



Universidad Nacional de Educación a Distancia
Universidad Complutense de Madrid
Máster en Ingeniería de Sistemas y de Control

Simulador SMP para Algoritmos de Control de ADCS

SMP Simulator for ADCS Control Algorithms

Alumno: **Antonio Márquez Marín**
Director: **Segundo Esteban San Román**
Codirector: **Yago Isasi Parache**

Curso académico: 2017-2018
Convocatoria de defensa: Septiembre

Universidad Nacional de Educación a Distancia
Universidad Complutense de Madrid
Máster en Ingeniería de Sistemas y de Control

Simulador SMP para Algoritmos de Control de ADCS

SMP Simulator for ADCS Control Algorithms

Tipo B: Proyecto específico propuesto por el alumno

Alumno: **Antonio Márquez Marín**
Director: **Segundo Esteban San Román**
Codirector: **Yago Isasi Parache**



Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Firma del alumno

A handwritten signature in blue ink, appearing to read 'Antonio', with a long horizontal stroke underneath.

Resumen

En las empresas que trabajan en el diseño, verificación y validación de satélites, existen normalmente dos grupos de trabajo bien diferenciados. Por un lado, está el grupo dedicado a la modelización matemática de los subsistemas del satélite, así como de los algoritmos de control. Las personas que integran este grupo suelen trabajar con herramientas y lenguajes orientados al cálculo matemático, siendo Matlab la herramienta más comúnmente utilizada. El otro grupo lo integran personas con un perfil más orientado a la ingeniería del software, y se dedican a tareas como el desarrollo del software empujado del satélite, los modelos de los componentes del satélite, o el software de pruebas. Trabajan con lenguajes de propósito general como C o C++. Para la implementación de los modelos de los subsistemas que componen el satélite, los programadores de este grupo suelen utilizar estándares de codificación para que puedan ser ejecutados en simuladores que implementen estos estándares. De este modo, la simulación de modelos para los proyectos de la Agencia Espacial Europea se realiza utilizando el estándar SMP, definido por la propia ESA.

El objetivo principal de este trabajo fin de máster es el desarrollo de un sistema que permita integrar las herramientas de trabajo de ambos equipos mediante la implementación de un simulador SMP2 y su posterior integración en Matlab. Para verificar el correcto funcionamiento del sistema, se desarrolla un sistema ADCS de un satélite que utilice tanto modelos SMP2 como modelos matemáticos en Matlab y Simulink.

Palabras clave

SMP2, Simulación, Sistemas de Control, Ingeniería del Software, ADCS, C++, Matlab, Simulink, ECSS, Control basado en cuaterniones, Modelado, Ingeniería Espacial

Abstract

In companies involved in the design, verification and validation of satellites, there are usually two working groups. On the one hand, there is a group dedicated to the mathematical modeling of the satellite subsystems, as well as the control algorithms. The engineers involved in this group usually work with tools and languages oriented to mathematical calculation. Matlab is the most commonly used tool. The other group is composed of people with a profile more oriented towards software engineering, and they are dedicated to tasks such as the development of satellite embedded software, satellite component models, or test software. They work with general purpose languages such as C or C ++. For the implementation of the models of the subsystems that compose the satellite, the programmers of this group usually use coding standards so that they can be executed in simulators that implement these standards. In this way, the simulation of models for the projects of the European Space Agency are performed using the SMP standard, defined by the ESA itself.

The main objective of this thesis is the development of a system that allows integrating the work tools of both teams through the implementation of an SMP2 simulator and its subsequent integration in Matlab. To verify the correct functioning of the system, an ADCS system is developed for a satellite that uses both SMP2 models and mathematical models in Matlab and Simulink.

Keywords

SMP2, Simulation, Control Systems, Software Engineering, ADCS, C++, Matlab, Simulink, ECSS, Quaternion-Based Control, Modelling, Space Engineering

Definiciones y acrónimos

<i>Acrónimo</i>	Definición
<i>ADCS</i>	Attitude Determination and Control System
<i>SMP</i>	Simulation Model Portability
<i>ECSS</i>	European Cooperation for Space Standardization
<i>PD</i>	Proportional-Derivative
<i>ESA</i>	European Space Agency

Agradecimientos

A todos mis amigos, familiares, compañeros y profesores, entre los que destaco (en orden alfabético) a:

Antonio y Juani (mis padres)
Gabriela Moreno
Miguel Ángel Nuevo
Segundo Esteban (mi tutor)
Yago Isasi

Índice

CAPÍTULO 1: Introducción	15
1.1. Motivación	15
1.2. Objetivos	15
1.3. Fases de trabajo	16
1.4. Organización del documento.....	17
CAPÍTULO 2: Estado del arte	18
2.1. El estándar SMP2	18
2.1.1. Introducción	18
2.1.2. Arquitectura	18
2.1.3. Fases de ejecución.....	19
2.1.4. Tipos de interfaces	20
2.1.4.1. Objetos.....	20
2.1.4.2. Componentes	21
2.1.4.3. Mecanismos de componentes	21
2.1.4.4. Mecanismos de modelos.....	22
2.1.4.5. Interfaces de gestión	23
2.1.4.6. Entorno de simulación	23
2.1.4.7. Servicios	24
2.1.4.8. Excepciones	24
2.1.5. Estados	24
2.1.5.1. Estados del entorno de simulación	24
2.1.5.2. Estados de un modelo	27
2.2. Satélites	28
2.2.1. Definición	28
2.2.2. Tipos de satélites.....	28
2.2.2.1. Por órbita	28
2.2.2.2. Por misión.....	29
2.2.3. Subsistemas de un satélite.....	29
2.2.3.1. Subsistema de energía	30
2.2.3.2. Subsistema de control térmico.....	30
2.2.3.3. Subsistema de seguimiento, telemetría y telecomando	30
2.2.3.4. Subsistema de control de actitud y orientación	31

2.2.3.5.	Subsistema de propulsión	32
2.2.3.6.	Subsistema estructural	32
2.2.3.7.	Subsistema de procesamiento de datos	33
CAPÍTULO 3: Desarrollo		34
3.1.	Componentes del sistema	34
3.2.	Entorno de simulación	35
3.2.1.	Clases principales	36
3.2.2.	Funcionamiento	36
3.2.2.1.	Carga de modelos	36
3.2.2.2.	Publicación de campos	37
3.2.2.3.	Publicación de operaciones	38
3.2.2.4.	Definición de eventos programados	38
3.2.2.5.	Ejecutar o parar simulador	38
3.2.2.6.	Envío de mensajes	39
3.2.2.7.	Registro de acciones a eventos	40
3.2.2.8.	Registro de arrays	40
3.3.	Intérprete SMP2 (línea de comandos)	41
3.3.1.	Diagrama de clases	41
3.3.2.	Clases principales	43
3.3.3.	Esquema general de funcionamiento	43
3.3.4.	Funcionalidades	44
3.3.4.1.	Cargar ensamblado	44
3.3.4.2.	Ejecutar y parar el simulador	47
3.3.4.3.	Consultar modelos cargados	47
3.3.4.4.	Consultar campos y operaciones publicadas	48
3.3.4.5.	Consultar tiempos de simulación	48
3.3.4.6.	Modificar tiempo de época y de misión	49
3.3.4.7.	Consultar y modificar el valor de un campo publicado	49
3.3.4.8.	Ejecutar una operación publicada	50
3.4.	Modelo SMP2 de ejemplo	50
3.4.1.	Descripción del modelo	50
3.4.1.1.	Campos publicados	51
3.4.1.2.	Métodos publicados	51
3.4.1.3.	Puntos de entrada	51
3.4.1.4.	Ejecución de puntos de entrada	52

3.4.2.	Diagrama de clases	52
3.4.3.	Interfaces.....	53
3.4.3.1.	Métodos de la interfaz IModel.....	53
3.4.3.2.	Métodos de la interfaz IDynamicInvocation	53
3.5.	Integración de entorno de simulación en Matlab	54
3.5.1.	Implementación de interfaz externa en C++	54
3.5.2.	Implementación de intérprete SMP2 en Matlab	56
3.6.	Implementación de sistema ADCS	58
3.6.1.	Introducción	58
3.6.2.	Componentes del sistema.....	58
3.6.3.	Modelos SMP2 de módulo ADCS.....	60
3.6.3.1.	Gyroscope	60
3.6.3.2.	Star Tracker	60
3.6.3.3.	Reaction wheel	61
3.6.3.4.	ADCS Controller	61
3.6.4.	Modelo dinámico en Simulink.....	64
3.6.4.1.	Módulo de gravedad	64
3.6.4.2.	Módulo traslacional	65
3.6.4.3.	Módulo rotacional.....	66
3.6.5.	Algoritmo de control PD.....	67
3.6.5.1.	Error entre cuaterniones.....	67
3.6.5.2.	Parte proporcional del controlador	68
3.6.5.3.	Parte derivativa del controlador.....	68
3.6.5.4.	Ley de control PD.....	69
3.6.5.5.	Lazo de control implementado	69
3.6.6.	Implementación de GUI con el controlador ADCS	70
3.6.7.	Resultados	71
CAPÍTULO 4: Conclusiones y líneas futuras.....		75
Bibliografía		78
Anexo 1. Diagramas de clases del entorno de simulación implementado.....		80
1.	GenericComponent.....	80
2.	Simulator	81
3.	Publication.....	82
4.	TypeRegistry	83
5.	Logger	84

6.	Scheduler.....	85
7.	TimeKeeper.....	86
8.	EventManager	87
Anexo 2. Principales métodos implementados de interfaces SMP2.....		88
1.	IComponent.....	88
2.	Interfaz ISimulator	88
3.	Interfaz IPublication.....	89
4.	Interfaz ITypeRegistry	91
5.	Interfaz Logger.....	91
6.	Interfaz IScheduler	91
7.	Interfaz ITimeKeeper	92
8.	Interfaz IEventManager.....	92
Anexo 3. Casos de uso y diagramas de secuencia de intérprete SMP2		93
1.	Cargar ensamblado	93
2.	Ejecutar simulación	94
3.	Parar simulación.....	95
4.	Consultar modelos cargados.....	96
5.	Consultar campos publicados de un modelo	96
6.	Consultar operaciones publicadas de un modelo	97
7.	Consultar tiempos de simulación	98
8.	Consultar valor de un campo publicado.....	99
9.	Consultar valor de un array publicado	100
10.	Modificar valor de un campo publicado	101
11.	Modificar valor de un array publicado	102
12.	Modificar tiempo de época.....	103
13.	Modificar tiempo de misión	104
14.	Ejecutar operación publicada	105
Anexo 4. Ficheros del código fuente		107
Anexo 5: Entorno de desarrollo		110
Anexo 6: Contenido de Makefile		112

Tabla de figuras

Figura 1. Roles de ingenieros de control e ingenieros del software.....	16
Figura 2. Arquitectura SMP2.....	19
Figura 3. Diagrama de interfaces de objetos.....	21
Figura 4. Diagrama de interfaces de componentes.	21
Figura 5. Diagrama de interfaces de mecanismos de componentes.	22
Figura 6. Diagrama de interfaces de mecanismos de modelos.	22
Figura 7. Diagrama de interfaces de gestión.....	23
Figura 8. Diagrama de interfaces de entorno de simulación.	23
Figura 9. Diagrama de interfaces de servicios.	24
Figura 10. Diagrama de estados del entorno de simulación.	27
Figura 11. Diagrama de estados de un modelo.	27
Figura 12. Esquema general del sistema implementado.	34
Figura 13. Carga de modelos en el simulador.....	37
Figura 14. Publicación de campos de un modelo.	37
Figura 15. Planificación y modificación de eventos.	38
Figura 16. Ejecución de simulador.	39
Figura 17. Envío de mensajes de log.	40
Figura 18. Emisión de eventos.....	40
Figura 19. Diagrama de clases del intérprete SMP2.....	42
Figura 20. Esquema de funcionamiento de intérprete SMP2.....	43
Figura 21. Ejemplo de uso de intérprete SMP2.	44
Figura 22. Roles y uso de ensamblados de modelos en el simulador.	45
Figura 23. Carga de un ensamblado por línea de comandos en intérprete SMP2.....	46
Figura 24. Carga de un ensamblado de modelos desde la clase Controller.	46
Figura 25. Ejecutar y parar la simulación en el intérprete SMP2.	47
Figura 26. Ejecutar varias operaciones de consulta en el intérprete SMP2.	48
Figura 27. Consultar y modificar los tiempos de simulación en el intérprete SMP2.....	49
Figura 28. Ejecución de una operación publicada desde la clase Controller.	50
Figura 29. Diagrama de clases del modelo SMP2 de ejemplo.....	52
Figura 30. Aplicación GUI con intérprete SMP2 en Matlab.	57
Figura 31. Diagrama de sistema de control ADCS implementado.	59
Figura 32. Diagrama de herencias de la clase genérica Adcs::Model.....	60
Figura 33. Pasos de control implementados por el controlador ADCS..	61
Figura 34. Diagrama de clases del modelo SMP2 para el controlador ADCS.....	63
Figura 35. Diagrama en simulink del modelo dinámico del satélite.....	64
Figura 36. Modelo traslacional del satélite.	66
Figura 37. Modelo rotacional del satélite.	67
Figura 38. Lazo de control PD para control de orientación basado en cuaterniones.	69
Figura 39. Aplicación GUI con controlador ADCS.....	71
Figura 40. Resultados del control PD con $K_p=1$ y $K_d=0.25$	72
Figura 41. Resultados del control PD con $K_p=0.5$ y $K_d=0.5$	73
Figura 42. Resultados del control PD con $K_p=0.25$ y $K_d=1$	73
Figura 43. Resultados del control PD con $K_p=0.1$ y $K_d=1$	73
Figura 44. Resultados del control PD con $K_p=0.5$ y $K_d=1$ y segundo ángulo de de prueba.	74
Figura 45. Resultados del control PD con $K_p=0.5$ y $K_d=1$ y tercer ángulo de prueba.	74
Figura 46. Diagrama de herencia de la clase GenericComponent.	80
Figura 47. Diagrama de clases de GenericComponent.	80
Figura 48. Diagrama de herencia de la clase Simulator.....	81
Figura 49. Diagrama de clases de Simulator.	81
Figura 50. Diagrama de herencia de la clase Publication.	82
Figura 51. Diagrama de clases de Publication.	82
Figura 52. Diagrama de herencia de la clase TypeRegistry.....	83
Figura 53. Diagrama de clases de TypeRegistry.	83
Figura 54. Diagrama de herencia de la clase Logger.	84

Figura 55. Diagrama de clases de Logger.....	84
Figura 56. Diagrama de herencia de la clase Scheduler.	85
Figura 57. Diagrama de clases de Scheduler.	85
Figura 58. Diagrama de herencia de la clase TimeKeeper.....	86
Figura 59. Diagrama de clases de TimeKeeper.	86
Figura 60. Diagrama de herencia de EventManager.....	87
Figura 61. Diagrama de clases de EventManager.....	87
Figura 62. Diagrama de secuencia para CU-001.	94
Figura 63. Diagrama de secuencia para CU-002.	95
Figura 64. Diagrama de secuencia para CU-003.	95
Figura 65. Diagrama de secuencia para CU-004.	96
Figura 66. Diagrama de secuencia para CU-005.	97
Figura 67. Diagrama de secuencia para CU-006.	98
Figura 68. Diagrama de secuencia para CU-007.	99
Figura 69. Diagrama de secuencia para CU-008.	100
Figura 70. Diagrama de secuencia para CU-009.	101
Figura 71. Diagrama de secuencia para CU-010.	102
Figura 72. Diagrama de secuencia para CU-011.	103
Figura 73. Diagrama de secuencia para CU-012.	104
Figura 74. Diagrama de secuencia para CU-013.	105
Figura 75. Diagrama de secuencia para CU-014.	106
Figura 76. Entorno de desarrollo en CentOS 7.	110
Figura 77. Entorno de desarrollo en Windows 10.	111

CAPÍTULO 1: Introducción

1.1. Motivación

En los proyectos espaciales, tales como la construcción de satélites, la simulación basada en modelos juega un papel muy importante, debido a la complejidad de los módulos a diseñar y a la frecuente falta de disponibilidad del sistema real. Implementando los modelos del sistema a construir, es posible trabajar y desarrollar las pruebas de validación directamente sobre ellos, además de poder analizar las variaciones en los componentes del sistema.

En este contexto, la Agencia Espacial Europea y varias empresas relacionadas con la industria espacial desarrollaron el estándar SMP (*Simulation Model Portability*), que define la interfaz entre los modelos simulados y las infraestructuras de simulación, con el fin de facilitar la reutilización y portabilidad de estos modelos independientemente de la plataforma utilizada para su desarrollo.

Los modelos que implementan el estándar SMP suelen ser desarrollados en C++. No obstante, Matlab es la herramienta más utilizada por los arquitectos de modelado dinámico y de control de satélites. Esto es debido principalmente a su paradigma más orientado al análisis matemático que permite la creación y el análisis de modelos dinámicos de forma más intuitiva para un ingeniero de sistemas. Posteriormente, el código generado en Matlab puede ser convertido a C/C++, y su adaptación al estándar SMP2 puede ser realizada de forma manual o utilizando herramientas como MOSAIC.

No obstante, las pruebas de los modelos en el motor del simulador deberán ser entonces realizadas fuera del entorno Matlab. La idea de este proyecto surge de la idea de interconectar este motor dentro de Matlab, con el propósito de ampliar las posibilidades de análisis del entorno de simulación y de facilitar la tarea a los desarrolladores de Matlab que necesiten analizar como se comportan los modelos en tiempo real.

1.2. Objetivos

La finalidad de este proyecto es el desarrollo de un simulador SMP2 en C++ que implemente los requisitos más importantes del estándar, y su integración en Matlab. También se diseñará un módulo de control de orientación y actitud (ADCS) de un satélite que será implementado en SMP2 y Simulink, y ejecutado con el motor integrado en Matlab. Implementando este sistema se cumplirá el principal objetivo de este proyecto, que es acercar las tareas del ingeniero de control al ingeniero del software. Se justifica la importancia de este objetivo a continuación.

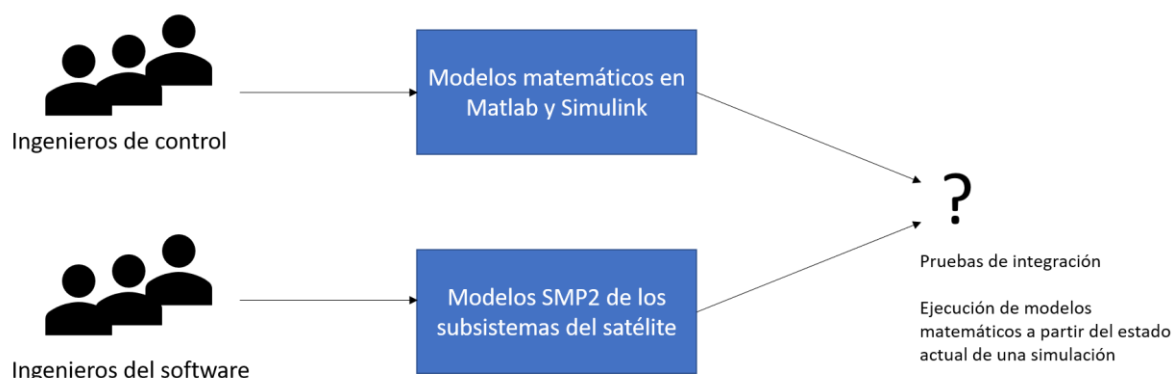


Figura 1. Roles de ingenieros de control e ingenieros del software.

En la Figura 1, se puede apreciar a grandes rasgos los resultados del trabajo de grupo de ingenieros de control y de ingenieros de software como parte del desarrollo de un sistema de simulación de un satélite. Los ingenieros de control se centran en la definición de modelos matemáticos que definan el comportamiento del satélite, así como de los algoritmos de control de actitud que deberán ser implementados en el software de a bordo del satélite. La tarea de los ingenieros del software se centra en la implementación de los modelos SMP2 de los sistemas del satélite a partir de la especificación de sus interfaces y protocolos. Cuando se desea combinar el trabajo de ambos equipos y realizar pruebas que utilicen ambos módulos, nos encontramos con un problema. ¿Se deben convertir los modelos implementados en Matlab al estándar SMP2? ¿O leer de alguna manera los resultados devueltos por Matlab desde los modelos SMP2

Es aquí donde cobra sentido el objetivo de nuestro proyecto. Se propone un sistema que sea capaz de integrar un simulador SMP2 en el entorno de Matlab, con el fin de agilizar las pruebas de verificación y validación del satélite.

Aparte de este objetivo principal, otra de las metas a conseguir es la implementación de un pequeño sistema ADCS que utilice componentes codificados en SMP2 y en Matlab. Con esto, demostraremos que la integración planteada es factible.

1.3. Fases de trabajo

Se describen brevemente a continuación las fases de trabajo seguidas para el desarrollo de este proyecto.

- Estudio de la problemática (estándar SMP2, modelado dinámico, algoritmos de control en módulos ADCS, etc.).
- Estudio y familiarización de todas las herramientas a utilizar necesarias para la implementación del sistema.

- Programación del motor de simulación SMP2 en C++.
- Programación de modelo SMP2 que cubra los requisitos del simulador.
- Programación de un intérprete del simulador en C++.
- Desarrollo de la integración en MATLAB.
- Programación de una GUI en MATLAB para el uso del simulador.
- Programación de modelos SMP2 de un módulo ADCS.
- Diseño del modelo dinámico del satélite en Simulink.
- Programación de algoritmo de control para la orientación del satélite en SMP2.
- Programación de aplicación en MATLAB que ejecute el motor SMP2 con el módulo ADCS y los modelos dinámicos.
- Análisis de resultados del sistema ADCS implementado.

1.4. Organización del documento

La estructura de esta memoria se ha dividido en los siguientes capítulos.

- **Introducción.** Capítulo donde se presenta el problema a resolver.
- **Estado del arte.** Capítulo donde se expone las bases teóricas de los conceptos relacionados con el ámbito del proyecto. Se explica en profundidad la arquitectura del estándar SMP2, así como la necesidad de los simuladores en proyecto espaciales. También se incluye un apartado a modo de introducción sobre satélites.
- **Desarrollo.** Capítulo donde se detalla el proceso seguido para la implementación del sistema en todas sus fases.
- **Conclusiones.** Capítulo donde se exponen algunas conclusiones relacionadas con la realización de este proyecto. También se proponen posibles futuras mejoras al sistema implementado.

CAPÍTULO 2: Estado del arte

2.1. El estándar SMP2

2.1.1. Introducción

Uno de los aspectos clave en muchos de los proyectos desarrollados en el sector espacial es la implementación de simuladores y modelos dinámicos preparados para ser ejecutados en tiempo real. Esto tiene un amplio conjunto de aplicaciones, como por ejemplo, el análisis de la dinámica de los mismos, el testeo de hardware sobre entorno simulado o el desarrollo de sistemas de entrenamiento.

Durante muchos años, la Agencia Espacial Europea estuvo implicada en el proceso de varios de estos proyectos. Uno de los principales problemas que se detectaron durante el ciclo de desarrollo fue que cada departamento trabajaba sobre diferentes entornos y lenguajes de programación. Esto daba como resultado un amplio número de simuladores para cada uno de los proyectos, lo que hacía difícil la integración entre ellos.

Con el fin de corregir este problema, y como se explicó brevemente en la introducción de esta memoria, se desarrolló el estándar SMP (*Simulation Portability Models*) por la propia Agencia Espacial Europea y otros organismos relacionados con el sector espacial. El principal objetivo de SMP era la definición de una manera común para el desarrollo de modelos y simuladores para que, de este modo, los modelos implementados según este estándar pudieran ser ejecutados en cualquier entorno de simulación SMP sin necesidad de modificar su código fuente.

La primera versión de SMP fue llamada SMP1 o también SMI, y se lanzó en 2001. Con esta versión, se desarrollaron con éxito muchos proyectos relacionados con la simulación espacial. No obstante, algunas de las limitaciones en esta versión impulsaron el desarrollo de SMP2, la segunda versión del estándar. En este proyecto se trabajará sobre esta versión, concretamente con SMP2 v1.2.

Actualmente, se está trabajando para la integrar SMP2 en la serie de estándares de ECSS (*European Committee for Space Standardisation*). El código ECSS para el nuevo estándar es E-TM-40-07¹.

2.1.2. Arquitectura

La arquitectura SMP2 consta principalmente de dos componentes, tal y como se muestra en la Figura 2. En nuestro proyecto, implementaremos ambos elementos de ambos componentes (simulador y modelos SMP2).

¹ El Código ECSS para el estándar de software relacionado con la industria espacial es E-TM-40.

- Una simulación que engloba una serie de instancias de modelos SMP2 con un comportamiento específico.
- Un entorno de simulación que proporciona (al menos) los siguientes servicios.
 - **Logger.** Permite gestionar y mostrar los distintos tipos de mensajes mandados por las diferentes componentes de simulación, ya sean servicios o modelos.
 - **Scheduler.** Se encarga de ejecutar las tareas periódicas (llamadas puntos de entradas) definidas en los modelos, de acuerdo con el tiempo de simulación.
 - **Time Keeper.** Almacena y gestiona los diferentes tipos de tiempos definidos en SMP2.
 - Tiempo de simulación. Representa el tiempo total transcurrido desde el comienzo de la simulación.
 - Tiempo de época. Es relativo al tiempo de simulación.
 - Tiempo de misión. Es relativo al tiempo de época.
 - Tiempo Zulú. No depende del tiempo de simulación, sino del reloj interno del computador.
 - **Event Manager.** Proporciona los mecanismos necesarios para la gestión de eventos asíncronos.

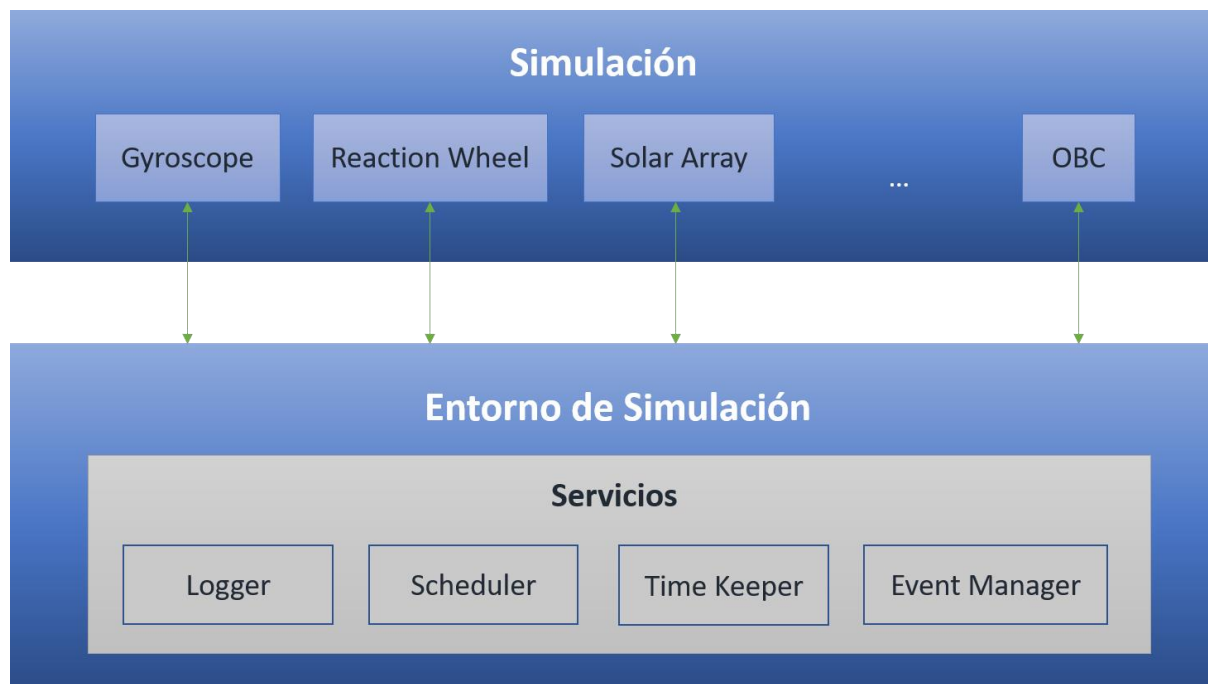


Figura 2. Arquitectura SMP2.

2.1.3. Fases de ejecución

Se explican brevemente a continuación, las fases de ejecución en SMP2.

- **Setup**
 - Las instancias de los modelos son creadas.
 - Se publican los campos y operaciones de cada modelo, llamando al método *publish*.
 - Los modelos son configurados, llamando al método *configure*.
 - Los modelos son conectados con otros modelos o servicios, llamando al método *connect*.
 - Se realiza la inicialización personalizada de cada modelo.
- **Execution**
 - El servicio *Scheduler* ejecuta los puntos de entrada de acuerdo al tiempo de simulación.
 - Los modelos interactúan unos con otros.
 - Los modelos llaman a los servicios del entorno de simulación.
- **Termination**
 - Se liberan los recursos utilizados por cada uno de los modelos.
 - Se para y cierra el entorno de simulación.

2.1.4. Tipos de interfaces

Todos los elementos de SMP2 están basados en interfaces. Dependiendo del rol del desarrollador SMP2, será necesario implementar unas interfaces u otras; así, el programador de modelos debe implementar interfaces como *IModel*, mientras que algunas de las interfaces a implementar por el programador de simuladores serían *ISimulator* o *IPublication*. En los siguientes apartados, se explican de forma breve algunos de los tipos de interfaces en SMP2.

2.1.4.1. Objetos

Aunque la mayoría de los elementos SMP2 son componentes, algunos de ellos exponen instancias que derivan directamente de *IObject*, como por ejemplo campos, puntos de entrada, eventos, o errores. Los objetos almacenan básicamente un nombre y una descripción.

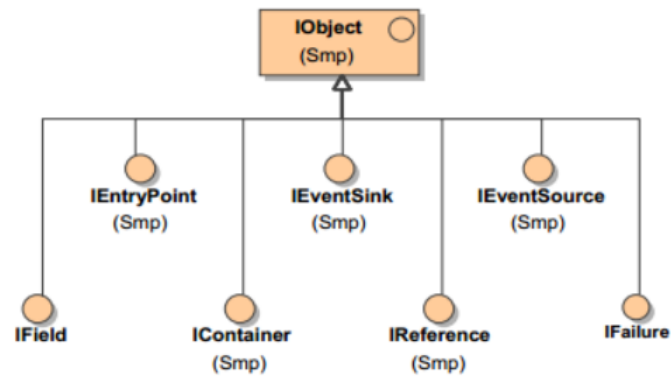


Figura 3. Diagrama de interfaces de objetos.

2.1.4.2. Componentes

Esta interfaz permite procesar todos los componentes SMP2 de forma similar. Las tres interfaces más importantes que derivan de la interfaz *IComponent* son *IModel* (para los modelos), *IService* (para los servicios) e *ISimulator* (para el entorno de simulación).

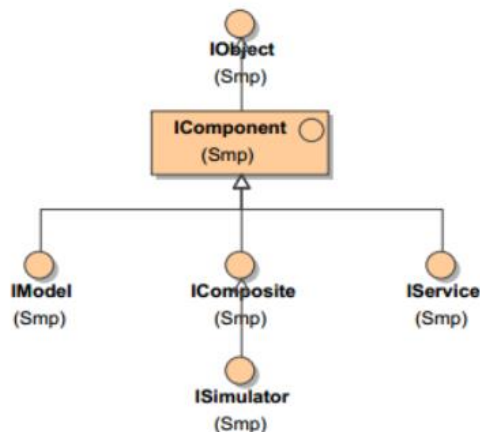


Figura 4. Diagrama de interfaces de componentes.

2.1.4.3. Mecanismos de componentes

El estándar SMP2 define algunos mecanismos para ampliar la funcionalidad de los componentes, como por ejemplo, el mecanismo de agregación (*IAggregate*), de composición (*IComposite*) o de invocación dinámica de métodos (*IDynamicInvocation*).

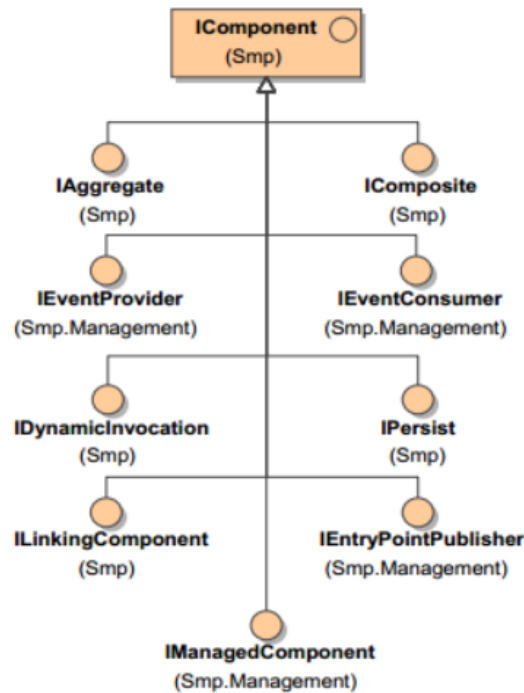


Figura 5. Diagrama de interfaces de mecanismos de componentes.

