PROYECTO INTEGRADOR DE LA CARRERA DE INGENIERÍA EN TELECOMUNICACIONES

GENERACIÓN DE MODELOS PARA SIMULADOR DE CARGA ÚTIL DE SATÉLITES GEO

Martín Gejo

Ing. Leonardo Handsztok
Director

Ing. Adrián García
Co-director

Miembros del Jurado
Dr. Fernando Bianchi (Instituto Balseiro)
Ing. Gustavo Wiman (INVAP)

17 de diciembre de 2018

Centro Atómico Bariloche

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

Resumen

En el presente trabajo se describe el diseño de una librería en C++ con la cual se pueden integrar simuladores de carga útil de satélites geostacionarios de comunicaciones. El trabajo fue realizado en el grupo de Segmento Terreno de INVAP, y la motivación para el mismo surge del hecho que los módulos de carga útil para los simuladores de ARSAT-1 y ARSAT-2 fueron provistos por el fabricante del componente. El objetivo del trabajo es lograr simuladores *standalone* de cargas útiles que sean fácilmente reconfigurables. Los componentes más importantes de una carga útil típica (y por ende, los que serán considerados en este trabajo) son el receptor (que cambia la frecuencia de las señales recibidas a la de bajada), el IMUX (que separa las distintas señales en distintos canales físicos), los TWTA (que amplifican las señales individuales), y el OMUX (que une todas las señales amplificadas en un único canal).

Para cada componente, se debe poder modelar su comportamiento térmico, consumo eléctrico, y efecto en la señal de RF que tiene a la entrada. Esto se logró desarrollando tres subsistemas, correspondientemente llamados **thermalManager**, **powerModel**, y **payloadModel**. Cada subsistema posee sus propios métodos para mantener actualizadas sus variables, y estos métodos deben llamarse con una cierta frecuencia para que la simulación sea representativa de lo que pasa en la realidad. En el trabajo también se describen los modelos de componente utilizados en el desarrollo de sus correspondientes clases, y se conforma una carga útil de prueba con la finalidad de validarlos.

Abstract

This document describes the design of a C++ library with which geostationary communications satellite payload simulators can be integrated. The work described was done at INVAP's Ground Segment group, and it's motivation stems from the fact that the payload modules for ARSAT-1 and ARSAT-2 simulators were provided by the payload manufacturer. The objective of this work is to achieve standalone payload simulators that can be easily reconfigured. The most important components of a typical payload (and by extension, the ones that will be considered for this work) are the receiver (that downconverts the received signal frequency to the corresponding downlink frequency), the IMUX (that separates the different signals in different physical channels), the TWTA (that amplifies an individual signal), and the OMUX (that joins all the amplified signals in a single channel).

For each component, its thermal behaviour, power consumption, and effect on the RF signal at its input have to be modeled. This was achieved by developing three subsistems, correspondingly called **thermalManager**, **powerModel**, and **payloadModel**. Each subsistem has its own methods to keep its own variables updated, and these methods have to be called with a certain frequency so that the simulation is realistic. This document also describes the component models used in the development of their corresponding classes, and a test payload is formed to validate them.

"Tenés toda la vida para vivir afuera del pabellón, pero sólo 3 años para vivir adentro."

Michel Gartner

"Los finales son como los goles. No se merecen, se hacen."

Livio Leiva

Nomenclatura

API	Application Programming Interface
CAMP	Channel Amplifier
CPSim	Custom Payload Simulator
DB	Database
DSS	Dynamic Satellite Simulator
EPC	Electronic Power Conditioner
HMI	Human-Machine Interface
HPA	High Power Amplifier
IMUX	Input Multiplexer
LNA	Low Noise Amplifier
OMUX	Output Multiplexer
PLSIM	Thales' Payload Simulator Module
RF	Radiofrecuencia
SAOCOM	Satélite Argentino de Observación con Microondas
SCC	Satellite Control Center
SMP	Simulation Model Portability
STL	C++ Standard Library
TWTA	Traveling-wave Tube Amplifier
TWT	Traveling-wave Tube
UML	Unified Modeling Language

Índice general

Capítulo 1

Introducción

En los proyectos satelitales se requiere una inversión de cientos de millones de dólares (excepto en los cubesat). Para estos proyectos, los modelos de vuelo e ingeniería tienen una disponibilidad acotada, y su costo es considerablemente mayor al de un simulador. Por otro lado, en un simulador se pueden probar protocolos frente a fallas sin causar las mismas en el modelo de vuelo. Es debido a esto que en los proyectos satelitales se utilizan extensivamente los simuladores.

El grupo de Segmento Terreno de INVAP es el encargado del desarrollo y mantenimiento de los simuladores de los proyectos en los que la empresa participa. estos simuladores obedecen un estándar de la Agencia Espacial Europea (ESA, por sus siglas en inglés) llamado Simulation Model Portability 2 (SMP2), y son de diseño modular. En el caso de los proyectos de satélites geoestacionarios realizados hasta la fecha (ARSAT-1 y ARSAT-2), el módulo correspondiente a la carga útil (el equipo encargado de la retransmisión de las señales) fue provisto por la empresa fabricante del mismo, en lugar de ser desarrollado por el grupo. Debido a que el grupo no tiene acceso al código fuente de estos módulos, existe interés para desarrollar los módulos de simulador de carga útil *in-house*.

este trabajo detalla el diseño y desarrollo de una librería en C++ mediante la cual se pueden implementar simuladores de carga útil *standalone* que obedecen SMP2, y sirve de puntapié inicial para poder desarrollar módulos compatibles con los simuladores que desarrolla el grupo.

En el capítulo 2 se explica en detalle que es un carga útil y cuales son sus componentes más importantes. En el capítulo 3 se describen las clases que componen la librería, y los modelos que se adoptan para los componentes. En el capítulo 4, se describe la utilización de la librería en un simulador de prueba, y se muestra que su comportamiento no es anómalo. Finalmente, en el capítulo 5, se da una breve descripción de cómo se debería escalar el proyecto para poder integrar el trabajo realizado en un producto útil.

Capítulo 2

Conceptos básicos

En este capítulo se describen los conceptos básicos referidos a los satélites geoestacionarios de comunicaciones, y cómo son los simuladores que desarrolla el grupo de Segmento Terreno. Toda la información general referida a los satélites y sus componentes se puede encontrar en [1] y [2].

2.1. Satélite geoestacionario de comunicaciones

Los satélites geoestacionarios de comunicaciones funcionan básicamente cómo espejos en el espacio. Dos o más estaciones terrenas se comunican por intermedio de un satélite, como se muestra en la figura 2.1.

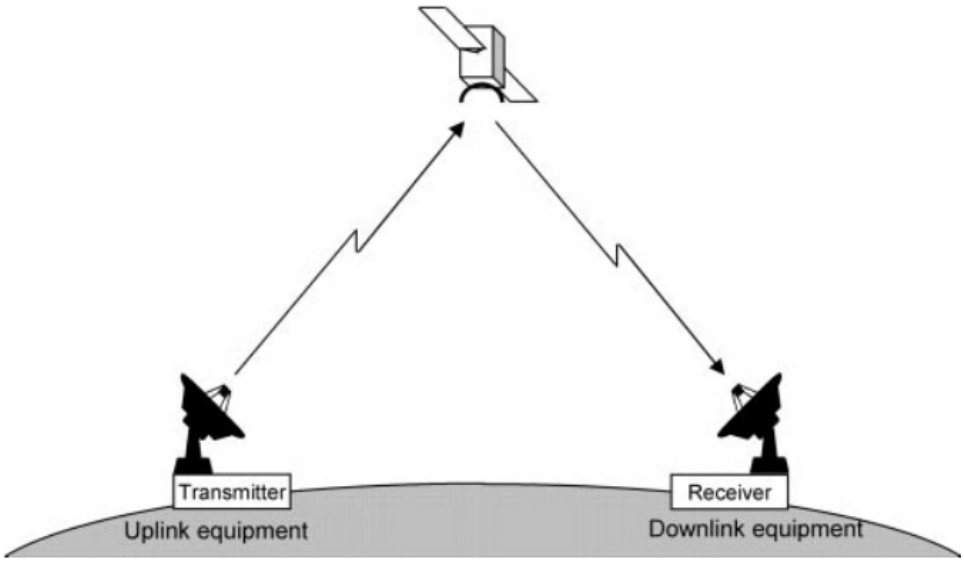

Figura 2.1: Enlace satelital entre dos estaciones terrenas [1].

Los satélites se pueden descomponer en dos grandes subsistemas:la carga útil o *payload*, que es el equipamiento específico a la finalidad del satélite (instrumentos, transmisores, etc.); y la plataforma de servicios, que provee a la carga útil de potencia, control térmico,

control de órbita y actitud[1], etcétera. En los satélites de comunicaciones, la carga útil consiste de al menos una cadena de amplificación de señales de radiofrecuencia, y dependiendo del proyecto, se pueden considerar a las antenas como parte de la carga útil o de la plataforma. Las cargas útiles de comunicaciones pueden ser de dos tipos: los de tipo *bent-pipe*, que retransmiten las señales que reciben sin procesarlas; y las procesadoras, que hacen algún tipo de procesamiento a la señal antes de transmitirla. De aquí en más, cuando aparezca el término "carga útil", será en referencia a una carga útil de comunicaciones de tipo bent-pipe[1][2].

2.2. Descripción de una carga útil

En general, todas las cargas útiles respetan un esquema común, que se presenta en la figura 2.2.

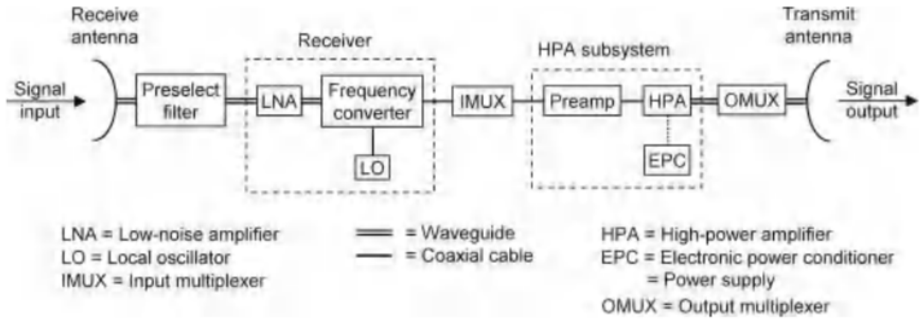

Figura 2.2: Cadena funcional típica de una carga útil [1].

Cuando una señal[2] llega al satélite desde la tierra, primero pasa por el filtro de preselección para minimizar el ruido y las interferencias. Después, la señal pasa por el receptor, que suele estar compuesto por un LNA (Low Noise Amplifier) y un downconverter. La función del receptor es cambiar la frecuencia de las señales a retransmitir, dado que si no estuviera esta etapa, las señales transmitidas interferirían a las recibidas.

Luego, el IMUX (Input Multiplexer), envía la señal a su correspondiente amplificador, según su frecuencia. El IMUX, del cual un esquema funcional se muestra en la figura 2.3, está básicamente compuesto por filtros pasabanda sintonizados para cada canal, y circuladores. Las componentes de la señal recibida que son rechazadas por el primer filtro son circuladas al segundo, y así sucesivamente. Si bien esto causa pérdidas adicionales a los canales físicos que corresponden a los últimos filtros de la cadena, dichas pérdidas se pueden compensar en la etapa de amplificación. La función del IMUX es separar los

[1]En el sector aeroespacial, se le dice actitud a la orientación con respecto a un sistema de referencia inercial.

[2]A excepción de los telecomandos, que toman un camino que los lleva a la computadora de a bordo. Esto excede al presente trabajo.

distintos canales en distintos caminos, de forma de evitar efectos de batido en la etapa de amplificación.

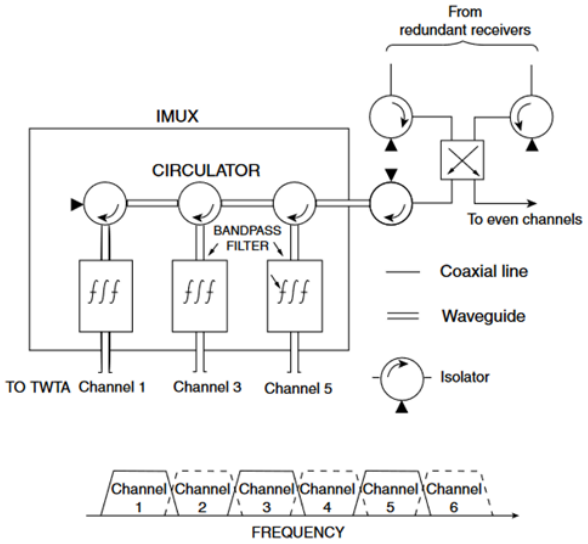

Figura 2.3: Esquema funcional de un IMUX [2].

