NMR_CORE: Un modelo de datos para representar experimentos de resonancia magnética nuclear de baja resolución

Trabajo Especial de Licenciatura en Ciencias de la Computación

20 de diciembre de 2013

Autor: Federico David Polacov

Directores:

Dr. Daniel Fridlender

Dr. Lucas Cerioni

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
Argentina

Resumen

En respuesta a los requerimientos actuales de la industria de la resonancia magnética nuclear (RMN) de baja resolución, diseñamos un modelo de datos, al que llamamos NMR_CORE, que permite la representación de los experimentos más utilizados en esta técnica en un entorno orientado a objetos utilizando los estándares de modelado de datos UML y OCL.

Debido a la falta de antecedentes en materia de modelado de datos sobre esta técnica, para la definición de los requerimientos de NMR_CORE se analizaron aplicaciones existentes en la industria de la RMN haciendo foco en los modelos intrínsecos en los cuales se basan las mismas.

Con este trabajo pretendemos alcanzar a lectores tanto de la disciplina de la física como de la computación buscando, en medida de lo posible, unificar los conceptos básicos de la definición y representación de experimentos de RMN de baja resolución en ambas disciplinas.

Clasificación: *H.2.1 – Logical Design*
Palabras Clave: Modelo de datos, UML, OCL, TD-NMR, RMN.

Agradecimientos

Quisiera agradecer a la empresa Spinlock, en especial a sus directivos, que siempre me apoyaron y pusieron todo lo que requerí a mi alcance para que yo lograra realizar este trabajo en tiempo y forma.

Muchísimas gracias a Lucas por su infinita paciencia y predisposición desde el comienzo de este trabajo, por haberme guiado en todo el proceso motivándome siempre y por todo el tiempo y empeño dedicado.

Gracias a la Lic. Carolina Dania por su gran apoyo, paciencia y predisposición para ayudarme, orientarme, enseñarme y corregirme en el proceso de escritura de este trabajo.

Por último quiero agradecer a mi familia, en especial a mis padres y hermana, por todo su amor, apoyo y ayuda.

Tabla de contenido

1 Introducción

Actualmente, la resonancia magnética nuclear (RMN) de baja resolución se encuentra en plena etapa de evolución y crecimiento, por lo cual, como cualquier tecnología en desarrollo, requiere de la asistencia del software para mejorar, simplificar y optimizar los procesos que se involucran.

En este trabajo se presenta el diseño de un modelo de datos para representar experimentos de RMN de baja resolución.

1.1 Problema existente y objetivo

El presente trabajo es realizado en colaboración con la empresa SPINLOCK S.R.L., a la cual nos referiremos a partir de aquí como cliente. El trabajo se enmarca dentro del plan general del cliente, el cual tiende a lograr un elevado y competitivo manejo y control de las tecnologías. Desde este punto de vista, es de fundamental relevancia el conocimiento y control de tecnologías de punta para el estado argentino a los fines de favorecer la independencia tecnológica del país.

La RMN de baja resolución (llamada también en algunos casos RMN en el dominio temporal o TD-NMR por sus siglas en inglés), es una técnica muy utilizada para el control de calidad y el control de procesos en el sector industrial. Esto se debe en gran parte, a que los equipos para realizar experimentos de RMN de baja resolución (también conocidos como espectrómetros) son de bajo costo y mayor portabilidad que los utilizados en otras ramas de la RMN como las imágenes o la espectroscopia de alta resolución. El creciente desarrollo de la TD-NMR en las últimas décadas ha permitido en la actualidad una exitosa implementación de la técnica en industrias tales como la alimentaria, la química de los polímeros, y la médica y farmacéutica. Hoy en día, la TD-NMR continúa en pleno crecimiento, y existen una gran variedad de proyectos en etapa de investigación y desarrollo para ampliar la implementación de la técnica en diversas industrias.

Actualmente, la complejidad existente en la variada gama de experimentos que pueden realizar los espectrómetros de TD-NMR exige el constante aumento de la precisión en la medición y los datos medidos requieren ser procesados por distintas metodologías matemáticas para poder obtener resultados útiles. Como se sabe, una de las principales utilidades que se le da a la computación es el control de procesos. Por este motivo, la mayoría de los espectrómetros existentes son comandados por computadoras, las cuales se encargan, según el experimento que se desee realizar, de hacer las configuraciones necesarias del equipo, ejecutar las mediciones y realizar el procesamiento de los datos obtenidos.

Cada uno de los fabricantes de estos espectrómetros posee sus propias plataformas de experimentación, que se adecuan mejor a los distintos componentes de hardware que se utilizan en la fabricación. Estas plataformas son de carácter propietarias y para tener acceso a las mismas es necesaria la compra del equipamiento. Existe un proyecto de software libre, orientado a la investigación en RMN, llamado CCNP (por *Collaborative Computing Project for*

NMR) [1, 2]. Dicho proyecto, está orientado a resolver problemas de computación en equipos de RMN de alta resolución. Los experimentos en esta rama de la RMN son muy distintos a los que se desean realizar en TD-NMR, por lo que este proyecto no es de utilidad para el caso que nos interesa.

El cliente antes mencionado se dedica a la fabricación de espectrómetros de TD-NMR y posee varias plataformas diseñadas para realizar distintos experimentos. Uno de los problemas que presenta el cliente es que a medida que se van descubriendo nuevas aplicaciones comerciales de los distintos experimentos de TD-NMR, la transición desde los experimentos de las plataformas de experimentación a los software comerciales es muy lenta, ya que no se posee una estructura clara y precisa que permita representar, de manera genérica, dichos experimentos. El objetivo de este trabajo es resolver este problema.

Este trabajo consiste en el diseño de un modelo de datos que permite representar los principales experimentos de TD-NMR existentes en la actualidad. Este modelo ha sido llamado NMR_CORE y está orientado a resolver un problema que involucra tanto a las disciplinas de la computación como de la física, motivo por el cual debió ser diseñado de manera que pueda ser comprendido tanto por los desarrolladores de software como por los investigadores especializados en TD-NMR. Para esto se utilizaron estándares de definición de modelos diseñados para satisfacer este requerimiento.

Si bien no se encontraron antecedentes sobre el desarrollo de este tipo particular de modelo de datos, en [3] se define uno orientado a la representación de experimentos de RMN de alta resolución, por lo cual puede ser considerado un trabajo relacionado al trabajo que se presenta en este documento.

1.2 Estructura del documento

A continuación se describen los capítulos que conforman este trabajo:

Capítulo 2: se introducen los conceptos necesarios para poder comprender el trabajo realizado, lo cual incluye una breve descripción de la RMN y el modelado de datos.

Capítulo 3: se describe todo el proceso de diseño del modelo de datos NMR_CORE, desde la recolección de requerimientos hasta la definición del modelo lógico con sus invariantes, mediante el uso de estándares.

Capítulo 4: se muestran distintos ejemplos que permiten analizar la usabilidad del modelo NMR_CORE en la práctica y compararlo con las representaciones existentes.

Capítulo 5: se presentan las conclusiones del trabajo y se comentan posibles extensiones a futuro.

2 Marco teórico

2.1 RMN: Resonancia magnética nuclear

La RMN entendida como fenómeno físico está basada en las propiedades mecánico-cuánticas de los núcleos atómicos. Desde la demostración de su existencia en 1945, día a día se han ido desarrollando técnicas que la utilizan en distintos campos de la ciencia.

Este fenómeno es utilizado para determinar propiedades tanto macroscópicas como microscópicas de muestras en cualquier estado, por lo que el alcance que tiene su utilización es muy amplio. La espectroscopía pulsada de alta resolución constituye una de las principales técnicas empleadas para obtener información física, química, electrónica y estructural sobre moléculas. La poderosa serie de metodologías existente provee información sobre la topología, dinámica y estructura tridimensional de moléculas en solución y en estado sólido [4]. Esto permite hacerse una idea de la enorme dimensión que ha cobrado la RMN como técnica aplicada en distintas disciplinas científicas desde su descubrimiento hasta la actualidad.

Posiblemente la implementación más difundida, y que se relaciona cotidianamente al nombre de la resonancia magnética como técnica, es la aplicación que le da la medicina, la cual utiliza la capacidad de realizar imágenes para hacer diagnósticos de los distintos órganos del cuerpo, debido a la clara representación visual de las partes que componen los mismos, como se puede observar en la figura 2.1.

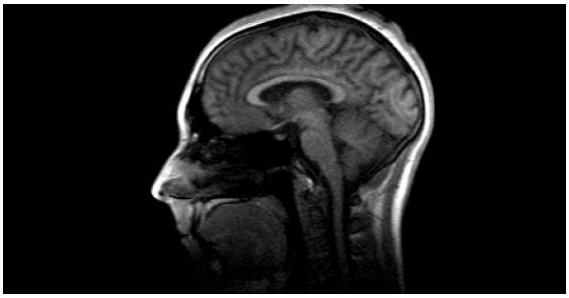

Figura 2.1: Imagen del cerebro obtenida utilizando RMN

La RMN de baja resolución por su parte constituye una rama de la RMN que puede ser implementada utilizando equipos de bajo costo y mayor portabilidad que en otros campos de la RMN, lo cual ha permitido su implementación en diversas áreas de la industria. El presente trabajo se desarrolla en base a metodologías de generación de pulsos de RF utilizando un equipamiento de RMN de baja resolución, por lo que introduciremos algunos conceptos necesarios para comprender esta técnica de experimentación, sin entrar en los detalles de la física involucrada. El lector interesado en profundizar acerca de las bases de la RMN puede consultar algunos de los libros de texto fundamentales citados en las referencias [5, 6, 7].

2.1.1 Muestra a analizar

Para comprender el fenómeno de RMN, introduciremos algunos conceptos elementales acerca de la composición microscópica de la muestra a analizar. Una muestra puede ser considerada como un conjunto de átomos, cada uno de los cuales posee un núcleo con una carga eléctrica y un momento angular o espín, el cual permite asociar al núcleo un momento magnético que puede ser representado por un vector. Para que el fenómeno de RMN sea posible, el espín nuclear debe ser no nulo. Un ejemplo de ello es el átomo de ^{1}H que es además el elemento que presenta mayor abundancia natural. Otros elementos con esta propiedad son el ^{19}F, el ^{31}P y el ^{13}C.

2.1.2 Experimentación en RMN

Para realizar un experimento de RMN, se coloca la muestra cuya composición se quiere estudiar en un campo magnético externo B_0, el cual tiene una dirección paralela al eje Z del sistema de coordenadas representado en la figura 2.2. Cuando la muestra contiene núcleos con espín no nulo, por ejemplo protones, al cabo de un tiempo la muestra se polariza alcanzando una magnetización de equilibrio debido a la suma neta de momentos magnéticos, que tendrá una dirección paralela al campo B_0 (y por lo tanto al eje Z). Este fenómeno se ve representado en la figura 2.2, en donde la flecha gris simboliza la magnetización neta. Luego, se somete la muestra a un campo de radiofrecuencia (RF) B_1, por un intervalo de tiempo que puede durar desde unidades a cientos de μs. Esto constituye un pulso de RF que es generado por un emisor y transmitido a la muestra por una antena. Cuando el campo de RF es aplicado a la frecuencia de Larmor (o "frecuencia de resonancia") $\omega = \gamma B_0$, donde γ se denomina constante giromagnética y tiene un valor característico para cada elemento, los núcleos absorben energía de RF y la magnetización de la muestra puede ser rotada hacia el plano transversal X-Y (figura 2.3). Luego del pulso de RF, la componente transversal de la magnetización precesa a la frecuencia de Larmor e induce en la antena una señal que puede ser enviada y amplificada a un receptor (figura 2.4). Esta señal es conocida como "señal de resonancia" y nos brinda información acerca de la muestra que deseamos estudiar. En la figura 2.5 se muestra un esquema de los componentes básicos de un espectrómetro de RMN.

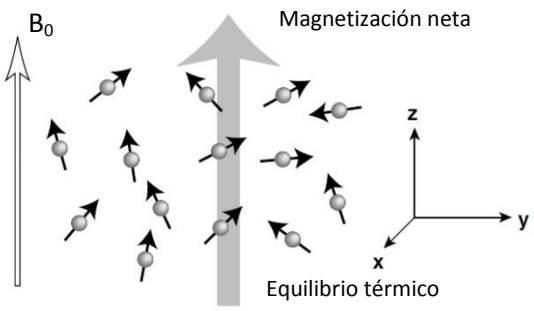

Figura 2.2: Muestra en equilibrio térmico dentro de un campo magnético B_0

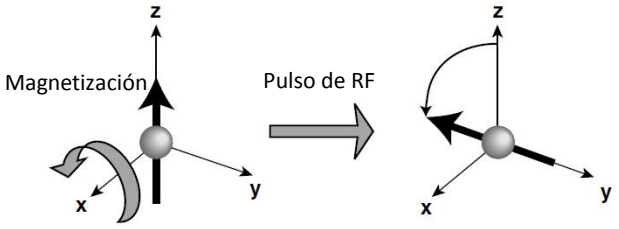

Figura 2.3: Rotación de la magnetización de la muestra al aplicar pulso de RF

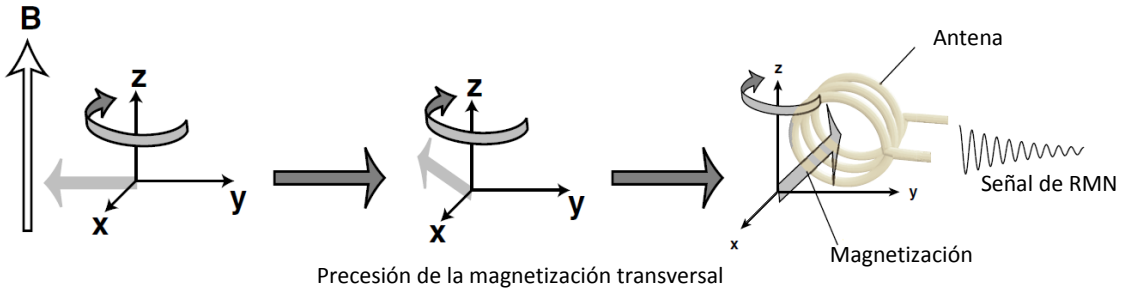

Figura 2.4: Detección de señal de RMN de la muestra en proceso de relajación transversal

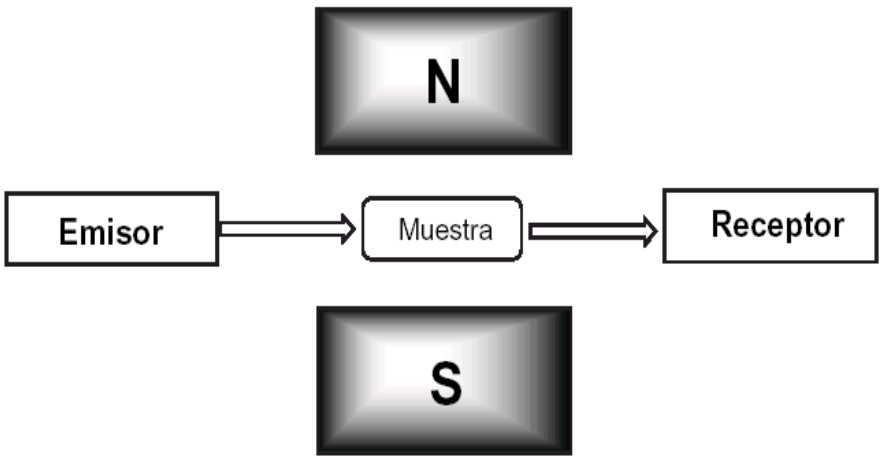

Figura 2.5: Esquema de componentes básicos de un espectrómetro de RMN

2.1.2.1 Procesos de relajación

En un experimento de RMN, se conoce como relajación al conjunto de procesos que ocurren luego de haber apartado a la muestra de su estado de equilibrio aplicando el pulso de RF hasta que la muestra vuelve nuevamente al estado de equilibrio en el cual se encontraba. Dos de los procesos más importantes son la relajación longitudinal y la relajación transversal.

La relajación longitudinal o relajación espín-red viene dada por el intercambio de energía entre espines nucleares y la red que rodea a este sistema. El efecto que produce es que la componente M_z de la magnetización luego de ser apartada del equilibrio por un pulso de RF se recupere según la ley

$$M_z(t) = M_{z0} (1 – \exp(-t/T_1)) \quad (1)$$

donde T_1 es el tiempo característico de recuperación denominado tiempo de relajación longitudinal o espín-red y M_{z0} es la componente Z de la magnetización en el estado de equilibrio.

La relajación transversal o relajación espín-espín viene dada por las interacciones que se producen entre espines vecinos. El efecto que produce es que la magnetización transversal decaiga según la ley

$$M_{xy}(t) = M_{xy0} \exp(-t/T_2) \quad (2)$$

donde T_2 es el tiempo característico de decaimiento denominado tiempo de relajación transversal o espín-espín y M_{xy0} es la componente transversal inicial de la magnetización.

Ambos tiempos de relajación T_1 y T_2 son dependientes de la movilidad molecular. Debido a esta propiedad, el estudio de las distribuciones de tiempos de relajación permite conocer propiedades de la muestra como el estado (sólido, líquido o gaseoso), la composición, estructura porosa, etc. La determinación y el estudio de las distribuciones de tiempos de relajación es una de las bases fundamentales de la TD-NMR.

2.1.3 Experimentos principales

Existe una amplia variedad de experimentos de TD-RMN, utilizados con distintos fines. En un experimento pueden combinarse varios pulsos de RF para obtener diferentes combinaciones de señales de RMN. Estas sucesiones son llamadas secuencias de pulsos. Los experimentos descriptos a continuación sirven para determinar los distintos T_2 y T_1 de la muestra, lo que es, como hemos visto, uno de los principales objetivos de la RMN de baja resolución. Las diferentes combinaciones de señales recibidas son generalmente digitalizadas en una computadora y procesadas para su análisis. En un experimento de RMN en la actualidad, la computadora comanda la generación de las secuencias de pulsos, la adquisición de las señales de RMN y el procesamiento de las mismas. A continuación se describen brevemente algunos de los experimentos más básicos.

2.1.3.1 FID: Decaimiento de inducción libre

La aplicación de un pulso de RF (B_1) por un cierto tiempo t_p, en una dirección perpendicular a B_0, rota la magnetización en un ángulo θ respecto del eje z dependiente de la magnitud del campo B_1 y su duración de la forma:

$$\theta = \gamma B_1 t_p \quad (3)$$

donde γ es la constante giromagnética única para cada tipo de átomo.