2.1.4.4. Mecanismos de modelos

Mientras que para el desarrollo de un modelo basta con una implementación mínima de la interfaz *IModel*, esto no permite usar la mayoría de las características del estándar. Por lo tanto, SMP2 define extensiones opcionales para la interfaz *IModel* que amplían las posibilidades del desarrollo de modelos.

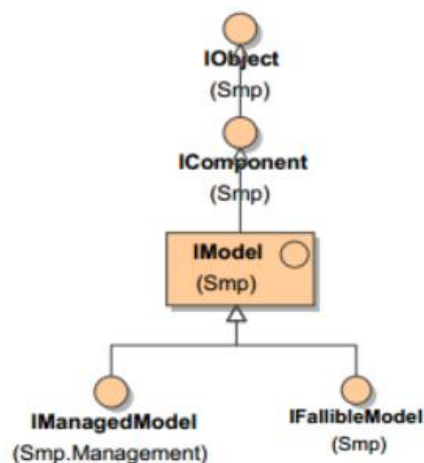


Figura 6. Diagrama de interfaces de mecanismos de modelos.

2.1.4.5. Interfaces de gestión

Las interfaces de gestión permiten que cualquier componente externo pueda acceder a cualquier mecanismo por su nombre, tanto los básicos como los opcionales para componentes y modelos.

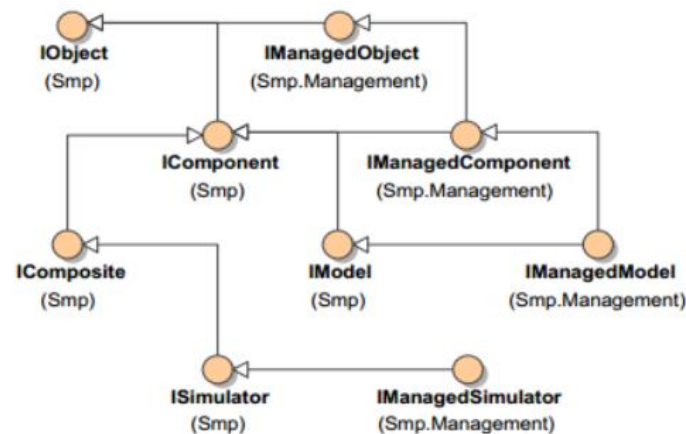


Figura 7. Diagrama de interfaces de gestión.

2.1.4.6. Entorno de simulación

Un entorno de simulación debe implementar la interfaz *ISimulator* para proporcionar acceso a los modelos y los servicios de una simulación. Esta interfaz deriva de *IComposite* para tener acceso a al menos a los contenedores de modelos y de servicios. El entorno de simulación debe proporcionar también un mecanismo de publicación de campos y operaciones para los modelos (*IPublication*).

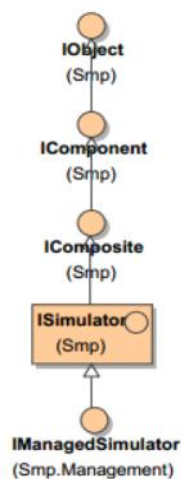


Figura 8. Diagrama de interfaces de entorno de simulación.

2.1.4.7. Servicios

Los servicios deben ser proporcionados por el entorno de simulación y consumidos por los modelos. Se definen para facilitar la interoperabilidad entre diferentes entornos compatibles con SMP2.

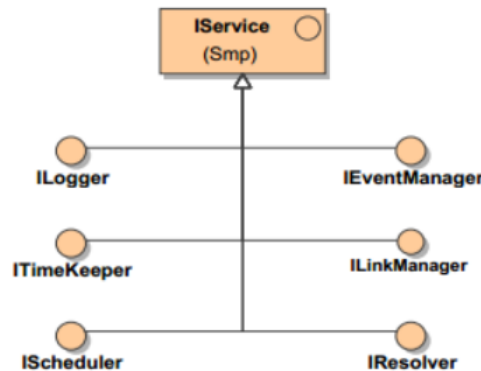


Figura 9. Diagrama de interfaces de servicios.

2.1.4.8. Excepciones

SMP2 también define excepciones específicas del estándar, además de enlazarlos a las operaciones de las interfaces. Todas las excepciones derivan de una clase común *Exception*.

2.1.5. Estados

2.1.5.1. Estados del entorno de simulación

El entorno de simulación debe estar siempre en uno de los estados definidos en el estándar. Además, cuando se produzca una transición de estados se deberá comprobar que está bien definida.

Los estados del entorno de simulación son los siguientes.

- **Building.**
 - En el estado *Building*, se crea la jerarquía de las instancias de los modelos a simular. Esto es realizado por un componente externo, no por el simulador.
 - Se entra en este estado después de la inicialización del entorno de simulación.
 - El siguiente estado a *Building* debe ser *Connecting*.

- **Connecting.**
 - En el estado *Connecting*, el entorno de simulación llama al método *Connect()* de cada uno de los modelos a simular.
 - Se entra en este estado después de llamar al método *Connect()* del entorno de simulación.
 - Una vez conectados todos los modelos, el siguiente estado debe ser *Initialising*.
- **Initialising.**
 - En el estado *Initialising*, el entorno de simulación ejecuta la inicialización de todos los puntos de entrada definidos.
 - Se entra en este estado cuando:
 - El entorno de simulación ha terminado de conectar todos los modelos.
 - De forma manual desde el estado *Standby*.
 - El siguiente estado a *Initialising* debe de ser *Standby*.
- **Standby.**
 - En el estado *Standby*, la simulación está parada y el tiempo de simulación no avanza.
 - Se entra en este estado cuando:
 - El entorno de simulación ha terminado de ejecutar la inicialización de todos los puntos de entrada en el estado *Initialising*.
 - Desde el estado *Storing*, el entorno de simulación termina de almacenar el estado actual de la simulación.
 - Desde el estado *Restoring*, el entorno de simulación termina de restaurar un estado almacenado.
 - De forma manual desde el estado *Executing* ejecutando el método *Hold()*.
 - Los siguientes estados a *Standby* pueden ser:
 - *Initialising*.
 - *Executing*.
 - *Storing*.
 - *Restoring*.
 - *Reconnecting*.
 - *Exiting*.

- **Executing.**
 - En el estado *Executing*, la simulación corre y el tiempo de simulación avanza. Los puntos de entrada serán ejecutados de acuerdo con el tiempo de simulación.
 - Se entra en este estado cuando se ejecuta el método *Run()* desde el estado *StandBy*.
 - El siguiente estado a *Executing* debe ser *Standby*.
- **Storing.**
 - En el estado *Storing*, el entorno de simulación almacena el estado actual de la simulación (valores de campos publicados, estado de los modelos, etc.) a un fichero.
 - Se entra en este estado cuando se ejecuta el método *Store()* desde el estado *StandBy()*.
 - El siguiente estado a *Storing* debe ser *Standby*.
- **Restoring.**
 - En el estado *Restoring*, el entorno de simulación restaura un estado almacenado en un fichero.
 - Se entra en este estado cuando se ejecuta el método *Restore()* desde el estado *Standby*.
 - El siguiente estado a *Restoring* debe ser *Standby*.
- **Reconnecting.**
 - En el estado *Reconnecting*, el entorno de simulación verifica que los modelos añadidos al simulador en el estado *Building* están correctamente publicados, configurados y conectados.
 - Se entra en este estado de forma manual desde el estado *Standby*.
 - El siguiente estado a *Reconnecting* debe ser *Standby*.
- **Exiting.**
 - En el estado *Exiting*, el entorno de simulación finaliza la ejecución de una simulación.
 - Se entra en este estado llamando al método *Exit()* desde el estado *Standby*.
 - El siguiente estado a *Exiting* es indefinido.

En la Figura 10 se detalla el flujo de estados para cada una de las fases de ejecución en el entorno de simulación.

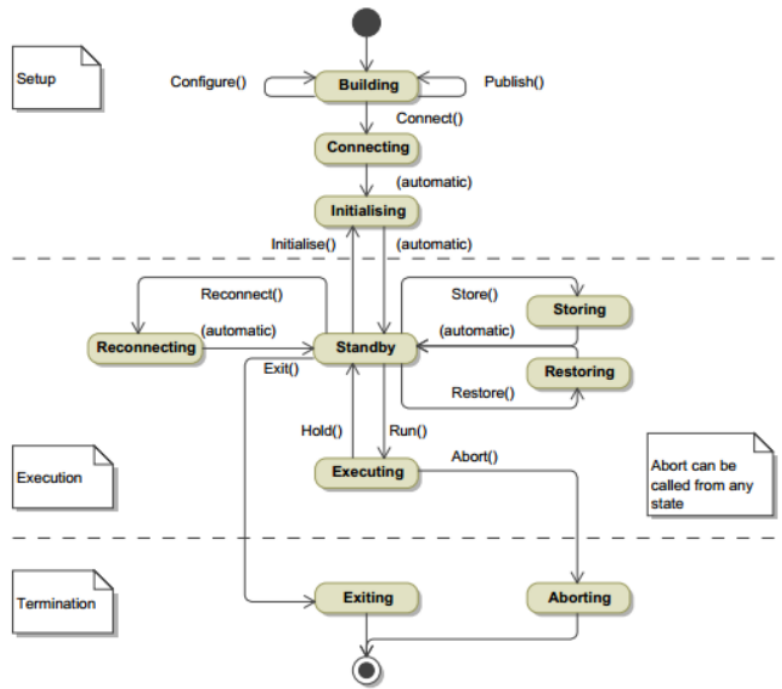


Figura 10. Diagrama de estados del entorno de simulación.

2.1.5.2. Estados de un modelo

Un modelo puede estar solamente en uno de los siguientes cuatro estados.

- **Created.** Cuando se crea la instancia del modelo.
- **Published.** Cuando el entorno de simulación llama al método *Publish()* del modelo.
- **Configured.** Cuando el entorno de simulación llama al método *Configure()* del modelo.
- **Connected.** Cuando el entorno de simulación llama al método *Connect()* del modelo.

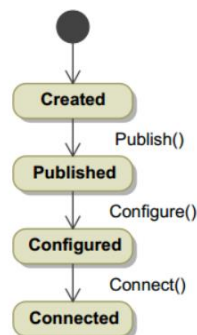


Figura 11. Diagrama de estados de un modelo.

2.2. Satélites

2.2.1. Definición

Básicamente, se puede definir un satélite natural como cualquier cuerpo celeste que gira alrededor de un planeta, como puede ser la luna con respecto a la tierra. Los satélites artificiales en cambio son dispositivos creados por el hombre que pueden ser utilizados para diferentes fines.

2.2.2. Tipos de satélites

A grandes rasgos, se pueden clasificar los diferentes tipos de satélites de acuerdo a dos características; la órbita que describen alrededor de la tierra y la misión para la que son diseñados.

2.2.2.1. Por órbita

- Satélites de **órbita geoestacionaria**. Realiza una órbita sobre el ecuador terrestre a la misma velocidad de la tierra, por lo que siempre quedan fijos sobre un punto terrestre. Esta órbita también es conocida como órbita de Clarke. Se colocan a una altura de 35786 kilómetros sobre la tierra, con el fin de equilibrar los valores de la fuerza gravitatoria y la fuerza centrípeta ejercida por la tierra. Son ideales para satélites de comunicaciones y de predicción meteorológica. El primer satélite de este tipo en ser lanzado fue el **Syncom 3** (1963), utilizado principalmente para la retransmisión de los juegos olímpicos de Japón de 1964.
- Satélites de **órbita baja terrestre**. Se colocan a una altura de entre 200 y 2000 kilómetros de altitud sobre la tierra, concretamente entre la atmósfera y los cinturones de Van Allen. Debido a su baja altitud, realizan un giro completo alrededor de la tierra en muy poco tiempo (alrededor de 90 minutos). En esta órbita se encuentran una gran cantidad de satélites, lo que puede llegar a ser un problema debido al riesgo de impactos entre ellos. En esta órbita se encuentran gran parte de las estaciones y laboratorios espaciales (como la Estación Espacial Internacional), así como satélites orientados al uso militar y de teledetección.
- Satélites de **órbita media terrestre**. Son los satélites que trabajan en la órbita comprendida entre los 2000 kilómetros (por encima de la órbita baja terrestre) y los 35786 kilómetros (por debajo de la órbita geoestacionaria). Tardan varias horas en realizar un giro sobre la tierra (generalmente entre 6 y 8 horas), y son usados sobre todo por satélites de posicionamiento (**GPS** o **Galileo**).
- Satélites de **órbita alta terrestre**. Son los que situados a una altitud superior a la órbita de Clarke. Describen una órbita muy elíptica, con periodos orbitales mayores a 24

horas. Son especialmente aptos para cubrir las zonas polares. Los satélites de este tipo más conocidos fueron los que se lanzaron en el marco del Proyecto **VELA** durante la guerra fría, cuya misión consistía en el espionaje de actividades militares rusas y en la detección de explosiones nucleares.

2.2.2.2. Por misión

Existen una gran cantidad de misiones de distinta naturaleza para las que puede ser diseñados un satélite. Se explican brevemente a continuación algunos de los tipos de satélites clasificados de acuerdo con su misión.

- Satélites de **comunicaciones**. Son el tipo más común de satélite. Se usan para la transmisión de radio, televisión, telefonía e Internet a nivel mundial.
- Satélites de **observación terrestre**. Se utilizan para obtener información útil sobre la tierra. Tienen multitud de aplicaciones, tales como previsión meteorológica (**METEOSAT**), control de medio ambiente (**ENVISAT**, **JASON**), o construcción de mapas. También se pueden englobar aquí los denominados satélites espías, orientados al uso militar.
- Satélites de **observación espacial**. Llamados también telescopios espaciales, son empleados principalmente para la observación de cuerpos celestes con el fin de obtener más conocimiento relativo al cosmos.
- Satélites de **geolocalización**. Permiten conocer la posición de un objeto sobre la superficie terrestre. Son utilizados por los sistemas **GPS**, **GLONASS** o **Galileo**.
- **Estaciones espaciales**. Son naves diseñadas para ser habitadas por seres humanas en su interior. La más conocida de ellas es la llamada **Estación Espacial Internacional**, administrada por agencias espaciales de varios países.
- **Biosatélites**. Son satélites diseñados para llevar organismos vivos en su interior, con el fin de estudiar el efecto sobre ellos.

2.2.3. Subsistemas de un satélite

Los subsistemas de un satélite pueden ser clasificados en dos grupos.

- Uno de ellos es la **carga útil (payload)**, que contiene todos los elementos necesarios para realizar la función para la que el satélite fue diseñado. Por ejemplo, un satélite de comunicaciones estaría compuesta principalmente por el subsistema de antenas y el subsistema de comunicaciones.
- El otro grupo es el llamado **plataforma o bus**, y su principal cometido es mantener operativo el satélite durante el tiempo de misión.

Se explican a continuación los diferentes subsistemas que conforman el segundo grupo.

2.2.3.1. Subsistema de energía

Su principal cometido consiste en suministrar energía eléctrica a cada uno de los módulos del satélite. Básicamente, consta de dos elementos principales.

- **Paneles solares.** Convierte la energía solar a energía eléctrica. Su eficiencia es de un 20-25% al comienzo de la misión, pero puede quedar degradada hasta llegar a ser de un 5-10% en los últimos meses de vida del satélite.
- **Baterías.** Para cuando el satélite se encuentre en eclipse con respecto al sol, es necesario contar con baterías que proporcionen energía hasta que el satélite pueda volver a obtener energía solar. Estas baterías son recargadas mientras los paneles solares están funcionando. Si las baterías no contienen la energía suficiente para el satélite durante el periodo de eclipse, algunos sistemas dejarán de funcionar.

2.2.3.2. Subsistema de control térmico

Este subsistema controla la temperatura interna del satélite para que los sistemas que requieren de un cierto rango de temperaturas puedan seguir funcionando. Para conseguir esto, se pueden emplear diferentes técnicas.

- Colocación de **mantas térmicas** en zonas críticas para proporcionar aislamiento con respecto al medio.
- **Espejos de radiación** que actúan como reflectores de la luz visible y ultravioleta proveniente del sol.
- **Tubos de calor**, utilizados principalmente en los periodos de eclipse para distribuir calor por todo el satélite.

2.2.3.3. Subsistema de seguimiento, telemetría y telecomando

El subsistema de telecomando y telemetría (TT&R) permite el envío de **telecomandos** desde tierra para recibir **telemetrías** con información sobre el estado y la configuración actual del satélite. Este subsistema permite la monitorización y el control del funcionamiento del satélite una vez que se encuentre en el espacio.

La comunicación entre la estación de tierra y el satélite se realiza mediante un canal de radiofrecuencia a través de unas antenas especializadas, capaces de trabajar tanto en condiciones normales como en condiciones anormales de operación (para lo cual se hace necesario el uso de antenas prácticamente omnidireccionales). Normalmente, se utilizan sistemas con poco ancho de banda y con modulación PSK o FSK.

Existen varios estándares relativos a la definición de servicios para un sistema de telemetría y telecomando. Uno de los más utilizados en el ámbito de las misiones espaciales europeas es el estándar **PUS²** (Packet Utilisation Standard). Este estándar no define el contenido de los paquetes de telemetría y telecomando, sino los tipos de servicios que deben ser implementados por el software de a bordo. El estándar reserva los servicios del rango 0-127 (aunque sólo se utilizan 16 de ellos), y permite a los desarrolladores la definición de nuevos servicios en el rango 128-255. Cada uno de estos servicios pueden a su vez contener varios subservicios.

2.2.3.4. Subsistema de control de actitud y orientación

El subsistema **ADCS** (en inglés, *Attitude Determination and Control*), consiste en el control de la orientación del satélite o algunos de sus componentes en una determinada dirección (por ejemplo, los paneles solares deberían apuntar al sol, así como los telescopios de un satélite de observación a la región que se desee explorar).

Para la realización de esta tarea, se necesitan **sensores** que permitan obtener la orientación del satélite con respecto a un sistema de referencia y de **actuadores** para mover el satélite y corregir el error con respecto al ángulo objetivo.

Algunos de los sensores más comunes que se pueden encontrar en un subsistema ADCS son los siguientes.

- **Giroscopios.** Son sensores de velocidad angular que miden el cambio de cada uno de los ángulos de navegación del satélite con respecto al espacio inercial. Estos cambios de velocidad pueden ser usados para la estimación de la orientación del satélite.
- **Sensores solares.** Se utilizan para determinar la dirección del sol con respecto al satélite. Estos sensores no pueden ser utilizados durante el periodo de eclipse.
- **Magnetómetros.** También llamados sensores magnéticos, miden la dirección del campo magnético de la tierra con respecto al satélite. De este modo, la posición puede ser obtenida comparando la salida de estos sensores con un modelo del campo magnético terrestre almacenado en el satélite.
- **Sensor estelar.** Utiliza las estrellas como sistemas de referencia, y su funcionamiento se basa en la obtención de imágenes del mapa celeste por medio de una cámara óptica. Debido a que se conocen con gran exactitud la posición de muchas estrellas, estas imágenes pueden ser analizadas para inferir así la posición y la orientación del satélite con una gran precisión.
- **Sensor de tierra.** Consiste básicamente en un dispositivo sensible al calor para medir la radiación infrarroja de la tierra. El calor recibido por estos dispositivos depende de la orientación del satélite con respecto a la tierra.

En cuanto a los actuadores, se explican brevemente a continuación alguno de los más utilizados.

² Este estándar se publicó en la ECSS bajo el nombre ECSS-E-70-41^a.

- **Ruedas de reacción.** Son ruedas con motores eléctricos que provocan un torque mediante el giro de cada una de ellas a una determinada velocidad. Para girar el satélite en cada uno de los ángulos de navegación se pueden usar tres de estas ruedas, aunque se suele añadir una cuarta rueda redundante.
- **Magnetorquers.** Se basa en una bobina que crea un campo magnético que interactúa con el campo magnético terrestre, lo que provoca un torque que gira el satélite.
- **Propulsores.** Proporcionan empuje a partir del combustible almacenado en la nave. Son usados para errores de gran escala que no pueden ser corregidos mediante los otros actuadores.

Una vez medida la orientación, y a partir de la orientación deseada, es necesario comparar estos dos valores y aplicar una técnica de control para obtener los torques a aplicar sobre los actuadores. En esta memoria, se puede encontrar un ejemplo de este lazo de control en el sistema ADCS implementado en la sección 3.6.

2.2.3.5. Subsistema de propulsión

El subsistema de propulsión proporciona los medios para ejecutar las operaciones comandadas por el subsistema ADCS con el fin de mantener la orientación deseada, así como las operaciones de cambio de órbita del satélite. El funcionamiento de este subsistema se basa en la tercera ley de Newton; expulsando materia a gran velocidad y temperatura a través de conductos de escape, se obtienen fuerzas de empuje en sentido contrario.

De acuerdo con la naturaleza de la energía utilizada para acelerar los propelentes, se pueden dividir los propulsores en dos grandes grupos.

- Propulsores **químicos**. Se basa en la generación de gases a muy elevadas temperaturas mediante la reacción química de propelentes.
- Propulsores **eléctricos**. Mientras que los propulsores químicos almacenan su energía en los propelentes, la energía requerida por los propulsores eléctricos es normalmente generada por los paneles solares. El propelente suele ser expulsado del orden de 20 veces más rápido que los propulsores químicos.

2.2.3.6. Subsistema estructural

Para garantizar que el satélite y todos sus componentes sea capaz de soportar todas las fuerzas e incidencias a las que se ve sometido una vez comenzada la misión (como por ejemplo, impactos de pequeños meteoritos, fuerzas gravitatorias del sol y la tierra o factores ambientales) es necesario contar con una estructura lo suficiente fuerte y rígida. La masa de esta estructura suele ser del 10% al 20% con respecto a la masa total del satélite, y en su construcción se suelen utilizar materiales como aluminio, titanio o acero.

2.2.3.7. Subsistema de procesamiento de datos

El componente principal de este subsistema es un computador de a bordo, que básicamente se encarga de interactuar con todos los componentes del satélite. El computador debe funcionar bajo un sistema operativo de tiempo real (como por ejemplo *VxWorks*, *LinuxRT* o *RTEMS*), ya existen restricciones muy críticas con respecto al tiempo en que se deben obtener y procesar los datos.

El desarrollo del software de a bordo del satélite debe correr en este computador, y es implementado generalmente por un equipo de personas con conocimientos tanto de control como de desarrollo de software, ya que la complejidad de este software (tanto a nivel arquitectónico como funcional) es cada vez mayor.

CAPÍTULO 3: Desarrollo

3.1. Componentes del sistema

El objetivo del sistema propuesto ha requerido del desarrollo de un conjunto de componentes que permitan simular modelos SMP2 desde Matlab. Como caso de prueba, se ha desarrollado un pequeño módulo de control de orientación y actitud de un satélite, en el que intervienen modelos SMP2 codificados en C++, clases en Matlab y un modelo dinámico definido en Simulink.

En la Figura 12 se muestra el esquema del sistema implementado, con las tecnologías utilizadas para la implementación de cada uno de los componentes. Las aplicaciones que puedan ser utilizadas por el usuario final se han representado como un círculo.

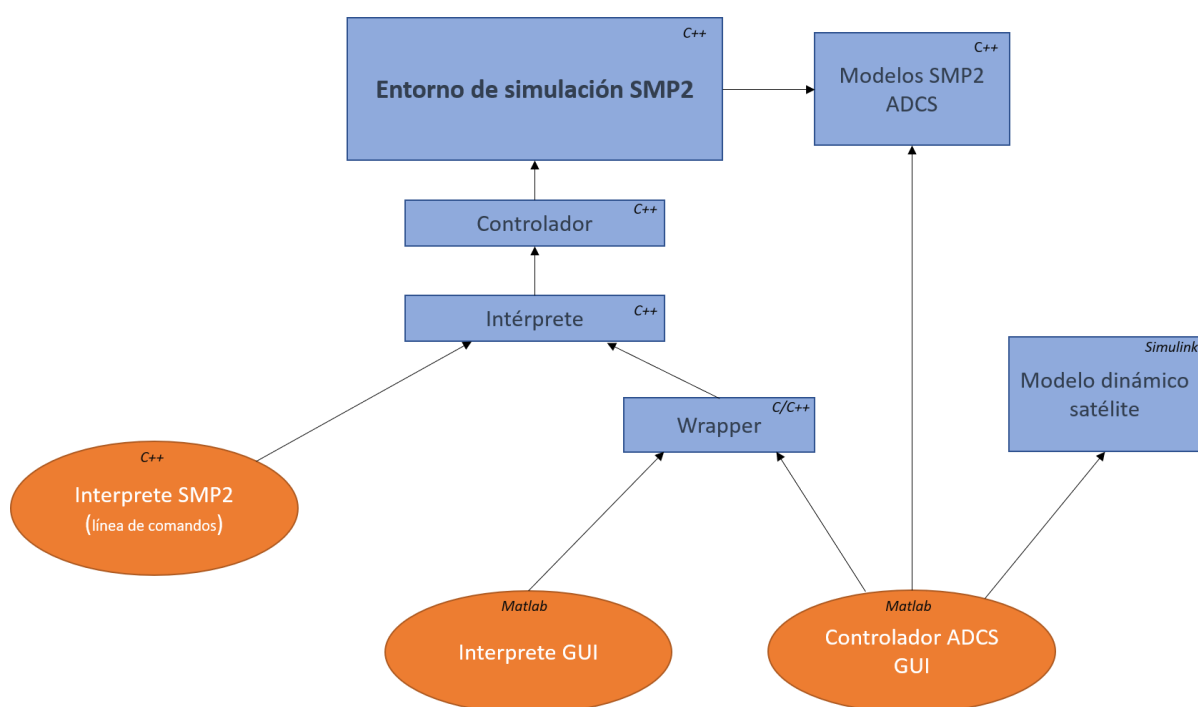


Figura 12. Esquema general del sistema implementado.

El componente más importante es el *entorno de simulación SMP2*, donde se implementan algunas de las interfaces del estándar SMP2. Esto permitirá la ejecución de un motor de simulación que interprete modelos implementados de acuerdo con el estándar.

El módulo *Controlador* actúa como una capa intermedia entre el entorno de simulación y los demás componentes. Es capaz de instanciar las clases del entorno de simulación e interactuar sobre ellas, así como de cargar ensamblados que definan un conjunto de modelos.

El componente *Intérprete* se puede entender como un traductor de las respuestas del entorno de simulación a un lenguaje legible por el usuario. Puede recibir órdenes (como por ejemplo, ejecutar simulación o consultar el valor de un campo de un modelo), a partir de las cuales ejecuta un conjunto de acciones en el *Controlador* y devuelve la respuesta dependiendo de la salida obtenida.

La función del componente *Wrapper* es permitir la integración del sistema en Matlab. Consiste en un conjunto de funciones externas que llaman al *Intérprete*.

Se ha desarrollado una aplicación basada en línea de comandos que permita al usuario final simular y trabajar con modelos SMP2. Básicamente, la aplicación recibe órdenes del usuario, las envía al *Intérprete* y escribe la respuesta obtenida.

Utilizando el componente *Wrapper*, se ha implementado una aplicación con el mismo propósito que la anterior, para ser ejecutada dentro del entorno de Matlab. La aplicación puede ser utilizada por medio de una interfaz gráfica.

Para la implementación del sistema de control de actitud de un satélite, se han desarrollado los siguientes componentes.

- Modelos SMP2 que contienen los modelos correspondientes a los sensores, actuadores y computador a bordo del sistema.
- Modelo en Simulink con el comportamiento dinámico del satélite.

Finalmente, se ha implementado una interfaz gráfica en Matlab para ejecutar el sistema de control, integrando los dos componentes anteriores.

3.2. Entorno de simulación

El entorno de simulación es el corazón del sistema, ya que es utilizado de forma directa o indirecta por todos los demás componentes.

Como se explicó en la sección 2.1.4, para el desarrollo del entorno de simulación es necesario la implementación de un conjunto de interfaces. En este proyecto, se han implementado un número amplio de interfaces y métodos que engloban las características más importantes del estándar³. Nuestro entorno de simulación es capaz de simular todos los modelos SMP2 desarrollados en nuestro sistema.

Es importante resaltar que las clases necesarias para la implementación de modelos SMP2 no se definen en este apartado, ya que no forman parte del entorno de simulación.

³ En el Anexo 2, se puede consultar los métodos implementados para cada una de las interfaces, así como una breve descripción de su función.

3.2.1. Clases principales

En este apartado, se explican brevemente el cometido de las clases más importantes que implementan el entorno de simulación. Para más detalles de implementación, se pueden consultar los diagramas de clases en el Anexo 1.

- **GenericComponent** implementa la interfaz *IComponent*, común a muchos de los elementos SMP2. Se utiliza como base de los componentes implementados.
- **Simulator** implementa la interfaz *ISimulator*. En esta clase se encuentran todos los métodos necesarios para ejecutar el entorno de simulación con un conjunto de modelos previamente cargados.
- **Publication** implementa la interfaz *IPublication*. Es utilizada para la publicación de métodos y atributos de los modelos.
- **TypeRegistry** implementa la interfaz *ITypeRegistry*. Esta clase es instanciada típicamente como un *singleton*, y permite la gestión de los tipos básicos SMP2, así como el registro de nuevos tipos definidos por el usuario.
- **Logger** es un servicio que implementa la interfaz *ILogger*. Es usada para gestionar los mensajes enviados por los diferentes componentes de un entorno de simulación.
- **Scheduler** es un servicio que implementa la interfaz *IScheduler*. Se encarga básicamente de gestionar la ejecución de los puntos de entradas definidos en el entorno de simulación, a partir de los diferentes tiempos.
- **TimeKeeper** implementa la interfaz *ITimeKeeper*. Almacena el tiempo actual de simulación, además del tiempo de misión y de época.
- **EventManager** implementa la interfaz *IEventManager*. Proporciona un mecanismo de notificación global entre los componentes del entorno de simulación. Así, cada componente puede registrar un punto de entrada para que sea ejecutado cuando se dispare un evento global.

3.2.2. Funcionamiento

Se explica a continuación el funcionamiento de varias de las características más importantes del entorno de simulación.

3.2.2.1. Carga de modelos

El estándar SMP2 deja abierta la manera en la que se deben cargar los ensamblados de modelos en el simulador. Esta tarea no es realizada en este sistema (en nuestro proyecto, la implementa

el módulo *Controlador*), pero si se deben utilizar algunos de los métodos de la clase *Simulator* durante el proceso.

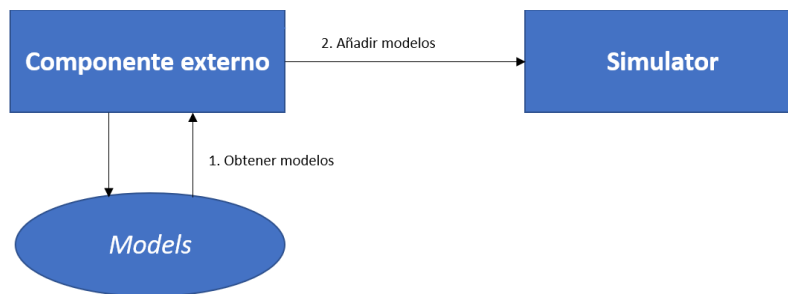


Figura 13. Carga de modelos en el simulador.

3.2.2.2. Publicación de campos

Las clases que implementan los SMP2 a simular pueden contener cualquier número de atributos, pero si se desea obtener o cambiar el valor de algunos de ellos desde el entorno de simulación, estos modelos deben publicar dichos atributos.

La publicación de campos es implementada por la clase *Publication*, pero es la clase *Simulation* la que llama a los métodos de publicación de cada uno de los modelos.

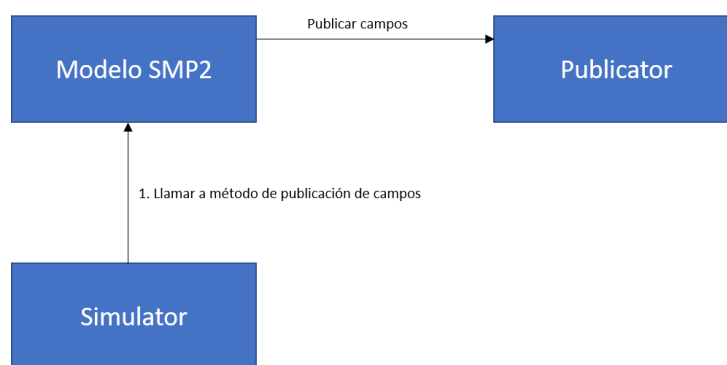


Figura 14. Publicación de campos de un modelo.

La clase *Publication* almacena una referencia a la dirección de memoria de cada uno de los campos publicados. De esta manera, cuando se desee modificar u obtener el valor de cualquiera de ellos, es necesario llamar al método correspondiente de *Publication* con el nombre del campo.

3.2.2.3. Publicación de operaciones

El flujo de ejecución para la publicación de una operación de un modelo es muy parecido al mostrado en la Figura 14. No obstante, el modelo debe además publicar los parámetros de la operación, por medio de un objeto devuelto por la clase *Publication*.

