sobre esta capa se encuentra la *abstracción de bibliotecas* cuyo objetivo es proporcionar una interfaz común a para el acceso a sus servicios como gestión de memoria, gestión de tareas y operaciones concurrentes. A un nivel superior se encuentra la capa *cFE (core Flight Executive)* que provee una serie de servicios independientes y, por tanto, aplicables en diversas misiones. Estos servicios incluyen: gestión de eventos, operaciones sobre almacenes de datos, comunicación mediante mensajes asíncronos, y servicios para la gestión de las en tiempo de ejecución. Finalmente, se encuentran las aplicaciones del sistema en el nivel más alto. Estas usan los servicios ofrecidos por la capa cFE y está compuesta de aplicaciones específicas a la misión, y aplicaciones pertenecientes a la librería de cFE. Estas brindan servicios típicos de las misiones espaciales como: envío y recepción de TM y TC, mantenimiento y gestión de los datos del sistema, filtrado de datos, y operaciones de FDIR.

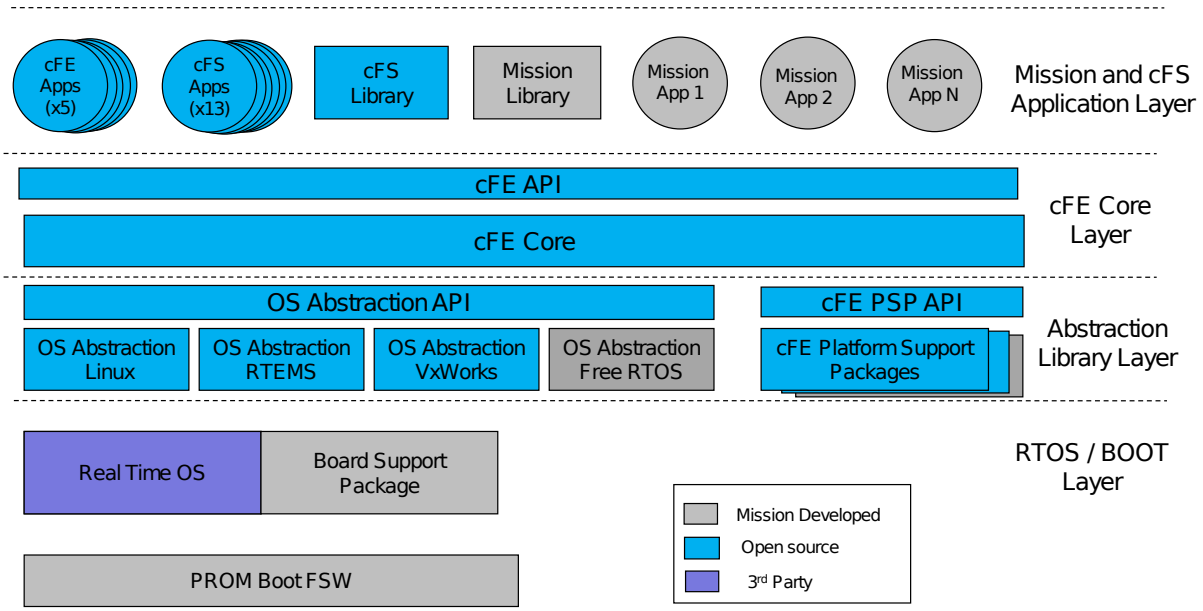


Figura 4.10: Arquitectura en capas de las aplicaciones desarrolladas con cFS [53].

La topología de las aplicaciones cFS se lleva a cabo mediante siguiendo un enfoque Pub-Sub (publicador-subscriptor) a través de un denominado bus software que posibilita el intercambio de eventos y mensajes asíncronos. cFS, también permite configurar el sistema en tiempo de ejecución. Por ejemplo, se pueden hacer cambios en la topología subscribiendo una aplicación a nuevos eventos, también es posible realizar operaciones para la creación y terminación de aplicaciones. Estas propiedades ofrecen buena flexibilidad en el sistema, pero el dinamismo hace que los tiempos de ejecución sean imprevisibles, dificulta el análisis temporal y planificabilidad del sistema. Por ello, cFS se considera una opción idónea para el desarrollo de software espacial, con requisitos temporales laxos y requisitos funcionales complejos.

4.2.4. F Prime

F Prime [55] es otro marco de desarrollo de código abierto creado y mantenido por la NASA. Su objetivo es ofrecer un conjunto de herramientas y librerías básicas para el software de las misiones espaciales facilitando el cumplimiento de sus requisitos de seguridad, fiabilidad y rendimiento. Presenta un modelo de componentes similar al de TASTE y también combina los paradigmas CBD y MBD para el diseño del sistema. Prueba de ello, es que los elementos de construcción más básicos son los componentes software y se caracterizan por encapsular un conjunto de datos (el estado del componente) y operaciones que las modifiquen. La figura 4.11 ilustra un esquema general de un componente F Prime, así como sus puertos de entrada (las interfaces provistas) y salida (las interfaces requieridas) [56].

Los componentes F Prime pueden ser de tres tipos: (i) *activos* donde cada instancia tiene un hilo de ejecución asociado y ofrece tanto operaciones síncronas como asíncronas, por tanto es necesario contar con una cola de espera para las invocaciones asíncronas; (ii) *pasivos* donde cada instancia del componente se comporta como un objeto típico en C++, por tanto, ejecutan en el hilo del llamador y sus puertos de entrada no tienen asociadas ninguna cola de espera;

y (iii) *encolados* que es una combinación entre componente activo y pasivo, pues, cuenta con una cola de espera para las invocaciones asíncronas, pero no tienen asociadas ningún hilo de ejecución dedicado, i.e.: ejecutan en el contexto del llamador.

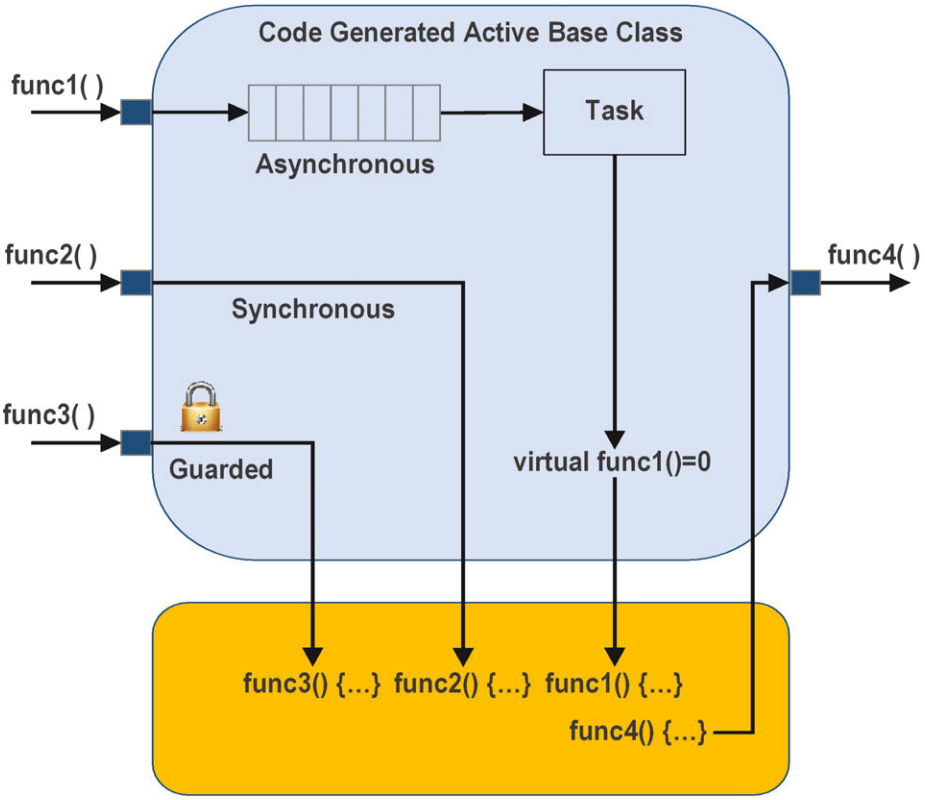


Figura 4.11: Componente software de F Prime [56].

Finalmente, a diferencia de cFS, F Prime presenta una topología estática que es inmutable en tiempo de ejecución. La comunicación entre sus componentes software siguen los patrones de ejecución propio de los RTES y no están restringidos a una comunicación asíncrona basada en Pub-Sub. No obstante, en caso de precisar de una arquitectura Pub-Sub, esta deberá ser construida de forma estática con los diversos tipos de conexiones disponibles, es decir, se deben definir durante el tiempo de compilación todas las conexiones. Estas características, hacen de F Prime una herramienta idónea para el desarrollo de RTES de menor escala ya que posibilita un comportamiento y ejecución previsibles del sistema.

Capítulo 5

Medios y Materiales

5.1. Medios de Tipo Hardware

5.1.1. Arquitectura Hardware del OBDH

Los sistemas OBDH, también llamados sistemas de Comando y Gestión de Datos (C&DH), son el componente central de los satélites ya que son responsables de la comunicación entre las unidades funcionales del vehículo y el segmento de tierra. En general, el OBDH se encarga de las siguientes funciones:

- Ejecución y re-dirección de TCs recibidos desde la GS.
- Generación de TM para su envío a la GS, esta incluye tanto datos científicos como de HK.
- Monitorización del estado del sistema y actividades para la FDIR.

El sistema OBDH de HERCCULES sigue la arquitectura Infraestructura Combinada de Gestión de Datos y Energía (CDPI) definida en [57]. Esta está compuesta de un OBC central encargado del control, supervisión, y adquisición de datos de los equipos del sistema a través de una serie de RTUs, también denominadas Unidad de Interfaz Remotas (RIUs). Las RTUs son tarjetas de E/S externas que están conectadas a los equipos de la plataforma y a los instrumentos de la carga útil mediante líneas analógicas y digitales, incluyendo una variedad de protocolos como Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI) y UART. En el caso de las conexiones analógicas, las RTU no sólo sirven como mediadores o interfaces para los dispositivos, sino que también contienen convertidores analógico-digitales (ADC), circuitos de multiplexación y de acondicionamiento para las señales según los requisitos de rendimiento de la misión. La CDPI incluye una RTU dedicada a la alimentación eléctrica, se trata de un sistema único e independiente que distribuye la energía a todas las demás unidades, incluido el OBC.

La figura 5.1 ilustra la topología general de esta arquitectura, incluyendo sus elementos y conexiones. En la parte superior se aprecian los sensores y actuadores de la plataforma y carga útil. Estos están conectados directamente a las RTUs que se encuentran en el segundo nivel de

la jerarquía. Finalmente, el OBC central se comunica con las demás RTU para el control y gestión de los datos. . Esta arquitectura ha sido adoptada con éxito en misiones espaciales previas como UPMSat-2 [58] o en las dos generaciones de Flying-Laptop [59]. En dichas misiones, tanto la carga útil como el control de la plataforma son gestionados por un único OBC. En el caso de HERCCULES, se emplea una Raspberry Pi (RPi) modelo 4B como OBC central, tres tarjetas electrónicas como RTUs que interactúan con los instrumentos, y una tarjeta dedicada a la distribución y el control de la potencia. Estos elementos y las interfaces de comunicación se discuten en profundidad en las siguientes sub-secciones.

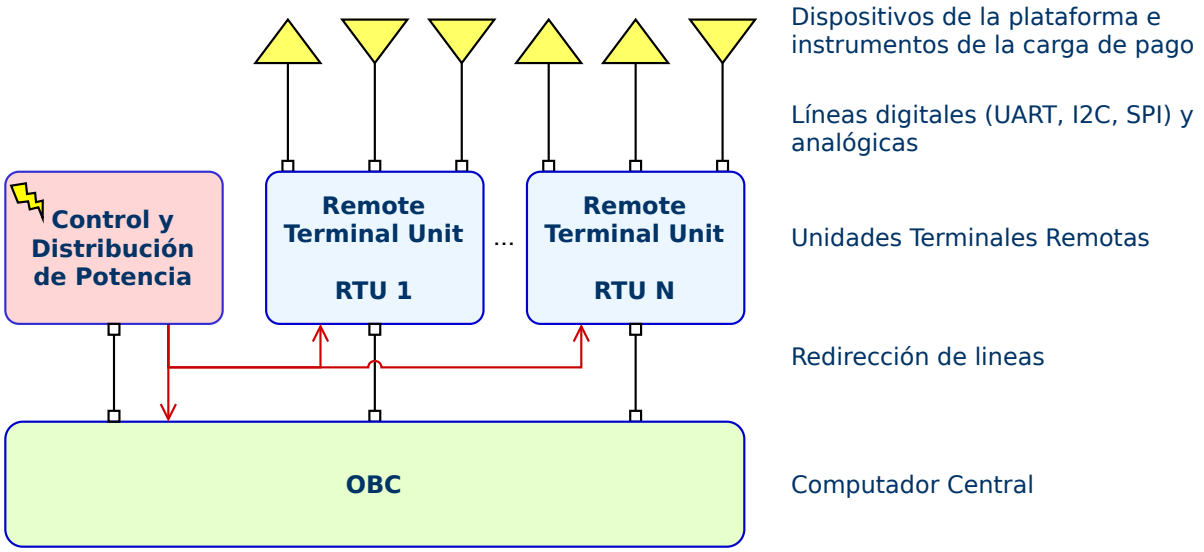


Figura 5.1: Topología general de la arquitectura CDPI.

5.1.2. Unidades Terminales Remotas (RTUs)

El sistema HERCCULES se ha subdividido funcionalmente en cinco subsistemas (HTL, EL, PCU, ATL y NADS) y como se introdujo previamente en la sección 5.1.1, el segmento de vuelo sigue la denominada arquitectura CDPI [57], conformada por un computador central (i.e.: el OBC) conectado a una serie de RTUs que sirven de interfaz con los equipos periféricos de la carga útil y plataforma. La presente subsección describe las tres RTUs empleadas en HERCCULES. Estas incluyen diversas funcionalidades como el control y distribución de la potencia, conversión de señales analógicas a digitales, acondicionamiento de señales analógicas, multiplexación de la E/S, entre otras. No existe gran diferencia entre subsistema y RTU; los subsistemas ofrecen una vista abstracta a nivel funcional, mientras que las RTUs dan soporte físico a estos subsistemas mediante tarjetas electrónicas, es decir, ofrecen una perspectiva más concreta. En lo que resta del documento, se emplean los términos “RTU” y “tarjeta de E/S” de forma indiscriminada.

HERCCULES consta de una caja electrónica (E-Box) que comprende el OBC, tres RTUs y soportes adicionales para facilitar la separación y el enrutado entre tarjetas electrónicas. La figura 5.2 ilustra la estructura mecánica de dicha E-Box. La parte izquierda de la figura muestra el soporte de las cuatro tarjetas electrónicas: PCU, OBC, SDPU y TMU. Dichas tarjetas siguen una estructura apilada (en ese orden) para mayor eficiencia y disponen de un conector tipo

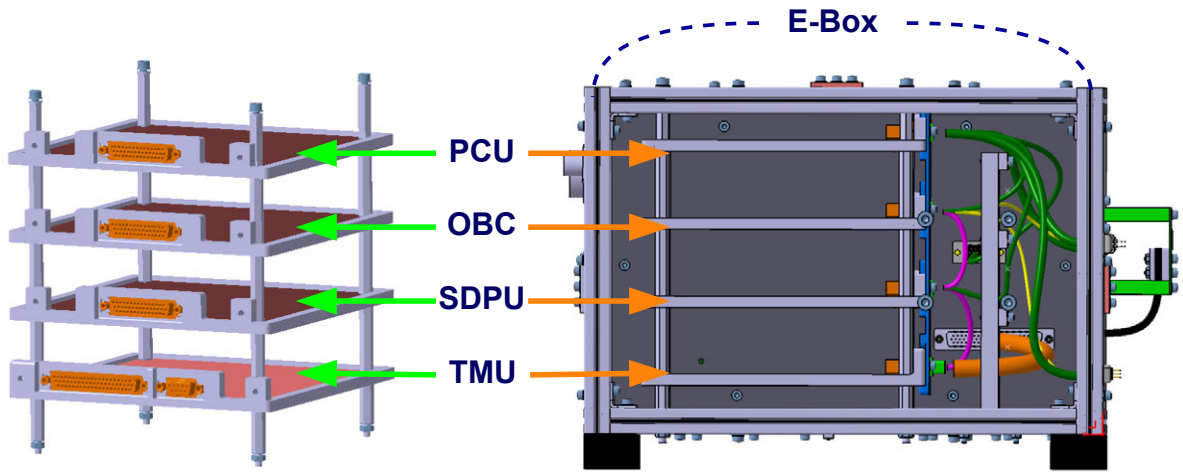


Figura 5.2: Esquema de la caja electrónica de HERCCULES. Adaptado de [14].

D-Sub como interfaz estándar entre tarjetas. Por otro lado, en la parte derecha se observa el ensamblado de las tarjetas electrónicas dentro de la E-Box, la conexión entre tarjetas, y el enrutado de dichas conexiones resultando en un diseño más cómodo y robusto.

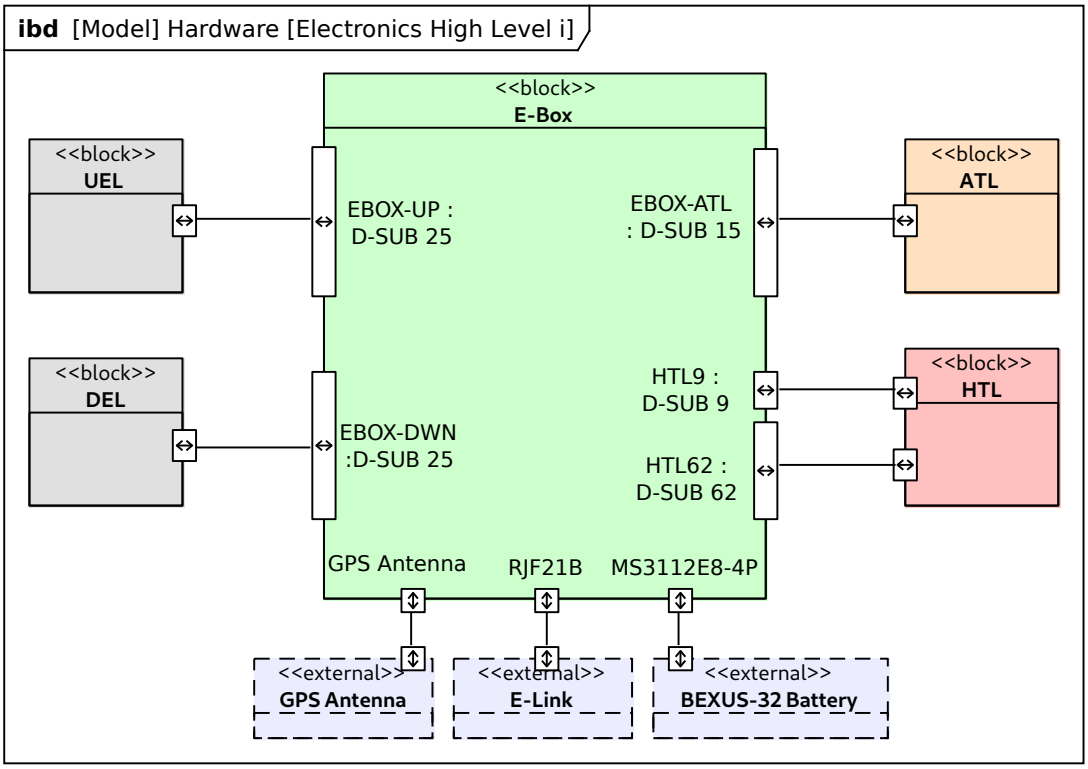


Figura 5.3: Diagrama Interno de Bloques de alto nivel para las tarjetas electrónicas.

Antes de ahondar en las RTUs, se presenta una perspectiva general de los subsistemas. Para ello, se emplearan Diagramas Internos de Bloques (IBDs) del lenguaje de modelado SysML [60], ya que permiten representar los aspectos estáticos de la electrónica de forma clara y concisa. Los puertos representan las interfaces D-Sub, mientras que los conectores modelan diversas líneas electrónicas acorde a la leyenda. La figura 5.3 muestra el modelo IBD a nivel de contexto, es

decir, se delimita el ámbito y las fronteras de la E-Box con el resto de experimentos.

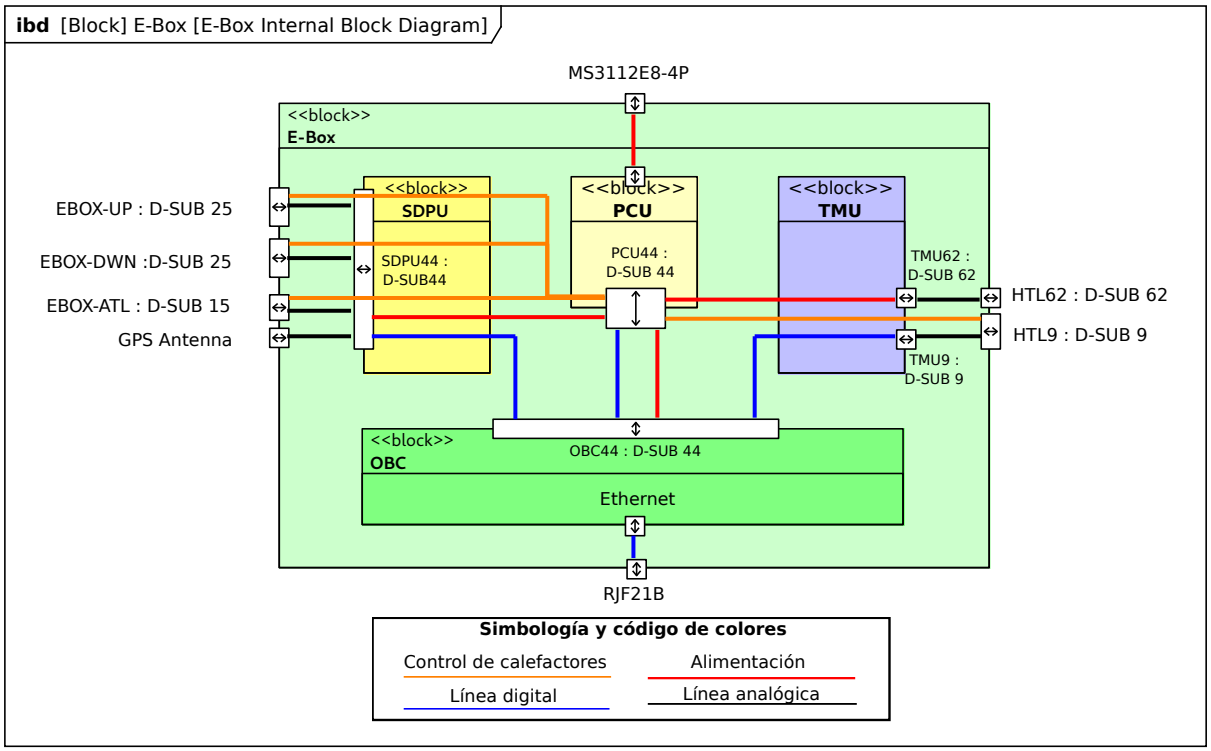


Figura 5.4: Modelo Diagrama Interno de Bloques de la E-Box.

Por otro lado, el modelo IBD presentado en la figura 5.4, desciende un nivel y profundiza en las partes internas de la E-Box. Nótese que por la definición de la arquitectura CDPI, *el OBC no es considerado un RTU*. Por ello, se encuentra aislado del resto de RTUs en una tarjeta dedicada. Este contiene un conector D-Sub de 44 pines el cual proporciona un único punto de comunicación para todos los experimentos y subsistemas. Véase la sección 5.1.3 para información detallada sobre el OBC. Por el contrario, las tarjetas electrónicas PCU, SDPU, y TMU son consideradas RTUs y son el objeto de las siguientes subsecciones.

5.1.2.1. Unidad de Control de Potencia (PCU)

Esta RTU es un mapeo directo del subsistema PCU. Para la monitorización del estado de esta tarjeta electrónica, este subsistema está equipado con un sensor digital de temperatura y otro de potencia/voltaje/corriente. Dichos sensores son accesibles mediante el protocolo I2C. La PCU recibe 28.8v de las baterías de la góndola y la distribuye a todas las demás RTUs a tres niveles diferentes (3.3v, 5v y 12v) mediante tres convertidores CC/CC. La línea de 12v se emplea para alimentar los calefactores de silicona, mientras que las líneas de 3.3 y 5v alimentan el resto de componentes electrónicos. Además de distribuir la energía y alojar los sensores, esta RTU contiene (i) interruptores para activar y desactivar la energía suministrada a otras RTUs y (ii) *drivers* necesarios para controlar los calefactores mediante PWM. Todos estos componentes electrónicos son de tipo COTS y se incluyen en el diagrama IBD presentado en la figura 5.5.

La tabla 5.1 resume las principales características de los sensores y actuadores equipados en la PCU. Se expone la cantidad, el tipo (digital [D] o analógico [A]), la interfaz (I/F) que sigue

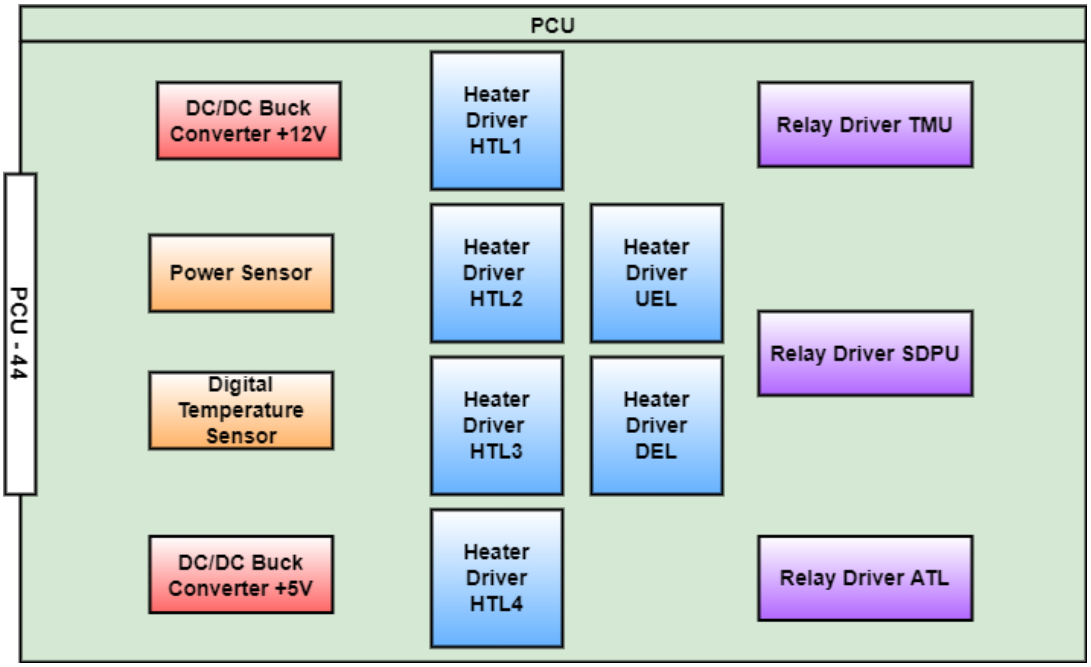


Figura 5.5: Modelo IBD de la PCU [61].

para la comunicación con el OBC, y una breve descripción junto con una referencia a la hoja de datos del dispositivo.

Tabla 5.1: Dispositivos de la Unidad de Control de Potencia.

Dispositivo	***Cant.***	***Tipo***	***I/F***	***Descripción***	***Ref.***
TC74	1	D	I2C	Sensor de temperatura con una resolución de 1 grado Celsius.	[62]
INA266	1	D	I2C	Sensor de potencia, corriente y voltaje.	[63]
IRFZ44PbF	9	A	GPIO	MOSFET para controlar los calefactores de silicona.	[64]
GU SOP 1	3	A	GPIO	Relé de estado sólido para activar y desactivar tarjetas electrónicas.	[65]

5.1.2.2. Unidad de Medición Térmica (TMU)

Esta RTU es responsable del acondicionamiento de la señal y el multiplexado de los 28 termistores PT1000 del subsistema HTL. Para ello, contiene el Conversor Analógico Digital (ADC) ADS1115 de tipo sigma-delta con 4 canales de entrada [66]. Cada canal está conectado a un multiplexor 8:1; de esta forma, se posibilita la lectura de hasta 32 (8×4) líneas analógicas. Se emplea una interfaz I2C como punto de acceso al ADC, y los cuatro multiplexores (uno por cada canal del ADC) están controlados por las mismas líneas de selección, es decir, tres pines GPIO. De este modo, el OBSW puede especificar las líneas analógicas deseadas y solicitar su conversión cuando sea necesario.

La figura 5.6 muestra el diagrama IBD de la TMU donde se pueden apreciar los elementos internos que lo componen y la cadena de procesos (o *pipeline*) que acondiciona las señales

analógicas provenientes de los termistores PT1000 para su posterior conversión en señales digitales.

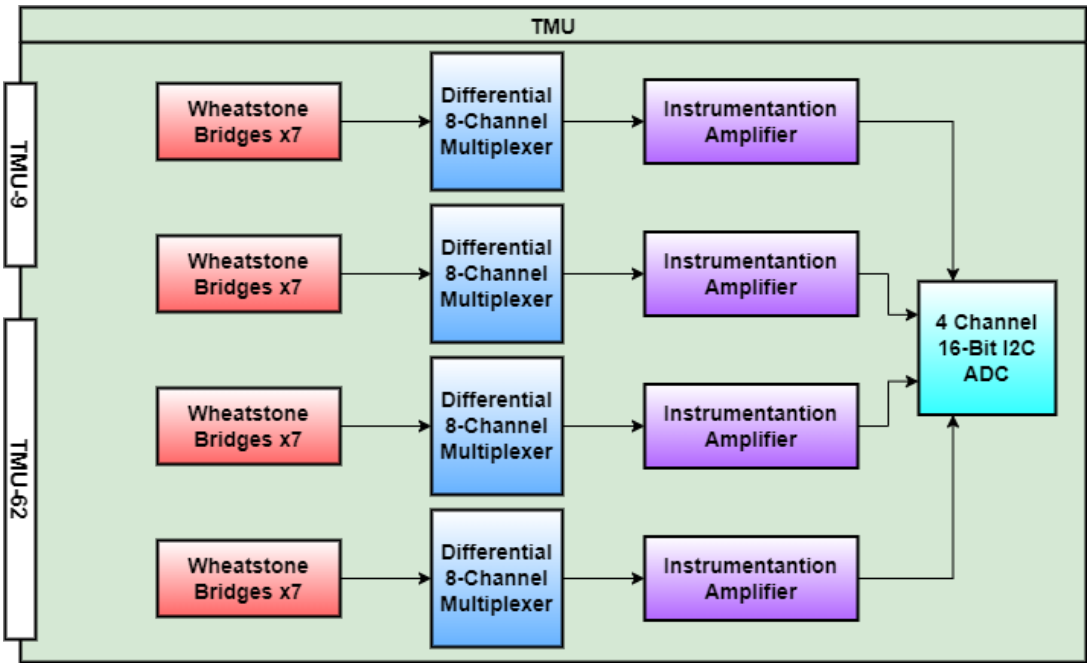


Figura 5.6: Modelo IBD de la TMU [61].

El diseño de estas tarjetas electrónicas se realizó en su totalidad por los estudiantes de grado Juan Manuel Redondo y Pedro Luis Barba bajo la supervisión del Prof. Javier Malo. Puede encontrar información más detallada sobre el diseño electrónico en [14, sección 4.5.6]. Las especificación de este subsistema define la siguiente función de transferencia T. Esta permite obtener las temperaturas de los termistores en $^{\circ}C$ a partir de la lectura en V del ADC, representado por la variable v.

$$T(v) = 34.18 \times v - 80.24$$

5.1.2.3. Unidad de Procesamiento de Datos Sensoriales (SDPU)

Esta RTU interactúa con equipos tanto analógicos como digitales. Concretamente, contiene sensores de los subsistemas EL, ATL y NADS. Las líneas analógicas se procesan de forma similar a la TMU. De hecho, contiene el mismo ADC y los mismos multiplexores que permiten recoger hasta 32 señales analógicas. El ADC y todos los sensores digitales, incluido los barómetros, se conectan al OBC a través del bus I2C-3. Por otro lado, el sensor GPS del NADS está conectado a la OBC mediante UART-0 y la IMU utiliza el bus I2C-1 por cuestiones de rendimiento. En este caso concreto, el software de vuelo no sólo es responsable de leer las mediciones de los sensores, sino también de orquestar el acceso de forma protegida, sin condiciones de carrera entre subsistemas concurrentes.

La figura 5.7 ilustra el IBD de la SDPU donde se aprecia el flujo de datos y el proceso de conversión a los que se someten las señales analógicas. También se aprecia que los barómetros absolutos y el sistema NADS se encuentra aislado del procesado de las sensores analógicos.

La tabla 5.2 describe las principales características de los sensores montados y procesados por

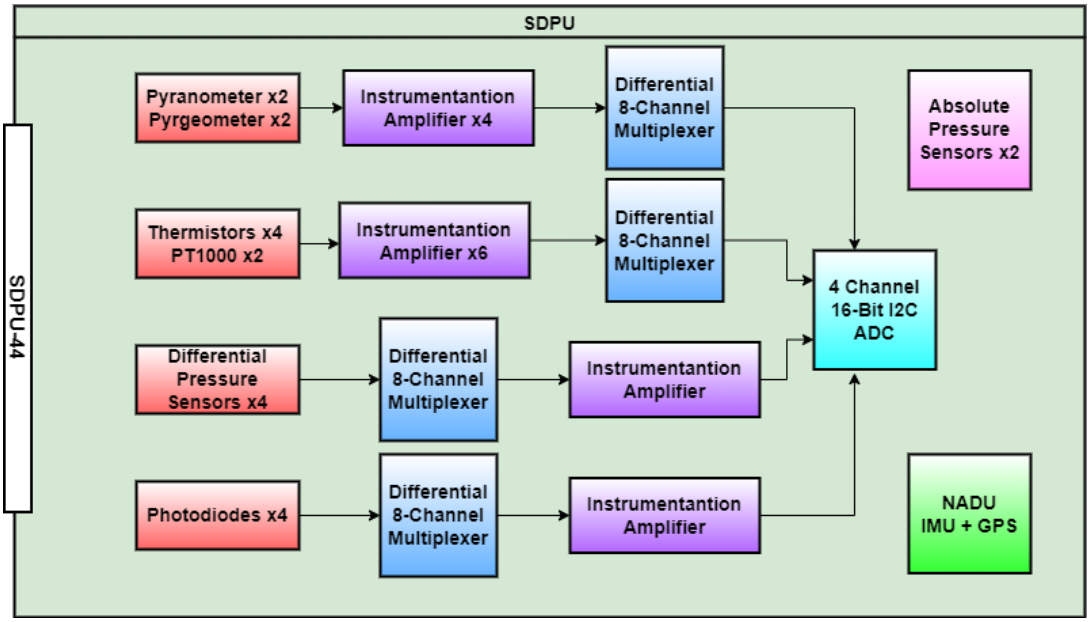


Figura 5.7: Modelo IBD de la SDPU [61].

la SDPU. Al igual que en secciones previas se muestra la cantidad, tipo (analógico [A] o digital [D]), I/F de comunicación, una breve descripción y la referencia a la hoja de datos.

Tabla 5.2: Dispositivos de la Unidad de Procesamiento de Datos Sensoriales.

Dispositivo	***C***	***T***	***I/F***	***Descripción***	***Ref.***
SR20	2	A	ADC	Piranómetro del EL. Mide la radiación solar, cubriendo un rango de longitud de onda (λ) de 0.285 hasta 3µm.	[67]
IR20	2	A	ADC	Pirgeómetro del EL. Mide la radiación térmica IR, cubriendo λ de 4.2 hasta 40µm. Este rango incluye gran parte de la radiación emitida por la Tierra.	[68]
LDES050BF6S	4	A	ADC	Anemómetro del EL. Basado en microflujos térmicos. Mide la presión en rangos de 25 a 500Pa.	[69]
LMS41PD-03	4	A	ADC	Fotodiodos del ATL.	[70]
ADS1115	1	D	I2C	ADC de 16-bit, con cuatro canales y de tipo σ/δ perteneciente a la SDPU.	[66]
MS5611	2	D	I2C	Barómetros del EL.	[71]
MIKROE1032	1	D	UART	Módulo GPS del NADS. Incluye el módulo u-blox 6. Sigue el protocolo National Marine Electronics Association (NMEA).	[72]
BNO055	1	D	I2C	IMU del NADS. Sensor tipo SMEM (microelectromecánico) de 9 ejes. Incluye un magnetómetro, acelerómetro y giroscopio.	[73]

Finalmente, la tabla 5.3 contiene las funciones de transferencia para las señales analógicas.

Todas las ecuaciones emplean la variable v como el valor leído en V por el ADC.

Tabla 5.3: Funciones de tranferencia de la SDPU.

Señal analógica	***Función de transferencia***
PT100	$T(°C) = 34.18 \times v - 80.24$
Termistor del ATL	$T(°C) = 13.85 \times v + 28.42$
Piranómetro	$I(mV) = 4.84 \times v - 4.84$
Pirgeómetro	$I(mV) = 3.66 \times v - 7.5$
Barómetro diferencial	$P(Pa) = 24.41 \times v - 50$

5.1.3. Computador de a Bordo (OBC)

Como se ha mencionado en la sección 5.1.1, el OBDH controla y gestiona todos los demás subsistemas. Por esa razón se precisa de un OBC que realice dichas actividades. En las misiones espaciales, los OBCs están sometidos a condiciones ambientales hostiles con rangos extremos de temperatura, radiación, presión, vibración, aceleración, entre otros. Por ende, dichas misiones requieren procesadores resistentes a la radiación y con tolerancia a fallos. Esto no se aplica en la misión HERCCULES, que volará en la estratosfera a 30km aproximadamente. Por ello, el OBC seleccionado es el computador COTS **Raspberry Pi Modelo 4B**, véase figura 5.8. Este OBC ofrece altas prestaciones y una gran variedad de herramientas para el desarrollo de software. La RPi 4B contiene un sistema en chip (SoC) Broadcom BCM2711 con un microprocesador ARM Cortex-A72 de 64 bits y cuatro núcleos que implementan el conjunto de instrucciones ARMv8 y trabajan a una frecuencia de 1.5GHz.

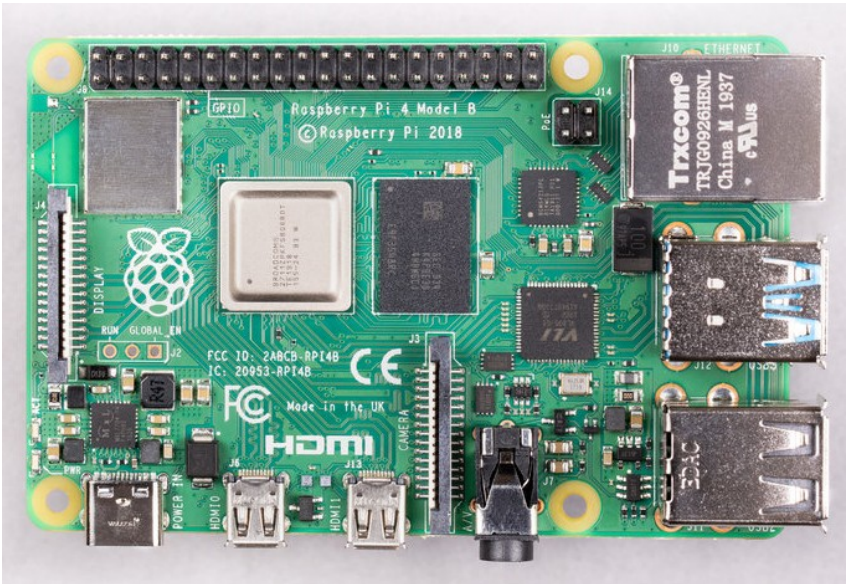


Figura 5.8: Raspberry Pi modelo 4B.