Un pulso de RF generalmente se representa por el ángulo de rotación de la magnetización y el eje alrededor del cual se produce esta rotación. Así por ejemplo $\pi/2_x$, representa un pulso que ha rotado a la magnetización $\pi/2$ (o 90°) alrededor del eje X.

El experimento más simple de RMN, consiste en aplicar un pulso de RF y observar la señal de RMN. Como vimos anteriormente, en este proceso: (i) se permite al sistema de espines de la muestra alcanzar el equilibrio térmico en el campo magnético externo, (ii) se rota a la magnetización mediante un pulso de RF enviado por el emisor y (iii) se detecta y amplifica por el receptor la señal de RF inducida por la precesión de la magnetización mientras alcanza su equilibrio térmico [5]. La señal generada se conoce como FID (del inglés *Free Induction Decay*). El pulso que se utiliza en este experimento es generalmente un pulso de $\pi/2$. La secuencia de pulsos se representa gráficamente como en la figura 2.6. La FID tiene por lo general un decaimiento exponencial como se observa en la figura. El tiempo de decaimiento característico se denomina T_2^* y es debido principalmente a las contribuciones de la relajación transversal y las inhomogeneidades de campo B_0.

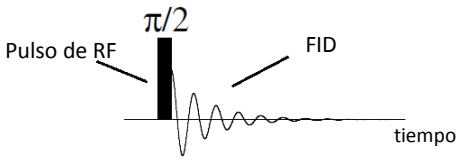

Figura 2.6: Secuencia de pulsos de RMN para detectar una FID

2.1.3.2 Eco de espín

El experimento eco de espín (o espín-eco) consiste en aplicar dos pulsos de RF separados por un tiempo t_1. El primero es un pulso de $\pi/2$ que rota la magnetización 90° al plano XY, originando una FID. Luego de un tiempo t_1, se aplica un pulso de π que invierte las fases de los espines. Esto genera un eco de la señal cuya amplitud se hace máxima en el tiempo $t_2 = t_1$, donde t_2 está medido a partir del pulso de π. La figura 2.7 representa gráficamente la secuencia de eco de espín.

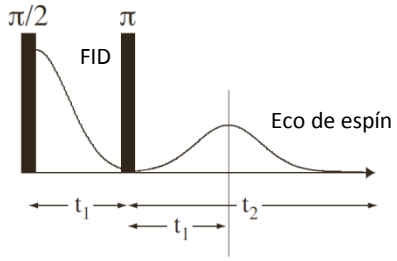

Figura 2.7: Secuencia de pulsos de un espín-eco

2.1.3.3 CPMG: Tren de ecos

La secuencia CPMG (por las siglas de sus inventores Carr-Purcell-Meiboom-Gill) consiste en aplicar una secuencia de eco de espín y, luego del pulso de π, volver a aplicar repetidamente pulsos de π a intervalos $t_2 = 2t_1$. Esto genera una sucesión de ecos, conocida como tren de ecos, cuyas amplitudes máximas van decayendo con la constante de relajación T_2. Por este motivo, esta secuencia es una de las más usadas para la determinación de los tiempos T_2 de una muestra. En la figura 2.8 se representa gráficamente la secuencia CPMG.

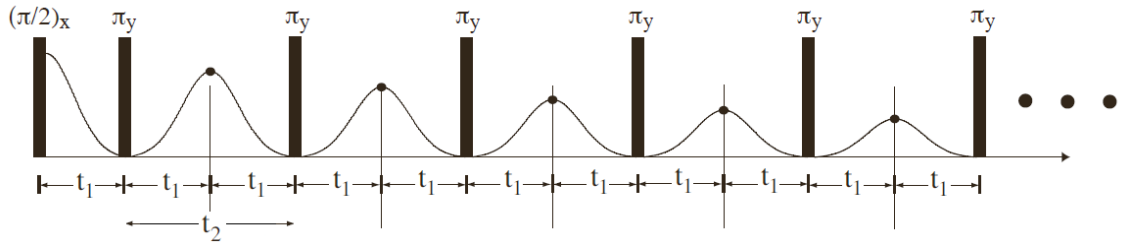

Figura 2.8: Secuencia de pulsos CPMG

2.1.3.4 Inversión-Recuperación

El método de inversión-recuperación se utiliza para medir el tiempo de relajación T_1 y consiste en la aplicación de un pulso de π, que lleva la magnetización en equilibrio a la dirección negativa del eje Z, y luego se espera un intervalo t_1, durante el cual la magnetización evoluciona hacia su valor de equilibrio. Finalmente, se observa la magnetización longitudinal resultante $M_z(t_1)$, rotando la misma al plano xy mediante un pulso de π/2. La magnetización $M_z(t_1)$ se recupera en el tiempo según:

$$M_z(t_1) = M_{zo}\,(1 - 2\exp(-t_1/T_1)) \qquad (4)$$

Repitiendo el mismo experimento para *n* valores de tiempo t_1, se puede reconstruir una sucesión de puntos descriptos por la ecuación (4), a partir de la cual se puede determinar el tiempo de relajación T_1. En la figura 2.9 se representa gráficamente la secuencia de inversión recuperación.

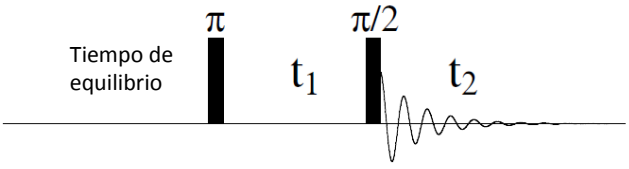

Figura 2.9: Secuencia de pulsos inversión-recuperación

2.2 Modelos de datos

Un modelo de datos define la estructura y el significado de los datos, por lo que generalmente se utiliza para especificar una base de datos. Existen varios términos, conceptualmente distintos, a los que se refiere cuando se habla de modelos de datos. En esta sección, se dará un panorama general de los mismos.

2.2.1 Arquitectura ANSI/SPARC

La arquitectura de modelos de datos dada por ANSI/SPARC [8, 9, 10], distingue tres niveles o esquemas de modelos: externo (o vista), conceptual y físico. Estos niveles están relacionados entre sí como se muestra en la figura 2.10.

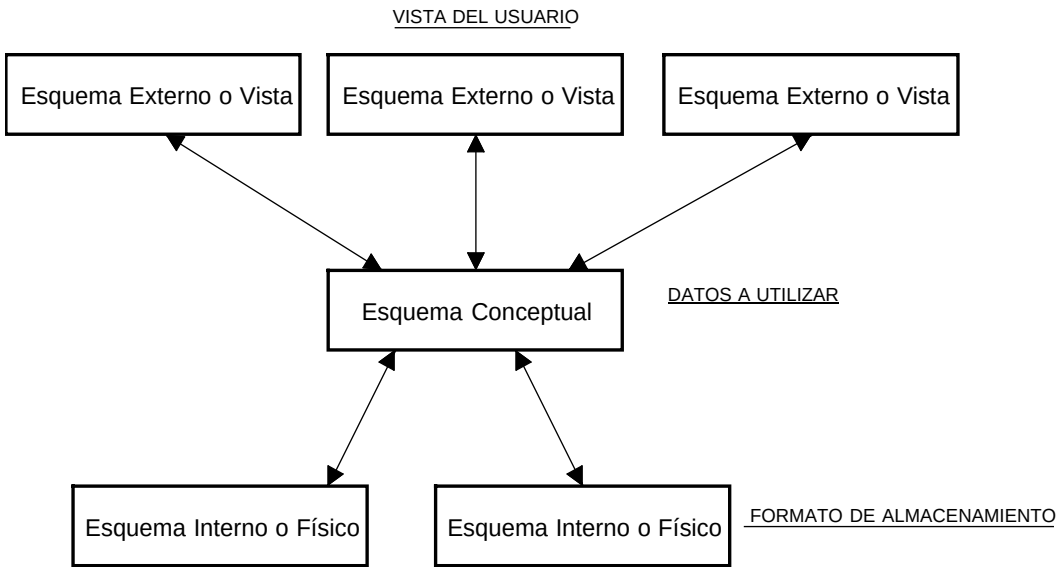

Figura 2.10: Arquitectura de modelo de datos dada por ANSI/SPARC

El nivel externo o vista se centra en la manera en la que se muestra la información a un usuario desde una perspectiva particular y con un propósito puntual. Puede haber muchas vistas distintas de un mismo esquema conceptual, y no necesariamente tienen que ser compatibles. En particular, las vistas son muy útiles para recolectar los requerimientos y las reglas que se aplican al área de trabajo. Los modelos de datos de la mayoría de las aplicaciones existentes son desarrollados tomando una vista de la sección de trabajo desde una perspectiva puntual del problema que intentan solucionar.

El esquema conceptual es la capa neutra del modelo, capaz de soportar cualquier vista que encaje con el alcance del proyecto y puede cambiar con el tiempo. Esto nos dice que este nivel no es el ideal para recolectar reglas, ya que posiblemente cambien.

El esquema físico representa el modo en que los datos serán guardados en la computadora. Puede haber distintos esquemas físicos para un mismo esquema conceptual. Como requerimiento, este nivel del modelo debe soportar el esquema conceptual.

2.2.2 Diseño de bases de datos

En la figura 2.11 [9] se muestran los principales pasos a seguir para el diseño completo de una base de datos, lo que incluye el modelado de datos.

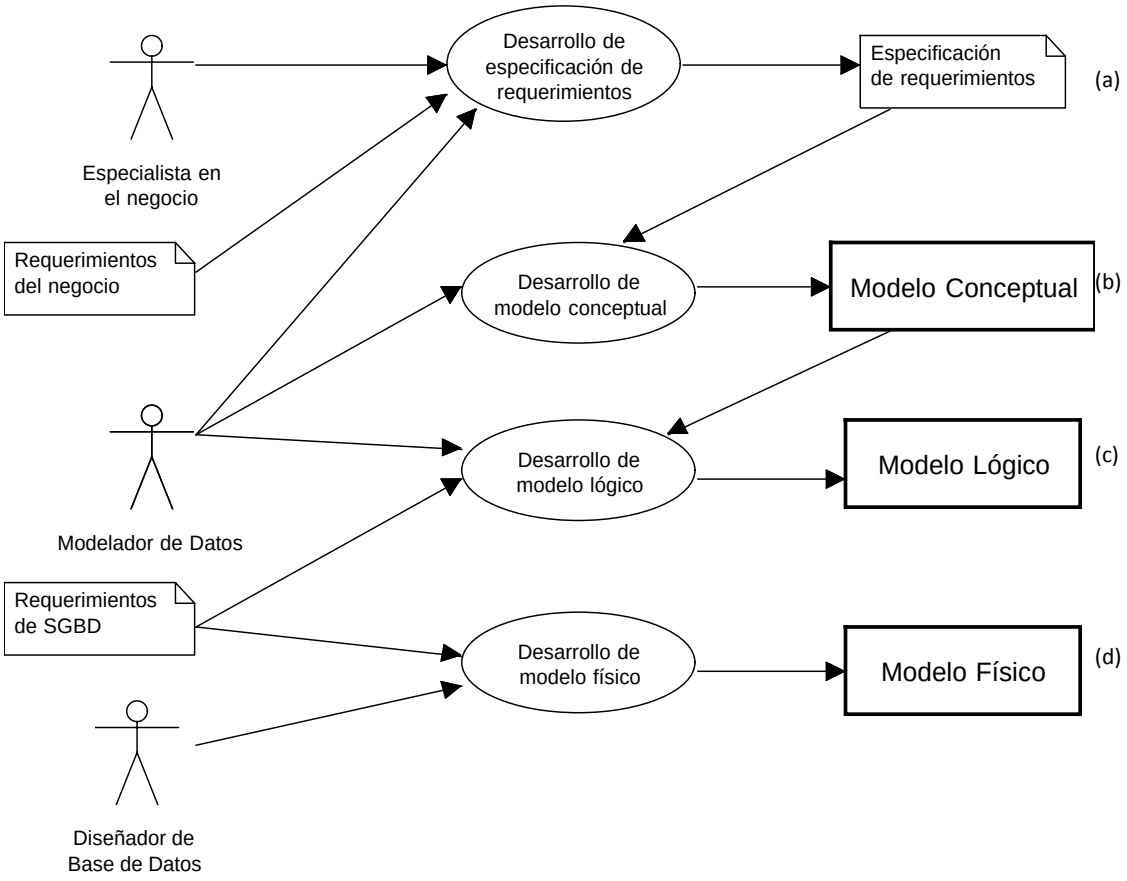

Figura 2.11: Vista general del proceso de diseño de una base de datos

En este diagrama se pueden observar tres modelos de datos distintos producidos en el proceso de especificación de una base de datos: el modelo conceptual, el modelo lógico y el modelo físico. Estos modelos son los definidos por ANSI/SPARC con la sutileza de que en este diagrama se le llama modelo lógico a un esquema externo específico.

Es importante hacer énfasis en el rol de la arquitectura de tres esquemas para aislar a los usuarios de los cambios que no son relevantes para ellos. Este aislamiento es la clave de los sistemas de gestión de base de datos actuales y es llamado independencia de los datos.

Veamos ahora, de manera superficial, cada paso del diseño identificando las personas y los objetos involucrados. En primer lugar, para comenzar el desarrollo de un modelo, tiene que existir la necesidad de representar la información de manera organizada para solucionar un problema existente. Esto sería lo que en el diagrama se describe como requerimientos del negocio. Con estos requerimientos, un modelador de datos junto con un especialista en el área sobre el cual se desea modelar, confeccionan un documento de especificación de requerimientos (a). Este documento deberá contener todos los requisitos que debe cumplir el

modelo de datos a desarrollar. Luego se analizan los requerimientos, identificando estructuras y las relaciones entre ellas, y se confecciona un modelo conceptual (b). El siguiente paso consiste en realizar una transición del modelo obtenido a una representación concreta de los datos a manejar. Esto implica definir qué propiedades tiene cada estructura en particular y qué función cumple cada propiedad. Cabe destacar que estas propiedades deben ser representadas de una manera que sea soportada por el manejador de bases de datos (SGBD, por Sistema de Gestión de Base de Datos) a utilizar (c). Finalmente, el diseñador de la base de datos deberá definir la forma en la que se almacenará el modelo (d).

A lo largo del tiempo se han ido utilizando distintas metodologías para realizar los pasos recién descriptos y dependen de la manera en la que se identifica la información. Por un lado, hay métodos que centran su diseño en analizar qué es lo que hace el sistema para el cual se está confeccionando el modelo y, una vez definido esto, deciden qué estructuras se requieren. Por otro lado, están los que primero analizan qué información es útil representar, y en base a eso definen los procesos del sistema. Últimamente, existe la tendencia en el software de utilizar objetos tanto para especificar sistemas de información como para desarrollarlos, por lo que parece ser que todos los procesos de desarrollo "giran en torno a los datos". Esto ha hecho que se generen estándares ampliamente utilizados para el desarrollo de tecnologías orientadas de objetos.

Para definir un modelo, principalmente los esquemas conceptuales y lógicos del mismo, el lenguaje natural suele dar lugar a confusiones, por lo que se requiere de estándares que permitan representar las estructuras, las relaciones entre ellas y las características que tienen, de manera que todas las personas que vean el modelo definido lo interpreten de la misma forma. Por ello, en las próximas secciones se introducen dos estándares de modelado de datos ampliamente utilizados: UML y OCL.

2.3 UML: El lenguaje unificado de modelado

El lenguaje unificado de modelado (Unified Modeling Language, UML) [11, 12, 13, 14] es un lenguaje con notación gráfica, de propósito general, que ayuda a describir, diseñar y documentar los modelos de un sistema. El mismo es ampliamente usado, ya que permite a los creadores de sistemas generar diseños que capturen sus ideas en una forma convencional y fácil de comprender para comunicarlas a otras personas. UML ofrece distintos tipos de diagramas para modelar los diferentes aspectos o vistas de un sistema. En este trabajo utilizaremos sólo tres de estos diagramas: los diagramas de clases, los diagramas de objetos y, en menor medida, los diagramas de secuencia.

2.3.1 Diagramas de clases

Los diagramas de clases describen los tipos de objetos involucrados en el sistema y las relaciones que existen entre los mismos. Además, estos diagramas muestran las propiedades y operaciones de cada uno de los objetos involucrados.

Un diagrama de clases está compuesto por:

- **Clases:** Son utilizadas para modelar los objetos involucrados en el sistema que tienen las mismas propiedades, relaciones y métodos. Cada objeto es una instancia de la clase a la que pertenece.
- **Atributos:** Son utilizados para modelar las propiedades estructurales de los objetos de una clase. Cada atributo tiene un nombre y un tipo, que especifica el dominio de los posibles valores del atributo.
- **Asociaciones:** Representan las relaciones estructurales entre clases. Cada conexión de una asociación se llama extremo de asociación y es común asignarle un nombre que dé idea de su función.
- **Multiplicidades:** Son utilizadas para indicar cuántas instancias de la clase que está conectada a un extremo de asociación pueden estar relacionados con una instancia de la clase que está conectada al otro extremo de la asociación. Las multiplicidades se definen mediante intervalos de números o mediante un número exacto. Una multiplicidad muy utilizada es la *, que significa 0 ó más instancias y es la multiplicidad que, por defecto, se asocia a un extremo de asociación.
- **Métodos:** Son utilizados para modelar las operaciones que todas las instancias de una clase implementan.
- **Generalizaciones:** Son utilizadas para generar una clase específica a partir de una clase más general. Cada instancia de la clase específica es también instancia de la clase general: tiene las características (propiedades, relaciones, métodos) de la clase general, además de las de su propia clase.
- **Clases Asociación:** Son utilizadas para asignar atributos a las relaciones entre clases representadas mediante asociaciones. Una clase asociación es representada como una clase que en vez de vincularse con otras clases, se relaciona a una asociación.

En la figura 6.1 del apéndice se muestra la forma en la cual se representa cada uno de estos elementos.

A continuación mencionamos un ejemplo tomado de [13]*:*

En la figura 2.12 se muestra el diagrama de clases que modela la “estructura” básica de una empresa que ofrece vehículos tanto para servicios de viajes regulares como para viajes privados. En concreto, La clase Trip modela los viajes. Contiene dos atributos **origin** y **destination** que modelan el origen y el destino de los viajes.

- La clase Coach modela los vehículos. Contiene dos atributos **model** y **numberOfSeats** que especifican el modelo del vehículo y los asientos que tiene.
- Las clases PrivateTrip y RegularTrip son subclases de Trip y modelan los dos tipos posibles de viajes: en concreto,
    - La clase PrivateTrip modela los viajes privados.