Después del IMUX se encuentra la etapa de amplificación, que se suele llamar HPA (High Power Amplifier). Existen dos tecnologías de amplificador de alta potencia para aplicaciones satelitales: los SSPA (Solid State Power Amplifier) y los TWTA (Traveling Wave Tube Amplifier). Por lejos, estos últimos son los más comunes en los satélites geoestacionarios de telecomunicaciones, y por esa razón, serán los únicos tenidos en cuenta en el presente trabajo. La etapa de amplificación es la más importante de un satélite de telecomunicaciones, y es donde se encuentran los componentes más caros. Un TWTA se fabrica a pedido con un tiempo de espera de aproximadamente un año, y su costo ronda los cien mil dólares.

En general, un TWTA está compuesto por tres componentes, aunque en la implementación existen distintas arquitecturas (ejemplos de las cuales se muestran en la figura 2.4). El CAMP (Channel Amplifier) es un preamplificador con un alto rango de ganancias, cuya función es amplificar la señal a su entrada a la potencia que admite el TWT (Travelling Wave Tube), y puede tener un linealizador que predistorsiona la señal a la salida para compensar las no linealidades del TWT. Por su parte, el TWT es donde ocurre la amplificación de la señal que permite que la misma sea demodulable por una estación terrena que se encuentra a 36.000 km de distancia. El último componente de un TWTA es el EPC (Electronic Power Conditioner), que provee la tensión necesaria para la operación del CAMP y el TWT.

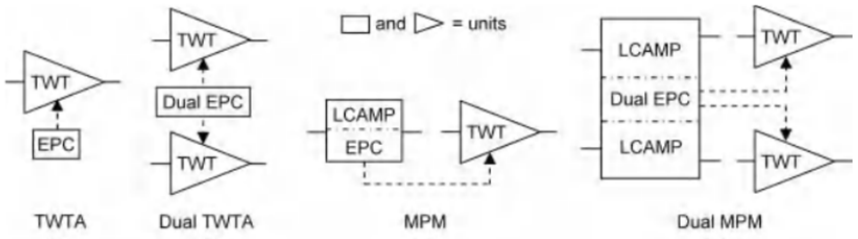

Figura 2.4: Distintas arquitecturas de TWTA [1].

Una vez que las señales de todos los canales son amplificados por separado, el OMUX (Output Multiplexer) las junta en un único canal físico, para poder ser enviadas a la antena transmisora. El OMUX consiste de una guía de ondas a la que se le acoplan las salidas de los distintos amplificadores. El OMUX se diseña específicamente para minimizar las pérdidas, ya que en esta etapa, cualquier pérdida es potencia de los amplificadores que se desperdicia.

Otros componentes de importancia son los heaters, que son resistencias utilizadas para el control térmico, y los switches, que se utilizan para agregarle redundancia a los componentes y subsistemas, de forma de no perder funcionalidad en caso de fallas.

2.3. Aspectos de simulación

Cómo se mencionó anteriormente, las simulaciones son una herramienta usada a lo largo de la duración de un proyecto satelital, desde sus etapas más tempranas, hasta que se alcanza el final de la vida útil del correspondiente satélite. Existen varios tipos de simuladores, y el de interés para el presente trabajo es el denominado simulador de operaciones. este es un simulador de propósito general que permite simular a grandes rasgos todos los aspectos del satélite. Se usa para entrenamiento de operadores, planificación de maniobras de vuelo, validación y mantenimiento de procedimientos de vuelo, y calificación de sistemas de control terreno, entre otras cosas. En la figura 2.5 se muestra esquemáticamente la composición típica de los simuladores que desarrolla el grupo de Segmento Terreno.

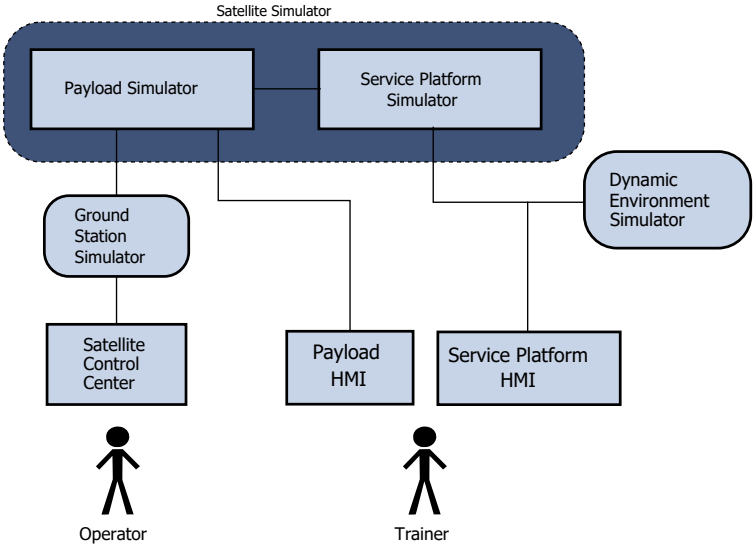

Figura 2.5: Esquema simplificado de la composición de DSS .

En este ejemplo, el entrenador configura el escenario de entrenamiento, y el operador interactúa con el simulador a través del SCC (Satellite Control Center), de la misma forma que lo haría si estuviera interactuando con el satélite.

Los simuladores desarrollados por el grupo obedecen un estándar llamado Simulation Model Portability 2 (SMP). Este estándar concierne al desarrollo de simuladores, y su propósito es promover tanto la portabilidad de modelos entre distintos ambientes de simulación y sistemas operativos, como la reutilización de los modelos. Se basa en una jerarquía de componentes y en las interfaces a través de las cuales se comunican. Como fue diseñada pensando en C++ como plataforma de implementación, prevé el uso de clases y de herencia múltiple[3], pero en teoría puede implementarse en cualquier lenguaje de programación. En la figura 2.6 se muestra la arquitectura que debe respetar un simulador que obedece SMP2.

[3]La herencia múltiple en SMP2 está prevista solo para clases abstractas, pero en la práctica se puede utilizar para herencia de implementación, si se tiene mucho cuidado.

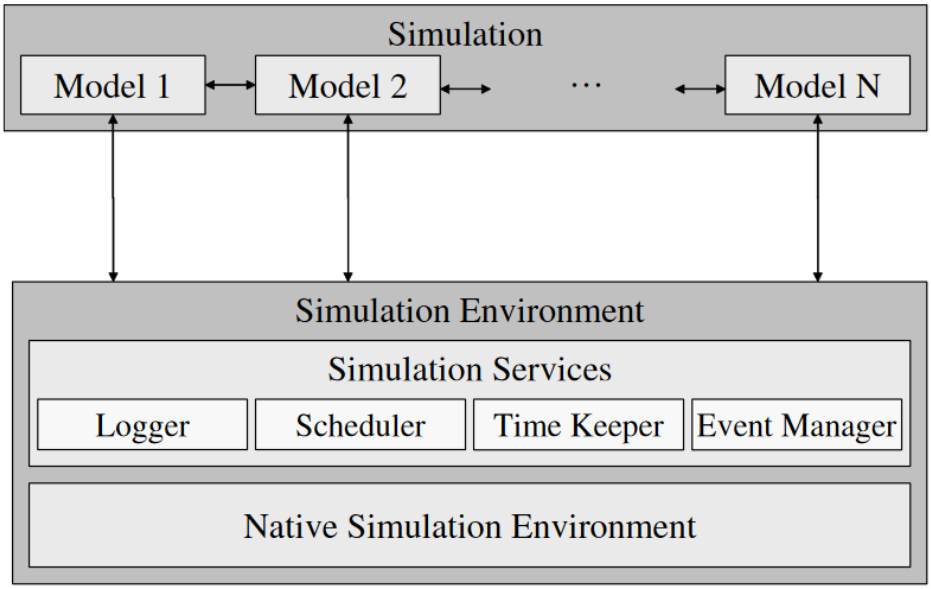

Figura 2.6: Arquitectura de un simulador según SMP2 [3].

El estándar [3] tiene una extensión considerable, pero lo que es importante entender en el contexto de este trabajo, es que un entorno de simulación tiene que proveer acceso a un servicio llamado Time Keeper, que es básicamente el reloj de la simulación; y a un servicio llamado Scheduler, cuya responsabilidad es invocar ciertas funciones llamadas entry points una dada cantidad de veces (que puede ser infinita), a partir de un determinado tiempo transcurrido en la simulación, y con una determinada frecuencia. Los entry points son métodos propios de los modelos, que no poseen ni argumentos ni valor de retorno. Por ejemplo, si se está simulando un componente con un reloj cuya resolución es de un milisegundo, se puede definir un entry point que le sume un milisegundo a la hora que lleva el reloj, y cada un millón de nanosegundos[4] el scheduler llamaría a ese entry point.

En la figura 2.7 se muestra la ventana principal de la interfaz gráfica del simulador de operaciones utilizado en ARSAT-1 y ARSAT-2, llamado Dynamic Satellite Simulator (DSS). Este simulador es de diseño modular, obedece hasta cierto punto a SMP2, y tiene una API a través de la cual se pueden consultar y modificar todas las variables de la simulación. Tanto las interfaces gráficas como los scripts de configuración y testing hacen uso de dicha API.

[4]La resolución de las simulaciones en SMP2 es de un nanosegundo.

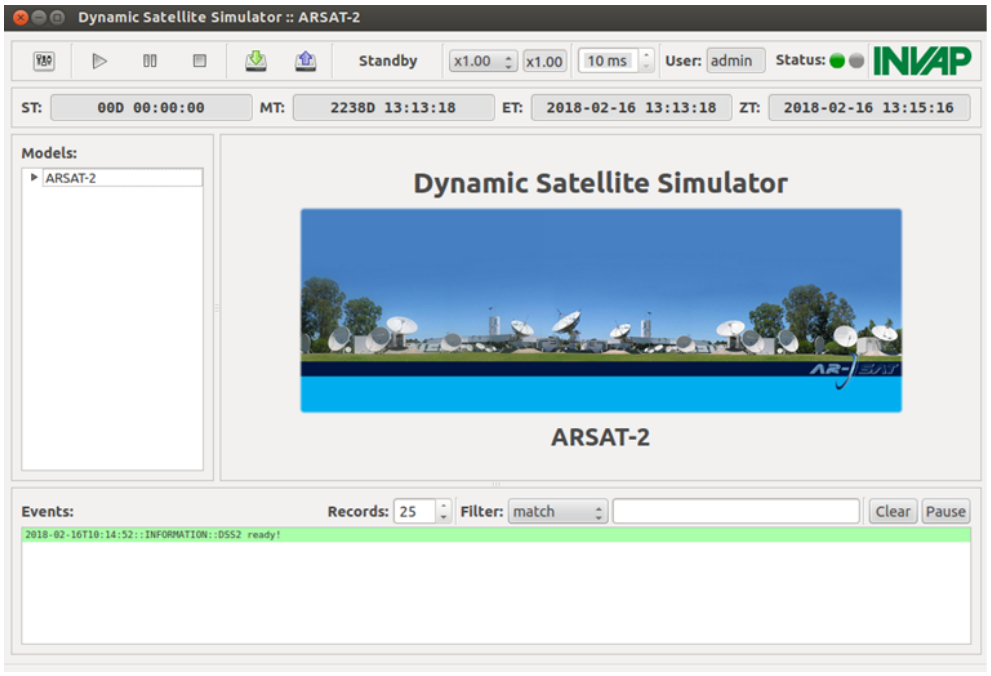

Figura 2.7: Ventana principal de la interfaz gráfica del simulador de operaciones de ARSAT-2.

Para los simuladores de operaciones de ARSAT-1 y ARSAT-2, el módulo de carga útil fue provisto por la misma empresa que la fabricó, Thales Alenia Space. La figura 2.8 pertenece a un documento de uso interno de INVAP, y muestra la integración del simulador de carga útil (llamado PLSIM) dentro de DSS. PLSIM interpreta los telecomandos que se le envían a la carga útil, y genera su propia telemetría en función de sus variables internas. También calcula el consumo eléctrico de todos sus componentes. Como no posee funcionalidad para realizar simulaciones térmicas, estas deben ser hechas externamente, calculando las disipaciones[5] a partir de los consumos eléctricos, y usando una API para sobrescribir los valores de temperatura de los distintos componentes.

[5] A lo largo del presente trabajo, se le dirá disipación a la componente de la potencia consumida que se transforma en calor. Si bien no es la definición aceptada universalmente, es la que se usa tanto dentro del grupo cómo en el código de los simuladores.