La placa del procesador está compuesta por un computador monoplaca (SBC), de modo que no sólo incluye la CPU (Cortex-A72), sino también los siguientes componentes hardware [74]:

- 8GB de memoria LPDDR4 RAM para la ejecución del OBSW.
- 64GB de unidad flash para el despliegue del OBSW y almacenamiento de la TM.
- 1 puerto RJ45 Gigabit Ethernet para la comunicación con la GS. Este se conecta al sistema E-Link.
- 40 lineas Entrada/Salida de Propósito General (GPIO) con 28 pines digitales a 3.3v para el nivel alto.

Adicionalmente a las funciones de E/S en los pines GPIO, la RPi soporta dos canales PWM con frecuencias de hasta 8kHz. Sin embargo, solo el primero es accesible desde el cabezal de pines de la RPi 4B. A continuación, se resumen los protocolos de comunicación ofrecidos por dicho OBC:

- **I2C**
  - Cantidad: 6.
  - Transmisión: Half-duplex.
  - Ancho de banda: hasta 100 kbps y 400kbps en *fast-mode*.
  - 7 bits para la direccion de los esclavos, configurable a 10 bits.
  - Hasta 127 esclavos con 7 bits ($2^7 - 1$).
  - Sin soporte para el *clock-stretching*.
- **SPI**
  - Cantidad: 5.
  - Transmisión: Full-duplex.
  - Ancho de banda: velocidad igual a la del reloj central.
  - Cada línea soporta distinto número de esclavos, dependiente de los *chip-select*.
- **UART**
  - Cantidad: 6.
  - Disponible a través de los pines GPIO.
  - Transmisión: Full-duplex.
  - Ancho de banda: Sin limitación, en teoría.
  - Opcionalmente se pude usar a través del puerto USB.

5.1.4. Interfaces de Comunicación

La considerable número de dispositivos implica el uso de diferentes protocolos y e interfaces para la transmisión de los datos. La plataforma y carga útil contiene dispositivos periféricos conectados al OBC mediante tarjetas electrónicas denominadas RTUs. En general, la conexión entre el OBC y los dispositivos se clasifica de la siguiente manera:

1. **Conexiones directas**. Esta es la primera variante y se encuentra en sensores digitales que se conectan directamente a los pines GPIO del OBC mediante diferentes tipos interfaces como I2C y UART. El protocolo I2C ofrece un esquema de comunicación regido por direcciones, permitiendo así una topología de conexiones punto-a-multipunto, en la cual varios sensores se conectan al mismo OBC a través de un único bus I2C. Por el contrario,

el protocolo UART es más costoso en términos de ocupación de pines GPIO, ya que sólo permite conexiones punto-a-punto.

2. **Conexiones indirectas**. La segunda variante está compuesta por sensores y actuadores analógicos que precisan de circuitos y equipos *intermediarios* para su conexión con el OBC. Dichos intermediarios incluyen (i) ADCs para obtener valores medibles de los sensores analógicos, y (ii) controladores hardware para encender/apagar las líneas de alimentación (relés de estado sólido) y controlar la potencia disipada por los calefactores a través de señales PWM.

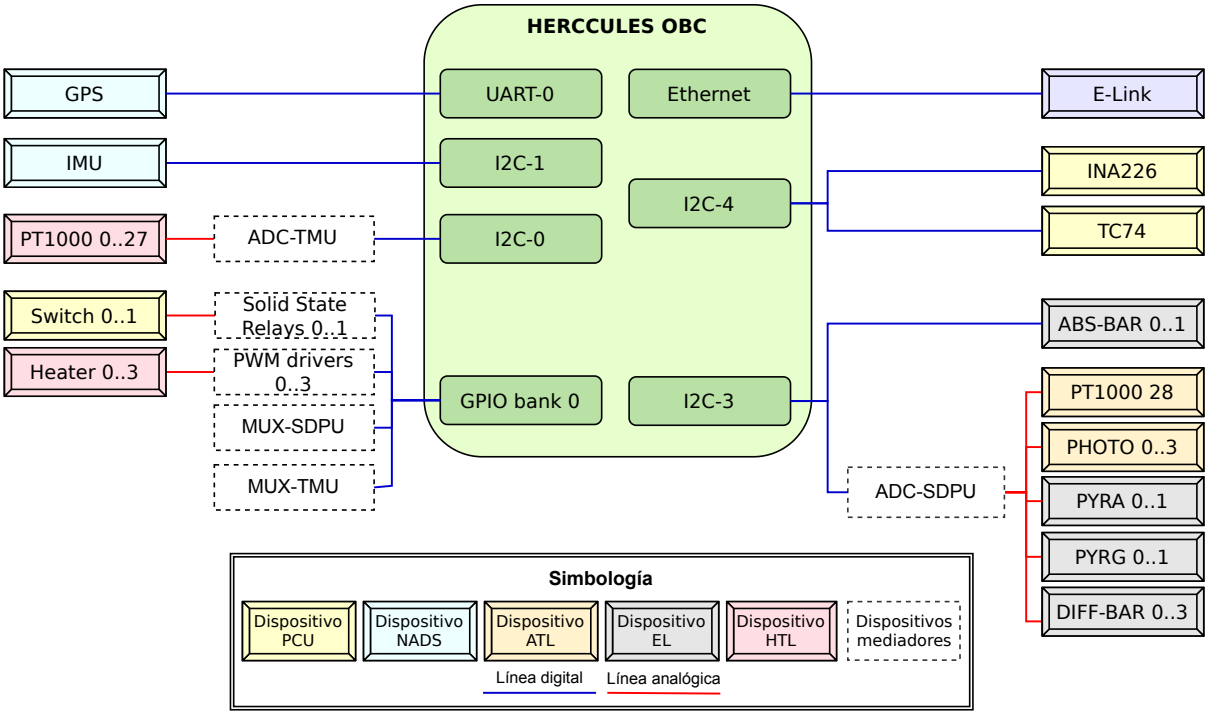


Figura 5.9: Arquitectura e interfaces hardware del OBC.

La figura 5.9 muestra las I/Fs hardware y la relación entre el OBC y los dispositivos externos mediante los tipos de conexiones antes mencionados. En aras de la claridad, el código de colores se mantiene compatible con el diagrama de contexto de la figura 1.6. Todos los sensores y actuadores funcionan a diferentes velocidades de transferencia y requieren distintas frecuencias de muestreo y control. Por ello, la jerarquía de buses se diseñó de forma tal que los sensores analógicos que compartan el mismo bus I2C y ADC tengan periodos de muestreo similares y, preferiblemente, pertenezcan al mismo subsistema. Por ejemplo, todos los sensores del UEL y DEL están conectados a la interfaz I2C-3. Concretamente, los dos barómetros absolutos son sensores digitales conectados al bus I2C-3, pero los sensores analógicos (piranómetros, pirgeómetros y anemómetros) se comunican con dicho bus mediante el ADC de la SDPU. Por otro lado, los equipos del ATL comparten el mismo bus I2C y ADC mencionados anteriormente debido a la escasez de interfaces y a que se encuentran en la misma tarjeta compartiendo circuitería adicional para el acondicionamiento de la señal.

A continuación se resumen la configuración de las interfaces I2C, UART y PWM para la RPi 4B empleada en HERCCULES. En cuanto a interfaces I2C, los buses I2C-0, I2C-3 e I2C-4

usan los controladores hardware del SoC con una frecuencia de 100kHz. Sin embargo, para el bus I2C-1 se emplea la variante *bit-bang* con una frecuencia similar. Esta es una variante programada que emplea el software como sustituto de los controladores hardware reales e implementa el protocolo de transmisión requerido. En cuanto a la línea serie UART-0, se ha configurado en modo 9600/8N1. Esto se traduce a una velocidad de 9600 baudios, 1 bit de inicio, 8 bits de datos, y sin bits de paridad. Con respecto a las líneas PWM, solo se usa un canal del controlador PWM-0. El segundo canal no se emplea ya que está ocupado por otros protocolos de comunicación. La línea PWM-1 no se usa porque solo está disponible en el 2º banco GPIO, inaccesible desde los cabezales de la RPi. Al igual que en los otros casos, las líneas que no emplean el controlador hardware emplean la variante bit-bang. Todas las líneas PWM se han configurado a 1kHz.

Además de estas interfaces hardware, se utilizaron once pines GPIO para controlar la activación de las líneas de alimentación empleadas para encender/apagar las placas de E/S, seleccionar salidas multiplexoras para las líneas de los sensores analógicos, y realizar el encendido/apagado de ciertos calentadores. La tabla 5.4 resume el uso de pines de todas estas interfaces, incluyendo los pines GPIO. Es importante destacar que aunque hay 53 dispositivos conectados a la RPi, sólo se utilizaron 25 pines GPIO de un total de 28. Esta configuración se logró debido a la multiplexación de las líneas analógicas y la comunicación a través de I2C, un bus multipunto.

Tabla 5.4: Uso de interfaces y pines de la RPi.

I/F	***Cantidad***	***Uso de I/F***	***Uso de pines***
GPIO	28	11	11
I2C	6	4	8
UART	6	1	2
PWM	2	4	4

El anexo A.1 recoge información detallada sobre la configuración de la RPi empleada en HERCCULES. El uso de todos los pines GPIO se presenta en la figura A.1 siguiendo la disposición física de los pines del cabezal GPIO. Se puede apreciar que solo existen tres pines libres, los demás están reservados, bien para las líneas de voltaje (*GND*, 5v y 3.3v), o bien para los instrumentos. La figura A.2 ilustra el diagrama de contexto detallado para el OBC donde se puede corroborar el uso de pines antes mencionado. Finalmente, el listado de código B.1 de dicho anexo muestra el fichero de configuración `/boot/config.txt` para establecer los modos y activar las diversas I/Fs.

5.2. Medios de Tipo Software

5.2.1. Herramientas de Desarrollo

El desarrollo del sistema HERCCULES requirió de varias herramientas (véase fig. 5.10), entre las que destacan:

- **Git** se empleó como Sistema de Control de Versiones (CVS) para el software de vuelo y tierra. Es una herramienta ampliamente usada en la industria para el seguimiento

Figura 5.10: Collage de herramientas software.

y control de los cambios sobre la línea base del proyecto. Esta herramienta facilitó la colaboración concurrente entre desarrolladores.

- **GitLab** fue la plataforma elegida como anfitriona de los servidores git. Ofrece funcionalidades gratuitas como Integración Continua (CI), empaquetado, y planificación que se utilizaron a lo largo del proyecto. El enlace al repositorio de HERCCULES se encuentra en [75]:

  `https://gitlab.com/AngelPerezM/herccules`

- **CMake** se usó para gestionar los procesos de compilación y construcción del sistema. Esta herramienta auto-genera una serie de *Makefiles* que en proyectos de esta envergadura serían tediosos de escribir y mantener.

- **Visual Studio Code (VSCode)** es un editor de código simple pero potente gracias a la variedad de extensiones disponibles. La característica más destacada es el soporte de *Language Servers* que potencian la experiencia de edición en muchos lenguajes de programación, incluidos C y C++. Durante el desarrollo también se instalaron extensiones para gestión de proyectos en *CMake* y *linters* para la detección de código sospechoso.

- **CLion** es un IDE para C y C++ de la empresa JetBrains que se empleó como editor de código durante las primeras fases del desarrollo. Aunque se disponía de licencia gratuita para su uso, se migró el proyecto a **VSCode** en aras de fomentar el uso de herramientas de código abierto.

- **Visual Paradigm** se usó como herramienta CASE para el modelado del software y sistema en UML y SysML, respectivamente.

- **OpenGeode** es el editor SDL que emplea TASTE por defecto y constituye una herramienta independiente. Es una herramienta de código abierto que ha sido desarrollada por la ESA y soporta una gran parte de los elementos de modelado de SDL.

- **OSATE** es una herramienta CASE de código abierto que permite la especificación gráfica y textual de sistemas RTES en el lenguaje de modelado AADL. Se eligió ya que es la única opción gratuita que da soporta a dicho lenguaje.
- **MS SharePoint** se usó para compartir documentación relativa al proyecto con todos los participantes y demás miembros autorizados.

5.2.2. Sistema Operativo

Raspberry Pi OS fue elegida para esta misión ya que es el SO oficial para los SBCs Raspberry Pi. Aunque no es un sistema operativo en tiempo real (RTOS), es un sistema operativo embebido basado en Debian, y como tal es compatible con la interfaz *Portable Operating System Interface X (POSIX)*, la cual soporta un modelo de planificación flexible [76]. POSIX incluye unos perfiles y extensiones de tiempo real (POSIX 13, 1b, 1d y 1j) que permiten definir configuraciones restringidas. Entre estas, se incluye la política de prioridades fija `SCHED_FIFO`. También proporciona operaciones para la gestión de acceso a recursos compartidos, incluyendo el protocolo de herencia de prioridad (`PTHREAD_PRIO_INHERIT`) y de techo de prioridad inmediato (`PTHREAD_PRIO_PROTECT`). En cuanto a la gestión del tiempo, POSIX incluye el reloj de tiempo real `CLOCK_MONOTONIC` que ofrece una resolución de 1ns y es útil para establecer los retardos de las tareas periódicas y esporádicas.

Al tratarse de un sistema Unix, Raspberry OS incluye las funciones `open`, `close`, `read` y `write` definidas en `unistd.h`; y la signatura `ioctl` definida en `ioctl.h` para operaciones más flexibles. En conjunto, estas funciones definen una Interfaz de Programación de Aplicaciones (API) estándar para el acceso a los controladores, facilitando el desarrollo de los módulos software de bajo nivel que interactúen con los sensores y actuadores. Todas estás características que ofrece Raspberry Pi OS facilitan el análisis de tiempo de respuesta de las tareas y la gestión de los dispositivos, convirtiéndola una opción viable para este proyecto.

La figura 5.11 muestra una perspectiva general de la arquitectura de un sistema Linux convencional, aplicable a Raspberry Pi OS.

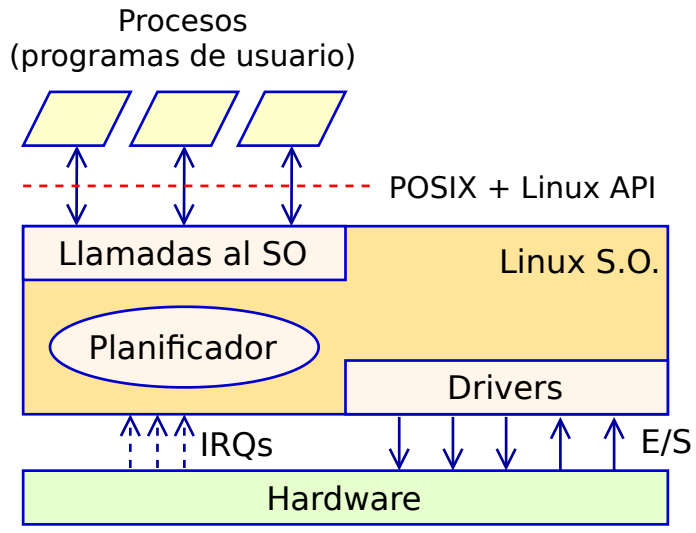


Figura 5.11: Arquitectura genérica de un sistema operativo Linux.

5.2.3. Bibliotecas

A lo largo del proyecto se emplearon diversas bibliotecas para facilitar el acceso a las I/Fs de comunicación y, en menor medida, para los componentes de alto nivel. Las bibliotecas de bajo nivel incluyen tanto manejadores de dispositivos accesibles en el espacio de usuario como las variantes bit-bang que emulan dichos protocolos. Como se ha mencionado en secciones previas, la variante bit-bang fue de especial interés ya que HERCCULES contiene una gran cantidad de dispositivos pero un número limitado de interfaces. A continuación, se listan todas las bibliotecas empleadas en HERCCULES:

- *i2c-dev*. En Linux, los buses I2C son controlados por el módulo del kernel `i2c-dev`. Asimismo, estos buses son accesibles desde el espacio de usuario a través del sistema de ficheros virtual `/dev`. La configuración de los dispositivos es posible mediante la inclución de las bibliotecas del sistema `sys/ioctl.h` y `linux/i2c-dev.h`.

- *Termios*. Similar al caso anterior, los puertos UART se representan por ficheros (generalmente siguiendo el patrón `/dev/tty*`) y se pueden configurar accediendo a las estructuras de datos y operaciones definidas en `termios.h` y archivos de cabecera complementarios.

- *pigpio*. Esta biblioteca [77] de código abierto está escrita en C y ofrece operaciones para configurar y controlar los pines GPIO de la RPi. Adicionalmente, ofrece la variante bit-bang para el control PWM.

- *minmea*. La lectura del sensor GPS no solo requiere el acceso al bus UART, sino también procesar los mensajes NMEA recibidos. Esta biblioteca [78] es de código abierto, está escrita en ISO C99 y su función principal es la descodificación de mensajes NMEA. Es una biblioteca atractiva para HERCCULES ya que se puede compilar en sistemas ARM y no emplea memoria dinámica.

5.2.4. Lenguajes de Programación

El segmento de vuelo se ha programado en tres lenguajes distintos dependiendo de la naturaleza del componente a implementar. Por un lado, los componentes de más bajo nivel que interactúan con los dispositivos periféricos (sensores y actuadores) se han escrito en C 11 y C++ 17. En cambio, los componentes de más alto nivel se han modelado en SDL ya que tienen un comportamiento representable mediante máquinas de estado. Estos modelos SDL son transformados en Ada por la herramienta OpenGeode de la ESA. Por otra parte, los tipos de datos se han encapsulado en un componente independiente a la parte funcional y se han especificados en el lenguaje ASN.1. La integración de todos estos componentes heterogéneos es responsabilidad de la herramienta TASTE, usada para el desarrollo del OBSW. Finalmente, el segmento de tierra se ha programada en el lenguaje C++ 17 y a diferencia del OBSW no está sujeto a restricciones de seguridad y tiene permitido usar memoria dinámica, excepciones, polimorfismo, recursividad, entre otras funcionalidades permitidas por el lenguaje.

5.3. Medios Bibliográficos

Para realizar esta tesis se han consultado artículos de revista, artículos de congresos, libros y trabajos de fin de titulación de diversas organizaciones, con especial interés en la Ingeniería de Software y su aplicación en el sector espacial. Principalmente se usaron los siguientes buscadores: Web of Science, O'Reilly, IEEEXplore, Archivo Digital UPM, Science Direct, ACM Digital Library y Springer Link. La búsqueda se limitó a textos escritos en Español e Inglés. La mayor parte de los artículos y libros en Inglés se encontró filtrando los títulos, palabras claves y resúmenes empleando las siguientes cadenas de búsqueda: **(i)** *("obsw" OR "on board software" OR "fsw" OR "flight software") AND ("satellite" OR "cubesat" OR "obdh" OR "cdh*")* para bibliografía relativa a software de sistemas espaciales; **(ii)** *("model based*" OR "component based*" OR "mb*" OR "cb*")* para bibliografía relativa a metodologías de desarrollo basada en modelos y componentes; **(iii)** *("space" OR "flight" OR "satellite") AND ("framework*" OR "toolchain" OR "toolset" OR "technology")* para bibliografía relativa a tecnologías y herramientas idóneas para el sector espacial; y **(iv)** *("rtes" OR "rts" OR "real time" OR "embedded") AND ("pattern*" OR "architect*" OR "design" OR "structure")* para bibliografía relativa a patrones de diseño y arquitectónicos en el sector RTES.

La tabla 5.5 contiene las referencias bibliográficas que se ha empleado como base del conocimiento y que han repercutido significativamente en el desarrollo del proyecto. En dicha tabla se muestra el tipo de texto respetando el siguiente convenio: artículo de revista (AR), artículo de conferencia (AC), libro de texto (LT), trabajo de fin de grado o máster (TFT), y tesis doctoral (PHD). Seguidamente, se reflejan los aportes de cada referencia con los objetivos específicos trazados en la sección 2.2.

Tabla 5.5: Medios bibliográficos relevantes.

Ref.	Tipo	OE	Descripción
[16]	LT	OE2, OE3, OE4	Sommerville explica conceptos esenciales de la Ing. de Software siguiendo un enfoque práctico. El cap. 21 se centra en sistemas de tiempo real e incluye una serie de patrones arquitectónicos identificados de forma empírica por el autor. También introduce las diversas metodologías de desarrollo y ciclos de vida, identificando su aplicabilidad en distintos contextos.
[2]	LT	OE1, OE2, OE4	Eickhoff da una introducción general a los sistemas informáticos en aplicaciones espaciales. Introduce los conceptos OBC, OBSW y GS. Identifica una arquitectura de referencia para software espacial.
[79]	LT	OE2, OE4, OE5	Eickhoff pasa de la teoría a la práctica en este libro aplicando los conceptos de [2] en la segunda generación del satélite Flying-Laptop. Proporciona una plataforma genérica para misiones satelitales que, a su vez, incluye un marco de trabajo para el software de vuelo.
[80]	AC	OE1	Miranda et al. hacen una revisión sistemática y comparativa de diversas herramientas y tecnologías para el desarrollo de software en sistemas espaciales. Incluyen tecnologías como TASTE, cFS y SAVOIR.

[32]	AR	OE2	Zalewzki discute en este artículo los principios arquitectónicos del software empleado en RTES. Se centra en los sistemas de control distribuidos e incluye patrones de diseño siguiendo una metodología de desarrollo estructurada.
[27]	AR	OE1, OE3	Salazar et al. presenta un marco de desarrollo basado en modelos y componentes para RTES críticos implementados en Ada. Dicho marco está basado en tecnologías como UML, AADL, y el perfil de Ravenscar. En el aspecto práctico, introduce el sistema Sistema de Control de Actitud (ACS) del microsatélite UPMSat-2.
[81]	AC	OE1, OE2	De la Puente et al. introduce los conceptos generales para el desarrollo de software en aplicaciones espaciales en base a su experiencia en el OBSW de UPMSat-2. Introduce la arquitectura software de dicho sistema, de los cuales se extrajeron algunos patrones de diseño.
[37]	PHD	OE2	Batz. centra su tesis en el desarrollo de un marco de trabajo para el desarrollo de futuro software espacial incluyendo aspectos como FDIR, comunicación mediante eventos y mensajes, entre otros. Su trabajo está basado en la arquitectura de referencia del trabajo realizado en Future Low-cost Platform (FLP) [79].

Capítulo 6

Gestión del Proyecto

6.1. Metodología

Esta tesis sigue una metodología confirmada por cuatro fases (figura 6.1): revisión bibliográfica, análisis del sistema, desarrollo y resultados. La fase de revisión bibliográfica permitió obtener una serie de tecnologías, metodologías, patrones arquitectónicos y de diseño adecuados a sistemas RTES, y aplicables en el sector espacial. Paralelamente, se realizó el análisis del sistema HERCCULES, de los cuales se generaron los requisitos, objetivos y definición a nivel de sistema. Estas dos fases fueron necesarias para efectuar el desarrollo y extraer los resultados de este trabajo. El desarrolló consistió en la elaboración de: (i) la arquitectura y estructura del sistema (software y hardware) y (ii) el diseño detallado del software del sistema y la configuración del OBC para su conexión con las tarjetas electrónicas. Posteriormente se realizaron las actividades de codificación y diseño de los subsistemas que consistieron de la lectura de los sensores reales, adquisición de datos del sistemas, interconexión y pruebas de integración entre subsistemas. Finalmente se analizaron los resultados obtenidos de la fase anterior y se extrajeron las conclusiones de esta tesis.

6.2. Ciclo de Vida Software

El OBSW de HERCCULES entra en la categoría de RTES flexible. Es un sistema empotrado de tiempo real porque interactúa con el entorno físico mediante dispositivos hardware y responde a los estímulos de entrada en un tiempo finito y determinado de tiempo [82]. Es de tipo "flexible" debido a que, como se definió en el capítulo 4, el incumplimiento de los plazos no afecta significativamente a la validez del sistema. El ciclo de vida de este tipo de sistemas está condicionado principalmente por la validez funcional y temporal, de modo que conviene descubrir los errores lo más pronto posible. Asimismo, independientemente de su nivel de criticidad y obviando las modificaciones de parámetros solicitadas por tele-comandos, el código fuente del software de RTES es difícil, sino imposible de modificar/reparar en tiempo de ejecución.

Por todo esto, el ciclo de vida escogido para el desarrollo de HERCCULES está basado en el Modelo-V que está inspirado un el modelo de proceso clásico (o en cascada) ampliamente adoptado en RTES, sobre todo en el sector aeroespacial. Contrario a las metodologías Ágiles

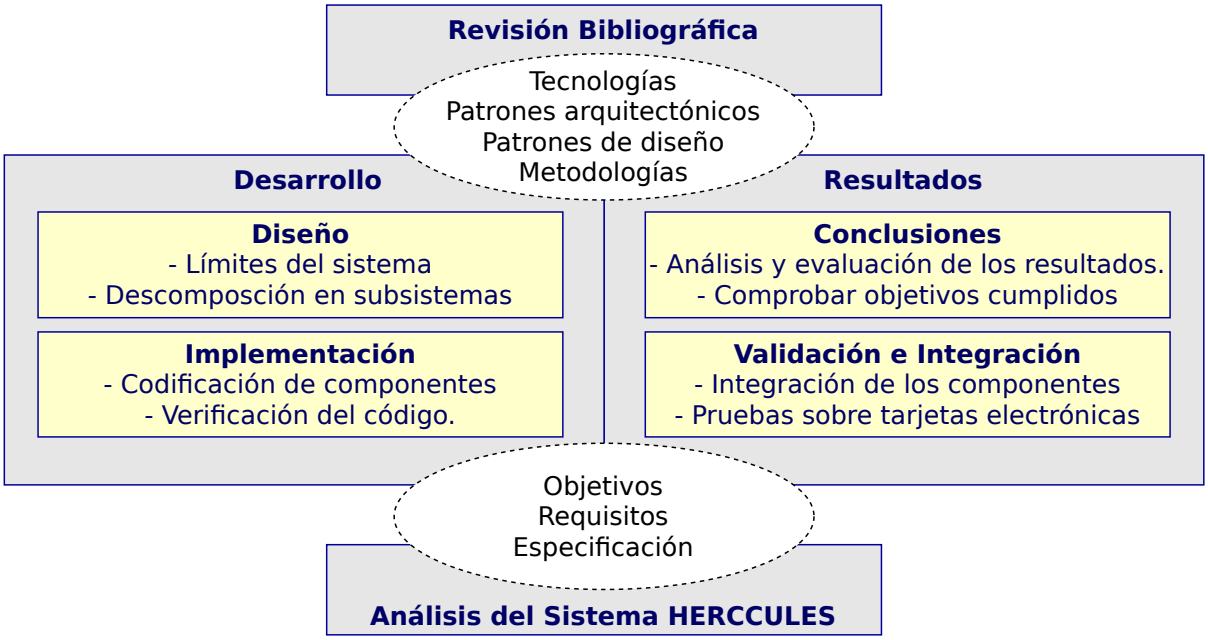


Figura 6.1: Metodología para el desarrollo de la tesis

que siguen un enfoque adaptativo, el Modelo-V está basado en un enfoque predictivo [83]. Esto lo hace adecuado para el desarrollo de (OBSW) sistemas espaciales debido a que los requisitos están claramente definidos en las fases iniciales [31]. Aunque el *Modelo-V* y el modelo en cascada tengan fases similares, el primero ofrece dos beneficios claves: (i) evita "parones" causados por tareas pendientes incompletas [84] y (ii) resalta la importancia de las campañas de verificación y validación, de hecho tiene un plan establecido para validar cada fase de desarrollo del proyecto [16].

La figura 6.2 ilustra las fases del Modelo-V. La parte izquierda del modelo representa las actividades de desarrollo, ahí se definen los requisitos y la especificación del sistema. Tras ello, se realiza el diseño arquitectónico del software y se divide el sistema en diversos subsistemas o paquetes funcionales. Posteriormente, se realiza el diseño de cada subsistema, subdividiéndolo en componentes software. En la fase de implementación, el diseño detallado se transforma en código fuente, ya sea manualmente o automáticamente. El flujo de trabajo continúa por el lado derecho de la "V", iniciando con las actividades de verificación para luego culminar con la validación del sistema. En el caso de HERCCULES, se estableció un conjunto de revisiones para evaluar el trabajo realizado en cada fase. Estos definen el progreso de la misión y están definidos en el plan de revisión del programa BEXUS que, a su vez, están inspirados en el estándar ECSS-E-ST-40C [85] y la guía ECSS-E-HB-40A [86] de la ESA:

- **Revisión de los Requisitos del Sistema – SRR**: Se evalúan los requisitos funcionales y no funcionales del sistema. También se comprueba que la planificación de la misión sea adecuada para los objetivos establecidos. Esta revisión se realizó en las instalaciones del IDR por los miembros del proyecto.
- **Revisión Preliminar de Diseño – PDR**: En esta revisión, los expertos de la ESA revisán el diseño preliminar de todos los subsistemas de HERCCULES, incluido el software. Aquí se demuestra que el diseño preliminar cumple con los requisitos del sistema sin

riesgos significativos. Los resultados de esta revisión sientan las bases para efectuar el diseño detallado.

- **Revisión Crítica del Diseño – CDR**: Esta revisión es realizada por miembros de la ESA y SNSA en el Centro de Investigación y Tecnología Espacial de la ESA (ESTEC). Se espera que en este punto del proyecto se haya realizado pruebas del software en placas de inserción (*bredboards*) o prototipos de alta fidelidad de la electrónica.

- **Revisión de Progreso de Integración – IPR**: Esta evaluación es realizada por dos expertos de la ESA y se lleva a cabo en las instalaciones del IDR. El objetivo es verificar que los recursos, procedimientos, y dispositivos para realizar las pruebas son adecuados

- **Revisión de Aceptación del Experimento – EAR**: Finalmente, durante esta revisión se espera que el la verificación y validación del proyecto se hayan superado. En caso contrario, el experimento HERCCULES no podrá ser considerado para la campaña.

Por ejemplo, los subsistemas software se identifican durante la fase de *Diseño Preliminar*, tras ello, se lleva a cabo la *Revisión de Diseño Preliminar (PDR)* y en caso de aceptación, se da paso a la fase de *Diseño Detallado*.

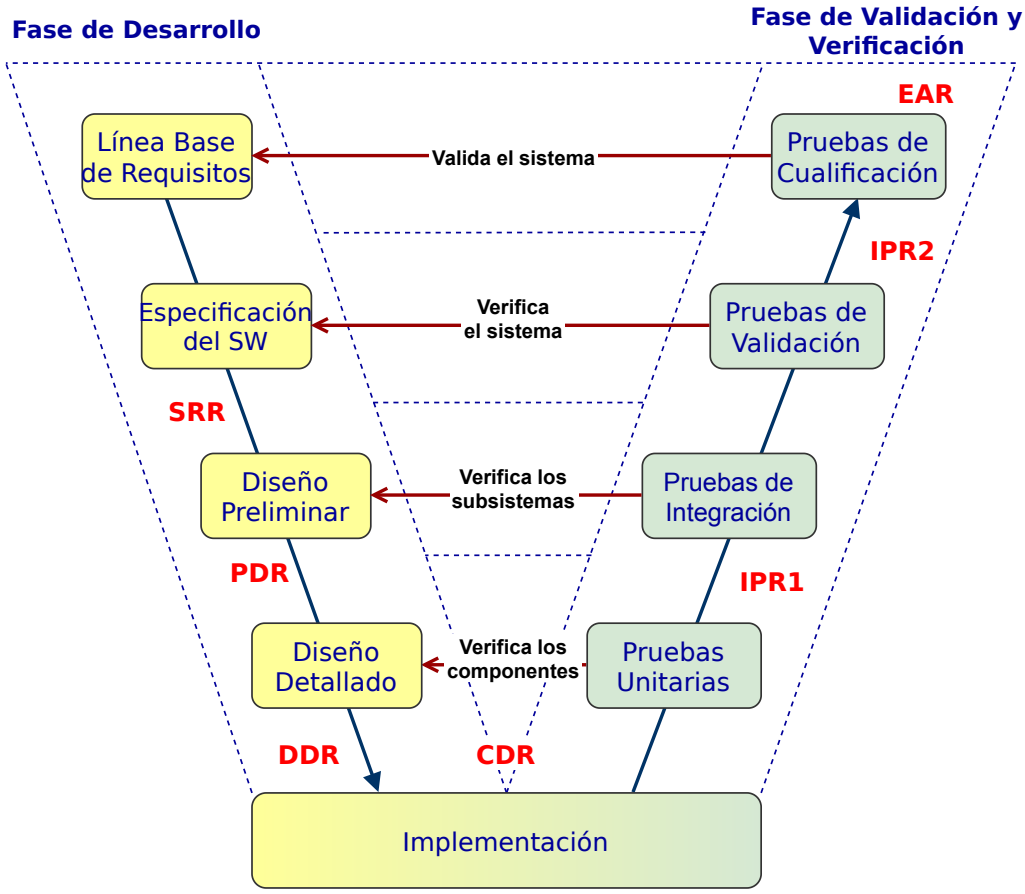


Figura 6.2: Modelo en V adoptado para el ciclo de vida del software del sistema HERCCULES

6.3. Planificación

El trabajo presentado en esta tesis tiene una duración total de 425 horas aproximadamente que se han distribuido desde el 28 de agosto de 2022 (cerca del inicio del semestre) hasta el 12 de enero de 2023. En ese lapso de tiempo, se realizaron las siguientes actividades:

- *Análisis y Educción de Requisitos*: Esta fase abarca gran parte de la primera actividad del ciclo de vida software (figura 6.2). Por lo tanto, en esta fase se realizó un estudio de la misión, y también se analizaron los requisitos del sistema para la obtención de una *línea base*. También fue necesario realizar reuniones con los miembros del proyecto para aclarar las dudas, acordar los requisitos y definir el Concepto de Operaciones (CONOPS) preliminar. Esta actividad tuvo una duración aproximada de trece días.
- *Estado del Arte*: Esta fase se corresponde con la etapa de análisis, es decir, se pretende estudiar, entender y clasificar lo que ya existe para crear algo que aún no existe. En este caso, se realizó un estudio del mercado actual, concretamente, se investigaron tecnologías y metodologías empleadas para el desarrollo del OBSW. También se estudiaron arquitecturas de OBSW y OBCs usados en misiones satélites y estratosféricas existentes. Estas tareas tuvieron una duración de dieciséis días.
- *Diseño Arquitectónico*: Esta actividad contempla las fases de diseño de alto nivel y bajo nivel, denominados diseño preliminar y detallado del ciclo de vida software. Ambos parte de la línea base, que contiene los requisitos del software del sistema. El diseño de alto nivel, hace referencia a la definición de los módulos o paquetes, y sus conexiones. Por otro lado el diseño de bajo nivel, refina cada paquete en diversos componentes software, estos a su vez se definen en objetos, al más bajo nivel. La duración de esta actividad fue de trece días.
- *Implementación*: Esta actividad se corresponde con su homónimo en el ciclo de vida software. Aquí se implementan todos los componentes software empleando las herramientas seleccionadas en la fase de análisis. Se ha subdividido en varias tareas, una por cada componente implementado. La duración total se llevó a cabo en cuarenta días.
- *Verificación y Validación*: Esta actividad está subdividida en tareas correspondientes con la fase de verificación y validación del ciclo de vida adoptado en HERCCULES. Se realiza en paralelo con la implementación y duró cincuenta días.
- *Documentación del TFM*: En esta actividad está reservada para la elaboración de esta memoria y la presentación.

La distribución de estas actividades se ilustra en el diagrama de Gantt, véase figura 6.3. La primera columna presenta el nombre de la actividad, tras ello, se muestra la duración y fecha de inicio de cada una. **La versión detallada se encuentra en el anexo A.2** y muestra la descomposición de las actividades en tareas. Los cinco hitos, representados como rombos al final de cada actividad, se corresponden con las fechas límite establecidas para culminar una actividad. Por ejemplo, el hito definido al final de la primera etapa (análisis y requisitos) no solo define el 11 de septiembre como límite para su completitud, sino que también describe los resultados esperados, los documentos de especificación del software del sistema (SSS) y de sus requisitos (SRS). Es importante aclarar que las actividades y tareas superpuestas (como la V&V y la implementación) no se efectúan en paralelo, sino de forma concurrente.

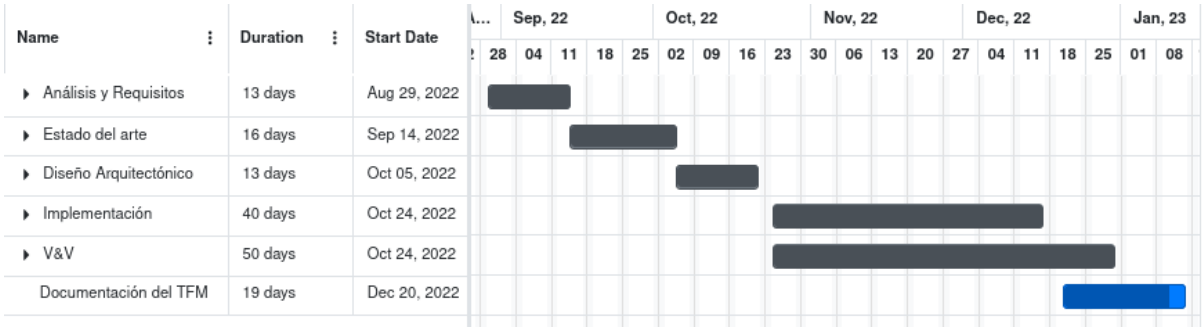


Figura 6.3: Gráfico de Gantt sintetizado del proyecto HERCCULES

6.4. Presupuesto

En esta sección se detallan los costes asociados a la realización y puesta en funcionamiento de este proyecto. Los costes se han organizado en tres grupos: Mano de obra, licencias de software, materiales hardware del OBDH y de sistemas externos como sensores o actuadores de los experimentos. Los precios de cada componente se corresponden con su precio de venta en el momento de la adquisición. Por lo tanto, es posible que su valor actual se haya visto modificado. No obstante, el precio del software con licencia de pago coincide con los precios de venta al público general.

6.4.1. Mano de Obra

Se estima que el coste promedio de ingeniería para un Ingeniero Informático junior es de **30** €/hora. Asimismo, en la planificación del proyecto se han estimado un total de **425** horas de trabajo. Por lo tanto sus honorarios ascienden a un valor aproximado de **12 750** €.

6.4.2. Licencias de Software

En este proyecto, se procuró usar en todo momento herramientas gratuitas de código abierto, y en su defecto, se optó por herramientas cuya licencia estaba cubierta por la UPM. Como se ha mencionado al inicio de la sección, para calcular los costes en licencias de software se han tomado en cuenta los precios de venta al público general, obviando las licencias académicas gratuitas y los convenios con la UPM.