  - La clase RegularTrip modela los viajes regulares. Contiene un atributo **availableSeats** que modela el número de asientos todavía disponibles en el vehículo que va a transitar la ruta.
- La clase Person modela los pasajeros de un viaje regular. Contiene un atributo **name** que modela el nombre del pasajero.
- La clase Trip, y por herencia, la clase PrivateTrip y RegularTrip, están relacionadas mediante la asociación *tripCoach*, que modela los viajes que realiza cada vehículo y en qué vehículos se realiza cada viaje.
- Cada viaje, representado con un objeto de la clase Trip, puede realizarse en muchos vehículos y cada vehículo, representado con un objeto de la clase Coach, puede realizar muchos viajes.
- Los extremos de asociación *coaches* y *trips* representan todos los vehículos asociados a un determinado viaje y todos los viajes asociados a un determinado vehículo, respectivamente.

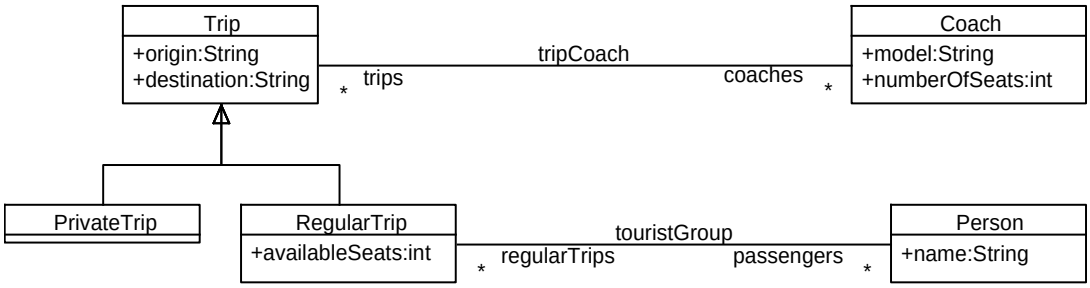

Figura 2.12: Diagrama de clases de empresa de vehículos

2.3.2 Diagramas de objetos

Un diagrama de objetos captura el estado de los objetos del modelo de un sistema en un determinado punto en el tiempo. Estos son comúnmente utilizados para el análisis y validación de los modelos que describen los diagramas de clase.

Basado en la descripción dada en [13], podemos decir que un diagrama de objetos está compuesto por:

- **Objetos:** Son instancias de las clases. Sus atributos (tanto propios como "heredados") pueden tener valores asignados.
- **Enlaces ("links"):** Son instancias de las asociaciones entre clases.

Continuando con el ejemplo tomado de Dania [13]:

En la figura 2.13 se muestra el diagrama de objetos que modela un estado concreto de la empresa de vehículos descrita en la figura 2.12, en el que:

- la empresa dispone actualmente de una flota de cinco vehículos, aunque cuatro de ellos ya están comprometidos.
- la ruta Cordoba-AltaGracia tiene actualmente un pasajero, y la ruta Neuquen-Misiones tiene dos pasajeros.

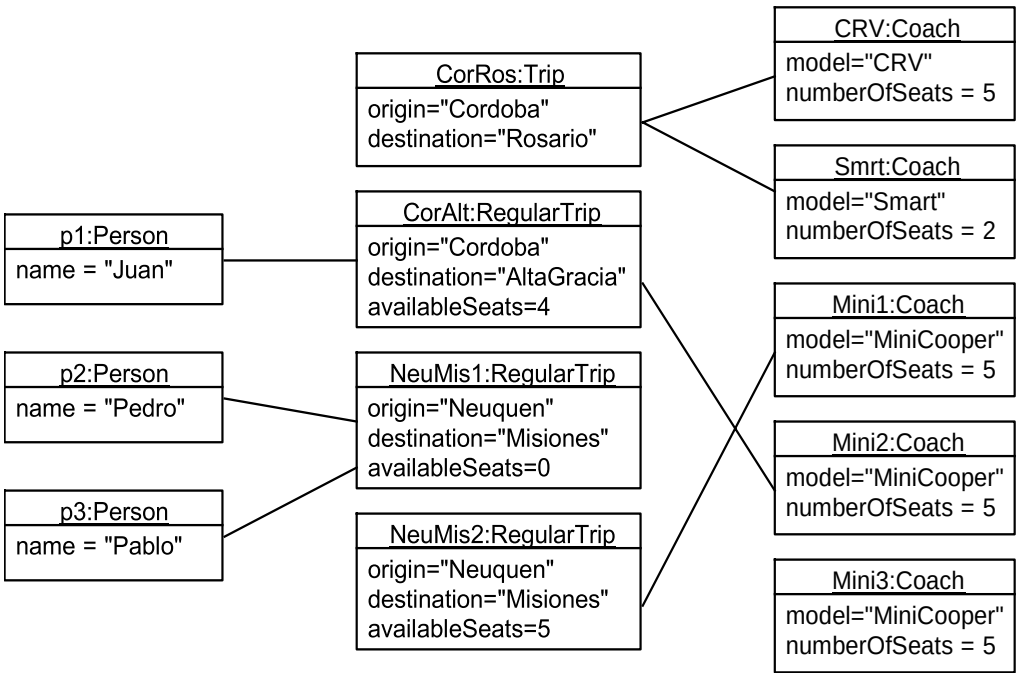

Figura 2.13: Diagrama de objetos de ejemplo de empresa de vehículos

2.3.3 Diagramas de secuencia

Un diagrama de secuencia muestra una interacción entre un grupo de objetos en un escenario particular. Esta interacción se puede visualizar en el diagrama de manera ordenada según la secuencia temporal de eventos que suceden en dicho escenario.

Un diagrama de secuencia está compuesto por:

- **Objetos:** Son instancias de las clases.
- **Mensajes:** Representan comunicaciones entre objetos.
- **Cuadros de activación:** Representan el tiempo que un objeto necesita para completar una tarea.
- **Líneas de vida:** Indican la presencia de un objeto en el escenario.

En un diagrama de secuencia, el eje vertical representa el tiempo, y en el eje horizontal se colocan los objetos y actores participantes en la interacción, sin un orden prefijado. Cada objeto o actor tiene una línea de vida, y los mensajes se representan mediante flechas entre los distintos objetos. El tiempo fluye de arriba hacia abajo.

Continuando con el ejemplo tomado de [13]:

En la figura 2.14 se muestra el diagrama de secuencia del escenario en el cual, el objeto *p1* consulta en qué modelo de vehículo se realiza el viaje por la ruta Cordoba-AltaGracia, según el diagrama de objetos de la figura 2.13.

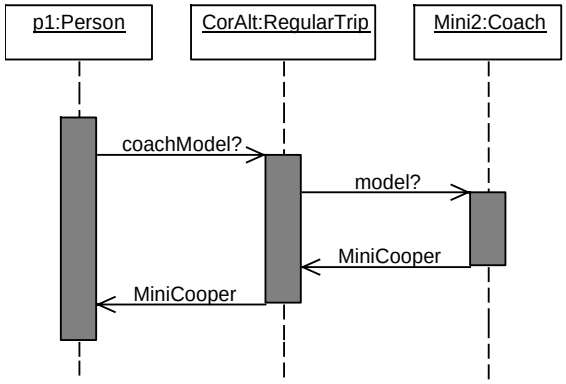

Figura 2.14: Diagrama de secuencia de ejemplo de empresa de vehículos

2.4 OCL: Un lenguaje de restricciones y consultas

Generalmente al hablar de modelos imaginamos dibujos que contienen varias burbujas con un nombre, flechas que las unen y comentarios explicativos. Si bien esto da una visión general simple del modelo, a la hora de interpretar un diagrama surgen preguntas del estilo: ¿para qué sirve esto? ¿Es válido esto otro? Esto se debe a que la información contenida en un diagrama no es completa, y al ser una representación generalmente informal, es imprecisa y en ocasiones, incluso inconsistente. Muchos de los defectos en un modelo se deben a las limitaciones de los diagramas que se están usando. Simplemente, podemos decir que un diagrama no puede expresar las declaraciones que deberían ser parte de una especificación minuciosa.

La notación UML está fuertemente basada en diagramas, por lo que para dotarlo del nivel de precisión y expresividad que se desea en ciertos aspectos de un diseño, se extendió el estándar con la especificación del lenguaje de restricciones de objetos (Object Contraint Language, OCL) [15]. OCL es un lenguaje textual utilizado como estándar, en UML 1.1, para especificar invariantes, precondiciones y postcondiciones. A partir de UML 2.0 se amplió su uso, incluyendo la definición de meta-modelos de dominio específico, la transformación de modelos, y el testing y la validación de modelos, entre otros.

Algunas de las propiedades que caracterizan a OCL son:

- *Lenguaje de especificación puro:* la evaluación de una expresión es instantánea y simplemente devuelve un valor sin cambiar nada en el modelo.
- *Lenguaje fuertemente tipado:* toda expresión tiene un tipo asociado que describe el dominio del resultado de dicha expresión. Los tipos se organizan en una jerarquía de tipos,

que determina cuándo dos tipos distintos son conformes. Se dice que un tipo *tipo1* es conforme con un tipo *tipo2* cuando una instancia del *tipo1* puede utilizarse en cada lugar donde se espera una instancia del *tipo2*.

- *Lenguaje contextualizado:* sus expresiones se escriben en el contexto que proporciona un modelo contextual.

Este lenguaje fue diseñado con el objetivo fundamental de hacerlo ampliamente utilizable: "deberá ser fácilmente escrito y leído por todos los profesionales de la tecnología de objetos y por sus clientes, es decir, gente que no son matemáticos o ingenieros en informática" [13] .

Los tipos de OCL se organizan en las siguientes categorías:

- **Tipos primitivos:** Son los tipos básicos Boolean, Integer, Real y String.
- **Tipos clase:** Son las clases del modelo contextual. Por ejemplo, Trip es un tipo OCL cuando el modelo contextual es el diagrama de clases de la figura 2.12.
- **Tipos colección:** Son los tipos parametrizados Set, Bag, OrderedSet y Sequence. Sus parámetros pueden ser cualquier otro tipo, incluidos los tipos colección. Por ejemplo, el tipo Set(Boolean) de los conjuntos de booleanos, o el tipo Sequence(Trip) de las secuencias de viajes, en el contexto del diagrama de clases de la figura 2.12.
- **Tipos tupla:** Es el tipo parametrizado Tuple. Para cada tipo Tuple los tipos de sus elementos pueden ser cualesquiera. Por ejemplo, Tuple{name:String, age:Integer} para pares de cadenas y enteros, o Tuple{name:String, parents:Set(Person)} para pares de cadenas y conjuntos de pasajeros, en el contexto del diagrama de clases de la figura 2.12.
- **Tipos especiales:** Son los tipos Invalid, Void y Any. Invalid conforma a todos los tipos excepto a Void: la única instancia del tipo Invalid es el valor oclInvalid. Void representa un tipo que conforma a todos los tipos: la única instancia de Void es undefined (o null). Any es el tipo al que todos los demás tipos conforman.

En las tablas 2.1, 2.2, 2.3, 2.4 y 2.5 se describe el subconjunto de las operaciones predefinidas de OCL utilizadas en este trabajo, las que incluyen operaciones sobre los tipos primitivos, sobre los tipos de clase proporcionados por el modelo contextual, y sobre los tipos colección. La descripción completa del lenguaje OCL se encuentra en [15].

Any	
(exp:Any).oclIsUndefined()	Devuelve verdadero si *exp* es *null* ó es *Invalid*
(exp_1:Any).oclIsKindOf(exp_2:Any)	Devuelve verdadero si *exp1* es una instancia (directa o indirecta) de la clase *exp2*
(exp_1:Any).oclIsTypeOf(exp_2:Any)	Devuelve verdadero si exp_1 es una instancia directa de la clase exp_2
(exp_1:Any).oclAsType(exp_2:Any)	Devuelve exp_1 transformada a la clase exp_2, si exp_1 es una instancia (directa o indirecta) de la clase exp_2. Si no, devuelve *Invalid*

Tabla 2.1: Operaciones sobre todos los tipos

Boolean	
(exp_1:Boolean).and(exp_2:Boolean)	Devuelve la conjunción de exp_1 y exp_2
(exp_1:Boolean).or(exp_2:Boolean)	Devuelve la disyunción de exp_1 y exp_2
(exp:Boolean).not()	Devuelve la negación de *exp*
(exp_1:Boolean).implies(exp_2:Boolean)	Devuelve la implicación de exp_1 a exp_2

Tabla 2.2: Operaciones sobre objetos de tipo Boolean

Sobre el modelo	
(exp:Col(Class)).allInstances()	Devuelve todas las instancias de *exp*
(exp:[Class | Collection(Class)]).atr	Devuelve el valor del atributo *atr* para cada uno de los elementos incluidos en *exp*
(exp:[Class | Collection(Class)]).rol	Devuelve todos los objetos enlazados por el extremo de asociación *rol* para cada uno de los elementos incluidos en *exp*

Tabla 2.3: Operaciones sobre los tipos de clase proporcionados por el modelo contextual

Collection(Basic)	
(exp:Collection(Basic)) → size()	Devuelve el número de elementos de la colección *exp*
(exp:Collection(Basic)) → isEmpty()	Devuelve verdadero si la colección *exp* es vacía
(exp:Collection(Basic)) → notEmpty()	Devuelve verdadero si la colección *exp* no es vacía
(exp_1:Collection(Basic)) → includes(exp_2:Basic)	Devuelve verdadero si la colección *exp_1* contiene el elemento *exp_2*
(exp_1:Collection(Basic)) → includesAll(exp_2:Collection(Basic))	Devuelve verdadero si la colección *exp_1* contiene a la colección *exp_2*
(exp_1:Collection(Basic)) → excludes(exp_2:Basic)	Devuelve verdadero si la colección *exp_1* no contiene el elemento *exp_2*
(exp_1:Collection(Basic)) → excludesAll(exp_2:Collection(Basic))	Devuelve verdadero si la colección *exp_1* no contiene a la colección *exp_2*
(src:Col(Basic)) → forAll(var|body)	Devuelve verdadero si la evaluación de *body* devuelve verdadero para cada elemento de la colección *src*
(src:Collection(Basic)) → select(var|body)	Devuelve la colección formada por todos los elementos de *src* para los que *body* evalúa a verdadero
(src:Collection(Basic)) → exists(var|body)	Devuelve verdadero si la evaluación de *body* devuelve verdadero algún de la colección *src*
(src:Collection(Basic)) → one(var|body)	Devuelve verdadero si la evaluación de *body* devuelve verdadero para uno y sólo un elemento de la colección *src*

Tabla 2.4: Operaciones sobre los tipo Collection

OrderedSet(Basic)	
(exp_1:OrderedSet(Basic)) → indexOf(exp_2:Basic)	Devuelve un *Integer* con la posición del elemento *exp_2* dentro de la colección *exp_1*

Tabla 2.5: Operación sobre el tipo OrderedSet

Continuando con el ejemplo tomado de Dania [13]:

Para ilustrar el uso de OCL, formalizamos a continuación algunas expresiones sobre el diagrama de objetos de la figura 2.13.

"El viaje *CorAlt* de Cordoba a AltaGracia" se formaliza en OCL como:

```
CorAlt
```

"Todos los viajes" se formaliza en OCL como:

```
Trip.allInstances()
```

"El número total de viajes" se formaliza en OCL como:

```
Trip.allInstances() → size()
```

"El número total de asientos disponibles en los viajes *NeuMis1* y *NeuMis2* de Neuquén a Misiones" se formaliza en OCL como:

```
NeuMis1.availableSeats + NeuMis2.availableSeats
```

"Los modelos de todos los vehículos" se formaliza en OCL como:

```
Coach.allInstances().model
```

"Todos los vehículos que están comprometidos para realizar viajes" se formaliza en OCL como:

```
Trip.allInstances().coaches
```

"Todos los viajes que tienen como origen la ciudad de Cordoba" se formaliza en OCL como:

```
Trip.allInstances() → select(t|t.origin = 'Cordoba')
```

"Si todos los viajes tienen como origen la ciudad de Cordoba" se formaliza en OCL como:

```
Trip.allInstances() → forAll(t|t.origin='Cordoba')
```

"Los nombres de los pasajeros de todos los viajes que tienen como destino la ciudad de Misiones" se formaliza en OCL como:

```
Trip.allInstances() → select(t|
    t.destination = 'Misiones').passengers.name
```

3 NMR_CORE: Núcleo de aplicaciones TD-NMR

Como ya mencionamos en el sección 1, el objetivo de este trabajo es especificar un modelo de datos que permita representar experimentos de resonancia magnética nuclear para equipos de baja resolución. Basándonos en la arquitectura dada por ANSI/SPARC vista en la sección 2.2.1, nos concentraremos en dar una definición detallada del esquema conceptual y del esquema lógico.

Nuestro modelo será desarrollado en base al paradigma orientado a objetos (aunque el modelo conceptual es neutro, se representará utilizando clases), ya que existen estándares bien definidos para este tipo de modelado que son ampliamente usados y son fáciles de entender por la personas que no trabajan en software, lo cual ha sido establecido como un requisito de nuestro objetivo. Estos estándares son UML y OCL, descriptos en las secciones 2.3 y 2.4 respectivamente, que combinados dan una definición bastante formal del modelo.

Si bien el esquema físico no es menos importante, depende mucho de la implementación y, en la actualidad, existen muchos ORM (*Object-Relational Mapping*) que toman como entrada el esquema lógico de un modelo, generan una base de datos relacional en cualquier motor estándar existente que da soporte a este esquema y proveen una interfaz que les da a los desarrolladores acceso directo a los objetos. Esto resuelve el esquema físico en el paradigma orientado a objetos.

3.1 Análisis de requisitos

Esta parte del proceso permite tener una visión concreta de las propiedades y reglas que se desean en la solución. En esta etapa, se trabaja en conjunto con un especialista en TD-NMR y con el cliente, quien define el alcance que debe tener el trabajo. También se realiza un análisis breve de las aplicaciones existentes, provistas por el cliente, que se utilizan para la experimentación en TD-NMR actualmente.

3.1.1 Necesidad del cliente

Actualmente, el cliente desarrolla aplicaciones que realizan experimentos predefinidos en equipos de RMN de baja resolución para satisfacer las necesidades de los mercados a los que apunta. Estas aplicaciones, al haber sido desarrolladas con metodologías ágiles, están orientadas a resolver un problema particular dentro de la variada cantidad de problemas que se pueden resolver con RMN.

Para facilitar la experimentación en esta área, el cliente desea normalizar la manera en que se define un experimento. Un experimento debe tener la configuración del equipo a utilizar para realizar la medición y la rutina a ejecutar para obtener los resultados. Una rutina es una secuencia de pasos a seguir que incluye: configuración del equipo, medición en el equipo y utilización de la señal adquirida en la medición para obtener los resultados deseados usando algún método de procesamiento de señales.