En este caso, *Publication* almacena en una estructura el nombre de la operación, sus atributos y una referencia del modelo al que pertenece. La lógica de la ejecución de una operación por medio de un componente externo no es tan inmediata como la consulta o modificación de un campo publicado, y es gestionada principalmente por el modelo.

Los modelos que necesiten publicar operaciones deben implementar la interfaz *IDynamicInvocation*. El método principal de esta interfaz sirve para invocar la operación, y recibe como parámetro el nombre de la operación y un objeto con la información sobre los parámetros de la operación y valores de los mismos.

3.2.2.4. Definición de eventos programados

Un modelo puede definir un conjunto de acciones que se ejecuten en un instante determinado de tiempo. Además, se puede especificar que estas acciones se ejecuten cada cierto periodo de tiempo, ya sea indefinidamente o por un cierto número de veces. Las acciones a ejecutar deben ser definidas como un punto de entrada, que ha de estar asociado a un método del modelo que implemente dichas acciones.

La gestión de estos eventos es realizada en la clase *Scheduler*, que proporciona varias funciones para planificar la ejecución de puntos de entrada de acuerdo al tiempo de simulación, tiempo de época y tiempo de misión. Esta clase permite además la modificación y el borrado de eventos añadidos previamente.

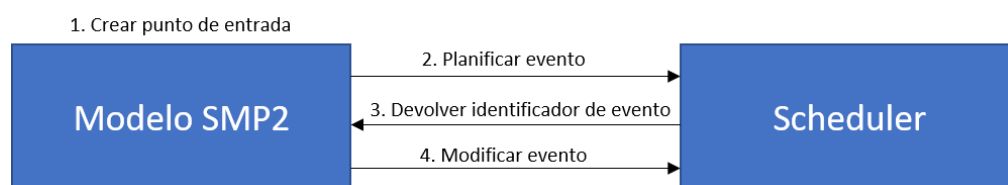


Figura 15. Planificación y modificación de eventos.

3.2.2.5. Ejecutar o parar simulador

Para ejecutar el motor de simulación, se debe llamar al método correspondiente de la clase *Simulator*. Esta clase lanza un *timer* en un hilo independiente, que inicializa un contador que almacena el tiempo transcurrido con la máxima precisión posible. El valor de este contador se usa para actualizar el tiempo actual de simulación en la clase *TimeKeeper*. En cada paso, el

timer llama al *Scheduler* para verificar si existen eventos programados que necesiten ser lanzados de acuerdo con el tiempo actual de simulación.

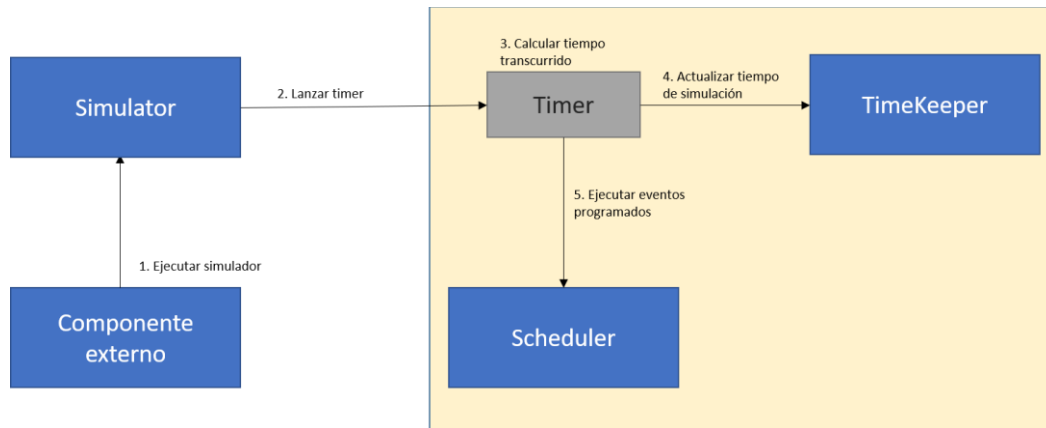


Figura 16. Ejecución de simulador.

Cuando se desee finalizar la simulación, se finaliza la ejecución del hilo asociado al *timer* lanzado previamente. El tiempo dejará de correr, y no se actualizará el *TimeKeeper* ni se ejecutarán eventos programados.

3.2.2.6. Envío de mensajes

El simulador es capaz de enviar mensajes de log al servicio implementado en la clase *Logger*, que almacenará la información del mensaje enviado en una estructura y la imprimirá por pantalla.

Los mensajes de log contienen la siguiente información.

- Modelo emisor del mensaje.
- Tipo de mensaje. Puede ser de *información*, de *advertencia* o de *error*.
- Contenido del mensaje.
- Tiempo de simulación en el que se emitió el mensaje.

Solamente la clase *Simulator* tiene acceso al servicio *Logger*, por lo que, si un modelo desea enviar mensajes de log, debe almacenar una referencia a la instancia de esta clase⁴.

⁴ Se puede obtener una referencia de la instancia del simulador cuando se ejecuta la fase de conexión. En esta fase, el simulador llama al método *Connect* de todos los modelos cargados, pasándole como parámetro la propia instancia del simulador.

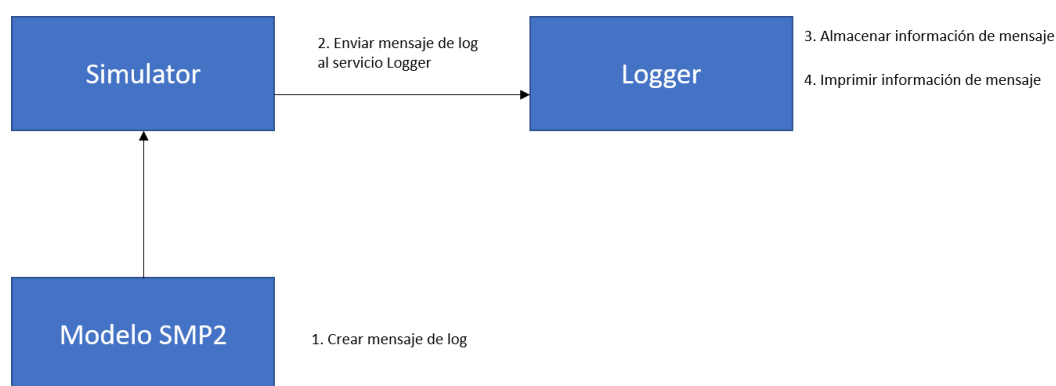


Figura 17. Envío de mensajes de log.

3.2.2.7. Registro de acciones a eventos

Existen en el entorno de simulación un conjunto de eventos que se lanzan en determinadas condiciones. Un modelo puede asociar un punto de entrada para que sea ejecutado cuando se active cualquiera de estos eventos. Ejemplos básicos podrían ser la ejecución de rutinas de inicialización cuando se ejecute el simulador, o el reseteo de algunas variables del modelo cuando se pare el simulador.

El servicio que gestiona los eventos del sistema está implementado en la clase *EventManager*. Esta clase contiene una lista donde se relacionan los puntos de entrada registrados con el evento correspondiente. Así, cuando ocurra algún evento en cualquiera de los componentes del entorno de simulación, se deberá emitir llamando a un método a este servicio, que ejecutará los puntos de entradas asociados a dicho evento.



Figura 18. Emisión de eventos.

3.2.2.8. Registro de arrays

Cuando un modelo necesite publicar un campo que no sea de un tipo básico, es necesario registrar su tipo antes de realizar la publicación. Ejemplos de tipos no básicos pueden ser

estructuras, tipo enumerados o arrays. En este proyecto se ha implementado el registro de arrays de elementos de tipo básico, ya que son los más útiles a la hora de definir un modelo.

El registro de tipos está implementado en la clase *TypeRegistry*. Cuando se desea publicar un array, se debe proporcionar la siguiente información.

- Nombre y descripción del tipo.
- Identificador único del tipo.
- Tipo básico de elemento que almacenará el array.
- Tamaño del array.

Una vez registrado un array, el modelo puede publicar un campo de este tipo llamando al método correspondiente de la clase *Publication*, y pasándole como parámetro el identificador único del tipo creado.

3.3. Intérprete SMP2 (línea de comandos)

Una vez implementado el entorno de simulación, el siguiente paso en nuestro proyecto fue desarrollar un intérprete SMP2 que haga uso de todas las capacidades desarrolladas.

En nuestro proyecto, se han programado dos intérpretes, uno en C++ y otro en Matlab. No obstante, la lógica de ambos intérpretes se ejecuta sobre las mismas clases, tal y como se puede apreciar en la Figura 12.

En el Anexo 3, se especifican en detalle los casos de uso de la aplicación desarrollada, así como los diagramas de secuencia para cada uno de ellos. Gracias a estos diagramas podemos apreciar con más claridad la interacción entre las clases del entorno de simulación

3.3.1. Diagrama de clases

Se muestra en la Figura 19 (siguiente página), el diagrama de clases de la aplicación implementada.

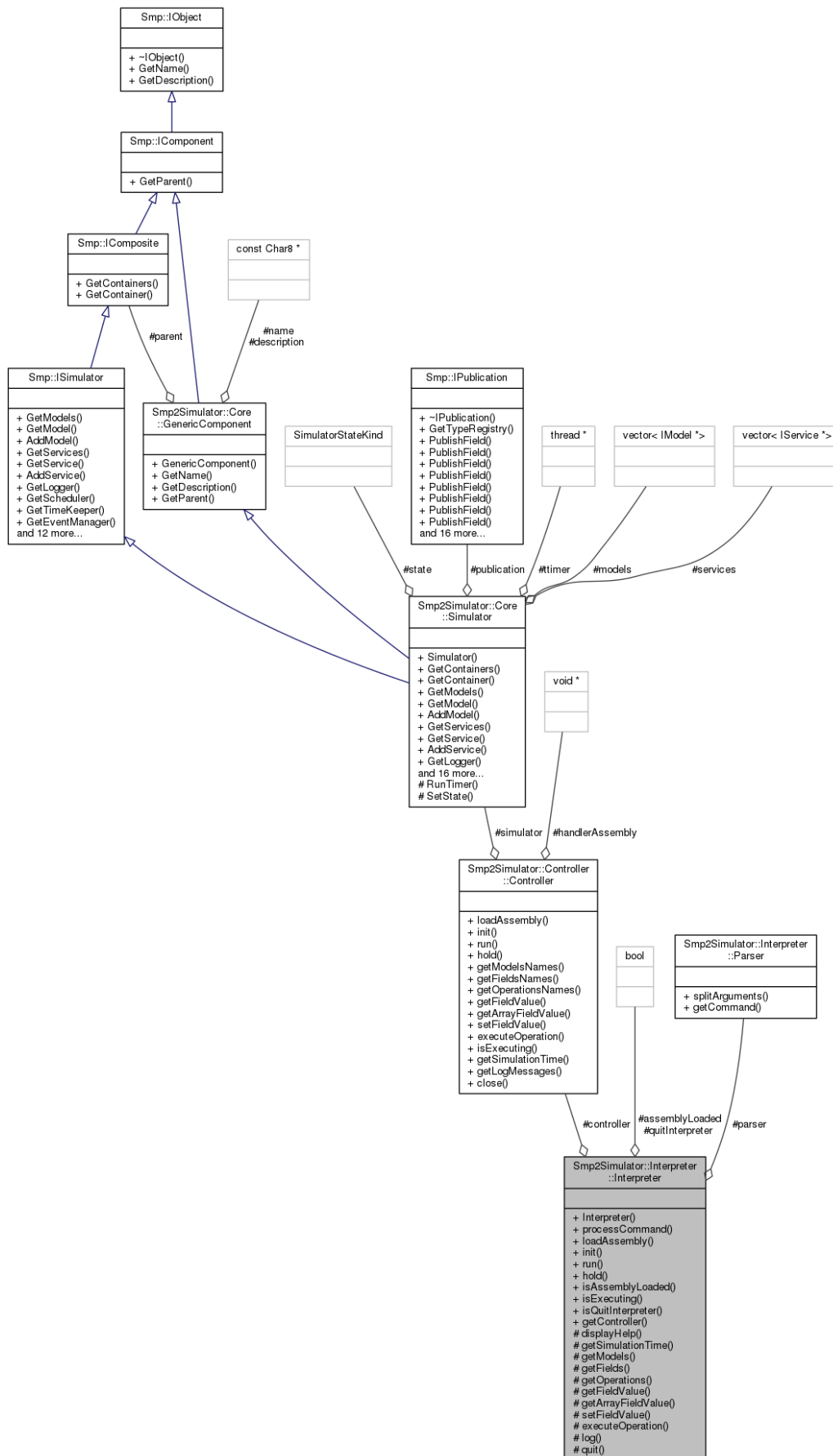


Figura 19. Diagrama de clases del intérprete SMP2.

3.3.2. Clases principales

Se explica brevemente el cometido de las clases implementadas para el desarrollo del intérprete.

- **Controller.** Representa el módulo *Controlador* de la Figura 12. Accede directamente al entorno de simulación para ejecutar acciones sobre él, además de cargar y administrar el *handler* que contiene la librería dinámica del ensamblado a utilizar por el intérprete.
- **Interpreter.** Representa el módulo *Intérprete* de la Figura 12. Cada vez que el usuario inserte un comando en la aplicación, se llamará a los métodos de una instancia de esta clase para ejecutar las acciones relativas al comando introducido y obtener la respuesta del sistema. El intérprete llamará a su vez a los métodos de una instancia de la clase *Controller* para interactuar con el entorno de simulación.
- **Parser.** Cuando el intérprete recibe un comando, se utiliza esta clase para parsearlo a una estructura fácilmente manipulable.

3.3.3. Esquema general de funcionamiento

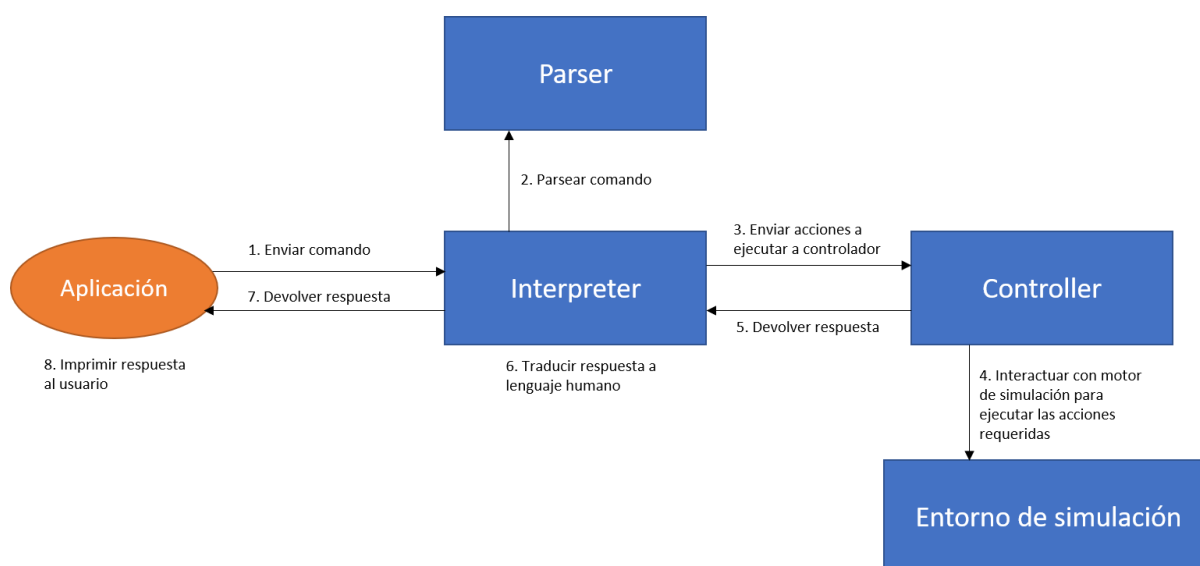


Figura 20. Esquema de funcionamiento de intérprete SMP2.

El funcionamiento de la aplicación se basa en imprimir las respuestas que describan el efecto de las acciones sobre el entorno de simulación de los comandos enviados por el usuario. En la Figura 20 se describen los pasos a seguir para el procesamiento de un comando por la aplicación, así como la conexión entre cada uno de los componentes que la forman.

Básicamente, cuando el usuario envía un comando al intérprete, se ejecuta la siguiente secuencia de pasos.

- La aplicación envía la cadena con el comando al *Interpreter*.
- El *Interpreter* parsea la cadena recibida a una estructura manipulable, utilizando la clase *Parser*.
- El *Interpreter* procesa el comando parseado.
 - Si no se reconoce el comando, se envía un mensaje de error a la aplicación.
 - Si se reconoce el comando, pero alguno de sus parámetros es inválido, se envía un mensaje de error a la aplicación.
 - En otro caso, se realizan las acciones asociadas al comando llamando a los métodos necesarios de la clase *Controller*.
- El *Controller* interactúa con el motor de simulación para ejecutar las acciones recibidas por la clase *Interpreter*.
- El *Interpreter* traduce la respuesta recibida de la clase *Controller* a una cadena que describa el resultado de la ejecución del comando recibido.
- La aplicación imprime la cadena resultante.

```
>> get fields
+ Example_SMP2.counter
+ Example_SMP2.lastSimulationTime
+ Example_SMP2.bufferTimes

>> set value Example_SMP2 counter 14
The field was updated successfully

>> get value Example_SMP2 counter
+ Value: 14
```

Figura 21. Ejemplo de uso de intérprete SMP2.

3.3.4. Funcionalidades

Se describen a continuación las funcionalidades de la aplicación implementada. Básicamente, consisten en el conjunto de comandos que pueden ser procesados por el intérprete.

3.3.4.1. Cargar ensamblado

Como se describió en el apartado 3.2.2.1, SMP2 no exige una forma común para la carga del ensamblado de modelos en el simulador.

En nuestro proyecto, se ha optado por implementar los ensamblados como una librería dinámica que contengan los modelos y un método externalizado llamado *Initialise* que reciba un objeto de la interfaz *ISimulator* y otro de *ITypeRegistry*.

El codificador de modelos, debe entonces compilar los modelos y el método externalizado. El usuario recibirá la librería dinámica resultante, y podrá utilizar el intérprete SMP2 para cargar el ensamblado.

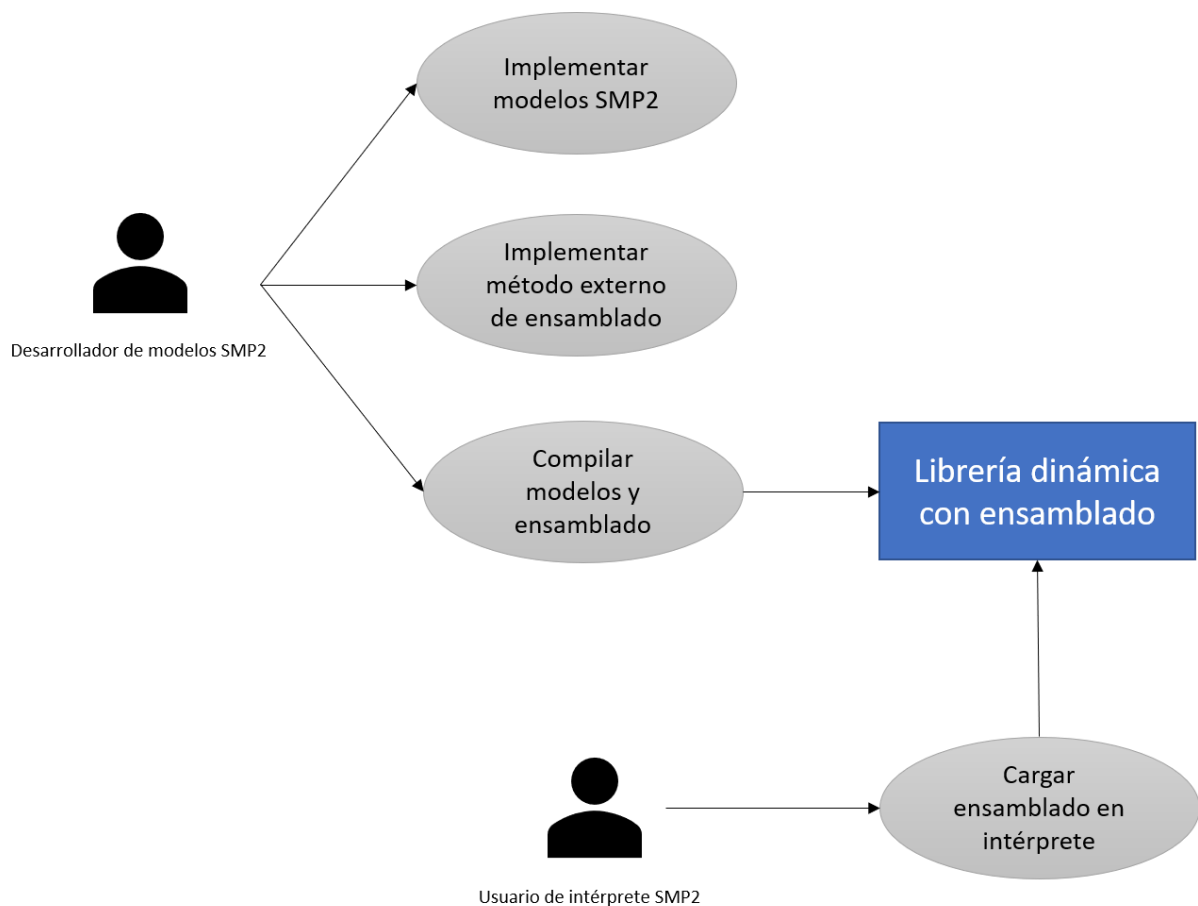


Figura 22. Roles y uso de ensamblados de modelos en el simulador.

Existen dos maneras para cargar el ensamblado desde el intérprete SMP2.

- Ejecutando la aplicación pasándole como parámetro el archivo con el ensamblado a cargar.
- Ejecutar la aplicación sin parámetros y escribir el comando **LOAD** con el nombre del archivo del ensamblado a cargar.

```
[svf@localhost build]$ ./main.out ./libAdcs.so

SMP2 Simulator

SMP2 interpreter based on command line

Antonio Márquez Marín
Master degree in computer system engineering (UNED/UCM)

Loading assembly from ./libAdcs.so ...
Assembly loaded!

Initializing assembly ...
Assembly initialized in the simulator successfully

Write 'help' to display commands available
>>
```

Figura 23. Carga de un ensamblado por línea de comandos en intérprete SMP2.

La carga del ensamblado está implementada en la clase *Controller*. Consiste en la siguiente serie de pasos.

- Creación de instancias de las clases *Simulator* y *TypeRegistry*.
- Carga de librería dinámica con ensamblado.
- Se verifica si la librería contiene el método externalizado requerido, y se llama a dicho método pasándole como parámetro las instancias creadas en el primer paso.
- El método de inicialización del ensamblado interactúa con el simulador y el registrado de tipos pasados como parámetro.

En la Figura 24, se representa un caso simplificado del funcionamiento descrito, donde el controlador envía a la librería dinámica una instancia de *Simulator* y éste añade varios modelos al mismo.

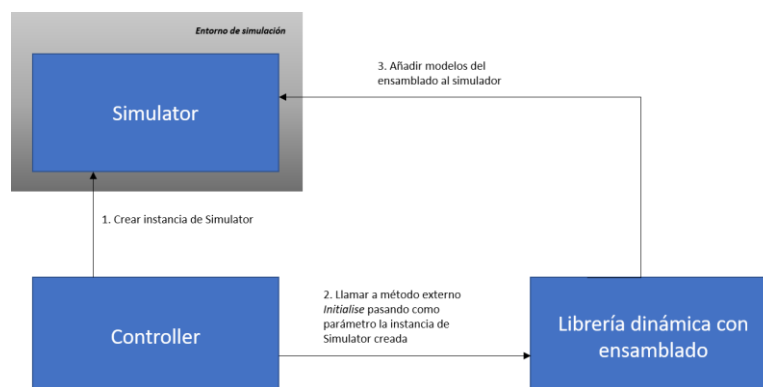


Figura 24. Carga de un ensamblado de modelos desde la clase *Controller*.

Una vez cargados los modelos, el *Controller* puede proceder a ejecutar los métodos de publicación (*publish*), configuración (*configure*) y conexión (*connect*) de cada uno de los modelos cargados, tal y como se especifica en la primera fase del apartado 2.1.3.

Se puede consultar el funcionamiento a nivel de clases de esta funcionalidad en el diagrama de secuencia de la Figura 62.

3.3.4.2. Ejecutar y parar el simulador

Se puede poner el simulación en estado de ejecución escribiendo el comando **RUN** en el intérprete. Esto provocará que el *Controller* llame a la instancia de *Simulator* en el entorno de simulación para iniciar la ejecución del motor de simulación. Se mostrarán los mensajes de logs recibidos por cada uno de los componentes de la simulación.

Para parar el motor de simulación basta con pulsar la tecla *ENTER* mientras el simulador esté en ejecución.

```
>> run
0 | Simulator | Information | Simulation executing

0 | Example_SMP2 | Information | Counter value: 1
1000000000 | Example_SMP2 | Information | Counter value: 2
2000000000 | Example_SMP2 | Information | Counter value: 3
3000000000 | Example_SMP2 | Information | Counter value: 4
4000000000 | Example_SMP2 | Information | Counter value: 5
5000000000 | Example_SMP2 | Information | Counter value: 6
6000000000 | Example_SMP2 | Information | Counter value: 7
7000000000 | Example_SMP2 | Information | Counter value: 8
8000000000 | Example_SMP2 | Information | Counter value: 9

8277759000 | Example_SMP2 | Information | Reset counter
8277759000 | Simulator | Information | Simulation paused

>> █
```

Figura 25. Ejecutar y parar la simulación en el intérprete SMP2.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en los diagramas de secuencia de la Figura 63 y la Figura 64.

3.3.4.3. Consultar modelos cargados

El nombre de todos los modelos cargados en el simulador, se puede consultar escribiendo el comando **GET MODELS** desde el intérprete. El *Controller* obtendrá todos los modelos cargados en la instancia de *Simulator*, y devolverá la lista de todos los nombres de ellos al *Interpreter*.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en los diagramas de secuencia de la Figura 65.

3.3.4.4. Consultar campos y operaciones publicadas

La aplicación permite consultar el nombre de los campos publicados por todos los modelos cargados escribiendo desde el intérprete **GET FIELDS**. Si lo que se desea es consultar las operaciones publicada, se deberá escribir **GET OPERATIONS**.

Para consultar los campos publicados por un modelo determinado, se escribe **GET FIELDS** seguido del nombre del modelo. De la misma manera, para consultar las operaciones, se escribe **GET OPERATIONS** seguido del nombre del modelo.

```
>> get models
+ Gyroscope
+ StarTracker
+ ReactionWheel
+ AdcsController

>> get fields ReactionWheel
+ torque

>> get operations Gyroscope
+ SetOmega
```

Figura 26. Ejecutar varias operaciones de consulta en el intérprete SMP2.

El *Controller* básicamente accederá a la instancia de *Publication* del entorno de simulador para obtener los campos u operaciones publicadas.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en los diagramas de secuencia de la Figura 66 y Figura 67.

3.3.4.5. Consultar tiempos de simulación

Para consultar el tiempo actual de simulación, el tiempo de época, y el tiempo de misión, se escribe el comando **GET TIMES** desde el intérprete. El *Controller* accederá al *TimeKeeper* a través de *Simulator*, y recuperará los valores actuales de estos tiempos en nanosegundos.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en el diagrama de secuencia de la Figura 68.

3.3.4.6. Modificar tiempo de época y de misión

La modificación del valor del tiempo de época (relativo al tiempo de simulación) puede realizarse escribiendo **SET EPOCH** y el nuevo valor en nanosegundos. Para modificar el valor del tiempo de misión (relativo al tiempo de época) se escribe **SET MISSION** y el tiempo en nanosegundos.

El controlador modificará los valores almacenados en la instancia de *TimeKeeper* por medio del *Simulator*.

```
>> run
1895362000 | Simulator | Information | Simulation executing

2572562000 | Simulator | Information | Simulation paused

>> get times
+ Simulation time: 2572562000
+ Epoch time: 2572562000
+ Mission time: 2572562000

>> set epoch 2000000000
The initial epoch time was updated successfully

>> get times
+ Simulation time: 2572562000
+ Epoch time: 2772562000
+ Mission time: 2772562000
```

Figura 27. Consultar y modificar los tiempos de simulación en el intérprete SMP2.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en los diagramas de secuencia de la Figura 73 y la Figura 74.

3.3.4.7. Consultar y modificar el valor de un campo publicado

La consulta de un campo de tipo básico publicado por un modelo se realiza mediante el comando **GET VALUE** seguido del nombre del modelo y del nombre del campo publicado a obtener. Para modificar el valor del campo, se escribe **SET VALUE** seguido del nombre del modelo, el nombre del campo publicado y el nuevo valor.

Para los campos de tipo array, la consulta se ejecuta con el comando **GET ARRAY** seguido del nombre del modelo, el nombre del array y la posición del valor a obtener. Para modificar un elemento del array, es necesario escribir **SET ARRAY** seguido del nombre del modelo, el nombre del array, la posición a modificar y el nuevo valor.

El *Controller* accederá a la instancia de *Publication* del entorno de simulación a través del *Simulator*, y obtendrá o modificará los valores de acuerdo a los parámetros recibidos.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en los diagramas de secuencia de la Figura 69, la Figura 70, la Figura 71 y la Figura 72.

3.3.4.8. Ejecutar una operación publicada

Para ejecutar operaciones publicadas por cualquier modelo se debe escribir el comando **EXECUTE**, el nombre del modelo, el nombre de la operación y el conjunto de valores a pasar como parámetros a la operación. El intérprete devolverá un mensaje de confirmación si todo ha ido bien, además del valor devuelto por la operación.

El *Controller* accederá a la instancia *Publication* a través de *Simulator*, que buscará si existe una operación publicada para el modelo especificado. En caso afirmativo, se delega la ejecución del método al modelo a partir del nombre de la operación.

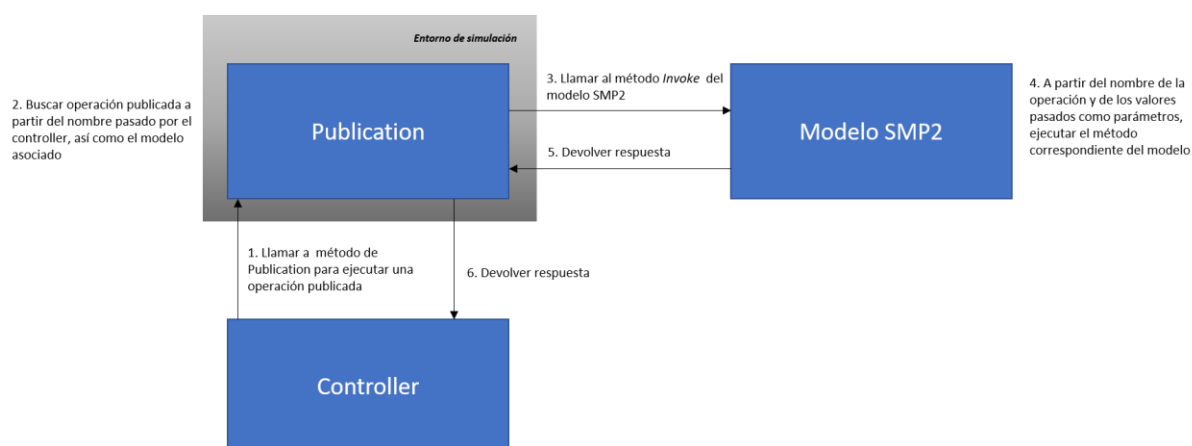


Figura 28. Ejecución de una operación publicada desde la clase Controller.

Se puede consultar el comportamiento a nivel de clases de esta funcionalidad en el diagrama de secuencia de la Figura 75.

3.4. Modelo SMP2 de ejemplo

Para realizar las pruebas sobre el entorno de simulación, se ha implementado un modelo SMP2 que cubre varias de las funcionalidades desarrolladas. Este modelo será pasado como entrada al intérprete SMP2 explicado en el siguiente apartado, que utiliza como motor el entorno de simulación implementado.

3.4.1. Descripción del modelo

El modelo se basa en la implementación de un contador sencillo, al que se le han añadido varias funcionalidades con el fin de testear algunas de las siguientes capacidades del entorno de simulación.

- Publicación de campos de tipos sencillos.
- Publicación de arrays de tamaño fijo.
- Planificación de ejecución cíclica de puntos de entrada.
- Registro de punto de entrada a eventos.
- Envío de mensajes.
- Obtención de tiempo de simulación.

3.4.1.1. Campos publicados

El modelo implementado contiene los siguientes campos, que son publicados dentro del método *Publish()*.

- Un contador (***counter***) de tipo *Int64* que incrementa su valor cada cierto número configurable de nanosegundos (por defecto se actualiza cada segundo)
- Un array (***bufferTimes***) de 80 elementos de tipo *Duration* que almacena los tiempos de simulación en los que se ha actualizado el contador.
- Un campo de tipo *Duration* (***lastSimulationTime***) que almacena el último tiempo de simulación en el que se ha actualizado el contador.