Tabla 6.1: Costes de las licencias de software.

Concepto	Unidades	Coste Unitario [€/u.]	Coste Total [€]
CLion	1	229.00	229.0
Visual Paradigm	1	99.00	99.99
Microsoft 365	1	53.70	53.70
		Total	**381.7**
		Total + IVA (21 %)	**461.86**

6.4.3. Costes Materiales

La sección de costes materiales se ha subdividido en función de su relación con el OBDH. La tabla 6.2 incluye los dispositivos empleados directamente para el desarrollo del OBDH como OBC, almacenamiento, conectores adicionales, entre otros. En cambio, los dispositivos pertenecientes a la plataforma o carga de pago que han sido elegidos por otros miembros del proyecto se muestran de forma independiente y se ha limitado a los dispositivos que guardan relación directa con el OBDH como sensores digitales (tabla 6.4) o analógicos (tabla 6.3) que brindan información al OBC. Puede encontrar un listado detallado en [14, Sec. 4.3]. A diferencia de los costes en concepto de licencia software, el coste unitario de los materiales tiene en cuenta el IVA del 21 %. Teniendo en consideración, estas tres categorías, se estima un total de **12 109.198 €** en concepto de costes materiales.

Tabla 6.2: Costes de los dispositivos del OBDH.

Concepto	Unidades	Coste Unitario + IVA [€/u.]	Coste Total [€]
RPi Model 4B+	2	61.61	123.22
Adaptador 5V/2.5A	1	18.90	18.90
T-cobbler	1	7.99	7.99
Bread-board 19100	4	24.01	96.04
Memoria micro-SD 32 G	1	13.00	13.00
Jumpers para pines GPIO	10	2.95	29.5
Conector UART-a-SSL	1	8.99	8.99
Cable de red cruzado Ethernet	1	11.34	11.34
		Total	**308.98**

Tabla 6.3: Costes de dispositivos analógicos, externos al OBDH

Concepto	Unidades	Coste Unitario + IVA [€/u.]	Coste Total [€]
RS Pro Silicon Heater Mat. No 245-499	6	22.71	136.26
RS Pro Silicon Heater Mat. No 245-528	2	29.45	58.9
Termistor PT1000	40	2.56	102.61
Piranómetro SR20-T2-05	2	1770	3540
Pirgeómetro IR20-T2	2	3275	6550
Anemómetro LDES050BF6S	5	96.37	481.85
Fotodiodo Lms41PD-03	6	50	300
Tarjeta para fotodiodos	6	55	330
MOSFET IRFZ44PbF para control PWM	8	3.05	24.4
		Total	**11 522.709**

Tabla 6.4: Costes de dispositivos digitales, externos al OBDH

Concepto	**Unidades**	**Coste Unitario + IVA [€/u.]**	**Coste Total [€]**
Módulo ADC ADS51115 de DFRobot Gravity	5	8.82	44.1
Adafruit 9-DOF IMU BNO055	2	32.61	65.22
GPS Click mikroBUS MIKROE-1032	2	54.46	108.61
Sensor de potencia TECNOIOT INA226	2	6	12
HiLetgo MS5611 pressure sensor module meter	4	11.49	45.96
Sensor de temperatura TC74 de 3.3 Voltios	1	1.309	1.309
		Total	**277.509**

Capítulo 7

Análisis del sistema

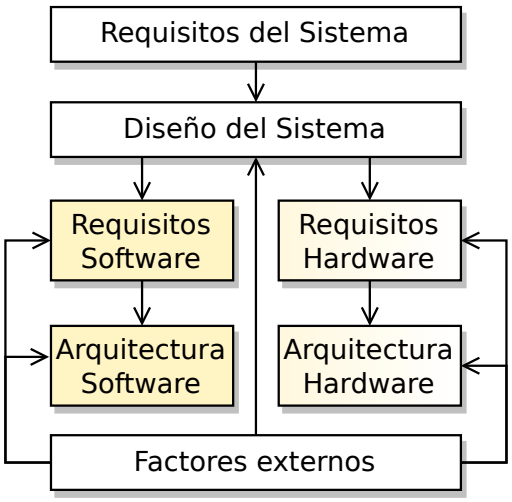


Figura 7.1: Requisitos de un RTES.

El proyecto HERCCULES es un RTES, es decir, se trata de un sistema compuesto por software que interactúa con el entorno a través del hardware. Por lo tanto, el diseño del OBSW no solo está sujeto a los requisitos software, sino que depende de los requisitos y diseño del propio sistema (figura 7.1). A su vez, los requisitos software, están subordinados a factores externos e internos. Los factores externos, como en los sistemas informáticos convencionales, comprenden restricciones ambientales, limitaciones monetarias, de personal, etc. Por otro lado los factores internos al proyecto HERCCULES son causados por la fuerte relación que existe entre el software y el hardware. En consecuencia, los requisitos software aquí descritos también reflejan limitaciones impuestas por el hardware y diseño del sistema. Por este motivo, antes de describir los requisitos del sistema, se proporciona una descripción funcional y operacional del sistema HERCCULES a través de su Concepto de Operaciones (CONOPS).

7.1. Concepto de Operaciones: Descripción General del Sistema

El Concepto de Operaciones guarda una estrecha relación con la funcionalidad del software del sistema ya que especifica los eventos desencadenados durante toda la misión, el orden en que se producen, y las acciones que debe realizar en su recepción. La figura 7.2 muestra el CONOPS de esta misión y brinda una perspectiva general de su comportamiento. En dicha figura se identifican las distintas fases o estados de la misión y las actividades que realiza el sistema en cada una de ellas. En el caso de HERCCULES, la transición de las fases depende de la altitud de la góndola y coincide con las etapas de la misión definidas en la campaña BEXUS-

32 y resumidas en la sección 1.1.2. A continuación se detallan las actividades realizadas por el sistema en las cinco etapas, centrándose en el comportamiento del OBSW:

1. *Prelanzamiento y lanzamiento*: La misión HERCCULES comienza inicia minutos antes del lanzamiento, donde todo el sistema se configura en un modo de bajo consumo con todos los subsistemas apagados. Tras ello, el operador encenderá todos los experimentos, mediante una serie de TCs, con el fin de probar la funcionalidad de los experimentos y verificar la comunicación con entre el segmento de vuelo y la estación terrenal. Si esta verificación se realiza sin complicaciones, el sistema vuelve al estado de bajo consumo. En caso contrario, los operadores deberán dirigirse a la góndola para comprobar las conexiones físicas, alimentación, etc. Esta fase culmina cuando el experimento se haya liberado.

2. *Ascenso*: Esta fase tiene lugar después de la fase de lanzamiento y culmina cuando la góndola alcanza la estratosfera, a una altitud que varía de 25 a 30km. La entrada a esta fase sucede de forma automáticamente en función de la presión atmosférica o a través de TCs enviados por los operadores de HERCCULES. En este punto, todos los subsistemas se encienden automáticamente y el OBSW da inicio a la adquisición y recogida de datos de todos sus subsistemas. Los calentadores del HTL y EL se controlan en base a las temperaturas obtenidas de los termistores PT1000. Adicionalmente, los operadores podrán controlar estos calentadores mediante TCs. Por motivos de seguridad los operadores podrán enviar TCs para reiniciar los dispositivos I2Cs, ya que estos están sujetos a fallos después de largos períodos en actividad. Gran parte de los datos obtenidos en el vuelo se envían como TM a la estación de tierra con un periodo de un segundo.

3. *Flote*: La transición a la fase de flote sucede aproximadamente de 1,5 horas desde el lanzamiento, cuando el globo alcanza la estratosfera. En esta fase, todas las actividades de los subsistemas permanecen sin cambios, a excepción del HTL que entra en un modo de flote específico, donde la potencia disipada por los calefactores difiere de los modos anteriores. Aquí, la TM se envía con el mismo periodo (1seg) y el sistema sigue a la escucha de los TCs.

4. *Descenso*: La entrada a esta fase es especificada por los operadores de HERCCULES mediante un TC específico. Una vez dentro, se apagan todos los subsistemas y calefactores de HERCCULES. Tras ello, se detiene el envío de la TM, el OBSW finaliza su ejecución y el OBC se apaga automáticamente.

5. *Recuperación* Tras haber recuperado la góndola, se extra la información grabada a bordo para dar inicio al análisis de los datos. Por último, el equipo científico de HERCCULES informa de los resultados técnicos.

Además de estas cinco etapas, HERCCULES contempla dos modos adicionales que permiten un control manual y autónomo de los calefactores de los experimentos. Estos modos operacionales son comúnmente denominados como estados de *apertura* y *cierre* en las misiones satélites [3, 87]. En el contexto de HERCCULES, estos modos permiten que en caso de consumo elevado o problemas con la batería (externa a la misión y gestionada por los organizadores de la campaña) los calefactores disipen menos potencia, según el criterio de los operadores de HERCCULES.

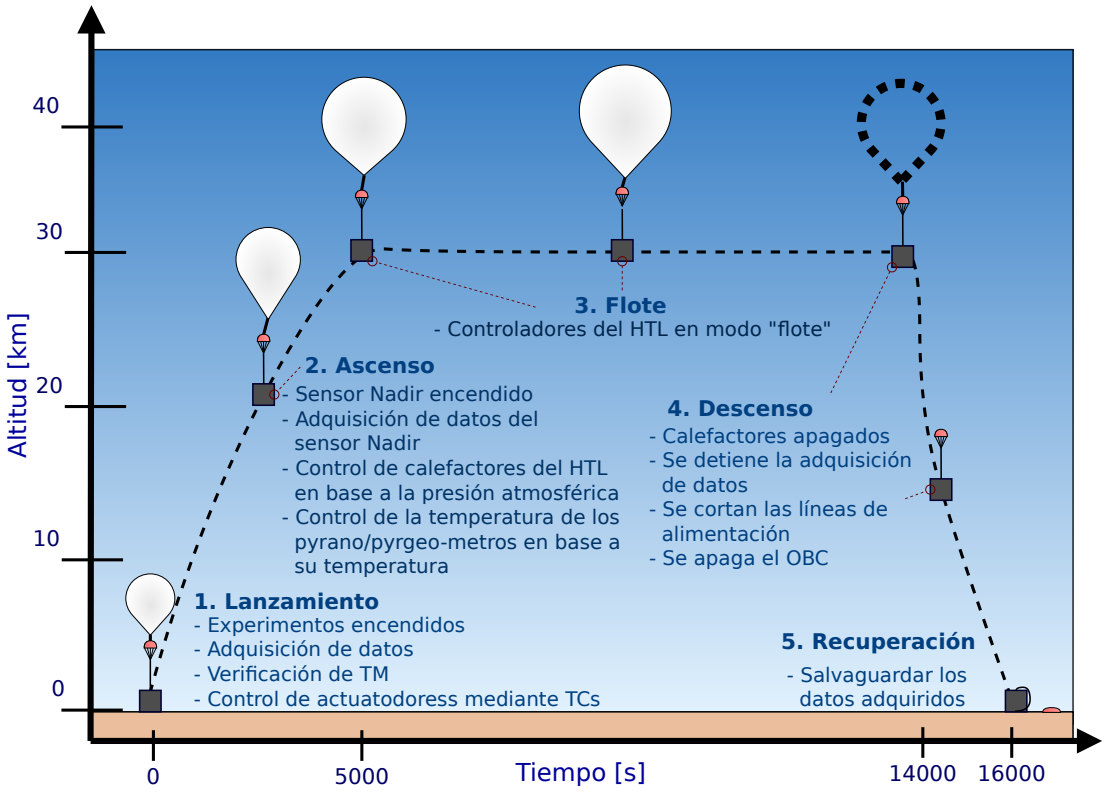


Figura 7.2: Concepto de Operaciones de HERCCULES.

7.2. Estrategia para la Especificación de los Requisitos

La estrategia seguida para la obtención de los requisitos está basada en la filosofía needs-features-requirements de Leffingwell y Widrig (figura 7.3) en la que se propone una técnica basada en una descripción jerárquica del sistema [88]. Las necesidades (*needs*) se corresponden con los objetivos de alto nivel del producto, que en este caso coincide con los objetivos primarios y secundarios del sistema HERCCULES. Estas necesidades u objetivos son satisfechas por las prestaciones o características del sistema (*features*). Las prestaciones describen de forma muy abstracta los servicios que proporciona el sistema. En el nivel más bajo de la pirámide se encuentra los requisitos (*requirements*) que describen el comportamiento del sistema con mayor detalle. Esta pirámide se puede descomponer en más niveles como casos de uso o escenarios, pero se han omitido para mantener la claridad y brevedad de la memoria. El objetivo de las secciones subsecuentes es brindar una descripción breve, pero completa, de los objetivos, prestaciones y requisitos elicitados para el software del sistema HERCCULES, prestando especial atención a los requisitos del software del segmento de vuelo.

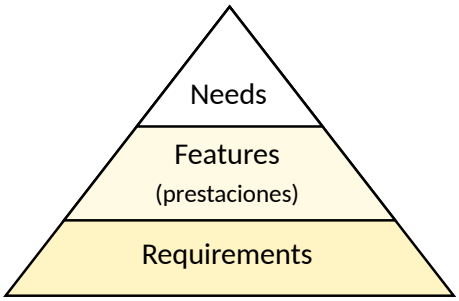


Figura 7.3: Piramide de Leffingwell y Widrig [88].

7.3. Necesidades

El sistema HERCCULES cuenta con dos partes bien diferenciadas: el segmento de vuelo (soportado por el OBSW) y terrestre (soportado por el GSW). Por lo tanto, conviene separar las necesidades software del sistema en función de los objetivos del segmento al que den soporte.

N-01 El OBSW debe *realizar por medio del ordenador embarcado la el control, la adquisición y la gestión de los datos provenientes de la plataforma y carga de pago de HERCCULES*.

N-02 El GSW debe *proporcionar a los operadores una plataforma que permita controlar y visualizar el experimento HERCCULES de forma remota*.

7.4. Prestaciones (features)

Las prestaciones o requisitos de alto nivel para el OBSW del sistema HERCCULES son:

P-01 Controlar y gestionar el estado de los dispositivos.

P-02 Adquirir los datos y controlar los experimentos y subsistemas de HERCCULES.

P-03 Gestionar los modos operacionales a nivel de sistema y subsistema.

P-04 Implementar las operaciones del OBDH, es decir, ejecutar TCs y enviar TM de/a la GS.

P-05 Almacenar en memoria no volátil todos los datos de TM adquiridos.

P-06 Almacenar en memoria no volátil el tiempo relativo, y modo de operación del sistema más reciente.

Por otro lado, el GSW del sistema HERCCULES debe implementar las siguientes prestaciones:

P-07 Recibir los datos (TM) enviados por el segmento de vuelo.

P-08 Guardar la TM en almacenamiento no volátil.

P-09 Permitir el envío de comandos (TCs) al segmento de vuelo.

P-10 Proveer una Interfaz Gráfica de Usuario (GUI) intuitiva para el envío de TCs y la visualización de la TM.

7.5. Requisitos del Software

En aras de la simplicidad y de acuerdo a las guía de recomendaciones del programa BEXUS [11], los requisitos se han clasificado en dos categorías: requisitos funcionales y no funcionales. Este último, a su vez, está subdividido en requisitos operacionales, de rendimiento y de diseño. La figura 7.4 ilustra estos cuatro tipos de requisitos y también representa el flujo de trabajo idealizado para la obtención de estos, aunque obvia su relación con los factores externos y las restricciones propios de un RTES.

Las siguientes subsecciones contienen la especificación formal de la línea base del sistema HERCCULES. Es decir, se describe el conjunto de requisitos que han sido aceptados y fijados por

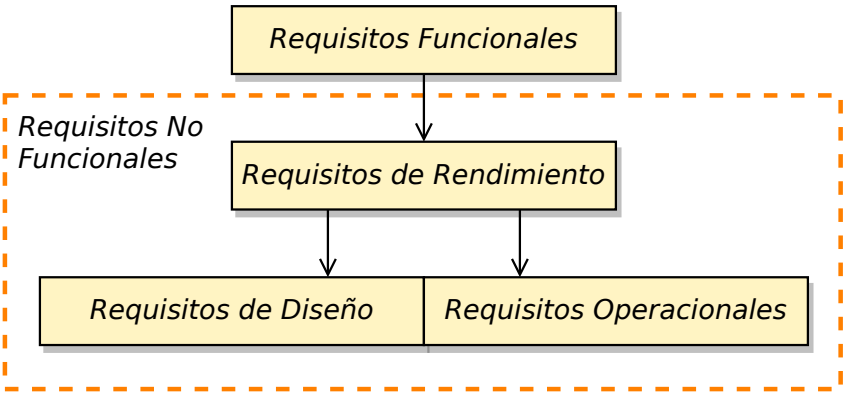


Figura 7.4: Clasificación de requisitos software y de sistema. Adaptado de [11, Fig. 2.1].

todos los miembros del proyecto. Cada subsección agrupa una serie de requisitos organizados en sendas tablas y compuestas de los siguientes atributos marcados en negrilla:

Un **Identificador (ID)** único para el requisito con formato **<O/D>-<YY>-<ZZ>**, donde:

- **O/D** representa la severidad del requisito. Los requisitos obligatorios están denotados con la **O**, y su comprobación se debe realizar con un método de verificación concreto. Por otro lado, los requisitos opcionales o deseables se representan con la **D** y su objetivo es contribuir en la mejora de rendimiento o prestaciones del software.
- **YY** hace referencia al tipo de requisito, puede ser funcional (**FU**), operacional (**OP**), de rendimiento (**RE**), o de diseño (**DI**).
- **ZZ** se corresponde con el identificador numérico de dos dígitos que permite rastrear unívocamente al requisito en cuestión.

Una **Descripción** resumida de los requisitos que respeta los atributos de calidad definidos en [86], es decir, se describen de forma breve, concisa, entendible y sin ambigüedades. Asimismo, la descripción de los requisitos obligatorios (marcados con la O) incluyen la palabra clave "debe" y los opcionales (marcados con la D) están descritos con la palabra clave "debería" (shall y should en Inglés, respectivamente).

Los **Métodos de Validación (MV)** empleados para demostrar que el requisito se pueda satisfacer. Se han establecido cuatro métodos en base al estándar ECSS-E-ST-10-02C [89] para Ingeniería de Sistemas de la ESA, su definición difiere de la original ya que han sido adaptados a la verificación de sistemas software:

- Prueba o Test (**T**): Es el tipo de verificación habitual de las aplicaciones informáticas de propósito general, donde a partir de un conjunto de entradas determinadas, se comprueba que el sistema genera las salidas esperadas.
- Análisis (**A**): Consiste en predecir o deducir (i.e. analizar) el sistema a partir de una serie de pruebas de estrés, experimentos (i.e. programas *demo*) o simulación de los cuales se extrae un estudio estadístico y análisis sobre el comportamiento del sistema. También incluye la *verificación por similitud*, en la cual se da por verificado un componente software si uno similar voló con éxito en misiones previas.

- Revisión de diseño (**R**): Emplea documentos de diseño (e.g.: esquemas, diagramas, modelos, etc.) para demostrar que el sistema funcionará según lo previsto. En el campo de la Ing. Software, también se puede considerar una revisión de diseño la inspección o revisión del código fuente, ya sea visual o automatizada con el uso de *linters* u otras herramientas.
- Inspección (**I**): A diferencia de la revisión de diseño, la verificación por inspección se realiza mediante la visualización o inspeccionado del *comportamiento del sistema*. Un ejemplo claro de verificación por inspección es cuando se comprueba el cambio en las lecturas de un sensor de temperatura que ha sido estimulado.

7.5.1. Requisitos Funcionales

Los requisitos funcionales describen los servicios, operaciones, o *funciones* que se deben implementar para que la misión pueda cumplir con sus objetivos. Puede encontrar los requisitos de rendimiento tanto del sistema como del software detallados en [14, sección 2.1].

Tabla 7.1: Requisitos funcionales.

ID	Descripción	MV
O-FU-01	El OBSW *debe* recoger y almacenar las temperaturas del HTL desde la fase de lanzamiento hasta el descenso.	T,I
O-FU-02	El OBSW *debe* recoger y almacenar la potencia disipada por los calefactores del HTL desde la fase de ascenso hasta el descenso.	T,I
O-FU-03	El OBSW *debe* controlar los calefactores del HTL de forma independiente de acuerdo a las especificaciones del propio subsistema [14].	T,I
O-FU-04	El OBSW *debe* implementar los distintos modos de operación para el control de los calefactores y cambiar el modos de operación de forma autónoma.	T,I
O-FU-05	El OBSW *debe* recoger y almacenar los datos del piranómetro desde la fase de lanzamiento hasta el descenso.	T,I
O-FU-06	El OBSW *debe* recoger y almacenar los datos del pirgeómetro desde la fase de lanzamiento hasta el descenso.	T,I
O-FU-07	El OBSW *debe* recoger y almacenar los datos del barómetro absoluto desde la fase de lanzamiento hasta el descenso.	T,I
O-FU-08	El OBSW *debe* recoger y almacenar los datos de la IMU y GPS desde la fase de lanzamiento hasta el descenso.	T,I
D-FU-09	El OBSW *debería* almacenar los datos del GPS en formato crudo (NMEA).	I
O-FU-10	El OBSW *debe* recoger y almacenar los datos del sensor de Nadir (i.e.: fotodiodos) desde la fase de ascenso hasta el descenso.	T,I
O-FU-11	El OBSW debe gestionar los modos del sistema y subsistemas (experimentos + laboratorios).	T,I
O-FU-12	El OBSW *debe* salvaguardar el último modo del sistema y subsistemas, así como el tiempo absoluto y relativo de la misión desde la fase de lanzamiento hasta el descenso.	I,R

O-FU-13	El OBSW *debe* añadir a los datos almacenados el tiempo absoluto, relativo, modo de operación del sistema, y del subsistema al que pertenece.	I,R
O-FU-14	El OBSW *debe* grabar los datos en distintos ficheros, uno por experimento.	I
O-FU-15	El GSW *debe* descodificar y mostrar todos los datos de TM recibidos.	T,I
O-FU-16	El GSW *debe* organizar los datos de TM recibidos en función del sistema al que pertenezca.	I,R
O-FU-17	El GSW *debe* mostrar un mensaje de error cuando se pierda la conexión con el segmento de tierra.	T, I, R
O-FU-18	El GSW *debe* almacenar la TM recibida.	T,I
D-FU-19	El GSW *debería* mostrar un testigo (por la GUI) que represente el estado de conexión con el segmento de tierra.	T,I
O-FU-20	El GSW *debe* proveer una opción que permita almacenar un TC en memoria persistente.	T,I
O-FU-21	El GSW *debe* proveer una opción que permita recuperar un TC almacenado previamente.	T,I

7.5.2. Requisitos de Rendimiento

Los requisitos de rendimiento se centran principalmente en los requisitos temporales del OBSW como los periodos de monitorización o tiempos límites de respuesta. También se cubren aspectos como precisión, exactitud, y tasas para la transmisión de los datos. Puede encontrar los requisitos de rendimiento tanto del sistema como del software detallados en [14, sección 2.2].

Tabla 7.2: Requisitos de rendimiento.

ID	Descripción	MV
O-RE-01	El OBSW *debe* recoger y almacenar los datos del subsistema PCU con un periodo máximo de 5 segundos.	I,R,A
O-RE-02	El OBSW *debe* recoger y almacenar los datos del EL (barómetros, piranómetros y pirgeómetros) con un periodo máximo de 1 segundo.	I,R,A
O-RE-03	El OBSW *debe* recoger y almacenar los datos de los sensores y actuadores del HTL con un periodo máximo de 10 segundos.	I,R,A
O-RE-04	El OBSW *debe* recoger y almacenar los datos de los dispositivos del ATL (fotodiodos y termistores) con un periodo máximo de 1 segundo.	I,R,A
O-RE-05	El OBSW *debe* recoger y almacenar los datos de los del GPS y magnetómetro (embebido en la IMU) con un periodo máximo de 1 segundo.	I,R,A
O-RE-06	El OBSW *debe* recoger y almacenar los datos del acelerómetro y giroscopio (embebidos en la IMU) con un periodo máximo de 0.01 segundos.	I,R,A
O-RE-07	El OBSW *debe* ejecutar los TCs con un plazo máximo de 5 segundos desde su envío en la GS.	I,R,A
O-RE-08	El OBSW *debe* salvaguardar el modo del (sub-)sistema, tiempo absoluto y relativo más reciente con un periodo máximo de 1 segundo.	I,R,A
O-RE-09	El OBSW *debe* proporcionar el tiempo relativo y absoluto de la misión con una precisión de 1 mseg.	I,R,A

O-RE-10	El OBSW *debe* enviar TM y recibir TCs con un ancho de banda máximo de 2 Mbps.	I,A
O-RE-11	El GSW *debe* enviar TCs y recibir TM con un ancho de banda máximo de 2 Mbps.	I,A
D-RE-12	El GSW *debería* informar al operador en, a lo sumo, 1 segundo de la perdida de conexión con el segmento de vuelo.	I,R,A
O-RE-13	El GSW *debería* informar al operador de la TM recibida en un plazo máximo de 1 segundo desde su recepción.	I,R,A
O-RE-14	La GUI del GSW debería tener un tiempo de respuesta máximo de 500 mseg.	I,R,A

7.5.3. Requisitos de Diseño

Los requisitos de diseño cubren los aspectos de confiabilidad/fiabilidad, disponibilidad, seguridad, mantenibilidad, portabilidad, usabilidad, y FDIR (i.e.: comportamiento en caso de error). En esta categoría también se han tenido en cuenta las restricciones técnicas, de negocio y las relacionadas con el cumplimiento de estándares. Puede encontrar los requisitos de diseño tanto del sistema como del software detallados en [14, sección 2.2].

Tabla 7.3: Requisitos de diseño.

ID	Descripción	MV
O-DI-01	El OBSW *debe* enviar la TM y recibir TCs a través de la unidad aeromóvil del sistema “E-Link”.	I
O-DI-02	El OBC/OBSW *debe* comunicarse con la unidad aeromóvil del sistema “E-Link” a través del protocolo Ethernet.	I
O-DI-03	La GSW *debe* enviar los TCs y recibir la TMs a través de la unidad terrestre del sistema “E-Link”.	I
O-DI-04	La GSW *debe* comunicarse con la unidad terrestre del sistema “E-Link” a través del portocolo Ethernet.	I
O-DI-05	El GSW *debe* conectarse a la red “E-Link Network” para la comunicación con el OBSW.	I,R
O-DI-06	El GSW no *debe* conectarse a la red “Guest Nerwork” ya que esta no puede estar conectada a Internet.	I,R
O-DI-07	El GSW y OBSW *deben* configurarse con sendas direcciones IP provistas por la ESA.	I,R
D-DI-08	La GS *debería* emplear un switch para distribuir el acceso a Internet a otras máquinas.	I
D-DI-09	La arquitectura del OBSW y GSW *debería* estar descrita/modelada en los lenguajes de modelado UML, HRT-UML y/o AADL.	R
O-DI-10	El OBSW *debe* implementar funcionalidad FDIR para no detenerse en caso de errores de comunicación causados por tiempos muertos y desconexiones fortuitas.	T,I,A

O-DI-11	El OBSW *debe* debe poder ser configurable en modo vuelo (abarca desde el ascenso hasta el descenso) incluso si no ha despegado o no se han cumplido las condiciones para el transito de estados.	T,I,R
O-DI-12	Tras pasar las pruebas FST durante la revisón EAR, los cambios en el OBSW *deben* ser mínimos y se deben ejecutar todas las pruebas de regresión. pasar	R
D-DI-13	El OBSW se *debería* implementar en el lenguaje de programación C++, C, Ada o Spark.	R
O-DI-14	El OBC *debe* apagar el módulo Wi-Fi.	I,R
D-DI-15	Durante las pruebas de comunicación se *deberían* simular perdidas de paquetes de red.	I,A
D-DI-16	El OBSW *debería* seguir, en la medida de los posible, los estándares de codificación de la NASA/JPL y MISRA para los lenguajes C y C++.	R

7.5.4. Requisitos Operacionales

Los requisitos de operacionales se resumen en la tabla 7.4 y se tratan de requisitos que el experimento debe cumplir para ser operado de forma segura y fiable. Los requisitos operacionales incluyen operaciones soportadas por el sistema en todas las fases de la misión, i.e.: desde el prelanzamiento hasta el descenso. En relación con el software, estos requisitos están fuertemente relacionados con los TCs que soporta. Puede encontrar los requisitos operacionales tanto del sistema como del software detallados en [14, sección 2.3].

Tabla 7.4: Requisitos de diseño.

ID	Descripción	MV
O-OP-01	El OBSW *debe* ser capaz de cambiar a cualquier modo de operación cuando lo solicite el operador a través de un TC.	T,I,R
O-OP-02	El OBSW *debe* soportar un modo autónomo configurable via TC. Dicho modo inhabilita al operador de controlar los actuadores y modos de operación de forma manual.	T,I,R
O-OP-03	El OBSW *debe* soportar un modo manual configurable via TC. Dicho modo permite que el operador pueda controlar los actuadores y modos de operación a través de TCs.	T,I,R
D-OP-04	El OBSW *debería* grabar los errores producidos en memoria persistente.	T,I,A
O-OP-05	El OBC/OBSW *debe* soportar múltiples secuencias de reinicio.	T,I
O-OP-06	La OBC *debe* reiniciarse si el OBSW está suspendido por más de 10 segundos.	T,I
O-OP-07	La comunicación entre OBSW y GSW *debe* ser capaz de restablecerse sin errores.	A,I

Capítulo 8

Sistema Desarrollado

Este capítulo tiene como objetivo describir las actividades que se han realizado para la implementación del sistema e incluyen: el diseño de alto nivel (o arquitectura software), el diseño de bajo nivel (o diseño detallado) y la la implementación del software del sistema HERCCULES Todas estas actividades se han llevado a cabo siguiendo las metodología CBD y MBD, con el soporte del conjunto de herramientas de TASTE. En consecuencia, el desarrollo del sistema aquí mostrado, no solo viene definido por el ciclo de vida en "V" que se definió en la planificación (sección 6.2), sino también por el proceso de desarrollo ASSERT.

8.1. Arquitectura General del Sistema

El proceso ASSERT, de forma similar al método HRT-HOOD, establece como primer paso el desarrollo de una arquitectura lógica que no tenga en consideración los aspectos de la plataforma de ejecución. Es decir, promueve la descomposición de la arquitectura en diversos módulos e interfaces funcionales que —como se puede intuir— están supeditados a los requisitos puramente funcionales del software del sistema (sección 7.5.1). En este caso concreto, el primer paso de diseño se ha visto ligeramente modificado ya que el software y el hardware se han diseñado conjuntamente (denominado co-diseño software/hardware). De modo que, la arquitectura lógica del sistema no solo refleja los elementos software puramente funcionales, sino también la arquitectura distribuida del sistema.

Como se vio a lo largo de la etapa de análisis, las necesidades, prestaciones y requisitos sugieren distribuir la funcionalidad del software en dos partes: OBSW y GSW. Asimismo, la arquitectura del sistema establece que dichas partes estarán desplegados en el OBC y computador de la GS, respectivamente. El diagrama de despliegue UML en la figura 8.1 modela la arquitectura lógica y de sistema con un alto nivel de abstracción. Se observa que (i) el OBSW es un componente que ofrece la interfaz `TC_Operations`, esta contiene la función `Send_TC` que permite el envío de los TCs desde la GS. Por otro lado, (ii) el GSW representa otro componente independiente desplegado en la GS. Este depende de la interfaz provista por el OBSW para la ejecución de los TCs e implementa la interfaz *TM_Reception* que permite al OBSW realizar el envío de la TM a través de la función `Send_TM`.

Las siguientes subsecciones se centran en la arquitectura software de estos dos sistemas (`GSW`

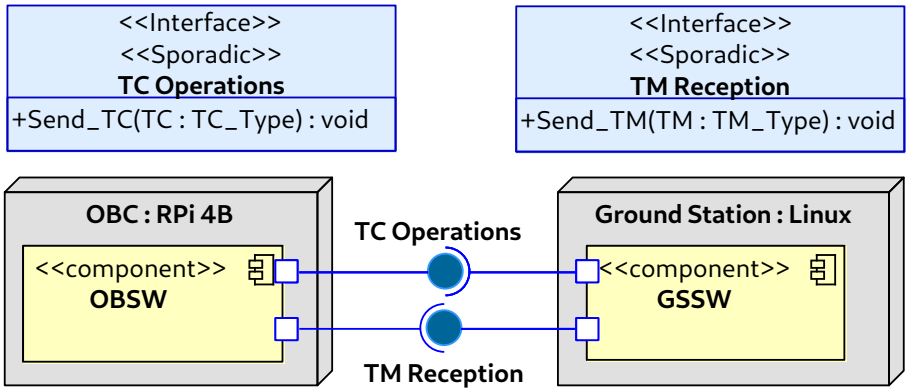


Figura 8.1: Diagrama de despliegue UML del software del sistema HERCCULES.

y `OBSW`). Para diferenciar los distintos niveles en la jerarquía de la arquitectura, se ha visto por conveniente, clasificar los módulos software en tres categorías, de acuerdo a su nivel de abstracción (de mayor a menor): *paquetes*, *componentes*, y *objetos terminales*. Por lo tanto, las arquitecturas descritas de aquí en adelante estarán conformadas por estos tres elementos básicos.

En cuanto a principios de diseño, se ha procurado que los componentes software cumplan características como acoplamiento mínimo, alta cohesión funcional, y grado de complejidad acorde a la jerarquía (relación inversa).

8.2. Arquitectura del Componente OBSW

Esta subsección tiene como objetivo describir el diseño lógico del OBSW. Sus requisitos funcionales sugieren una descomposición formada por los siguientes paquetes (figura 8.2):

- **Capa de Abstracción de Hardware** (**`HAL`**, por sus siglas en Inglés): Su cometido es proveer una serie de operaciones que permitan facilitar el acceso a los dispositivos (sensores y actuadores) de la plataforma y la carga de pago del experimento HERCCULES. Asimismo, este paquete abstrae las tres RTUs, es decir: PCU, SDPU, y TMU.
- **Subsistemas** (**`Subsystems`**): Este paquete encapsula componentes que gestionan los experimentos, laboratorios y subsistemas del experimento HERCCULES, a saber: HTL, PCU, EL, NADS, y ATL. También incluye componentes encargados de sincronizar el acceso concurrente a las tarjetas electrónicas o RTUs, por ello, tiene una dependencia (flecha punteada) con la `HAL`.
- **Gestor del sistema** (**`Manager`**): Su función primordial es, como se puede intuir, gestionar los aspectos que afecten a todo el experimento, es decir, está a cargo del modo de operación, la distribución y ejecución de los TCs y el procesamiento de los eventos generados por otros paquetes. En consecuencia, este paquete tiene una dependencia circular/-bidireccional con todos los subsistemas de HERCCULES (i.e.: paquete `Subsystems`).
- **Banco/Almacén de Datos Central** (**`Data_Pool`**): El OBSW adopta una arquitectu-

ra "orientada/centrada en los datos", prueba de ello es la presencia de este paquete. Su función es encapsular y gestionar el acceso concurrente a los datos generados por los subsistemas. De esta forma, el `Data_Pool` actúa como mediador para los otros paquetes ya que ofrece un punto de acceso común a los datos recogidos por los subsistemas. Brinda un nivel mayor de abstracción que el `hal`, ya que se centra en los datos y no en los dispositivos que los genera.

- **Registro de Datos (`Data_Storage`)** Este paquete ofrece los servicios para el almacenamiento de los datos (como TM) en memoria persistente (i.e.: la tarjeta microSD de la RPi). El paquete `TT&C` depende de este paquete para el grabado de la TM. Mientras que los paquetes `Manager` y `Subsystems` registran los eventos como cambios de modo y/o errores.
- **Telemetría, seguimiento y orden `TT&C`**: Este paquete lleva a cabo las funciones de comunicación con la GS. Se encarga de monitorizar la TM del situada en la `Data_Pool` y enviarla a la GS periódicamente. Asimismo, descodifica los TCs enviados por la GS y los redirige al `Manager` para su ejecución.

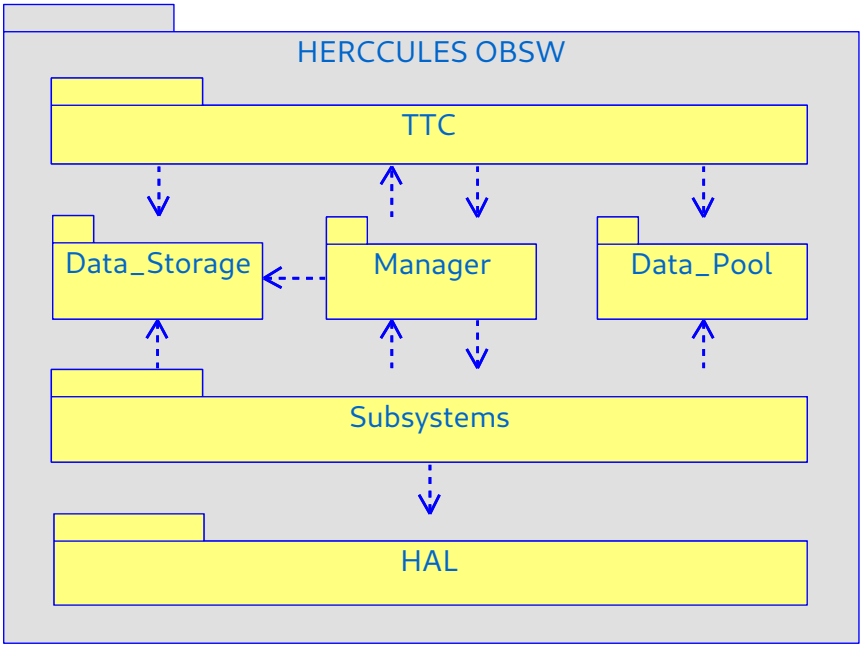


Figura 8.2: Diagrama de paquetes UML del segmento de vuelo.

La figura 8.2 ilustra el diagrama de paquetes del sistema modelado en el estándar UML. Se pueden apreciar todos los paquetes antes discutidos y sus relaciones de dependencia. Bajo esta arquitectura, existe una capa adicional correspondiente al SO y sus manejadores de dispositivos. Ambos representan el *entorno de ejecución* y dan soporte a todos los paquetes del OBSW.

La estructura de esta arquitectura se ha creado sobre la base de los siguientes patrones y estilos arquitectónicos: (i) patrón de *Datos Compartidos*, donde el `Data_Pool` cumple el rol del elemento `Data_Store`, (ii) estilo de *Capas Jerarquicas*, donde el `HAL` representa la capa de más bajo nivel, y los demás elementos conforman una capa a nivel de aplicación, y (iii)

el patrón *Proxy*, donde el paquete `TT&C` actúa como sustituto del GS, ocultando a los demás paquetes la comunicación remota subyacente.

Cabe destacar que los tipos de datos empleados a lo largo del desarrollo se ha encapsulado en un paquete independiente denominado **`Data_Types`**. Esta decisión de diseño viene dada por el proceso ASSERT de TASTE, el cual promueve la separación entre tipos de datos y funciones de transformación sobre esos tipos. Este diseño también se empleó en el OBSW del microsatélite artificial UPMSat-2, donde el paquete `Basic_Types` hace la función del paquete `Data_Types` de HERCCULES.