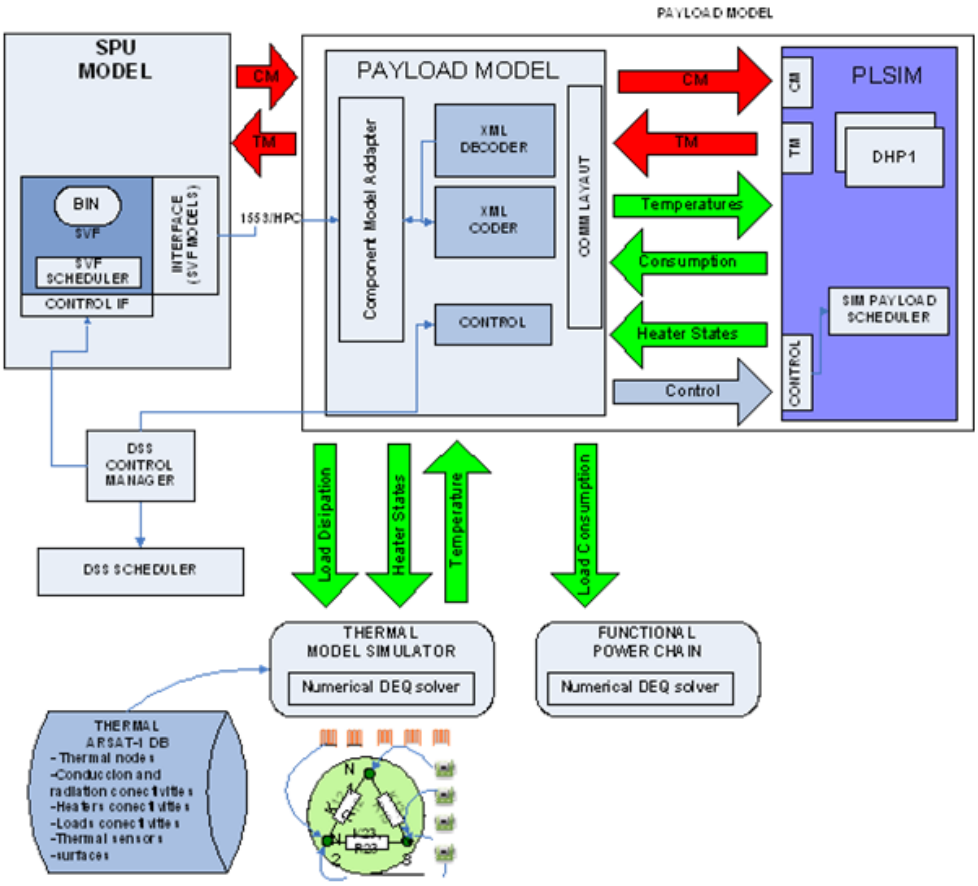

Figura 2.8: Integración de PLSIM dentro de DSS.

El trabajo consiste en diseñar una librería en C++ que permita modelar el comportamiento de los componentes más importantes de una carga útil, de forma de poder simular la carga útil de cualquier satélite e independizarse del fabricante para los simuladores de los proyectos futuros. Los fenómenos a ser simulados pueden dividirse en tres partes: el modelo térmico de los componentes, los consumos y disipaciones de potencia eléctrica y la representación de las señales de RF dentro de la cadena funcional. Los componentes que debe incluir la librería son: IMUX, receptor, TWT, CAMP, EPC, y OMUX.

La librería deberá ser capaz de proveer los métodos para crear cargas útiles configurables, que sigan el patrón mostrado en la figura 2.2, pudiendo variar la cantidad de amplificadores sin requerir esfuerzo extra del desarrollador. Por ejemplo, si se tiene el código para generar el simulador correspondiente a la carga útil de la figura 2.9 (la cual tiene dos amplificadores), y se necesita agregarle un amplificador para que se corresponda a la carga útil de la figura 2.10, debería poder hacerse cambiando pocas líneas de código.

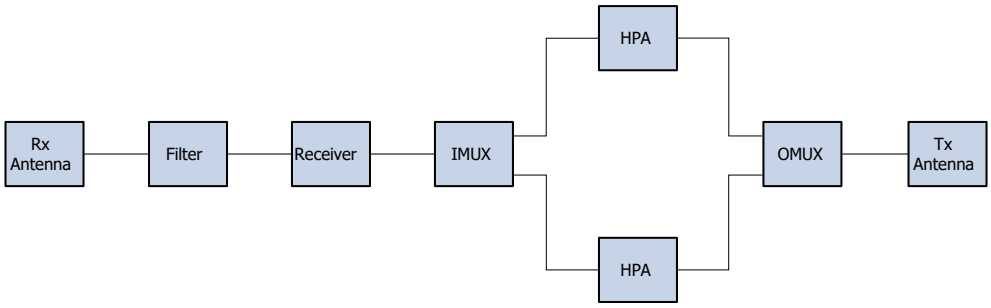

Figura 2.9: Carga útil compuesta por dos HPA.

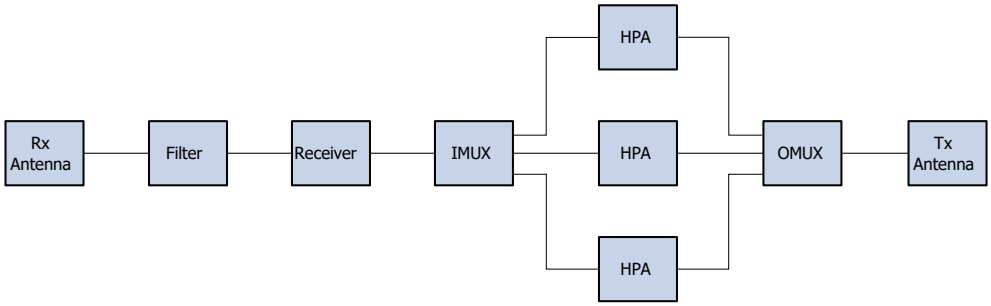

Figura 2.10: Carga útil compuesta por tres HPA.

Debido a que el simulador que se utilizó como banco de prueba de la librería se llama **CPSim**[6], siguiendo las reglas de nomenclatura del grupo, la librería desarrollada se llama **libcpsim**.

[6] Custom Payload Simulator

Capítulo 3

Composición de un simulador de carga útil

Un simulador de carga útil hecho con **libcpsim** tiene tres subsistemas[1] que interactúan entre sí, llamados **thermalManager**, **powerModel**, y **payloadModel**, como se muestra en la figura 3.1.

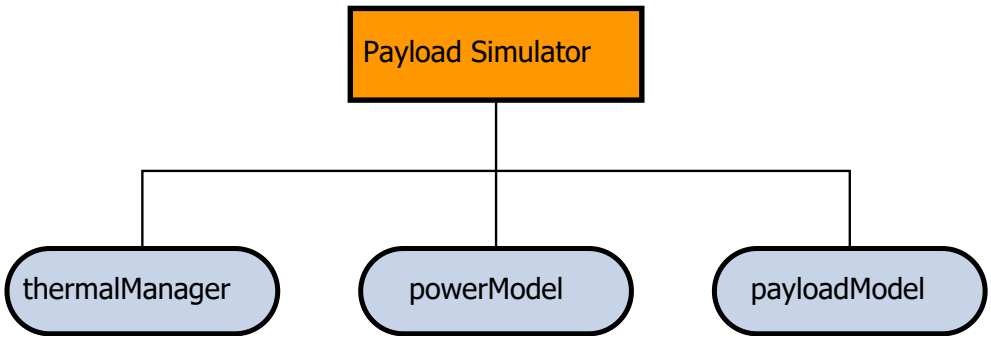

Figura 3.1: Esquema de los subsistemas que componen a un simulador de carga útil hecho con **libcpsim**.

El subsistema **thermalManager** es el encargado de modelar el comportamiento térmico, **powerModel** es el encargado de modelar los consumos de potencias de las distintas cargas, y **payloadModel** es el encargado de modelar el comportamiento de las señales de RF mientras viajan por la cadena funcional.

Para obedecer SMP2, el simulador debe proveer los servicios requeridos por el estándar, que son Time Keeper, Scheduler, Logger, y Event Manager. En la clase a la que pertenece el simulador, desarrollada a priori por el grupo, ya están integrados estos servicios.

[1] Los subsistemas son, a su vez, atributos de la clase a la que pertenece el simulador.

3.1. Subsistema térmico (thermalManager)

Un modelo térmico es un conjunto de grafos no orientados cuyos nodos son idénticos, pero poseen distintas aristas. Cada nodo representa un punto en una discretización del objeto que se quiere simular y tiene asignados un identificador único, una capacidad térmica (que es constante), y una temperatura (que puede variar con el tiempo). Dependiendo del grafo, las aristas pueden representar las conductividades o las radiatividades entre nodos. Se muestra en la figura 3.2 un posible ejemplo de discretización del modelo térmico de dos tubos amplificadores montados sobre un honeycomb. Todas las relaciones entre nodos son de conductividad, exceptuando la que está marcada con una 'R', que representa una relación de radiatividad. **CPSim** no tiene en cuenta las radiatividades entre elementos, así que su modelo térmico consiste de un único grafo, cuyas aristas representan las conductividades.

Figura 3.2: Posible discretización de dos tubos amplificadores y la superficie donde están montados. La R identifica la relación de radiatividad entre nodos que no tienen un camino de conducción entre ellos.

Afortunadamente, el grupo posee en su repositorio una librería para crear un modelo térmico y resolver la evolución temporal de su temperatura, con lo cual simplemente hay que integrarla en **libcpsim**. Los modelos térmicos se pueden crear tanto a partir de código fuente, como de la carga de una base de datos SQLite. Es posible incluso crear un modelo térmico a partir de una base de datos, y modificarlo desde código fuente, aunque por supuesto, esto requiere que el desarrollador conozca de antemano la base de datos.

La librería con la que se crea y resuelve el modelo térmico se llama **libthermalcore**. Dentro de ella, está la clase a la que pertenece el **thermalManager**. Esta posee como atributo al modelo térmico y provee los métodos para inicializar y modificar el mismo, y también para resolver un paso de tiempo de las temperaturas del modelo, dadas sus disipaciones. Esto se hace por medio del método Runge-Kutta de orden 4 para la ecuación discretizada del calor.

3.2. Subsistema de potencia eléctrica (powerModel)

Uno de los objetivos originales del proyecto era que las clases de **libcpsim** hagan uso del modelo de potencia usado en el simulador de SAOCOM[2]. Luego de analizar el código del mismo, se llegó a la conclusión de que eso no sería viable, debido a la dependencia

[2]Satélite Argentino de Observación con Microondas.

con clases de componentes específicos de SAOCOM. Por consiguiente, se optó por diseñar un modelo de potencia que sea autosuficiente, de forma que se pueda exportar a otros proyectos futuros. El modelo diseñado supone que todos los componentes son resistivos puros, y que hay una única fuente, que es de tensión. Bajo estas hipótesis, se diseñó un algoritmo basado en árboles[3] con raíz, que para una dada tensión aplicada, determina el consumo de corriente. Al ejecutar este algoritmo se actualizarán los valores de corriente y potencia del circuito en cada paso de simulación.

En la figura 3.3 se muestra un ejemplo de un circuito representado con el modelo de potencia de **libcpsim**. Las cargas son nodos hoja[4], y los subcircuitos serie y paralelo forzosamente deben tener hijos. Cada elemento calcula su resistencia, y con ella, la corriente que lo atraviesa. Para las cargas es su propia resistencia, mientras que para los nodos serie es la suma de resistencias de sus hijos, y para los nodos paralelo es el recíproco de la suma de conductancias de sus hijos. Cada nodo, independientemente de su tipo, tiene asignados valores de tensión y corriente para el componente o subcircuito que representa, de forma que se pueden realizar consultas a nivel componente y subcircuito. Esto es útil para mostrar de forma simple los valores a través de una interfaz gráfica. Por otro lado, cada hijo de un nodo paralelo tiene asignado una variable booleana, que representa una llave de corte: si está en 1, el subcircuito que corresponde a ese hijo está cerrado, y si está en 0, está abierto y se le asignan valores nulos de tensión y corriente.

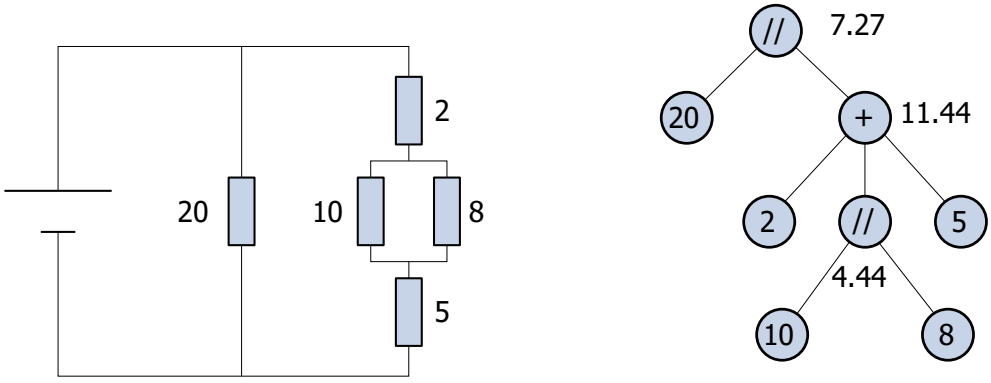

Figura 3.3: Ejemplo de construcción de un árbol de circuito en el modelo de potencia de **libcpsim**. El nodo serie se representa con "+", y los nodo paralelo con "//".

El circuito de ejemplo que se encuentra en la figura 3.3 fue recreado en un circuit solver con una tensión aplicada de 10V, como se muestra en la figura 3.4, y se replicó en una versión temprana de **CPSim**, de forma de validar el algoritmo. El output de la validación se presenta en la figura 3.5, y su resultado de corriente y tensión para cada elemento es idéntico al del circuit solver.

[3]En teoría de grafos, se llama árbol a un grafo conexo que no tiene ciclos.
[4]Se llama así a los nodos de un árbol que no tienen hijos.