3.4.1.2. Métodos publicados

Se han desarrollado las siguientes operaciones en el modelo, que son también publicadas cuando el motor de simulación ejecuta el método *Publish()*.

- El método ***SetCounter*** para modificar el valor de *counter*.
- El método ***SetPeriodTime*** para cambiar el periodo en el que el valor del contador es actualizado.

3.4.1.3. Puntos de entrada

El modelo inicializa en el constructor dos puntos de entrada.

- El punto de entrada ***entryPointCounter***, asociado al método *Count* que realiza las siguientes acciones.

- Incrementa el campo *counter* a 1.
 - Actualiza el campo *lastSimulationTime* al tiempo de simulación actual.
 - Guarda el tiempo de simulación actual en el array *bufferTimes*.
 - Manda un mensaje al *Logger*.
- El punto de entrada ***entryPointReset***, asociado al método Reset que realiza las siguientes acciones.
 - Cambia el valor de *counter* a 0.
 - Manda un mensaje al *Logger*.

3.4.1.4. Ejecución de puntos de entrada

- Se ha programado un evento que ejecuta el punto de entrada ***entryPointCounter*** de forma cíclica. El periodo de ejecución es configurable mediante el método *SetPeriodTime*, siendo inicialmente 1 segundo.
- El punto de entrada ***entryPointReset*** se ha registrado al evento ***SMP_EnterStandbyId***, lo que significa que se ejecutará siempre que el simulador cambie al estado Standby desde cualquier otro estado.

3.4.2. Diagrama de clases

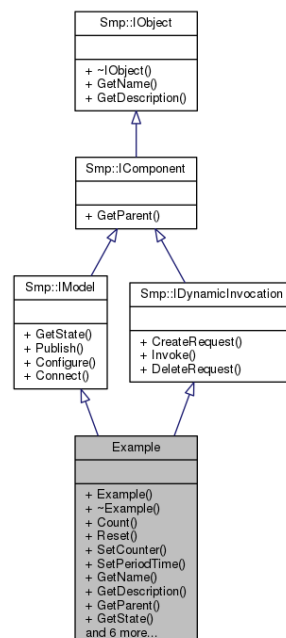


Figura 29. Diagrama de clases del modelo SMP2 de ejemplo.

3.4.3. Interfaces

Tal y como se aprecia en la Figura 29, el modelo implementa las interfaces *IModel* e *IDynamicInvocation*, por lo que debe implementar los métodos de ambas interfaces. Además, es necesario crear una clase para definir los puntos de entrada que deberá implementar la interfaz *IEntryPoint*.

3.4.3.1. Métodos de la interfaz IModel

La interfaz *IModel* exige implementar los siguientes cuatro métodos.

- ***GetState()***. Devuelve el estado actual del modelo.
- ***Publish(IPublication* receiver)***.
 - Verifica que el estado del modelo es *Created*. Si es así, se actualiza el estado a *Publishing*. En otro caso, se lanza una excepción de tipo *InvalidModelState*.
 - Publica los métodos y operaciones descritos anteriormente.
 - Guarda la instancia de *Publication* que se recibe como parámetro.
- ***Connect(ISimulator* simulator)***.
 - Verifica que el estado del modelo es *Configured*. Si es así, se actualiza el estado a *Connected*. En otro caso, se lanza una excepción de tipo *InvalidModelState*.
 - Guarda las instancias de los servicios básicos (*Logger*, *Scheduler*, *EventManager*, *TimeKeeper*) a partir del objeto *ISimulator* pasado como parámetro.
 - Configura en el *Scheduler* la ejecución cíclica del punto de entrada *entryPointCounter*.
 - Configura en el *EventManager* la ejecución del punto de entrada *entryPointReset* cuando el simulador pase al estado *Standby*.
- ***Configure(ILogger* logger)***.
 - Verifica que el estado del modelo es *Publishing*. Si es así, se actualiza el estado a *Configured*. En otro caso, se lanza una excepción de tipo *InvalidModelState*.

3.4.3.2. Métodos de la interfaz IDynamicInvocation

La interfaz *IDynamicInvocation* debe ser implementada por todos aquellos modelos que soporten la ejecución dinámica de cualquiera de sus métodos publicados. Como vimos en el apartado 3.4.1.2, nuestro modelo tiene dos métodos publicados, por lo que es necesario implementar los métodos de esta interfaz.

- **CreateRequest(String8 operationName).** Devuelve un objeto *IRequest* que contiene los metadatos necesarios para realizar la ejecución dinámica de la operación. En nuestro modelo, se delega la creación de este objeto a *IPublication*, concretamente llamando al método del mismo nombre.
- **Invoke(IRequest *request).** A partir del objeto generado con el método anterior, se implementan los pasos necesarios para ejecutar un método publicado.
 - Verifica si el nombre del método se corresponde a alguno de los métodos publicados.
 - Verifica si el número de parámetros pasados al método es correcto. Si no es el caso, se lanza la excepción *InvalidParameterCount*.
 - Verifica si el número de parámetros pasados al método es correcto. Si no es el caso, se lanza la excepción *InvalidParameterCount*.
 - Verifica si el tipo de cada uno de los parámetros pasados al método es correcto. Si no es el caso, se lanza la excepción *InvalidParameterType*.
- **DeleteRequest(IRequest *request).** Libera los recursos del objeto creado con el método *CreateRequest*. Nuevamente, se delega la destrucción del objeto a *IPublication*.

3.5. Integración de entorno de simulación en Matlab

Con el entorno de simulación, el controlador que accede a él, y el intérprete que permite al usuario interactuar con el simulador, llegamos a uno de los pasos clave de este proyecto, que consiste en la integración del sistema SMP2 desarrollado en Matlab.

Como se explicó en la introducción de este documento, en las empresas de satélites suelen convivir dos grupos diferenciados de programadores; los modeladores (que suelen utilizar Matlab) y los desarrolladores de Software (que suelen utilizar C++ y estándares como SMP). La ejecución de un motor SMP2 en Matlab, así como la posibilidad de interactuar con el entorno de simulador, facilitará el trabajo de ambos equipos.

Para realizar esta integración, se han seguido una serie de pasos que son explicados en los siguientes apartados.

3.5.1. Implementación de interfaz externa en C++

Se ha implementado un conjunto de métodos en C++ para ser compilados como una librería dinámica, y posteriormente cargados en Matlab mediante la función **loadlibrary**. Esta función recibe como parámetros la ruta de la librería dinámica a cargar y un fichero de cabecera con la definición de los métodos publicados de la librería. Un aspecto importante es que este fichero de cabecera debe estar escrito en C, por lo que no se permite la importación de bibliotecas de C++ en el mismo, como por ejemplo la biblioteca estándar (definida en el namespace *std*). No

obstante, la implementación de los métodos si admite el uso de características de C++. El contenido del fichero de cabecera se presenta en el Listado 1.

La implementación de la interfaz externa crea una instancia de la clase *Interpreter*, que será utilizada por los métodos publicados. Se explica a continuación el propósito y el funcionamiento de cada uno de estos métodos.

- ***loadAssembly(const char* assemblyFile)***. Carga una librería dinámica con un ensamblado de modelos.
- ***run()***. Ejecuta la simulación.
- ***hold()***. Para la simulación.
- ***isExecuting()***. Devuelve si el simulador está en estado de ejecución.
- ***getLogsCount()***. Obtiene el número total de mensajes recibidos por el servicio *Logger*.
- ***getLogs(uint_64_t index)***. Obtiene la cadena de mensajes recibidos por el servicio *Logger* a partir de la posición especificada. Este método efectúa los siguientes pasos.
 - Obtiene el controlador de la instancia de la clase *Interpreter*.
 - Llama al método *getLogMessages* del controlador.
 - Obtiene la cadena de cada uno de los mensajes recibidos.
 - Construye una cadena concatenando las cadenas del paso anterior.
- ***processCommand(const char* command)***. Procesa el comando recibido como parámetro, y obtiene la respuesta de este. Llama directamente al método que procesa los comandos de la instancia de la clase *Interpreter*.
- ***close***. Cierra el ensamblado cargado (si lo hubiera).

```
#include <inttypes.h>

int loadAssembly(const char* assemblyFile);
bool init();
void run();
void hold();
bool isExecuting();
int64_t getSimulationTime();
uint64_t getLogsCount();
const char* getLogs(uint64_t index);
const char* processCommand(const char* command);
void close();
```

Listado 1. Contenido de header de interfaz externa.

Como se puede comprobar, se ha aprovechado la arquitectura implementada en C++, con el fin de que no sea necesario implementar código duplicado en la interfaz externa.

Tras cargar la librería dinámica con **loadlibrary**, se puede llamar a los anteriores métodos utilizando la función **calllib**. En el Listado 2 se presenta un script en Matlab con un ejemplo de uso, donde se ejecutan los siguientes pasos.

- Carga de librería dinámica de interfaz externa con método **loadlibrary**.
- Llamada al método *loadAssembly* para cargar el ensamblado contenido en el fichero *libExample.dll* (contiene el ensamblado con el modelo explicado en el apartado 3.4).
- Llamada al método *run* para ejecutar el simulador.
- Espera de 30 segundos.
- Llamada al método *hold* para parar el simulador.
- Llamada al método *getLogs* para obtener todos los mensajes recibidos por el servicio *Logger* recibidos durante la ejecución del simulador.
- Llamada al método *processCommand* con el parámetro '*get value Example_SMP2 lastSimulationTime*', para obtener el valor del campo *lastSimulationTime* del modelo *Example_SMP2*.
- Llamada al método *close* para cerrar el ensamblado.
- Descarga de librería dinámica de interfaz externa usando el método **unloadlibrary**.

```
loadlibrary('extern.so', 'extern.h', 'alias', 'extern');
calllib('extern', 'loadAssembly', 'libExample.dll');
calllib('extern', 'run');
pause(30);
calllib('extern', 'hold');
logs = calllib('extern', 'getLogs', 0);
lastSimulationTime = calllib('extern', 'processCommand', 'get value
Example_SMP2 lastSimulationTime');
calllib('extern', 'close');
unloadlibrary('extern')
```

Listado 2. Script matlab con ejemplo de uso de la interfaz externa.

Se ha implementado una clase en Matlab (*smp2manager*) que abstrae al desarrollador de todas estas llamadas, y gestiona la lógica del simulador.

3.5.2. Implementación de intérprete SMP2 en Matlab

Partiendo de la clase *smp2manager* implementada en el apartado anterior, se ha desarrollado una aplicación GUI con un intérprete en Matlab, utilizando el entorno GUIDE.

Las funcionalidades de este intérprete son las mismas que las desarrolladas en el apartado 3.3., ya que ambos utilizan las mismas clases para acceder al motor de simulación. Los casos de uso son también muy parecidos (ver Anexo 3).

La pantalla principal de la aplicación desarrollada está dividida en las siguientes secciones.

- Sección de **control de la simulación**, donde se encuentran dos botones para ejecutar y parar la simulación y una etiqueta que muestra el valor en segundos del tiempo actual de simulación.
- Sección de **log**, donde se muestran los mensajes recibidos por los diferentes componentes de la simulación.
- Sección de **línea de comandos**, donde el usuario puede ejecutar cualquiera de los comandos aceptados por el intérprete.
- Sección de **salida de comandos**, donde se muestra la salida de los comandos introducidos por el usuario.

Se muestra en la Figura 30 el aspecto final de la aplicación, donde se pueden identificar fácilmente las secciones explicadas.

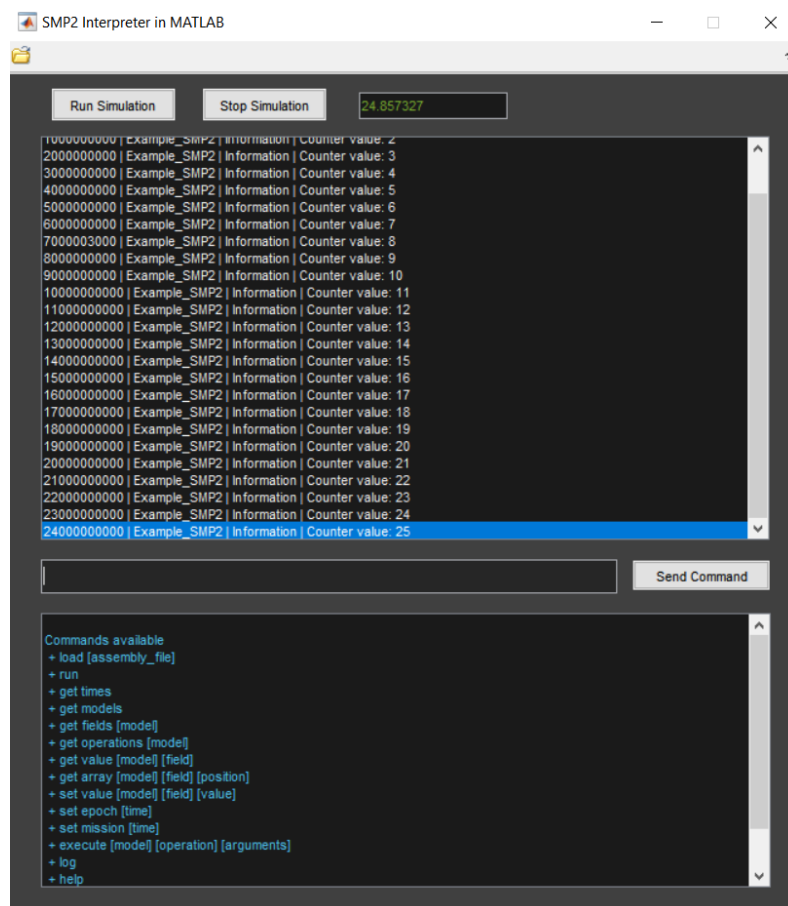


Figura 30. Aplicación GUI con intérprete SMP2 en Matlab.

3.6. Implementación de sistema ADCS

3.6.1. Introducción

Los sistemas de control de orientación y actitud (ADCS) son componentes esenciales a la hora de abordar el diseño de un satélite. Es importante que este sistema sea capaz de corregir los errores de actitud de forma rápida y precisa. Además del algoritmo de control, es muy importante que los sensores sean precisos y con el mínimo margen de error; por este motivo, se suelen usar varios sensores cuyas salidas serán procesadas por el software de a bordo para obtener la posición y la orientación del satélite. Estos sistemas necesitan ser verificados a fondo por el equipo de diseño del satélite, por lo que la implementación de modelos que simulen sus componentes es un aspecto muy importante, ya que permite realizar las pruebas antes de instalar el módulo en el sistema real.

Los sistemas ADCS reales son muy complejos y contienen una gran cantidad de componentes. Se pueden consultar las bases de control de algunos de estos sistemas en los artículos (11), (12) o (23) de la bibliografía. En este apartado, se ha desarrollado un módulo simplificado, con sensores ideales que devolverán la posición y la orientación real del satélite. Se ha implementado un modelo dinámico del satélite que devuelve un conjunto de valores que definen su estado en cada momento, y un algoritmo de control basado en cuaterniones para controlar su orientación.

3.6.2. Componentes del sistema

Tras completar la integración del entorno de simulación SMP2 en Matlab, se procede a diseñar un sistema ADCS con el objetivo de ser utilizado como caso de prueba de todo el sistema. Este sistema ADCS constará de los siguientes componentes.

- El modelo matemático del satélite, que tomará como entrada la **velocidad inicial**, la **posición inicial**, la **orientación inicial**, la **velocidad angular inicial**, la **inercia** del satélite, la **masa** del satélite, las **fuerzas** aplicadas sobre el satélite debido al modelo gravitatorio y el **torque** aplicado por las ruedas de reacción. Devolverá como salida la **orientación**, la **posición**, la **velocidad** y la **aceleración** del satélite.
- Un conjunto de modelos SMP2 con los sensores (**gyroscope** y **star tracker**), actuador (**reaction wheel**) y controlador ADCS (**ADCS Controller**).

Una vez definidos estos componentes, se implementa un script en Matlab que ejecutará la siguiente secuencia de pasos.

- Inicialización de motor de simulación con un ensamblado que contenga los modelos SMP2 del sistema ADCS.

- Inicialización de los parámetros iniciales de los modelos matemáticos para el satélite y las ruedas de tracción.
- Ejecución del lazo de control, donde se realizarán las siguientes acciones en cada paso de simulación.
 - Simulación de los modelos matemáticos durante un tiempo determinado, tomando como condiciones iniciales los valores actuales del estado del satélite y como torque a aplicar el valor contenido en el modelo SMP2 *reaction wheel*.
 - Actualización del modelo SMP2 *gyroscope* a partir de la salida correspondiente a la **velocidad angular** del modelo matemático del satélite.
 - Actualización del modelo SMP2 para el *star tracker* a partir de la salida correspondiente a la **orientación** del modelo matemático del satélite
 - Llamada a una rutina del modelo SMP2 *ADCSController* que ejecutará la siguiente secuencia de pasos⁵.
 - Obtener los valores de los sensores.
 - Ejecutar el algoritmo de control a partir de un valor de referencia y de los valores obtenidos en el paso anterior.
 - Establecer el torque en el modelo SMP2 para el *reaction wheel*.

En la figura, se muestra el esquema del sistema implementado. El algoritmo de control para obtener el torque de salida se ejecutará en el modelo SMP2 *ADCS Controller*, a partir de los valores de la orientación y la posición angular obtenidos de los sensores.

Esta arquitectura nos permite aprovechar la potencia de ambos sistemas. En entornos reales, será factible simular los complejos modelos SMP2 implementados a partir de la definición de las interfaces de los subsistemas del satélite, con los cálculos matemáticos desarrollados por los ingenieros de control.

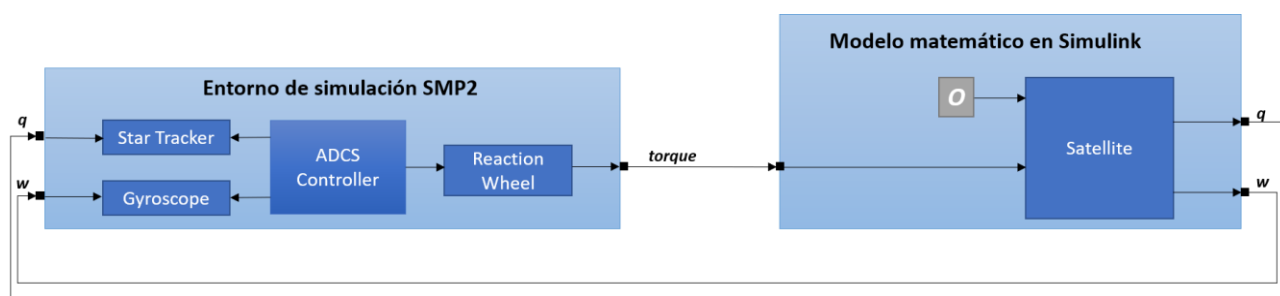


Figura 31. Diagrama de sistema de control ADCS implementado.

⁵ El modelo permite también ejecutar esta tarea de forma periódica, en un hilo independiente al del principal bucle de ejecución. En este caso, conviene ajustar los tiempos de simulación y de rutina de control.

3.6.3. Modelos SMP2 de módulo ADCS

Para la implementación de los componentes del módulo ADCS se utilizará el estándar SMP2, con el propósito de ser simulados con nuestro entorno.

Se ha implementado una clase genérica de la que heredarán cada uno de los modelos, que codifica una serie de pasos comunes para varios métodos. Se muestra su diagrama de herencias en la Figura 32.

Para los sensores, se ha partido de la base de que son modelos ideales, por lo que estos almacenarán en cada instante de tiempo los valores actuales de la dinámica del satélite. Por lo tanto, el giroscopio devolverá la velocidad angular del satélite, y el *star tracker* la posición.

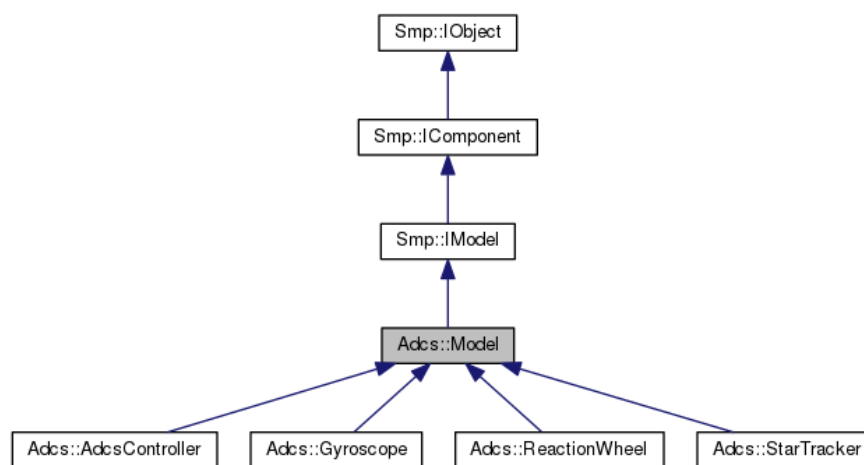


Figura 32. Diagrama de herencias de la clase genérica *Adcs::Model*.

3.6.3.1. Gyroscope

El modelo SMP2 para el giroscopio, contiene los siguientes elementos.

- Un array de 3 elementos que almacena el vector con la velocidad angular del satélite.
- Una operación publicada **SetOmega** que establece el valor de los elementos del vector que define la velocidad angular del satélite.
- Una operación **GetOmega** que devuelve el valor de los elementos del vector que define la velocidad angular del satélite.

3.6.3.2. Star Tracker

El modelo SMP2 para el star tracker, contiene los siguientes elementos.

- Un array de 4 elementos que almacena el cuaternión que representa la orientación del satélite con respecto al sistema de referencia ECI.
- Una operación publicada **SetQuaternion** que establece el valor de los elementos del cuaternión.
- Una operación **GetQuaternion** que devuelve el valor de los elementos del cuaternión.

3.6.3.3. Reaction wheel

El modelo SMP2 para las ruedas de reacción, contiene los siguientes elementos.

- Un campo publicado (sólo lectura) con un array de 3 elementos que almacena el torque a aplicar.

3.6.3.4. ADCS Controller

El modelo SMP2 para el controlador ADCS representa la parte del software de a bordo que procesa los valores de los sensores, ejecuta el algoritmo de control y manda la salida a las ruedas de reacción.

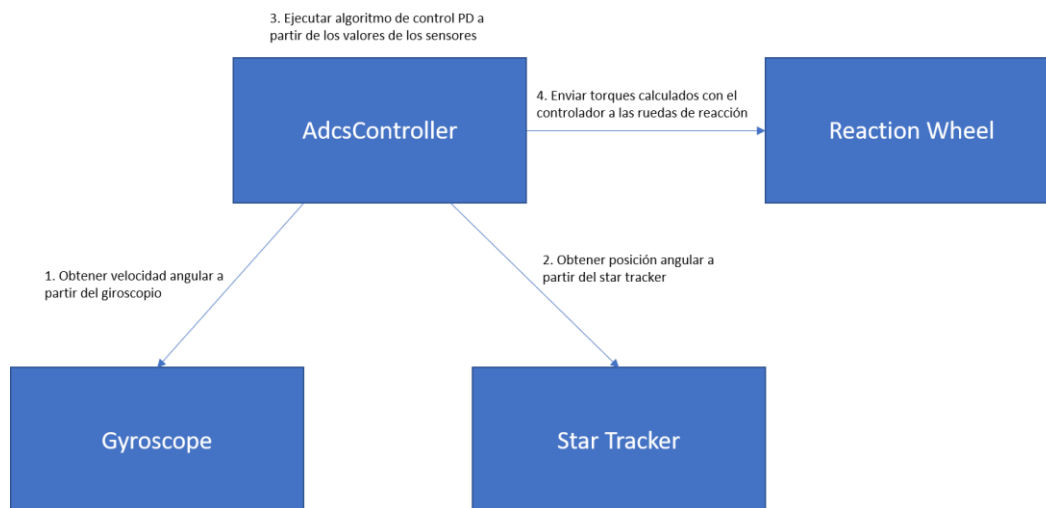


Figura 33. Pasos de control implementados por el controlador ADCS..

Este modelo se compone de los siguientes elementos.

- Una referencia a la instancia del *giroscopio* utilizada en la simulación.
- Una referencia a la instancia del *star tracker* utilizada en la simulación.
- Una referencia a la instancia de las *ruedas de reacción* utilizada en la simulación.

- Una operación publicada **ExecuteControlStep** que ejecuta la siguiente serie de pasos.
 - Llamada al método *AcquireSensorsData*, que obtiene la orientación, posición y velocidad del satélite a partir de las instancias del giroscopio, *star tracker* y ruedas de reacción.
 - Llamada al método *ExecutePd*, que ejecuta el algoritmo PD de control de acuerdo a los valores obtenidos de los sensores para obtener el torque a aplicar en el actuador.
 - Llamada al método *SendTorqueRw*, que actualiza el torque de las ruedas de reacción.
- Un punto de entrada **entryPointStep**, asociada a la operación explicada en el paso anterior.
- Una operación publicada **EnableCyclicalControlEvent** que establece una tarea periódica que ejecuta la operación *ExecuteControlStep* cada cierto número de nanosegundos.
- Una operación publicada **DisableCyclicalControlEvent** que deshabilita la tarea periódica para ejecutar la operación *ExecuteControlStep*.

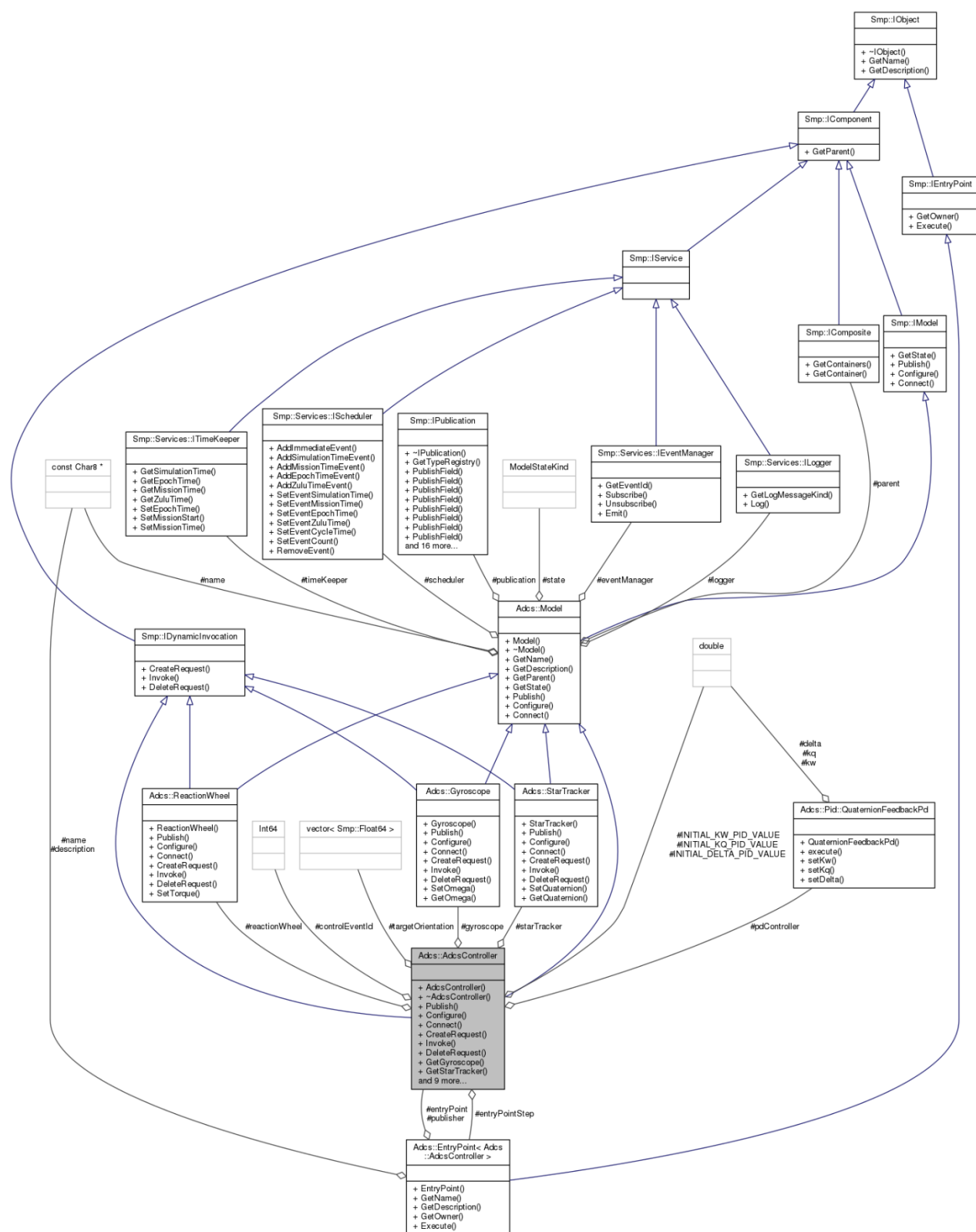


Figura 34. Diagrama de clases del modelo SMP2 para el controlador ADCS.

3.6.4. Modelo dinámico en Simulink

Se muestra en la Figura 35 el diagrama en Simulink del modelo que implementa la dinámica del satélite. Se resaltan tres módulos diferenciados en el mismo, que se explican en los siguientes apartados.

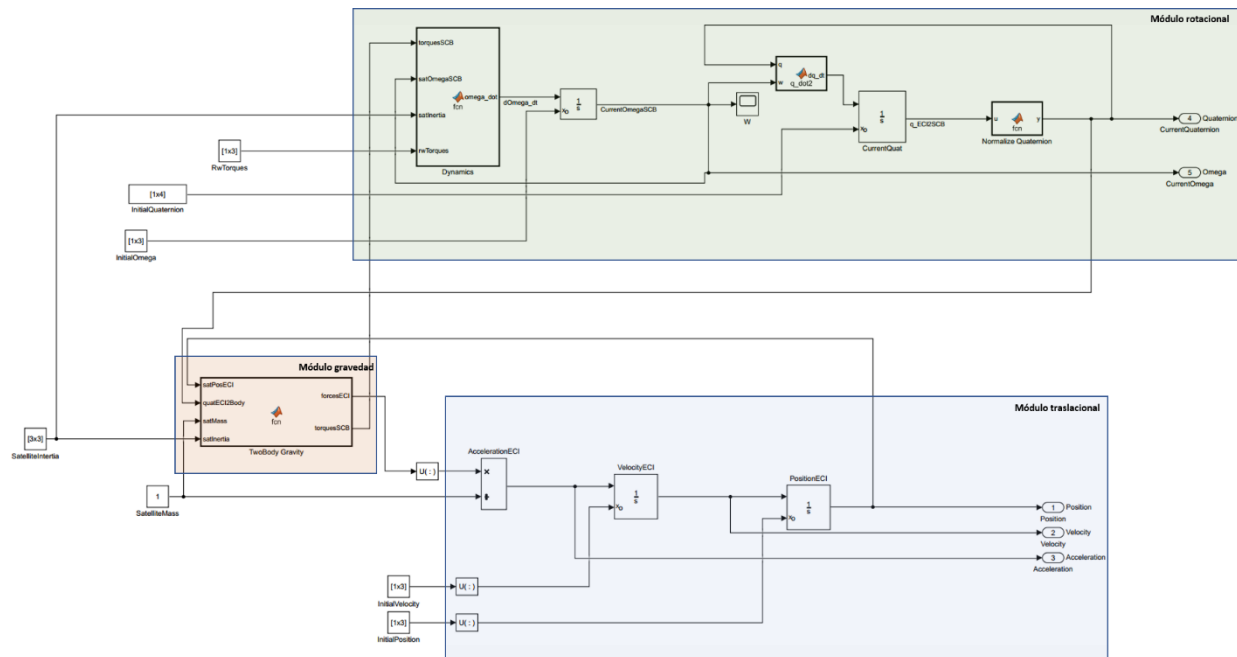


Figura 35. Diagrama en simulink del modelo dinámico del satélite.

3.6.4.1. Módulo de gravedad

Implementa un modelo de gravedad de la tierra, donde se obtienen los **vectores de fuerzas** a aplicar sobre el **modelo rotacional** (torque) y el **modelo traslacional** a partir de las siguientes entradas.

- Posición actual del satélite.
- Orientación actual del satélite con respecto al sistema de referencia ECI.
- Masa del satélite.
- Matriz de inercia del satélite.

Se presenta las ecuaciones implementadas de este modelo en el código del Listado 3.

```
function [forcesECI, torquesSCB] = fcn(satPosECI, quatECI2Body, satMass, satInertia)
    GM = 398600441800000; % m3/s2
    satPosECI = [satPosECI(1) satPosECI(2) satPosECI(3)];
    satPosECINorm = norm(satPosECI);
    satPosECIt = satPosECI/satPosECINorm;
    forcesECI = -GM*satPosECIt*satMass/(satPosECINorm*satPosECINorm);
    satPosSCB = qvq(quatECI2Body, satPosECIt);
    torquesSCB = (3*GM)/(satPosECINorm*satPosECINorm*satPosECINorm)* cross(satPosSCB,
    satInertia*satPosSCB');
```

Listado 3. Función en Matlab para obtener las fuerzas ejercidas por el modelo gravitatorio de la tierra sobre el satélite.