3.1.2 Aplicaciones existentes

En la actualidad, el cliente posee una amplia variedad de aplicaciones desarrolladas. La mayoría son aplicaciones comerciales orientadas a usuarios que implementan la RMN en diversas áreas de la industria con pocos conocimientos en los distintos procesos de medición involucrados. Hay dos aplicaciones que no forman parte de la mayoría mencionada y están orientadas a la investigación y a la generación dinámica de experimentos. *Condor Lite* y *Condor IDE* son estas herramientas y nos serán muy útiles para detectar las propiedades deseadas para el modelo. A continuación, se procederá a analizar ambas aplicaciones focalizándose en el modelo intrínseco sobre el cual se ejecutan.

3.1.2.1 Condor Lite

La creación de esta aplicación tuvo como principal objetivo simplificar la configuración y ejecución de cinco experimentos puntuales de TD-NMR.

Antes de entrar en cada experimento en particular, notemos algunas características de esta plataforma de experimentación:

- Cada experimento tiene su propia configuración del equipo, su propia secuencia de pulsos a ejecutar y sus propios parámetros de preprocesamiento de la señal adquirida.
- Hay un solo experimento independiente. Los restantes, utilizan en alguna medida a este mismo en su ejecución.
- Hay dos experimentos en los que se pueden diferenciar dos etapas: la primera, en la cual se realiza la medición y preprocesamiento de la señal, y la segunda, que es opcional y procesa la señal resultante de la primera etapa.
- Las configuraciones de todos los experimentos se agrupan en un solo archivo. A estos archivos se les da el nombre de proyectos.

La aplicación tiene incorporados los siguientes experimentos:

- Ajuste de Frecuencia *(independiente)*
- Curva de nutación
- CPMG
- Inversion-Recovery (Inversión-Recuperación)
- Inversion-Recovery-CPMG (Inversión-Recuperación-CPMG)

Veamos ahora, brevemente, en qué consiste cada uno de estos experimentos.

Ajuste de Frecuencia

Este experimento consiste en realizar 4 tareas secuenciales:

1. Configuración de las frecuencias y las fases del pulso de RF del equipo con los valores propios del experimento.
2. Ejecución de la secuencia de pulsos configurada en el experimento (usualmente FID).
3. Procesamiento de la señal adquirida, que determina la frecuencia correcta que debe utilizarse para medir.
4. Reemplazo en la configuración del experimento los valores de frecuencia actuales por los valores obtenidos del procesamiento.

Curva de nutación

Este experimento utiliza el ajuste de frecuencia en su ejecución. Consiste en realizar ocho tareas secuenciales:

1. Realizar un ajuste de frecuencia.
2. Reemplazo en la configuración del experimento los valores de frecuencia actuales por los valores resultantes de la configuración del experimento de ajuste de frecuencia.
3. Ejecución de la secuencia de pulsos configurada en el experimento (usualmente FID).
4. Procesamiento de la señal, que devuelve el punto máximo de la magnitud de la señal.
5. Reconfiguración de la secuencia de pulsos.
6. Repetición del paso 3 a 5, *n* veces.
7. Procesamiento de los *n* puntos generados en el paso 4 (devuelve dos valores, que son *p90* y *p180*, los que representan los tiempos que deben durar los pulsos de RF para obtener pulsos de $\pi/2$ y π).
8. Visualización de resultados del procesamiento

CPMG

Este experimento tiene dos etapas. La primera etapa consiste en realizar cinco tareas secuenciales:

1. Realizar un ajuste de frecuencia.
2. Reemplazo en la configuración del experimento los valores de frecuencia actuales por los valores resultantes de la configuración del experimento de ajuste de frecuencia.
3. Ejecución de la secuencia de pulsos CPMG configurada en el experimento.
4. Procesamiento de la señal utilizando la configuración del experimento (devuelve una señal).
5. Visualización de la señal resultante del procesamiento (con opción de almacenarla).

La segunda etapa es opcional y consiste en 3 tareas secuenciales:

1. Selección del método de procesamiento a utilizar.
2. Procesamiento de la señal resultante de la primera etapa.
3. Visualización en una cuadrícula de los resultados (con opción de almacenarlos).

Inversion-Recovery

Al igual que el experimento CPMG, este experimento tiene dos etapas. La primera etapa consiste en realizar ocho tareas secuenciales:

1. Realizar un ajuste de frecuencia.
2. Reemplazo en la configuración del experimento los valores de frecuencia actuales por los valores resultantes de la configuración del experimento de ajuste de frecuencia.
3. Ejecución de la secuencia de pulsos Inversion-Recuperación configurada en el experimento.
4. Procesamiento de la señal utilizando la configuración del experimento (devuelve una señal).
5. Reconfiguración de la secuencia de pulsos.
6. Repetición del paso 1 al 5, *n* veces.
7. Procesamiento de las *n* señales generadas en el paso 4 utilizando la configuración del experimento (devuelve una señal).
8. Visualización de la señal resultante del procesamiento (con opción de almacenarla).

La segunda etapa es opcional y consiste en realizar los mismos pasos que en la segunda etapa del experimento CPMG.

Inversion-Recovery-CPMG

Este experimento consta de una sola etapa. La misma consiste en realizar los mismos pasos realizados en la primera etapa del experimento Inversion-Recovery, con la diferencia de que la secuencia de pulsos ejecutada en el paso 3, en este caso, será una secuencia Inversión-Recuperación-CPMG, en vez de una Inversión-Recuperación (en este caso el pulso de $\pi/2$ es reemplazado por una secuencia CPMG).

Luego de ver brevemente en qué consiste cada uno de los experimentos, fijemos algunos conceptos o nombres que aparecerán en el diseño. Las iteraciones que se realizan en la curva de nutación se conocen con el nombre de *barrido*. Este proceso, consiste en realizar varias mediciones secuenciales en las cuales se va modificando progresivamente algún parámetro de la secuencia de pulsos. En este caso puntual, lo que se va modificando es la duración del pulso de RF.

Por último, notemos que el ajuste de frecuencia es una pieza fundamental en todos los experimentos, ya que, como vimos en la sección 2.1, la sincronización entre la intensidad del campo magnético y el pulso de RF es un requisito indispensable de la RMN.

3.1.2.2 Condor IDE

Esta aplicación, es la plataforma más completa que posee el cliente en cuanto a capacidad de experimentación. Provee dos lenguajes mediante los cuales se le permite al usuario programar sus propios experimentos. En sí, la aplicación es un compilador de los lenguajes E+ y P+, por lo que la capacidad de representación tiene como límite el poder de expresividad que provean los mismos.

Centraremos la búsqueda de requerimientos en la comprensión de la filosofía con la que fue construido este sistema y sus respectivos lenguajes, sin entrar en detalles acerca de la sintaxis de los mismos.

El lenguaje P+ permite programar y representar secuencias de pulsos, las cuales consisten en una sucesión de eventos, de los cuales algunos son pulsos de RF (irradiación de potencia), otros son intervalos de espera, y otros son intervalos de adquisición de la señal.

El lenguaje E+, por otro lado, fue diseñado para la programación de experimentos de mediana y alta complejidad. Este lenguaje, interactúa con P+ para realizar la adquisición de la señal y añade muchos comandos de alto nivel para el tratamiento de la señal y para lograr una interacción con el usuario del equipo. Además de esto, E+ dispone de una serie de sentencias que permiten construir la interfaz gráfica de la aplicación con la que interactúa el usuario.

Como E+ provee una funcionalidad acotada, se le dio la posibilidad de poder importar funciones compiladas con cierta estructura predefinida para realizar distintos procesamientos, que pueden ser utilizadas a lo largo de un experimento. Esta clase de librerías matemáticas dejan la puerta abierta a que el software se pueda adaptar a las nuevas metodologías que puedan surgir.

Uno de los problemas que presenta esta aplicación, es que la cantidad de funcionalidad que brinda el lenguaje E+ complejiza el proceso de experimentación. Además, si bien permite expresar muchos procesamientos de señal, no fue diseñado para hacer esta tarea por lo que no provee el poder de expresividad y la eficiencia de los lenguajes diseñados con ese propósito.

En este trabajo se utiliza como antecedente el modelo intrínseco de esta aplicación, buscando lograr un modelo que permita representar toda la funcionalidad de la misma, focalizándose en dar una solución precisa a cada problema en particular.

3.1.3 Recolección de requerimientos

En base al estudio de las necesidades del cliente y el análisis de las aplicaciones utilizadas actualmente por el mismo, definiremos cuales son los requisitos que fijan la filosofía sobre la cual deberá desarrollarse nuestro modelo.

A continuación, se listarán los dichos requerimientos:

- *Ampliamente genérico:*

Se desea que el modelo tenga la capacidad de representar los experimentos de RMN más utilizados de manera integral, lo que implica configuraciones del equipo, adquisiciones de señal, procesamiento de las señales adquiridas y presentación de resultados. Los experimentos en los que se deberá basar el diseño son los descritos en el capítulo 2.1 combinados con la manera secuencial de armar los pasos involucrados en la experimentación que tiene la aplicación *Condor Lite*.

- *Simplicidad en la interpretación:*

Si bien este modelo será diseñado con el fin de estandarizar la representación de experimentos RMN a nivel de software, dichos experimentos serán diseñados por personas idóneas en los aspectos teóricos de la RMN. Por esto, es requisito que la solución dada pueda ser comprendida no sólo por los desarrolladores de aplicaciones de RMN, sino también por los futuros usuarios con conocimientos teóricos de la física aplicada.

- *Granularidad:*

Generalmente, cuando uno habla de un experimento de RMN, se refiere a una secuencia de experimentos o tareas a realizar para obtener un resultado, por lo que en la realidad se pierde la granularidad en la definición. En el modelo, será necesaria la correcta agrupación de tareas, de manera que cada grupo o conjunto de las mismas tenga un significado que brinde la noción del principio físico que se está utilizando. Además, deberá haber agrupaciones de tareas que representen la parte del experimento que se está realizando en el equipo de RMN.

- *Modificación de parámetros del experimento en tiempo de ejecución:*

En algunos casos suele ser necesario ajustar algunos parámetros de un experimento en tiempo de ejecución, ya que al trabajar con equipos electrónicos surgen errores inducidos por diferentes factores como la temperatura, la calidad de los componentes, etc., por lo que será requisito que el modelo brinde soporte para poder realizar esta acción.

- *Soporte de lenguajes de alto nivel para procesamiento:*

Como este modelo está orientado a usuarios con conocimientos específicos de RMN y matemática, y no se exige que tengan amplios conocimientos de programación, los distintos cálculos matemáticos que se realizan a lo largo de un experimento deberían poderse expresar en algún lenguaje de alto nivel comúnmente utilizado por estas personas, como por ejemplo: *MatLab*, *SciLab*, *Octave*, etc.

- *Librerías de procesamiento:*

El procesamiento de una señal implica realizar distintos tipos de cálculos matemáticos que utilizan como entrada a la misma, junto con alguna información extra. Hay muchos cálculos que se utilizan en repetidas ocasiones y para evitar la repetición de código será un requerimiento poder utilizar procedimientos ya programados dentro de otros procedimientos. Para esto, se requieren librerías que agrupen los procedimientos según su funcionalidad y que sean accesibles desde cualquier otro procedimiento.

- *Compatibilidad con el hardware de RMN del cliente:*

Si bien, se busca una generalización de la experimentación en una rama de la RMN, la solución dada en este trabajo deberá brindar un soporte completo al equipo que el cliente provea.

- *Almacenamiento de señales y resultados:*

La representación de un experimento dada por el modelo deberá permitir identificar los resultados de interés del mismo. Es necesario notar que no todas las señales obtenidas serán resultados, ya que pueden ser adquisiciones utilizadas dentro de un experimento que no dan información útil al usuario. Lo mismo sucede con los resultados obtenidos de los procesamientos. Es por esto, que se requerirá distinguir los resultados que brindan la información importante de los utilizados internamente.

3.2 Esquema Conceptual

Este nivel es el que define la base teórica del modelo, por lo que es considerado el más importante y ningún concepto que se deba contemplar debe excluirse. Este esquema dará a las personas con conocimientos en RMN las bases requeridas para poder utilizar el modelo para representar sus experimentos.

3.2.1 Definición de la base estructural

El primer paso a realizar en el desarrollo de este nivel consiste en identificar de manera concreta las clases involucradas en NMR_CORE. Para esto, procederemos a analizar detalladamente cada uno de los requerimientos obtenidos del análisis realizado en la sección anterior.

Se requiere poder representar cualquier experimento de RMN. Esto nos obliga de alguna manera a que exista la clase Experimento. Como se observó en el análisis de las aplicaciones existentes en la sección 3.1.2, un experimento consiste en: realizar una o varias mediciones, en cada una de las cuales se configura el equipo con los parámetros deseados, se ejecuta una secuencia de pulsos y se adquiere una señal. Luego, con las señales obtenidas se realizan distintos cálculos matemáticos para obtener un resultado; dicho resultado es el valor de

interés del experimento. Esta descripción básica de un experimento nos deja dos clases claramente definidas, que son: Medicion y Procesamiento.

Podemos decir entonces, que un experimento está conformado por: mediciones y procesamientos. Para poder representar a un experimento como un conjunto de objetos, utilizaremos la clase abstracta Proceso, la cual generaliza las mediciones y los procesamientos.

Si bien esta definición de experimento es correcta, en la realidad para lograr resultados útiles se recurre a realizar secuencias de experimentos que nos brindan más información de la muestra que se está analizando, por lo que necesitaremos una clase más, que contenga esta secuencia de experimentos. A la misma la llamaremos Proyecto. Para prevenir una posible confusión, vale la pena notar que un experimento de RMN será representado en nuestro modelo por un objeto de tipo Proyecto y no por uno de tipo Experimento. Esto será lo que nos dé la granularidad solicitada por los requerimientos.

Dijimos que un proyecto está compuesto por una serie de experimentos, pero para que esto tengo sentido, los experimentos deben estar ordenados según el orden en que deben ejecutarse. Lo mismo sucede con cada experimento en particular y sus procesos. En la figura 3.1 se puede ver un diagrama de secuencia que muestra cómo se ejecuta, paso a paso, un proyecto.

Por último, definamos la acción que se realiza en cada proceso. Por un lado, una medición tendrá una secuencia de pulsos asociada, que será el "código fuente" de lo que se ejecutará en el equipo. Por otro lado, un procesamiento ejecutará un procedimiento matemático. Esto lo representaremos en el modelo utilizando dos clases: SecuenciaDePulsos, la cual estará relacionada con Medición; y Procedimiento, la cual estará asociada con Procesamiento.

Ya definida la base estructural de NMR_CORE, en la figura 3.2 se da una vista de la misma en forma de diagrama de clases.

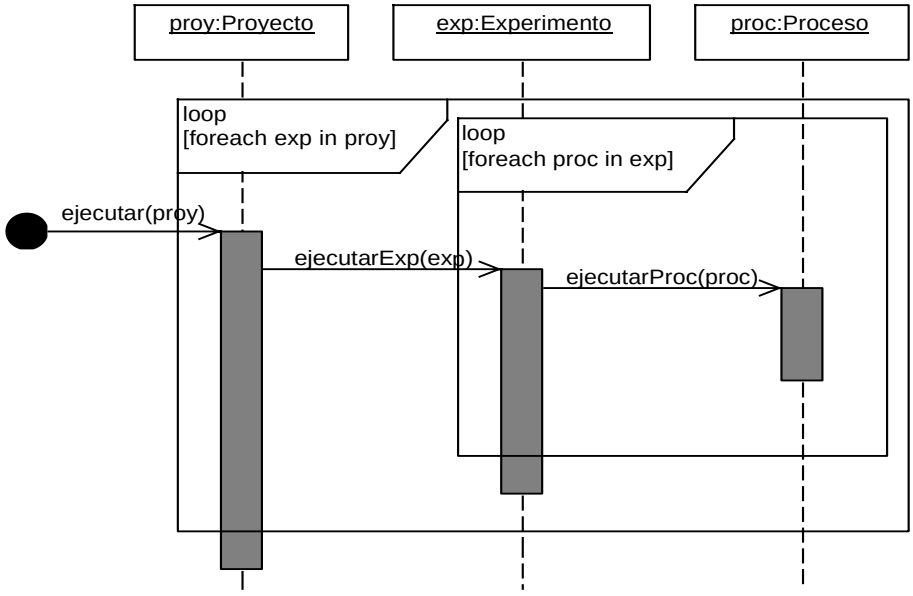

Figura 3.1: Ejecución de un proyecto

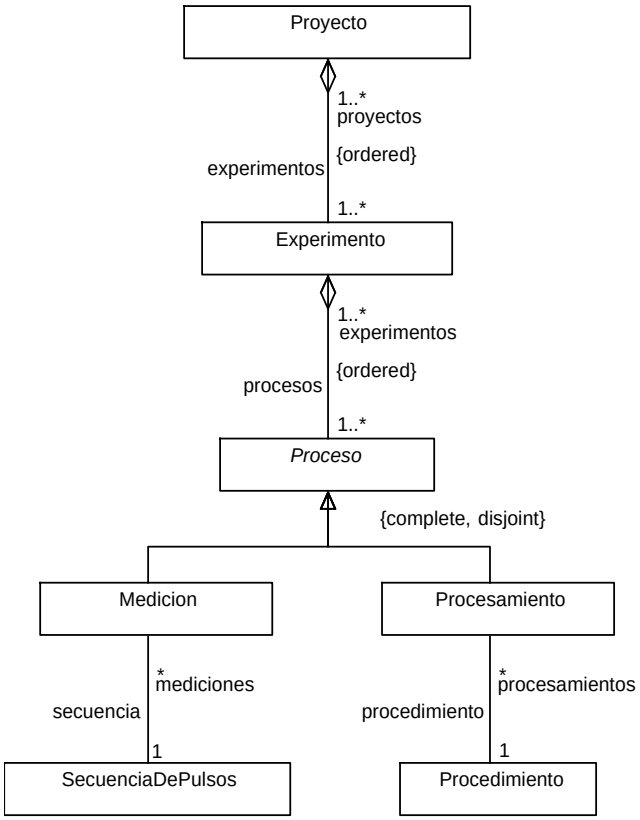

Figura 3.2: Base estructural de NMR_CORE

3.2.2 Parámetros y relaciones principales

Como la clase Medicion será la encargada de ejecutar una secuencia de pulsos en el equipo, requerirá las propiedades de configuración, tanto de la secuencia como del equipo, y un objeto que represente la señal obtenida luego de la ejecución. De esta manera tenemos dos clases nuevas, que serán: Propiedad y Salida. Si bien, en este caso de la medición, la salida es una señal, usaremos como nombre de la clase salida, que es más general y nos permitirá relacionar los procesos con más facilidad.