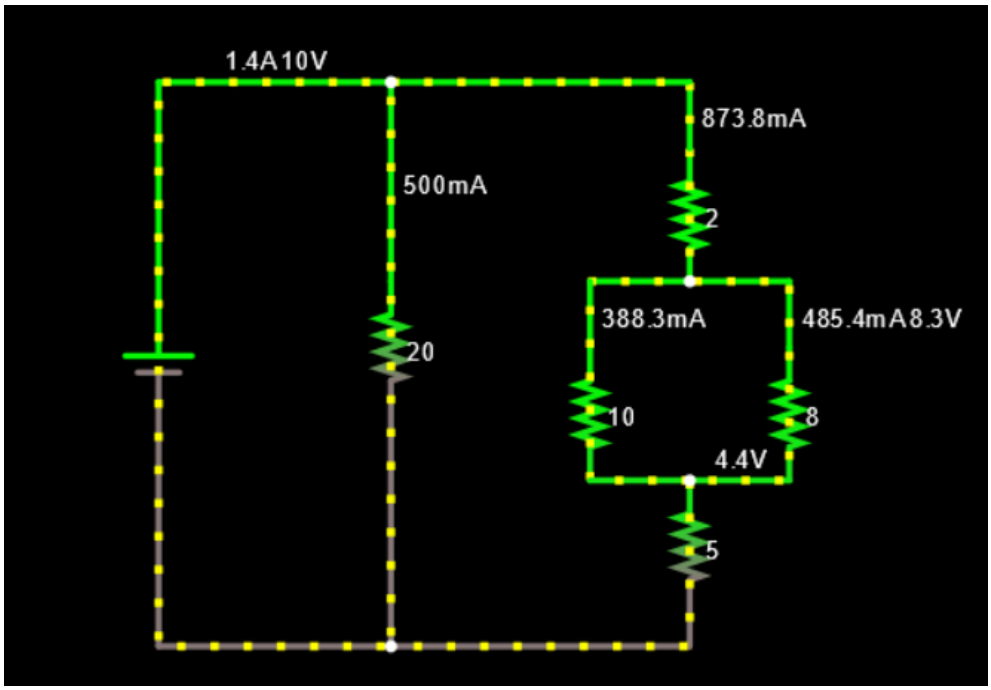

Figura 3.4: Ejemplo de un circuito simulado en el modelo de potencia de **libcpsim**.

```
dss at cpsim-dev-container in ~/dev/CPSim/cpsim/src on develop [?]
❯ ./run
....
Starting CPSim
Debug Mode
Running CPSim in foreground
CONNECT
Voltaje Total = 10 V, Corriente total = 1.37379 A.
r20: Voltaje = 10 V, Corriente = 0.5 A.
r2: Voltaje = 1.74757 V, Corriente = 0.873786 A.
r10: Voltaje = 3.8835 V, Corriente = 0.38835 A.
r8: Voltaje = 3.8835 V, Corriente = 0.485437 A.
r5: Voltaje = 4.36893 V, Corriente = 0.873786 A.
```

Figura 3.5: Validación del algoritmo en una versión temprana de **CPSim**.

Se considera que el circuito probado es lo suficientemente complejo como para validar el correcto funcionamiento del algoritmo.

Si bien las cargas simuladas son todas resistivas, se desarrollaron clases para los tres tipos de cargas que suele incorporar el grupo en sus simuladores: cargas de resistencia constante, cargas de corriente constante, y cargas de potencia constante. El modelo utilizado para las cargas de corriente constante y potencia constante es tal, que la carga modifica su valor de resistencia para mantener constante el parámetro de interés. Vale aclarar que las cargas de corriente y potencia constante no necesariamente tienen su parámetro fijo a lo largo del tiempo, a pesar de la semántica. Por ejemplo, un EPC se puede modelar como una carga de potencia constante, donde en cada paso de tiempo, la

potencia que consume depende de la potencia de señal de RF que tiene a la entrada el TWT al que alimenta.

En la figura 3.6 se muestra el diagrama UML de las clases que componen el modelo de potencia.

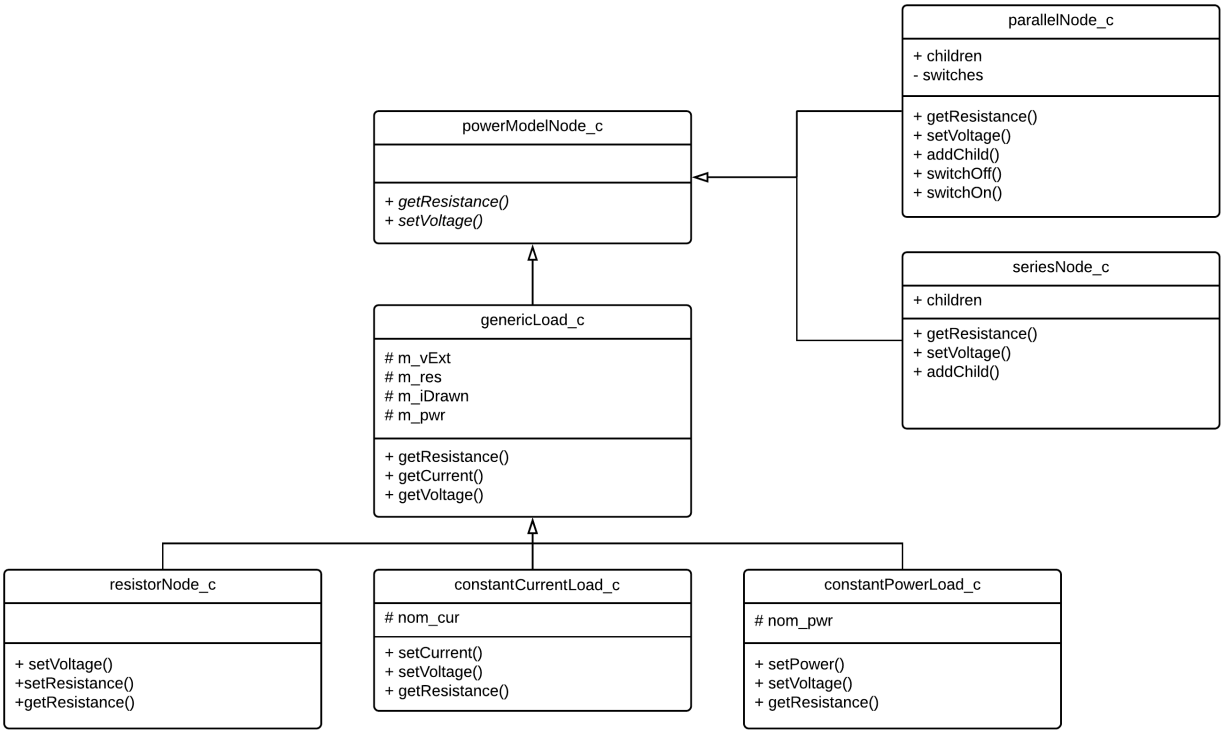

Figura 3.6: Diagrama UML de las clases que forman el modelo de potencia.

Un nodo del modelo puede ser una carga, serie, o paralelo. En el primer caso no puede tener hijos, mientras que en los últimos dos debe hacerlo forzosamente. Todos los nodos tienen los métodos **setVoltage** y **getResistance**, y el comportamiento de estas funcionalidades dependerá de la clase. Todas las cargas tienen valores de resistencia, tensión, corriente, y potencia y además las cargas de potencia o corriente constante tienen un campo para el valor nominal del parámetro de interés. Esto es necesario, porque en caso de recibir tensión nula (por ejemplo si está abierto el circuito al cual pertenece el componente), el parámetro efectivo, sea potencia o corriente, será nulo.

3.3. Subsistema de señales de RF (payloadModel)

En **libcpsim**, las señales se modelan de espectro plano. Se eligió esta representación debido a su simplicidad, puesto que no es un requerimiento simular el contenido de las señales transmitidas entre estaciones terrenas. Sólo se necesita conocer la potencia de la señal para calcular cuánto se calientan los tubos amplificadores. Los únicos parámetros

tenidos en cuenta son la potencia, y las frecuencias de corte superior e inferior.

Si bien la implementación actual de la librería sólo tiene en cuenta la frecuencia central y potencia de las señales simuladas; agregar un tercer parámetro no resulta dificultoso, y permite proveer mayor funcionalidad en el futuro.

Todos los componentes que amplifican, atenúan, o modifican la frecuencia de las señales de RF son de uno de dos tipos, llamados **singleSignal** y **multiSignal**. Como el nombre lo indica, los **singleSignal** son componentes que a su entrada tienen una sola señal de RF, y los **multiSignal**, son componentes que a su entrada tienen un conjunto de señales de RF, representado por un vector STL en la implementación. El hacer dicha distinción permite que no se requieran modelar filtros, cómo se verá mas adelante. Se optó por esta solución, dado que resultaba demasiado complejo diseñar una algoritmia totalmente genérica que use filtros, y es probable que para desarrollar un modulo de DSS con **libcpsim**, haya que diseñarla y cambiar el tratamiento de las señales de RF a nivel componente.

Todos los componentes del tipo **singleSignal** poseen un valor llamado **gain** y un método virtual llamado **computeGain** que establece su valor; y todos los componentes de tipo **multiSignal** poseen un vector de ganancias (del mismo tamaño que el vector de señales) llamado **gainVector**, y un método virtual llamado **computeGainVector** que establece sus valores. Esto es debido a que las clases de componentes base fueron diseñadas con la idea de que a futuro puedan derivar en clases más complejas que reescriban parte del comportamiento, pero de forma que se mantenga la compatibilidad con los métodos de los modelos del simulador. Por ejemplo, si se hace una clase para un TWT específico que hereda de la que existe en **libcpsim**, se puede complejizar el método **computeGain** tanto como se desee y el simulador en el que se utilice seguirá funcionando como si se hubiera usado la clase base sin necesidad de escribir código extra.

Además, todos los componentes de tipo **singleSignal** y **multiSignal** poseen una función llamada **transferFunction**, que es de tipo void y no toma argumentos. Cuando se invoca a la **transferFunction** de un componente, se le hace alguna modificación a la señal (o vector de señales) que tiene a la entrada, cuyo resultado se guarda en la señal (o vector) de entrada del componente que le sigue en la cadena.

En la figura 3.7 se muestra un diagrama UML de las clases de componente diseñadas en la librería.

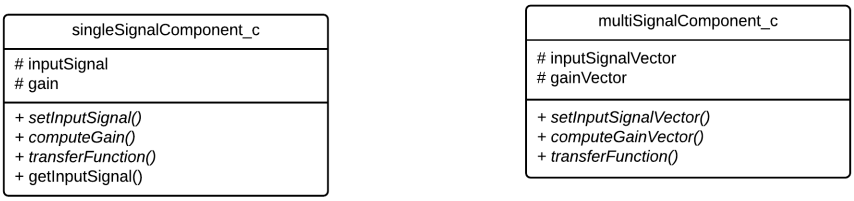

Figura 3.7: Diagrama UML de las clases singleSignal y multiSignal.

3.4. Entry points

Para actualizar las distintas variables de simulación, se utilizan entry points. Cada entry point es un puntero a un método sin argumento ni valor de retorno, el cual se asigna al Scheduler para ser llamado con la frecuencia deseada. En la práctica, dicha frecuencia suele ser de no mas de algunas veces por segundo. Para los simuladores hechos con **libcpsim**, los entry points corresponden a métodos de los tres subsistemas.

Cómo se mencionó anteriormente, el entry point de **thermalManager** itera sobre todos los nodos del modelo térmico, usando el método RK4 para actualizar las temperaturas dadas las disipaciones.

El entry point de **powerModel** le establece al nodo raíz del modelo de potencias el valor de tensión de bus por medio de su método **setVoltage**, y llama a su método **getResistance** para obtener la resistencia equivalente del circuito. Con la resistencia equivalente, se pueden calcular los consumos de corriente y potencia. Además, el método **setVoltage** se propaga a lo largo del árbol que representa el modelo de potencias, estableciendo todos los valores de tensión, corriente, potencia, y/o resistencia, según corresponda. Esto es debido a que para los nodos paralelo y serie, **setVoltage** invoca al **setVoltage** de sus hijos.

Por último, **payloadModel** posee dos entry points: **signalPropagationEP** y **powerConsumptionEP**. El entry point **signalPropagationEP** establece las señales que tiene la carga útil a la entrada, e invoca a la función de transferencia del receptor. Por como está programada la lógica interna de las funciones de transferencia, la función de transferencia de cada componente de la carga útil invoca la función de transferencia del componente que le sigue. Para evitar errores de sincronización, la propagación de invocaciones se frena en el OMUX. Luego de que la función de transferencia del receptor retorna, se invoca la función de transferencia del OMUX. Finalmente, se copia el vector de señales de salida del OMUX al vector de señales de salida de **payloadModel**. El entry point **powerConsumptionEP** itera sobre todos los HPA, invocando su método **updateEPCConsumption**.