Las siguientes subsecciones presentan la arquitectura de estos paquetes. Los más complejos están descritos siguiendo las recomendaciones de la guía ECSS-E-HB40-A [86] y el estándar ECSS-E-ST-40C [85] para Ingeniería de Software, donde se divide la arquitectura software en arquitectura dinámica y estática. Por un lado, la *arquitectura estática*, que se centra en la estructura, es decir, los módulos funcionales (o componentes software) y las relaciones que existen entre ellos (o las interfaces). Por otro lado, la *arquitectura dinámica* describe las partes no funcionales y el comportamiento dinámico de las tareas. Esta última coincide con la arquitectura física de la metodología HRT-HOOD y la vista de concurrencia del proceso ASSERT. En cambio, la arquitectura estática se corresponde con la arquitectura lógica de HRT-HOOD y la vista de interfaz del proceso ASSERT.

8.2.1. Capa de Abstracción de Hardware (HAL)

8.2.1.1. Arquitectura Estática

La capa de abstracción de hardware está inspirada en la arquitectura FLP [2, 79] y se descompone en cuatro componentes que refleja la topología hardware del OBDH. Como se puede observar en el diagrama de la figura 8.3, al más bajo nivel se encuentra el componente Gestor de Buses (**`Bus_Handlers`**) que se encarga de gestionar el acceso a los buses de datos I2C y UART del OBC mediante las interfaces `I2CBusOperations` y `UARTOperations`, respectivamente. Estas operaciones son envoltorios de las bibliotecas *i2c-dev* y *Termios* y ofrecen operaciones con un mayor nivel de abstracción. Por ejemplo, este componente se encarga de gestionar el alineamiento de los bits y de llamar a los servicios del SO que precise una simple transmisión de datos. Al mismo nivel se encuentra la librería **`PiGPIO`** —presentada en la sección 5.2.1— que ofrece operaciones para el control de pines GPIO incluyendo soporte para la variante bit-bang de PWM.

En un nivel superior se encuentra el componente **`Equipment_Handlers`** o Gestor de Dispositivos que, como su nombre indica, brinda acceso a los dispositivos conectados a los buses y pines GPIO del sistema. Este componente ofrece un conjunto de interfaces de más alto nivel para la lectura de los sensores digitales como IMU, GPS o ADCs, el manejo de los interruptores MOSFET y el control de los calefactores mediante PWM. Este componente actúa como fachada o *proxy* ya que oculta a los clientes las operaciones y protocolos de comunicación de los dispositivos. Nótese que las operaciones relativas a la IMU (denominadas `IMU_Operations`) se ofrecen a otros componentes externos para evitar el problema de *sumidero* que se produce por la sobrecarga de llamadas entre capas.

Finalmente, en la parte superior se encuentra el componente **`Board_Support`** que brinda operaciones a nivel de tarjeta electrónica. Este componente organiza internamente todos

los equipos pertenecientes a un mismo subsistema de forma estática. Por ejemplo, la interfaz `PCUOperations` comprende operaciones para el acceso a un sensor de temperatura, uno de potencia, y una serie de interruptores y controladores PWM.

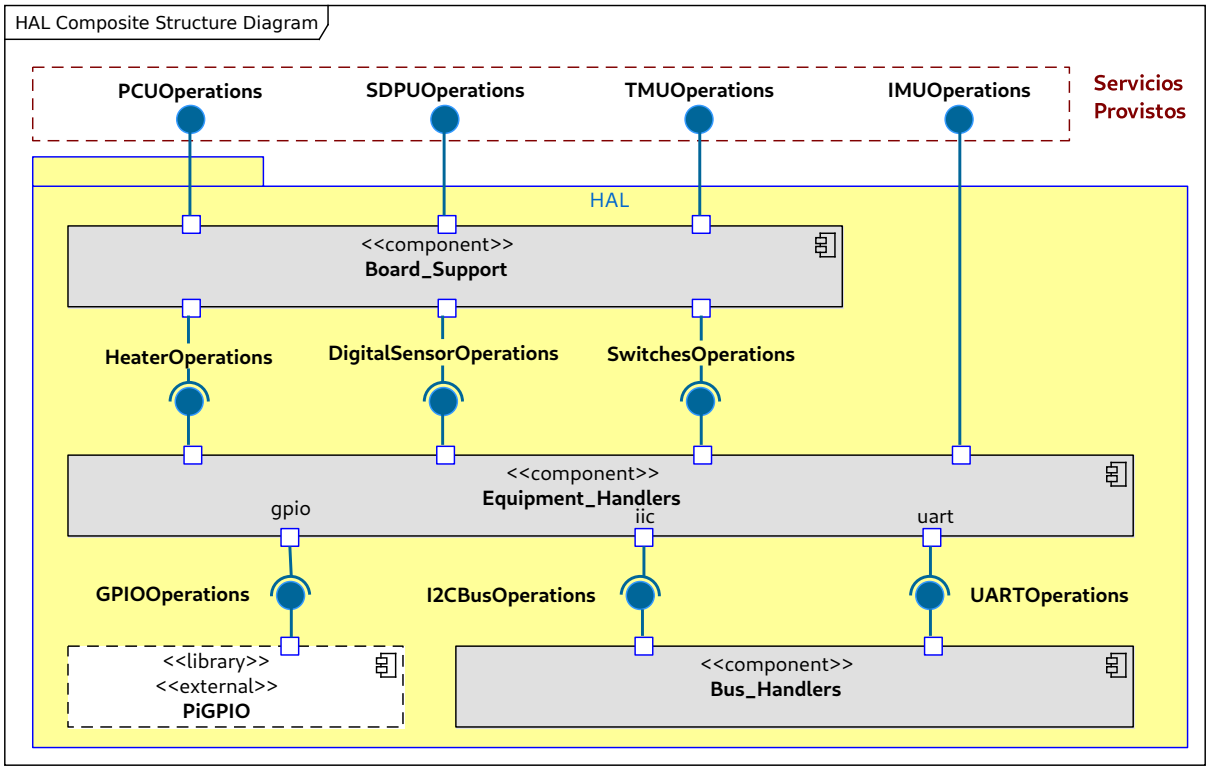


Figura 8.3: Diagrama de estructuras compuestas UML de la HAL.

8.2.1.2. Arquitectura Dinámica

Los componentes de este sistema son pasivos, es decir, todas las operaciones se ejecutan en el contexto de la tarea llamadora. Asimismo, estas operaciones no tienen asociada ninguna protección o mecanismo de sincronización para el acceso a los buses compartidos. Esta es una decisión de diseño que fue tomada para evitar el acceso a recursos protegidos (o *mutex* en *nix) anidados de forma involuntaria. Asimismo, el análisis del sistema será menos tedioso si la asignación de roles a los elementos de tiempo real (i.e.: tareas y recursos) se restringe a un menor número de componentes.

8.2.2. Componente Data_Pool

8.2.2.1. Arquitectura Estática

La figura 8.4 presenta la descomposición del componente Almacén de Datos en cinco objetos protegidos. Cada uno, representa los datos capturados por cada uno de los subsistemas y solo se almacena el valor más reciente, es decir, siguen el comportamiento *blackboard*. Estos datos se modifican a través de las operaciones de la interfaz `Modify_DP` y se leen mediante la interfaz `Read_DP`.

Las flechas punteadas representan la delegación de las PIs del componente en sus objetos estáticos definidos internamente. En la parte derecha se muestran el nombre (no la signatura completa) de las operaciones que conforman cada interfaz.

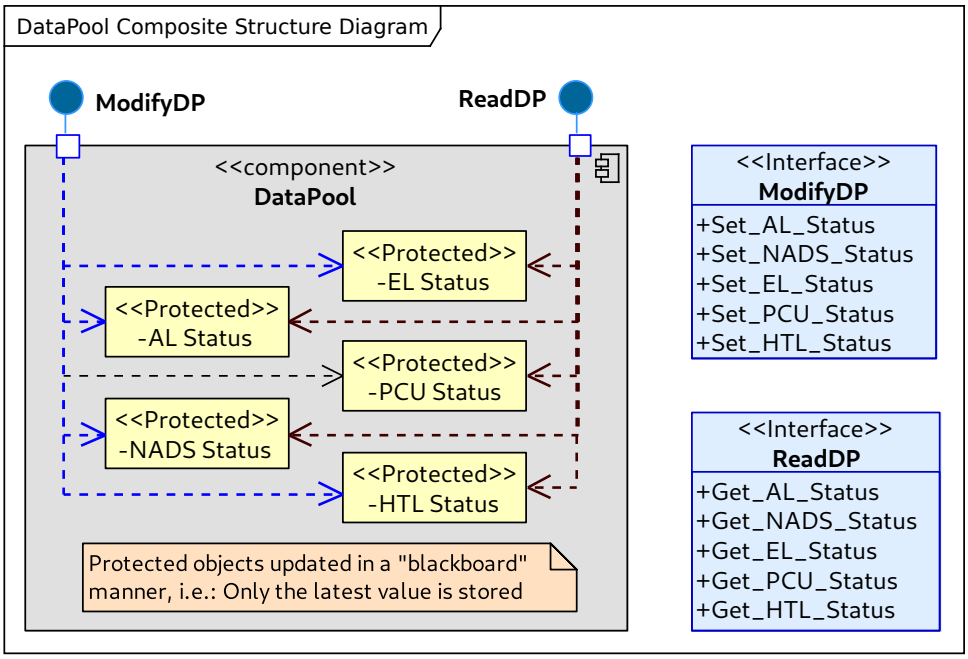


Figura 8.4: Diagrama de estructuras compuestas UML de la Data Pool.

El tipo de dato de estos cinco objetos protegidos tiene una estructura dependiente al subsistema. Por ejemplo, el objeto `PCU_Status`, que es modificado por el subsistema PCU, contiene datos de potencia, tensión, intensidad, temperatura e interruptores (*switches*). En cambio el objeto *EL_Status* comprende los datos de los sensores de radiación, presión, entre otros. A pesar de estas diferencias todos estos objetos comparten campos comunes, véase el requisito funcional O-FU-13. En consecuencia, se ha definido una plantilla genérica para el tipo de dato almacenado por estos objetos (8.1).

Tabla 8.1: Estructura genérica para los datos almacenados en la `Data_Pool`

Campos	Descripción
Snapshot_Time	Tiempo absoluto (tomado del GPS) en que se capturó el dato útil.
Mission_Time	Tiempo relativo desde el inició de la misión en que se capturó el dato útil.
Mode	Contiene el modo operativo del subsistema en el instante en que se capturó el dato útil. Es decir, el tipo de dato de este campo depende de cada subsistema, luego, tiene asociado un tipo genérico.
Payload	A diferencia de los campos anteriores, este no contiene meta-datos, sino el dato útil. Su contenido depende del subsistema, por tanto, se trata de un campo de tipo genérico.

8.2.2.2. Arquitectura Dinámica

En cuanto a la arquitectura dinámica, este componente incluye siete objetos protegidos e independientes que aseguran la exclusión mutua en accesos concurrentes. La división en cinco objetos tiene un nivel de granularidad adecuado ya que, por ejemplo, si solo se emplea un único objeto protegido se generarían bloqueos innecesarios entre tareas que no guardan ninguna relación entre sí. Por el contrario, si se emplean más objetos protegidos para los diversos campos de los estados, se provocaría una sobrecarga absurda.

8.2.3. Componente Manager

8.2.3.1. Arquitectura Estática

El componente `Manager` está subdividido en cuatro componentes software como se puede apreciar en la figura 8.5. Este ofrece la interfaz `Notify_Event` que permite a otros subsistemas notificar los eventos que hayan detectado, como por ejemplo: los cambios en la presión absoluta, fallos con algún sensor, lecturas anómalas, entre otros. Por otra parte, la interfaz `Process TC` permite que este componente ejecute los TCs. Para llevar a cabo estos servicios, el `Manager` debe acceder a gran parte de las interfaces ofrecidas por otros componentes. Por ejemplo, los TCs que permiten controlar los calefactores del HTL y EL, obligan a que este componente posea la RI `HTL_Heater_Operations` y `EL_Heater_Operations`. Se puede observar que el componente `BalloonMode` contiene y gestiona el estado actual del sistema, por eso, es accedido tanto por el gestor de eventos como de TCs.

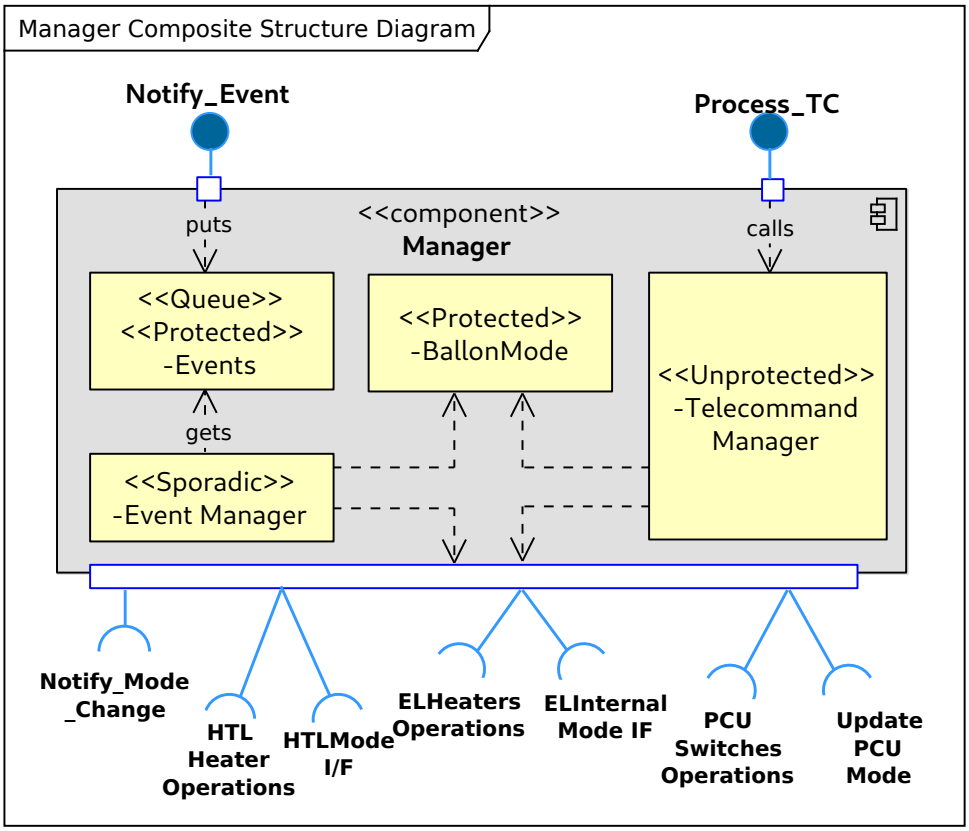


Figura 8.5: Diagrama de estructuras compuestas UML del Manager.

8.2.3.2. Arquitectura Dinámica

En cuanto a la arquitectura dinámica, el sistema está compuesto por una hebra y dos objetos protegidos. Concretamente, el componente `Event_Manager` tiene un MIAT de un segundo y está a la espera de los eventos ubicados en la cola `Events`. Por otro lado, el `Telecommand_Manager` se trata de un componente no protegido que se ejecuta en el contexto del llamador. Tanto el `Event_Manager` como el `Telecommand_Manager` acceden al objeto protegido `Balloon_Mode` que se encarga de procesar los eventos que puedan generar una transición de estado, es decir, `Balloon_Mode` implementa y protege la máquina de estados. La elección de un objeto protegido para la gestión de los eventos permite el acceso concurrente seguro entre las dos tareas.

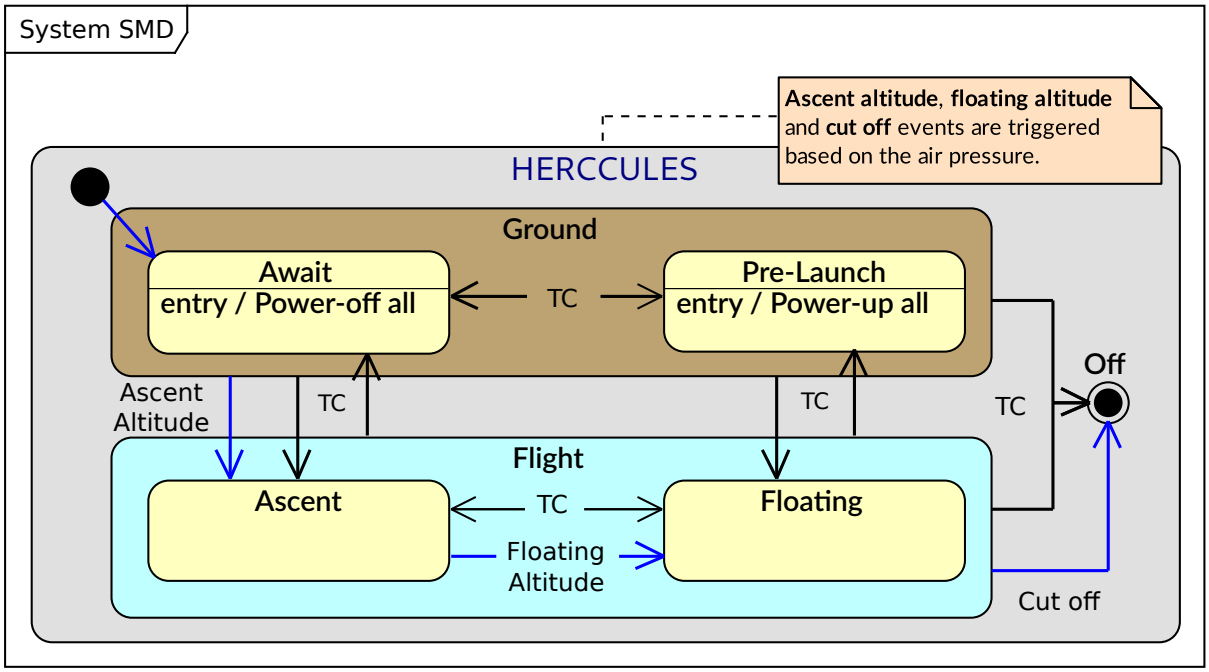


Figura 8.6: Diagrama de estados UML de los modos gestionados por el `Manager`.

Los modos de operación gestionados por el componente `Event_Manager` están fuertemente relacionados con las fases de la campaña y el CONOPS de HERCCULES (sección 7.1). Cabe recalcar que la gestión de los eventos orquestados por este componente, se ha definido en base a la arquitectura software de la plataforma FLP [79] y del microsatélite UPMSat-2 [90] La figura 8.6 muestra el diagrama de estados UML que implementa este gestor. Las flechas de color azul representan las transiciones activadas de forma automática por otros subsistemas a través de eventos, mientras que las flechas de color negro representan transiciones activadas por los operadores a través de TCs. En consecuencia, toda transición automática se genera por los eventos recibidos en la PI `Notify_Event`, y toda transición manual se genera por los TCs recibidos en la PI `Process_TC`.

En el nivel más alto se observan que el sistema puede estar bien en tierra o en vuelo. Asimismo, dentro de estos modos, el sistema podrá estar en distintos submodos. Por ello, los modos de vuelo y tierra se modelan como estados compuestos. Las acciones realizadas por este componente durante las transiciones y la entrada a nuevos estados se resume a continuación:

- **Modo Tierra (`Ground`)**: Este es el primer estado del software del sistema HERCCULES que simboliza la posición física del experimento (en tierra). Inicialmente, el sistema está sin fuente de alimentación y no ejecuta ningún programa. Cuando el sistema se enciende se produce una transición de estado al modo Espera (`Await`).
  - **Submodo de Espera (`Await`)**: En este modo, el sistema cuenta está conectado a una fuente de alimentación, luego se carga el OBSW de forma automática e inicia su ejecución. Durante este modo todos los experimentos se encuentran sin alimentación, por tanto, no se registra ningún dato ni se envía TM. Sin embargo, el OBSW está conectado con la GS y a la escucha de TCs.
  - **Submodo de Pre-lanzamiento (`Pre-Launch`)**: Este modo enciende todos los subsistemas, de esta forma, se permite a los operadores verificar el estado de los dispositivos. En síntesis, este modo se corresponde con la fase de Pre-Lanzamiento especificada en el CONOPS.
- **Modo de Vuelo (`Flight`)**: Se trata de un modo al cual se transita (i) de forma automática cuando se haya alcanzado la altitud de ascenso (generado por el evento *Ascent_Altitude*), o (ii) de forma manual mediante el envío de un TC. Asimismo, durante este modo se puede transitar al estado de Apagado u `Off` de forma automática cuando se detecte el evento de descenso o `Cut_Off`. Si algún subsistema está apagado, el OBSW lo enciende.
  - **Submodo de Ascenso (`Ascent`)**: Cuando el sistema está en este modo, el HTL empieza a disipar potencia constante y el EL se activa, dando comienzo a la adquisición de sus datos. Una vez alcanzada la actitud de flotación (basada en la presión absoluta) se produce una transición al modo Flotante. Asimismo, esta transición puede ser activada en cualquier momento mediante TCs.
  - **Submodo de Flote (`Floating`)**: Este modo solo afecta al comportamiento de los controladores del HTL que se explican más adelante en su sección dedicada.
- **Modo Apagado (`Off`)**: Este modo es alcanzable desde cualquier otro, ya sea de forma automática cuando se detecte el descenso o de forma manual cuando el operador envíe un TC específico. Al entrar en este modo, se cortan las fuentes de alimentación, por tanto, la adquisición y registro de los datos se detiene.

Como se ha visto en la arquitectura de alto nivel (figura 8.2) y la arquitectura estática de este componente (figura 8.5), el gestor de eventos mantiene una relación bidireccional con los componentes del paquete **`Subsystems`**, que encapsula los cuatro experimentos del proyecto (detallado en la sección 8.2.6). La relación entre todos estos componentes se presenta en el diagrama de comunicación de la figura 8.7. Se pueden identificar (i) los componentes activos (es decir, que cuentan con su propio hilo de ejecución) en recuadros dobles, (ii) los componentes pasivos en recuadros simples, (iii) los eventos como flechas azules y (iv) los mensajes síncronos como flechas negras. En dicho diagrama se identifican los siguientes escenarios:

(1) Un cambio en el modo del sistema afecta al comportamiento de todos los demás subsistemas. Por lo tanto, dicho cambio es notificado a todos los subsistemas a través de la RI `Notify_Mode_Change`. Cabe destacar que los subsistemas `EL` y `ATL` están orquestados por **`SDPU_Manager`**, por eso, son notificados de forma indirecta.

(2) Los eventos detectados por los subsistemas que pueden provocar un cambio en el modo del sistema son notificados al **`Manager`** a través de la interfaz `Notify_Event`. Estos eventos son enviados de forma asíncrona y se alojan en la cola FIFO *Events*. A la llegada de un evento, el objeto esporádico **`Event_Manager`** se activa y lo procesa. Si dicho procesamiento implica un cambio de modo vuelve a ocurrir el escenario **(1)**.

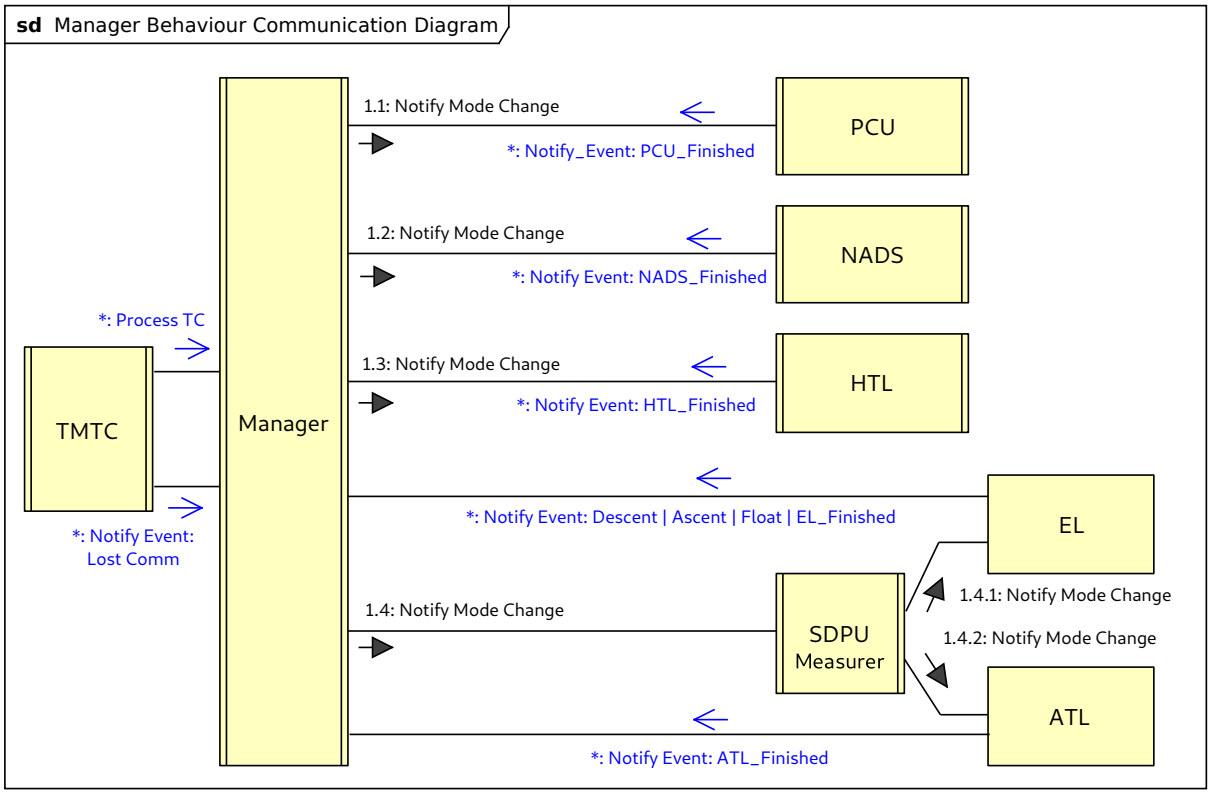


Figura 8.7: Diagrama de comunicación UML para la gestión de eventos orquestados por el `Manager`.

8.2.4. Componente Data_Storage

Los elementos internos del componente `Data_Storage` y las interfaces que implementa se ilustran en la figura 8.8. Por un lado la interfaz **`CSV_Operations`** incluye operaciones para la creación de ficheros CSV, el registro de entradas, grabar el fichero CSV en memoria persistente. Adicionalmente, este componente incluye la clase **`String`** que implementa una cadena de caracteres restringiendo el uso memoria dinámica, y manteniendo un atributo con el tamaño de dicha cadena (equivalente a los *Strings* en Ada o los *Slices* en Rust). Por otro lado, la interfaz `Images`, inspirado en el atributo `'Image` de Ada, consiste de una sola operación genérica que permite convertir al tipo de dato String (antes mencionado) los variables enteros (de 8, 16 o 32 bits) y de coma flotante (de doble o simple precisión).

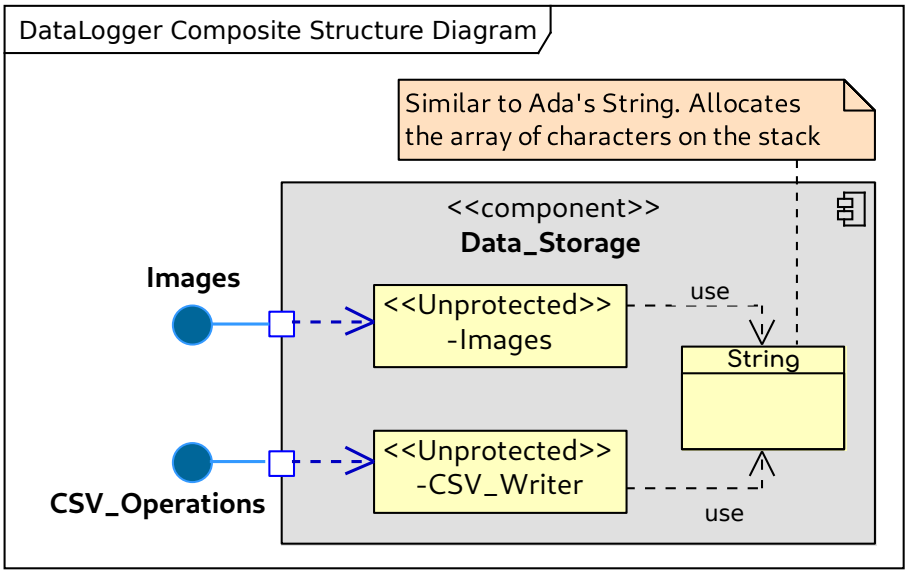


Figura 8.8: Diagrama de estructuras compuestas UML del Data Storage.

8.2.5. Componente (TT&C)

Arquitectura Estática

Este componente está dividido funcionalmente en cinco objetos como se aprecia en la figura 8.9.

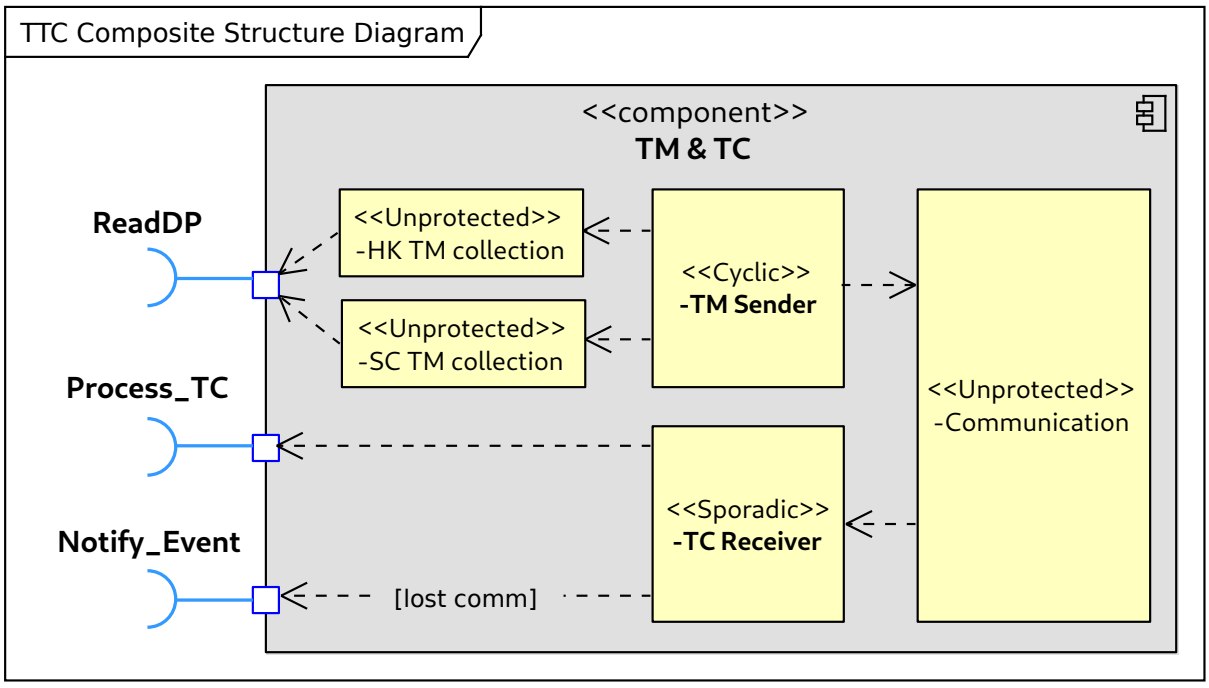


Figura 8.9: Diagrama de estructuras compuestas UML del `TT&C`.

Por un lado, el objeto **`TM_Sender`** se encarga de la recogida de la TM para su envío a la estación de tierra. La lectura de la TM se realiza accediendo a `Read_DP` que TM incluye

datos científicos (datos leídos de los instrumentos) y de HK (metadatos de los equipos del experimento). Por ello, **`TM_Sender`** descompone su funcionalidad en dos objetos dependiendo del tipo de TM producido. Las frecuencias de envío de esta TM están determinadas por los requisitos del OBSW (sección 7.5) y se han sintetizado en la tabla 8.2. En la primera columna se presenta la categoría de la TM, las variables específicas se muestran en la segunda columna, la tercera columna contiene el periodo de transmisión y la última contiene el tamaño de la variable en bits.

Tabla 8.2: Categorías de la TM enviada a la GS.

Categoría	Variable	Periodo [s]	Tamaño [b]
HK **ID:** 0×00	Temperaturas del ATL.	10	16×2
	Temperaturas de la PCU.	10	32
	Corriente y voltaje de la batería	10	32×2
Científicos **ID:** 0×01	Temperaturas de las placas y aire del HTL.	1	16×8
	Potencias disipadas por el HTL.	1	16×4
	Piranómetros y pirgeómetros del UEL y DEL	1	16×4
	Anemómetro del EL	1	16×4
	Barómetros del EL	1	32×2
	IMU del NADS	1	32×9
	GPS del NADS	1	32×4
	Sensor de Nadir del ATL	1	32×4

En cuanto al objeto **`TC_Receiver`**, este lleva a cabo la recepción de los TC y accede a las RIs `Procces_TC` y `Notify_Event`, ambas implementadas por el Gestor o **`Manager`**. Estas interfaces le permiten redirigir la ejecución de los TCs y notificar los eventos de desconexión (evento `lost_comm`) al Gestor del sistema. En base a los requisitos de operación (sección 7.5.4), se han creado cuatro categorías de TCs que se resumen en la tabla 8.3.

Tabla 8.3: Categorías de los TCs recibidos de la GS.

ID	Nombre	Parámetro 1	Parámetro 2
0×00	Cambio de Modo	Nuevo modo	–
0×01	Empezar Control Manual	ID del calefactor	–
0×02	Detener Control Manual	ID del calefactor	–
0×03	Controlar Calefactor	ID del calefactor	Potencia a disipar
0×04	Reiniciar dispositivo I2C	ID del dispositivo	–

Finalmente, el objeto **`Communication`** abstrae las operaciones de envío y recepción de los datos. Esta comunicación con el GS se realiza empleando (i) el protocolo TCP/IP para la recepción de los TCs debido a que ofrece una comportamiento orientada a conexión donde se respeta el orden de envío y (ii) el protocolo UDP para el envío de la TM ya que se trata de una conexión orientada al flujo de los datos.

Arquitectura Dinámica

En cuanto a la arquitectura dinámica, este componente cuenta con dos tareas, una cíclica y otra esporádica. El envío de la TM científica se realiza de forma periódica con una frecuencia de 1 Hz, por ello, el objeto **`TM_Sender`** tiene asociado una tarea cíclica con la misma frecuencia de activación. Sin embargo, la TM de HK se debe enviar con un periodo máximo de 10 segundos (0.1 Hz). Para cumplir con dicho requisito, esta tarea cíclica hace la función de divisor de frecuencia (como un *prescaler*) e invoca al objeto **`HK_TM_collection`** cada diez activaciones. Por otra parte, **`TC_Receiver`**, este tiene asignado una tarea esporádica ya que solo ejecuta en la recepción de los recibidos mediante invocaciones bloqueantes a la llamada del sistema `recv`.

El comportamiento de este subsistema es dependiente del estado de la conexión con la GS. Esto se ve reflejado en el diagrama de estados en la figura 8.10, donde se aprecian los dos modos de operación. Inicialmente este componente estará en el modo `Connected` donde enviará periódicamente la TM y estará a la escucha de los TCs. Este estado se abandona cuando el objeto **`Communication`** detecta la desconexión a tierra, suceso que da paso al modo `Communicaction Lost`. La desconexión a tierra no solo afecta el comportamiento de este módulo, sino también a otros subsistemas que estén en modo automático a la escucha de los comandos enviados por los operadores. En consecuencia, el objeto receptor de TCs (**`TC_Receiver`**) notifica al **`Manager`** de dicha desconexión mediante el evento `Lost_Comm`. Finalmente, cuando la conexión se haya reestablecido, se vuelve al estado inicial.

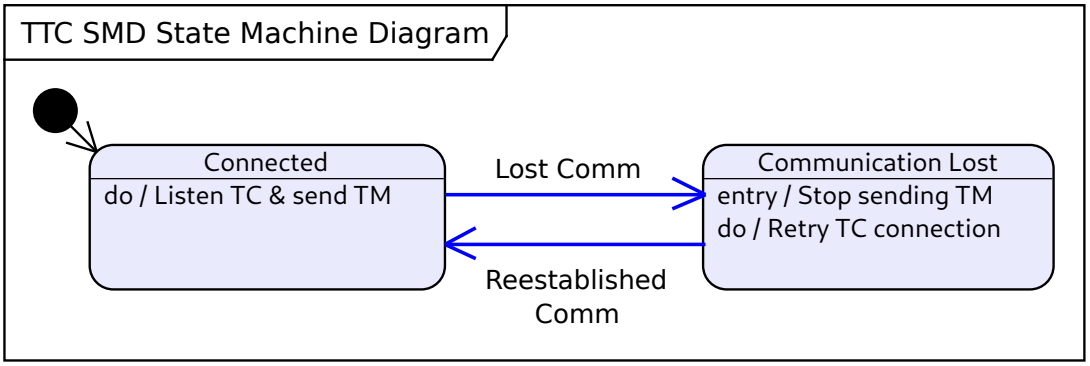


Figura 8.10: Diagrama de estados UML del `TT&C`.

8.2.6. Componente Subsystems

La figura 8.11 muestra la descomposición del paquete Subsistemas en siete componentes. Este sistema tiene un nivel de cohesión lógico ya que reúne componentes con poco grado de interacción. Esta relación también refleja la arquitectura del OBDH, pues los subsistemas **`EL`**, **`HTL`** y **`SDPU_Measurer`** tienen mayor grado de cohesión debido a que los subsistemas que gestionan, EL y ATL, comparten el mismo bus I2C y ADC embebidos en la tarjeta SDPU.

En aras de la brevedad y a diferencia de las secciones previas, la descripción de los componentes internos a este paquete incluye tanto el comportamiento dinámico como estático. Esta decisión también está fundamentada en que el diseño de la arquitectura de estos componentes ha estado supeditado a las características dinámicas como el acceso a los recursos compartidos.

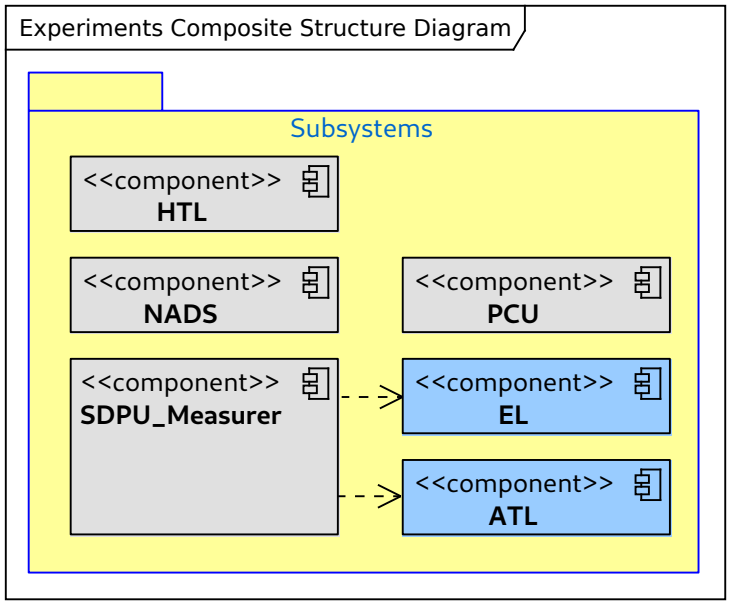


Figura 8.11: Diagrama de estructuras compuestas UML de los Subsistemas.