3.6.4.2. Módulo traslacional

Este módulo tiene como salida la **aceleración, velocidad y posición** del satélite. Son calculadas a partir de cuatro entradas.

- Posición inicial del satélite.
- Velocidad inicial del satélite.
- Fuerza ejercida por el modelo gravitatorio de la tierra.
- Masa del satélite.

Para obtener estas salidas, se aplica la ecuación de la segunda ley de Newton

$$F = m \cdot a$$

$$F = m \cdot \dot{v}$$

$$F = m \cdot \ddot{s}$$

Donde

- F = Fuerza ejercida sobre el satélite.
- m = Masa del satélite.
- v = Velocidad del satélite.
- s = Posición del satélite.

A partir de esta ecuación, se obtiene en el primer bloque la **aceleración** de satélite, dividiendo la fuerza gravitatoria por la masa del satélite. En los dos siguientes bloques, se utilizan integradores para obtener la **velocidad** y la **posición**.

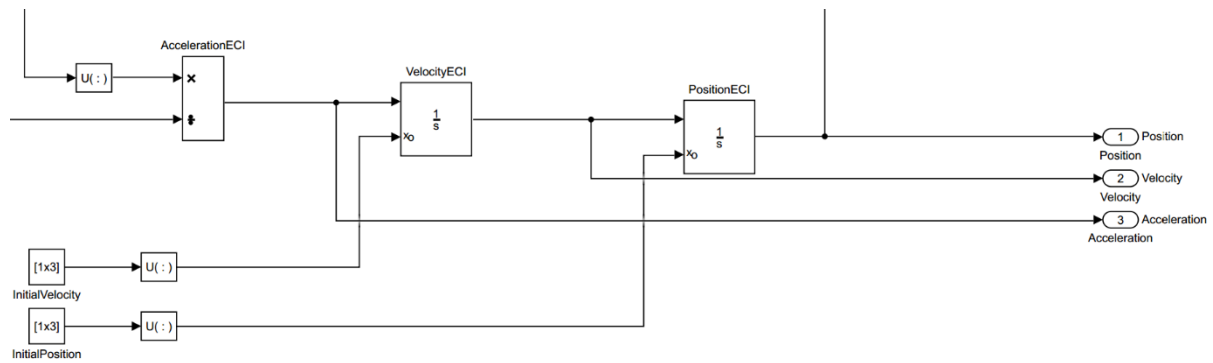


Figura 36. Modelo traslacional del satélite.

3.6.4.3. Módulo rotacional

Es el modelo más complejo de los tres. Sus salidas son un **cuaternión** con la **orientación** del satélite con respecto al sistema de referencia ECI de la tierra, y un vector con la **velocidad angular** del satélite. Son calculadas a partir de las siguientes entradas.

- Torque ejercido por el modelo gravitatorio terrestre.
- Velocidad angular inicial del satélite.
- Cuaternión con la orientación inicial del satélite.
- Matriz de inercia del satélite.
- Torque ejercido por cada una de las tres ruedas de reacción.

El primer bloque, obtiene la **aceleración angular** a partir de la matriz de inercia del satélite, el torque gravitatorio, la velocidad angular del satélite y las fuerzas ejercidas por las ruedas de reacción. Las ecuaciones utilizadas en este bloque se muestran en el código del Listado 4.

```
function omega_dot = fcn(torquesSCB, satOmegaSCB, satInertia, rwTorques)
    satInertia_dot = 0;
    totTorquesSCB = torquesSCB - cross(satOmegaSCB, satOmegaSCB*satInertia) - rwTorques' -
    satInertia_dot;
    omega_dot = totTorquesSCB/satInertia;
```

Listado 4. Función en Matlab para obtener la aceleración angular del satélite.

Integrando la aceleración calculada en el paso anterior, se obtiene la **velocidad angular** del satélite. Este valor se redirige a una de las salidas del modelo.

En el siguiente paso, se construye un cuaternión a partir del vector de velocidad angular, añadiendo un 0 como valor escalar del mismo. Posteriormente, se realiza la multiplicación entre este cuaternión y el cuaternión con la orientación actual del satélite. Dividiendo el

resultado por dos, obtenemos un cuaternión con la velocidad angular del satélite. Si lo integramos, se obtiene el cuaternión que representa la **orientación** del satélite con respecto al sistema de referencia ECI.

Por último, se normaliza este cuaternión, obteniendo otra de las salidas del modelo.

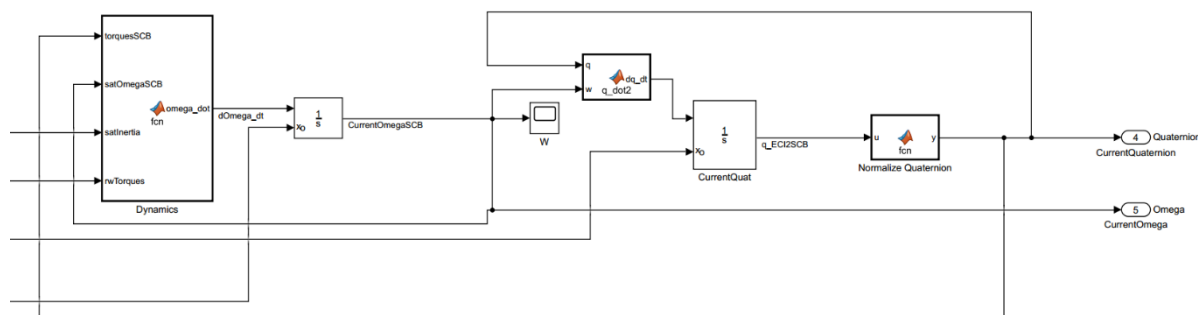


Figura 37. Modelo rotacional del satélite.

3.6.5. Algoritmo de control PD

El objetivo del algoritmo de control diseñado es mantener la orientación del satélite en unos valores determinados. El controlador se ha implementado en una clase C++, y se utiliza en modelo SMP2 *AdcsController*, tal y como se vio en la sección 3.6.3.4.

Durante la realización de este proyecto, se barajó la idea de utilizar otro tipo de controladores (como PID clásico o fuzzy). Finalmente, tras analizar la problemática y después de varios ensayos, se optó por este tipo de control.

También se pensó en utilizar ángulos de Euler para la representación para la orientación, pero se decidió trabajar con cuaterniones, principalmente para evitar algunas de las limitaciones asociadas al uso de los ángulos de Euler.

El algoritmo implementado se basa en la publicación (11). En este artículo se proponen algunas técnicas de control de actitud para el satélite estonio ESTCube-2, previsto para ser lanzado en 2019. Estos sistemas están pensados para ser utilizados en actuadores del propio satélite, como el torque magnético o las ruedas de reacción.

3.6.5.1. Error entre cuaterniones

Uno de los primeros problemas a los que nos debemos enfrentar en nuestra estrategia de control, es que la orientación a controlar por el sistema está representada en cuaterniones.

El cálculo del error entre dos orientaciones expresadas en cuaterniones no es tan inmediato como el de otras medidas. Si por ejemplo, queremos programar un sistema para corregir la posición de un objeto representada en coordenadas cartesianas, el cálculo de error entre ambas

coordenadas se realiza restando cada uno de sus componentes. Sin embargo, esta formulación no nos sirve para este caso.

Este error de orientación puede ser calculado mediante el producto de Kronecker entre q_{ref}^{-1} , que representa la inversa del cuaternión con la orientación deseada, y el cuaternión con la orientación devuelta por el sensor.

$$q_{error} = q_{ref}^{-1} \oplus q$$

El resultado de esta multiplicación expresado como productos de matrices, se define a continuación.

$$q_{error} = \begin{bmatrix} q_{ref_0} & q_{ref_1} & -q_{ref_2} & -q_{ref_3} \\ -q_{ref_1} & q_{ref_0} & q_{ref_3} & -q_{ref_2} \\ q_{ref_2} & -q_{ref_3} & q_{ref_0} & -q_{ref_1} \\ q_{ref_3} & q_{ref_2} & q_{ref_1} & q_{ref_0} \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

3.6.5.2. Parte proporcional del controlador

La parte proporcional hace uso del error de cuaternión (calculado en la sección 3.6.5.1) entre la orientación medida y la orientación deseada. Multiplicando la ganancia proporcional y este error, obtenemos la ecuación de control deseada.

$$K_p \cdot q_{error}$$

3.6.5.3. Parte derivativa del controlador

La parte derivativa del controlador toma como entrada la velocidad angular actual del satélite. Su funcionamiento se basa en el principio de que el torque producido por las ruedas de reacción para el control de actitud es opuesto a la tasa de cambio del momento angular de las ruedas de reacción. Esto se puede expresar con la siguiente ecuación.

$$T_c = -J \cdot \dot{\omega} = -\dot{H}$$

donde:

- T_c es el torque generado por el sistema.
- J es el momento de inercia.
- w es la velocidad angular del motor.
- H es el momento angular de las ruedas de reacción.

Por lo tanto, una sencilla ley de control que se puede inferir a partir de lo anterior es la multiplicación de la ganancia derivativa y la velocidad angular del satélite.

$$K_d \cdot \omega$$

3.6.5.4. Ley de control PD

Sumando la parte proporcional (P) y la derivativa (D), obtenemos por lo tanto la ecuación que gobernará nuestro controlador.

$$\text{torque} = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = K_p \cdot q_{error} + K_d \cdot \omega$$

3.6.5.5. Lazo de control implementado

A partir de las anteriores conclusiones, se implementa el algoritmo PD. Este algoritmo recibirá como entrada los valores calculados por el modelo dinámico rotacional del satélite (velocidad angular y orientación).

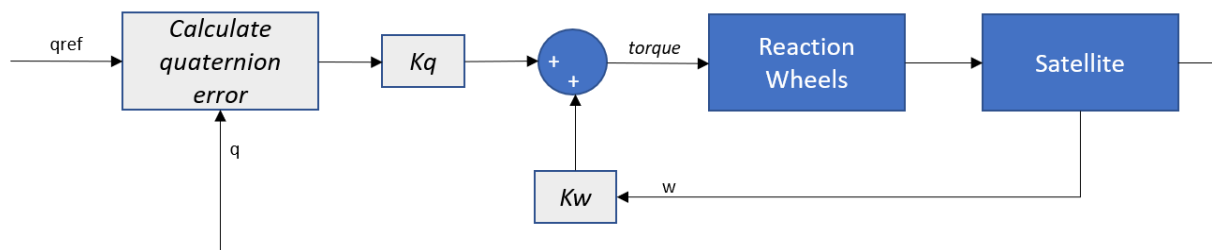


Figura 38. Lazo de control PD para control de orientación basado en cuaterniones.

En la Figura 38 se puede apreciar más claramente como funciona el lazo de control implementado. Básicamente, el algoritmo consta de los siguientes pasos.

- A partir del valor de la orientación actual y la orientación de referencia, se calcula el error de cuaternión.
- Se obtiene la parte vectorial del resultado de la ecuación anterior, y se multiplica por el coeficiente de ganancia K_q .
- Se multiplica la velocidad angular por el coeficiente de ganancia K_w .
- Se suman los dos resultados anteriores, obteniendo el torque a aplicar sobre las ruedas de reacción.

3.6.6. Implementación de GUI con el controlador ADCS

Utilizando nuevamente el entorno GUIDE, se ha implementado una interfaz gráfica que permite la ejecución en paralelo de los modelos dinámicos en Simulink y de los modelos SMP2. Dicha interfaz consta de los siguientes componentes.

- Dos botones en la parte superior de la interfaz para **ejecutar** o **parar** la simulación.
- Una sección situada en la parte izquierda de la interfaz donde se muestra el tiempo total de simulación y los valores devueltos por el modelo dinámico en **Simulink**, así como los torques a aplicar por las ruedas de reacción.
- Una sección situada en la parte derecha de la interfaz donde se muestra el log recibido por el motor de simulación SMP2.
- Debajo del log SMP2 existen varios controles donde se pueden definir los coeficientes de ganancia de control (K_p , K_d) y el cuaternión que define la orientación de referencia (Q_1 , Q_2 , Q_3 , Q_4).
- Dos gráficas en la parte inferior de la interfaz donde se muestra la evolución de los valores del torque aplicado por cada una de las ruedas de reacción, y de los cuatro elementos del cuaternión con la orientación del satélite.

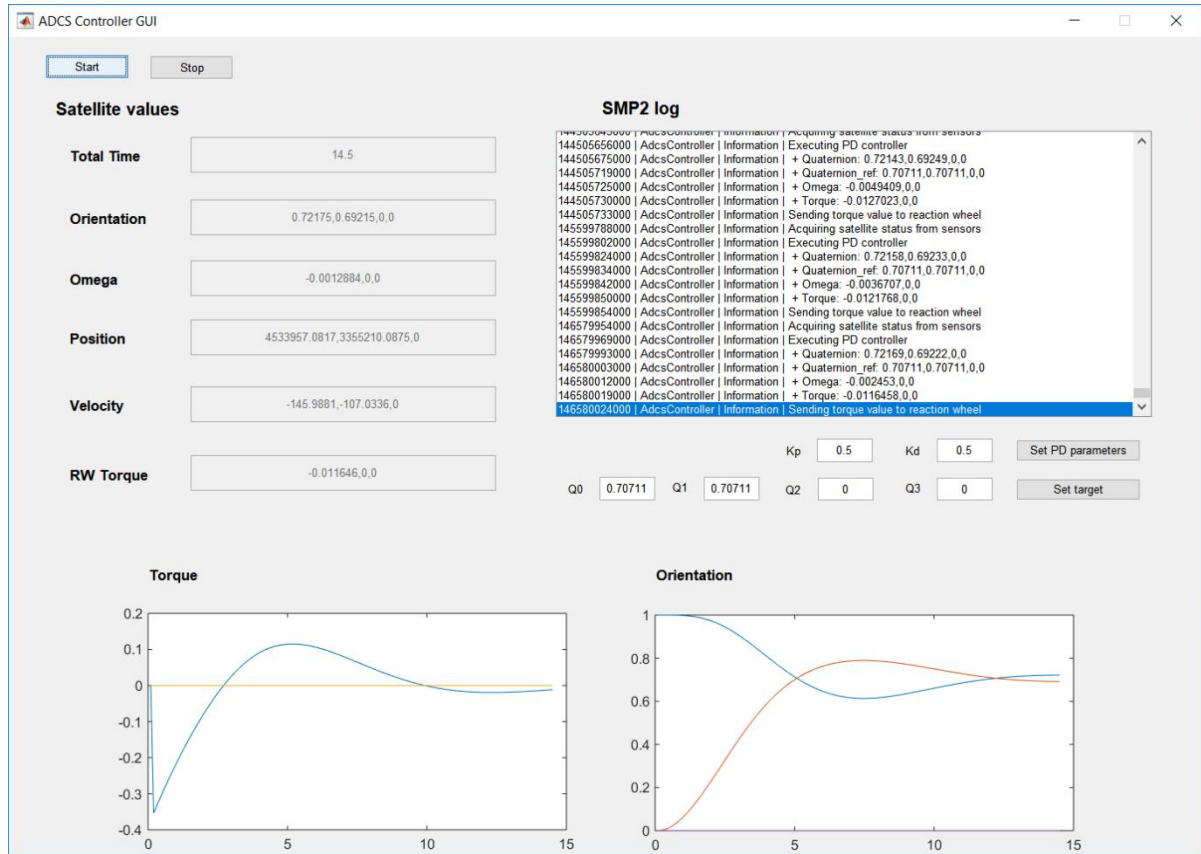


Figura 39. Aplicación GUI con controlador ADCS.

3.6.7. Resultados

Con la aplicación en el apartado anterior, el usuario puede experimentar con diferentes valores para los coeficientes de control, con el fin de estudiar la influencia de estos en la orientación del satélite.

No es el objetivo de este trabajo hacer un análisis detallado del algoritmo de control implementado, pero si se estudia el comportamiento del sistema para algunos valores de entrada para verificar el correcto funcionamiento del sistema ADCS desarrollado.

Los valores iniciales para el modelo dinámico se definen como siguen.

- $s_0 = (7e6 \cdot \cos(\frac{\sqrt{3}}{2}), 7e6 \cdot \sin(\frac{1}{2}), 0)$
- $v_0 = (0, 1, 0)$
- $m = 1$
- $Sat_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- $Q_i = (1, 0, 0, 0)$

- $\omega = (0, 0, 0)$
- $T_0 = (0, 0, 0)$

Donde:

- s_0 es la posición inicial del satélite con respecto a la tierra en metros.
- v_0 es la velocidad inicial del satélite en metros por segundo.
- m es la masa del satélite en kilogramos.
- Sat_i es la matriz de inercia del satélite en kilogramos por metro cuadrado.
- Q_i es el cuaternión que define la orientación del satélite con respecto a la tierra.
- ω es la velocidad angular inicial del satélite en radianes por segundo.
- T_0 es el torque inicial aplicado por las ruedas de reacción en Newtons por metro.

Definimos el ángulo de referencia a 90 grados sobre el eje X con respecto a la tierra. El cuaternión unitario que representa esta rotación es $Q_{ref} = (\cos(\frac{\pi}{2}), \sin(\frac{\pi}{2}), 0, 0)$.

Se ejecuta a continuación la aplicación con varios valores para los coeficientes de ganancia, y se analiza la respuesta del sistema de control. El tiempo de paso utilizado es de 0.1 segundos.

- $K_p = 1, K_d = 0.25$

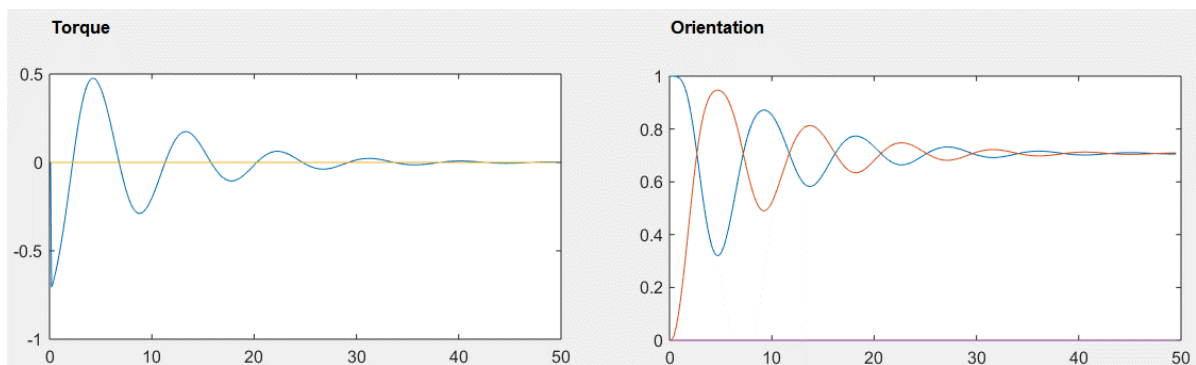


Figura 40. Resultados del control PD con $K_p=1$ y $K_d=0.25$.

- $K_p = 0.5, K_d = 0.5$

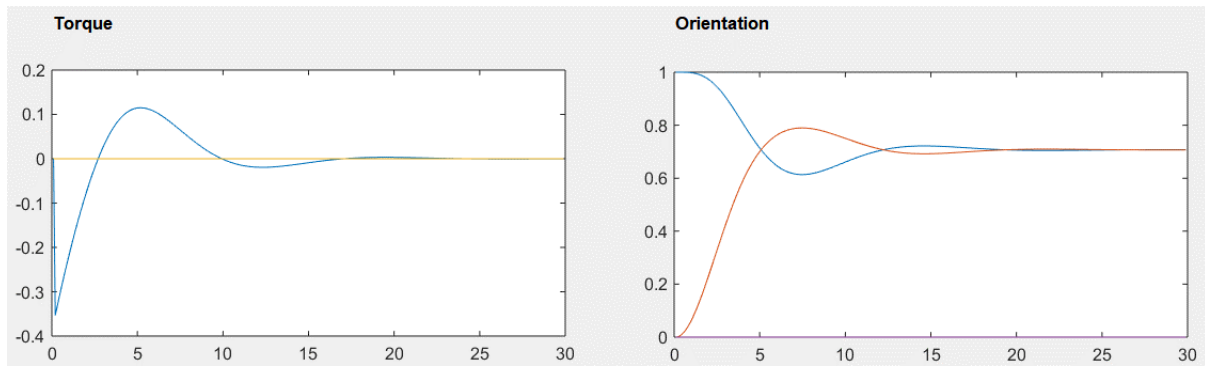


Figura 41. Resultados del control PD con $K_p=0.5$ y $K_d=0.5$.

- $K_p = 0.25, K_d = 1$

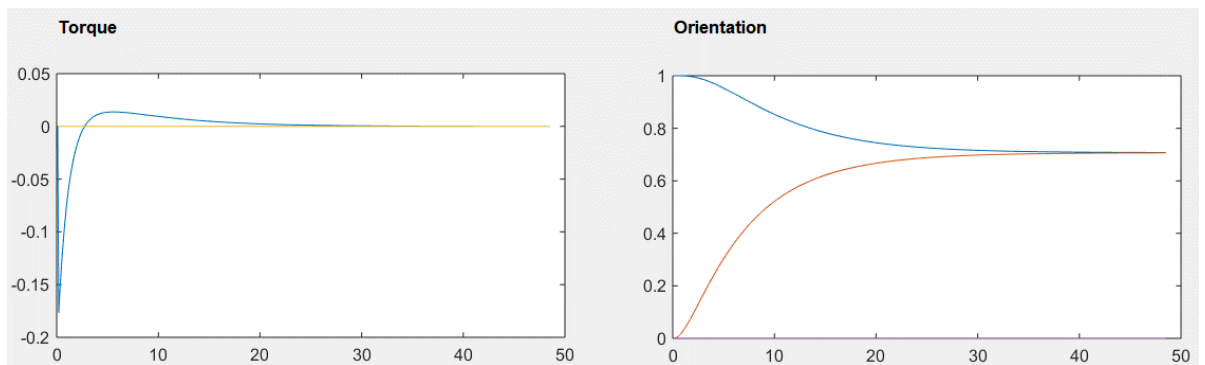


Figura 42. Resultados del control PD con $K_p=0.25$ y $K_d=1$

- $K_p = 0.1, K_d = 1$

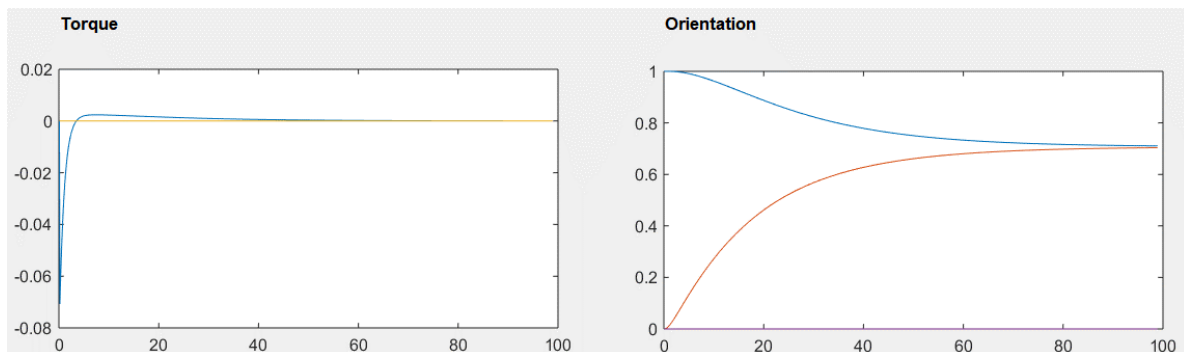


Figura 43. Resultados del control PD con $K_p=0.1$ y $K_d=1$

A partir de las cuatro simulaciones realizadas, se deduce que cuanto mayor sea el valor de $\frac{K_p}{K_d}$, el sistema presenta periodos más largos de oscilación antes de alcanzar su fase estable. En cambio, cuanto menor sea el valor los periodos de oscilación serán más cortos, pero el sistema tarda más tiempo en alcanzar los valores de referencia.

Con los coeficientes de ganancia $K_p = 0.5$ y $K_d = 1$, se ejecuta el sistema con otros dos ángulos de referencia.

- $Q_{ref} = (\cos(\frac{\pi}{2}), 0, \sin(\frac{\pi}{2}), 0)$

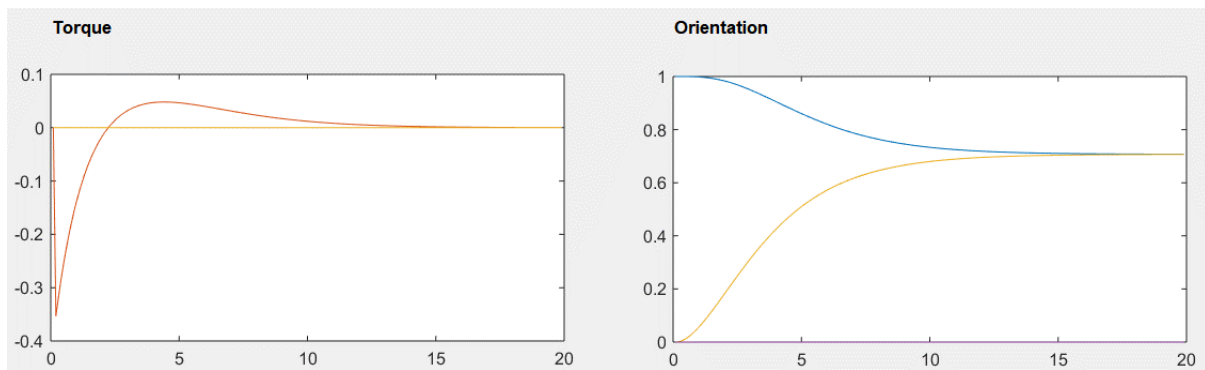


Figura 44. Resultados del control PD con $K_p=0.5$ y $K_d=1$ y segundo ángulo de prueba.

- $Q_{ref} = (\cos(\frac{\pi}{2}), 0, \sin(\frac{\pi}{2}), 0) \cdot (\cos(\frac{\pi}{8}), 0, 0, \sin(\frac{\pi}{8}))$, lo que equivale en ángulos de Euler al vector (90, 0, 45)

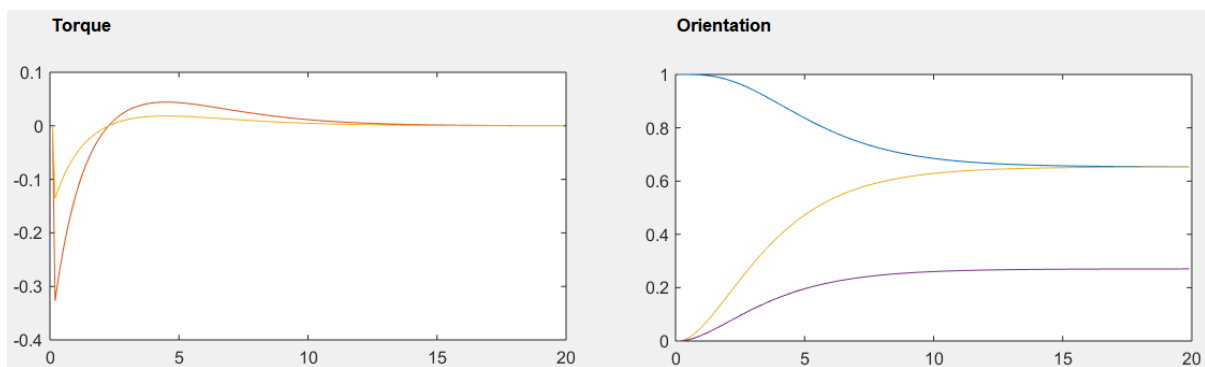


Figura 45. Resultados del control PD con $K_p=0.5$ y $K_d=1$ y tercer ángulo de prueba.

CAPÍTULO 4: Conclusiones y líneas futuras

El objetivo inicial de este trabajo fin de máster consistía en el desarrollo de diferentes algoritmos de control para un módulo ADCS utilizando el estándar SMP2 y C++. No obstante, teniendo en cuenta que no existían intérpretes libres de SMP2 se reorientó este objetivo posteriormente.

Analizando la problemática relacionada con los proyectos espaciales, se fijó como nuevo objetivo el desarrollo de un sistema que integrara las herramientas utilizadas por el equipo de desarrollo de software y el equipo de modelado matemático en las compañías dedicadas al diseño de satélites. Con la integración de los diferentes módulos implementados en el marco de esta tesis, se ha logrado este cometido.

Con este nuevo objetivo se puso más énfasis en la parte de desarrollo de software que en la parte de sistemas de control con respecto al objetivo original. Ha sido necesario desarrollar un conjunto amplio de módulos en diferentes tecnologías, con el fin de integrar las herramientas comúnmente utilizadas por los programadores (C++, SMP2) con las utilizadas por los modeladores (Matlab, Simulink).

Durante la fase de desarrollo de este proyecto, se ha trabajado en entornos diferentes. Para la implementación de los módulos desarrollados en C++ se ha utilizado un sistema operativo basado en Linux. En cambio, para la parte desarrollada en Matlab, se ha utilizado el sistema operativo Windows. Esto no es casual; en muchas de las empresas dedicadas al sector espacial, los dos equipos de trabajo suelen utilizar diferentes sistemas operativos dependiendo de la naturaleza de las tareas a implementar.

La conclusión más importante que se puede extraer de este proyecto es que se ha demostrado que es posible integrar un motor de simulación SMP2 en el entorno Matlab. Gracias a esto, se podrían cargar los complejos modelos SMP2 de los subsistemas de un satélite (con todas sus interfaces, cableado, protocolos...) en Matlab, y probar los modelos matemáticos desarrollados en este entorno de acuerdo con los valores devueltos por los respectivos modelos en cada instante de simulación.

De esta manera, un ingeniero del software puede probar cualquier módulo matemático desarrollado en Matlab/Simulink en modelos SMP2 sin necesidad de convertir el código a C++ primero, y a SMP2 después, que es lo que se suele hacer normalmente. Este puede representar un ahorro considerable de tiempo en determinadas fases de un proyecto de satélites.

La posibilidad de cerrar un lazo de control entre la salida proporcionada por un modelo dinámico definido en Simulink con el motor de simulación SMP2, tal y como se ha realizado en este trabajo, puede resultar muy útil a la hora de planificar el desarrollo de los sistemas de control en el software de a bordo de un satélite real. En lugar de implementar estos algoritmos a partir de los modelos dinámicos en Simulink y realizar las pruebas sobre el entorno de simulación, se podrían ejecutar los tests directamente sobre la salida proporcionada por Simulink y comprobar que el sistema funciona correctamente.

Otra destacada conclusión derivada del desarrollo de esta tesis es la conveniencia de usar estándares de desarrollo para proyectos relacionados con la simulación de componentes de

vehículos espaciales. El uso de estos estándares simplifica en gran manera la portabilidad de modelos entre distintas plataformas y proyectos de simulación espacial. A este respecto, el estándar SMP2 es una opción utilizada en muchos de estos proyectos a nivel europeo.

El trabajo desarrollado tiene mucha potencial para ser ampliado con otras muchas funcionalidades y aplicaciones que exploten sus capacidades. Se proponen a continuación algunas propuestas de líneas futuras.

- **Completar implementación de entorno SMP2.** El entorno de simulación SMP2 desarrollado, implementa la mayoría de las características más importantes del estándar, pero no es una implementación completa. Esto puede ocasionar que algunos módulos más complejos puedan no funcionar en el simulador.
- **Integrar entorno SMP2 con más sistemas.** Además de Matlab, existen más lenguajes y herramientas orientadas al cálculo matemático que son utilizados por modeladores de sistemas dinámicos. Una posible línea de desarrollo sería la integración del entorno en varias de estas herramientas.
- **Implementación de intérprete más funcional y visual.** Los intérpretes desarrollados en este trabajo pueden resultar incómodos de utilizar, ya que requieren la introducción de comandos por parte del usuario. Utilizando como base el mismo entorno de simulación implementado, podría desarrollarse intérpretes más interactivos y gráficos con el fin de facilitar la tarea al usuario final. Una opción sería el desarrollo de un intérprete basado en una aplicación web.
- **Implementación de framework de desarrollo de modelos.** Una propuesta ambiciosa sería el desarrollo de un entorno de desarrollo donde se pudieran definir a alto nivel las características de los diferentes subsistemas de un satélite, así como parte de los algoritmos de control. Este sistema generaría automáticamente los modelos dinámicos en Simulink y los modelos SMP2, y crearía también los scripts necesarios para ejecutar el sistema completo.
- **Módulo de análisis en Matlab a partir de las salidas devueltas por el motor de simulación.** Básicamente consistiría en un programa que, a partir de un motor de simulación en ejecución pudiera almacenar varios valores de las salidas de los modelos SMP2 cada cierto intervalo de tiempo, con el fin de aplicar algoritmos de análisis de datos sobre ellos o dibujar gráficas que definan el comportamiento del modelo.
- **Implementación de entorno de simulación SMP2 en otros lenguajes.** C++ es el lenguaje más utilizado en el ámbito del desarrollo de modelos SMP2. No obstante, pudiera ser interesante el desarrollo de motores de simulación en otros lenguajes de naturaleza diferente a C++.
- **Desarrollo de módulo ADCS genérico.** El módulo ADCS desarrollado en este trabajo, consiste únicamente en un módulo dinámico y algoritmo de control. Una ampliación útil de este sistema sería la implementación de un módulo genérico donde el usuario pudiera elegir los algoritmos a utilizar a partir de una biblioteca con los algoritmos más comunes y utilizados en el ámbito de los sistemas ADCS. El usuario también podría

configurar las condiciones del modelo dinámico del satélite a partir de un conjunto amplio de variables que acercara la simulación más a la realidad, así como especificar el ensamblado de modelos SMP2 a utilizar y la forma en la que estos interactúan con los componentes en Matlab.