Es importante aclarar que las propiedades, la secuencia de pulsos y la salida de una medición dependen en su totalidad del equipo con el cual se trabaje. Esto no influye en esta definición de modelo, ya que todos los equipos tienen configuraciones y devuelven una señal. Por este motivo, no será necesario hacer notar esta dependencia.

Ahora bien, una propiedad y una salida son esencialmente lo mismo, ya que tienen un nombre que las identifica y toman un valor (el tipo del valor no nos interesa por ahora), pero difieren en el uso que se le da. Aunque se puede decir que crear dos entidades distintas para representar lo mismo no tiene sentido, para facilitar la comprensión conceptual del modelo dejaremos ambas clases, pero se hará notar esta similitud creando una clase abstracta que las generalice. Esta clase se llamará Parametro.

Representaremos con la clase Valor, al valor que toman las propiedades y la salida. En cómo se relacionan las clases Propiedad y Salida con esta clase es en donde haremos notar la diferencia entre ambas. Por un lado, las propiedades siempre tienen que tener un valor asignado, ya que las mismas definen las características con las que se ejecuta un proyecto. Además, como muchas propiedades representan configuraciones del equipo, puede ocurrir que un valor le corresponda a varias propiedades. Por el contrario, las salidas no tienen ningún valor asignado hasta que se ejecuta el proceso al que pertenecen, y luego de realizar una ejecución se les asocia uno.

En la figura 3.3 se pueden ver las relaciones de las nuevas clases con la clase Medicion.

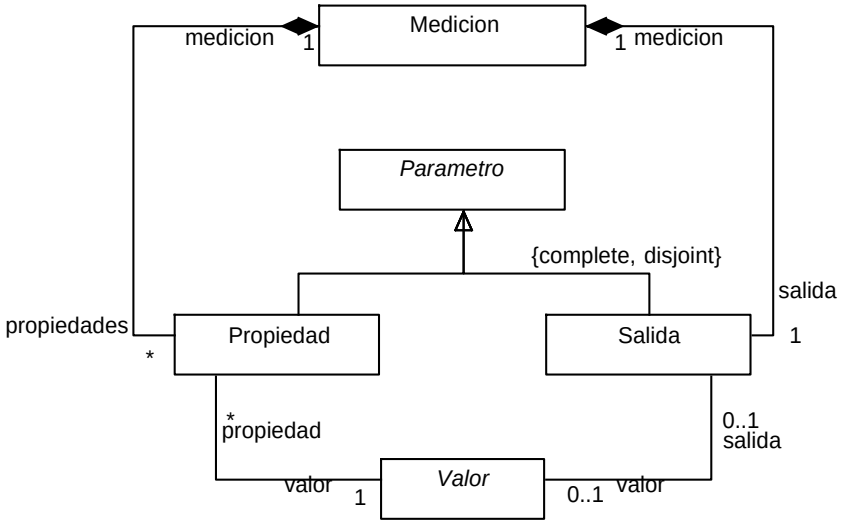

Figura 3.3: Diagrama de clases (no definitivo) en torno a la clase Medicion

Analizando la ejecución de un procesamiento, notamos que éste también requerirá parámetros de entrada, que serán la unión de propiedades del mismo y propiedades y salidas de una o varias mediciones, y alguna representación de las salidas obtenidas del procedimiento. Esto nos muestra que ambos procesos, es decir mediciones y procesamientos, tienen propiedades y salidas (aquí es donde la definición genérica de la entidad salida en la medición se vuelve útil). Un procesamiento, además tiene parámetros de mediciones que los representaremos como una agregación de parámetros. Para reflejar la similitud entre las propiedades y salidas de los procesamientos y las de las mediciones, modificaremos levemente el diagrama de la figura 3.3 extendiéndolo al que se puede observar en la figura 3.4.

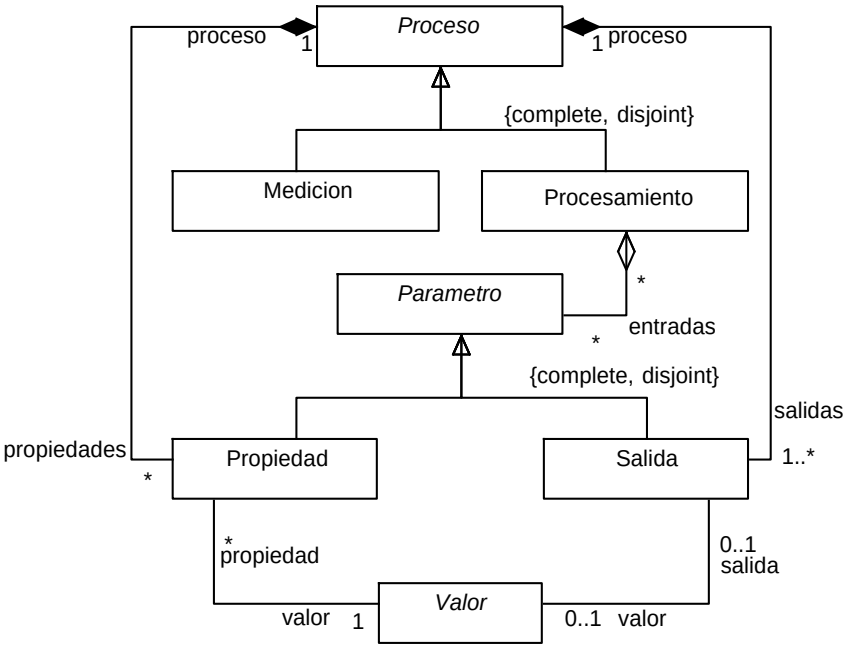

Figura 3.4: Diagrama de clases (final) en torno a la clase Proceso

Una ventaja que nos dan las clases abstractas Proceso y Parametro, es que permiten hacer indistinguibles los parámetros de las mediciones y de los procesamientos entre sí, por lo que al definir las entradas de los procesamientos como una agregación, se nos está dando el capacidad de que, además de poder usar parámetros (ie. propiedades y salidas) de mediciones, podamos usar parámetros de procesamientos como entradas de otros procesamientos. Esta característica le da al modelo la capacidad de particionar un procesamiento complejo en varios más simples ("divide y vencerás").

Como se mencionó anteriormente, las salidas no tienen ningún valor asignado hasta que se ejecuta el proceso al que pertenecen. Esto nos dice que una salida puede dar valores distintos de dos ejecuciones del mismo proyecto, lo que refleja la clara dependencia entre el valor de la salida y la ejecución vigente. Por este motivo deberemos contemplar ejecuciones de un proyecto en nuestro modelo. Para representar las mismas, usaremos una clase Ejecucion. Cada elemento de esta clase estará asociado a un proyecto en particular. Lo más interesante de esta clase, es que será la encargada de almacenar todos los valores de las salidas que se obtengan. Estos valores son los resultados de la ejecución del proyecto. Esta utilidad es representada en el modelo con una relación de composición entre la clase Valor y la clase Ejecucion. Además, notemos en la figura 3.4 que la clase Salida está asociada a lo sumo a un valor, por lo que necesitaremos una relación más entre Valor y Salida que nos sirva para identificar a qué salida pertenecen cada uno de los valores de una ejecución.

En la figura 3.5 se muestra la clase Ejecución y sus respectivas relaciones con las demás clases.

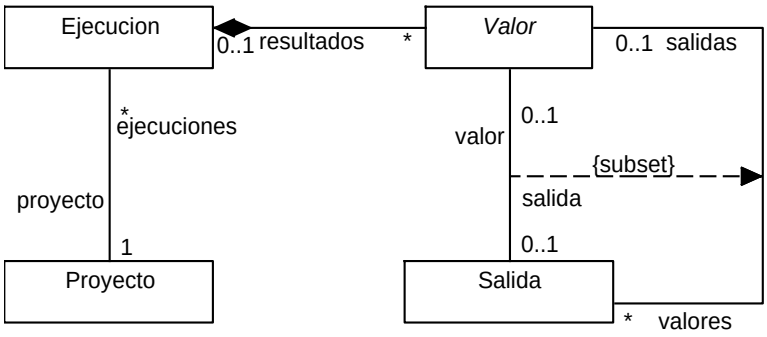

Figura 3.5: Clase Ejecucion y sus relaciones con las otras clases

3.2.3 Modificación de parámetros de procesos en tiempo de ejecución

Se requiere poder cambiar parámetros de los procesos en tiempo de ejecución. Esto se puede traducir en nuestro modelo en una relación entre propiedades y salidas. Podemos decir que el valor de una salida podrá ser asignado a varias propiedades, siempre y cuando las mismas tengan el mismo tipo. Se dará la libertad de que pueda haber varias salidas que asignen una misma propiedad. Esta decisión exige mucha precaución en la definición de las restricciones, ya que una definición poco precisa podría derivar en una reconfiguración no determinista de alguna propiedad y, por ende una ejecución no deseada de un experimento.

En la figura 3.6 se muestra la relación creada para satisfacer este requerimiento.

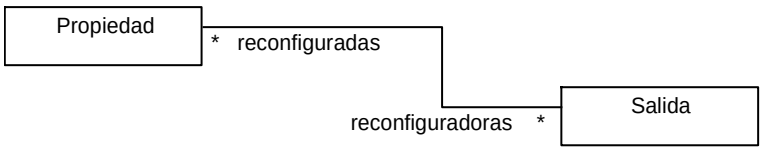

Figura 3.6: Relación que permite la reconfiguración de propiedades

3.2.4 Librerías de procesamiento

Se necesita agrupar los procedimientos en distintos conjuntos, por lo cual se creará la clase LibreriaMatemática, la cual tiene una relación de composición con la clase procedimiento. Esto significa que cada objeto de esta clase contendrá procedimientos, lo que satisface en parte el requerimiento. Además, se nos solicita que los procedimientos puedan usar otros procedimientos en su interior (sin importar si pertenecen o no a la misma librería), por lo que agregaremos una relación entre procedimientos que indicará que cualquier procedimiento puede usar otros procedimientos.

En la figura 3.7 se muestra la nueva clase LibreriaMatemática y las relaciones creadas para satisfacer el requisito solicitado.

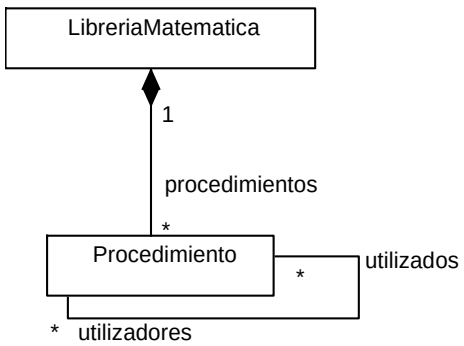

Figura 3.7: Clase LibreriaMatematica y relación que permite la reutilización de *procedimientos*

3.2.5 Experimentos con barrido en parámetros

El último concepto que nos hace falta incorporar a nuestro modelo es el de barrido. Realizar un barrido equivale a ejecutar una secuencia de mediciones, las cuales difieren sólo en los valores de algunas propiedades de la secuencia de pulsos. Una medición con barrido dará como salida un conjunto de señales, las que representaremos en el modelo como una señal que contendrá la unión de las señales resultantes. Esto simplifica la definición de medición en el modelo, ya que no será necesario manejar múltiples salidas de una misma medición.

Una primera solución posible para la representación de barridos es la que se puede ver en la figura 3.8, en la cual se añade una clase Barrido. La misma tiene una relación con Propiedad, la cual deberá ser restringida de manera que sólo se pueda hacer barrido en las propiedades de la secuencia de pulsos de la medición. Además, se añade la clase ValoresBarrido, que contiene los distintos valores que irá tomando la propiedad en cada paso del barrido.

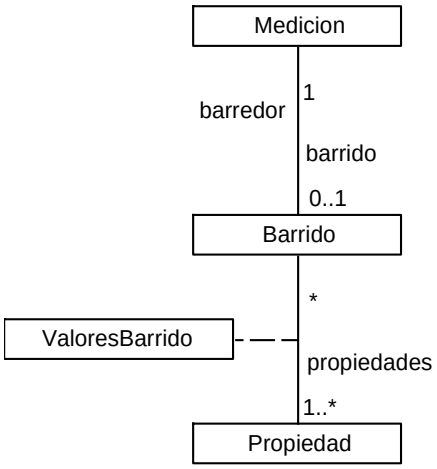

Figura 3.8: Clase Barrido y ValoresBarrido

Si bien la solución dada es válida, como se vio en el análisis del *Condor Lite* (sección 3.1.2), hay ocasiones en las que se desea ejecutar algunos procesos entre cada medición del barrido, lo cual nos obliga a extender esta representación para permitir dicho comportamiento. La extensión consiste en agregar una relación entre la medición con barrido y otro proceso. Esta relación significa que luego de ejecutar la primera medición del conjunto de mediciones que componen un barrido, se deberán ejecutar los procesos subsiguientes hasta llegar al proceso que es extremo de asociación de la relación. Luego se ejecutará la siguiente medición del barrido y nuevamente repetiremos los procesos subsiguientes. Esta secuencia se repetirá hasta ejecutar todas las mediciones del barrido. Veamos un ejemplo para entender este comportamiento.

Ejemplo:

El proyecto *Barrer* contiene un solo experimento llamado *Bucle*. Este experimento contiene 3 procesos. Los 3 procesos se listan a continuación en el orden que están contenidos:

1. *Med*: medición con barrido de 2 pasos en la propiedad *tw* que tiene asociada los valores de barrido 10 y 11. Relacionado con el proceso *Dummy_1*.
2. *Dummy_1*: procesamiento que no tiene ni propiedades ni entradas y tiene una salida.
3. *Dummy_2*: procesamiento que no tiene ni propiedades ni entradas y tiene una salida.

En la figura 3.9 se muestra una pseudo-ejecución del proyecto *Barrer*. La misma tiene como fin esclarecer el concepto de barrido dentro del modelo. Los métodos utilizados y las interacciones que ocurren entre los objetos son a modo ilustrativo y no reflejan la realidad, ya que el proceso de ejecución de un proyecto no entra dentro del alcance de este trabajo.

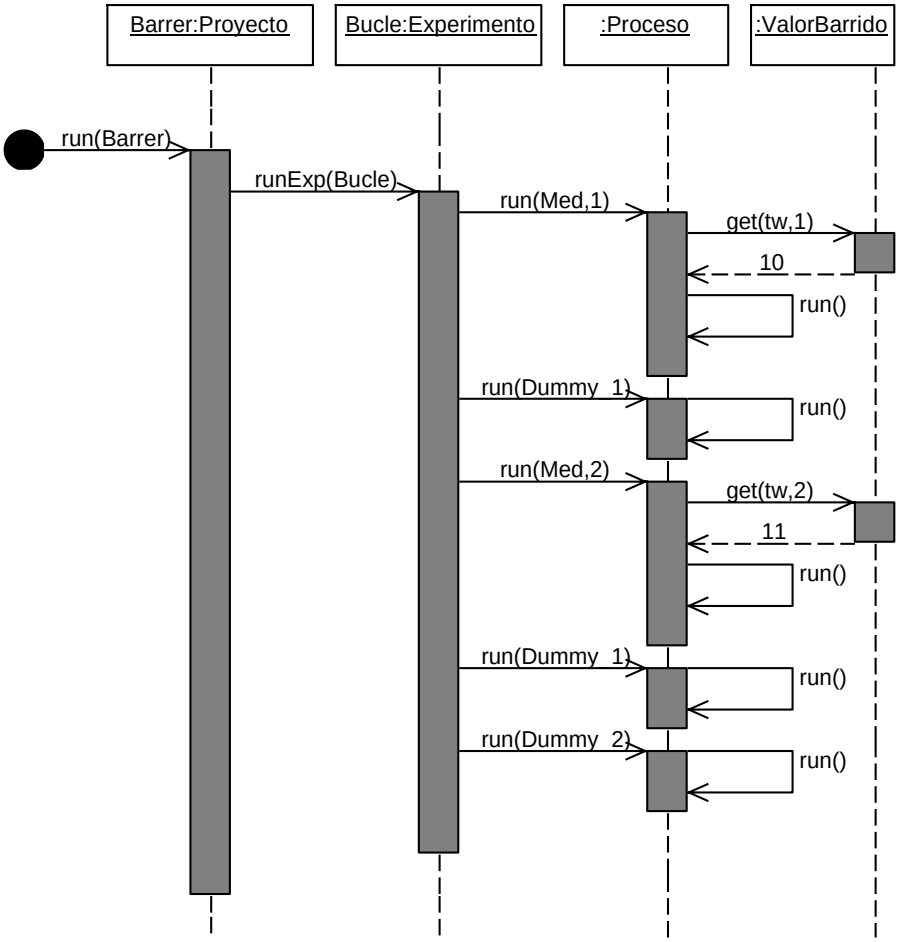

Figura 3.9: Pseudo-ejecución del proyecto *Barrer*

Esta solución introduce complejidad al modelo, por lo que deberá ser contemplada y restringida con sumo cuidado en la definición lógica de los invariantes.

En la figura 3.10 se muestra la relación definida.

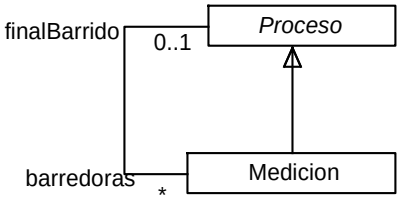

Figura 3.10: Relación que permite *bucles* dentro de la ejecución de un barrido

En la figura 6.2 del apéndice se puede ver el diagrama de clases completo del esquema conceptual del modelo.

3.3 Esquema Lógico

Este esquema le dará a los desarrolladores de aplicaciones de TD-NMR el motor requerido para poder abstraerse de la representación del experimento a realizar y poder focalizarse en las herramientas que le dan al usuario el potencial para manipular el experimento de una manera controlada y obtener los resultados deseados.

Si bien no hay una técnica para transformar un modelo conceptual en uno lógico, generalmente es un paso bastante mecánico que cosiste en caracterizar cada una de las clases con atributos y, una vez hecho esto, fijar las restricciones que tiene el modelo. Esto último lo haremos definiendo invariantes del modelo en el lenguaje natural y, para añadirle formalismo, escribiremos los mismos en el lenguaje OCL.

3.3.1 Agregado de atributos

Cada una de las clases definidas en el esquema conceptual requiere una serie de atributos para ser identificadas y utilizadas por los sistemas de la manera esperada. Es en este nivel de modelado en el cual deben reflejarse estas características. Para esto iremos analizando cada clase en particular adicionando los atributos.