8.2.6.1. Componente HTL

En primer lugar, el **`HTL`** es el componente encargado de las funciones del laboratorio homónimo (es decir, el HTL). Por ello, debe adquirir y almacenar las lecturas de los termistores, los estados de los calefactores, y proporcionar servicios que faciliten la tarea OBDH del OBSW como el envío de TM o ejecución de los TCs. Como se muestra en la figura 8.12, este componente se ha subdividido en cuatro objetos sobre los que se distribuyen todas estas funcionalidades.

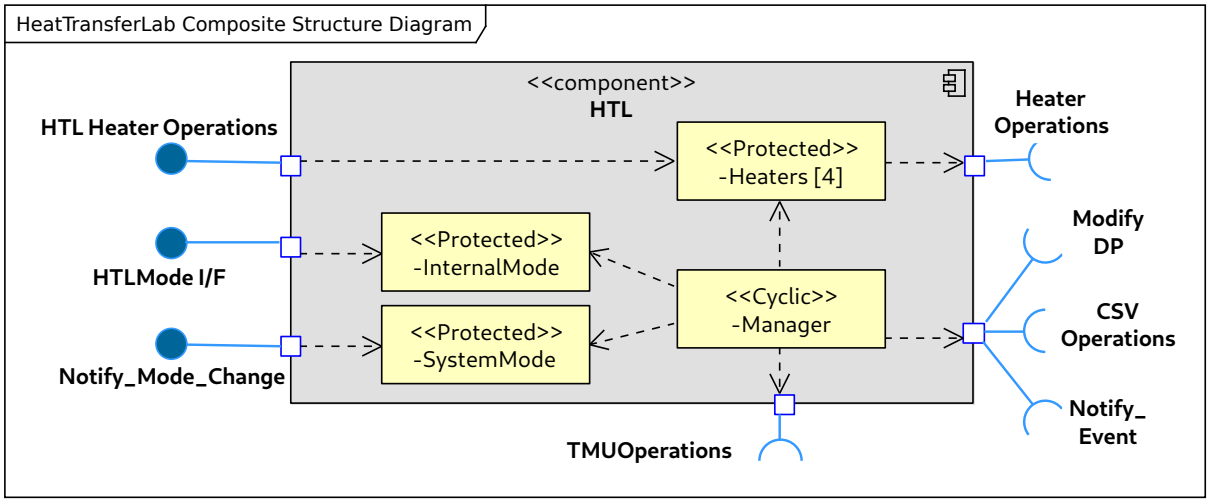


Figura 8.12: Diagrama de estructuras compuestas UML del HTL.

Por un lado, este componente cuenta con el objeto **`Manager`** que actúa como un gestor interno de las actividades de monitorización de los termistores, control de los calefactores según el modo de operación del sistema y subsistema actual, registro de los estados de los calefactores y de las medidas de los termistores en memoria persistente (mediante `CSV_Operations`) y en el almacén de datos o **`Data_Pool`** del OBSW (mediante `ModifyDP`). Todas estas actividades

se realizan de forma cíclica con un periodo de diez segundos (requisito O-RE-03, tabla 7.2).

Las notificaciones recibidas por la RI `Notify_Mode_Change` se delegan al objeto protegido **`System_Mode`** que actúa como pizarra (*blackboard*) es decir, conserva el valor más reciente en memoria no volátil. Se trata de un componente protegido que asegura la exclusión mutua entre el gestor interno (objeto **`Manager`**) y el gestor de modos global del OBSW, también denominado **`Manager`** (sección 8.2.3). Por otro lado, el objeto protegido ***Heaters*** realiza todas las funciones relacionadas con el control y monitorización del estado de los factores. Solo se encarga de los aspectos de concurrencia ya que las operaciones de bajo nivel las delega al **`hal`** mediante la RI *Heater_Operations*.

Finalmente, el objeto protegido **`Internal_Mode`** implementa la máquina de estados interna al sistema HTL (figura 8.13). Dichos estados se han definido en base a las especificaciones del laboratorio HTL y definen la potencia disipada por los calefactores en las fases de la misión.

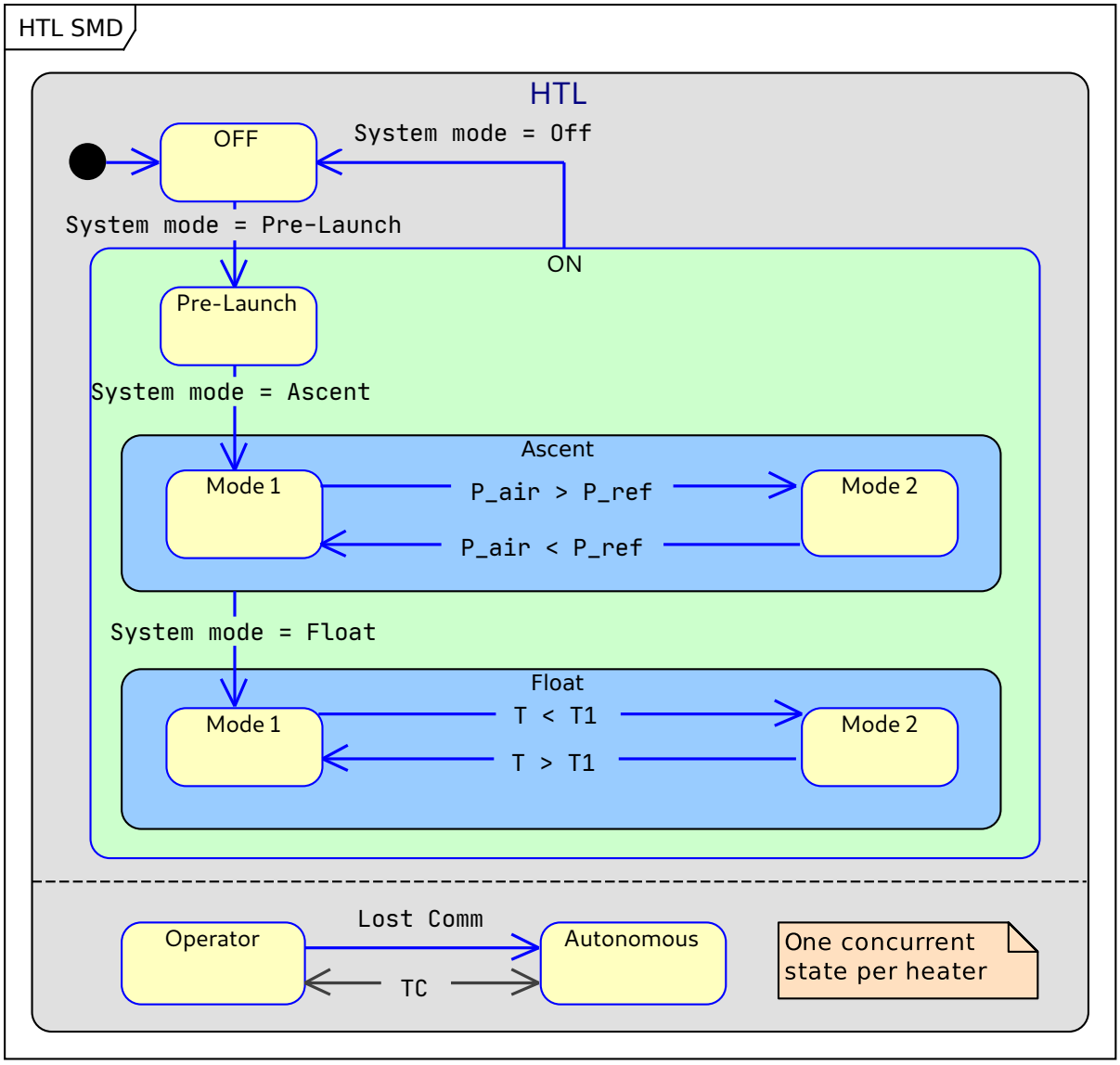


Figura 8.13: Diagrama de estados UML del HTL.

También se aprecian dos estados concurrentes: el primero (*Operator*) permite que el operador pueda controlar los calefactores mediante TCs y el modo autónomo (*Autonomous*) hace que los calefactores solo puedan ser controlados por este componente. Al igual que el objeto **`System_Mode`**, lo hace en forma de pizarra y asegura la exclusión mutua entre el gestor global e interno a este componente.

8.2.6.2. Componente PCU

Como se ilustra en la figura 8.14, este componente sigue una estructura similar al HTL. El objetos protegido **`System_Mode`** es una instancia del mismo objeto presentado anteriormente, por tanto, tiene el mismo comportamiento. Por el contrario, el objeto **`Manager`** ejecuta periódicamente cada cinco segundos, según las especificaciones del sistema, que monitoriza el estado de los interruptores (*switches*) y de los sensores digitales de potencia, voltaje, corriente y temperatura. El comportamiento de dicho **`Manager`** depende del estado del sistema, por ello, debe leerlo en todas las activaciones.

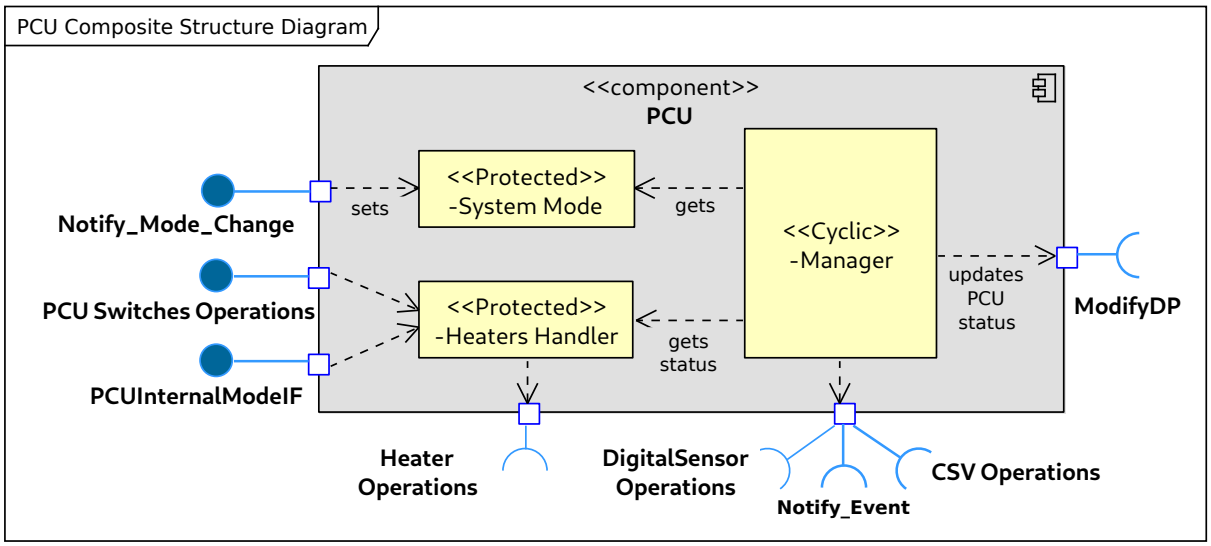


Figura 8.14: Diagrama de estructuras compuestas UML del PCU.

8.2.6.3. Componente NADS

Este componente esta conformado por dos elementos como se puede observar en la figura 8.15. Cuenta con un objeto periódico denominado **`Measurer`** que se activa con un periodo de un segundo. En cada activación lee el modo de la misión, alojado en el objeto protegido **`System_Mode`**, ya que su comportamiento depende de este.

En los modos de ascenso, flote, y prelanzamiento, adquiere los datos de la IMU y el sensor GPS a través de la RI `DigitalSensorOperations`. Las lecturas obtenidas se almacenan tanto en memoria persistente como no persistente mediante las RIs `CSV_Operations` y `Notify_Event`. En cualquier otro modo de operación, este objeto no realiza ninguna actividad. El diagrama de estados de la figura 8.16 muestra gráficamente este comportamiento.

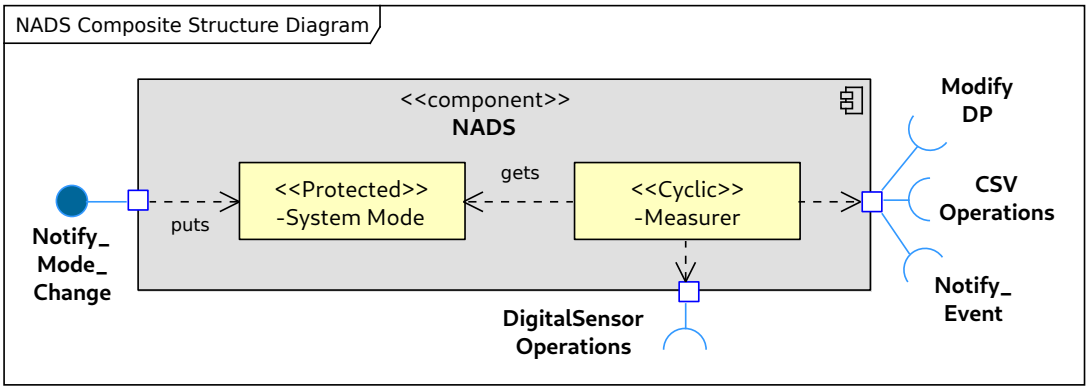


Figura 8.15: Diagrama de estructuras compuestas UML del NADS.

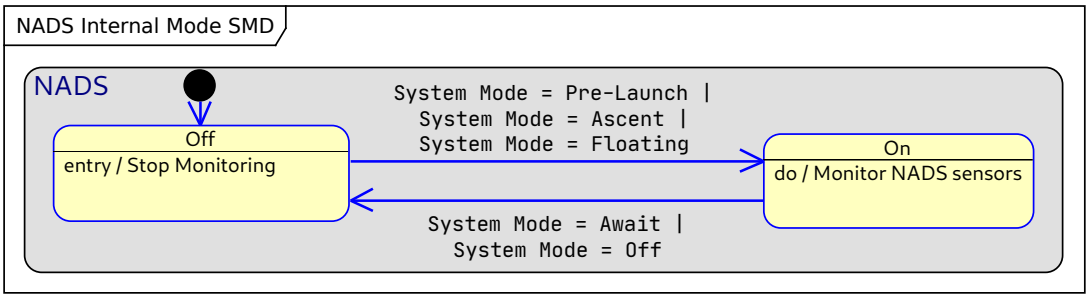


Figura 8.16: Diagrama de estados UML del NADS.

8.2.6.4. Componente SDPU_Measurer y sus Delegados

La figura 8.17 muestra el diagrama de estructuras compuestas, donde se pueden identificar todos los componentes, objetos y las relaciones de delegación y composición.

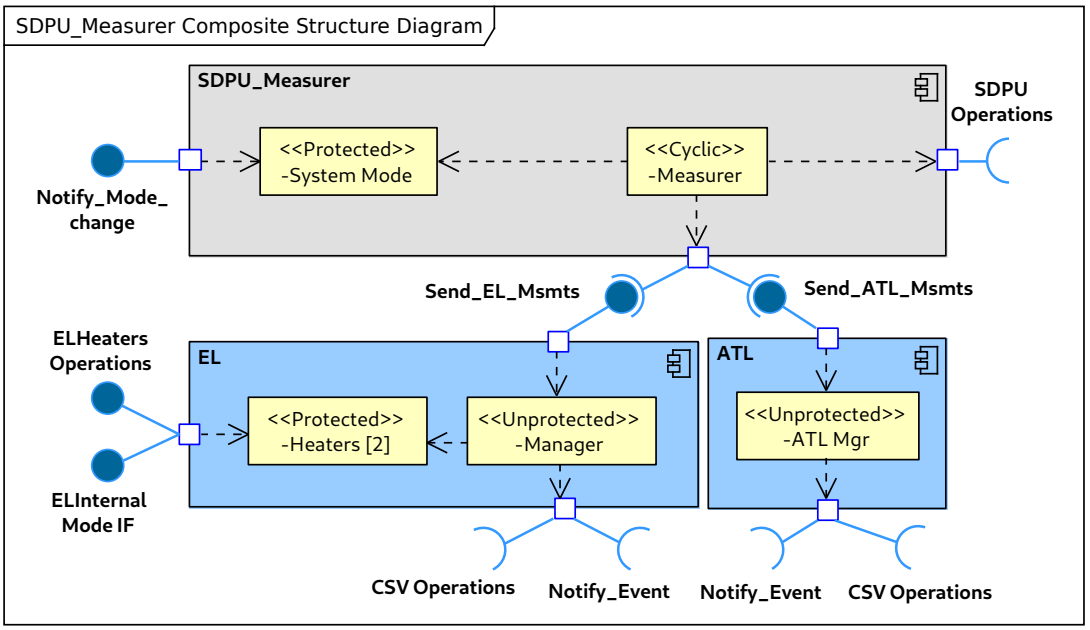


Figura 8.17: Diagrama de estructuras compuestas UML del SDPU, EL ATL.

Este componente sigue la misma estructura que el NADS, la diferencia reside en que el objeto medidor (`Measurer`) delega al EL y ATL las operaciones sobre los datos que se haya obtenido. Por un lado el **`ATL`**, recibe estas medidas y las almacena en memoria persistente en el formato y con los campos adecuados. El componente **`EL`** tiene un comportamiento similar, pero también cuenta con calefactores que debe monitorizar y controlar periódicamente. Estas operaciones se delegan al objeto protegido **`Heater`** que cumplen una función similar a los `Heater_Handler` de la PCU.

8.3. Arquitectura del componente GSW

La figura 8.18 muestra la arquitectura del componente GSW, donde también se puede apreciar la relación que existe con el OBSW. La estructura interna de este componente está basada en el patrón de diseño Modelo-Vista-Presentador (MVP). Este patrón está dirigido a las aplicaciones enfocadas en proporcionar interfaces gráficas para el usuario. Presenta varios beneficios, de entre los que destacan la separación e independencia entre la lógica de negocio, los estilos de presentación visual, y la consistencia entre ambas.

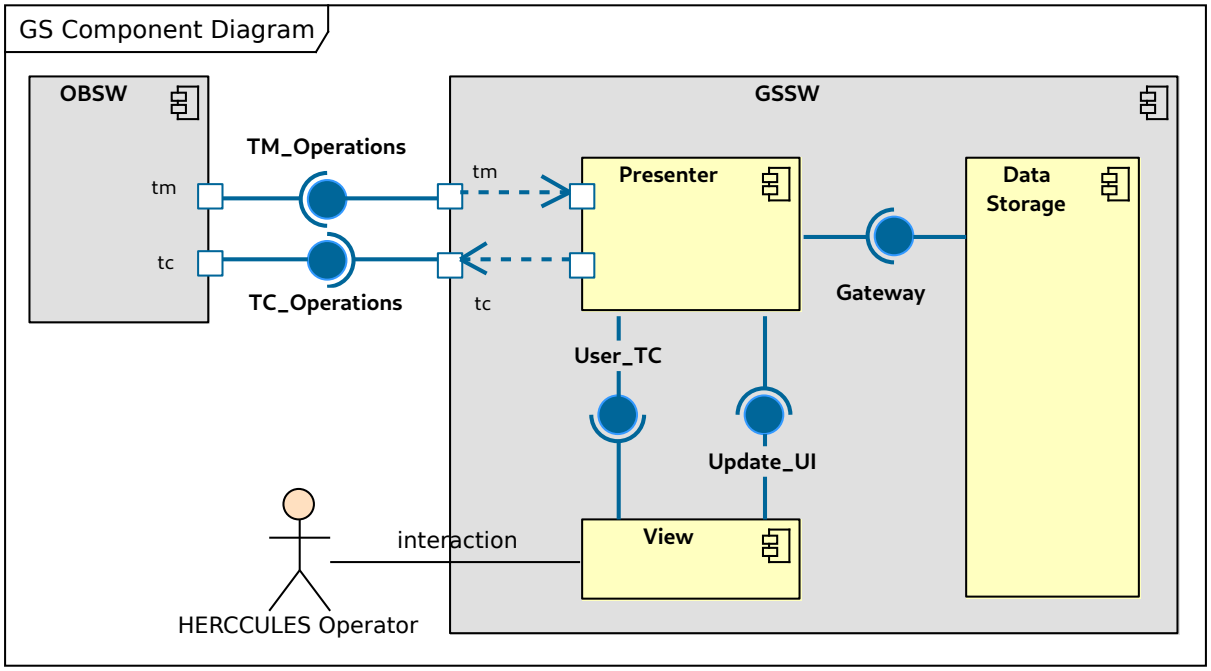


Figura 8.18: Diagrama de componentes UML para la GSW.

Los componentes que conforman el GSW se describen a continuación:

- **Presentador (`Presenter`)**: Este componente recibe periódicamente la TM a través de su PI `TM_Operations`, y los transforma en un formato adecuado para su procesamiento. Asimismo, se encarga de actualizar la interfaz del usuario invocando los servicios `Update_UI` del componente **`View`** y los guarda en memoria persistente a través de la interfaz `Gateway`. Este componente también se encarga del envío de los TCs al OBSW invocando a las funciones y métodos de la RI `TC_Operations`. El envío de los TCs se realiza a petición de los clientes mediante su PI `User_TC`.

- **Vista (`View`)**: Este componente implementa la presentación de la TM mediante el marco de desarrollo Qt. También se encuentra a la escucha de los eventos o acciones ejecutados por el usuario que son comunicados al **`Presentador`** para su gestión. La actualización de la vista se expone al resto de componentes a través de la interfaz `Update_UI`.
- **Almacén de Datos (`Data_Storage`)**: Este componente ofrece sus servicios a través de la interfaz `Gateway` cuyo propósito es brindar operaciones que permitan almacenar los datos en memoria persistente y recuperarlos, si fuera necesario.

8.4. Diseño de Bajo Nivel

El diseño de bajo nivel es un termino subjetivo. En esta tesis de máster, se considera como diseño de bajo nivel el resultado de transformar los componentes, paquetes y objetos en los mecanismos para la codificación que ofrece el lenguaje, sea de programación o de modelado. La capa de más bajo nivel se ha implementado como un proyecto en C++ independiente a las herramientas TASTE ya que no requiere de ninguna abstracción de tiempo real. En este caso, el diseño de bajo nivel se corresponde con las tipos de datos, módulos funcionales (*namespaces*, *structs* y *clases*), así como las relaciones que mantengan (agregación, composición, dependencia y herencia). En cambio, el resto de paquetes se han implementado con la herramienta TASTE. Por consiguiente, el diseño de bajo nivel en esos casos se corresponde con el diseño de las cuatro vistas del proceso ASSERT introducidos en la sección 4.2.1 (datos, interfaces y concurrencia). En ambos caso, el código fuente de HERCCULES se distribuye bajo la licencia *GNU GPLv3* y se encuentra accesible desde el siguiente enlace [75]:

`https://gitlab.com/AngelPerezM/herccules`

8.4.1. Diseño del Paquete HAL

Este paquete se ha desarrollado como un componente independiente en C++ con el soporte de la herramienta CMake para la gestión de los módulo y sus dependencias. La traducción del diseño de alto a bajo nivel sigue la siguiente se realizó en base a los siguientes principios:

- Un componente software se traduce a una librería estática.
- El conjunto de PIs se corresponde con una serie de funciones, métodos o subprogramas estáticos encapsulados en un espacio de nombres (*namespace* en C++) y exportados en el/los ficheros de cabecera (terminados en *.h*).
- El par PI-RI se corresponde con la llamada de un cliente a una función, método o subprograma implementado de forma externa.

Por tanto el concepto de interfaz como conjunto de funciones virtuales no es aplicable en este caso. Esta decisión se tomó por motivos de simplicidad y rendimiento, ya que el excesivo uso de llamadas a funciones virtuales tiene un comportamiento temporal imprevisible y puede generar sobrecargas indeseadas.

Los componentes software `Board_Support`, `Equipment_Handlers` y `Bus_Handlers`)

se han configurado y empaquetaron en sendas librerías estáticas respetando una estructura común:

- `CMakeLists.txt`: Directorio que contiene la configuración específica al componente software como los ficheros fuentes de compilación.
- `include`: Contiene los ficheros de cabecera terminados en `.h` que son expuestos a otros componentes (PIs).
- `src`: Contiene los ficheros de cabecera privados y los fuente terminados en *.cpp* que definen tanto las funciones declaradas en las cabeceras tanto públicas como privadas.
- `Demos`: Contiene ficheros fuente compilables a programas de demostración que permiten realizar la verificación por inspección.
- `Tests`: Contiene pruebas unitarias y de integración automatizadas. A diferencia de los programas de demostración verifican la validez de este componente de forma automática con casos de prueba distintos.

A modo de ejemplo, el extracto de código 8.1 ilustra el contenido del fichero de cabecera del componente **`TMU`** encargado de implementar todas la PI `PCUOperations`. Esta cabecera, por tanto contiene las declaraciones de las funciones especificadas en dicha PI, así como tipos de datos enumerado que ayudan a abstraer las operaciones de bajo nivel.

```
namespace board_support::tmu {
    enum PT1000_ID : uint8_t {
        PT1000_1,  PT1000_2,  PT1000_3,  PT1000_4,  PT1000_5,  PT1000_6,
        PT1000_7,  PT1000_8,  PT1000_9,  PT1000_10, PT1000_11, PT1000_12,
        PT1000_13, PT1000_14, PT1000_15, PT1000_16, PT1000_17, PT1000_18,
        PT1000_19, PT1000_20, PT1000_21, PT1000_22, PT1000_23, PT1000_24,
        PT1000_25, PT1000_26, PT1000_27, PT1000_28
    };

    enum MUXChannel : uint8_t {
        CH0 = 0U, CH1 = 1U, CH2 = 2U, CH3 = 3U,
        CH4 = 4U, CH5 = 5U, CH6 = 6U,
    };

    bool initialize();
    void finalize();

    int16_t readPT1000LineFrom(PT1000_ID pt1000ID);

    /**
    * @example if muxInput is CH0 you will select the first channel for all
    * multiplexers M1 to M4, which corresponds to the PT1000_1, PT1000_8,
    * PT1000_15, and PT1000_22 thermistors.
    * @return array with four readings, where the index i represents the
    * ith channel from the ADC.
    */
    std::array<int16_t, 4U> readAllADCChannels(MUXChannel muxInput);

    bool resetADC();

    /**
```

```
32      * @brief Transfer function for PT100s
33      */
34     float pt1000ToCelsius(int16_t raw);
35 }
```

Listado 8.1: Componente TMU escrito en C++.

La implementación de estas funciones requiere el uso de las operaciones ofrecidas por los objetos declarados en **`Equipment_Handlers`**. Estos objetos abstraen todos los dispositivos hardware empleados en HERCCULES y se han definido de forma estática en en el cuerpo del módulo **`Board_Support`**. La TMU, por ejemplo, cuenta con un ADC ADS5115, y un multiplexor para la selección de las líneas analógicas. En conjunto, estos dos dispositivos (digitales) posibilitan la lectura de los veintiocho termistores PT1000.

Como se ha visto en el listado 8.1, el componente TMU provee servicios de alto nivel que involucran la colaboración entre los objetos estáticos (líneas 10-11). El siguiente listado 8.2 muestra la declaración de estos objetos, así como la implementación del método de inicialización que los configura en un estado operativo. Se puede apreciar que dicha configuración, a su vez, necesita inicializar los componentes **`Bus_Handlers`** (líneas 19-22) y los objetos estáticos antes definidos (líneas 25-34).

```
1  #include "ADS1115_ADC.h"
2  #include "MUX.h"
3  // --snip--
4  namespace { // STATIC objects, flags and functions:
5      bs::ADS1115_ADC adc;
6      bs::MUX         mux;
7
8      bool initialized {false};
9      // --snip--
10 }
11 namespace board_support::tmu {
12     bool initialize() {
13         if (!bh::initialize()) {
14             return false;
15         }
16         if (!initialized) {
17             bool muxInitialized {
18                 mux.initialize(tmu::conf::muxPin0, tmu::conf::muxPin1,
19                                tmu::conf::muxPin2)
20             };
21             bool adcInitialized {
22                 adc.initialize(tmu::conf::i2cBusId, tmu::conf::i2cAddress,
23                                tmu::conf::adcMode)
24             };
25             initialized = adcInitialized && muxInitialized;
26         }
27         return initialized;
28     }
29     // -- snip --
```

Listado 8.2: Componente TMU escrito en C++.

El diseño del resto de paquetes siguen una estructura y proceso similar, con operaciones básicas para la incialización o finalización y otras más complejas que implican la coordinación entre varios objetos estáticos. El anexo C.1 incluye los listados de código fuente de los módulos más relevantes del paquete **`HAL`**.

8.4.2. Diseño del Paquete Data_Types

Esta paquete se corresponde con la DV en TASTE y, por lo tanto, está escrita en el lenguaje de especificación ASN.1. TASTE se encarga de la traducción de esta descripción formal en código fuente C y Ada. Como se ilustra en la figura 8.19, este paquete se ha organizado en cinco sub-paquetes en función de la relación entre los tipos de datos que define.

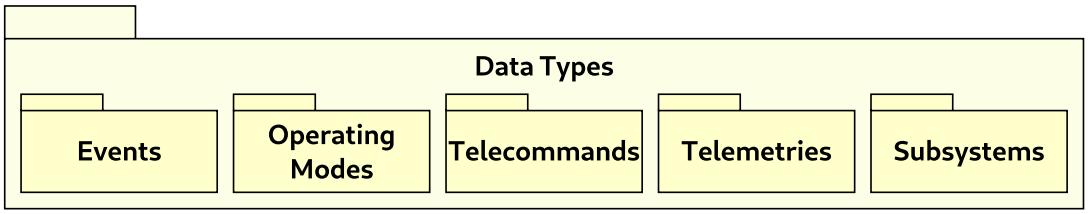


Figura 8.19: Organización de la DV en paquetes.

El paquete `DataTypes` es la raíz y por eso contiene una serie de tipos básicos que son empleados por todos los paquetes hijos y abarca desde tipo enteros hasta la representación del tiempo absoluto y relativo. Este paquete es el más estable ya que no depende de otros. El siguiente extracto de código 8.3 muestra la definición en ASN.1 del módulo raíz:

```
DataTypes DEFINITIONS AUTOMATIC TAGS ::= BEGIN
        Null-Type ::= SEQUENCE {} -- null as empty sequence
        Boolean-Type ::= BOOLEAN  -- renaming

        --  Integers  ------------------------------------------------------------
        UINT8-Type  ::= INTEGER (0 .. 255)
        UINT16-Type ::= INTEGER (0 .. 65535)
        INT8-Type   ::= INTEGER (-128 .. 127)
        INT16-Type  ::= INTEGER (-32768 .. 32767)
        // ..snip..

        --  Floating point types  ------------------------------------------------
        FLOAT32-Type ::= REAL (-3.4E+38 .. +3.4E+38)
        FLOAT64-Type ::= REAL (-1.7E+308 .. +1.7E+308)

        --  Strings  -------------------------------------------------------------
        OCTSTR-VARIABLE-LEN  ::= OCTET STRING (SIZE(0 .. 1024))
        CHARSTR-VARIABLE-LEN ::= IA5String (SIZE(0 .. 1024))

        --  Time types  ----------------------------------------------------------
        Relative-Time-Type ::= FLOAT32-Type
        Absolute-Time-Type ::= SEQUENCE { -- Secs and microSecs since the UNIX epoch
            secs  Epoch-Type, usecs Epoch-Type
        }
        Epoch-Type ::= INT32-Type

        --  BASIC Analogue & Digital Signals  ------------------------------------
        Analogue-Raw-Data ::= INT16-Type -- 16 bit ADC
        Heater-Power-Type ::= REAL (0.0 .. 2.0)
        Switch-Status ::= ENUMERATED { on, off }
END
```

Listado 8.3: Extracto del paquete DataTypes escrito en ASN.1.

Los paquetes `Events` y `OperatingModes` contienen un tipo de dato para los eventos y para los modos de operación del sistema y subsistemas, respectivamente. Se ha decidido separarlos de los tipos básicos ya que son susceptibles a sufrir cambios, por ejemplo, que se creen nuevos eventos para la gestión de errores o modos complementarios. Para la definición de estos tipos se emplearon los tipos enumerados (`ENUMERATED` en la sintaxis de ASN.1). Por razones de brevedad, se ha omitido la descripción de estos paquetes. Por otra parte, los paquetes de TMs y TCs, definen tipos de datos más complejos conformados, a su vez, por otros tipos definidos internamente de forma separada. De esta forma se evitan estructuras con niveles de anidamiento excesivos y permite la reusabilidad de tipos a un nivel más granular. Ambos tienen dependencia sobre el paquete `OperatingModes`, ya que estos se emplean como campos internos. A modo de ejemplo, el listado de código 8.4 ilustra un extracto de la definición formal de los TCs en ASN.1. Los tipos auxiliares se han omitido por razones de brevedad.

```
DataTypes-Telecommands DEFINITIONS AUTOMATIC TAGS ::= BEGIN
        IMPORTS Heater-Power-Type  FROM DataTypes
                Balloon-Mode       FROM DataTypes-OperatingModes;

        --  T E L E C O M A N D S  ------------------------------------------------

        TC-Type ::= CHOICE {
                change-balloon-mode        TC-Change-Balloon-Mode,
                start-manual-control       TC-Start-Manual-Control,
                stop-manual-control        TC-Stop-Manual-Control,
                control-experiment-heater  TC-Control-Experiment-Heater,
                restart-device             TC-Restart-Device
        }

        -----------------------------------
        -- TC-0: Change Balloon's Mode TC --
        -----------------------------------
        TC-Change-Balloon-Mode ::= SEQUENCE {
                new-mode  Balloon-Mode
        }

        ---------------------------------------------------------------------
        -- TC-1 & TC-2 & TC-3: Start & Stop & Control experiment heaters --
        ---------------------------------------------------------------------
        TC-Start-Manual-Control ::= SEQUENCE {
                heater  Heater-ID
        }

        TC-Stop-Manual-Control ::= SEQUENCE {
                heater  Heater-ID
        }

        TC-Control-Experiment-Heater ::= SEQUENCE {
                heater        Heater-ID,
                heater-power  Heater-Power-Type
        }

        ------------------------------
        -- TC-4: Restart Device TCs --
        ------------------------------
        TC-Restart-Device ::= SEQUENCE {
                device-id  Restartable-Device-ID
        }

        --  A U X I L I A R Y  ---------------------------------------------------
        // ..snip..
END
```

Listado 8.4: Extracto del paquete `Telecommands` escrito en ASN.1.

Se puede apreciar en las líneas 2 y 3 de dicho listado la dependencia de `Telecommands` con el paquete raíz y de modos operacionales. Asimismo, se emplea el tipo de dato `Choice` para definir los TCs, de esta forma, el OBSW y GSW dependen de un único tipo de dato que aúna todas las opciones de TCs disponibles. Por otro lado, el paquete `Telemetries` sigue una estructura similar con tipos de datos auxiliares que dan soporte a la creación de datos más complejos. En este caso, todas las TMs siguen en patrón común que se ha definido con una las plantillas ofrecidas soportadas por ASN.1, como se puede apreciar en el listado 8.5. Se observa que el tipo `TM-Template` contiene el parámetro genérico `Payload-Type` que define el tipo de dato del campo `payload` (línea 8). De esta forma, la creación del resto de TMs se ve simplificada y se centra en la parte significativa (el *payload*). Este listado también incluye la definición de la TM de HK en las líneas 15-20, donde se identifica el uso de la plantilla para la creación de una secuencia de datos más compleja.

```
1  ---------------------------
2  -- Template for TM types --
3  ---------------------------
4  TM-Template {Payload-Type} ::= SEQUENCE {
5          sequence-number  UINT32-Type,
6          timestamp        Epoch-Type,
7          baloon-mode      Balloon-Mode,
8          payload          Payload-Type
9  }
10
11 -----------------------------
12 -- TM-1: Housekeeping (HK) --
13 -----------------------------
14 HK-TM-Type ::= TM-Template {
15         SEQUENCE {
16                 atl-hk  ATL-HK-TM-Type,
17                 pcu-hk  PCU-HK-TM-Type
18         }
19 }
20
21 ATL-HK-TM-Type ::= SEQUENCE {
22         snapshot-time  Epoch-Type,
23         temperatures   SEQUENCE (SIZE(2)) OF Analogue-Raw-Data
24 }
25
26 PCU-HK-TM-Type ::= SEQUENCE {
27         snapshot-time  Epoch-Type,
28         temperature    INT8-Type,
29         v-bat          FLOAT32-Type,
30         a-bat          FLOAT32-Type
31 }
```

Listado 8.5: Extracto del paquete `Telemetries` escrito en ASN.1.

Por último, el paquete de `Subsystems` es el que aloja todas los tipos de datos generados por los subsistemas y que son publicados en la *Data Pool* del OBSW. De forma similar a las TM, este hace uso de plantillas que reflejan la estructura definida en la tabla 8.1. En este caso se precisan dos parámetros genéricos: `Mode-Type` que especifica el tipo del modo del subsistema y `Payload-Type` que especifica el tipo del contenido útil. Ambos tipos son específicos de cada subsistema y por cuestiones de brevedad solo se presenta el tipo definido para el subsistema ATL (líneas 11-21). **La definición de los paquetes restantes se encuentra detallada en el anexo C.2**.

```
-------------------------------------------------
-- Template for the creation of subsystem data --
-------------------------------------------------
Subsystem-Data-Template {Mode-Type, Payload-Type} ::= SEQUENCE {
        snapshot-time  Absolute-Time-Type,   -- specifies when the payload was measured
        mission-time   Relative-Time-Type,   -- and its relative timestamp.
        mode           Mode-Type,            -- operating mode of the subsystem
        payload        Payload-Type          -- actual payload
}

------------------
-- Attitude Lab --
------------------
Att-Lab-Data ::= Subsystem-Data-Template {
        Att-Lab-Mode,
        Att-Lab-Data-Measurements
}
Att-Lab-Data-Measurements ::= SEQUENCE {
        photodiodes  SEQUENCE (SIZE(4)) OF Analogue-Raw-Data,
        thermistors  SEQUENCE (SIZE(2)) OF Analogue-Raw-Data
}
```

Listado 8.6: Extracto del paquete `Subsystems` escrito en ASN.1.

8.4.3. Diseño en TASTE

Como se ha mencionado al inicio de este capítulo, todos los paquetes de alto nivel pertenecientes al OBSW han sido desarrollados con el conjunto de herramientas TASTE de la ESA. Esta sección tiene como objetivo presentar un resumen de la estrategia seguida para transformar los componentes definidos en la arquitectura de alto nivel, a los bloques básicos de construcción de un proyecto en TASTE.

La creación, desarrollo, e implementación de todos los componentes que se describen a continuación hacen uso de la máquina virtual de TASTE, que incluye todas las dependencias instaladas. La herramienta con la que se ha tenido mayor contacto ha sido *SpaceCreator*. Esta herramienta ofrece una funciona como *plug-in* del IDE tastse y ofrece una GUI para la edición todos los modelos de TASTE. La figura 8.20 ilustra parte el componente NADS empleando este IDE. En la parte superior se encuentra la IV que contiene las distintas FNTs que conforman este componente. En la parte izquierda se observan todos los proyectos TASTE empleados para el desarrollo del OBSW. Finalmente, en la parte inferior se puede observar la implementación en *C++*, de la función `Measure_And_Actuate` del componente `NADS_Manager`.