Como los entry points son llamados con una determinada frecuencia, es importante asegurarse que los mismos no sean bloqueantes, es decir, que se ejecuten lo suficientemente rápido como para no frenar las demás rutinas que componen a un simulador. A tal fin, se hicieron pruebas del entry point del modelo de potencia (cuyo nombre técnicamente es **EP**, pero para evitar confusiones será llamado **powerEP**), y el entry point **signalPropagationEP** de **payloadModel**, para medir cuánto tardan en ejecutarse en función de cuantos TWT están siendo simulados. La cantidad de TWT usados en las pruebas varió entre 2 y 10. Los resultados de las pruebas se muestran en la figura 3.8. La computadora utilizada para las pruebas es una provista por Thales para correr el simulador de operaciones de ARSAT-1, que tiene un CPU Intel Xeon E5640 de 4 nucleos a 2.67 GHz, un

GPU Nvidia Quadro 2000, 8 GB de RAM DDR3 a 1.33 GHz, y su sistema operativo es Ubuntu 14.04 LTS.

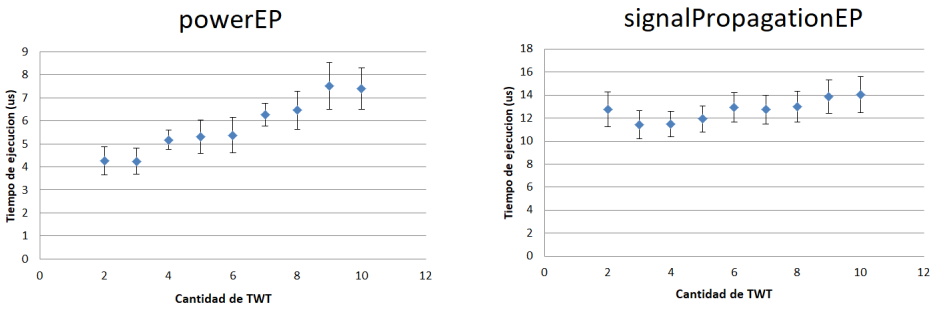

Figura 3.8: Tiempo de ejecución de los entry points en función de la cantidad de amplificadores.

Se puede ver que para las cantidades de HPA probadas, los tiempos de ejecución rondan entre 4 y 14 microsegundos, con lo que permiten invocar a los entry points una frecuencia más que suficiente, dado que en la práctica, no se espera que se ejecuten mas de algunas veces por segundo.

Capítulo 4

Clases de componente

La librería posee clases para los distintos componentes que suelen encontrarse en una carga útil. Estas clases tienen los atributos y métodos necesarios para simular el comportamiento característico de los componentes que representan. En este capítulo se describirán las clases de componente desarrolladas en **libcpsim**, cuyo diagrama UML se muestra en la figura 4.1.

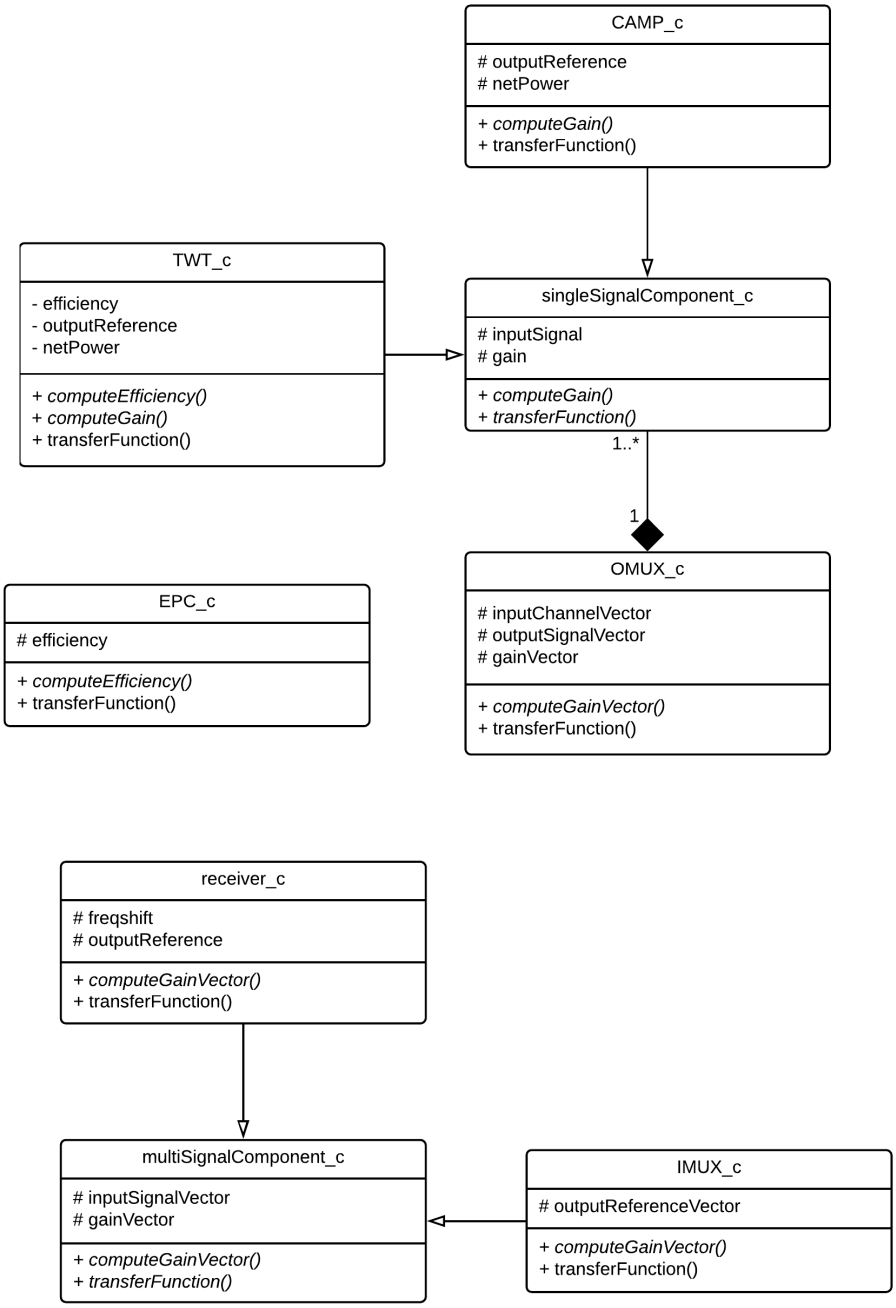

Figura 4.1: Diagrama UML de las clases de componente desarrolladas en **libcpsim**.

Cómo los EPC no interactúan de forma directa con las señales, no necesitan heredar de **singleSignal** ni **multiSignal**. Por otro lado, la clase OMUX posee un vector de elementos de tipo **singleSignal** que representan los distintos canales de entrada, en vez de heredar directamente de esa clase.

Si bien no se muestra en el diagrama UML, todas las clases de componente heredan de una clase común llamada **payloadComponent**, que existe para poder aprovechar el

polimorfismo. En otras palabras, si se hace un vector de punteros a elementos de **payloadComponent**, se pueden agregar punteros a todos los componentes, independientemente de su tipo. Además de ciertas funciones requeridas por SMP, la clase **payloadComponent** posee únicamente el método virtual puro **transferFunction**.

Los únicos componentes que son cargas en el modelo de potencia son los EPC. Técnicamente, los receptores deberían serlo también, ya que internamente poseen un LNA, pero se consideró que el consumo es despreciable y se decidió no agregarlos al modelo de potencias. Si se quisiera hacer, solo se requiere cambiar algunas líneas de código. Además, no todos los modelos poseen nodos del modelo térmico, ya que los modelos térmicos de los componentes replican a los usados en el simulador de ARSAT-2 y en ése modelo sólo los EPC, CAMP, TWT, y OMUX tienen nodos. La única diferencia es que la implementación actual de las clases de componente contempla un sólo nodo por componente (aunque las funciones que agregan los nodos al modelo térmico toman un vector, que de momento siempre es de un único elemento). Esto se debe a que cada nodo tiene asignado un número como identificador único, y se requiere una lógica de nivel intermedio que agregue al modelo todos los nodos de componente (cada uno con un número que esté libre) junto con las relaciones de conductividad entre ellos. La dificultad de diseñar esta lógica radica en que para agregar las relaciones de conductividad de los nodos al modelo térmico se necesitan los identificadores de cada nodo y no se saben a priori.

Por último, en todo el código se asume que todos los valores están expresados en unidades del Sistema Internacional, por lo tanto las potencias se asumen en Watts, las frecuencias en Hertz, etc.

4.1. Receptor

El receptor es un componente **multiSignal**. Como valor de referencia, **computeGainVector** establece todas las ganancias en 1,5. No se tienen en cuenta efectos de distorsión, ya que el LNA del receptor siempre se usa en la zona lineal [1]. La función de transferencia, cuyo código se muestra en la figura 4.2, llama a **computeGainVector**, y llena un vector con las señales que tiene a la entrada, multiplicadas por la ganancia correspondiente y corridas en frecuencia. Luego, le asigna este vector a la entrada del elemento siguiente, y llama a su función de transferencia.

```
void receiver_c::transferFunction(){
    int size = inputSignalVector.size();
    std::vector<signal_c> outputSignalVector(size);
    computeGainVector();
    for (int i = 0; i < size; i++){
        outputSignalVector[i].setFreqHigh((inputSignalVector[i].getFreqHigh())+freqShift);
        outputSignalVector[i].setFreqLow((inputSignalVector[i].getFreqLow())+freqShift);
        outputSignalVector[i].setPower((inputSignalVector[i].getPower())*gainVector[i]);
    }
    outputReference->setInputSignalVector(std::move(outputSignalVector));
    outputReference->transferFunction();
}
```

Figura 4.2: Función de transferencia de la clase receptor.

4.2. IMUX

El multiplexor de entrada es un componente **multiSignal** que tiene un vector de punteros a componentes de salida llamado **outputReferenceVector**. Como valor de referencia, **computeGainVector** establece todas las ganancias en 1 (supone que no hay pérdidas). La función de transferencia, cuyo código se muestra en la figura 4.3, llama a **computeGainVector**, y a cada elemento del vector de señales que tiene a la entrada, lo multiplica por su ganancia correspondiente, lo asigna a la entrada del elemento correspondiente de **outputReferenceVector**, e invoca la función de transferencia de dicho elemento.

```
void IMUX_c::transferFunction(){
    int size = inputSignalVector.size();
    computeGainVector();
    signal_c outputSignal;
    for ( int i = 0; i<size ; i++ )
    {
        outputSignal = inputSignalVector[i];
        outputSignal.setPower(outputSignal.getPower()*gainVector[i]);
        outputReferenceVector[i]->setInputSignal(outputSignal);
        outputReferenceVector[i]->transferFunction();
    }
}
```

Figura 4.3: Función de transferencia de la clase IMUX.

4.3. CAMP

El amplificador de canal es un componente **singleSignal**. El método **computeGain** de esta clase ajusta la ganancia de tal forma que el TWT que le sigue en la cadena tenga 97,5 W de potencia a la salida. Esto se debe a que el modelo térmico del simulador de ARSAT-2 especifica una potencia total consumida de 150 W para los TWT de banda Ku, con una eficiencia del 65 %. Por otro lado, su disipación y consumo están fijos en 3,3 W, datos sacados también del modelo térmico del simulador de ARSAT-2. Esta disipación probablemente sea debida a la electrónica digital interna del CAMP. La función

de transferencia, cuyo código se muestra en la figura 4.4, llama a **computeGain**, multiplica la señal que tiene a la entrada por la ganancia calculada, la asigna a la entrada del componente que le sigue en la cadena, e invoca la función de transferencia de dicho componente.

```
void CAMP_c::transferFunction(){
    double inputPower, outputPower;
    signal_c outputSignal = inputSignal;
    computeGain();
    inputPower = inputSignal.getPower();
    outputPower = inputPower*gain;
    outputSignal.setPower(outputPower);
    netPower = 3.3;
    dissipation = 3.3;
    outputReference->setInputSignal(outputSignal);
    outputReference->transferFunction();
}
```

Figura 4.4: Función de transferencia de la clase CAMP.

4.4. EPC

El EPC no interactúa directamente con la señal de RF, así que no tiene función de transferencia[1]. Es el único componente que es parte del modelo de potencia y está modelado como una carga de potencia constante. La potencia que consume es la suma de las potencias que consumen los componentes que alimenta, dividido por su eficiencia. La misma se fijó en 94 %, valor típico según [1]. En la figura 4.5, se muestra a modo de ejemplo el diagrama UML de la herencia de clases para el EPC, ya que es el único componente que forma parte de los tres subsistemas del simulador.