3.3.1.1 Procesos

Comenzaremos analizando esta clase por ser el motor de nuestro modelo. Para identificar los procesos utilizaremos el atributo **nombre** de tipo alfanumérico, el cual será único. Además se añadirá el atributo **descripcion**, que contenga la descripción de la tarea que realiza.

Notemos que la ejecución de un proyecto consiste en ejecutar una secuencia de procesos, por lo que los objetos de este tipo tendrán un atributo, llamado **estado,** que represente el estado en el que están. Se permitirán 3 estados: *EnReposo*, *EnEjecucion* y *Pausado*.

Un proceso por defecto está *EnReposo*. Durante la ejecución de un proyecto, se ejecuta un proceso a la vez siguiendo el orden dado por el experimento, por lo que el estado *EnEjecucion* se utilizará para identificar el proceso activo. Una vez que se termina de ejecutar

un proceso, el mismo pasa nuevamente al estado *EnReposo*. La ejecución de un proceso no podrá ser interrumpida, por lo que surge la pregunta: ¿Para qué es usado el estado *Pausado*? Utilizaremos este estado sólo para las mediciones con barrido. Recordemos que una medición con barrido implica realizar una especie de bucle en la ejecución (explicado en la sección 3.2.5). Entonces veamos a este tipo de medición como un conjunto de procesos. Cuando comienza una ejecución, se coloca a la misma en el estado *EnEjecucion*. Luego de concluir la primera medición, se le asigna el estado *Pausado* y se continúa ejecutando los procesos siguientes normalmente hasta ejecutar el vinculado con el barrido. Luego de ejecutar éste, como se ha explicado anteriormente, se realiza la segunda medición del barrido repitiendo a posteriori lo mismo hasta realizar todas las mediciones del barrido. Una vez realizada la última medición del mismo, se coloca en estado *EnReposo* y se continúa normalmente con la ejecución.

En la figura 3.11 se muestra la clase Proceso con su enumeración de estados EstadoProceso en UML.

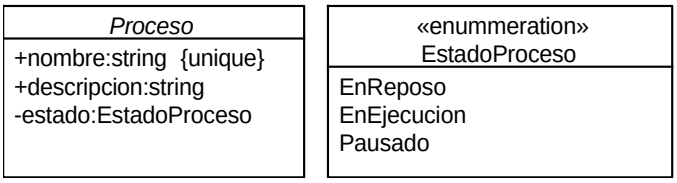

Figura 3.11: Atributos de Proceso y definición de EstadoProceso

Diferenciaremos las mediciones comunes de las con barrido con un atributo booleano **conBarrido**. Para caracterizar las mediciones con barrido nos hará falta saber qué medición del barrido se está ejecutando. Para esto usaremos un atributo entero en la clase medición con el nombre **pasoActualBarrido**, que indicará qué índice de medición del barrido se está ejecutando. En el caso de las mediciones comunes, este atributo no será utilizado.

En la figura 3.12 se muestra la clase Medicion en UML.

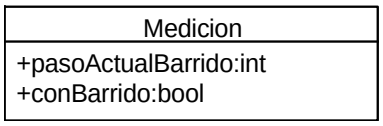

Figura 3.12: Atributos de Medicion

La clase Procedimiento no necesitará atributos adicionales a los que hereda de Proceso.

3.3.1.2 Procedimiento y LibreriaMatemática

Un procedimiento será una función matemática escrita en algún lenguaje de programación de alto nivel. Será necesario especificar en qué lenguaje está escrito cada procedimiento en particular, por lo que tendrá un atributo **tipo**, el cual podrá tomar los valores: *SciLab*, *MatLab* u

Octave. Para agregar un nuevo lenguaje, lo único que hay que hacer es adicionar el nombre del mismo al listado de lenguajes soportados. El código se almacenará en forma de texto en el atributo **codigo** y se agregará un atributo **nombre**. Permitiremos que haya procedimientos con el mismo nombre, siempre y cuando tengan distinto tipo. Esto nos permitirá escribir los mismos procedimientos en distintos lenguajes.

Agruparemos los procedimientos en librerías identificadas por un nombre alfanumérico único.

En la figura 3.13 se muestran las clases LibreriaMatematica, Procedimiento y la enumeración de sus tipos TipoProcedimiento en UML.

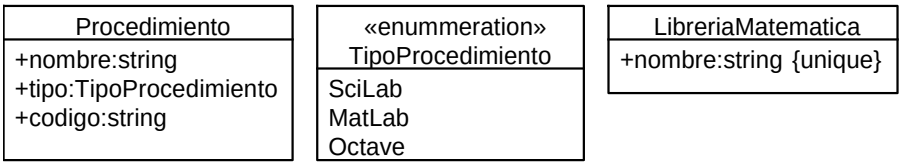

Figura 3.13: Atributos de Procedimiento y LibreriaMatematica, y definición de TipoProcedimiento

3.3.1.3 Barrido y ValoresBarrido

Si bien el barrido trae muchas complicaciones a la hora de manejar estados de procesos, representar un barrido en el modelo será tan simple como decir cuántos pasos tiene utilizando el atributo entero **pasos**.

Cada relación que existe entre un objeto de la clase Propiedad y uno de la clase Barrido tiene asociado un objeto de la clase ValoresBarrido, que está compuesto por el atributo **valores**. Este atributo es un arreglo de objetos de la clase Valor, con longitud igual a la cantidad de pasos del barrido.

En la figura 3.14 se muestran las clases Barrido y ValoresBarrido en UML.

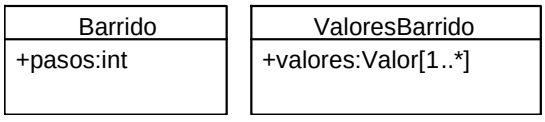

Figura 3.14: Atributos de Barrido y ValoresBarrido

3.3.1.4 Parámetro, Propiedad, Salida y Valor

Los objetos de la clase Parametro son una pieza clave del modelo, ya que se utilizan para representar configuraciones del equipo de RMN, señales resultantes, entradas y salidas de un

procesamiento y las otras propiedades de los procesos. Por este motivo, necesitamos una amplia variedad de tipos que cubran el alcance de la utilidad que se les da.

Daremos cinco tipos de parámetros: *Matriz*, *ArregloDeMatrices*, *Numero*, *Mensaje* y *Booleano*. El nombre de cada tipo da una visión clara de la forma que tendrán los valores, por lo que no será necesario explicarlos uno por uno. El tipo de cada objeto de la clase Parametro será representado en su atributo **tipo**. Usaremos el atributo alfanumérico **nombre** para identificar cada uno de los objetos de esta clase.

En la figura 3.15 se muestra la clase Parametro y la enumeración de sus tipos TipoParametro en UML.

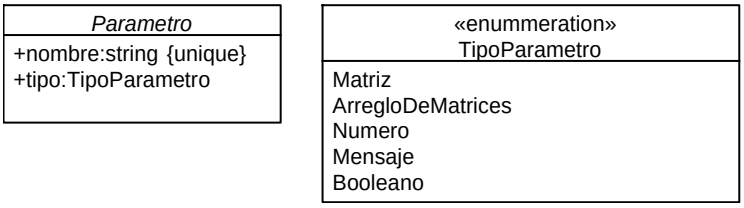

Figura 3.15: Atributos de Parametro y definición de TipoParametro

Por otra parte, la clase abstracta Valor debe poder representar cualquiera de los tipos primitivos de UML y, en adición, arreglos y matrices de dos y tres dimensiones. Para esto, se generaron cinco nuevas clases que permiten representar todos los tipos de parámetros. Estas son: Mensaje, Booleano, Numero, Matriz y ArregloDeMatrices. Para la representación de la clase Matriz se requirió crear una clase auxiliar Arreglo. La definición de estas clases se puede ver en la figura 3.16.

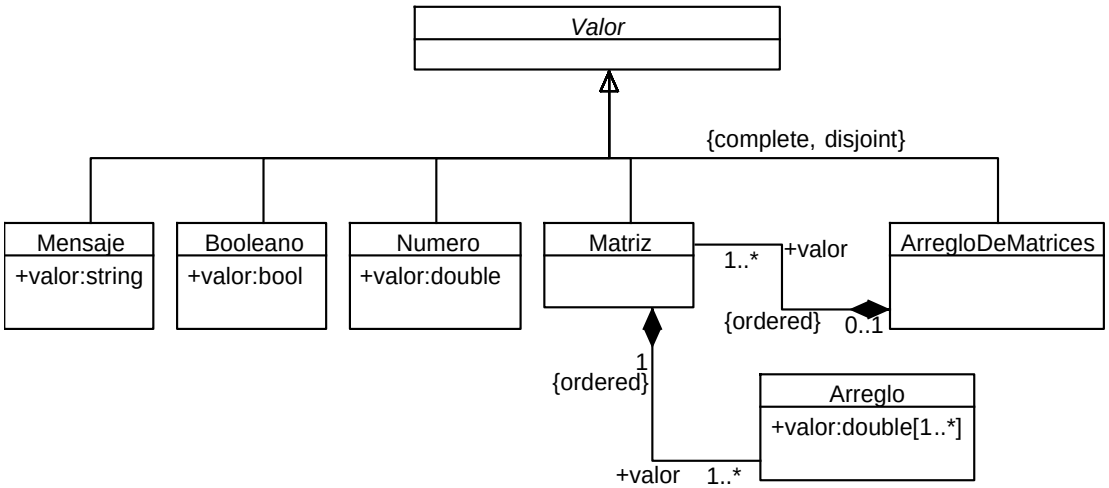

Figura 3.16: Subclases generalizadas por la clase Valor

Generalmente, hay muchas salidas de los procesos que conforman un proyecto que son utilizadas en procesos internos y que no se consideran resultados del experimento RMN, sino más bien resultados intermedios que ayudan a determinar los resultados buscados. Para

distinguir los resultados de un proyecto de los demás, usaremos el atributo booleano **esResultado.**

También agregaremos el atributo **asignada,** que dirá si un objeto de este tipo tiene asociado un valor.

En la figura 3.17 se muestra la clase Salida en UML.

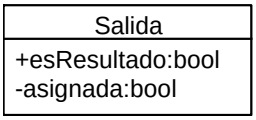

Figura 3.17: Atributos de Salida

La clase Propiedad no necesitará atributos adicionales a los que hereda de Parametro.

3.3.1.5 Ejecución

La clase Ejecucion contiene los resultados de los experimentos realizados, por lo que será importante que tenga un atributo **descripcion** que guarde la información de la muestra que se analizó. Almacenaremos la fecha de inicio y la fecha de fin de la ejecución en formato "dd-MM-aaaa_HH:mm:ss" en los atributos **fechaInicio** y **fechaFin**. Además, tendrá un atributo **activa** que indicará si la ejecución se encuentra en curso o ya finalizó y, cada objeto de esta clase se identificará con un entero que será el valor del atributo **id**.

En la figura 3.18 se muestra la clase Ejecucion en UML.

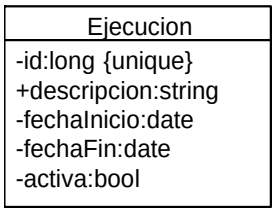

Figura 3.18: Clase Ejecucion

3.3.1.6 Proyecto y Experimento

Un objeto de la clase Proyecto posee un atributo alfanumérico **nombre** que debe ser único entre los de su tipo. Lo mismo sucede con los objetos de la clase Experimento.

Además la clase Experimento tiene un atributo **descripcion**, en el cual deberá contener una breve descripción que caracterice la secuencia de procesos que ejecuta.

El atributo **activo** de la clase Proyecto nos dice si existe alguna ejecución activa del proyecto. En cambio, el atributo **activo** de la clase Experimento identifica al experimento que

está ejecutando sus procesos. Esta diferencia sutil será muy útil para definir las restricciones del modelo.

En la figura 3.19 se muestran las clases Proyecto y Experimento en UML.

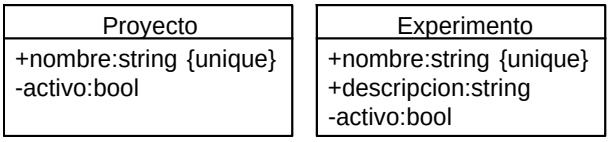

Figura 3.19: Atributos de Proyecto y Experimento

En la figura 6.4 del apéndice se puede ver el diagrama de clases completo del esquema lógico del modelo.

3.3.2 Restricciones

El diagrama UML de NMR_CORE da una visión de los distintos objetos involucrados en un experimento de RMN, los atributos que los definen y la forma en cómo se relacionan. Hay distintos invariantes que debería cumplir el modelo para poder garantizar el uso correcto de la estructura. Estos invariantes no se pueden expresar de manera gráfica, por lo que debimos buscar una alternativa. Si bien la definición en el lenguaje natural de estos invariantes es necesaria, para darle formalismo a la definición del modelo también los escribiremos en el lenguaje OCL, visto en la sección 2.4.

Invariantes relacionados con la clase Proceso

1) A lo sumo un proceso puede estar en ejecución

```
Proceso.allInstances() →
    (select(p|p.estado = EstadoProceso::EnEjecucion) → size()) <= 1
```

Invariantes relacionados con la clase Medicion

2) Una medición tiene una y sólo una salida

```
Medicion.allInstances() → forAll(m|m.salidas → size() = 1)
```

3) Las propiedades de la secuencia de pulsos de una medición son parte de las propiedades de dicha medición

```
Medicion.allInstances() → forAll(m|
    m.propiedades → includesAll(m.secuencia.propiedades))
```

4) El proceso asociado a una medición por el extremo de asociación **finalBarrido** debe ser parte del mismo experimento que dicha medición, y debe estar en un orden posterior

```
Medicion.allInstances() → forAll(m| m.conBarrido  implies
    m.experimientos → forAll(e|
    (e.procesos → includes(m.finalBarrido)) and
    (e.procesos → indexOf(m.finalBarrido)) >=
    (e.procesos → indexOf(m))))
```

5) El paso actual de un barrido no puede ser mayor que cantidad de pasos del barrido

```
Medicion.allInstances() → forAll(m|m.conBarrido implies
    (m.pasoActualBarrido > 0 and
     m.pasoActualBarrido <= m.barrido.pasos))
```

6) Solo las mediciones con barrido pueden tomar el estado pausado

```
Proceso.allInstances() → forAll(p|
    p.estado = EstadoProceso::Pausado implies
    (p.oclIsTypeOf(Medicion) and m.oclAsType(Medicion).conBarrido))
```

7) Una medición con barrido debe tener fijado **finalBarrido**, **barrido** y **pasoActualBarrido**

```
Medicion.allInstances() → forAll(m|m.conBarrido implies
    (not(m.finalBarrido.oclIsUndefined()) and
    not(m.barrido.oclIsUndefined()) and
    not(m.pasoActualBarrido.oclIsUndefined())))
```

8) Si la salida de una medición está asignada y el valor del atributo **pasoActualBarrido** es menor que la cantidad de pasos del mismo, entonces la medición no puede estar en estado de reposo

```
Medicion.allInstances() → forAll(m|(m.conBarrido and
    (m.salidas → forAll(s|s.asignada)) and
    m.pasoActualBarrido < m.barrido.pasos)
    implies m.estado <> EstadoProceso::EnReposo)
```

Invariantes relacionados con la clase Procesamiento

9) Una salida de un procesamiento no puede ser entrada del mismo

```
Procesamiento.allInstrances() → forAll(p|
    p.entradas → excludesAll(p.salidas))
```

Invariantes relacionados con la clase Ejecucion

10) A lo sumo una ejecución puede estar activa

```
Ejecucion.allInstances() → select(e|e.activa) → size() <= 1
```

11) Si no hay ninguna ejecución activa, ninguna de las salidas de los procesos existentes debe estar asignada.

```
(Ejecucion.allInstances() → select(e|e.activa) → size() = 0) implies
    (Salida.allInstances() → forAll(s|not(s.asignada)))
```

12) La ejecución de un proyecto debe obtener al menos un resultado por cada una de las salidas de los procesos de los experimentos del proyecto en cuestión

```
Ejecucion.allInstances() → forAll(e|not(e.activa) implies
    (e.proyecto.experimentos → forAll(exp|exp.procesos →
     forAll(p|p.salidas → forAll(s|s.valores →
     exists(v|e.resultados → includes(v)))))))
```

Invariantes relacionados con la clase Procedimiento

13) Un procedimiento no puede utilizarse a si mismo

```
Procedimiento.allInstances() → forAll(p| p.utilizados → excludes(p))
```

14) No puede haber más de un procedimiento del mismo tipo, con el mismo nombre

```
Procedimiento.allInstances() →
    forAll(p1,p2|(p1.nombre = p2.nombre and p1.tipo = p2.tipo) implies
    p1 = p2)
```

15) Un procedimiento sólo puede utilizar procedimientos que tengan el mismo tipo que éste

```
Procedimiento.allInstances() → forAll(p1|
    p1.utilizados → forAll(p2|p2.tipo = p1.tipo))
```

Invariantes relacionados con la clase Salida

16) Los valores de una salida se corresponden con el tipo de la misma

```
Salida.allInstances() → forAll(s| s.valores → forAll(val|
    (s.tipo=TipoParametro::Numero implies val.oclIsTypeOf(Numero))
    and
    (s.tipo=TipoParametro::Mensaje implies val.oclIsTypeOf(Mensaje))
    and
    (s.tipo=TipoParametro::Booleano implies val.oclIsTypeOf(Booleano))
    and
    (s.tipo=TipoParametro::Matriz implies val.oclIsTypeOf(Matriz))
    and
    (s.tipo = TipoParametro::ArregloDeMatrices implies
     val.oclIsTypeOf(ArregloDeMatrices))))
```

17) Una salida está asignada si y sólo si tiene un valor asociado

```
Salida.allInstances() → forAll(s|s.asignada implies
    not(s.valor.oclIsUndefined()))
Salida.allInstances() → forAll(s|
    not(s.valor.oclIsUndefined()) implies s.asignada)
```

Invariantes relacionados con la clase Barrido

18) Las propiedades de un barrido, deben ser propiedades de la secuencia de pulsos de la medición asociada al mismo

```
Barrido.allInstances() → forAll(b|
    b.barredora.secuencia.propiedades → includesAll(b.propiedades))
```

19) Un barrido debe tener como mínimo un paso

```
Barrido.allInstances() → forAll(b|b.pasos > 0)
```

20) La cantidad de valores de barrido asociados a cada propiedad del barrido debe ser igual a la cantidad de pasos del barrido

```
Barrido.allInstances() → forAll(b|
    b.ValoresBarrido → forAll(v|v.valores → size() = b.pasos))
```