La transformación del diseño de alto nivel a TASTE ha sido intuitiva. ya que TASTE, al igual que los componentes y objetos definidos previamente, siguen los enfoques CBD, con un modelo de componentes inspirado en el meta-modelo de HRT-UML. La estrategia para efectuar esta transformación se resume en el siguiente listado:

1. Un componentes o paquete (es decir, un elemento no terminal) se corresponde con una FNT compuesta de varias FNT.
2. Una PI se corresponde con una o varias PIs de TASTE.
3. Una RI se corresponde con una o varias RIs de TASTE
4. Un objeto no protegido se corresponde con una PI no protegida que puede ser implementada por cualquier FNT.

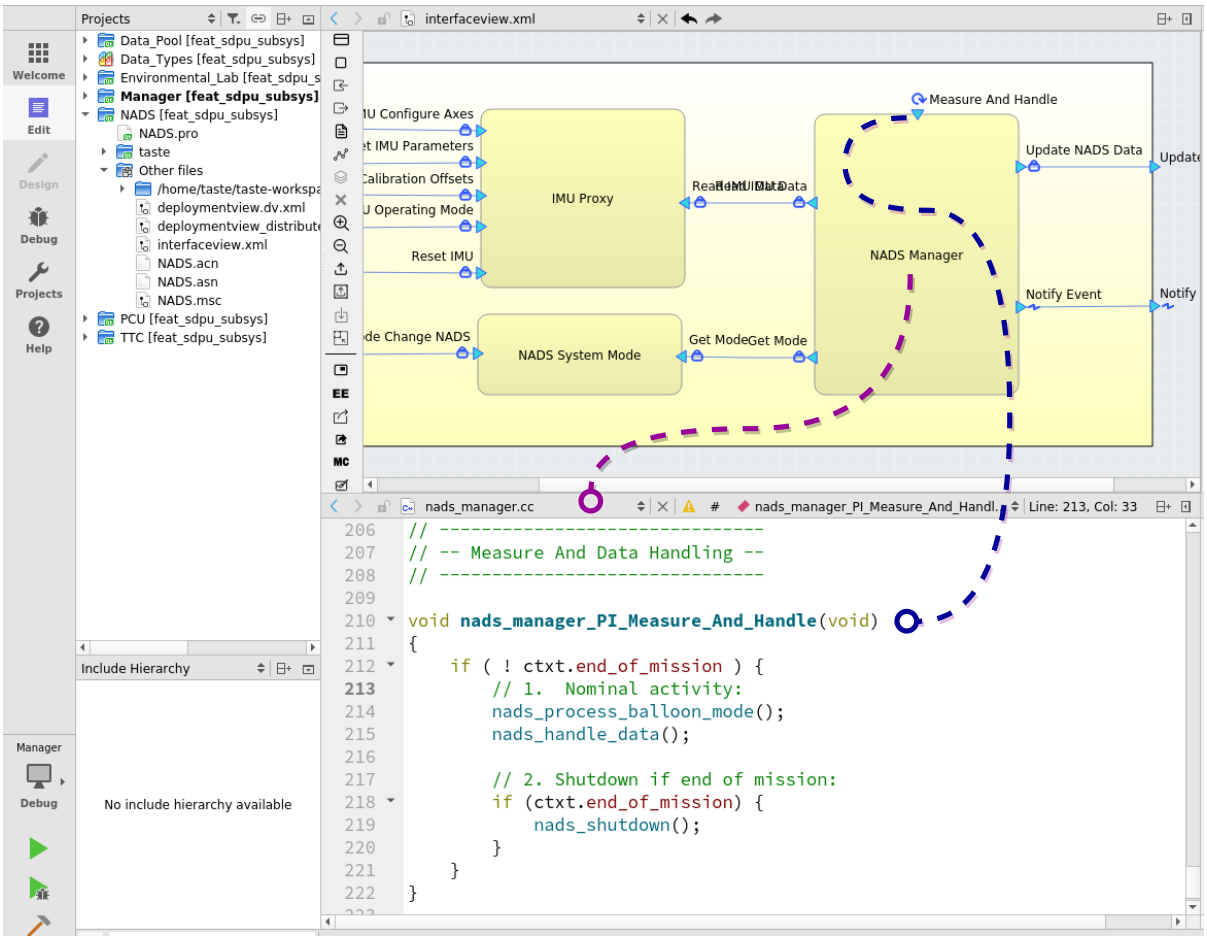


Figura 8.20: Captura de pantalla de una FNT desarrollada en SpaceCreator.

5. Un objeto protegido se corresponde con una FNT que solo implementa PIs protegidas y no protegidas.

6. Un objeto cíclico se corresponde con una FNT que solo implementa un PI periódica y cero o más PIs no protegidas.

7. Un objeto esporádico se corresponde con una FNT que solo implementa una PI esporádica y cero o más PIs no protegidas.

El resultado de haber seguido estas siete reglas se ilustra en la figura 8.21. La disposición de las FNTs se ha mantenido de forma similar al diagrama de comunicación UML del OBSW (figura 8.7). En la parte izquierda se puede identificar la FNT *Herccules Manager* que se corresponde con el `Manager` de la arquitectura de alto nivel. Se puede observar que este componente depende de todos los experimentos y subsistemas, por eso se encuentra conectado a gran parte de las PIs ofrecidas por todas las demás funciones TASTE. Asimismo, en la parte superior se identifican las dos PIs que implementa: `Notify_Event` y `Process_TC`. La primera está conectada a gran parte de las FNTs, prueba de ello es el conglomerado de líneas azules (conexiones) en la parte izquierda del sistema. Esto se debe a que el la topología de las conexiones en TASTE es estática, luego, se deben especificar todas las conexiones de forma implícita.

A modo de ejemplo, las siguientes secciones detallan el contenido de las FNT *HERCCULES*

Manager y *PCU*. Esta decisión se fundamenta en el nivel de complejidad ligeramente superior que tienen con respecto al resto. **El diseño detallado (modelos) y el código fuente relevante del resto de componentes se ha incluido en el anexo A y C, respectivamente.** Cabe destacar que, en algunos casos, los nombres adoptados en la arquitectura de alto nivel difieren de su correspondiente en TASTE por restricciones de la herramienta, como es el caso de *Manager* o *Mode*, que pertenecen al conjunto de palabras reservadas de TASTE.

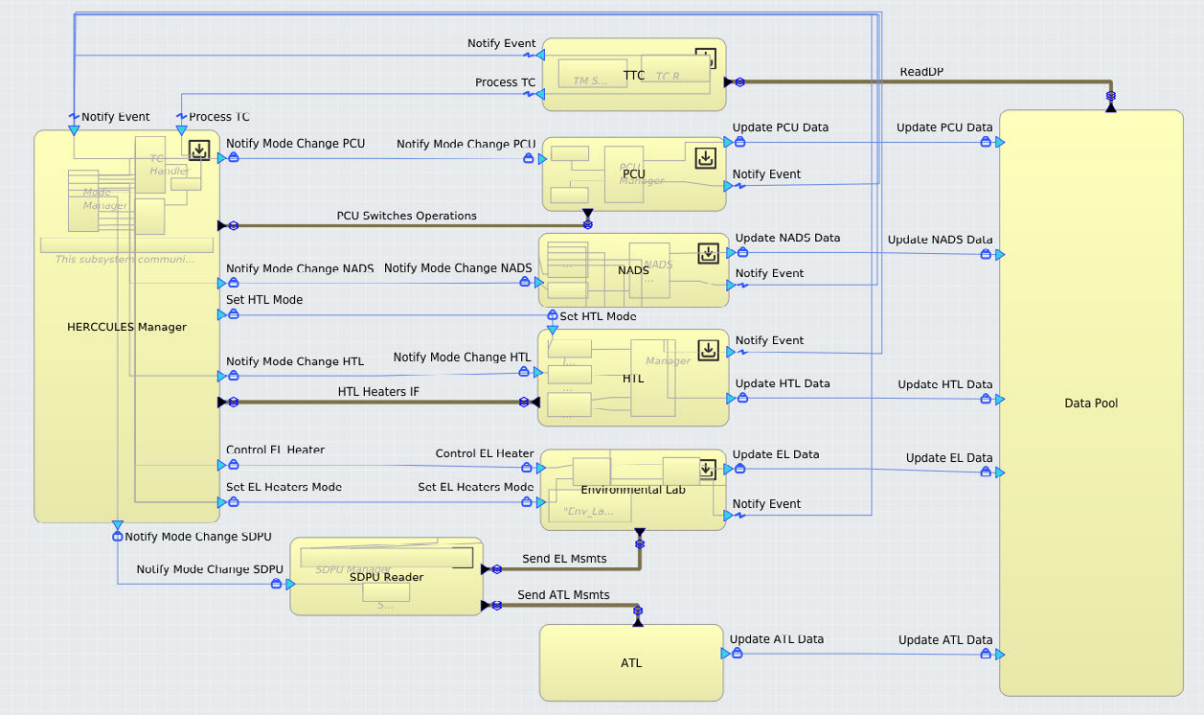


Figura 8.21: Captura de pantalla de una FNT desarrollada en TASTE/SpaceCreator.

8.4.3.1. Diseño del Manager

Como se puede apreciar en la figura 8.22 el *Manager* tiene una estructura interna que similar a la descomposición al modelo de alto nivel en UML. El siguiente listado describe sus componentes internos:

- *Manager_Facade*: Esta FNT sirve como fachada, de ahí el nombre, que provee una interfaz de alto nivel a los clientes. Se creó con el objetivo de abstraer los aspectos de concurrencia del resto de módulos.
- *TC_Handler*: Esta FNT cumple la funcionalidad del objeto `Telecommand_Manager` y se encarga de procesar los TCs provenientes de tierra. Se ha implementado en el lenguaje de modelado SDL y está conectado a gran parte de las RIs relativas al control de calefactores y también a la FNT *Mode_Manager* para solicitar los cambios de modo del sistema.
- *Event_Handler*: Lleva a cabo las funciones de `Event_Manager` y se comunica con el *Mode_Manager* para notificarle los sucesos que impliquen un cambio de modo en el sistema. Asimismo, está conectado a las RIs *Set_EL_Heaters_Mode* y *Set_HTL_Heaters_Mode* para establecerlos en modo automático en caso de perdida de conexión.

- *Mode_Manager*: Se corresponde con el objeto `Balloon_Mode` y, por tanto, implementa la máquina de estados del sistema. Esta FNT también se ha implementado en SDL debido a los elementos que proporciona para especificar estados y transiciones.

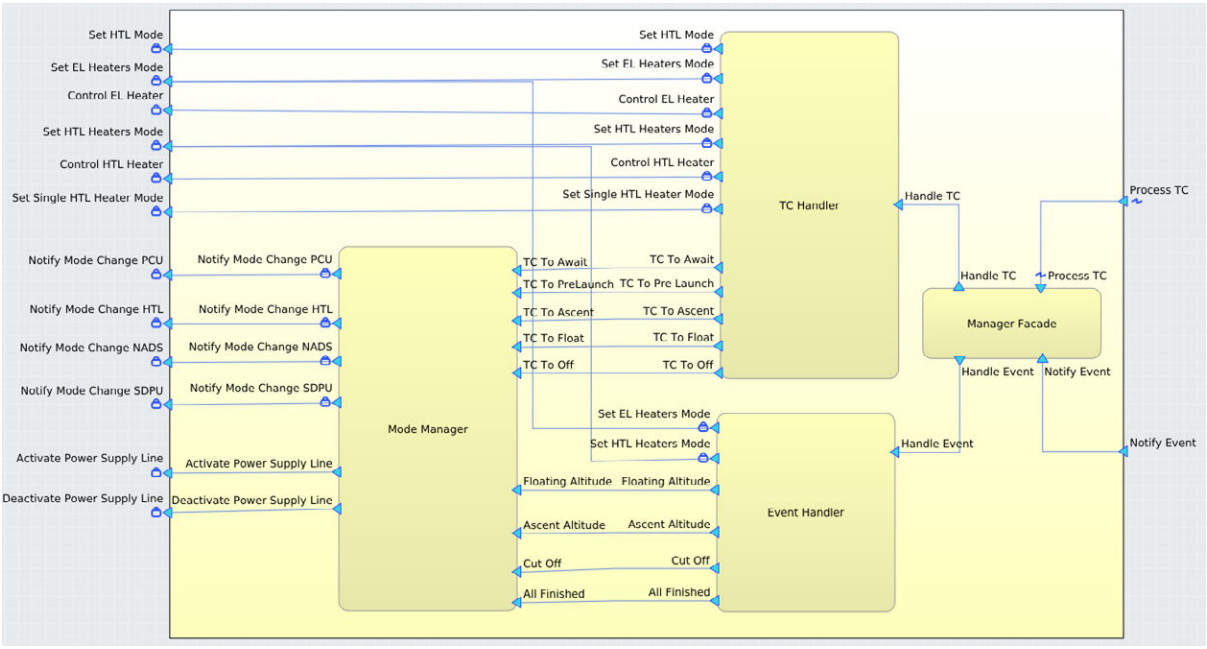


Figura 8.22: Modelo IV del Manager en TASTE.

La figura 8.23 muestra la implementación del componente *TC_Handler* en SDL. Se puede observar que cuenta con un solo estado, *Wait_For_TC*, durante el cual se está a la espera de la llegada de los TCs.

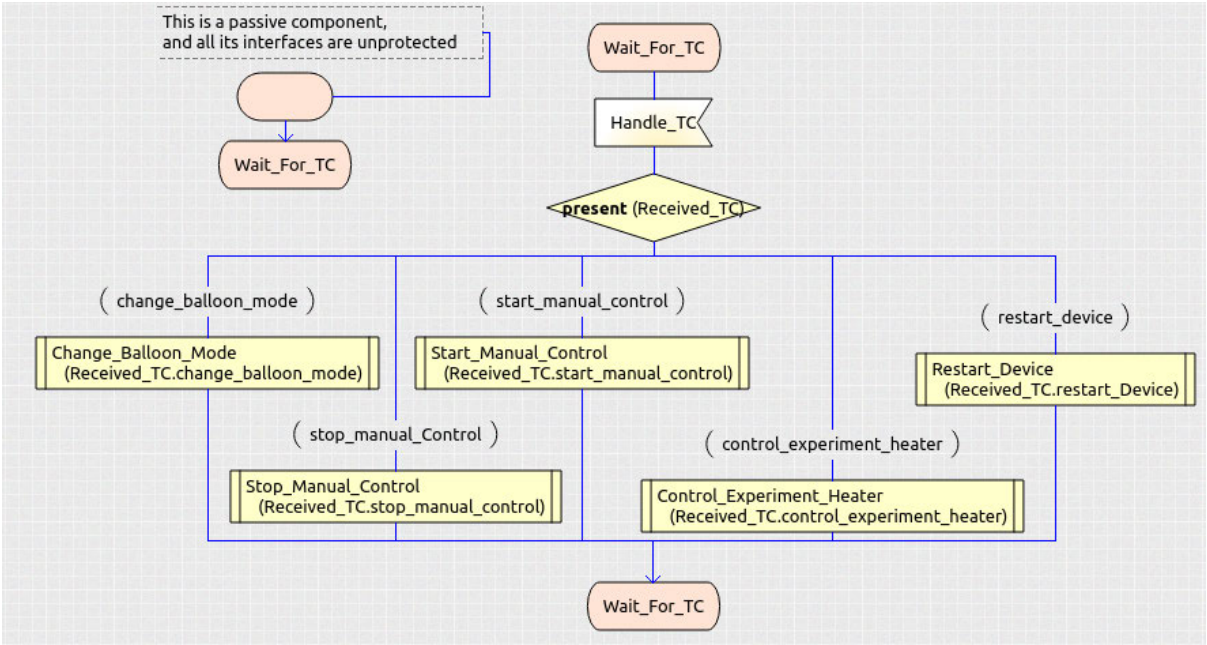


Figura 8.23: Implementación en SDL de Event Handler.

Cuano *Manager Facade* invoque la interfaz *Handle_TC*, *Event Handler* se activa y recibe el parámetro de entrada *Received_TC* como parte del evento generado. El tipo de este parámetro se corresponde con `TC-Type` que se ha definido en ASN.1 (listado de código 8.4). Tras ello, analiza que clase de TC ha recibido mediante el operador *present* y dependiendo de su valor invocará a un procedimiento u otro. Tras haber ejecutado el TC, vuelve al estado inicial a la espera de más TC que procesar. El nivel de complejidad del resto de componentes es similar o menor a este, por ello, se da por concluida la descripción de esta FNT.

8.4.3.2. Diseño de la PCU

La figura 8.24 muestra el diseño de la PCU en TASTE y guarda la siguiente relación con su diseño de alto nivel:

- *System Mode*: Se corresponde con el objeto protegido del mismo nombre. Por ello, contiene dos interfaces protegidas: *Notify Mode Change PCU* que permite al *Manager* actualizar el modo del sistema y *Get Mode* que permite al gestor interno (*PCU Manager*), la lectura del modo actual.
- *Power Supply Lines* Cumple la funcionalidad del objeto protegido `Heaters Handlers`, luego, implementa las interfaces *Activate* y *Deactivate Power Supply Lines*, las cuales permiten activar y desactivar las líneas de potencia. También provee la interfaz *Get PS Lines Status* para obtener su estado actual más reciente.
- *PCU Manager* Esta FNT lleva a cabo las funciones del objeto cíclico `Manager`. Por este motivo implementa la PI cíclica *Tick*. En cada activación, invoca la RI *Get Mode*, ya que su comportamiento depende del estado global del sistema.

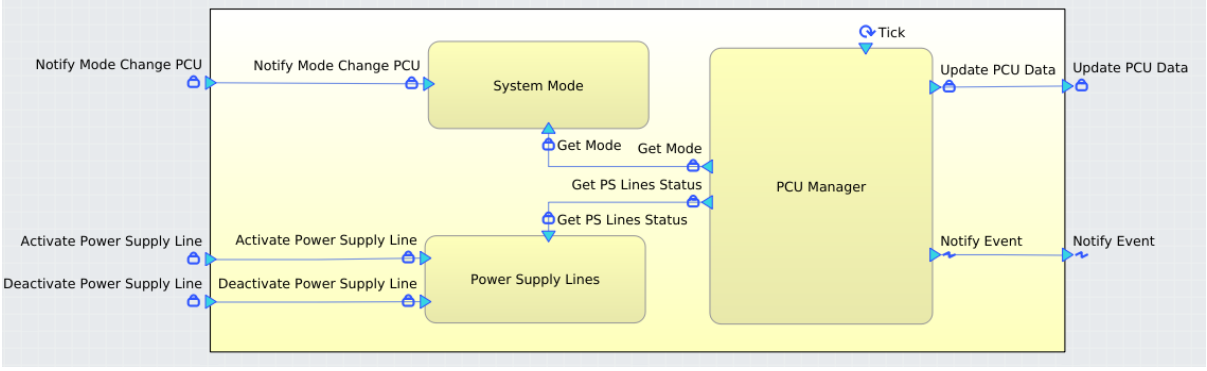


Figura 8.24: Modelo IV de la PCU en TASTE.

Todas estas FNT han sido implementadas en C++. TASTE se encarga de generar los ficheros de cabecera de los componentes, donde incluye con las declaraciones de todas las PIs y RIs. Las PIs son aquellas funciones que deben ser implementadas por el componente que las contiene y las RIs son funciones declaradas en componentes externos. El listado de código 8.7 muestra el fichero de cabecera que genera TASTE para la FNT *PCU Manager*. Se puede apreciar que la PI *Tick* se corresponde con la función `pcu_manger_PI_Tick`. Por otro lado, las interfaces requeridas son definidas como funciones externas y nombradas con el prefijo `pcu_manager_RI`.

```c
#pragma once
#include "dataview-uniq.h" // Data View implemented in C

void pcu_manager_startup(void);

/* Provided interfaces */
void pcu_manager_PI_Tick( );

/* Required interfaces */
extern void pcu_manager_RI_Get_Mode(asn1SccBalloon_Mode *);

extern void pcu_manager_RI_Get_PS_Lines_Status
                (asn1SccPCU_PS_Lines_Status *);

extern void pcu_manager_RI_Notify_Event(const asn1SccBalloon_Events *);

extern void pcu_manager_RI_Update_PCU_Data(const asn1SccPCU_Data *);
```

Listado 8.7: Especificación autogenerada por TASTE para la función PCU-Manager.

Finalmente, el listado 8.8 presenta un extracto de la implementación de dicho componente. Las funciones auxiliares en el espacio de nombres estático se ha omitido en favor de mostrar la implementación completa de la PI *Tick* y de las funciones de inicializacion y finalización.

```c
namespace { // Auxiliary static data and functions
    pcu_manager_state ctxt;

    //--snip--

    void pcu_measure_data() {
        // Get time:
        auto && [secs, usecs] = time_management::absolute_time();
        ctxt.pcu_data.snapshot_time = {secs, usecs};
        ctxt.pcu_data.mission_time  = time_management::mission_time();

        // Read actuators status:
        pcu_manager_RI_Get_PS_Lines_Status
            (&ctxt.pcu_data.payload.switches);

        // Read digital data, dependant on the PCU mode:
        read_digital_sensors_data();
    }
}

// --  Provided Interfaces  ---------------------------------------

// -- Startup & Shutdown --

/**
* @note This startup routine is invoked by the TASTE runtime,
* which guarantees that it's executed only once, even if it was
* invoked multiple times.
*/
```

```
30 void pcu_manager_startup(void) {
31     (void) bs::pcu::initialize_switches();
32 }
33
34 /**
35 * This is the shutdown routine that shall be invoked when
36 * "end of mission" is reached.
37 */
38 void pcu_manager_shutdown(void) {
39    ctxt.logfile.save_file();
40    // Notify the Manager that this subsystem has finished:
41    asn1SccBalloon_Events finish {asn1SccBalloon_Events_pcu_finished};
42    pcu_manager_RI_Notify_Event(&finish);
43 }
44
45 // -- Tick --
46 /**
47 * Periodic activity that performs the core logic of the PCU subsystem
48 *
49 * Why do we check the 'has_finished' condition twice?
50 * - The condition is updated inside 'pcu_process_balloon_mode'.
51 * - This is a periodic activity.
52 */
53 void pcu_manager_PI_Tick(void) {
54     if ( ! ctxt.end_of_mission ) {
55         // 1.  Nominal activity:
56         pcu_process_balloon_mode();
57         pcu_measure_data();
58         pcu_store_data();
59         pcu_publish_data();
60         // 2.  Shutdown if end of mission:
61         if (ctxt.end_of_mission) {
62             pcu_manager_shutdown();
63         }
64     }
65 }
```

Listado 8.8: Componente PCU implementado C++.

Capítulo 9

Validación del sistema

En este capítulo se describe el proceso de validación del OBSW de HERCCULES. que se corresponde con la segunda mitad del ciclo de vida software (modelo en "V"). El objetivo es demostrar que el software se comporta de acuerdo a las especificaciones del sistema y cumple con los requisitos establecidos. Como se discutió en la especificación de estos requisitos, se han propuesto cuatro métodos de validación: mediante pruebas automatizadas, análisis, revisión e inspección. El conjunto de de pruebas definidos para validar HERCCULES se realizó a nivel unitario, de integración y de sistema. Como se verá a continuación, todas estas pruebas se han validado, mayoritariamente, empleando los métodos de pruebas automatizadas e inspección.

9.1. Pruebas Unitarias del OBSW

Este tipo de pruebas tiene como objetivo validar de forma independiente los componentes software del diseño detallado. En el caso de HERCCULES existe una gran variedad de módulos, que a su vez, se han descompuesto en diversos objetos terminales. Por ello, estas pruebas se han centrado principalmente en los componentes de más bajo nivel que tienen acceso directo con los dispositivos físicos del sistema. También se realizaron pruebas unitarias sobre el gestor de los modos de operación del sistema y el componente encargado de almacenar los datos en la memoria persistente del OBC.

En conjunto estas pruebas unitarias han permitido probar las prestaciones (*features*) relativas a la adquisición de datos de los sensores, el control de los actuadores, y la gestión de los modos de operación. A continuación se presenta una clasificación de las pruebas unitarias que se han definido a lo largo del proyecto. En su descripción se incluye el método de validación de la prueba y las prestaciones que pretenden validar.

- *Almacenamiento de los datos*: El almacenamiento de la TM adquirida durante el vuelo es tan importante como la extracción de los datos. Por este motivo, el módulo encargado del grabado de los datos ha sido sometido a una serie de pruebas automatizadas en las cuales se corrobora la escritura de diversos datos (de distintos tipos) en memoria persistente.
- *Lectura de los sensores digitales de la PCU*: La PCU cuenta únicamente con sensores digitales comunicados con el OBC mediante el bus I2C. La realización exitosa de es-

tas pruebas, por tanto, no solo implican la validez de los componentes que gestionan estos sensores digitales, sino también permite validar, indirectamente, el componente `Bus_Handlers` que se encarga de la gestión de las comunicaciones con el bus I2C. Las pruebas pertenecientes a esta categoría se validaron mediante una serie de programas de demostración que muestra a los operadores los valores de los sensores leídos. Por tanto, se trata de una validación por inspección.

- *Lectura de los sensores digitales de la SDPU*: Los sensores digitales de la SDPU incluyen dos barómetros absolutos. Al igual que el caso anterior, se crearon programas de demostración que permitían inspeccionar las medidas adquiridas en tiempo real.

- *Lectura de los sensores de la NADS*: Los sensores del NADS incluyen una IMU conectada por I2C y un sensor GPS conectado por UART. Luego, las pruebas de esta categoría permitieron validar tanto los componentes encargados de leer la IMU y GPS, como las funciones encargadas de leer el puerto serie de la RPi. En este caso también se emplearon programas de demostración que permitieron inspeccionar las lecturas de la IMU y GPS en tiempo real.

- *Lectura de datos analógicos mediante el ADC*: En este caso se sometió a prueba el módulo encargado de comandar y leer las conversiones del ADC. La realización de estas pruebas precisó de tarjetas electrónicas adicionales que proporcionaban niveles de tensión fijos. De forma similar a las pruebas anteriores, las pruebas se validaron mediante inspección visual del experimento con programas de demostración.

- *Control de multiplexores*: La capa `Equipment_Handlers` contiene clases que abstraen el acceso a los multiplexeros. En esencia, estas controlan tres pines GPIO y configuran sus estados de acuerdo a la selección de la línea del multiplexor sobre la que se desee trabajar. La consecución exitosa de estas pruebas no solo depende de la validez de las clases antes mencionadas, sino también de la librería *piGPIO*, encargada de las operaciones de más bajo nivel para los pines GPIO.

- *Control PWM de los calefactores*: Estas pruebas permitieron verificar el comportamiento de la bilbioteca *piGPIO* para generar ondas PWM. Dichas pruebas se validaron empleando osciloscopios, de esta forma se pudo comprobar la configuración de parámetros como *duty-cycle* o frecuencia de onda.

- *Gestión del modo de operación del sistema*: Finalmente se probó el *Manager* de forma independiente y con diversos casos de prueba automatizados. Estos casos de prueba corroboraron las transiciones entre estados a partir de una serie de eventos simulados con el conjunto de herramientas de TASTE.

Todas estas pruebas se realizaron siguiendo un enfoque de prototipos incrementales. Esto debido a que tanto el software como el hardware se desarrollaron en paralelo. Por tanto, en las primeras fases se llevaron a cabo pruebas con sensores conectados al OBC mediante placas de pruebas (*protoboards*). En el caso particular de las pruebas automatizadas, se empleó la biblioteca Google Test (*g-test*) para crear y gestionar de las pruebas unitarias para el lenguaje de programación C++.

9.2. Pruebas de Integración

Las pruebas de integración tienen como objetivo validar los componentes que aúnan los módulos validados mediante pruebas unitarias. Estas pruebas permiten comprobar el funcionamiento de los componentes de manera individual y de forma conjunta. En el caso de HERCCULES, se realizaron pruebas sobre prototipos de alta fidelidad para las tarjetas de E/S: SDPU, PCU, TMU y NADS. A modo ilustrativo se muestra el prototipo desarrollado por el equipo de electrónica de la TMU (figura 9.1). En dicha placa se pueden identificar integrados un ADC, un multiplexor, y siete salidas de las cuales seis están destinadas a resistencias que proporcionan tensiones fijas y una dedicada al termistor PT1000 (parte superior izquierda).

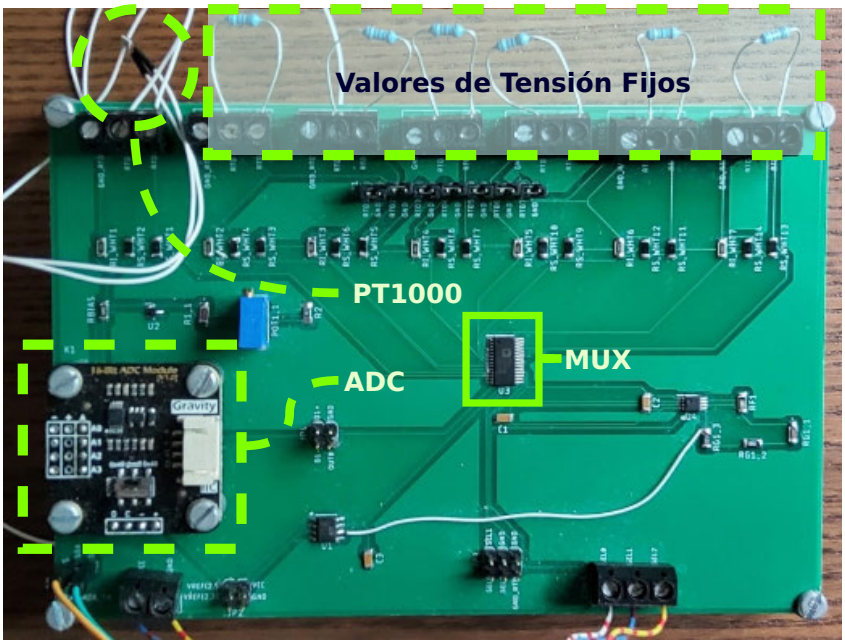


Figura 9.1: Prototipo de alta fidelidad de la placa de E/S de la TMU.

Gracias a estos prototipos se pudieron validar los componentes pertenecientes al paquete `Board Support`, es decir, PCU, SDPU y TMU. El conjunto de pruebas de integración se validó mediante la inspección del sistema empleando programas de demostración. En la figura 9.2 se puede apreciar la salida por el terminal del programa de demostración del prototipo de la TMU. Estos datos mostrados con un periodo de dos segundos, permiten al operador visualizar las muestras obtenidas de todos los cuatro canales del ADC. Los resultados visualizados por la terminal también se almacenaron en memoria persistente, lo cual permitió llevar a cabo un análisis más detallado.

Asimismo, estos prototipos se aprovecharon para la validación de los componentes de alto nivel implementados en TASTE como el HTL, el PCU, NADS, EL y ATL. Estos módulos se probaron en proyectos independientes y las dependencias hacia elementos inexistentes se simularon con componentes de tipo “mock”. Por ejemplo, la figura 9.3 muestra la IV del proyecto de prueba para el componente NADS. Se puede apreciar en la parte inferior el componente que implementa la funcionalidad del NADS y en la parte superior los dos bloques de tipo GUI que TASTE transforma de forma automática en interfaces gráficas que permiten al usuario inspeccionar el comportamiento de dicho subsistema.

```
pi@raspberrypi:~/herccules/release/bin/Demos $ sudo ./TMUDemo
[PIGPIO] Successful initialization!
[TMUDemo] tmu initialized!
[TMUDemo] reading at 8 samples/second
Please, introduce the desired sample period:
2
Period: 2 secs, 0nsecs
Channel: 0 - 0 = 2.35512
Channel: 0 - 1 = 2.35238
Channel: 0 - 2 = 2.35337
Channel: 0 - 3 = 2.34925
Channel: 1 - 0 = 3.83313
Channel: 1 - 1 = 3.8505
Channel: 1 - 2 = 3.84075
Channel: 1 - 3 = 3.86538
Channel: 2 - 0 = 0.040875
Channel: 2 - 1 = 0.04025
Channel: 2 - 2 = 0.037375
Channel: 2 - 3 = 0.039
```

Figura 9.2: Salida por consola del programa de demostración para el prototipo de la TMU.

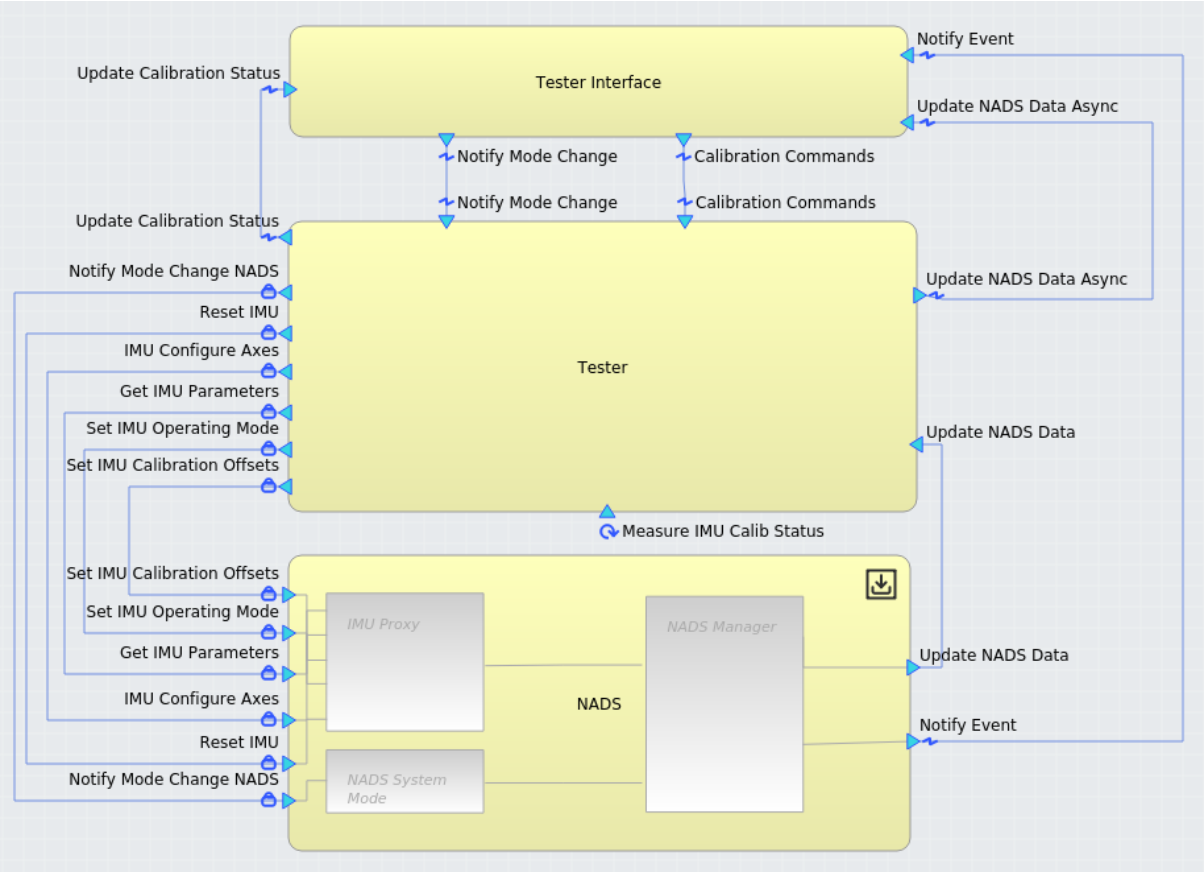


Figura 9.3: Proyecto en TASTE para la prueba del componente NADS.

Por otra parte, la validación del gestor de modos del sistema, se realizó empleando componentes *mock* que simulaban la funcionalidad de componentes que, para el momento de las pruebas, no habían sido desarrollados. En la figura 9.4 se puede apreciar el modelo empleado para la validación del *Manager*. En la parte superior se encuentra un bloque que representa la interfaz gráfica del usuario, mientra que en la parte inferior derecha se encuentra el *Manager* que está

sometido a pruebas. Finalmente, en la parte inferior izquierda se aprecian los *Mock* de los experimentos encargados de implementar y hacerse pasar por todas las PIs del componente *Subsystems*.

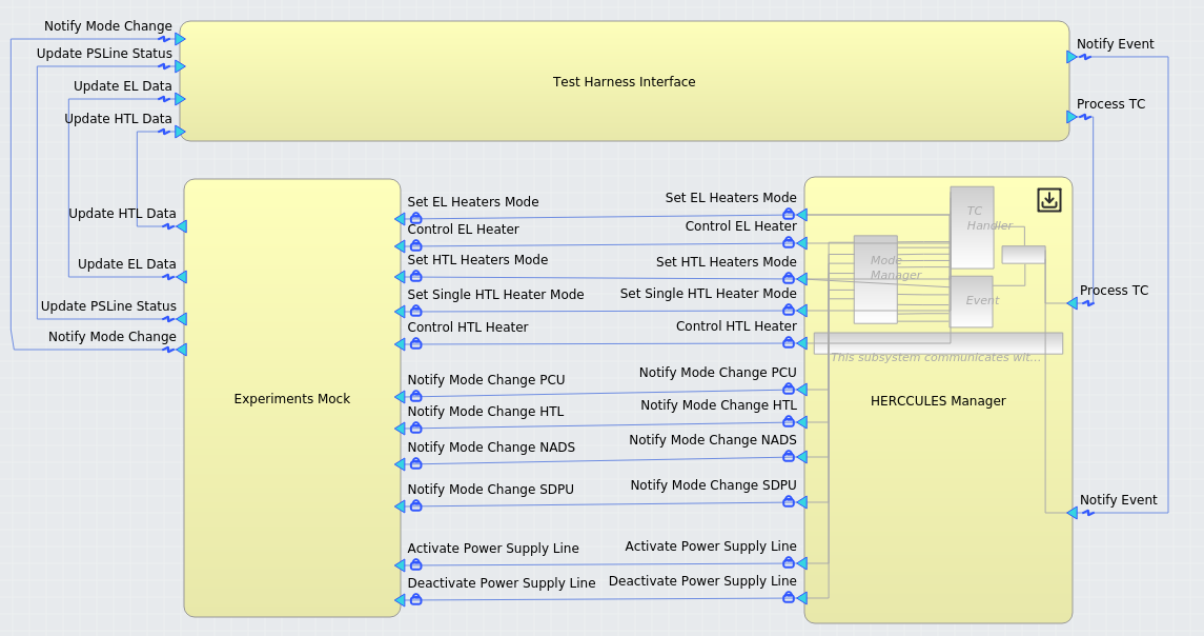


Figura 9.4: Proyecto en TASTE para la prueba del componente Manager.

9.3. Pruebas Sobre Tarjetas Electrónicas

Estas pruebas se realizaron sobre las tarjetas electrónicas (no prototipos) de la SDPU, PCU y TMU. Se realizaron pruebas a nivel de tarjeta de forma independiente, e integradas para validar la funcionalidad del OBSW completo. En la figura 9.5 se puede apreciar el OBC conectado al resto de subsistemas. A su vez, estas tarjetas ensamblan los sensores digitales como ADC, barómetros, termómetros, entre otros. En gran parte de las tarjetas, se han omitido omitido los sensores analógicos reales. En estos casos, el equipo de electrónico desarrollo unos denominados *test-becnh* que pretermitieron establecer valores fijos a la entrada de todas las líneas analógicas. Por lo tanto, a partir de estos valores bien definidos, se pudo crear un conjunto de pruebas automatizadas en las cuales se probó la validez del OBSW comparando las lecturas obtenidas del ADC con los valores esperados establecidos en formato CSV. De esta manera se facilitó la modificación de futuras pruebas por cualquier miembro del proyecto ajeno al desarrollo del software. De forma complementaria, se crearon programas de demostración, con interfaz gráfica incluida, para la inspección visual y control del sistema en tiempo real.