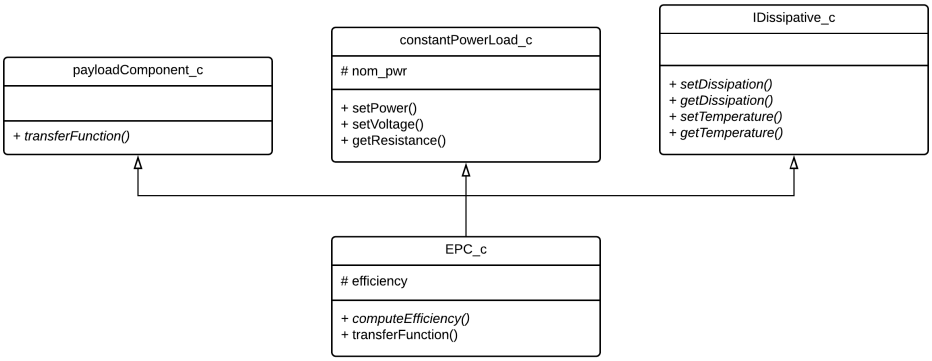

Figura 4.5: Diagrama UML de la herencia de clases para el EPC.

Por simplicidad, se muestran los métodos más importantes. **IDissipative** es una clase perteneciente a **libthermalcore**, que representa los componentes que disipan calor.

[1] En la implementación, tiene un método **transferFunction** que no hace nada.

4.5. TWT

El TWT es un componente **singleSignal**. En la figura 4.6 se muestra una curva de potencia de salida vs potencia de entrada típica para un TWT, encontrada [4]. El método **computeGain** calcula la ganancia usando una versión linealizada de dicha curva, que se muestra en la figura 4.7. Su eficiencia se fijó en 65 %, dado que es el valor especificado en el archivo de configuración del modelo térmico del simulador de ARSAT-2 para los TWT de banda Ku.

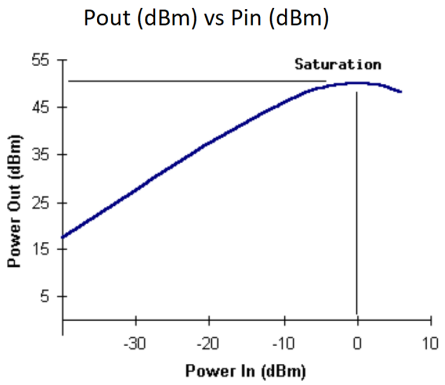

Figura 4.6: Curva de potencia de salida vs potencia de entrada típica de un TWT [4].

Como no se tienen datos reales sobre la ganancia de los TWT, el método **computeGain** no toma en cuenta ni la temperatura ni la frecuencia de las señales amplificadas. Debido a esto, un TWT simulado que pertenece a esta clase amplifica de igual manera señales de 100 Hz como de 1 THz, sin importar si el modelo térmico reporta temperaturas cercanas al cero absoluto o a las de la superficie del sol. Por supuesto, esto no es realista, pero todas las clases fueron diseñadas de forma que el método **computeGain** (o **computeGainVector**) pueda ser redefinido en una clase derivada, de forma de hacer uso de todas las variables que el diseñador de modelo considere pertinentes para el cálculo de ganancia.

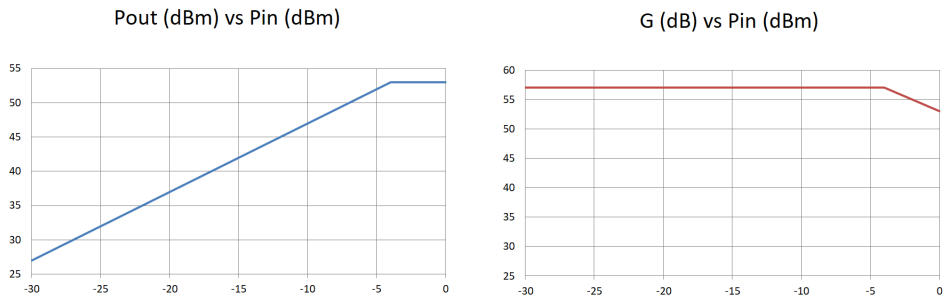

Figura 4.7: Curvas de potencia de salida y ganancia vs potencia de entrada usadas en la clase TWT. Ambas curvas son equivalentes.

La función de transferencia, cuyo código se muestra en la figura 4.8, llama a **computeGain** y **computeEfficiency** (que si bien en esta clase siempre asigna la eficiencia en 0.65, se puede redefinir en una clase derivada de la misma forma que **computeGain**), multiplica la señal que tiene a la entrada por la ganancia calculada, la asigna a la entrada del componente que le sigue en la cadena, e invoca la función de transferencia de dicho componente. Además, calcula cuanta potencia va a consumir del EPC que tiene asociado, y cuánta va a ser su disipación.

```
void TWT_c::transferFunction(){
    double inputPower, outputPower;
    signal_c outputSignal = inputSignal;
    computeGain();
    computeEfficiency();
    inputPower = inputSignal.getPower();
    outputPower = inputPower*gain;
    outputSignal.setPower(outputPower);
    netPower = (outputPower - inputPower)/efficiency;
    dissipation = netPower * (1-efficiency);
    outputReference->setInputSignal(outputSignal);
    outputReference->transferFunction();
}
```

Figura 4.8: Función de transferencia de la clase TWT.

Si bien la función de transferencia de los canales de entrada del OMUX (que en la carga útil de prueba es el siguiente componente en la cadena) no hace nada, se invoca a la función de transferencia del componente siguiente para que se pueda introducir un componente extra entre el TWT y el OMUX (como por ejemplo un switch de RF) sin romper la funcionalidad del simulador.

4.6. Etapas de amplificación

Como existen distintas configuraciones posibles de HPA (algunas de las cuales se mostraron en la figura 2.4), se diseñó una clase abstracta de la cual deben heredar las distintas clases de HPA usadas. Esta clase base está compuesta por tres métodos virtuales puros[2]: **pushToDissipativeVector**, que agrega los componentes del HPA al modelo térmico; **pushToLoadVector**, que agrega los componentes del HPA al modelo de potencia; y **updateEPCConsumption**, que calcula cuanto consume y cuanto disipa/disipan el/los EPC del subsistema.

Con esta clase base, se diseñaron 2 clases para HPA, llamadas HPA genérico 1 y 2. Se muestra en la figura 4.9 el diagrama de clases correspondiente a los HPA diseñados.

[2]Los métodos virtuales puros son métodos que no están definidos, y deben definirse en las clases derivadas.

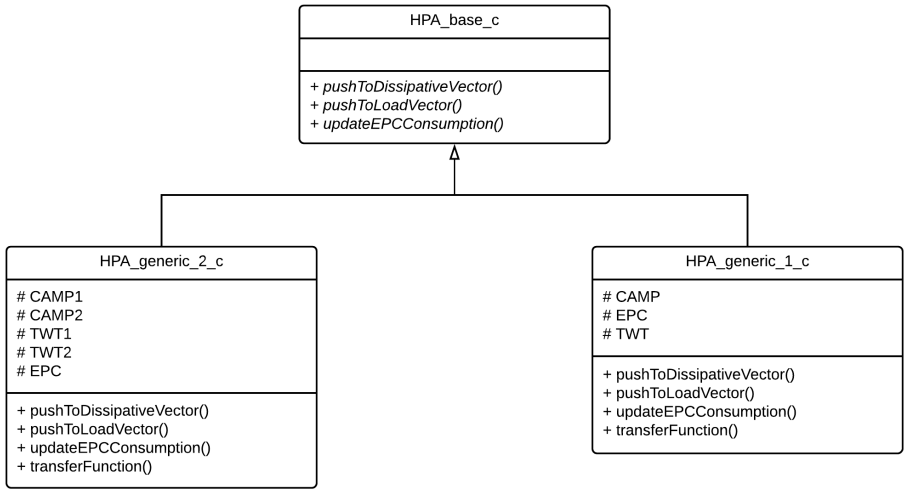

Figura 4.9: Diagrama UML de los subsistemas HPA desarrollados.

La clase **HPA_generic_1** fue creada para verificar la integración de las distintas partes que componen la librería, pero no se terminó utilizando en la versión final de **CPSim**. Consiste de un CAMP y un TWT alimentados por un EPC. Por otro lado, la clase **HPA_generic_2** es producto de un intento de recrear los TWTA duales de banda Ku que existen en ARSAT-2. Está compuesta por un EPC, que alimenta a dos CAMPs y dos TWTs, que amplifican dos señales distintas.

4.7. OMUX

El multiplexor de salida tiene un vector de **singleSignals** que representa a los canales de entrada. Por simplicidad, la ganancia de cada uno de los elementos de dicho vector está fija en 1, y la clase posee un atributo **gainVector** y un método **computeGainVector** como si fuera de tipo **multiSignal**. Como valor de referencia, **computeGainVector** establece todas las ganancias en 1 (se asume que el OMUX no tiene pérdidas). La función de transferencia multiplica cada señal de los canales de entrada por su respectiva ganancia, y la agrega al vector de señales de salida.

```
void OMUX_c::transferFunction(){

    int size = inputChannelVector.size();
    signal_c aux;
    computeGainVector();
    for(int i = 0 ; i < size ; i++)
    {
        aux = inputChannelVector[i].getInputSignal();
        aux.setPower(aux.getPower()*gainVector[i]);
        outputSignalVector[i] = aux;
    }

}
```

Figura 4.10: Función de transferencia de la clase OMUX.

Capítulo 5

Prueba de concepto

5.1. Carga útil de prueba

Habiendo desarrollado las clases y los métodos para integrar una carga útil, se decidió hacer una simulación de una versión simplificada de la de ARSAT-2. La carga útil simulada consiste de una cadena de amplificación completa, con un sólo TWTA dual, modelado en base a los de banda Ku presentes en ARSAT-2, montado sobre la cara norte y respetando las conductividades térmicas del simulador original. A fin de representar el sistema de control térmico, todos los nodos del honeycomb se unieron con una cierta conductividad a un nodo extra, que está fijo a 50 °C, cómo se muestra en la figura 5.1.

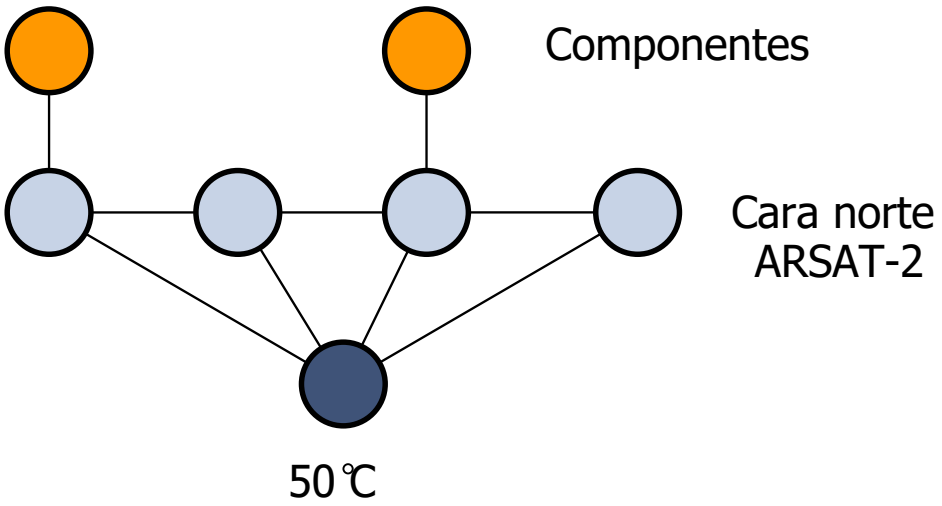

Figura 5.1: Representación simplificada del modelo térmico de la carga útil de prueba.

Además del TWTA, la cadena simulada contiene un receptor, IMUX, y OMUX, instanciados en base a las clases descriptas en el capítulo anterior. Se muestra un diagrama de la carga útil simulado en la figura 5.2.

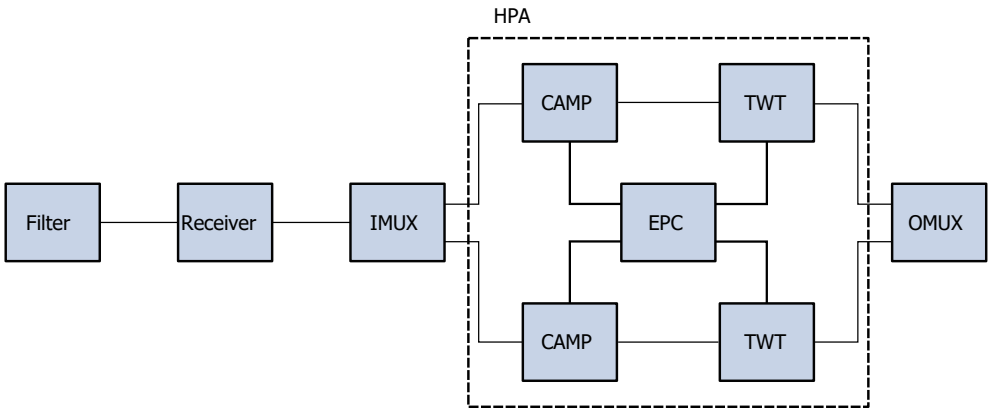

Figura 5.2: Diagrama en bloques de los componentes de la carga útil simulada.

El vector de señales de entrada está compuesto por dos señales: una de 14 GHz con una potencia de -75 dBm, y una de 14.2 GHz con una potencia de -85 dBm. Estos son valores típicos para las señales de banda Ku que llegan a un satélite geoestacionario.