Invariantes relacionados con la clase Propiedad

21) Todos los valores de barrido que toma una propiedad son del mismo tipo que la propiedad

```
Propiedad.allInstances() → forAll(p|p.ValoresBarrido →
    forAll(v|v.valores → forAll(val|
    val.oclType() = p.valor.oclType())))
```

22) Las propiedades no pueden ser de tipo *Matriz*, ni de tipo *ArregloDeMatrices*

```
Propiedad.allInstances() → forAll(p|
    p.tipo <> TipoParametro::Matriz and
    p.tipo <> TipoParametro::ArregloDeMatrices)
```

23) El valor de una propiedad se corresponde con el tipo de la misma

```
Propiedad.allInstances() → forAll(p|
    (p.tipo = TipoParametro::Numero implies
     p.valor.oclIsTypeOf(Numero)) and
    (p.tipo = TipoParametro::Mensaje implies
     p.valor.oclIsTypeOf(Mensaje)) and
    (p.tipo = TipoParametro::Booleano implies
     p.valor.oclIsTypeOf(Booleano)))
```

24) Las propiedades pueden ser reconfiguradas sólo por salidas del mismo tipo

```
Propiedades.allInstances() → forAll(p|
    p.reconfiguradoras → forAll(s|s.tipo=p.tipo))
```

25) Una propiedad puede ser configurada por a lo sumo una salida de un mismo proceso

```
Propiedades.allInstances() → forAll(p|p.reconfiguradoras →
    forAll(s1,s2|s1.proceso=s2.proceso implies s1=s2))
```

Invariantes relacionados con la clase Experimento

26) A lo sumo hay un experimento activo

```
Experimento.allInstances() → select(e|e.activo) → size() <= 1
```

27) Si un experimento no está activo, todos sus procesos están en estado de reposo

```
Experimento.allInstances() → forAll(e|not(e.activo) implies
    (e.procesos → forAll(p|p.estado=EstadoProceso::EnReposo)))
```

28) Si hay más de una medición con barrido dentro de un experimento, los rangos de procesos que utilizan cada uno de los barridos no pueden solaparse, es decir, la intersección uno a uno entre estos rangos debe ser: nula ó igual a uno de los dos rangos.

```
Experimento.allInstances() → forAll(e| e.procesos →
    forAll(m1:Medicion,m2:Medicion|
    (m1 <> m2 and m1.conBarrido and m2.conBarrido) implies
    if (e.procesos → indexOf(m1)) > (e.procesos → indexOf(m2)) then
     (e.procesos → indexOf(m1.finalBarrido)) <=
     (e.procesos → indexOf(m2.finalBarrido)) or
     (e.procesos → indexOf(m1)) >
     (e.procesos → indexOf(m2.finalBarrido))
    else
     (e.procesos → indexOf(m2.finalBarrido)) <=
     (e.procesos → indexOf(m1.finalBarrido)) or
     (e.procesos → indexOf(m2)) >
     (e.procesos → indexOf(m1.finalBarrido))
    endif))
```

29) Si un experimento está activo, existe un proyecto activo que lo contiene

```
Experimento.allInstances() → forAll(e|e.activo implies
    (e.proyectos → one(p|p.activo)))
```

30) Los procesos de un experimento se ejecutan en orden

```
Experimento.allInstances() → forAll(e|e.procesos →
    forAll(p1,p2|
    ((e.procesos → indexOf(p1)) < (e.procesos →indexOf(p2)) and
     p2.estado = EstadoProceso::EnEjecucion) implies
    p1.salidas → forAll(s|s.asignada)))
```

Para ejecuciones sin barrido el invariante anterior es suficiente para que se ejecuten en orden. Como nuestro modelo soporta mediciones con barrido, para garantizar el orden de ejecución de los mismos deberemos agregar los dos nuevos invariantes dados a continuación.

Si un proceso está en estado de ejecución, entonces los procesos subsiguientes al mismo no tienen asignados ningún valor en sus salidas

```
Experimento.allInstances() → forAll(e|e.procesos →
    forAll(p1,p2|
    ((e.procesos → indexOf(p1)) < (e.procesos →indexOf(p2)) and
     p1.estado = EstadoProceso::EnEjecucion) implies
    p2.salidas → forAll(s|not(s.asignada))))
```

Mientras una medición con barrido esta pausada, no se pueden ejecutar los procesos subsiguientes al asociado con la medición por medio de **finalBarrido**

```
Medicion.allInstances() → forAll(m|
    (m.conBarrido and m.estado=EstadoProceso::Pausado) implies
    m.experimentos → forAll(e|e.activo implies
    (e.procesos → forAll(p|
    (e.procesos → indexOf(p)) > (e.procesos → indexOf(m.finalBarrido))
    implies p.estado=EstadoProceso::EnReposo)))
```

Invariantes relacionados con la clase Proyecto

31) Dentro de un proyecto, no puede haber dos experimentos que contengan el mismo proceso

```
Proyecto.allInstances() → forAll(p|p.experimentos → forAll(e1,e2|
    e1 <> e2 implies
    (e1.procesos → excludesAll(e2.procesos))))
```

32) Los experimentos de un proyecto se ejecutan en orden

```
Proyecto.allInstances() → forAll(p|p.experimentos → forAll(e1,e2|
    ((p.procesos → indexOf(e1)) < (p.procesos → indexOf(e2)) and
     e2.activo) implies
    (not(e1.activo) and
     e1.procesos → forAll(proc|proc.salidas → forAll(s|s.asignada))))
```

33) Un proyecto esta activo si y sólo si hay una ejecución activa del mismo

```
Ejecucion.allInstances() → forAll(e|
    e.activa implies e.proyecto.activo)
Proyecto.allInstances() → forAll(p|
    p.activo implies (p.ejecuciones → one(e|e.activa)))
```

4 Análisis

Habiendo concluido la etapa de diseño de NMR_CORE, analizaremos la funcionalidad que brinda este modelo en la experimentación práctica. Para esto, seleccionamos dos experimentos ampliamente utilizados en la RMN de baja resolución: ajuste de frecuencia e inversión-recuperación, y los representamos utilizando objetos del modelo. Luego, estudiamos el comportamiento de dichos objetos a lo largo de una ejecución de cada uno de los experimentos antes mencionados.

En los ejemplos analizados se utilizaron datos reales. Las propiedades de los procesos que dependen de equipo a utilizar se corresponden con las del espectrómetro SLK-200, proporcionado por el cliente, que se describe a continuación.

4.1 Espectrómetro utilizado: SLK-200

El SLK-200 es un espectrómetro de TD-NMR ampliamente utilizado para analizar la composición de semillas y aceites en los laboratorios de control de calidad a nivel nacional e internacional. Este equipo puede detectar la presencia y concentración de aceite y humedad en semillas de girasol y otras oleaginosas.

Entre los módulos electrónicos que componen este equipo, podemos encontrar: un programador de pulsos (PP) con capacidad de manejar 16 salidas, un sintetizador digital directo (DDS) que permite sintetizar una señal combinando dos valores de frecuencia y cuatro de fase, dos canales de recepción con una resolución de 12 bits y un módulo encargado de la adquisición y el almacenamiento de la señal (AD) con dos buffers de 128 KB.

Para poder realizar una medición en este espectrómetro, se requieren: una secuencia de pulsos a ejecutar (codificada en instrucciones del PP), los dos valores de frecuencia a utilizar en la secuencia de pulsos (que rondan en los 11.5 Mhz) y los cuatro valores de fase, y la frecuencia de muestreo del AD.

4.2 Casos de estudio

Los dos experimentos que analizaremos fueron introducidos en la descripción de la plataforma *Condor Lite* en la sección 3.1.2. Los mismos fueron seleccionados del conjunto de experimentos en TD-NMR existentes ya que cubren todos los conceptos que generalmente pueden encontrarse en el diseño de este tipo de experimentos.

4.2.1 Ajuste de frecuencia

Realizar un ajuste de frecuencia consta de 4 pasos: configurar el equipo, medir una FID, realizar un procesamiento de la señal obtenida y reconfigurar la frecuencia de los pulsos de RF de las secuencias de pulso a utilizar.

En la figura 4.1 podemos ver la estructura del Proyecto *afProy*, que realiza este experimento de RMN. Notemos que contiene sólo al Experimento *afExp*, el cual tiene 2 procesos: la Medicion *fidMed* y el Procesamiento *ftProc*.

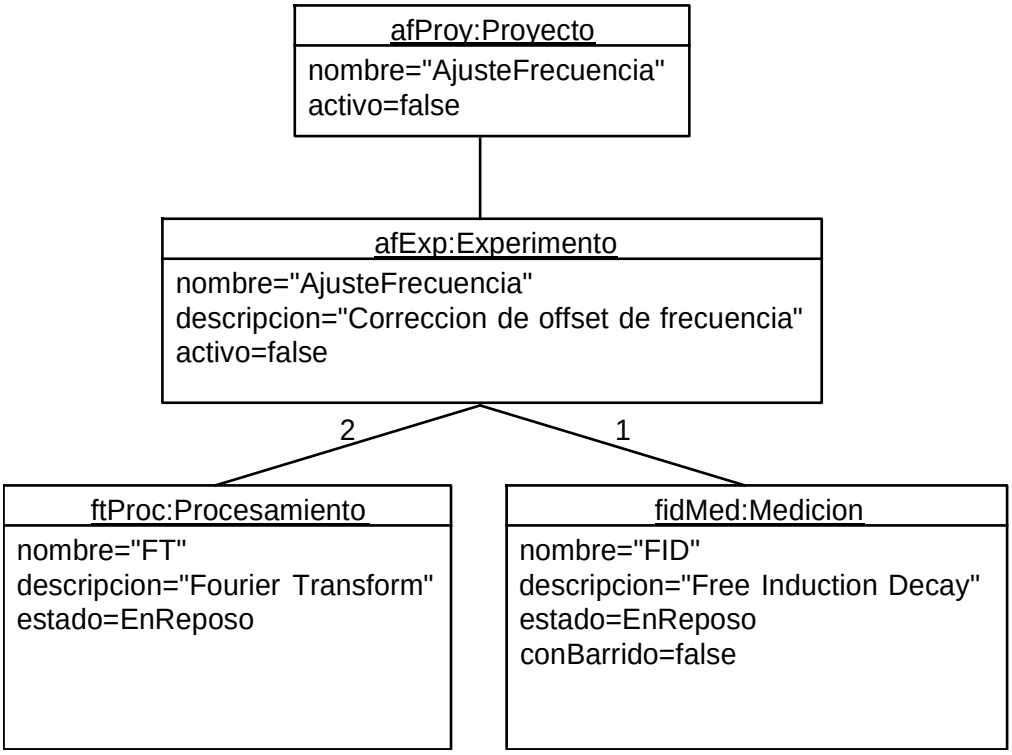

Figura 4.1: Estructura del objeto de tipo Proyecto *afProy*

Ahora hay que agregarle los objetos de las clases Propiedad y Salida asociados a cada uno de los procesos, además de un objeto de tipo SecuenciaDePulsos a *fidMed* y un objeto de tipo Procedimiento a *ftProc*.

Por el equipo que se está utilizando (descripto en la sección 4.1.2), las propiedades básicas que debe tener una medición son: dos frecuencias, cuatro fases y una frecuencia de muestreo. *fidMed* ejecutará la secuencia de pulsos *fidSec* y tiene como salida una señal. *fidSec* es la secuencia vista en la sección 2.1 para adquirir una FID, la cual requiere una propiedad que indique la duración del pulso de RF de $\pi/2$. En la figura 4.2 se puede ver al objeto *fidMed* en forma de diagrama de objetos.

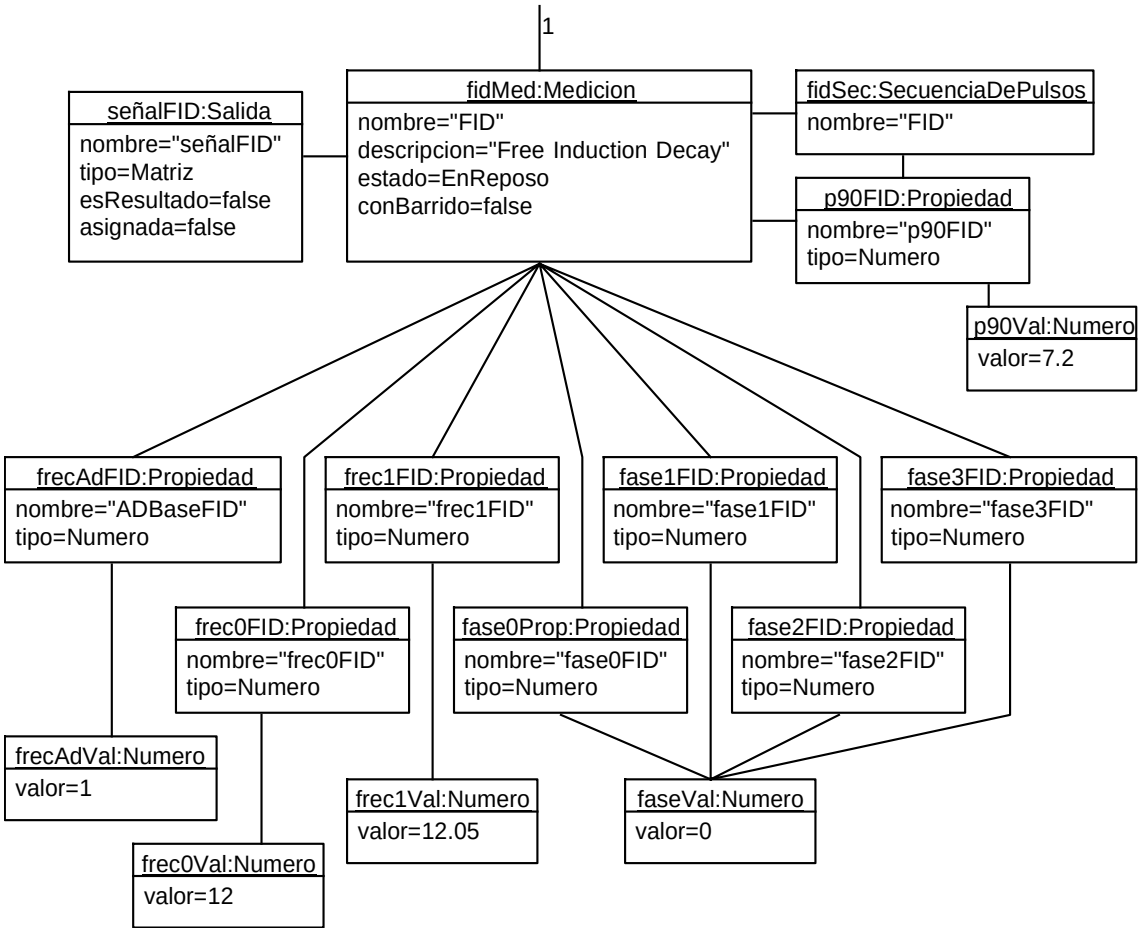

Figura 4.2: Diagrama de objetos de la Medicion *fidMed*

Es necesario aclarar que la omisión del valor del atributo código del objeto *fidSec* en el diagrama de objetos de la figura 4.2 es intencional, ya que no se considera necesario mostrarlo para los fines prácticos del modelo. Existen distintas maneras de representar dicho código, las cuales no entran en el alcance de este trabajo. Nos conformaremos con suponer que la ejecución de la secuencia de pulsos con nombre "FID" ejecuta la medición denominada como FID explicada en la sección 2.1.

Ahora pasemos a representar el objeto *ftProc*. Este procesamiento toma como entradas las frecuencias utilizadas en la medición y la señal devuelta por la misma y, luego de ejecutar el procedimiento devuelve las frecuencias de resonancia ajustadas. Si bien existen distintos métodos matemáticos para realizar este cálculo, en la mayoría de los casos se utiliza la transformada de Fourier para obtener la señal en el dominio de la frecuencia. Los métodos que permite utilizar el *Condor Lite* son FFT (*Fast Fourier Transform*) y DFT (*Discrete Fourier Transform*).

Supongamos que se quiere utilizar FFT. Este procedimiento, además de tomar las frecuencias y la señal, toma un entero, que se conoce como ZeroFilling. Por este motivo, *ftProc* deberá tener una propiedad propia, la que llamaremos *zFillingFT*. *ftProc* tiene como salidas *frec0FT* y *frec1FT*.

En la figura 4.3 se muestra el diagrama de objetos del procesamiento *ftProc*.

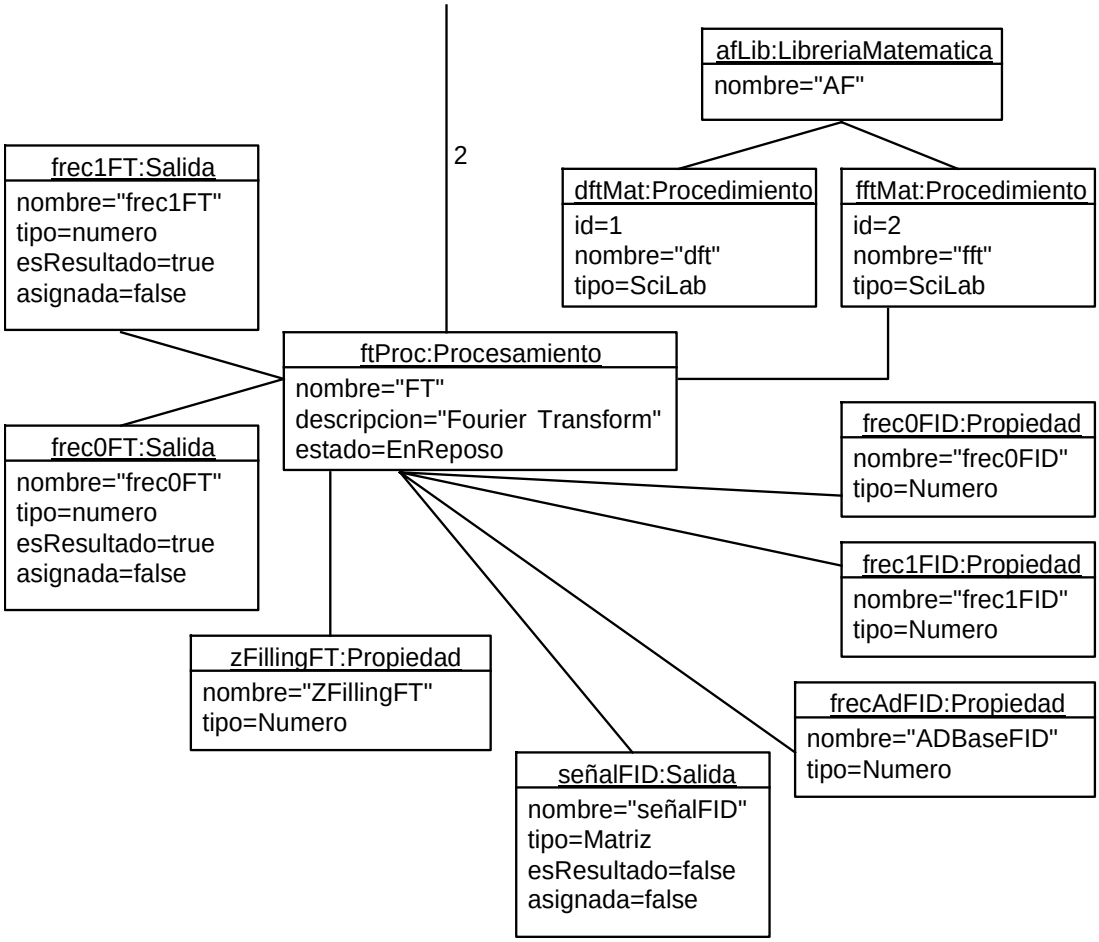

Figura 4.3: Diagrama de objetos del Procesamiento *ftProc*

El fin de ejecutar este experimento es configurar las frecuencias en los valores que corresponden, por lo que relacionaremos cada una de las salidas de *ftProc* con el valor que frecuencia de la medición *fidMed* que le corresponde. Este enlace se muestra en la figura 4.4.