- **Desarrollo de un modelo SMP2 de un computador básico utilizado en satélites sencillos.** Este desarrollo permitiría cargar el código nativo a utilizar en el computador de a bordo del satélite real en el modelo, y verificar si el sistema se comporta correctamente.

Bibliografía

- (1) European Cooperation for Space Standardization, *Simulation modelling platform - Volume 1: Principles and requirements (ECSS-E-TM-40-07 Volume 1A)*, ESA Requirements and Standards Division, 2011.
- (2) European Cooperation for Space Standardization, *Simulation modelling platform - Volume 2: Metamodel (ECSS-E-TM-40-07 Volume 2A)*, ESA Requirements and Standards Division, 2011.
- (3) European Cooperation for Space Standardization, *Simulation modelling platform - Volume 3: Component model (ECSS-E-TM-40-07 Volume 3A)*, ESA Requirements and Standards Division, 2011.
- (4) European Cooperation for Space Standardization, *Simulation modelling platform - Volume 4: C++ Mapping (ECSS-E-TM-40-07 Volume 4A)*, ESA Requirements and Standards Division, 2011.
- (5) European Cooperation for Space Standardization, *Simulation modelling platform - Volume 5: SMP usage (ECSS-E-TM-40-07 Volume 5A)*, ESA Requirements and Standards Division, 2011.
- (6) European Space Agency, *SMP 2.0 Handbook (EGOS-SIM-GEN-TN-0099)*, 2005.
- (7) European Space Agency, *SMP 2.0 Metamodel (EGOS-SIM-GEN-TN-0100)*, 2005.
- (8) European Space Agency, *SMP 2.0 Component Model (EGOS-SIM-GEN-TN-0101)*, 2005.
- (9) D. Vallado, *Fundamentals of Astrodynamics and Applications*, Microcosm Press Year, 2013.
- (10) K. Groÿekatthöfer, Z. Yoon, *Introduction into quaternions for spacecraft attitude representation*, Technical University of Berlin, 2012.
- (11) I. Ofodile, *Design and Comparison of Attitude Control Modes for ESTCube-2*, University of Tartu, 2017.
- (12) E. Fresk, G. Nikolakopoulos, *Full Quaternion Based Attitude Control for a Quadrotor*, European Control Conference, 2013.
- (13) J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, Springer Aerospace Technology, 2011.

- (14) National Aeronautics and Space Administration *GOES I-M Databook*, Space Systems/Loral, 1996.
- (15) Chong-Hun Zee, *Theory of Geostationary Satellites*, Springer Netherlands, 1989.
- (16) O. Montenbruck, *Satellite Orbits: Models, Methods and Applications*, Springer-Verlag Berlin Heidelberg, 2000.
- (17) M. Mazkarian, *Interferencias en Satélites Geoestacionarios*, Universidad Nacional Autónoma de México, 2011.
- (18) T.Iia, *Satellite Communications: System and its Design Technology*, IOS Press, 2000.
- (19) Olivier L. de Weck, *Attitude Determination and Control (ADCS)*, Massachusetts Institute of Technology, 2001.
- (20) Agencia Espacial Mexicana, *Ingeniería de Sistemas Espaciales aplicado a una Misión CanSat*.
- (21) M.J. Sidi, *Spacecraft Dynamics and Control: A Practical Engineering Approach*, Cambridge University Press, 2000.
- (22) J. Wertz, *Spacecraft Attitude Determination and Control*, Springer Netherlands, 1978.
- (23) J.M. Vitorino, *Design and development of the ECOSats Attitude Determination and Control System (ADCS) onboard software*, Técnico Lisboa, 2013.
- (24) H. Trentelman, *Control Theory for Lineal Systems*, University of Groningen, 2002.

Anexo 1. Diagramas de clases del entorno de simulación implementado

1. GenericComponent

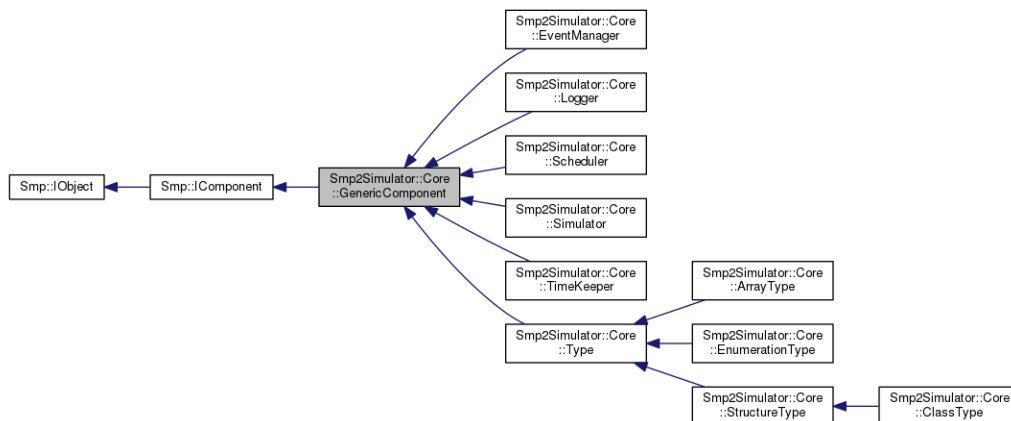


Figura 46. Diagrama de herencia de la clase GenericComponent.

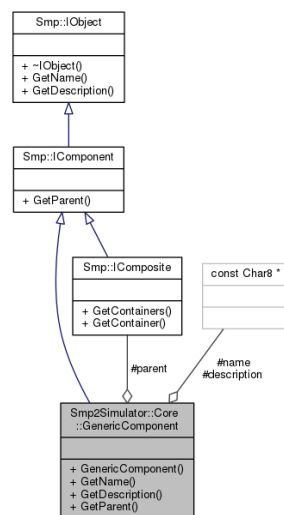


Figura 47. Diagrama de clases de GenericComponent.

2. Simulator

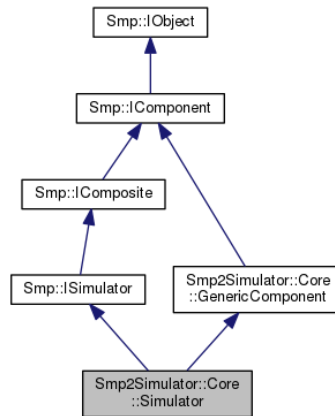


Figura 48. Diagrama de herencia de la clase Simulator.

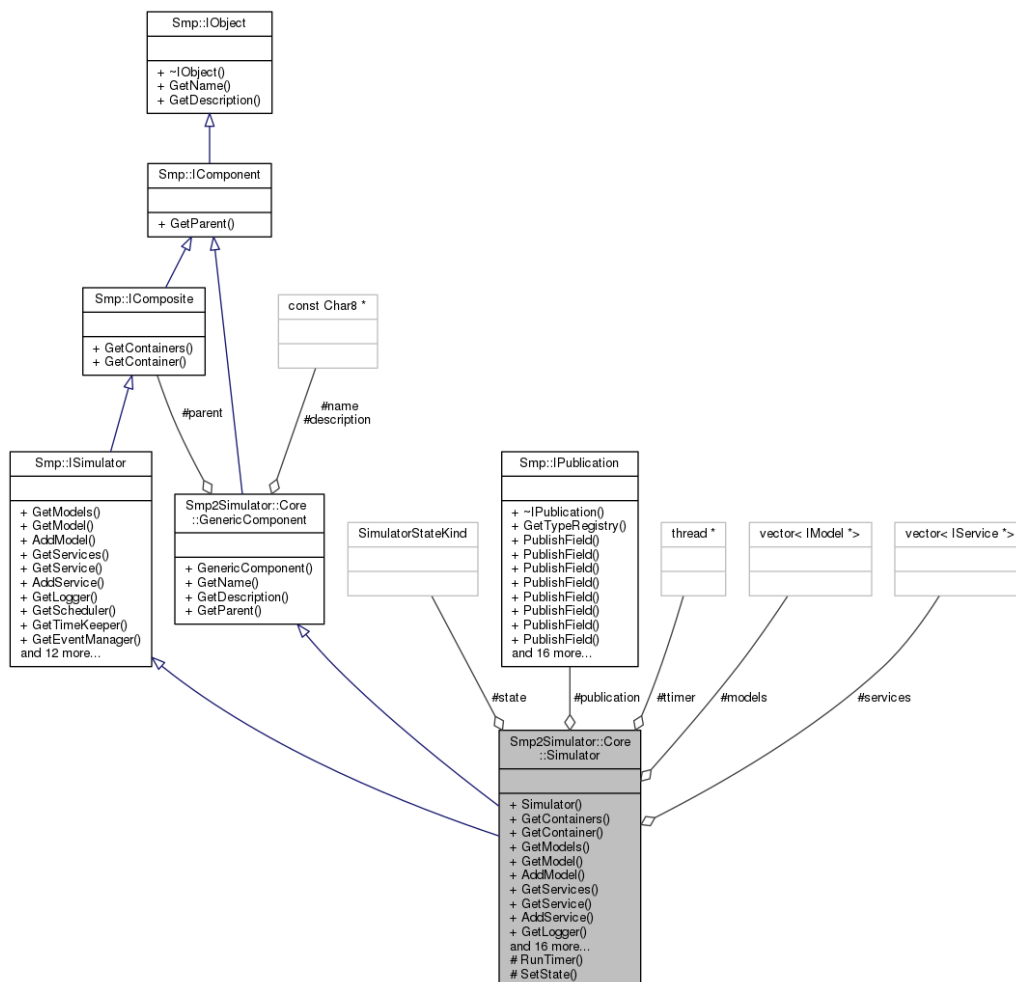


Figura 49. Diagrama de clases de Simulator.

3. Publication

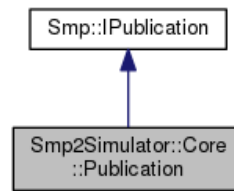


Figura 50. Diagrama de herencia de la clase Publication.

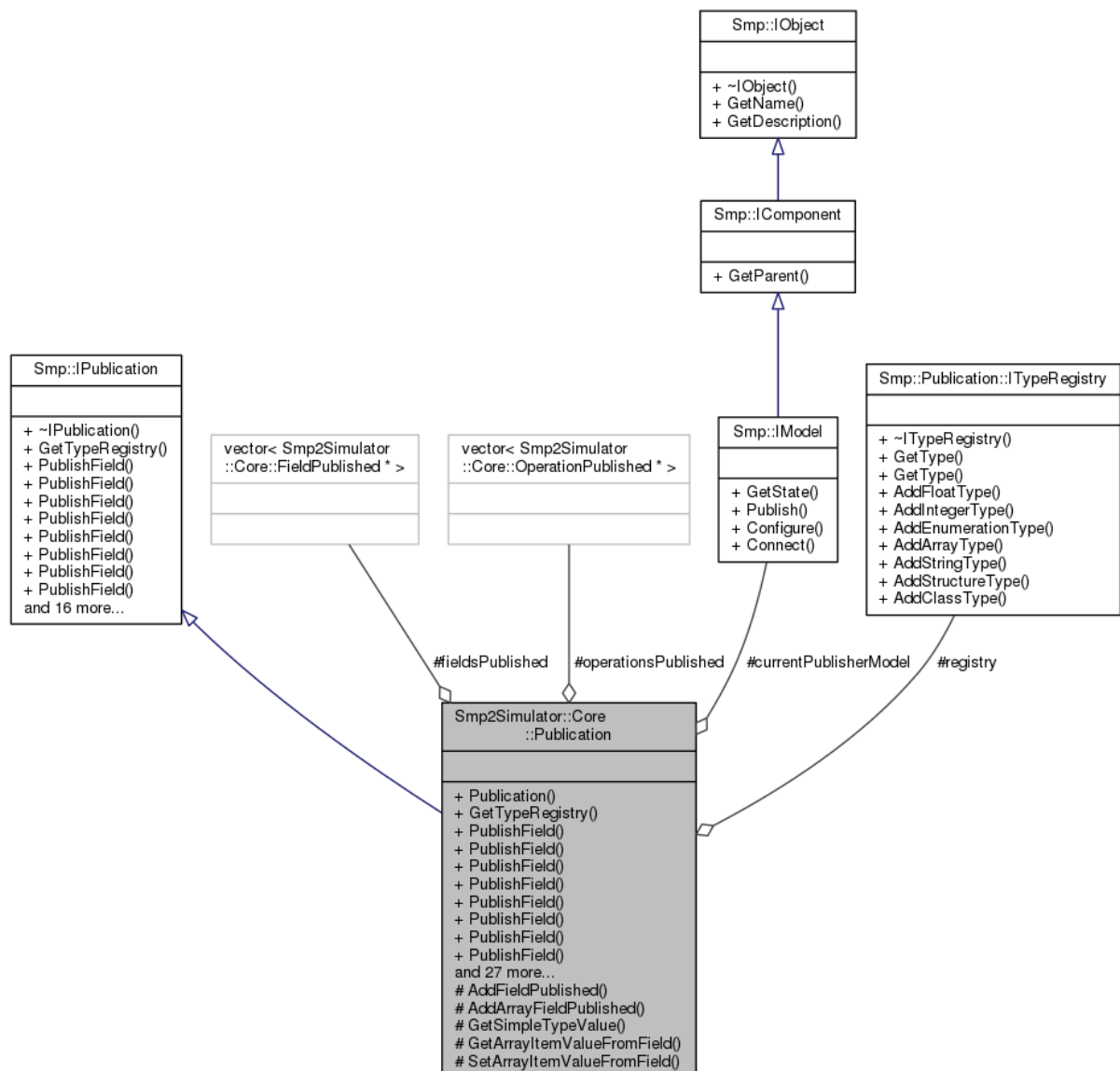


Figura 51. Diagrama de clases de Publication.

4. TypeRegistry

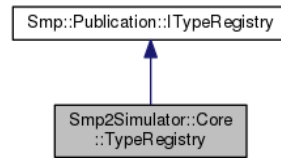


Figura 52. Diagrama de herencia de la clase TypeRegistry.

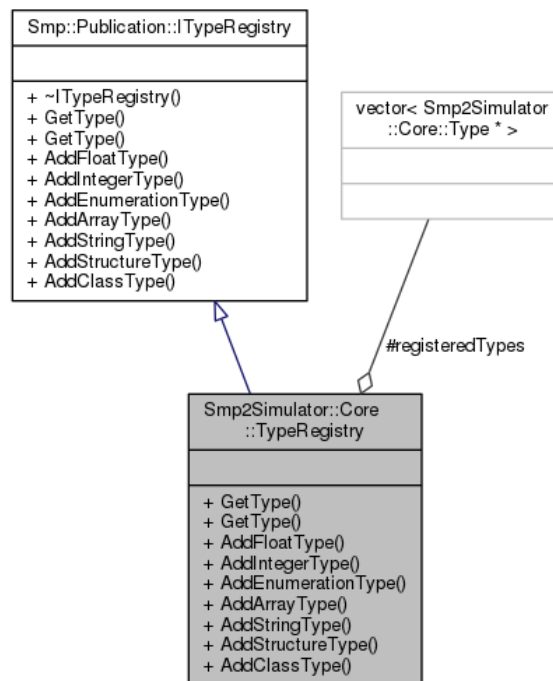


Figura 53. Diagrama de clases de TypeRegistry.

5. Logger

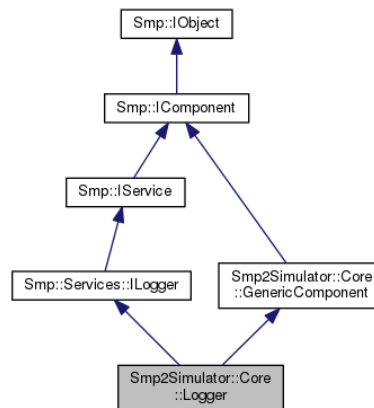


Figura 54. Diagrama de herencia de la clase Logger.

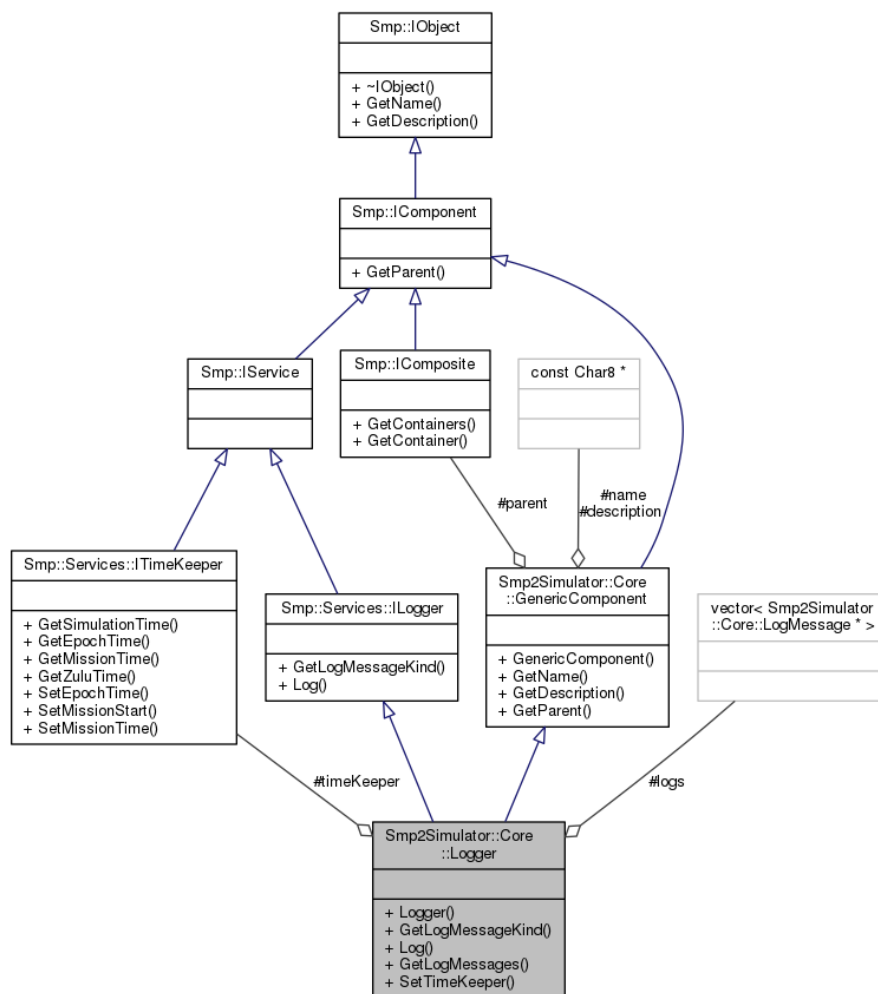


Figura 55. Diagrama de clases de Logger.

6. Scheduler

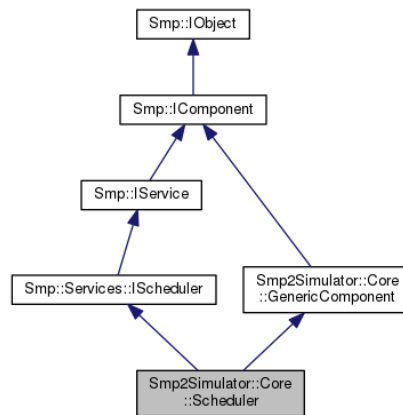


Figura 56. Diagrama de herencia de la clase Scheduler.

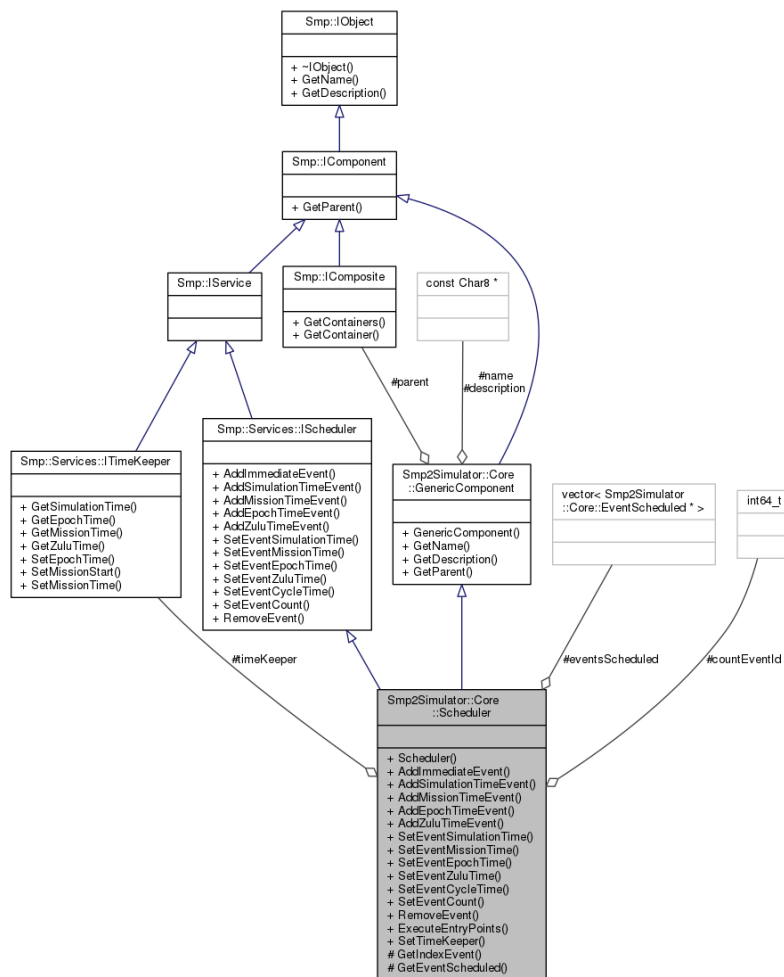


Figura 57. Diagrama de clases de Scheduler.

7. TimeKeeper

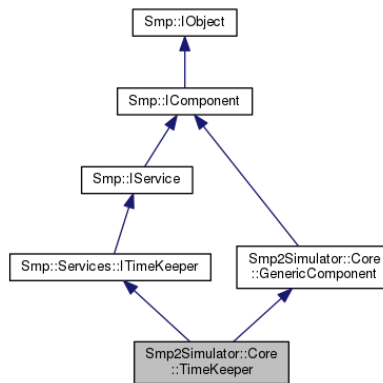


Figura 58. Diagrama de herencia de la clase TimeKeeper.

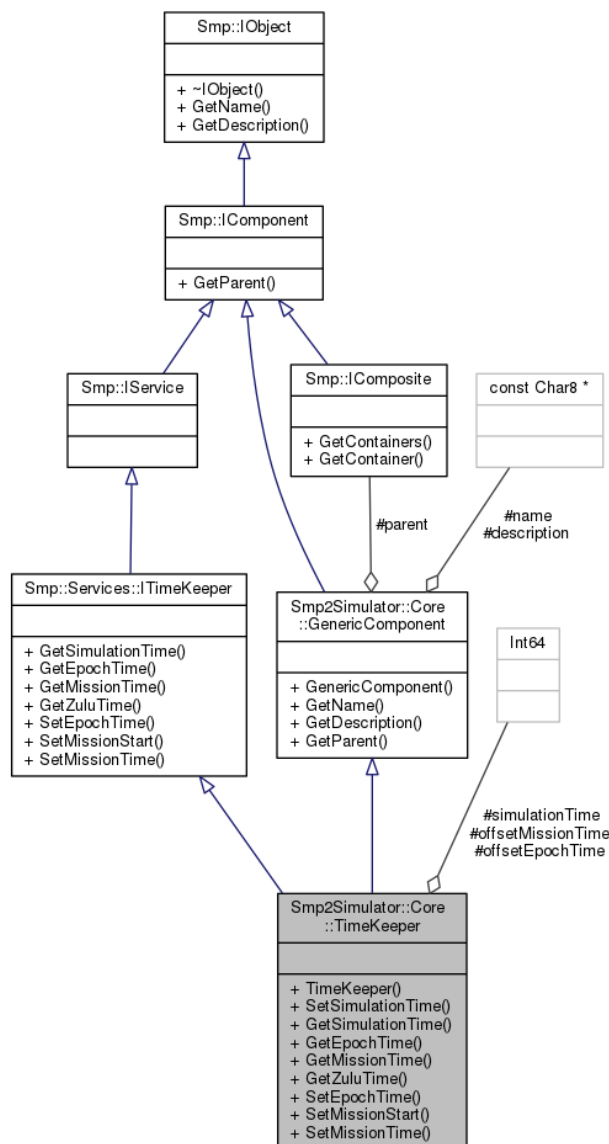


Figura 59. Diagrama de clases de TimeKeeper.

8. EventManager

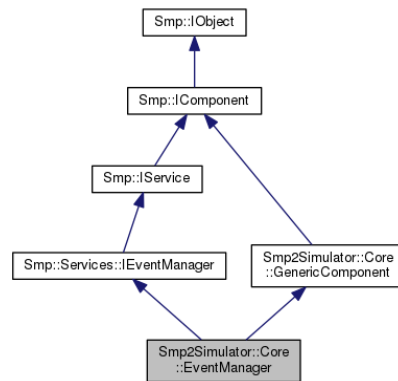


Figura 60. Diagrama de herencia de EventManager.

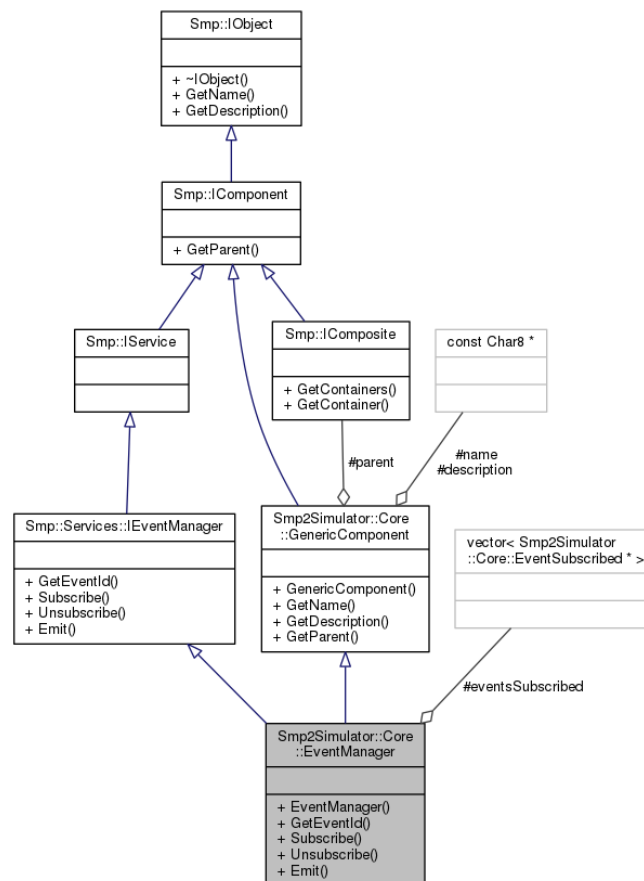


Figura 61. Diagrama de clases de EventManager.

Anexo 2. Principales métodos implementados de interfaces SMP2

1. IComponent

- **GetName()**. Devuelve el nombre del componente.
- **GetDescription()**. Devuelve la descripción del componente.
- **GetParent()**. Devuelve el elemento inmediatamente superior al que pertenece el componente.

2. Interfaz ISimulator

- **Publish()**. Llama a los métodos *Publish* de todos los modelos cargados en el entorno de simulación. Este método sólo puede ser llamado sólo si el simulador se encuentra en el estado *Building*.
- **Configure()**. Llama a los métodos *Configure* de todos los modelos cargados en el entorno de simulación. Este método sólo puede ser llamado sólo si el simulador se encuentra en el estado *Building*.
- **Connect()**. Llama a los métodos *Connect* de todos los modelos cargados en el entorno de simulación. Este método sólo puede ser llamado sólo si el simulador se encuentra en el estado *Building*. Después de ser ejecutado, el simulador pasa al estado *Connecting*.
- **Initialise()**. Inicializa todos los puntos de entrada del entorno de simulación. Este método sólo puede ser llamado manualmente si el simulador se encuentra en el estado *Standby*. Después de ser ejecutado, el simulador pasa al estado *Initialising*.
- **Run()**. Ejecuta el motor de ejecución, haciendo que el tiempo de simulación progrese. Se ejecutarán los puntos de entrada configurados de acuerdo al valor del tiempo de simulación. Este método sólo puede ser llamado si el simulador se encuentra en el estado *Standby*. Después de ser ejecutado, el simulador pasa al estado *Executing*.
- **Hold()**. Para el motor de ejecución, pausando el tiempo de simulación. Este método sólo puede ser llamado si el simulador se encuentra en el estado *Executing*. Después de ser ejecutado, el simulador pasa al estado *Standby*.
- **GetState()**. Devuelve el estado actual del simulador.
- **GetModels()**. Devuelve todos los modelos creados en el contenedor raíz del entorno de simulación. Debido a que estos modelos implementan la interfaz *IComposite*, pueden contener otros modelos dentro de ellos.

- ***GetModel(String8 name)***. Devuelve el modelo del contenedor raíz con el nombre pasado como parámetro. Si dicho modelo no existe, se devuelve un puntero nulo.
- ***AddModel(IModel* model)***. Añade un modelo al contenedor raíz del simulador. Es llamado típicamente en el método que inicializa el ensamblado del entorno de simulación.
- ***GetServices()***. Devuelve la colección de todos los servicios registrados en el simulador. Entre ellos estarán los servicios predefinidos implementados en nuestro entorno de simulación (*Logger*, *EventManager*, *Scheduler*, *TimeKeeper*).
- ***GetLogger()***. Devuelve la interfaz del servicio *Logger*.
- ***GetEventManager()***. Devuelve la interfaz del servicio *EventManager*.
- ***GetScheduler()***. Devuelve la interfaz del servicio *Scheduler*.
- ***GetTimeKeeper()***. Devuelve la interfaz del servicio *TimeKeeper*.

3. Interfaz IPublication

- ***GetTypeRegistry()***. Devuelve la instancia del registrador de tipos usado por el entorno de simulación.
- ***PublishField(String8 name, String8 description, Char8* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Char8*. Se llama típicamente desde el método *Publish()* de un modelo. Se puede definir además si el campo será visible en el entorno de simulación, si debe ser tenido en cuenta a la hora de almacenar el estado de la simulación, y si es de entrada y/o salida.
- ***PublishField(String8 name, String8 description, Bool* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Bool*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Int8* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Int8*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Int16* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Int16*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Int32* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Int32*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Int64* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Int64*. Se llama típicamente desde el método *Publish()* de un modelo.

- ***PublishField(String8 name, String8 description, UInt8* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *UInt8*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, UInt16* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *UInt16*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, UInt32* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *UInt32*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, UInt64* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *UInt64*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Float32* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Float32*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, Float64* address, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de tipo *Float64*. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishField(String8 name, String8 description, void* address, Uuid typeUuid, Bool view, Bool state, Bool input, Bool output)***. Publica un campo de cualquier tipo registrado previamente en el registrador de tipos. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishArray(String8 name, String8 description, Int64 count, void* address, SimpleTypeKind type, Bool view, Bool state, Bool input, Bool output)***. Publica un campo array de tamaño fijo que almacena cualquiera de los tipos básicos SMP2. Se llama típicamente desde el método *Publish()* de un modelo.
- ***PublishOperation(String8 name, String8 description, Smp::Uuid returnTypeUuid)***. Publica un método que retorna un tipo básico u otro registrado con el registrador de tipos. Devuelve un objeto *IPublishOperation* que permitirá añadir parámetros a la operación.
- ***GetFieldValue(String8 fullName)***. Devuelve el valor de un campo publicado a partir de su nombre completo. Por ejemplo, si se desea obtener el valor del campo *torque* del modelo *ReactionWheel*, el nombre del campo sería *ReactionWheel.torque*.
- ***SetFieldValue(String8 fullName, AnySimple value)***. Establece el valor de un campo publicado a partir de su nombre completo. Hay que tener en cuenta que sólo se podrá modificar los campos que hayan sido publicados con el flag *output* a *true*.
- ***CreateRequest(String8 operationName)***. Devuelve un objeto *IRequest* que contendrá la información necesaria para ejecutar un método previamente publicado.