5.2. Integración programática

Cuando se crea el simulador, se llama a un método llamado **CreateModels**. Dentro de este método es que se crean e integran todos los componentes a ser simulados. En la figura 5.3 se presenta un diagrama en bloques del mismo.

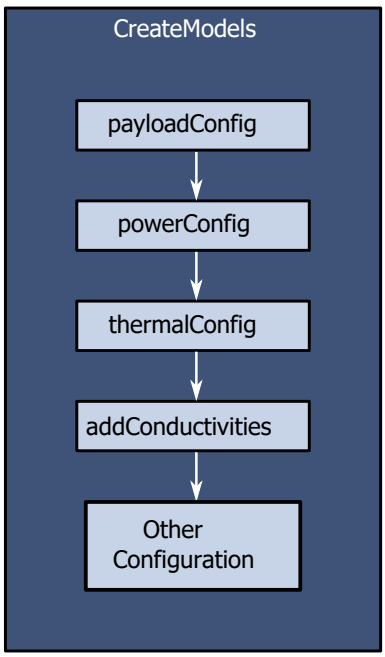

Figura 5.3: Diagrama en bloques del método **CreateModels**.

CreateModels está compuesto de los métodos **payloadConfig**, **powerConfig**, **thermalConfig**, **addConductivities**, y los métodos de configuración de bajo nivel que no

están en ninguno de los mencionados. Se muestra el código de **CreateModels** en la figura 5.4.

```
void CPSim_c::CreateModels()
{

    //SetSpeedFactor(100);

    printEP = std::make_unique<Smp::Mdk::EntryPoint>("CPSim_print_entry_point", "CPSim Print Entry Point Object", \
            this, &::CPSim::CPSim_c::print);
    GetScheduler()->AddSimulationTimeEvent(printEP.get(), 1.0 * SECS, 1 * SECS, -1);

    int channels = 2;
    payloadConfig(channels);
    powerConfig();
    thermalConfig();
    addConductivities();

    std::map <uint32_t, INVAP::THERMALCORE::ThermalNodeClass_c> & nodesMap =
            thermalManager->getCurrentThermalModel().getNodes();


    receiver->setFreqShift(-2e9);
    payloadModel->setInputSignal(14e9, 0.1e9, 3.1622e-11, 0); // freqLow = 13.95 GHz, freqHigh = 14.05 GHz, -75 dBm
    payloadModel->setInputSignal(14.2e9, 0.1e9, 3.1622e-12, 1); // freqLow = 14.15 GHz, freqHigh = 14.25 GHz,  -85 dBm
    powerModel->setVoltage(50);
    std::default_random_engine generator;
    std::normal_distribution<double> dist(323.15,.5);   // add a bit of randomness to the initial temperature distribution
    for (auto const& elem : nodesMap){
            nodesMap[elem.first].temperature = dist(generator);
        }
    nodesMap[1].temperature = 323.15;
    thermalManager->getCurrentThermalModel().lockNodeTemperature(1, 323.15);
}
```

Figura 5.4: Código del método **CreateModels**.

La primera linea, que está comentada, configura el factor de velocidad de simulación en 100. Esto es útil para ver las temperaturas alcanzadas en el estacionario sin tener que esperar. Luego, se crea un entry point para imprimir cierta información en la consola, y se le pasan al scheduler las condiciones de su invocación. Se configura el corrimiento de frecuencia del receptor en -2 GHz, que es un valor típico para comunicaciones satelitales en banda Ku, se crean y asignan las señales de entrada de la carga útil, y se establece la tensión de bus en 50 V, que es el valor nominal en ARSAT-2. Por último, se asignan valores aleatorios a las temperaturas de todos los nodos del modelo térmico con una distribución gaussiana cuya media es 50 °C y cuya desviación estándar es 0.5 °C , y se fija el valor del nodo que representa el sistema de control térmico en 50 °C.

En la figura 5.5, se muestra el código del método **payloadConfig**.

```
void CPSim_c::payloadConfig(int channels){
    payloadModel = new payloadModel_c("PL_PayloadModel", "Payload RF signal model", this, channels);
    AddModel(payloadModel);
    receiver = new receiver_c("Receiver", "Payload receiver", payloadModel, channels);
    AddModel(receiver);
    IMUX = new IMUX_c("IMUX", "Payload Input Multiplexer", payloadModel, channels);
    AddModel(IMUX);
    for (int i = 0; i<channels/2; i++)
    {
        HPAG2Vector.push_back(new HPA_generic_2_c(std::string("HPA" + std::to_string(i)).c_str(),
        std::string("Payload HPA number " + std::to_string(i)).c_str(), payloadModel));
        AddModel(HPAG2Vector[i]);
        AddModel(HPAG2Vector[i]->getEPC());
        AddModel(HPAG2Vector[i]->getTWT1());
        AddModel(HPAG2Vector[i]->getTWT2());
        AddModel(HPAG2Vector[i]->getCAMP1());
        AddModel(HPAG2Vector[i]->getCAMP2());
    }
    OMUX = new OMUX_c("OMUX", "Payload Output Multiplexer", payloadModel, channels);
    AddModel(OMUX);

    receiver->setOutputReference(IMUX);
    for (int i = 0; i<channels; i = i + 2)
    {
        IMUX->setOutputReference(HPAG2Vector[i/2]->getCAMP1(),i);
        IMUX->setOutputReference(HPAG2Vector[i/2]->getCAMP2(),i+1);
        HPAG2Vector[i/2]->setCAMP1OutputReference(HPAG2Vector[i/2]->getTWT1());
        HPAG2Vector[i/2]->setCAMP2OutputReference(HPAG2Vector[i/2]->getTWT2());
        HPAG2Vector[i/2]->setTWT1OutputReference(OMUX->getInputReference(i));
        HPAG2Vector[i/2]->setTWT2OutputReference(OMUX->getInputReference(i+1));
    }
    payloadModel->setReceiver(receiver);
    for (int i = 0; i<channels/2; i++){
        payloadModel->addHPA(HPAG2Vector[i]);
    }
    payloadModel->setOMUX(OMUX);

    HPAVector.insert(HPAVector.begin(), HPAG2Vector.begin(), HPAG2Vector.end());

}
```

Figura 5.5: Código del método **payloadConfig**.

Este método toma de argumento la cantidad de canales que va a procesar la carga útil. Esto define unívocamente la cantidad de amplificadores que va a tener la carga útil, cuando se usa un sólo tipo de HPA. El método crea todos los componentes necesarios y los configura para la cantidad de canales indicada. A cada componente, le asigna la referencia de salida que corresponde, e inserta todos los elementos de tipo HPA en un vector llamado **HPAVector**.

En la figura 5.6, se muestra el código del método **powerConfig**.

```
void CPSim_c::powerConfig(){
    powerModel =  new powerModel_c("PL_PowerModel", "Payload power model", this);
    AddModel(powerModel);
    root = std::make_unique<parallelNode_c>();
    powerModel->setRoot(root.get());
    auto loadVector = powerModel->getLoadVector();

    for (auto it = HPAVector.begin(); it!=HPAVector.end(); it++){
        (*it)->pushToLoadVector(loadVector);
    }
    // YOU CAN KEEP ON ADDING COMPONENTS TO THE LOAD VECTOR HERE

    for (auto it = loadVector.begin(); it!=loadVector.end(); it++)
        root->addChild(*it);

}
```

Figura 5.6: Código del método **powerConfig**.

Este método asigna un nodo paralelo como raíz del modelo de potencia, y llena un vector de cargas con los HPA. La lógica está diseñada para que se agreguen mas subcircuitos al vector de cargas en caso de ser necesario. Luego, todos los elementos del vector de cargas se agregan como hijos del nodo raíz.

En la figura 5.7, se muestra el código del método **thermalConfig**.

```
void CPSim_c::thermalConfig(){
    putenv((char*)"THERMAL_DB_PREFIX=/home/dss/dev/CPSim/cpsim/lib/libcpsim/thermalSP.sqlite");
    std::map<std::string, INVAP::THERMALCORE::IDissipative_c *> dissipatives;
    std::map<uint32_t, INVAP::THERMALCORE::ThermalNodeClass_c> newNodes;
    std::map<INVAP::THERMALCORE::HarnessType_e, std::map<std::string,
        std::vector<INVAP::THERMALCORE::ThermalHarnessClass_c>>> harnessMapMap;
    std::map<std::string, std::vector<INVAP::THERMALCORE::ThermalHarnessClass_c>> harnessMap;
    std::vector<INVAP::THERMALCORE::ThermalHarnessClass_c> harnessVector;

    std::vector<customDissipative_c*> customDissipativeVector;
    for (auto it = HPAVector.begin(); it!=HPAVector.end(); it++){
        (*it)->pushToDissipativeVector(customDissipativeVector);
    }
    customDissipativeVector.push_back(OMUX);

    thermalManager = new INVAP::THERMALCORE::ThermalManagerClass_c("PL_ThermalManager", "Payload thermal manager", this);
    INVAP::THERMALCORE::ThermalModelClass_c &model = thermalManager->getCurrentThermalModel();
    INVAP::THERMALCORE::ThermalHarnessClass_c harness;
    harness.setType(INVAP::THERMALCORE::IDISSIPATIVE);
    std::string name;
    int id = 10000;
    for (auto it = customDissipativeVector.begin(); it!=customDissipativeVector.end(); it++){
        name = (*it)->GetDissipativeName();

        std::vector<INVAP::THERMALCORE::ThermalNodeClass_c> &nodeVector = (*it)->getNodeVector();
        for (auto nodeIt = nodeVector.begin(); nodeIt != nodeVector.end(); nodeIt++)
        {
            nodeIt->setId(id);
            newNodes[id] = *nodeIt;
            harness.setNodeId(id++);
            harness.setComponentMnemonic(name);
            harnessVector.push_back(harness);
        }
        dissipatives[name] = *it;
        harnessMap[name] = harnessVector;
        harnessVector.clear();
    }
    model.addDissipativeComponents(dissipatives);
    model.addNodes(newNodes);
    harnessMapMap[INVAP::THERMALCORE::IDISSIPATIVE] = harnessMap;
    model.addHarness(harnessMapMap);
    AddModel(thermalManager);
    thermalManager->setDBInitializer();
    thermalManager->initialize();
    model.checkComponentsConsistency();
    thermalManager->setCurrentDeploymentState(0);
}
```

Figura 5.7: Código del método **thermalConfig**.

Cuando se ejecuta el método **initialize** de la librería térmica, se carga el modelo de la base de datos cuya ruta está en la variable de entorno **THERMAL_DB_PREFIX**, y, para que sea fácil cambiar la ruta cuando sea necesario, la primera línea de **thermalConfig** establece dicha variable de entorno. Luego, se agregan los nodos de cada componente al modelo térmico (incluso antes que los nodos de la plataforma) con el identificador a partir de 10000. En otro método llamado **addConductivities** se deben configurar manualmente las relaciones que conciernen a los nodos agregados programáticamente, ya sea entre ellos o con los nodos de la plataforma de servicio. En la carga útil de ejemplo, se agregaron las relaciones tal cual están descriptas en los archivos de configuración del modelo térmico de ARSAT-2.

5.3. Validación del simulador

Con la carga útil de prueba integrada, se procedió a realizar dos simulaciones, cambiando únicamente los métodos que imprimen en consola. Esto fue necesario, ya que no se pudo utilizar la API desarrollada por el grupo. La primera simulación, cuya salida se muestra en la figura 5.8, se realizó para validar el procesamiento que realiza la carga útil en las señales simuladas, y verificar que las potencias consumidas y disipadas por el EPC y los TWT sean adecuadas.

```
SIGNAL ENTRY POINT
Vector de señales a la entrada:
Señal 0: f = 1.4e+10 Hz, p = 3.1622e-11 W
Señal 1: f = 1.42e+10 Hz, p = 3.1622e-12 W
Vector de señales a la salida:
Señal 0: f = 1.2e+10 Hz, p = 97.5 W
Señal 1: f = 1.22e+10 Hz, p = 97.5 W
POWER ENTRY POINT
Tension de bus: 50 V
consumo EPC: 326.865 W
disipacion EPC: 20.2656 W
disipacion TWT: 52.4999 W
```

Figura 5.8: Validación de las señales a la salida de la carga útil, y de los consumos y disipaciones de potencia.