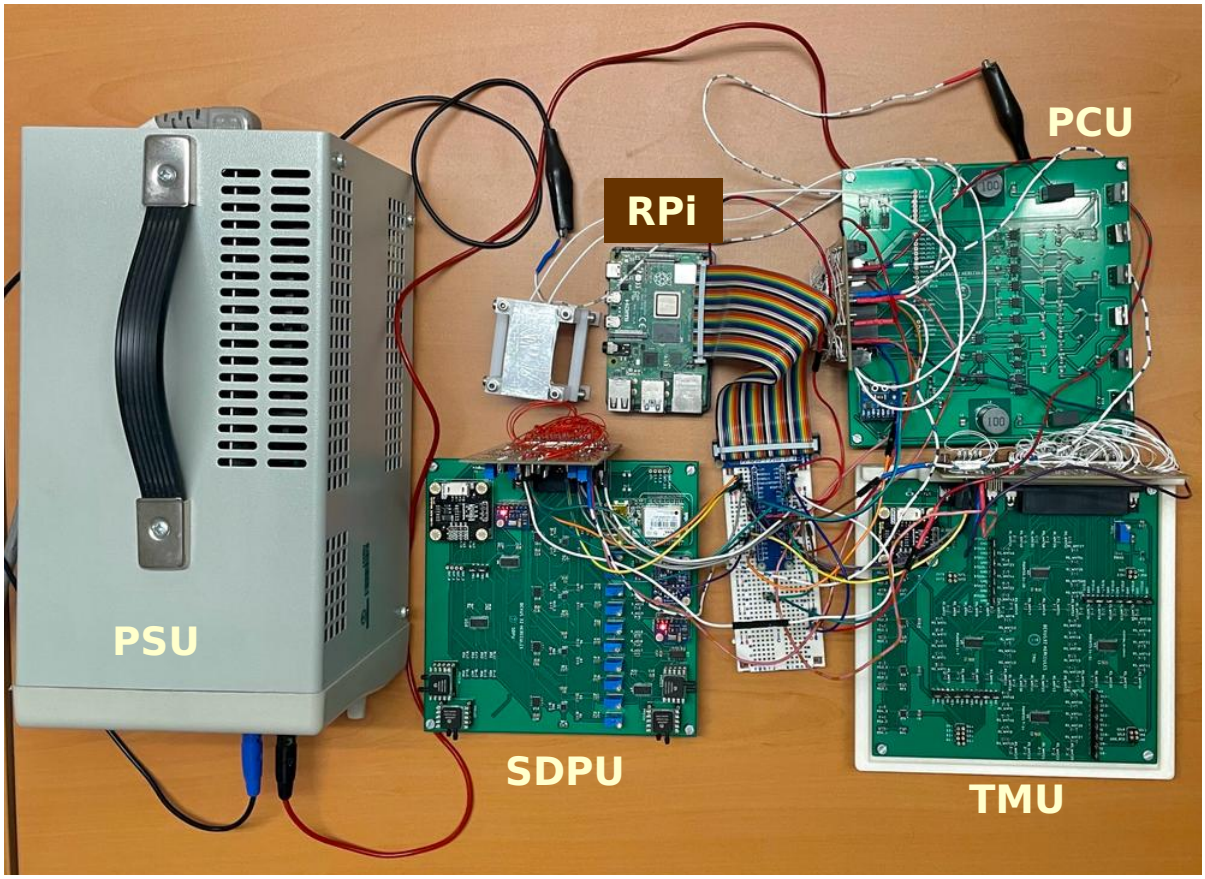


Figura 9.5: OBC conectado a los RTU.

Capítulo 10

Resultados y Discusión

En esta sección se presentan un resumen de los los resultados obtenidos tras haber ejecutado las pruebas unitarias, de integración y de sistema definidas para la validación del OBSW. Asimismo, se incluye un análisis de los mismos para valorar el rendimiento y la funcionalidad del sistema conforme a sus requisitos.

10.1. Validación de la PCU

La PCU está compuesta de un sensor de corriente, voltaje y potencia, un sensor de temperatura, controladores PWM, e interruptores para el encendido y apagado de los subsistemas. La principal estrategia para la validación de esta tarjeta electrónica se realizó mediante **inspección visual** a través de diversos programas de demostración. También se realizó una validación por **análisis**, aunque en menor medida, ya que se realizaron ensayos estadísticos de las medidas monitorizadas.

La validación de la lectura del sensor de potencia, voltaje y corriente se realizó inspeccionando la evolución de las mediciones tras haber manipulado la tensión de salida de la fuente de alimentación como se ilustra en la figura 10.1. En el instante 0 se estableció una tensión de 26 V, tras ello la fuente se configuró a 28 y 28.8 V, para finalmente llegar a los 30 V.

Figura 10.1: Cambios en la fuente de alimentación.

La figura 10.2 presenta la GS de HERCCULES, donde se pueden identificar los datos de TM capturados para la PCU. La gráfica que se aprecia en la parte inferior ilustra cien muestras de la tensión de entrada en el sistema cuando todos los sistemas se encuentran apagados (*switches*

en estado *off*). Se puede ver que dicha gráfica es un reflejo de las variaciones en al fuente de alimentación.

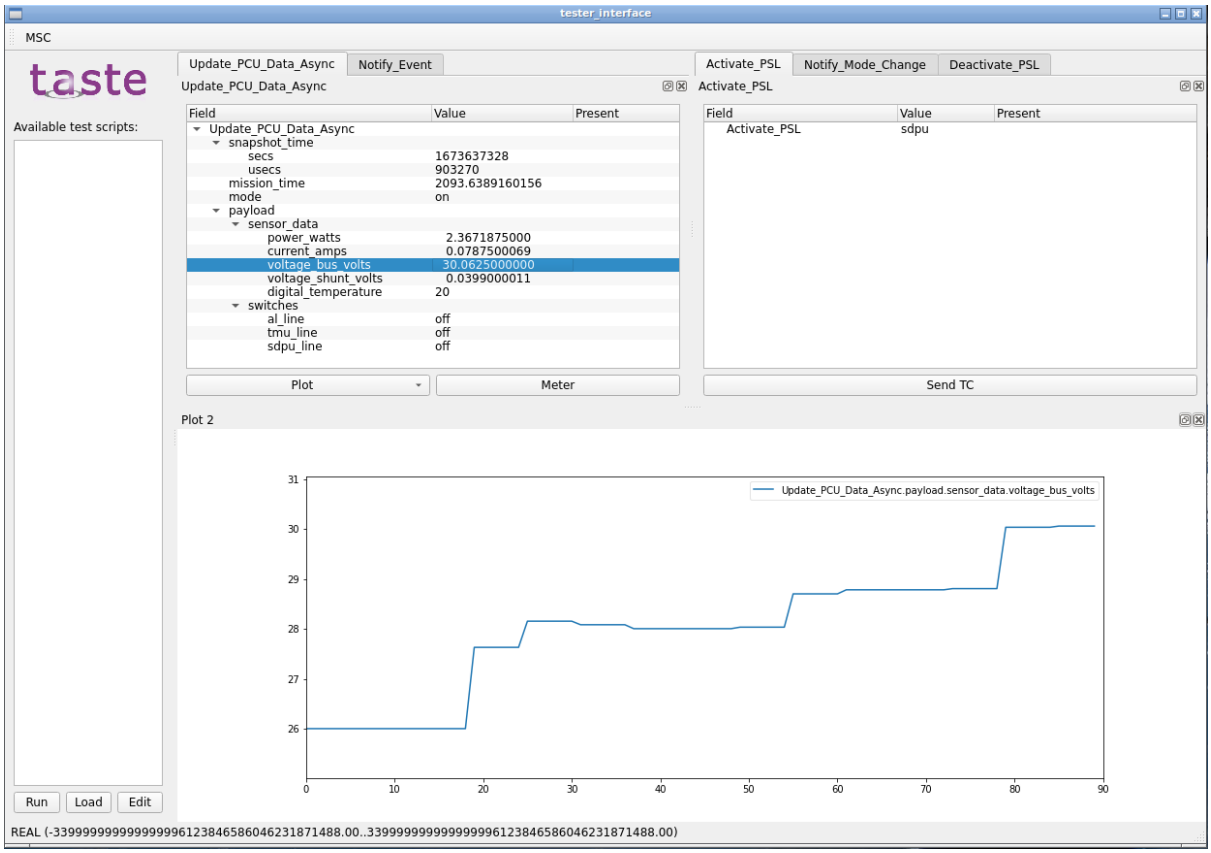


Figura 10.2: GUI de la GS autogenerada por TASTE.

Los módulos de temperatura y el control de los interruptores y controladores PWM se validaron de forma similar al sensor de voltaje, y por razones de brevedad se han omitido en esta sección.

10.2. Validación de la TMU

La TMU está compuesta de un ADC y un multiplexor que permiten seleccionar los 28 termistores del HTL. La validación de este subsistema se apoya en las *test-bench* que proporcionan valores de tensión fijos en todas las entradas. Se realizaron dos tipos de pruebas:

- Pruebas automatizadas con *g-test* que comparan las medidas obtenidas por todos los canales con los valores esperados. Las pruebas se han configurado con un error máximo de 1 mV.
- Análisis estadístico de cien muestras obtenidas por todos los canales para obtener el error medio en las mediciones.

En el primer caso se ejecutaron las pruebas múltiples veces y no se detectaron fallos. La figura 10.3 muestra la salida por consola tras haber ejecutado dichas pruebas. En cuanto al análisis de estas pruebas se obtuvieron los resultados que se resumen en la figura 10.4 donde se observar que las diferencias nimias entre los valores esperados y los capturados.

```
pi@raspberrypi:~/herccules/release/bin/TMUTests $ sudo ./TMUTests
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from TMUTests
[PIGPIO] Successful initialization!
,ADC Channel 0,ADC Channel 1,ADC Channel 2,ADC Channel 3
Reading row from MUX channel 0:  | 18847 | 18839 | 18841 | 18842
Reading row from MUX channel 1:  | 30855 | 30968 | 30930 | 31092
Reading row from MUX channel 2:  | 356 | 317 | 316 | 366
Reading row from MUX channel 3:  | 18802 | 18864 | 18824 | 18714
Reading row from MUX channel 4:  | 18894 | 18900 | 18984 | 18824
Reading row from MUX channel 5:  | 18904 | 18777 | 18820 | 18826
Reading row from MUX channel 6:  | 18900 | 18749 | 19168 | 18824
[ RUN      ] TMUTests.InitializationSuccess
[       OK ] TMUTests.InitializationSuccess (0 ms)
[ RUN      ] TMUTests.CheckAllADCChannels
[       OK ] TMUTests.CheckAllADCChannels (3641 ms)
[ RUN      ] TMUTests.CheckIndividualReadingsFromEachAnalogueLine
[       OK ] TMUTests.CheckIndividualReadingsFromEachAnalogueLine (7145 ms)
[ RUN      ] TMUTests.CompareIndividialReadingsWithCompleteReadings
[       OK ] TMUTests.CompareIndividialReadingsWithCompleteReadings (7281 ms)
[----------] 4 tests from TMUTests (18069 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (18079 ms total)
[  PASSED  ] 4 tests.
```

Figura 10.3: Pruebas automatizadas para la TMU.

Mult plexor	N° Canal	Temp.ideal(C)	Vo.ideal(V)	Vo.medida (V)	Temp.medida(C)	Error (V)	Error (C)
MULT 0	CANAL 0	0.288	2.3559	2.361	0.287	0.0051	-0.002
	CANAL 1	51.592	3.8569	3.836	50.876	0.0209	-0.716
	CANAL 2	-78.717	0.0445	0.0411	-78.833	0.0034	-0.116
	CANAL 3	0.096	2.3503	2.3462	-0.045	0.0041	-0.141
	CANAL 4	0.488	2.3618	2.3592	0.399	0.0026	-0.089
	CANAL 5	0.53	2.363	2.363	0.529	0	-0.001
	CANAL 6	0.514	2.3626	2.3605	0.444	0.0021	-0.07
MULT 1	CANAL 0	0.252	2.3549	2.3525	0.17	0.0024	-0.081
	CANAL 1	52.073	3.871	3.8542	51.499	0.0168	-0.574
	CANAL 2	-78.881	0.0397	0.041	-78.837	0.0013	0.044
	CANAL 3	0.358	2.358	2.3561	0.293	0.0019	-0.065
	CANAL 4	0.517	2.3626	2.3641	0.567	0.0015	0.05
	CANAL 5	-0.01	2.3472	2.349	0.051	0.0018	0.061
	CANAL 6	-0.13	2.3437	2.3434	-0.141	0.0003	-0.011
MULT 2	CANAL 0	0.262	2.3552	2.3544	0.235	0.0008	-0.027
	CANAL 1	51.912	3.8663	3.8442	51.157	0.0221	-0.755
	CANAL 2	-78.883	0.0396	0.0377	-78.949	0.0019	-0.066
	CANAL 3	0.192	2.3531	2.3495	0.068	0.0036	-0.124
	CANAL 4	0.873	2.373	2.3705	0.786	0.0025	-0.087
	CANAL 5	0.174	2.3526	2.3511	0.123	0.0015	-0.051
	CANAL 6	1.662	2.3961	2.3871	1.353	0.009	-0.309
MULT 3	CANAL 0	0.265	2.3553	2.3491	0.054	0.0062	-0.211
	CANAL 1	52.603	3.8865	3.8697	52.028	0.0168	-0.574
	CANAL 2	-78.673	0.0458	0.0397	-78.881	0.0061	-0.208
	CANAL 3	-0.281	2.3393	2.333	-0.496	0.0063	-0.216
	CANAL 4	0.19	2.3531	2.3479	0.013	0.0052	-0.176
	CANAL 5	0.197	2.3533	2.3467	-0.028	0.0066	-0.225
	CANAL 6	0.19	2.3531	2.3466	-0.031	0.0065	-0.221

Figura 10.4: Datos estadísticos capturados de la TMU.

10.3. Validación de la SDPU

La SDPU contiene una serie de sensores digitales como barómetros absolutos y relativos pero también cuenta con una serie de entradas analógicas para la lectura de los piranómetros, pirgeómetros, termistores y barómetros diferenciales (también llamados anemómetros).

Se ha seguido la misma metodología que el proceso de validación de la TMU. En el caso de la pruebas automatizadas se tienen en cuenta los valores de tensión fijos que se esperan obtener en cada entrada analógica. En cuanto a los sensores digitales, los únicos que se han podido probar de forma automática, fueron los barómetros ya que la presión atmosférica se puede obtener de diversas estaciones meteorológicas disponibles en la red (un valor aproximado de 954 mBar en el momento en que se redacta esta memoria). A esto se suma el hecho de que la SDPU cuenta con dos barómetros, por lo tanto se pueden comparar ambas medidas. La figura 10.5 muestra la salida por consola de la ejecución de las pruebas automatizadas con *g-test*.

```
pi@raspberrypi:~/herccules/release/bin/SDPUTests $ sudo ./SDPUTests
[==========] Running 16 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 16 tests from SDPUTests
[PIGPIO] Successful initialization!
,ADC Channel 0,ADC Channel 1,ADC Channel 2,ADC Channel 3
Reading row from MUX channel 0: 8000 | 2182 | 16384 | 0 |
Reading row from MUX channel 1: 8000 | 17944 | 16384 | 0 |
Reading row from MUX channel 2: 16384 | 30633 | 16384 | 0 |
Reading row from MUX channel 3: 16384 | 2184 | 16384 | 0 |
Reading row from MUX channel 4: 0 | 18847 | 0 | 0 |
Reading row from MUX channel 5: 0 | 30693 | 0 | 0 |
Reading row from MUX channel 6: 0 | 0 | 0 | 0 |
[ RUN      ] SDPUTests.InitializationSuccess
[       OK ] SDPUTests.InitializationSuccess (0 ms)
[ RUN      ] SDPUTests.MultipleInitializations
[       OK ] SDPUTests.MultipleInitializations (0 ms)
[ RUN      ] SDPUTests.CheckAllADCChannelsReverse
[       OK ] SDPUTests.CheckAllADCChannelsReverse (655 ms)
[ RUN      ] SDPUTests.CheckAllADCChannels
[       OK ] SDPUTests.CheckAllADCChannels (0 ms)
[ RUN      ] SDPUTests.CheckPyranometerReadings
[       OK ] SDPUTests.CheckPyranometerReadings (316 ms)
[ RUN      ] SDPUTests.CheckPyrgeometerReadings
[       OK ] SDPUTests.CheckPyrgeometerReadings (316 ms)
[ RUN      ] SDPUTests.CheckPyranometersAndPyrgeometersTemperatures
[       OK ] SDPUTests.CheckPyranometersAndPyrgeometersTemperatures (632 ms)
[ RUN      ] SDPUTests.CheckNadirTemperature
[       OK ] SDPUTests.CheckNadirTemperature (316 ms)
[ RUN      ] SDPUTests.CheckNadirPhotodiodes
[       OK ] SDPUTests.CheckNadirPhotodiodes (632 ms)
[ RUN      ] SDPUTests.CheckDifferentialBarometers
[       OK ] SDPUTests.CheckDifferentialBarometers (632 ms)
[ RUN      ] SDPUTests.CheckAbsoluteBarometers
[       OK ] SDPUTests.CheckAbsoluteBarometers (36 ms)
[ RUN      ] SDPUTests.AbsoluteBarometersAreSimilar
[       OK ] SDPUTests.AbsoluteBarometersAreSimilar (36 ms)
[ RUN      ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel0
[       OK ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel0 (262 ms)
[ RUN      ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel1
[       OK ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel1 (262 ms)
[ RUN      ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel2
[       OK ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel2 (262 ms)
[ RUN      ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel3
[       OK ] SDPUTests.CompareIndividialReadingsWithCompleteReadingsFromChannel3 (262 ms)
[----------] 16 tests from SDPUTests (4622 ms total)

[----------] Global test environment tear-down
[==========] 16 tests from 1 test case ran. (4638 ms total)
[  PASSED  ] 16 tests.
```

Figura 10.5: Pruebas automatizadas para la SDPU.

Tras la ejecución de estas pruebas, se realizó una inspección visual del comportamiento del experimento a través las gráfica obtenidas con la GS de HERCCULES. Por motivos de brevedad solo se introduce el análisis de los dos pirgeómetros (sensor analógico) y de los dos barómetros (sensores digitales). La figura 10.6 muestra la evolución de cien muestras en bruto capturadas del pirgeómetro del UEL (traza azul) y del DEL (traza naranja). La entrada del ADC correspondiente al pirgeómetro del UEL es de 16375, mientras que la entrada al canal del pirgeómetro del DEL es de 16400.

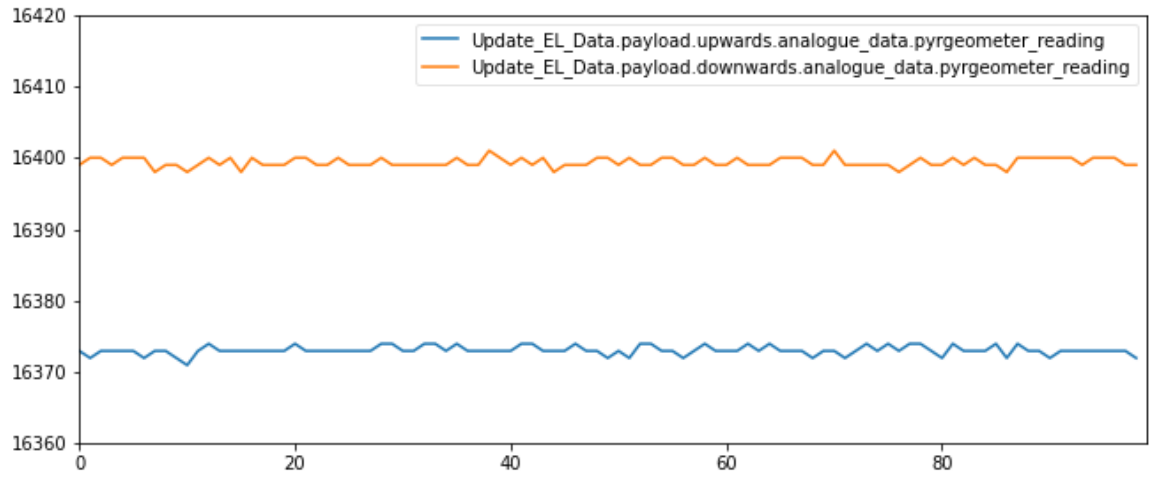


Figura 10.6: Datos capturados de los pirgeómetros de la SDPU.

Por otro lado, la figura 10.7 muestra cien muestras de los dos sensores de presión absolutos en mBar. Se observa que ambas trazas tienen valores cercanos a los 954mBar, con ligeras variaciones, menores a 0.25 mBar.

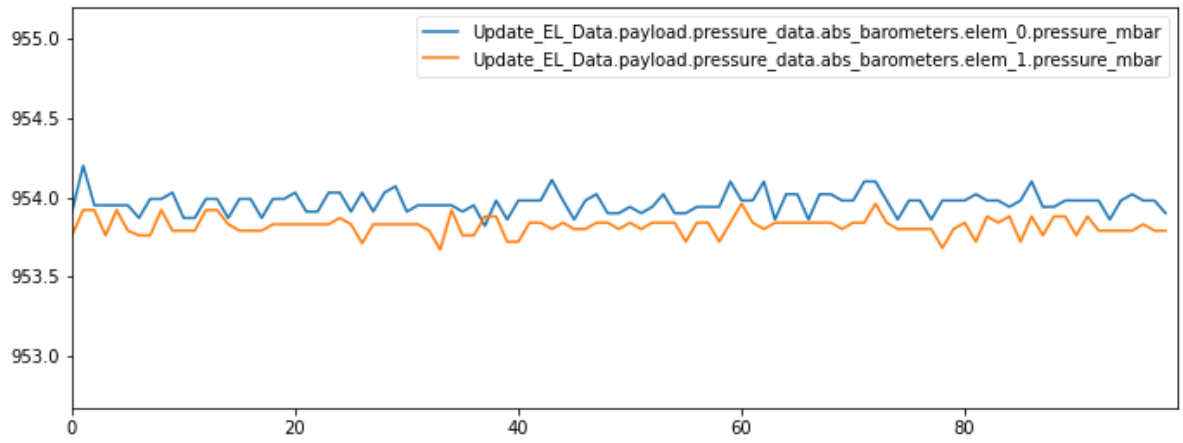


Figura 10.7: Datos capturados de los barómetros absolutos de la SDPU.

Capítulo 11

Conclusiones

Los objetivos planteados en este trabajo de in de máster se han cumplido satisfactoriamente y dentro de los plazos establecidos. En concreto, durante las primeras fases de análisis del proyecto HERCCULES se estudiaron diversas tecnologías, marcos de desarrollo (*frameworks*) y bibliotecas desarrolladas específicamente para el desarrollo e implementación de software de aplicaciones espaciales. Como parte del estado del arte, en la sección 5.2.1 se introdujeron los conceptos básicos de las tres tecnologías más relevantes en el sector: TASTE de la ESA, y F-Prime y cFS de la NASA. Con este trabajo, se alcanza el objetivo específico **OE1**.

En las etapas previas al desarrollo se estudiaron diversos proyectos como UPMSat-2 o FLP. También se revisó bibliografía relacionada con el desarrollo de software RTES. En ambos casos se extrajeron principios de diseño y distintos tipos de soluciones a problemas comunes en el sector. Por ejemplo, la representación de eventos mediante sucesos asíncronos, la arquitectura basada en capas jerárquicas que permiten abstraer la lógica funcional del hardware, o la separación de los datos de la lógica de negocio (impulsado por TASTE y el proceso ASSERT). Esto contribuyo al desarrollo ágil del OBSW de HERCCULES, cuya arquitectura está basada en las diversas soluciones propuestas por proyectos similares. Estas actividades contribuyen con el objetivo específico **OE2**.

En la sección 3.3 introduce los paradigmas CBD y MBD de forma general. En dicha sección se enuncian los beneficios y desventajas que ofrecen en el ámbito de los RTES. También se introduce el concepto de *modelo de componentes* que aúna las ventajas de ambas metodologías. Asimismo, se introdujo el método de diseño de HRT-HOOD y el proceso ASSERT, ambos destinados al desarrollo de RTES fundamentados en las metodologías MBD y CBD. Con estas contribuciones se cubre el objetivo específico **OE3**.

Los procesos de verificación y validación (capítulo 9) de los componentes software del sistema HERCCULES se han efectuado siguiendo los principios y recomendaciones de la campaña BEXUS-32. Estas están basadas en los estándares Cooperación Europea para la Normalización Espacial (ECSS) aplicados en el desarrollo de proyectos espaciales europeos. Estas actividades contribuyen al cumplimiento del objetivo específico **OE4**.

Finalmente en el capítulo 10 se presentan los resultados obtenidos tras haber seguido el plan de pruebas enunciados en el capítulo 9. Durante el procesos de validación se realizaron tanto

pruebas a nivel de componentes (unitarias y de integración) como de sistema. Con este trabajo se da por alcanzado el objetivo específico **OE5**.

Capítulo 12

Líneas Futuras

Como se ha podido ver a lo largo de este documento, la estación terrenal se ha generado de forma automática con TASTE en base a la especificación de las interfaces provistas (para la recepción de TM) y requeridas (para el envío de TCs). Esta ofrece una GUI funcional pero rudimentaria. Por lo tanto, una línea de trabajo futura para este proyecto implica la mejora de la GS a través de software diseñado específicamente para el sistema HERCCULES. Asimismo, a lo largo del desarrollo del OBSW con la herramienta TASTE, se han detectado algunos fallos y puntos de mejora con el conjunto de herramientas de TASTE, especialmente con la interfaz gráfica. De modo que el trabajo de realizado abre una segunda línea de trabajo que contribuir en la mejora de dicha herramienta.

Bibliografía

[1] M. Barr and A. Oram, *Programming Embedded Systems in C and C++*, 1st ed. USA: O'Reilly and Associates, Inc., 1998.

[2] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations*. Springer Berlin Heidelberg, Enero 2012.

[3] M. Macdonald and V. Badescu, *The International Handbook of Space Technology*, ser. Astronautical Engineering. Springer, Enero 2014.

[4] Universidad Politécnica de Madrid. (2021) Página web del proyecto HERCCULES. Accedido: 20-12-2022. [Online]. Available: https://blogs.upm.es/herccules/

[5] ESA Education. About REXUS/BEXUS. Accedido: 15-12-2022. [Online]. Available: https://www.esa.int/Education/Rexus_Bexus/About_REXUS_BEXUS

[6] O. Widell, O. Norberg, S. Kemi, L. Poromaa, O. Persson, A. Stamminger, and P. Turner, "REXUS BEXUS – a Swedish-German co-operation for university student experiments on rockets and balloons," Septiembre 2008.

[7] Instituto Universitario de Microgravedad "Ignacio Da Riva". Página web de IDR. Accedido 20-12-2022. [Online]. Available: https://www.idr.upm.es/index.php/es/

[8] Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos (STRAST). Página web de STRAST. Accedido: 20-12-2022. [Online]. Available: http://www.dit.upm.es/~str/index.html

[9] K. Dannenberg. (2022) The Organisers of the REXUS/BEXUS programme. Documento privado.

[10] Agencia Espacial Europea. (2021, diciembre) New student teams selected to fly experiments on REXUS and BEXUS. [Online]. Available: https://www.esa.int/Education/Rexus_Bexus/New_student_teams_selected_to_fly_experiments_on_REXUS_and_BEXUS

[11] A. F. Schmidt and SSC, *BEXUS user manual*, 8th ed., RX/BX Organisers, 06 2021. [Online]. Available: http://rexusbexus.net/wp-content/uploads/2021/06/BX_REF_BEXUS_User-Manual_v8_03Jun21.pdf

[12] D. González-Bárcena, A. Fernández-Soler, I. Pérez-Grande, and Ángel Sanz-Andrés, "Real data-based thermal environment definition for the ascent phase of polar-summer long duration balloon missions from esrange (sweden)," *Acta Astronautica*, vol. 170, pp. 235–250, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576520300357

[13] M. I. P. Grande, A. P. S. Andres, N. Bezdenejnykh, A. Farrahi, P. Barthol, and R. Meller, "Thermal control of sunrise, a balloon-borne solar telescope," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 225, no. 9, pp. 1037–1049, 2011. [Online]. Available: https://oa.upm.es/39517/

[14] HERCCULES team members, "Herccules's students experiment document," Universidad Politécnica de Madrid, Tech. Rep., 2022, accedido 25-12-2022. [Online]. Available: https://blogs.upm.es/herccules/wp-content/uploads/sites/979/2022/11/BX32_HERCCULES_SED_v3-0_31Aug22.pdf

[15] E. White, *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, Inc., 2011.

[16] I. Sommerville, *Software Engineering*, 10th ed. Pearson Education Limited, 2015.

[17] T. Henzinger and J. Sifakis, "The discipline of embedded systems design," *Computer*, vol. 40, pp. 32 – 40, 11 2007.

[18] A. Burns and A. Wellings, *Analysable real-time systems : programmed in Ada*, 4th ed. Massachusetts?: Adisson Wesley, 2016.

[19] M. K. Robert Oshana, *Software Engineering for Embedded Systems*, 2nd ed. Newnes, June 2019.

[20] A. Burns and A. J. Wellings, *Hrt-Hood: A Structured Design Method for Hard Real-Time Ada Systems*, 1995.

[21] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[22] M. Richards and N. Ford, *Fundamentals of Software Architecture. An Engineering Approach*. O'Reilly Media, Inc., 01 2020.

[23] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan and Claypool Publishers, 2017.

[24] P. Feiler and D. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[25] M. D'Alessandro, S. Mazzini, M. Di Natale, and G. Lipari, "Hrt-uml: a design method for hard real-time systems based on the uml notation," vol. 509, p. 33, 06 2002.

[26] B. Selic, "Using uml for modeling complex real-time systems." vol. 1474, 01 1998, pp. 250–260.

[27] E. Salazar, A. Alonso, M. A. de Miguel, and J. A. de la Puente, "A model-based framework for developing real-time safety ada systems," in *Reliable Software Technologies –*

Ada-Europe 2013, H. B. Keller, E. Plödereder, P. Dencker, and H. Klenk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 127–142.

[28] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-Oriented Analysis and Design with Applications, Third Edition*, 3rd ed. Addison-Wesley Professional, 2007.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[30] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st ed. USA: Prentice-Hall, Inc., 1979.

[31] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. USA: McGraw-Hill, Inc., 2009.

[32] J. Zalewski, "Real-time software architectures and design patterns: Fundamental concepts and their consequences," *IFAC Proceedings Volumes*, vol. 32, no. 1, pp. 1–13, 1999, 24th IFAC/IFIP Workshop on Real Time Programming WRTP 99, Schloss Dagstuhl, Germany, 30 May - 3 June. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017399585

[33] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems," *ACM SIGAda Ada Letters*, vol. XXIV, 04 2003.

[34] J. T. Taylor and W. T. Taylor, *Patterns in the Machine*. Apress, 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-6440-9

[35] J. Beningo, *Embedded Software Design*. Apress, 2022. [Online]. Available: https://doi.org/10.1007/978-1-4842-8279-3

[36] J. A. de la Puente, J. Garrido, E. Salazar, J. Zamorano, and A. Alonso, "Using internet-based technologies in a university satellite project," *IFAC-PapersOnLine*, vol. 48, no. 29, pp. 82–86, 2015, iFAC Workshop on Internet Based Control Education IBCE15. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896315024751

[37] B. Bätz, "Design and implementation of a framework for spacecraft flight software," 2020. [Online]. Available: http://elib.uni-stuttgart.de/handle/11682/11222

[38] SPACEBEL. Spacecraft On Board Software. Accedido el 6-01-2023. [Online]. Available: http://www.s3l.be/usr/files/di/fi/2/OnBoardSoftware2017_201812121418.pdf

[39] J. A. de la Puente. (1997) Diseño de sistemas de tiempo real: Introducción a HRT-HOOD. [Online]. Available: http://www.isa.uniovi.es/docencia/TiempoReal/Recursos/RTS-PL/HRT-HOOD.pdf

[40] Grupo de Investigación STRAT-UPM. Repositorio github de obdh_labs. [Online]. Available: https://github.com/STR-UPM/OBDH_LABS

[41] S. Mazzini, S. Puri, and A. Stragapede, "A uml2 profile for hard real-time modeling," vol. 602, p. 11, 07 2005.

[42] E. Conquet, "ASSERT: a step towards reliable and scientific system and software engineering." in *Embedded Real Time Software and Systems (ERTS2008)*, Toulouse, France, Jan. 2008. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02270318

[43] J. Garrido, J. A. de la Puente, J. Zamorano, M. A. de Miguel, and A. Alonso, "Timing analysis tools in a model-driven development environment," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5875–5880, 2017, 20th IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896317318244

[44] M. Panunzio and T. Vardanega, "A metamodel-driven process featuring advanced model-based timing analysis," in *Reliable Software Technologies – Ada Europe 2007*, N. Abdennadher and F. Kordon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 128–141.

[45] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *Advances in Object-Oriented Information Systems*, J.-M. Bruel and Z. Bellahsene, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 290–297.

[46] M. Melouk, Y. Rhazali, and H. Youssef, "An approach for transforming cim to pim up to psm in mda," *Procedia Computer Science*, vol. 170, pp. 869–874, 2020, the 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920305603

[47] J. A. de la Puente, J. F. Ruiz, and J. Zamorano, "An open ravenscar real-time kernel for gnat," in *Reliable Software Technologies Ada-Europe 2000*, H. B. Keller and E. Plödereder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 5–15.

[48] I. T. U. (ITU). Recommendations itu-t x.680 683: Abstract syntax notation one (asn.1). [Online]. Available: https://www.itu.int/rec/t-rec-x.680/en

[49] M. Perrotin, E. Conquet, J. Delange, A. Schiele, and T. Tsiodras, "Taste: A real-time software engineering tool-chain overview, status, and future," 01 2011, pp. 26–37.

[50] Maxime Perrotin. Kazoo. Accedido: 05-01-2023. [Online]. Available: https://taste.tuxfamily.org/wiki/index.php?title=Kazoo

[51] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications," in *Reliable Software Technologies – Ada-Europe 2009*, F. Kordon and Y. Kermarrec, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–250.

[52] Hugues, Jérome and Bechir, Zalila and Laurent, Paulet, "Middleware and tool suite for high integrity systems," in *Proceedings of RTSS-WiP*, 2006.

[53] E. J. Timmons. Core Flight System (cFS) Training - cFS Caelum. NASA. Accedido el 08-01-2023. [Online]. Available: https://ntrs.nasa.gov/citations/20210022378

[54] ——. Core Flight System (cFS) Background and Overview. NASA. Accedido el 08-01-2023. [Online]. Available: https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf

[55] NASA. (2023) Repositorio git de F Prime. Accedido: 04-01-2023. [Online]. Available: https://nasa.github.io/fprime/

[56] *F Prime: an open-source framework for small-scale flight software systems*. [Online]. Available: https://core.ac.uk/works/19055070

[57] J. Eickhoff, Ed., *A Combined Data and Power Management Infrastructure*. Springer Berlin Heidelberg, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-35557-8

[58] J. Zamorano, J. Garrido, J. Cubas, A. Alonso, and J. A. de la Puente, "The design and implementation of the upmsat-2 attitude control system," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 11 245–11 250, 2017, 20th IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S240589631732205X

[59] J. Eickhoff, B. Cook, P. Walker, S. Habinc, R. Witt, and H.-P. Roser, "Common Board Design for the OBC I/O Unit and The OBC CCSDS Unit of The Stuttgart University Satellite "Flying Laptop"," in *DASIA 2011 - Data Systems In Aerospace*, ser. ESA Special Publication, L. Ouwehand, Ed., vol. 694, Aug. 2011, p. 45.

[60] O. Casse, *SysML in Action with Cameo Systems Modeler*, O. Casse, Ed. Elsevier, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781785481710500049

[61] J. M. Redondo, P. B. Navarrete, and J. Malo, "HERCCULES. Revisión Crítica de Diseño – Subsistema de electrónica," Tech. Rep., 2022, documento privado.

[62] *TC74 – Tiny Serial Digital Thermal Sensor*, Microchip, 2002, accedido: 16-12-2022. [Online]. Available: https://www.mouser.es/datasheet/2/268/21462c-73653.pdf

[63] *INA226 High-Side or Low-Side Measurement, Bi-Directional Current and Power Monitor with I2C Compmatible Interface*, Texas Instruments, 06 2011, accedido: 16-12-2022, Rev. 15-08-2015. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina226.pdf

[64] *IRFZ44 – Power MOSFET*, VISHAY, 10 2021, accedido: 16-12-2022. [Online]. Available: https://www.mouser.es/datasheet/2/427/irfz44-1768892.pdf

[65] *Miniature SOP4-pin type with high capacity up to Max. 1.6A in the series*, Panasonic, 04 2022, accedido: 16-12-2022. [Online]. Available: https://www3.panasonic.biz/ac/e_download/control/relay/photomos/catalog/semi_eng_gu1a_aqy21_gs.pdf?f_cd=301102

[66] *ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs With Internal Reference, Oscillator, and Programmable Comparator*, Texas Instruments, 05 2009, accedido: 16-12-2022. [Online]. Available: https://www.ti.com/lit/ds/symlink/ads1114.pdf

[67] *USER MANUAL SR20. Secondary standard pyranometer*, Hukseflux, accedido: 16-12-2022. [Online]. Available: https://www.hukseflux.com/uploads/product-documents/SR20_manual_v1814.pdf

[68] *USER MANUAL IR20. Research grade pyrgeometer*, Hukseflux, accedido: 16-12-2022. [Online]. Available: https://www.hukseflux.com/uploads/product-documents/IR20_manual_v2005.pdf

[69] *LDE series – digital low differential pressure sensors*, First Sensor, accedido: 16-12-2022. [Online]. Available: https://www.mouser.mx/datasheet/2/313/DS_Standard-LDE_E_11815-607254.pdf

[70] *Lms41PD-03 series*, LED Microsensor NT, accedido: 16-12-2022. Rev.041017. [Online]. Available: http://lmsnt.com/datasheets/PD/Lms41PD/-03/Lms41PD-03_Series_rev041017.pdf

[71] *MS5611-01BA03 – Barometric Pressure Sensor, with stainless steel cap*, TE connectivity, 06 2017, accedido: 16-12-2022. [Online]. Available: https://www.mouser.es/datasheet/2/418/6/ENG_DS_MS5611_01BA03_B3-1134567.pdf

[72] MOUSER Electronics, *Página web del sensor MIRKOE-1032*, accedido 16-12-2022. [Online]. Available: https://www.mouser.es/ProductDetail/Mikroe/MIKROE-1032?qs=rzLl1kzNw5mzypK2dGSdmg%3D%3D

[73] *BNO055 – Intelligent 9-axis absolute orientation sensor*, Bosch Sensortec, 06 2016, accedido: 16-12-2022. [Online]. Available: https://www.mouser.es/datasheet/2/783/BST_BNO055_DS000-1509603.pdf

[74] Raspberry Pi (Trading) Ltd. (2019) Raspberry Pi 4 Model B datasheet. Accedido: 15-11-2022. [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf

[75] Ángel Grover Pérez Muñoz. (2021) Repositorio git del proyecto HERCCULES. Accedido: 03-01-2023. [Online]. Available: https://gitlab.com/AngelPerezM/herccules

[76] M. G. Harbour and C. D. Locke, "Tostadores y POSIX," vol. 129, 1997. [Online]. Available: https://www.istr.unican.es/publications/mgh-cdl-1997a.pdf

[77] Joan. (2022) Repositorio público de la biblioteca pigpio. Accedido: 03-01-2023. [Online]. Available: http://abyz.me.uk/rpi/pigpio/cif.html

[78] Kosma Moczek. (2022) Repositorio git de la biblioteca minmea. Accedido: 03-01-2023. [Online]. Available: https://github.com/kosma/minmea

[79] J. Eickhoff, Ed., *The FLP Microsatellite Platform*. Springer International Publishing, 2016. [Online]. Available: https://doi.org/10.1007%2F978-3-319-23503-5

[80] D. J. F. Miranda, M. Ferreira, F. Kucinskis, and D. McComas, "A comparative survey on flight software frameworks for 'new space' nanosatellite missions," *Journal of Aerospace Technology and Management*, Oct. 2019. [Online]. Available: https://doi.org/10.5028/jatm.v11.1081

[81] J. A. de la Puente, A. Alonso, J. Zamorano, J. Garrido, E. Salazar, and M. De Miguel, "Experience in spacecraft on-board software development," vol. 35, pp. 55–60, 03 2014.