- ***DeleteRequest(IRequest *request)***. Destruye un objeto *IRequest* construido con la función anterior.

4. Interfaz ITypeRegistry

- ***GetType(SimpleTypeKind type)***. Devuelve un objeto *IType* que define un tipo básico SMP2.
- ***GetType(Uuid typeUuid)***. Devuelve un objeto *IType* que define un tipo previamente registrado por el usuario con un uuid determinado. Si el tipo no ha sido registrado, se devuelve un puntero nulo.
- ***AddArrayType(String8 name, String8 description, Uuid typeUuid, Uuid itemTypeUuid, Int64 itemSize, Int64 arrayCount)***. Registra un tipo array de tamaño fijo que almacene un conjunto de elementos de un tipo simple SMP2.

5. Interfaz Logger

- ***Log(const IObject* sender, String8 message, LogMessageKind kind)***. Recibe un mensaje de un tipo determinado proveniente de uno de los componentes del sistema. En nuestra implementación, este mensaje y su información relacionada se imprime en pantalla (para el intérprete en C++), y además, se almacena en memoria (para ser recuperado por el intérprete en Matlab).
- ***GetLogMessageKind(Smp::String8 messageKindName)***. Devuelve el tipo de un mensaje a partir de su nombre.
- ***GetLogMessages()***. Devuelve la información de todos los mensajes recibidos.

6. Interfaz IScheduler

- ***ExecuteEntryPoints()***. Ejecuta todos los puntos de entrada pendientes a partir del tiempo de simulación actual.
- ***AddImmediateEvent(const Smp::IEntryPoint* entryPoint)***. Añade un evento que será ejecutado de forma inmediata.
- ***AddSimulationTimeEvent(const Smp::IEntryPoint* entryPoint, Smp::Duration simulationTime, Smp::Duration cycleTime, Smp::Int64 count)***. Añade un evento que será ejecutado de manera periódica un determinado número de veces de acuerdo al tiempo de simulación.

- ***AddSimulationTimeEvent(const Smp::IEntryPoint* entryPoint, Smp::Duration simulationTime, Smp::Duration cycleTime, Smp::Int64 count)***. Añade un evento que será ejecutado de manera periódica un determinado número de veces de acuerdo al tiempo de simulación.
- ***AddMissionTimeEvent(const Smp::IEntryPoint* entryPoint, Smp::Duration missionTime, Smp::Duration cycleTime, Smp::Int64 count)***. Añade un evento que será ejecutado de manera periódica un determinado número de veces de acuerdo al tiempo de misión.
- ***AddEpochTimeEvent(const Smp::IEntryPoint* entryPoint, Smp::DateTime epochTime, Smp::Duration cycleTime, Smp::Int64 count)***. Añade un evento que será ejecutado de manera periódica un determinado número de veces de acuerdo al tiempo de época.
- ***RemoveEvent(Smp::Services::EventId event)***. Elimina un evento almacenado previamente en el planificador.

7. Interfaz ITimeKeeper

- ***GetSimulationTime()***. Devuelve el valor del tiempo actual de simulación.
- ***GetEpochTime()***. Devuelve el valor del tiempo actual de época.
- ***GetMissionTime()***. Devuelve el valor del tiempo actual de simulación.
- ***SetEpochTime(Duration epochTime)***. Establece el offset entre el tiempo de época y el tiempo de simulación.
- ***SetMissionTime(Duration missionTime)***. Establece el offset entre el tiempo de misión y el tiempo de época.

8. Interfaz IEventManager

- ***GetEventId(String8 eventName)***. Devuelve el identificador de un evento a partir de su nombre.
- ***Subscribe(EventId event, const IEntryPoint* entryPoint)***. Registra un punto de entrada para que sea ejecutado cuando se emita un evento determinado.
- ***Unsubscribe(EventId event, const IEntryPoint* entryPoint)***. Elimina el registro entre un punto de entrada y un evento.
- ***Emit(EventId evento)***. Emite un evento determinado. Los componentes del entorno de simulación serán notificados de este nuevo evento, y los puntos de entrada registrados con el evento serán ejecutados.

Anexo 3. Casos de uso y diagramas de secuencia de intérprete SMP2

1. Cargar ensamblado

CU-001	
Descripción	El sistema deberá permitir al usuario cargar un ensamblado con un conjunto de modelos a partir de una librería dinámica.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token LOAD y la ruta de la librería dinámica con el ensamblado. 2) El sistema cargará la librería dinámica. 3) El sistema llamará al método <i>Initialise</i> de la librería dinámica, pasándole como parámetro una instancia del tipo <i>Simulator</i> y otra del tipo <i>TypeRegistry</i> previamente creadas. 4) El sistema ejecutará las acciones de publicación (<i>Publish</i>) de cada uno de los modelos del ensamblado. 5) El sistema ejecutará las acciones de configuración (<i>Configure</i>) de cada uno de los modelos del ensamblado. 6) El sistema ejecutará las acciones de conexión (<i>Connect</i>) de cada uno de los modelos del ensamblado. 7) El sistema notificará al usuario que la carga del ensamblado se ha realizado correctamente.
Extensiones	2a) Si el sistema no encuentra la librería dinámica a partir de la ruta proporcionada por el usuario, se devolverá un mensaje de error. 2b) Si la librería proporcionada por el usuario no se corresponde con un ensamblado de modelos válido, se devolverá un mensaje de error. 3a) Si ocurre un error durante la ejecución del método <i>Initialise</i> se devolverá un mensaje de error al usuario, y se descargará la librería dinámica de memoria. 4a) Si ocurre un error durante la ejecución de los métodos de publicación de los modelos, se devolverá un mensaje de error al usuario, y se descargará la librería dinámica de memoria. 5a) Si ocurre un error durante la ejecución de los métodos de configuración de los modelos, se devolverá un mensaje de error al usuario, y se descargará la librería dinámica de memoria. 6a)

Si ocurre un error durante la ejecución de los métodos de conexión de los modelos, se devolverá un mensaje de error al usuario, y se descargará la librería dinámica de memoria.

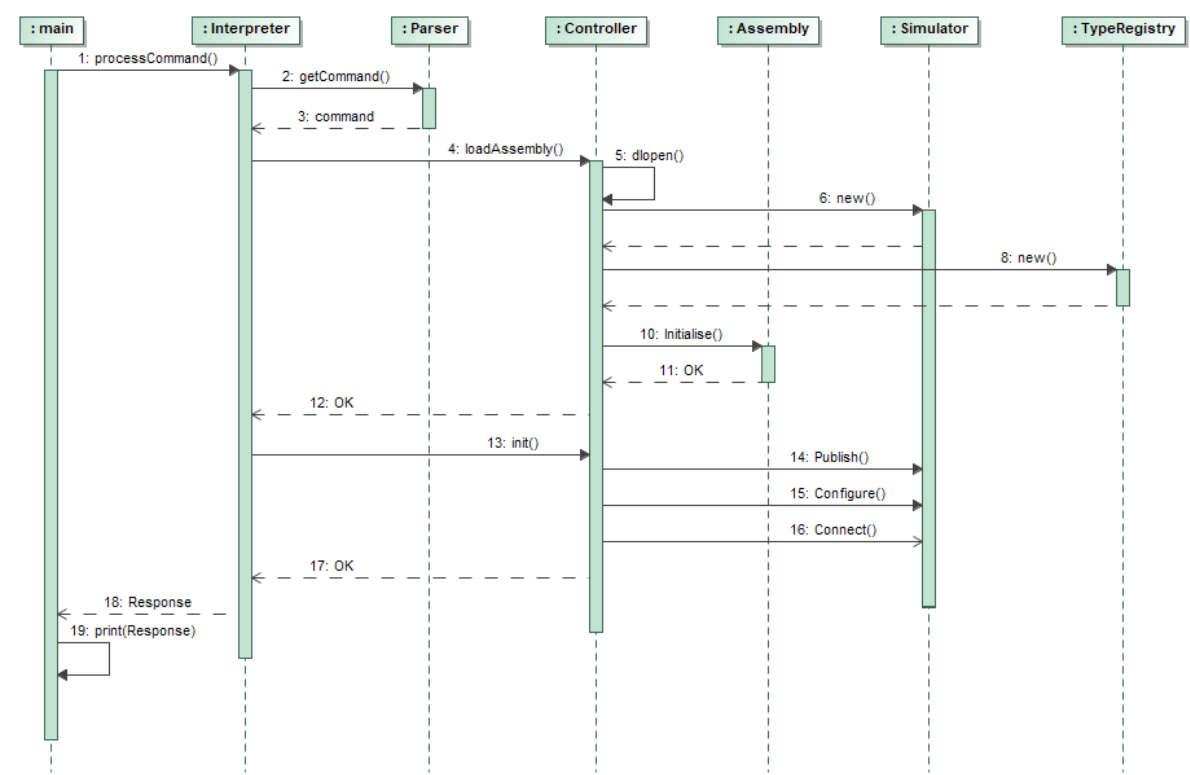


Figura 62. Diagrama de secuencia para CU-001.

2. Ejecutar simulación

CU-002	
Descripción	El sistema deberá permitir al usuario ejecutar la simulación.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token RUN . 2) El sistema ejecutará el simulador, mostrando los mensajes enviados por cada uno de los componentes en cada instante de tiempo.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el simulador ya se encuentra en ejecución, se mostrará un mensaje notificándolo al usuario.

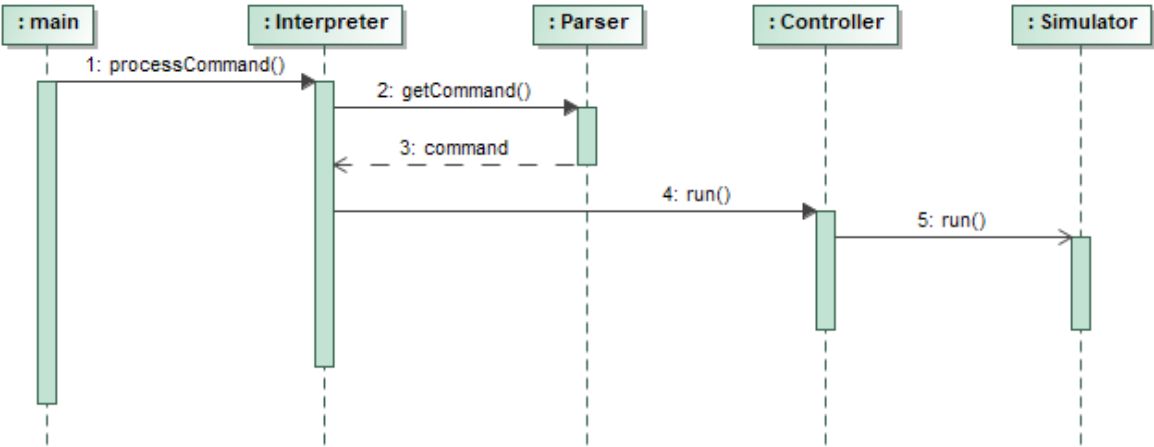


Figura 63. Diagrama de secuencia para CU-002.

3. Parar simulación

CU-003	
Descripción	El sistema deberá permitir al usuario parar la simulación.
Secuencia normal	1) El usuario ejecuta el simulador (CU-002). 2) El usuario deberá pulsar la tecla ENTER mientras el simulador está en ejecución. 3) El sistema parará el simulador, congelando el tiempo de simulación al valor actual.

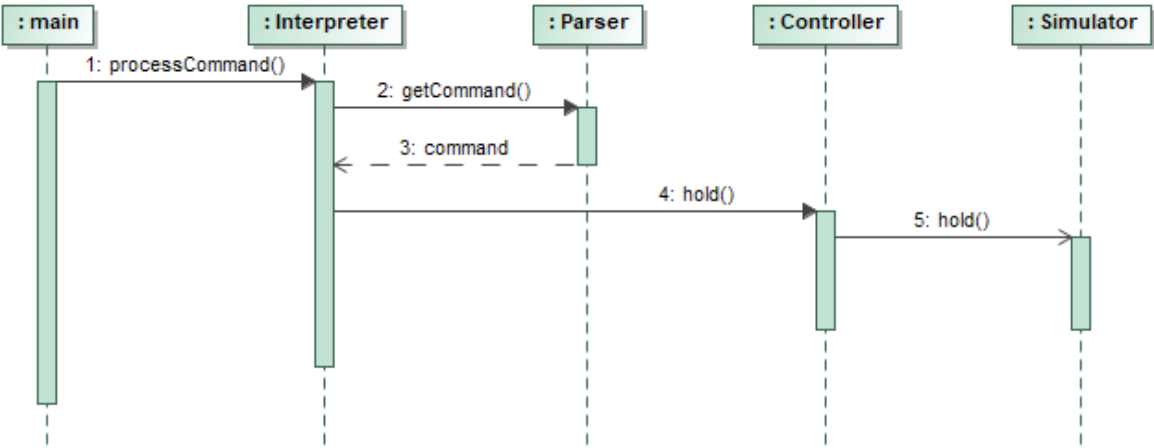


Figura 64. Diagrama de secuencia para CU-003.

4. Consultar modelos cargados

CU-004	
Descripción	El sistema deberá permitir al usuario consultar el nombre de todos los modelos cargados.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token GET MODELS. 2) El sistema mostrará al usuario la lista de los nombres de los modelos cargados.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario.

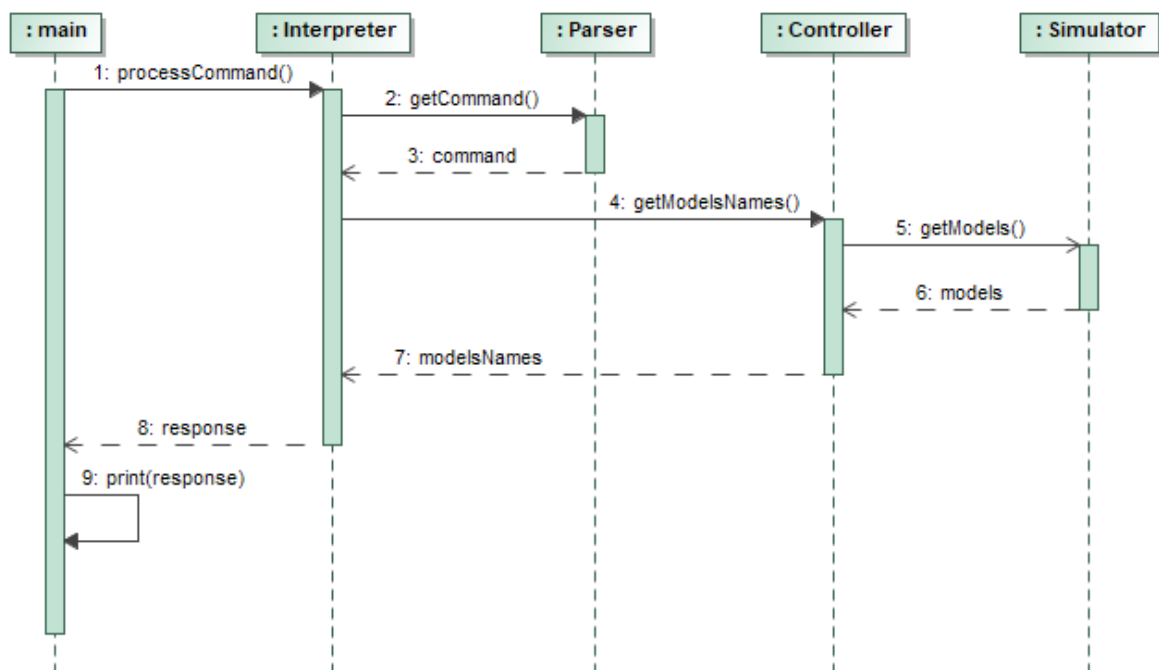


Figura 65. Diagrama de secuencia para CU-004.

5. Consultar campos publicados de un modelo

CU-005	
Descripción	El sistema deberá permitir al usuario consultar el nombre de los campos publicados de un modelo.

Secuencia normal	1) El usuario escribirá en la línea de comandos el token GET FIELDS y el nombre del modelo a consultar. 2) El sistema mostrará al usuario la lista de los nombres de los campos publicados por el modelo especificado.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el modelo especificado no existe, se mostrará un mensaje de error al usuario. 2c) Si el modelo especificado no tiene campos publicados, se mostrará un mensaje de notificación al usuario.

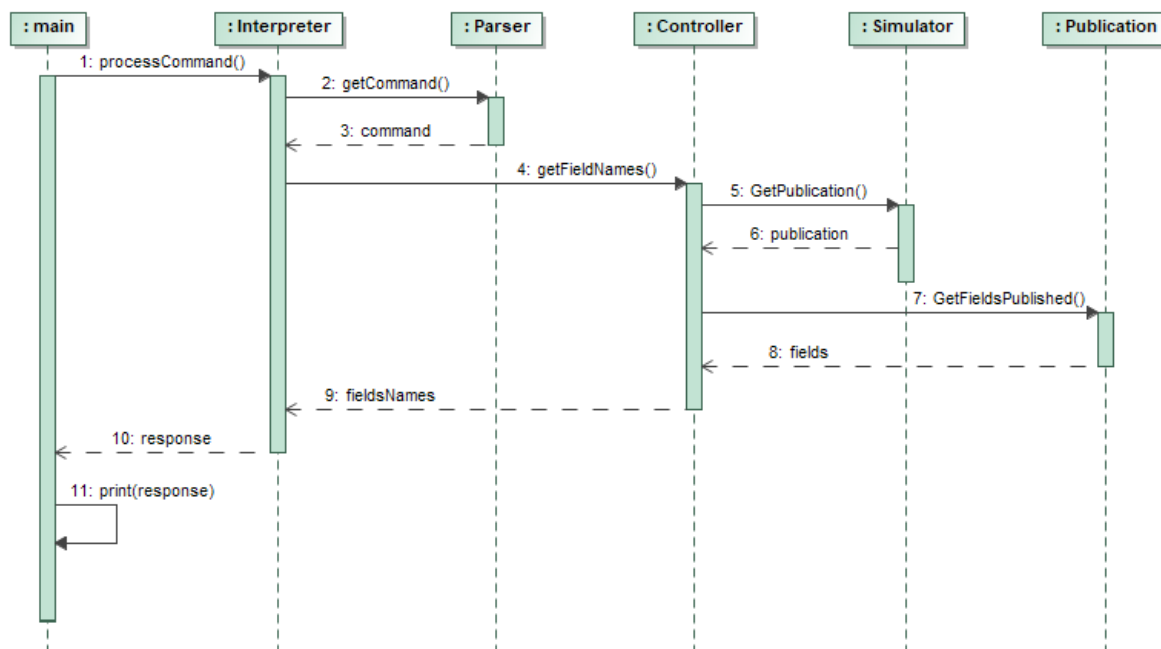


Figura 66. Diagrama de secuencia para CU-005.

6. Consultar operaciones publicadas de un modelo

CU-006	
Descripción	El sistema deberá permitir al usuario consultar el nombre de las operaciones publicadas de un modelo.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token GET OPERATIONS y el nombre del modelo a consultar. 2) El sistema mostrará al usuario la lista de los nombres de las operaciones publicadas por el modelo especificado.

Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el modelo especificado no existe, se mostrará un mensaje de error al usuario. 2c) Si el modelo especificado no tiene operaciones publicadas, se mostrará un mensaje de notificación al usuario.
--------------------	---

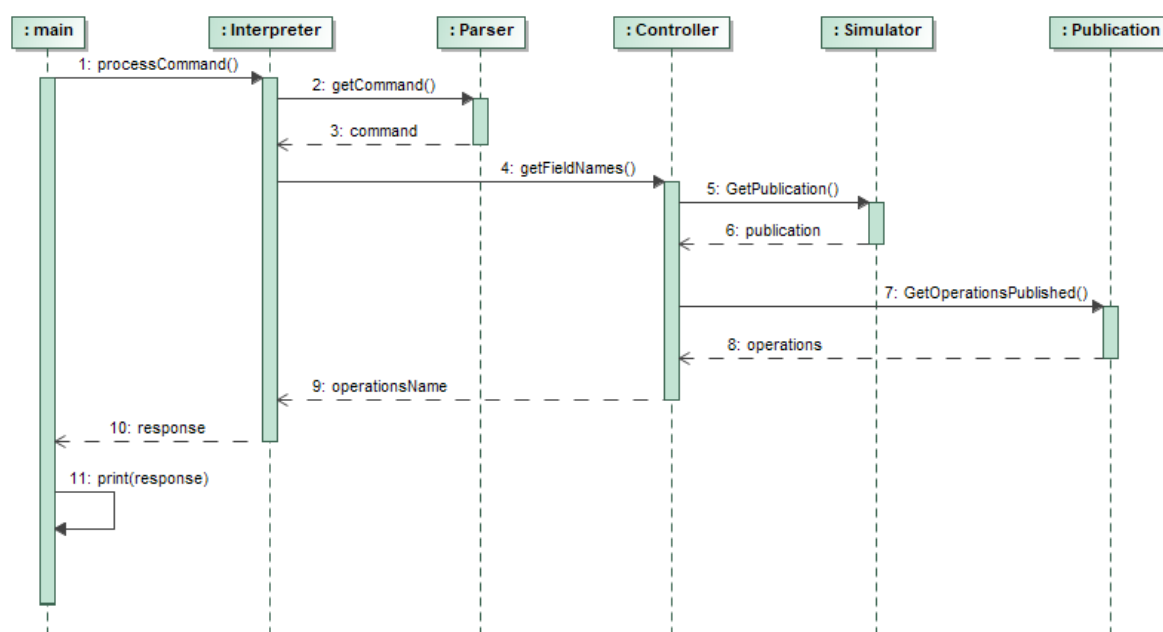


Figura 67. Diagrama de secuencia para CU-006.

7. Consultar tiempos de simulación

CU-007	
Descripción	El sistema deberá permitir al usuario consultar el valor actual de los diferentes tiempos de simulación.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token GET TIMES. 2) El sistema mostrará al usuario el valor actual del tiempo de simulación, del tiempo de época y del tiempo de misión.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario.

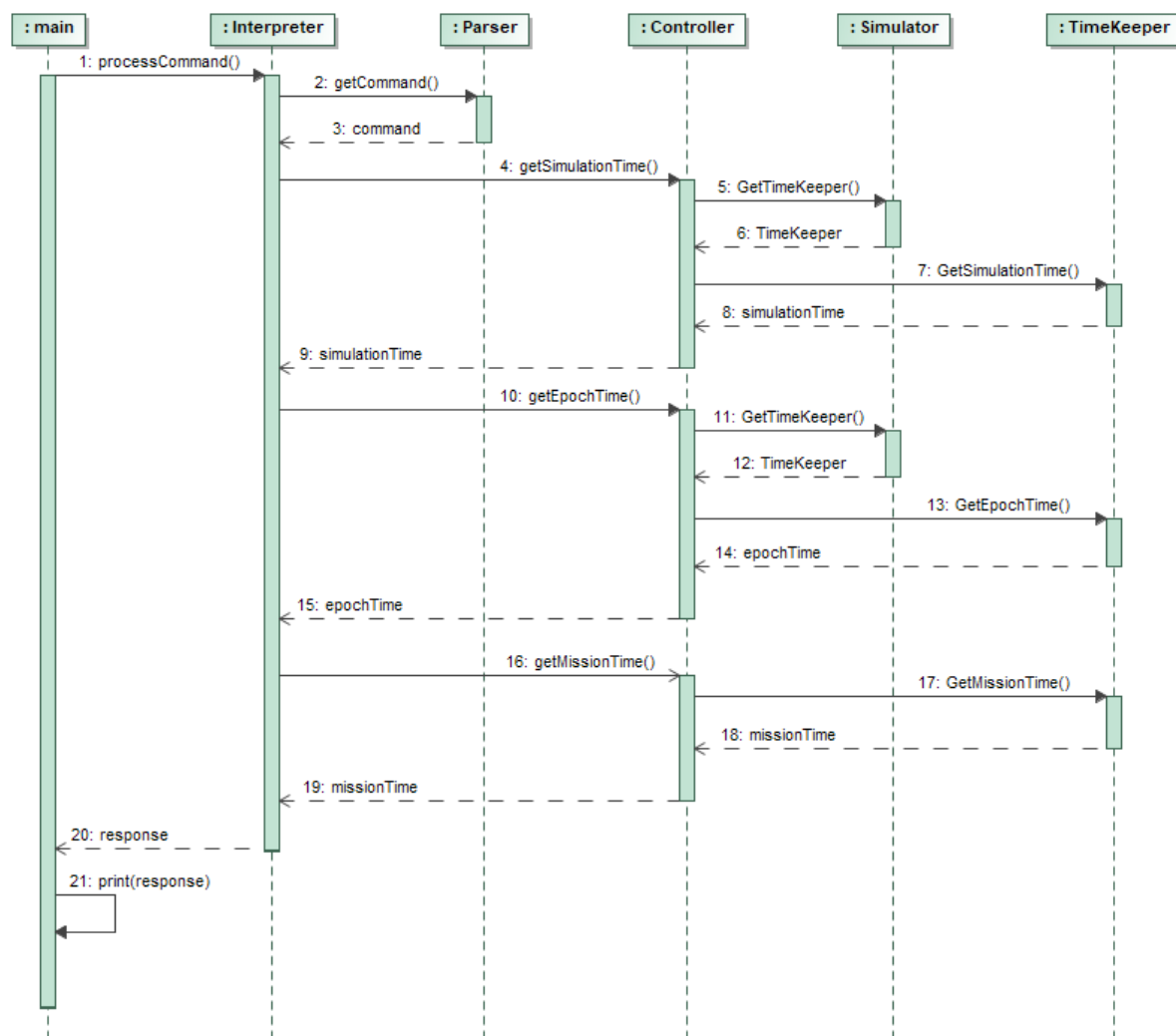


Figura 68. Diagrama de secuencia para CU-007.

8. Consultar valor de un campo publicado

CU-008	
Descripción	El sistema deberá permitir al usuario consultar el valor de un campo publicado por un modelo.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token GET VALUE y la ruta completa del campo a consultar. 2) El sistema mostrará al usuario el valor actual del campo especificado.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b)

Si el campo especificado no existe, se mostrará un mensaje de error al usuario.

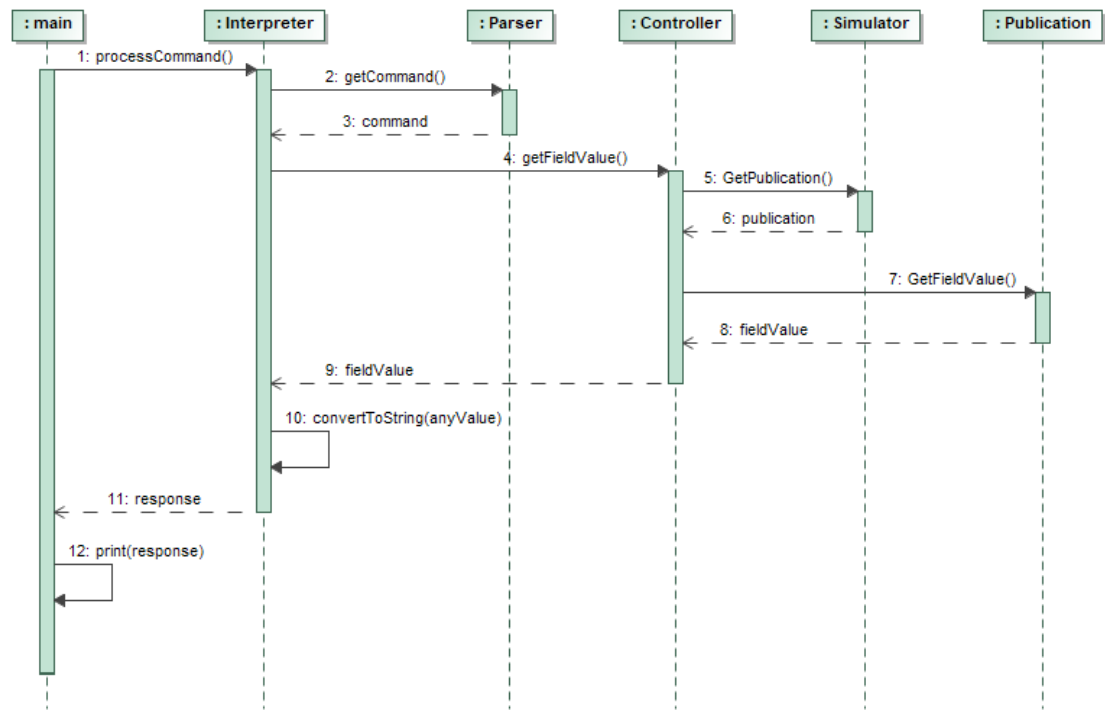


Figura 69. Diagrama de secuencia para CU-008.

9. Consultar valor de un array publicado

CU-009	
Descripción	El sistema deberá permitir al usuario consultar el valor de un campo publicado por un modelo.
Secuencia normal	3) El usuario escribirá en la línea de comandos el token GET ARRAY VALUE, la ruta completa del campo y el número de elemento del array a consultar. 4) El sistema mostrará al usuario el valor actual de la posición especificada del array.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el campo especificado no existe, se mostrará un mensaje de error al usuario. 2c) Si la posición especificada no es válida, se mostrará un mensaje de error al usuario.

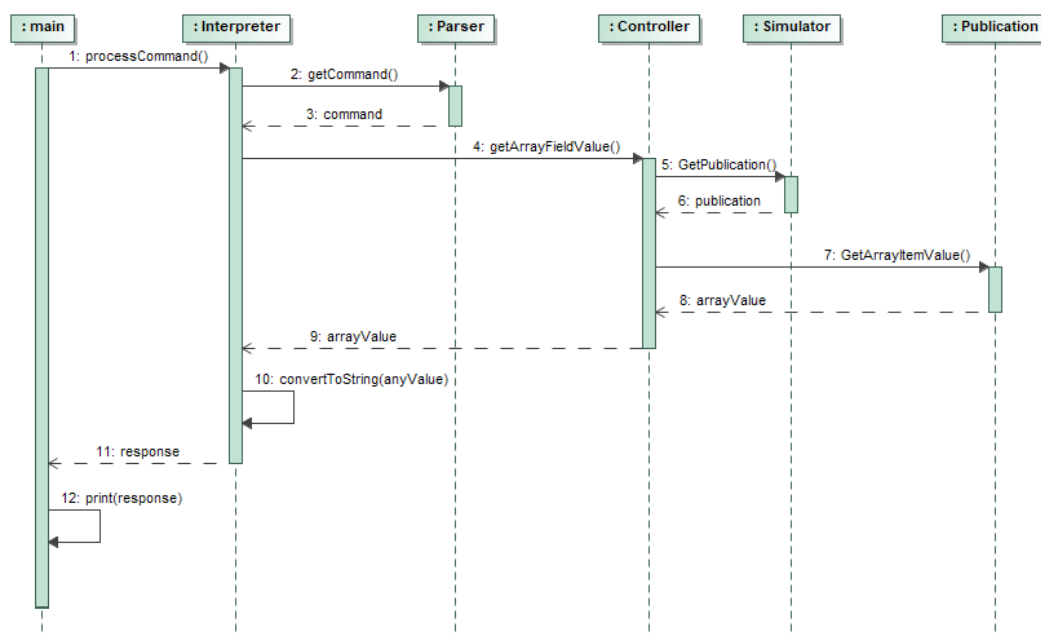


Figura 70. Diagrama de secuencia para CU-009.

10. Modificar valor de un campo publicado

CU-010	
Descripción	El sistema deberá permitir al usuario modificar el valor de un campo de tipo básico publicado por un modelo.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token SET VALUE, la ruta completa del campo y el nuevo valor del mismo. 2) El sistema modificará el valor del campo especificado. 3) El sistema notificará al usuario de que el campo se ha modificado correctamente.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el campo especificado no existe, se mostrará un mensaje de error al usuario. 2c) Si el campo especificado no es de entrada, se mostrará un mensaje de error al usuario.