Las señales son amplificadas a 97,5 W y corridas 2 GHz en frecuencia, como era de esperarse. En el código del simulador de ARSAT-2, la disipación de los EPC de banda Ku es de 22 W, mientras que en la simulación es de 20,26 W. En la figura 5.9 se muestran las temperaturas finales luego de que el sistema llega a los valores estacionarios. El nodo 1 es el que está fijo a 50 °C, los nodos 310 a 331 corresponden a la cara norte de ARSAT-2, el nodo 10000 corresponde al EPC, los nodos 10001 y 10002 corresponden a los CAMP, los nodos 10003 y 10004 corresponden a los TWT, y el nodo 10005 corresponde al OMUX.

```
Node ID:1 Temperature: 323.15      Node ID:324 Temperature: 323.904
Node ID:310 Temperature: 323.15    Node ID:325 Temperature: 346.708
Node ID:311 Temperature: 323.494   Node ID:326 Temperature: 323.898
Node ID:312 Temperature: 323.151   Node ID:327 Temperature: 323.206
Node ID:313 Temperature: 323.163   Node ID:328 Temperature: 323.229
Node ID:314 Temperature: 323.156   Node ID:329 Temperature: 326.241
Node ID:315 Temperature: 323.204   Node ID:330 Temperature: 323.258
Node ID:316 Temperature: 323.175   Node ID:331 Temperature: 323.154
Node ID:317 Temperature: 323.543   Node ID:10000 Temperature: 326.729
Node ID:318 Temperature: 323.266   Node ID:10001 Temperature: 324.891
Node ID:319 Temperature: 324.443   Node ID:10002 Temperature: 324.26
Node ID:320 Temperature: 323.208   Node ID:10003 Temperature: 350.869
Node ID:321 Temperature: 323.897   Node ID:10004 Temperature: 355.03
Node ID:322 Temperature: 323.856   Node ID:10005 Temperature: 323.976
Node ID:323 Temperature: 323.961
```

Figura 5.9: Representación simplificada del modelo térmico de la carga útil de prueba.

Las temperaturas de los TWT en el estado estacionario se estabilizaron entre 80 °C y 85 °C, o sea entre 30 °C y 35 °C sobre la temperatura del panel en el cual están montados,

valores que se encuentran dentro del rango operativo de los TWT.

Capítulo 6

Desarrollo de un simulador completo

Actualmente, las clases de la librería permiten integrar simuladores de carga útil que corren por sí solos, pero que no tienen compatibilidad con los demás proyectos que desarrolla el grupo. Si se quiere darle utilidad al trabajo realizado, será necesario implementar interfaces de comunicación con otros módulos. Este capítulo ofrece una breve descripción de alto nivel de los pasos que deben seguirse para lograr tal finalidad.

6.1. Modelos administrados

Cuando se configuran los modelos de un simulador desde el código fuente, es necesario recompilar cada vez que se quiere introducir un cambio en la configuración. Una alternativa a esto contemplada en SMP es el uso de modelos administrados. La idea consiste en guardar toda la configuración inicial de los modelos en archivos parseables, y cargarla al iniciar el simulador. De esta forma, se pueden realizar cambios en la disposición de los modelos sin alterar el ejecutable. Esto presenta varias ventajas, entre ellas una menor pérdida de tiempo en compilación, y la posibilidad de permitirle a un diseñador realizar modificaciones sin riesgo de introducir bugs de programación. En la figura 6.1 se muestra un ejemplo de cómo podría estar compuesto **thermalManager**, una vez que su modelo sea totalmente administrado.

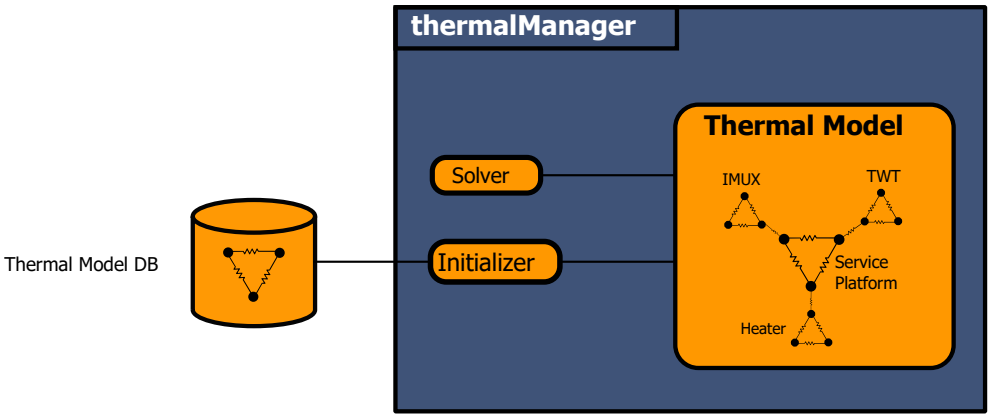

Figura 6.1: Esquema de la composición de **thermalManager**, una vez que el modelo térmico está totalmente contenido en una base de datos.

En este ejemplo, el modelo térmico se encuentra guardado en una base de datos externa, y el método **initializer** del **thermalManager** es el responsable de inicializar el modelo térmico del simulador con el contenido en dicha base de datos. El método **initializer** pertenece a **libthermalcore** y se invoca cuando se ejecuta el simulador.

Actualmente, la funcionalidad de modelos administrados existe (en el sentido que se utilizó herencia de clases apropiada para tal fin) para todos los modelos internos del simulador de carga útil, pero sólo fue utilizada para el modelo térmico del honeycomb sobre el cual están montados los componentes simulados de la carga útil. Además, para **powerModel** y **payloadModel**, no está desarrollado el protocolo según el cual se guardarían los modelos, y por consiguiente, tampoco están desarrollados los métodos que parsean las bases de datos que las guardarían.

6.2. Integración de modelos

Cómo paso intermedio para integrar el simulador de carga útil dentro de DSS, debe definirse la interacción con un simulador de plataforma, o debe hacerse un simulador de satélite con todos los modelos de carga útil y plataforma. Si se opta por mantener separados los simuladores de carga útil y de plataforma, cómo se muestra en la figura 6.2, será necesario definir interfaces entre ellos para el control de la simulación, la actualización de modelos internos, y la sincronización de time keepers.

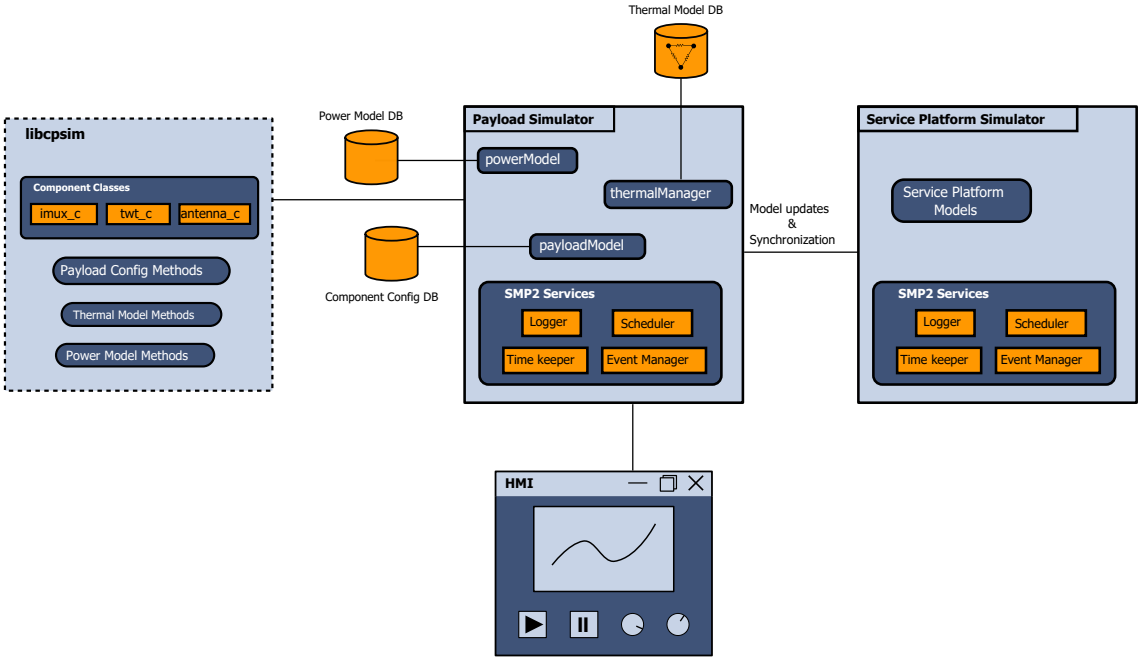

Figura 6.2: Interacción entre simuladores de carga útil y de plataforma.

Como se ha mencionado anteriormente, también se puede optar por hacer un único simulador de satélite que integre los modelos tanto de carga útil, cómo de plataforma; cómo se muestra en la figura 6.3. La ventaja que presentaría esto es que no se tendrían que definir interfaces entre los dos simuladores, y no habría que preocuparse de discrepancias entre los modelos internos ni desincronización de time keepers.

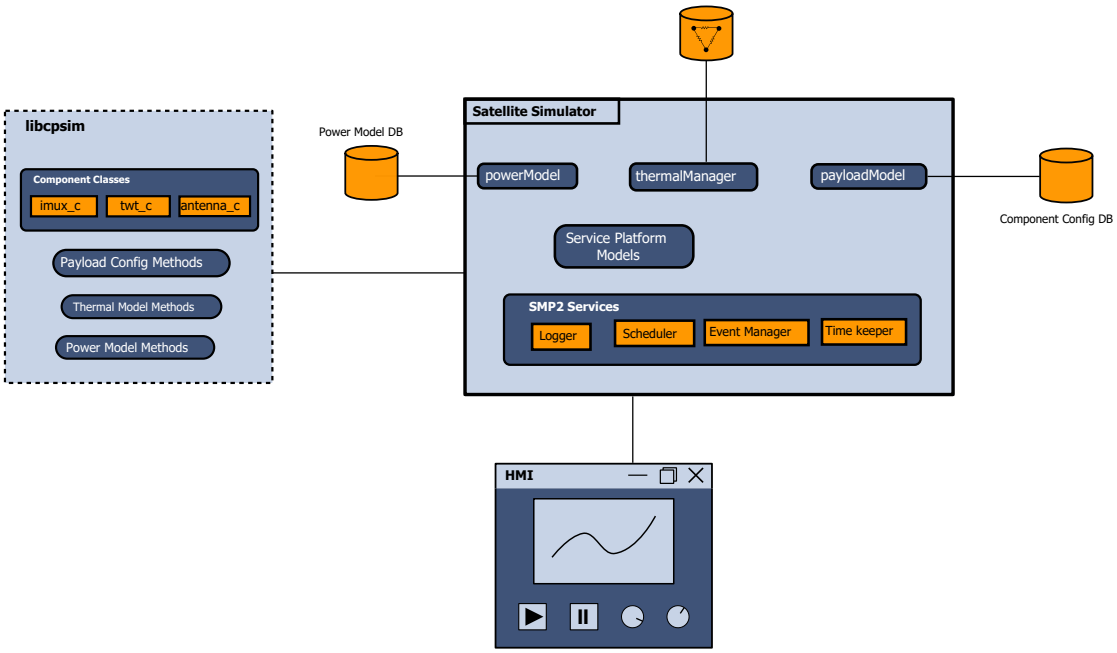

Figura 6.3: Simulador de satélite completo.

Independientemente del método de integración de plataforma de servicios que se elija,

será conveniente que los modelos internos se guarden en bases de datos externas que sean cargadas a la hora de ejecutar el simulador. Además, debe definirse una HMI para que se puedan consultar y establecer las variables de simulación, ya sea por medio de una interfaz gráfica o línea de comandos. La HMI también debe proveer las herramientas para controlar la simulación, por ejemplo para pausar, despausar, y cargar o guardar el estado de la misma.

Finalmente, se debe poder integrar el simulador de satélite en un simulador de operaciones comercial, para lo cual hay que definir y desarrollar interfaces de comunicación y control de la simulación entre los módulos que corresponden al satélite, y los demás módulos del simulador de operaciones.

Capítulo 7

Conclusiones

Se diseñó una librería en C++ que permite integrar simuladores de carga útil *standalone* que obedecen SMP2. Los simuladores desarrollados con dicha librería tienen tres grandes subsistemas: **thermalManager** que se encarga de calcular la evolución temporal de las temperaturas de los componentes simulados y cuya funcionalidad había sido desarrollada anteriormente por el grupo de Segmento Terreno, **powerModel** que se encarga de calcular los consumos y disipaciones de potencia en cada paso de tiempo, y **payloadModel**, que se encarga de calcular la propagación de las señales de RF a lo largo de la cadena de amplificación. Cada subsistema actualiza sus modelos por medio de entry points, que son invocados regularmente a lo largo de la simulación. Se diseñaron clases para los componentes que componen una carga útil, las cuales poseen un método **transferFunction** que simula el efecto del componente en la señal. Con la librería desarrollada, se integró una carga útil de prueba basado en el de ARSAT-2, y se verificó que su comportamiento no sea anómalo.

Si se desea continuar con el proyecto, se deben definir los protocolos de comunicación entre el simulador de carga útil y el simulador de plataforma (o incluir los modelos de plataforma y carga útil en un sólo simulador), además de programar un tratamiento de señal de RF más general. También sería provechoso que los tres subsistemas sean modelos administrados, de forma que se puedan guardar en bases de datos independientes del código fuente del simulador.

Bibliografía

[1] Braun, Teresa. Satellite Communications Payload and System. Wiley, 2012.

[2] Maral, Gérard y Bousquet, Michel. Satellite Communications Systems. Wiley, 2002.

[3] SMP 2.0 Handbook. EGOS-SIM-GEN-TN-0099, Issue 1.2, 28-Oct-2005.

[4] Langton, Charan. All About Traveling Wave Tube Amplifiers (TWTA). http://complextoreal.com/wp-content/uploads/2013/01/twta.pdf