[82] S. J. Young, "Real time languages, design and development," 1982.

[83] S. Iglesias Cofán and A. Formella, "12 - onboard software," in *Cubesat Handbook*, C. Cappelletti, S. Battistini, and B. K. Malphrus, Eds. Academic Press, 2021, pp. 237–250. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128178843000126

[84] M. Bradac, D. Perry, and L. Votta, "Prototyping a process monitoring experiment," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 774–784, 1994.

[85] *ECSS-E-ST-40C – Space engineering. Software*, ESA, 03 2009.

[86] *ECSS-E-HB-40A Software engineering handbook*, ESA, 12 2013.

[87] I. Latachi, T. Rachidi, M. Karim, and A. Hanafi, "Reusable and reliable flight-control software for a fail-safe and cost-efficient cubesat mission: Design and implementation," *Aerospace*, vol. 7, no. 10, 2020. [Online]. Available: https://www.mdpi.com/2226-4310/7/10/146

[88] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*, 2nd ed. Pearson Education, 2003.

[89] *ECSS-E-HB-40A Space Engineering: Verification*, ESA, 03 2009.

[90] J. A. de la Puente, J. Garrido, J. Zamorano, and A. Alonso, "Model-driven design of real-time software for an experimental satellite," *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 19, pp. 1592–1598, 01 2014.

Anexo A

Esquemas de diseño

A.1. Computador de a bordo

Uso	Pines GPIO		Uso
-	3v3	5v	-
I2C1 SDA	GPIO 2	5v	-
I2C1 SCL	GPIO 3	GND	-
I2C3 SDA	GPIO 4	GPIO 14	UART TX
-	GND	GPIO 15	UART RX
I2C4 SCL	GPIO 17	GPIO 18	I2C4 SDA
MUX TMU:0	GPIO 27	GND	-
MUX TMU:1	GPIO 22	GPIO 23	Downward EL Heater
-	3v3	GPIO 24	MUX SDPU bit 0
MUX TMU:2	GPIO 10	GND	-
PWM H1	GPIO 9	GPIO 25	MUX SDPU bit 1
PWM H2	GPIO 11	GPIO 8	MUX SDPU bit 2
-	GND	GPIO 7	NO
I2C0 SDA	GPIO 0	GPIO 1	I2C0 SCL
I2C3 SCL	GPIO 5	GND	-
PWM H3	GPIO 6	GPIO 12	NO
PWM H4	GPIO 13	GND	-
NO	GPIO 19	GPIO 16	MOSFET AL
MOSFET SDPU	GPIO 26	GPIO 20	MOSFET TMU
-	GND	GPIO 21	Upward EL Heater

Figura A.1: Uso de los pines GPIO para el OBC de HERCCULES.

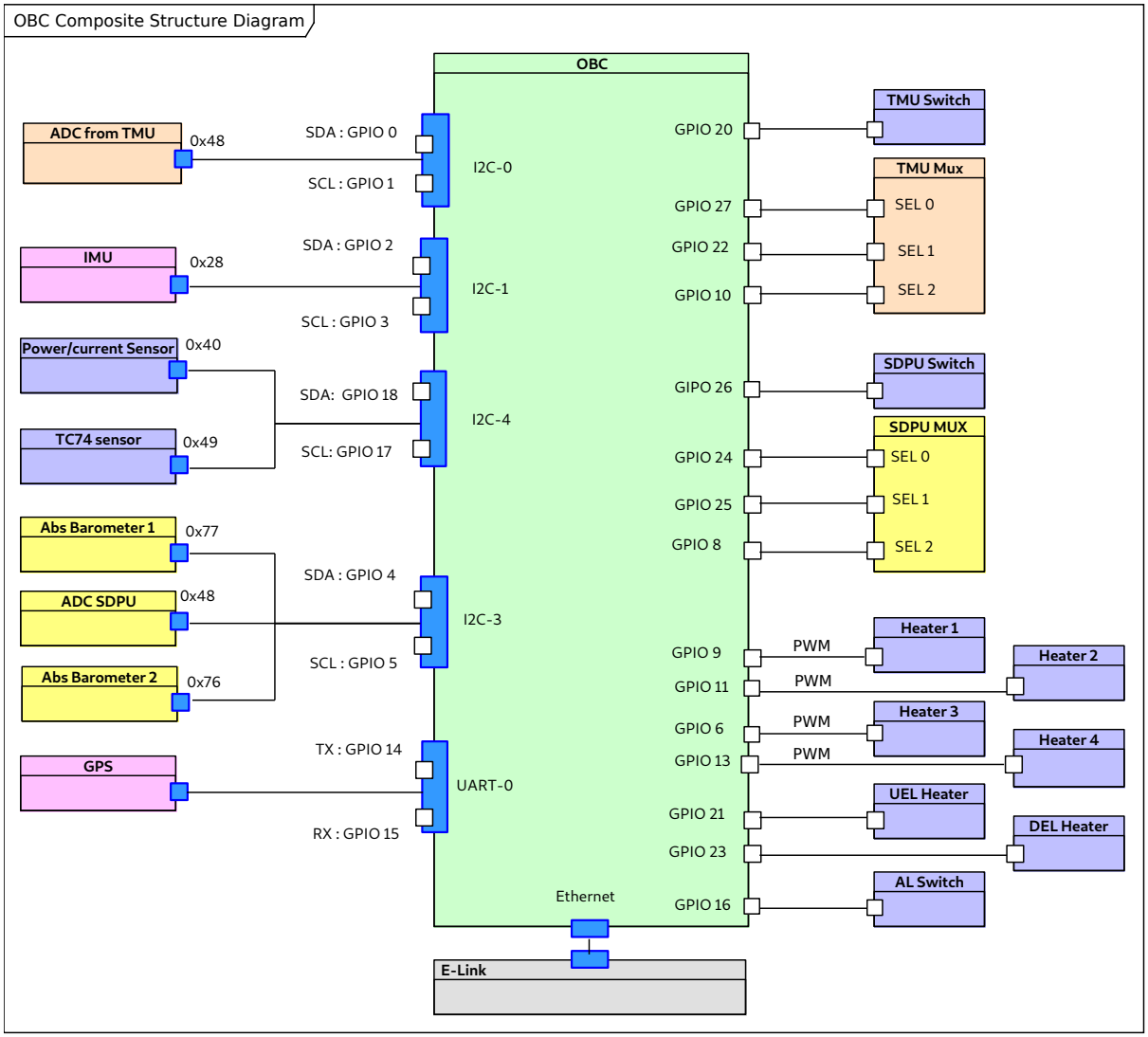


Figura A.2: Modelo UML de estructura compuesta del OBC y los periféricos.

A.2. Gestión del proyecto

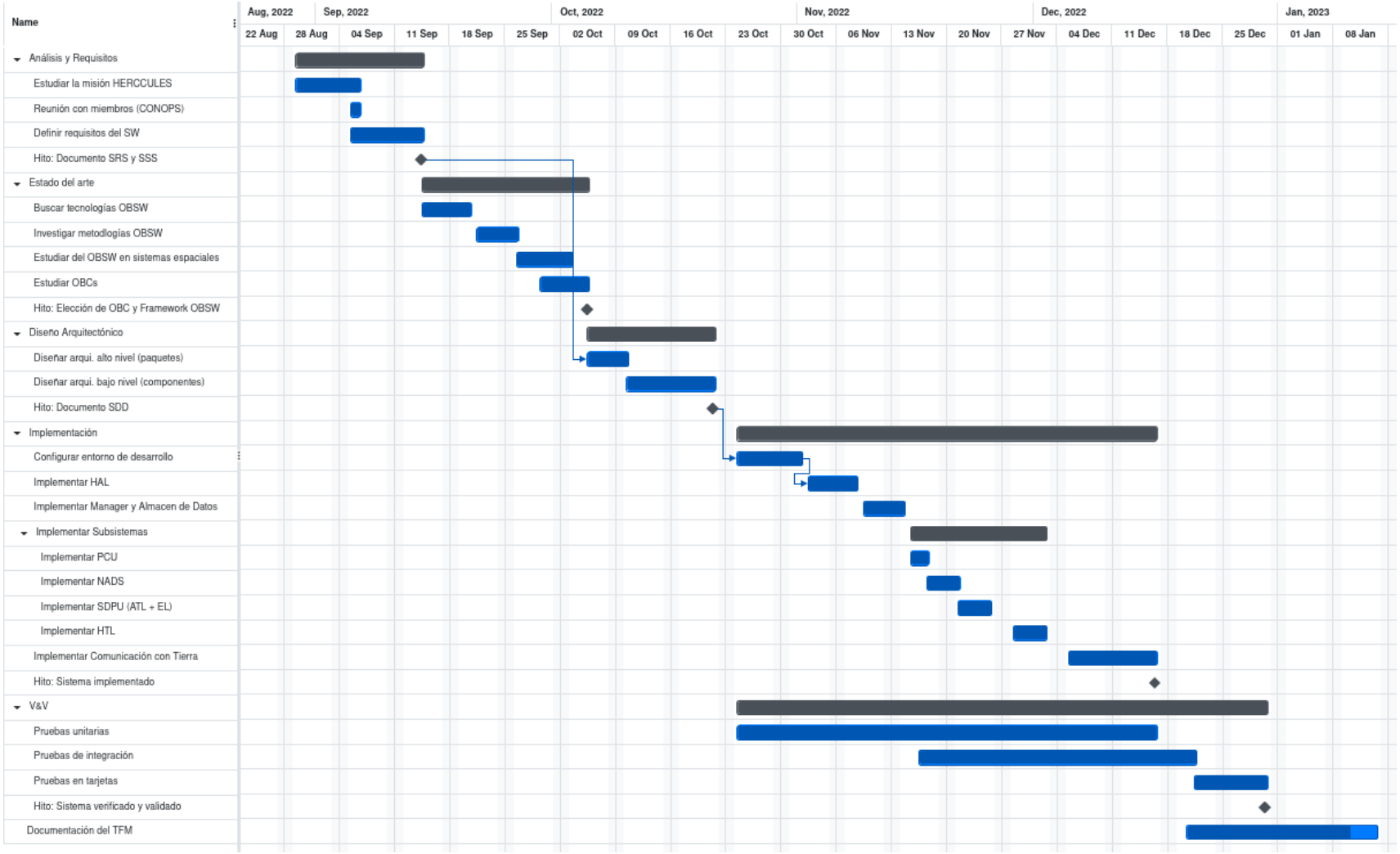


Figura A.3: Gráfico de Gantt detallado del proyecto HERCCULES

A.3. Modelos de la Vista de Interfaz de TASTE

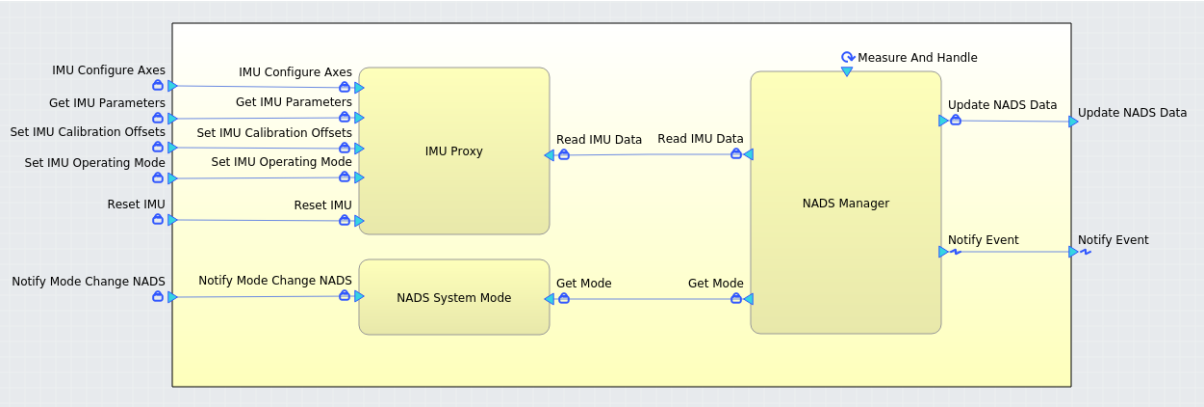


Figura A.4: Modelo IV del NADS.

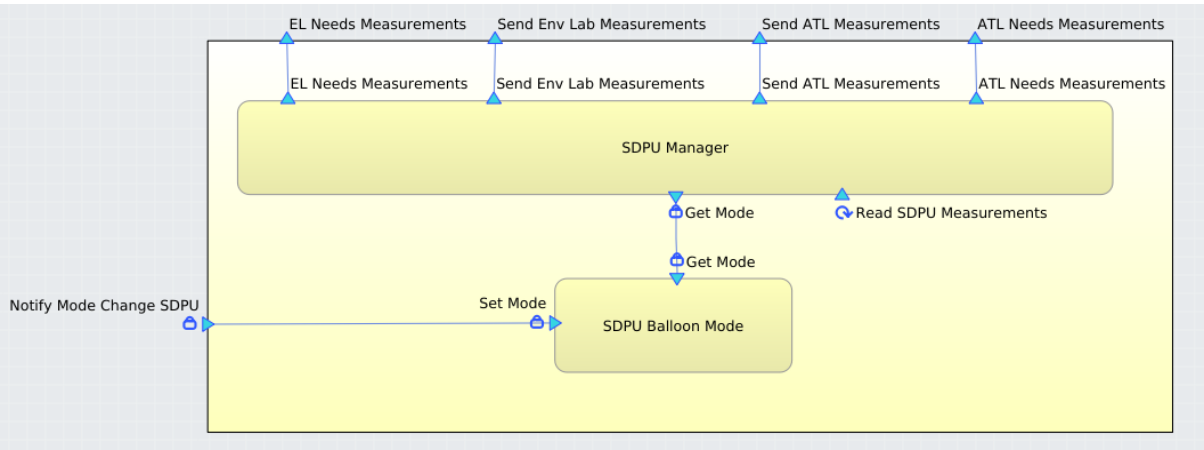


Figura A.5: Modelo IV del SDPU-Reader.

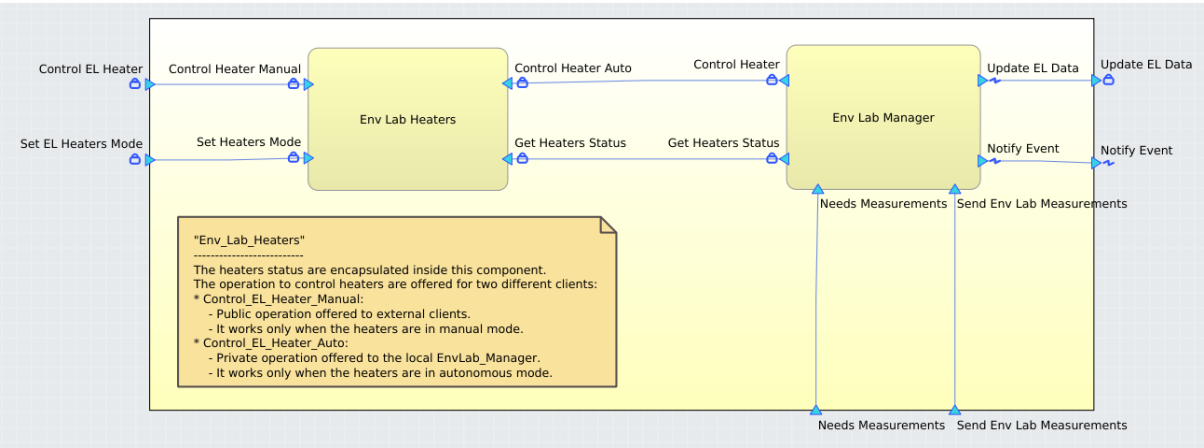


Figura A.6: Modelo IV del EL.

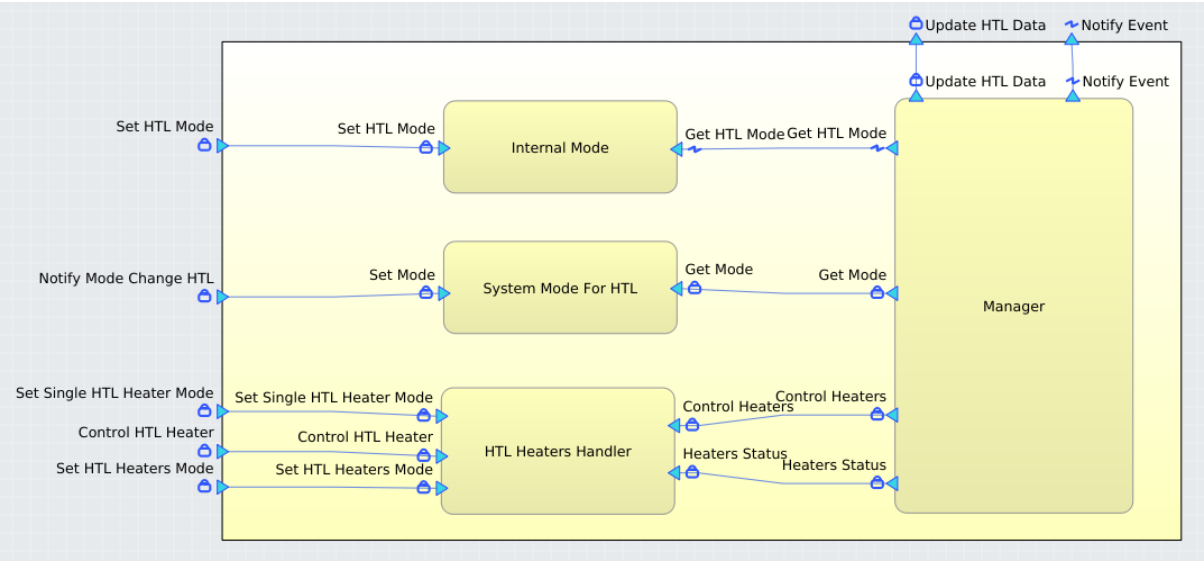


Figura A.7: Modelo IV del HTL.

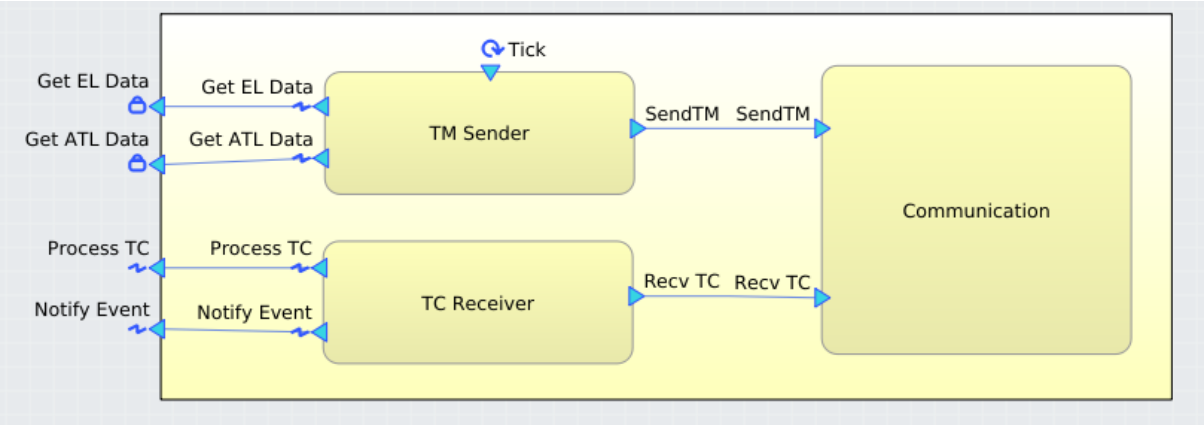


Figura A.8: Modelo IV de TM&TC.

Anexo B

Configuración del Sistema

B.1. Computador de a bordo

```
####################
# HERCCULES Config #
####################

# NOTE: Run dtoverlay -h i2c0 to learn about the i2c0 overlay.
# Execute "vcdbg log msg" to see logged messages.

dtdebug=on
# enables vcdbg log messages

# Bluetooth
############
dtoverlay=disable-bt
# disable Bluetooth

# I2C interfaces
#################
dtparam=i2c_arm=on
# enable the ARM's I2C I/F

dtparam=i2c_arm_baudrate=400000
# speed up I2C I/F frequency up to 400kHz

dtparam=i2c_vc=on
# enable I2C I/F reserved of VideoCore processor
# enable I2C-11 and I2C-12
# disables GPIO10 and GPIO11
#dtoverlay=i2c0,pins_0_1
# enable HW I2C-0 and use GPIOs 0 (SDA) and 1 (SCL), freq 400kHz:
# - ADC for TMU.


dtoverlay=i2c3,pins_4_5,baudrate=400000
# enable HW I2C-3 and use GPIOs 4 (SDA) and 5 (SCL), freq 400kHz:
# - Absolute barometer 1 & 2
# - ADC for SDPU.

dtoverlay=i2c-gpio,bus=1,i2c_gpio_sda=2,i2c_gpio_scl=3,i2c_gpio_delay_us=1
# enable SW I2C-1 and use GPIOs 2 (SDA) and 3 (SCL), freq ~200kHz:
# - BNO055 IMU requires bit-banging I2C beacuse of clock stretching!

dtoverlay=i2c-gpio,bus=4,i2c_gpio_sda=18,i2c_gpio_scl=17,i2c_gpio_delay_us=2
# enable SW I2C-4 and use GPIOs 18 (SDA) and 17 (SCL), freq ~100kHz:
```

```
# - INA226 power & current sensor.
# - TC74 temperature sensor. Limited to 100KHz

# SPI interfaces
################
# dtparam=spi=on
# enable 4 Chip Selects from SPI1 (overlay created by HERCCULES team)
# dtoverlay=spi1-4cs

# UART
######
enable_uart=1
# UART enabled

# Watchdog
```

Listado B.1: Fichero de configuración `/boot/config.txt` de las interfaces de la RPi usada en HERCCULES.

Anexo C

Código Fuente

C.1. Paquete HAL

En aras de la brevedad, los siguientes listados de código omiten las inclusiones de ficheros cabecera.

Listado C.1: Cabecera del componente SDPU.

```
namespace board_support::sdpu {

    bool initialize();

    void finalize();

    enum AnalogDevice : uint8_t {
        UP_PYRANOMETER = 0U,
        UP_PYRGEOMETER, DOWN_PYRANOMETER, DOWN_PYRGEOMETER,

        THERM_1 = 4U,
        THERM_2,  THERM_3, THERM_4,
        PT1000_1, PT1000_2,

        DIFF_BAROM_1 = 10U,
        DIFF_BAROM_2, DIFF_BAROM_3, DIFF_BAROM_4,

        PHOTODIODE_1 = 14U,
        PHOTODIODE_2, PHOTODIODE_3, PHOTODIODE_4
    };

    int16_t readRawFrom(AnalogDevice analogDevice);

    /**
    * In the SDPU it does not make sense to read  the four channels
    * of the ADC when the MUXs' select lines are in channels 6 or 7.
    * Hence, the upper limit for the channel is 5.
    */
    enum MUXChannel : uint8_t {
```

```
30          CH0 = 0U,
31          CH1 = 1U,
32          CH2 = 2U,
33          CH3 = 3U,
34          CH4 = 4U,
35          CH5 = 5U
36      };
37
38      std::array<int16_t, 4U> readAllADCChannels(MUXChannel muxChannel);
39
40      struct BarometerReading {
41          // Calculated with second order temperature compensation (see
                datasheet fig. 3):
42          float temperature_celsius;
43          float pressure_milliBar;
44
45          // Raw digital pressure and temperature data:
46          /// AKA D1 from pg. 8
47          int32_t temperature_raw;
48          /// AKA D2 from pg. 8
49          int32_t pressure_raw;
50      };
51
52      enum PressureSensor {
53          PS1 = 0U,
54          PS2 = 1U
55      };
56
57      bool getPressureFrom(PressureSensor ps, BarometerReading &reading);
58      bool resetPressureSensor(PressureSensor ps);
59
60      bool resetADC();
61
62      float pt1000ToCelsius(int16_t raw);
63      float thermistorToCelsius(int16_t raw);
64      float pyranometersTomV(int16_t raw);
65      float pyrgeometersTomV(int16_t raw);
66      float diffBarometersToPa(int16_t raw);
67  }
```

Listado C.2: Cabecera del componente PCU.

```
1   namespace board_support::pcu {
2       enum PowerSupplyLines : uint8_t {
3           SDPU_LINE = 0U,
4           TMU_LINE  = 1U,
5           AL_LINE   = 2U
6       };
7
8       enum EnvLabPWMHeaters : uint8_t {
9           UpwardsELHeater   = 0U,
10          DownwardsELHeater = 1U
11      };
12
```

```
13      enum HTLPWMHeater : uint8_t {
14          Heater1 = 0U,
15          Heater2 = 1U,
16          Heater3 = 2U,
17          Heater4 = 3U
18      };
19
20      bool initialize_switches();
21      bool initialize_sensors();
22
23      bool activatePowerSupplyFor(PowerSupplyLines line);
24      bool deactivatePowerSupplyFor(PowerSupplyLines line);
25
26      bool setEnvLabPWMDutyCycle(EnvLabPWMHeaters heater,
27                                 float pwmPercentage);
28      bool setHTLPWMDutyCycle(HTLPWMHeater heater, float pwmPercentage);
29
30      /**
31      * Configures the TC74 in Normal mode where TC74 consumes 200uA.
32      */
33      bool activateTC74();
34      bool readTC74(int8_t &temperature);
35      /**
36      * Configures the TC74 in Stand-By mode where the TC74 consumes 5uA.
37      */
38      bool deactivateTC74();
39
40      struct PowerAndCurrentData {
41          float busVoltage_V;
42          float shuntVoltage_V;
43          float power_W;
44          float current_A;
45      };
46
47      bool readPowerAndCurrent(PowerAndCurrentData & data);
48      bool resetPowerAndCurrentSensor();
49  }
```

Listado C.3: Implementación del componente PCU.

```
1   namespace {
2       using namespace equipment_handlers;
3       using namespace board_support::pcu;
4
5       TC74   tc74;
6       INA226 ina226;
7
8       std::array<PWM_GPIO,conf::NUMBER_OF_PWM_HEATERS> pwmHeaters;
9       std::array<PWM_GPIO,conf::NUMBER_OF_ENV_LAB_HEATERS> envLabHeaters;
10      std::array<Switch,conf::NUMBER_OF_POWER_SUPPLY_LINES> powerSupplyLines;
11
12      // -----------------
13      // Private functions
14      // -----------------
```

```
15
16      bool setupPwmHeaters() {
17          bool pwm1Success = pwmHeaters.at(HTLPWMHeater::Heater1)
18                               .initialize(conf::HEATER_1_PIN);
19          bool pwm2Success = pwmHeaters.at(HTLPWMHeater::Heater2)
20                               .initialize(conf::HEATER_2_PIN);
21          bool pwm3Success = pwmHeaters.at(HTLPWMHeater::Heater3)
22                               .initialize(conf::HEATER_3_PIN);
23          bool pwm4Success = pwmHeaters.at(HTLPWMHeater::Heater4)
24                               .initialize(conf::HEATER_4_PIN);
25
26          bool uelSuccess = envLabHeaters
27              .at(EnvLabPWMHeaters::UpwardsELHeater)
28              .initialize(conf::UP_ENV_LAB_HEATER_PIN);
29          bool delSuccess = envLabHeaters
30              .at(EnvLabPWMHeaters::DownwardsELHeater)
31              .initialize(conf::DOWN_ENV_LAB_HEATER_PIN);
32
33          bool pwmConfigSuccess = pwm1Success && pwm2Success
34                                && pwm3Success && pwm4Success
35                                && uelSuccess  && delSuccess;
36
37          for (auto & htlPwm : pwmHeaters) {
38              if (!htlPwm.setPWMFrequency(conf::pwmFrequency)) {
39                  pwmConfigSuccess = false;
40              }
41              if (!htlPwm.setDutyCycleRange(conf::pwmDutyCycleRange)) {
42                  pwmConfigSuccess = false;
43              }
44          }
45
46          for (auto & elPwm : envLabHeaters) {
47              if (!elPwm.setPWMFrequency(conf::pwmFrequency)) {
48                  pwmConfigSuccess = false;
49              }
50              if (!elPwm.setDutyCycleRange(conf::pwmDutyCycleRange)) {
51                  pwmConfigSuccess = false;
52              }
53          }
54          return pwmConfigSuccess;
55      }
56
57      bool initializeINA226() {
58          bool inaSuccess  = ina226.initialize(conf::inaI2CBusId, conf::
                inaI2CAddress);
59          if (inaSuccess) {
60              ina226.setMode(conf::inaMode);
61              ina226.setAverage(conf::average_samples);
62              ina226.setBusVoltageConversionTime(conf::conversionTime);
63              ina226.setShuntVoltageConversionTime(conf::conversionTime);
64              ina226.setMaxCurrentShunt(conf::maxCurrent_A,
65                                        conf::shunt_OHMs);
66          }
67          return inaSuccess;
68      }
69  }
```

```
70
71  namespace board_support::pcu {
72      bool initialize_switches() {
73          static bool switches_initialized = false;
74          if (!switches_initialized) {
75              bool power1Success = powerSupplyLines.at(AL_LINE).
76                  initialize(conf::AL_GPIO_PIN);
77              bool power2Success = powerSupplyLines.at(TMU_LINE)
78                  .initialize(conf::TMU_GPIO_PIN);
79              bool power3Success = powerSupplyLines.at(SDPU_LINE)
80                  .initialize(conf::SDPU_GPIO_PIN);
81
82              bool pwmSuccess = setupPwmHeaters();
83
84              switches_initialized = power1Success && power2Success
85                                  && power3Success && pwmSuccess;
86          }
87          return switches_initialized;
88      }
89
90      bool initialize_sensors() {
91          if (!bus_handlers::initialize()) {
92              return false;
93          }
94          static bool sensors_initialized = false;
95          if (!sensors_initialized) {
96              bool tc74Success = tc74.initialize(conf::tc74I2CBusId,
97              conf::tc74I2CAddress,
98              conf::tc74Mode);
99              bool inaSuccess  = initializeINA226();
100
101             sensors_initialized = tc74Success && inaSuccess;
102         }
103         return sensors_initialized;
104     }
105
106     bool activatePowerSupplyFor(PowerSupplyLines line) {
107         return powerSupplyLines.at(line).switchOn();
108     }
109
110     bool deactivatePowerSupplyFor(PowerSupplyLines line) {
111         return powerSupplyLines.at(line).switchOff();
112     }
113
114     bool setEnvLabPWMDutyCycle(EnvLabPWMHeaters heater,
115                                float pwmPercentage) {
116         return envLabHeaters.at(heater).setDutyCycle(pwmPercentage);
117     }
118
119     bool setHTLPWMDutyCycle(HTLPWMHeater heater, float pwmPercentage) {
120         return pwmHeaters.at(heater).setDutyCycle(pwmPercentage);
121     }
122
123     bool activateTC74() {
124         return tc74.setMode(TC74::Mode::Normal);
125     }
```

```
126
127     bool readTC74(int8_t &temperature) {
128         return tc74.readTemperature(temperature);
129     }
130
131     bool deactivateTC74() {
132         return tc74.setMode(TC74::Mode::Standby);
133     }
134
135     /**
136     * Note that requestData() clears the "conversion ready" register.
137     * Then, if we invoke requestData() before conversionReady(),
138     * and the elapsed time between these calls is lower than
139     * the conversion time, we will definitely get "false".
140     */
141     bool readPowerAndCurrent(PowerAndCurrentData & data) {
142         bool isReady = ina226.conversionsReady();
143         if (isReady) {
144             data.shuntVoltage_V = ina226.getShunt();
145             data.busVoltage_V   = ina226.getBusVoltage();
146             data.current_A      = ina226.getCurrent();
147             data.power_W        = ina226.getPower();
148             ina226.requestData();
149         }
150         return isReady;
151     }
152
153     bool resetPowerAndCurrentSensor() {
154         ina226.reset();
155         bool inaSuccess = initializeINA226();
156         return inaSuccess;
157     }
158 } // board_support::pcu
```

C.2. Vista de datos del proyecto HERCCULES

Listado C.4: Vista de datos para los subsistemas de HERCCULES escrita en ASN.1

```
1  DataTypes-Subsystems DEFINITIONS ::= BEGIN
2
3  IMPORTS // ..snip..
4
5  ------------------------------------------------
6  -- Template for the creation of subsystem data --
7  ------------------------------------------------
8  Subsystem-Data-Template {Mode-Type, Payload-Type} ::= SEQUENCE {
9          snapshot-time  Absolute-Time-Type,   -- specifies when the payload was measured
10         mission-time   Relative-Time-Type,   -- and its relative timestamp.
11         mode           Mode-Type,            -- operating mode of the subsystem
12         payload        Payload-Type          -- actual payload
13 }
14
15 -------------------------
16 -- Power & Control Unit --
17 -------------------------
18 PCU-Data ::= Subsystem-Data-Template {
19         PCU-Mode,
```

```
        SEQUENCE {
                sensor-data SEQUENCE {
                        power-watts          FLOAT32-Type,
                        current-amps         FLOAT32-Type,
                        voltage-bus-volts    FLOAT32-Type,
                        voltage-shunt-volts  FLOAT32-Type,
                        digital-temperature  INT8-Type
                },

                switches PCU-PS-Lines-Status
        }
}

Power-Supply-Line-ID ::= ENUMERATED {
        sdpu, -- SDPU: Sensor Data Processing Unit PCB
        tmu,  -- TMU:  Thermal Measurement Unit PCB
        atl   -- ATL:  Attitude Lab PCB
}

PCU-PS-Lines-Status ::= SEQUENCE { -- Power Supply Lines Status
        al-line    Switch-Status,
        tmu-line   Switch-Status,
        sdpu-line  Switch-Status
}

-----------------------
-- Heat Transfer Lab --
-----------------------
HTL-Data ::= Subsystem-Data-Template {
        HTL-Mode,
        SEQUENCE {
                thermistors  SEQUENCE (SIZE(28)) OF Analogue-Raw-Data,
                heaters      SEQUENCE (SIZE(4))  OF Heater-Power-Type
        }
}

HTL-Heater-ID ::= ENUMERATED {
        experiment1-heater,
        experiment2-heater,
        experiment3-heater,
        experiment4-heater
}

-----------------------
-- Environmental Lab --
-----------------------
EnvLab-Data ::= Subsystem-Data-Template {
        Env-Lab-Mode,
        SEQUENCE {
                upwards       EnvLab-Experiment-Data,
                downwards     EnvLab-Experiment-Data,
                pressure-data EnvLab-Pressure-Data
        }
}

-- Experiment's data:

EnvLab-Experiment-Data ::= SEQUENCE {
        -- Env Lab heater status, affects both pyra/pyrgeo-meter --
        heater Heater-Power-Type,

        -- Env Lab Experiments sensors readings --
        analogue-data EnvLab-Experiment-Data-Sensors
}

EnvLab-Experiment-Data-Sensors ::= SEQUENCE {
        pyranometer-reading      Analogue-Raw-Data,
        pyrgeometer-reading      Analogue-Raw-Data,
        pyranometer-temperature  Analogue-Raw-Data,
        pyrgeometer-temperature  Analogue-Raw-Data
```

```
}

-- Pressure data:

EnvLab-Pressure-Data ::= SEQUENCE {
        dif-barometers  SEQUENCE (SIZE(4)) OF Analogue-Raw-Data,
        abs-barometers  SEQUENCE (SIZE(2)) OF Absolute-Barometer
}

Absolute-Barometer ::= SEQUENCE {
        pressure-raw Analogue-Raw-Data,
        pressure-mbar FLOAT32-Type,

        temperature-raw Analogue-Raw-Data,
        temperature-celsius FLOAT32-Type
}

-- ID for EL's heaters.

EnvLab-Heater-ID ::= ENUMERATED {
        upwards-heater,
        downwards-heater
}

-------------------------------------------------
-- Navigational & Attitude Determination System --
-------------------------------------------------
-- Nads data = IMU + GPS

NADS-Data ::= Subsystem-Data-Template {
        NADS-Mode,
        SEQUENCE {
                imu IMU-Data,
                gps GPS-Data
        }
}

-- IMU & its auxiliary data types:
IMU-Data ::= SEQUENCE {
        sensors-data IMU-Sensors-Data,
        fusion-data  IMU-Fusion-Data,
        temperatures IMU-Temperatures
}

IMU-Sensors-Data ::= SEQUENCE {
        acceleration      Axis-Data,
        mag-field         Axis-Data,
        angular-velocity  Axis-Data
}

IMU-Fusion-Data ::= SEQUENCE {
        euler-orientation       Axis-Data,
        liner-acceleration      Axis-Data,
        gravity                 Axis-Data,
        quaternion-orientation  Quaternion-Data
}

IMU-Temperatures ::= SEQUENCE {
        temperature-accel INT8-Type,
        temperature-gyro  INT8-Type
}

Axis-Data ::= SEQUENCE {
        x FLOAT32-Type,
        y FLOAT32-Type,
        z FLOAT32-Type
}

Quaternion-Data ::= SEQUENCE {
        w FLOAT32-Type,
```

```
160           x FLOAT32-Type,
161           y FLOAT32-Type,
162           z FLOAT32-Type
163   }
164
165   ------------------
166   -- Attitude Lab --
167   ------------------
168   Att-Lab-Data ::= Subsystem-Data-Template {
169           Att-Lab-Mode,
170           Att-Lab-Data-Measurements
171   }
172
173   Att-Lab-Data-Measurements ::= SEQUENCE {
174           photodiodes  SEQUENCE (SIZE(4)) OF Analogue-Raw-Data,
175           thermistors  SEQUENCE (SIZE(2)) OF Analogue-Raw-Data
176   }
177
178   END
```