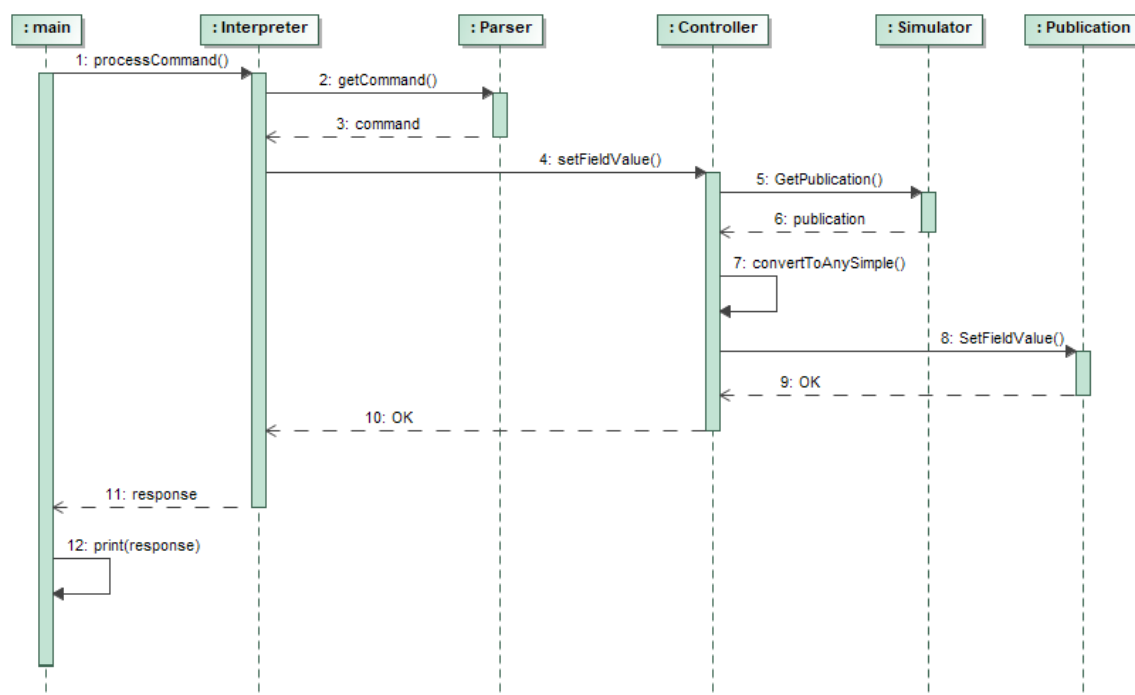


Figura 71. Diagrama de secuencia para CU-010.

11. Modificar valor de un array publicado

CU-011	
Descripción	El sistema deberá permitir al usuario modificar el valor de un campo de tipo array publicado por un modelo.
Secuencia normal	4) El usuario escribirá en la línea de comandos el token SET ARRAY, la ruta completa del campo, la posición del array a modificar y el nuevo valor del mismo. 5) El sistema modificará el valor del campo especificado. 6) El sistema notificará al usuario de que el campo se ha modificado correctamente.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el campo especificado no existe, se mostrará un mensaje de error al usuario. 2c) Si el campo especificado no es de entrada, se mostrará un mensaje de error al usuario.

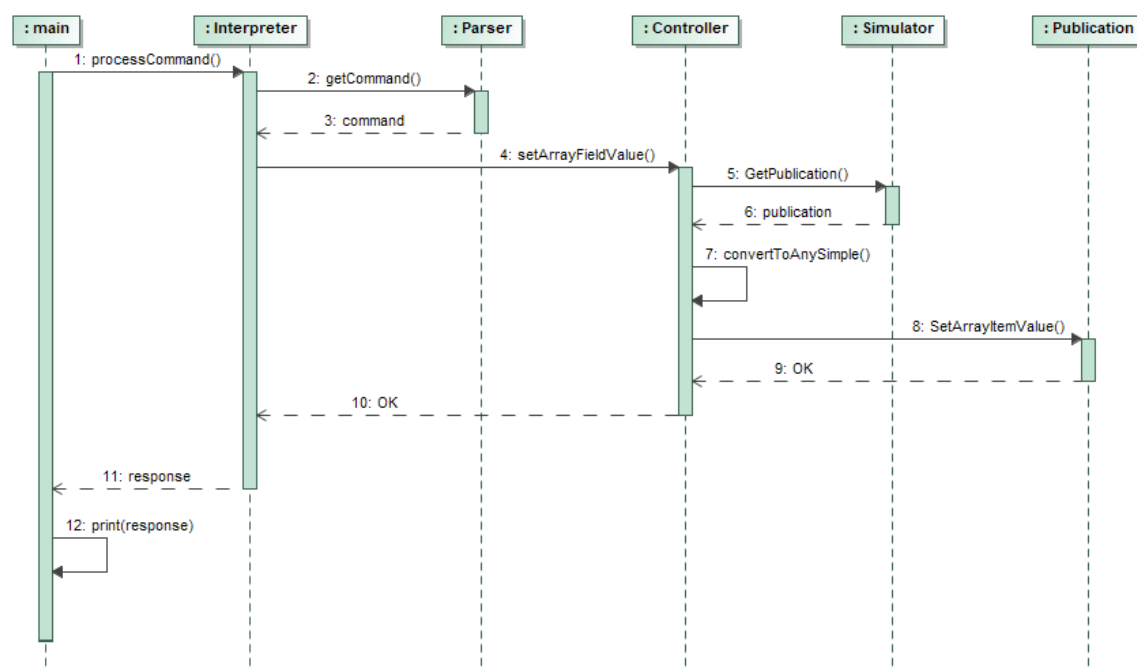


Figura 72. Diagrama de secuencia para CU-011.

12. Modificar tiempo de época

CU-012	
Descripción	El sistema deberá permitir al usuario modificar el valor inicial del tiempo de época.
Secuencia normal	7) El usuario escribirá en la línea de comandos el token SET EPOCH y el nuevo tiempo de época. 8) El sistema modificará el tiempo de época en el entorno de simulación. 9) El sistema notificará al usuario de que el tiempo de época se ha actualizado correctamente.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el tiempo de época introducido no es correcto, se mostrará un mensaje de error al usuario.

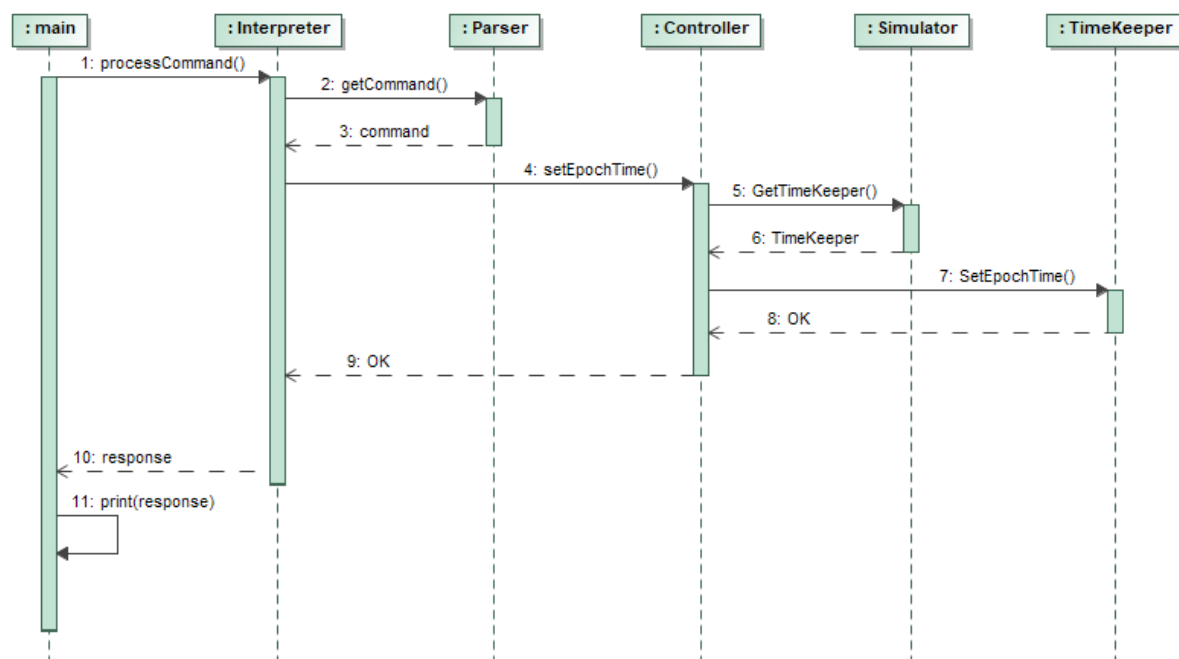


Figura 73. Diagrama de secuencia para CU-012.

13. Modificar tiempo de misión

CU-013	
Descripción	El sistema deberá permitir al usuario modificar el valor inicial del tiempo de misión.
Secuencia normal	10) El usuario escribirá en la línea de comandos el token SET MISSION y el nuevo tiempo de misión. 11) El sistema modificará el tiempo de misión en el entorno de simulación. 12) El sistema notificará al usuario de que el tiempo de misión se ha actualizado correctamente.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si el tiempo de misión introducido no es correcto, se mostrará un mensaje de error al usuario.

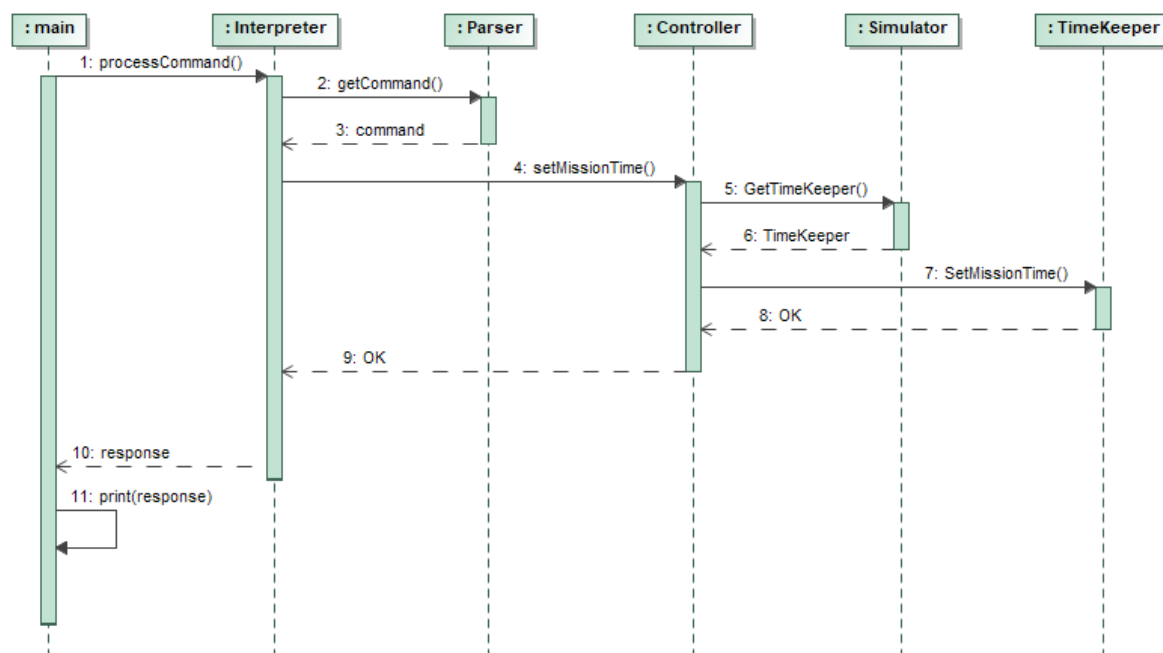


Figura 74. Diagrama de secuencia para CU-013.

14. Ejecutar operación publicada

CU-014	
Descripción	El sistema deberá permitir al usuario ejecutar una operación publicada por un modelo.
Secuencia normal	1) El usuario escribirá en la línea de comandos el token EXECUTE, la ruta completa de la operación y los argumentos de la misma. 2) El sistema ejecutará la operación especificada. 3) El sistema notificará al usuario de que la operación se ha ejecutado correctamente.
Extensiones	2a) Si no se ha cargado un ensamblado previamente, se mostrará un mensaje de error al usuario. 2b) Si la operación especificada no existe, se mostrará un mensaje de error al usuario. 2c) Si el número especificado de parámetros es incorrecto, se mostrará un mensaje de error al usuario. 2d) Si alguno de los argumentos especificados tiene un valor o tipo incorrecto, se mostrará un mensaje de error al usuario.

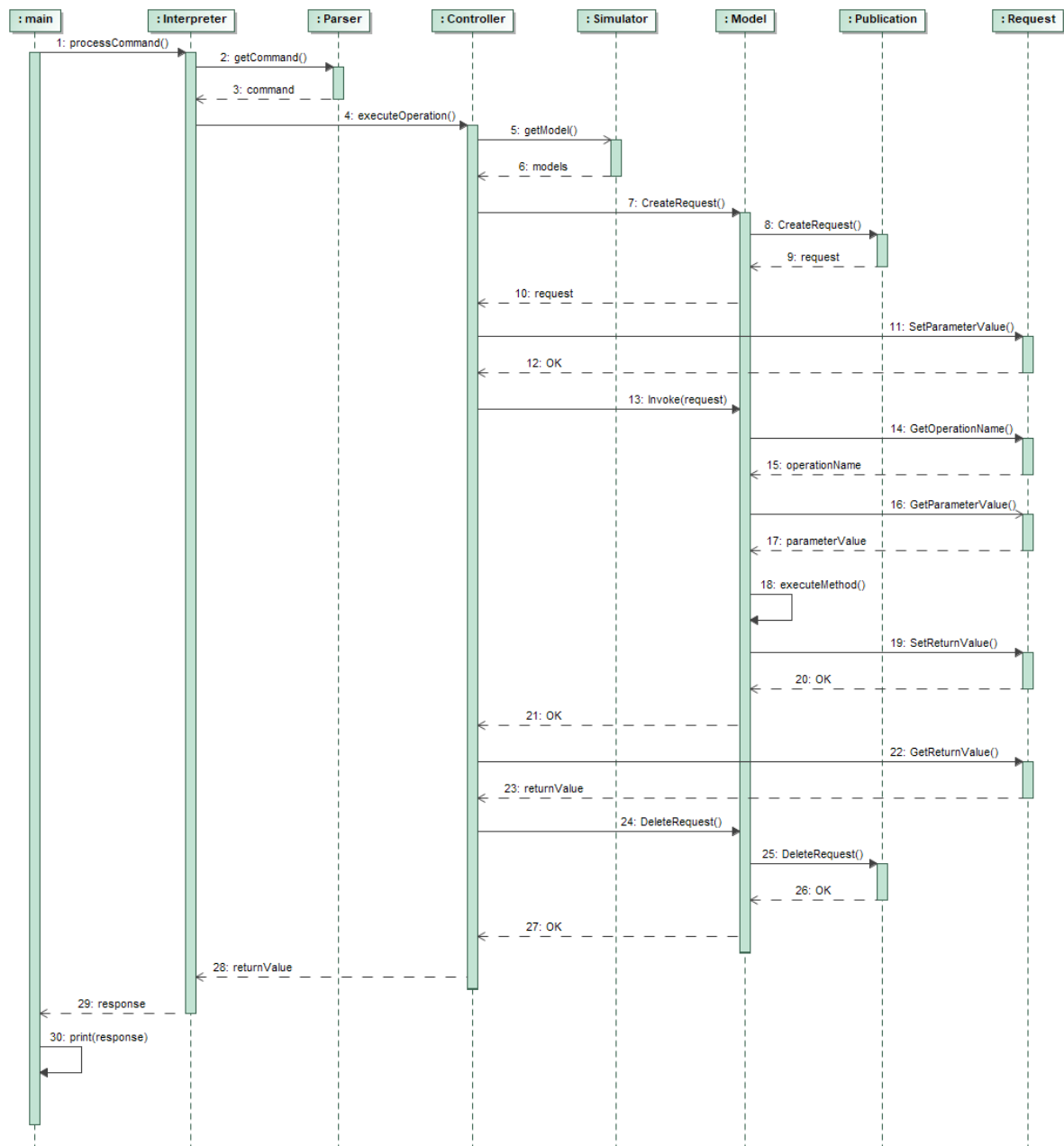


Figura 75. Diagrama de secuencia para CU-014.

Anexo 4. Ficheros del código fuente

Se listan a continuación los ficheros que componen el código fuente entregado adjunto a esta memoria, así como una pequeña descripción de cada uno de ellos.

Fichero	Descripción
Makefile	Conjunto de directivas para compilar el entorno de simulación, el intérprete SMP2 y los modelos SMP2.
include/Smp	<i>Directorio que contiene las interfaces SMP2 a implementar por el entorno de simulación y los modelos.</i>
src/Controller/Controller.h	Cabecera con la definición de la clase Smp2Simulator::Controller:Controller.
src/Controller/Controller.cpp	Implementación de la clase Smp2Simulator::Controller:Controller.
src/Core/Common.h	Cabecera con la definición de estructuras y métodos comunes utilizados por el entorno de simulación.
src/Core/Common.cpp	Implementación de métodos comunes utilizados por el entorno de simulación.
src/Core/EventManager.h	Cabecera con la definición de la clase Smp2Simulator::Core::EventManager.
src/Core/EventManager.cpp	Implementación de la clase Smp2Simulator::Core::EventManager.
src/Core/GenericComponent.h	Cabecera con la definición de la clase Smp2Simulator::Core::GenericComponent.
src/Core/GenericComponent.cpp	Implementación de la clase Smp2Simulator::Core::GenericComponent.
src/Core/Logger.h	Cabecera con la definición de la clase Smp2Simulator::Core::Logger.
src/Core/Logger.cpp	Implementación de la clase Smp2Simulator::Core::Logger.
src/Core/Publication.h	Cabecera con la definición de la clase Smp2Simulator::Core::Publication.
src/Core/Publication.cpp	Implementación de la clase Smp2Simulator::Core::Publication.
src/Core/PublishOperation.h	Cabecera con la definición de la clase Smp2Simulator::Core::PublishOperation.
src/Core/PublishOperation.cpp	Implementación de la clase Smp2Simulator::Core::PublishOperation.
src/Core/Request.h	Cabecera con la definición de la clase Smp2Simulator::Core::Request.
src/Core/Request.cpp	Implementación de la clase Smp2Simulator::Core::Request.
src/Core/Scheduler.h	Cabecera con la definición de la clase Smp2Simulator::Core::Scheduler.
src/Core/Scheduler.cpp	Implementación de la clase Smp2Simulator::Core::Scheduler.
src/Core/Simulator.h	Cabecera con la definición de la clase Smp2Simulator::Core::Simulator.
src/Core/Simulator.cpp	Implementación de la clase Smp2Simulator::Core::Simulator.

src/Core/TimeKeeper.h	Cabecera con la definición de la clase Smp2Simulator::Core::TimeKeeper.
src/Core/TimeKeeper.cpp	Implementación de la clase Smp2Simulator::Core::TimeKeeper.
src/Core/TypeRegistry.h	Cabecera con la definición de la clase Smp2Simulator::Core::TypeRegistry.
src/Core/TypeRegistry.cpp	Implementación de la clase Smp2Simulator::Core::TypeRegistry.
src/Interpreter/extern.h	Cabecera de interfaz externa en C con la definición de los métodos necesarios para la integración del simulador SMP2 en MATLAB.
src/Interpreter/extern.cpp	Implementación de los métodos de la de interfaz externa para la integración del simulador SMP2 en MATLAB.
src/Interpreter/Interpreter.h	Cabecera con la definición de la clase Smp2Simulator::Interpreter::Interpreter.
src/Interpreter/Interpreter.cpp	Implementación de la clase Smp2Simulator::Interpreter::Interpreter.
src/Interpreter/main.cpp	Fichero con el código del intérprete SMP2 en C++.
src/Interpreter/Parser.h	Cabecera con la definición de la clase Smp2Simulator::Interpreter::Parser.
src/Interpreter/Parser.cpp	Implementación de la clase Smp2Simulator::Interpreter::Parser.
src/Models/Adcs/AdcsController.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::AdcsController.
src/Models/Adcs/AdcsController.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::AdcsController.
src/Models/Adcs/Assembly.cpp	Fichero con el código del ensamblado de los modelos SMP2 del controlador ADCS.
src/Models/Adcs/EntryPoint.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::EntryPoint
src/Models/Adcs/EntryPoint.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::EntryPoint.
src/Models/Adcs/Gyroscope.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::Gyroscope
src/Models/Adcs/Gyroscope.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::Gyroscope.
src/Models/Adcs/Model.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::Model
src/Models/Adcs/Model.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::Model.
src/Models/Adcs/ReactionWheel.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::ReactionWheel
src/Models/Adcs/ReactionWheel.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::ReactionWheel.
src/Models/Adcs/StarTracker.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::StarTracker
src/Models/Adcs/StarTracker.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::StarTracker.
src/Models/Adcs/Pid/GenericPid.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::Pid::GenericPid
src/Models/Adcs/Pid/GenericPid.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::Pid::GenericPid

src/Models/Adcs/Pid/QuaternionFeedbackPd.h	Cabecera con la definición de la clase Smp2Simulator::Models::Adcs::Pid::QuaternionFeedbackPd
src/Models/Adcs/Pid/QuaternionFeedbackPd.cpp	Implementación de la clase Smp2Simulator::Models::Adcs::Pid::QuaternionFeedbackPd
src/Models/Example/Assembly.cpp	Fichero con el código del ensamblado del modelo SMP2 de ejemplo.
src/Models/Example/Example.h	Cabecera con la definición de la clase Smp2Simulator::Models::Example::Example
src/Models/Example/Example.cpp	Implementación de la clase Smp2Simulator::Models::Example::Example.
matlab/adcscontrollergui.fig	Fichero con la definición de la interfaz gráfica en GUIDE del controlador ADCS.
matlab/adcscontrollergui.m	Código de la interfaz gráfica del controlador ADCS.
matlab/dynamic_satellite.mdl	Implementación en Simulink del modelo dinámico del satélite.
matlab/normalize_quaternion.m	Implementación de función para normalizar un cuaternión.
matlab/quaternion_mult.m	Implementación de función para multiplicar dos cuaterniones.
matlab/satellite.m	Implementación de clase para gestionar la ejecución del modelo dinámico del satélite en Simulink.
matlab/smp2gui.fig	Fichero con la definición de la interfaz gráfica en GUIDE del intérprete SMP2 en Matlab.
matlab/smp2gui.m	Código de la interfaz gráfica del intérprete SMP2 en Matlab.
matlab/smp2manager.m	Implementación de clase para gestionar la ejecución de los métodos del entorno de simulación SMP2.

Anexo 5: Entorno de desarrollo

Para el desarrollo en C++ (entorno de simulación, intérprete SMP2 y modelos SMP2) se ha trabajado con el sistema operativo **CentOS 7** y las siguientes herramientas.

- Compilador GNU C++
- Eclipse Neón 3 IDE for C/C++ Developers
- Make
- Git

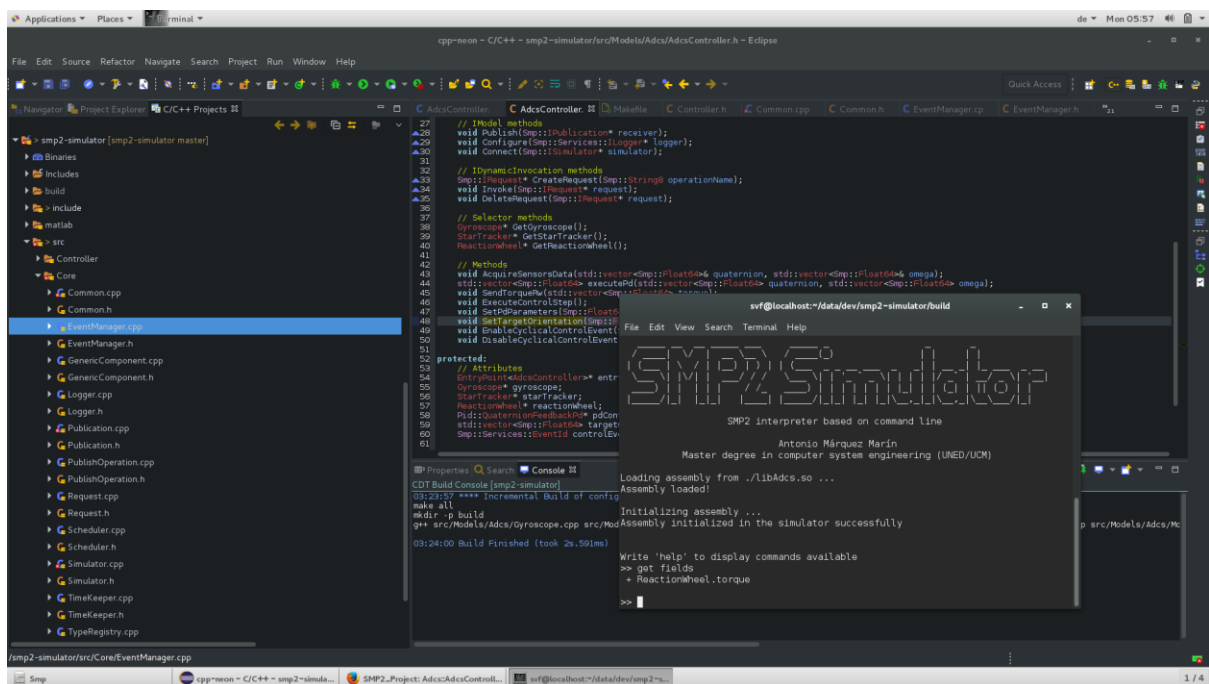


Figura 76. Entorno de desarrollo en CentOS 7.

En cuanto al desarrollo en Matlab (integración, intérprete GUI SMP2, modelos dinámicos) se ha trabajado con el sistema operativo **Windows 10** y las siguientes herramientas.

- Matlab
- Compilador MinGW C++
- Simulink
- Make
- Git

- Cygwin

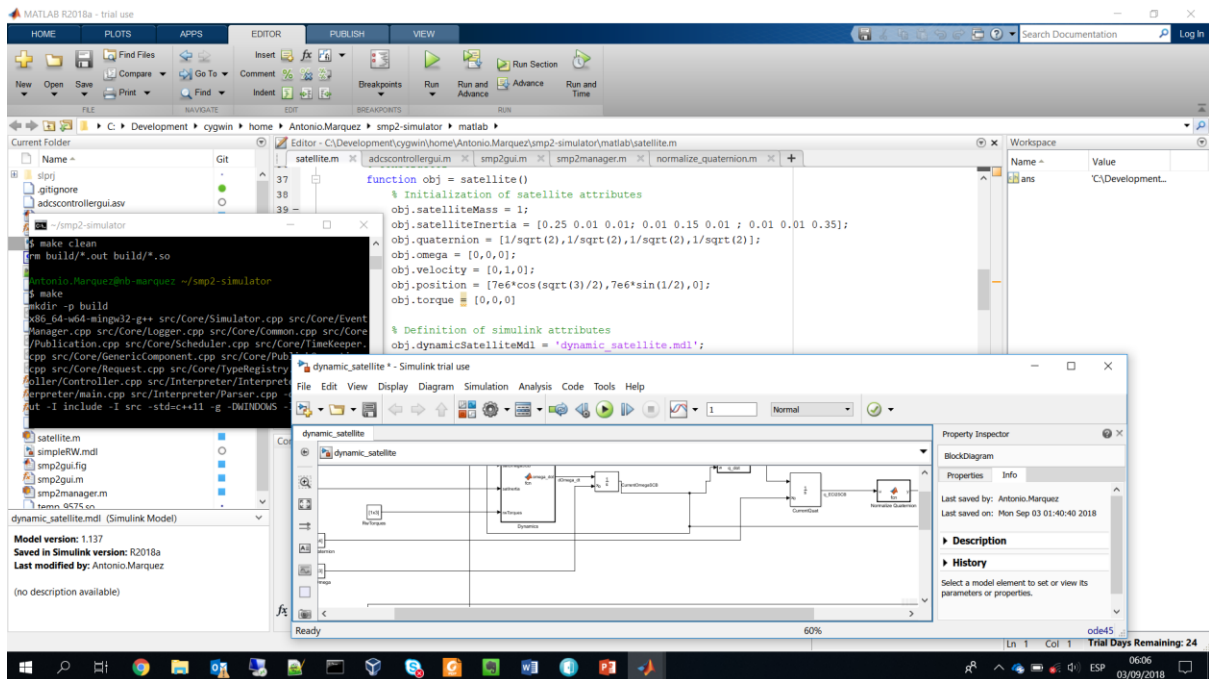


Figura 77. Entorno de desarrollo en Windows 10.

Anexo 6: Contenido de Makefile

Para compilar el sistema en **Linux**, se puede hacer uso de la herramienta GNU *Make* y del fichero **Makefile** con las directivas de compilación, cuyo contenido se presenta en este anexo.

Para la compilación en **Windows**, se recomienda utilizar **Cygwin** con el compilador de C++ MinGW instalado. De este modo, se puede realizar la compilación de la misma manera que en Linux, aunque antes es necesario establecer un valor para la variable de entorno WINDOWS (*export WINDOWS=1*).

La compilación del sistema completo, así como de cada uno de los módulos del mismo, se puede realizar escribiendo los siguientes comandos dentro del directorio raíz del proyecto.

- **make**. Compila todo el sistema.
- **make interpreter**. Compila el intérprete SMP2 para consola.
- **make models**. Compila los modelos SMP2.
- **make wrapper**. Compila la interfaz externa a utilizar en Matlab.

```
# project name
PROJECT_NAME := SMP2-Simulator

# compiler
ifdef WINDOWS
CC := x86_64-w64-mingw32-g++
else
CC := g++
endif

# build folder
BUILD_DIR := build

# matlab folder
MATLAB_DIR := matlab

# source code of core
SRC_CORE := src/Core/Simulator.cpp src/Core/EventManager.cpp src/Core/Logger.cpp \
            src/Core/Common.cpp src/Core/Publication.cpp src/Core/Scheduler.cpp
src/Core/TimeKeeper.cpp \
            src/Core/GenericComponent.cpp src/Core/PublishOperation.cpp
src/Core/Request.cpp src/Core/TypeRegistry.cpp

# source code and binaries of smp2 interpreter
SRC_INTERPRETER := src/Interpreter/Interpreter.cpp src/Interpreter/main.cpp
src/Interpreter/Parser.cpp
OBJ_INTERPRETER := $(BUILD_DIR)/main.out

# source code and binaries of smp2 models
SRC_EXAMPLE_MODEL := src/Models/Example/Example.cpp src/Models/Example/Assembly.cpp
OBJ_EXAMPLE_MODEL := $(BUILD_DIR)/libExample.so
SRC_ADCS_MODEL := src/Models/Adcs/Gyroscope.cpp src/Models/Adcs/ReactionWheel.cpp
src/Models/Adcs/StarTracker.cpp \
            src/Models/Adcs/AdcsController.cpp
src/Models/Adcs/Model.cpp src/Models/Adcs/Assembly.cpp \
            src/Models/Adcs/Pid/GenericPid.cpp
src/Models/Adcs/Pid/QuaternionFeedbackPd.cpp
OBJ_ADCS_MODEL := $(BUILD_DIR)/libAdcs.so

# source code of smp2 controller
SRC_CONTROLLER := src/Controller/Controller.cpp

# source code, header and binaries of smp2 wrapper
```

```

SRC_WRAPPER      :=      src/Interpreter/extern.cpp      src/Interpreter/Interpreter.cpp
src/Interpreter/Parser.cpp
HEADER_WRAPPER := src/Interpreter/extern.h
OBJ_WRAPPER := $(BUILD_DIR)/extern.so

# flags
COMMON_FLAGS := -std=c++11 -g
ifdef WINDOWS
COMMON_FLAGS += -DWINDOWS
endif
SHARED_FLAGS = -shared -fPIC
INTERPRETER_FLAGS := $(COMMON_FLAGS) -ldl -lpthread
MODELS_FLAGS := $(COMMON_FLAGS) $(SHARED_FLAGS)
WRAPPER_FLAGS := $(INTERPRETER_FLAGS) $(SHARED_FLAGS)
INCLUDE_FLAGS = -I include -I src

all: $(OBJ_INTERPRETER) $(OBJ_EXAMPLE_MODEL) $(OBJ_ADCS_MODEL) $(OBJ_WRAPPER)

interpreter: $(OBJ_INTERPRETER)

$(OBJ_INTERPRETER): $(SRC_INTERPRETER) $(SRC_CONTROLLER) $(SRC_CORE)
    mkdir -p $(BUILD_DIR)
    $(CC) $(SRC_CORE) $(SRC_CONTROLLER) $(SRC_INTERPRETER) -o $(OBJ_INTERPRETER)
$(INCLUDE_FLAGS) $(INTERPRETER_FLAGS)

models: $(SRC_EXAMPLE_MODEL) $(OBJ_ADCS_MODEL)

$(OBJ_EXAMPLE_MODEL): $(SRC_EXAMPLE_MODEL)
    mkdir -p $(BUILD_DIR)
    $(CC) $(SRC_EXAMPLE_MODEL) -o $(OBJ_EXAMPLE_MODEL) $(INCLUDE_FLAGS) $(MODELS_FLAGS)

$(OBJ_ADCS_MODEL): $(SRC_ADCS_MODEL)
    mkdir -p $(BUILD_DIR)
    $(CC) $(SRC_ADCS_MODEL) -o $(OBJ_ADCS_MODEL) $(INCLUDE_FLAGS) $(MODELS_FLAGS)

wrapper: $(OBJ_WRAPPER)

$(OBJ_WRAPPER): $(SRC_CORE) $(SRC_CONTROLLER) $(SRC_WRAPPER) $(HEADER_WRAPPER)
    mkdir -p $(BUILD_DIR)
    $(CC) $(SRC_CORE) $(SRC_CONTROLLER) $(SRC_WRAPPER) -o $(OBJ_WRAPPER) $(INCLUDE_FLAGS)
$(WRAPPER_FLAGS)
    @cp $(OBJ_WRAPPER) $(MATLAB_DIR)
    @cp $(HEADER_WRAPPER) $(MATLAB_DIR)

clean: $(BUILD_DIR)
    rm $(BUILD_DIR)/*.out $(BUILD_DIR)/*.so

```