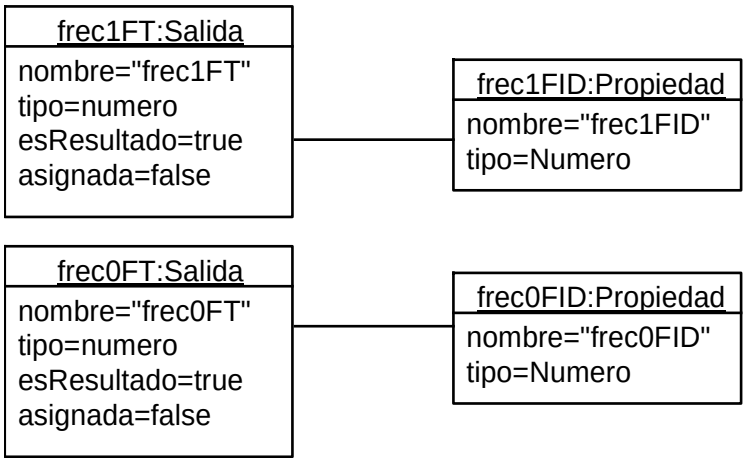

Figura 4.4: Relaciones de *reconfiguración* de las frecuencias

Notemos en el diagrama de la figura 4.1 que *afProy* se encuentra inactivo. Esto quiere decir, que no hay ningún objeto de la clase Ejecución relacionado al mismo que se encuentre activo. Para entender más a fondo NMR_CORE, veamos los cambios que ocurren en los objetos a lo largo de una ejecución de *afProy*.

Debido a las restricciones impuestas al modelo, el primer paso para ejecutar *afProy* consiste, solamente, en crear un objeto de tipo Ejecucion asociado al mismo. Al hacer esto, automáticamente *afProy* pasa a estar activo (por la restricción 33). En este instante, el único experimento que puede ser activado es *afExp* (por la restricción 32). El diagrama de la figura 4.5 captura los cambios ocurridos en los objetos, una vez realizada la activación de *afexp*.

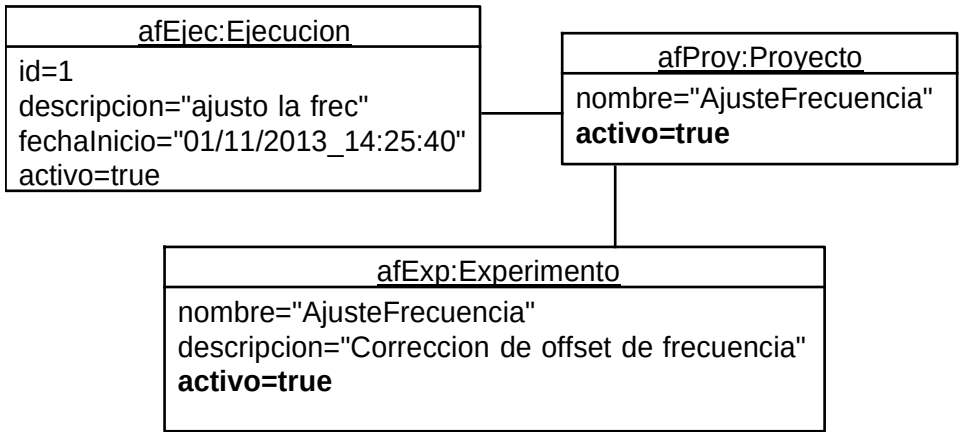

Figura 4.5: Cambio de estado de *afProy* y *afExp* al crearse *afEjec*

Luego de esto, se ejecuta el primer proceso de *afExp* (por la restricción 30), que es *fidMed*. Si bien, ejecutar una medición implica realizar muchas tareas, las mismas no se ven reflejadas en el modelo. Sólo varía el estado de *fidMed*, como se ve en la figura 4.6.

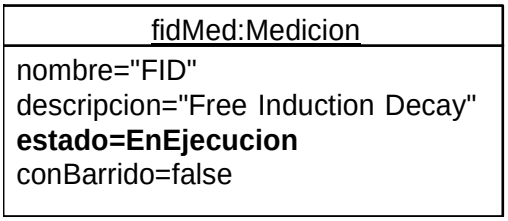

Figura 4.6: Objeto *fidMed* durante su ejecución

Al concluir la ejecución de *fidMed*, se le asigna a la salida *señalFID* un valor de tipo Matriz (por la restricción 16), como se puede ver en la figura 4.7. *señalFIDVal* es una señal del equipo descrito en la sección 4.1.2, por lo que está compuesta por: un arreglo de tiempos, un arreglo con el canal real de la señal y otro con el canal imaginario. No se muestra el atributo **valor** de cada uno de los arreglos debido a su longitud. Notemos también que este valor está relacionado con la ejecución *afEjec*.

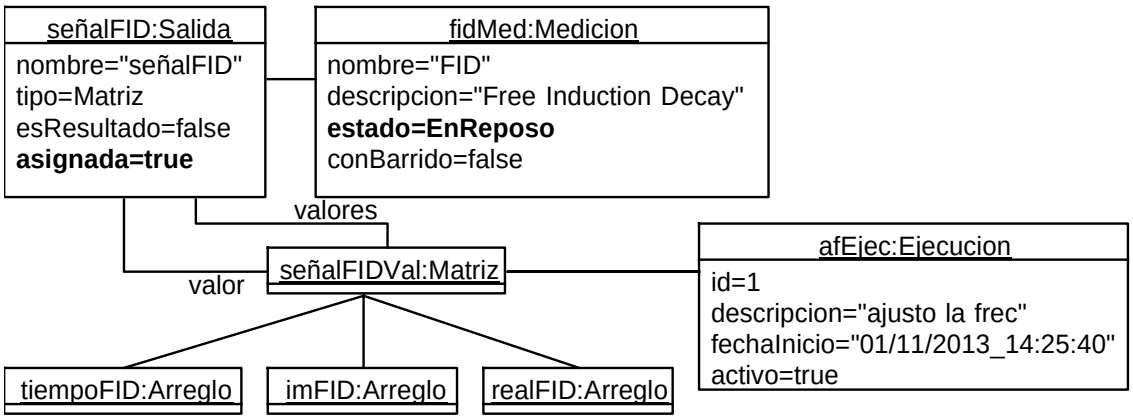

Figura 4.7: Objetos modificados luego de que *fidMed* vuelve al estado *EnReposo*

El paso subsiguiente, consiste en ejecutar el procesamiento *ftProc* (por la restricción 30). Analicemos en detalle, el momento puntual en el cual finaliza la ejecución de *ftProc*. En ese instante, se le asigna un valor a **frec0FT** y otro a **frec1FT**, lo que automáticamente hace que las propiedades **frec0FID** y **frec1FID** se asocien a estos valores, respectivamente. Además se asocia estos valores a la ejecución *afEjec*. Esta situación se muestra en la figura 4.8.

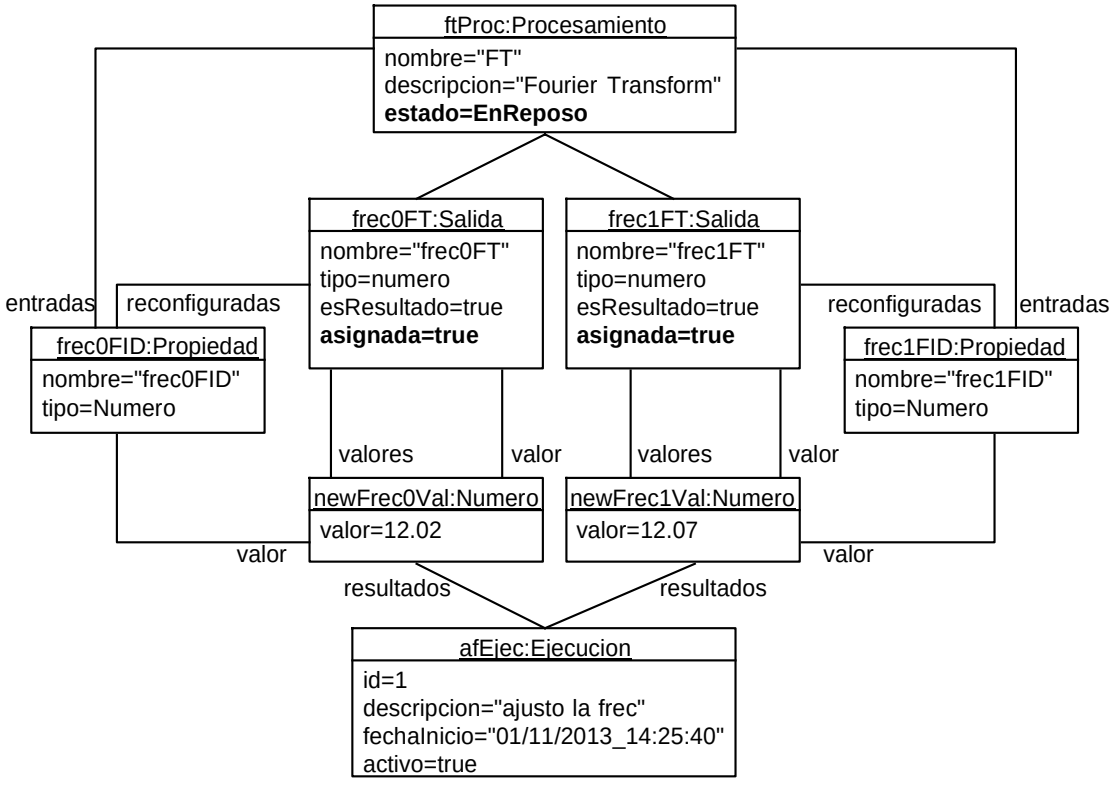

Figura 4.8: Objetos modificados luego de que *ftProc* vuelve al estado *EnReposo* (antes de que se reconfiguren los valores de frecuencia)

Notemos que si bien podríamos concluir del diagrama de la figura 4.8 que al estar todos los procesos en estado *EnReposo* y con todas las salidas asignadas, ya ha finalizado la ejecución, la restricción 11 del modelo nos lo impide, ya que es necesario eliminar todas las asociaciones

con extremo de asociación **valor** de las salidas, para que el atributo **asignada** de todas las salidas sea falso (por la restricción 17), y por lo tanto, se pueda dar por finalizada la ejecución. Aunque esta restricción implica realizar un "trabajo extra" antes de desactivar la ejecución, nos garantiza que al comenzar una nueva ejecución ninguna salida figurará como asignada, lo que de ocurrir permitiría la ejecución desordenada de los procesos, y, por ende, derivaría en una inconsistencia en el modelo. El estado final de los objetos una vez concluida la ejecución se puede ver en la figura 4.9.

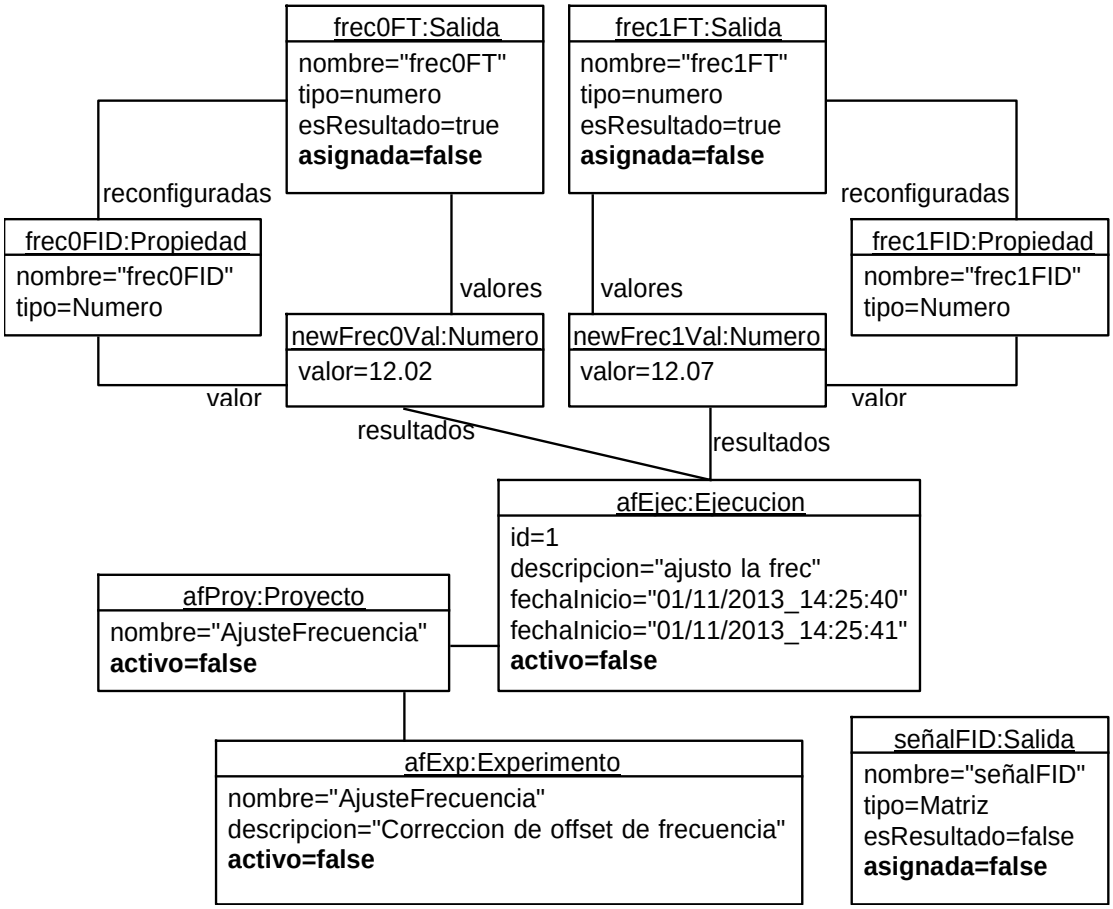

Figura 4.9: Estado final de los objetos al finalizar la ejecución de *afProy*

La ejecución analizada a lo largo de esta sección nos permite observar en acción varias de las restricciones de integridad impuestas al modelo. En el ejemplo de la sección siguiente, se mostrarán algunas otras más.

4.2.2 Inversión-Recuperación

Este experimento consiste en realizar un barrido sobre la propiedad que representa el tiempo *t* en la secuencia de pulsos descrita en la sección 2.1.3.4. El barrido sobre *t* tiene 10 pasos, los cuales deben distribuirse en el intervalo [1s, 20s] con una distribución logarítmica.

Este experimento será representado en NMR_CORE con el objeto de tipo Proyecto *InvRec*, el cual tendrá dos objetos de la clase Experimento: *afExp*, definido en el ejemplo de ajuste de frecuencia de la sección 4.2.1, e *invRecExp*. Este útlimo, constará de dos procesos: uno de tipo Medicion *tBarrierMed* y en otro de tipo Procesamiento *ft2Proc*. En la figura 4.10 se puede ver la representación de *InvRec*.

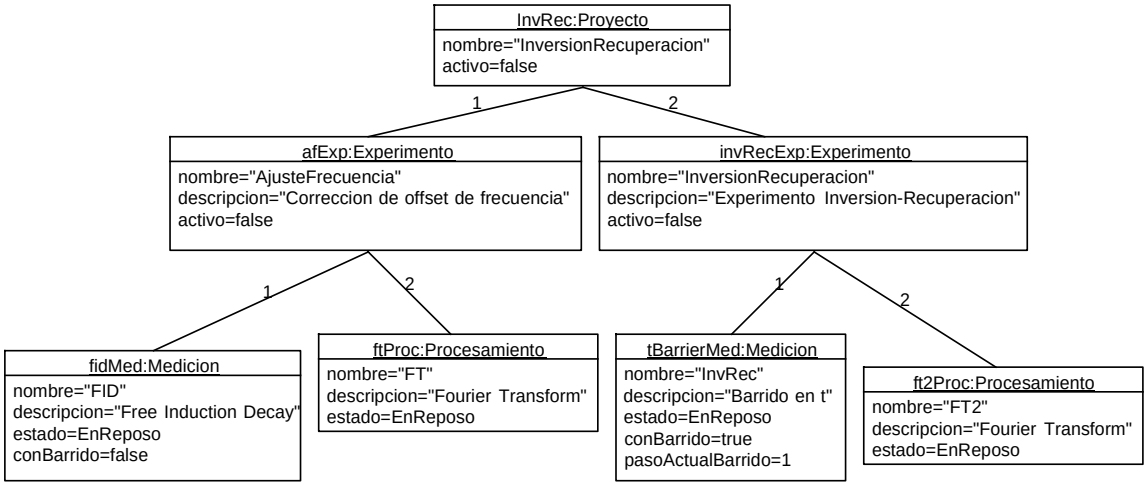

Figura 4.10: Estructura del objeto de tipo Proyecto *InvRec*

En este experimento, luego de que se ejecuta cada medición del barrido, deseamos que se ajuste la frecuencia. Además, antes de ejecutar *tBarrierMed*, necesitamos ajustar por primera vez la frecuencia. Por este motivo, será necesario vincular las salidas de *ftProc* con las frecuencias de *tBarrierMed.* Aunque estén en distintos experimentos, esto está permitido en el modelo.

Como una medición de barrido ejecuta, luego de cada medición, los procesos subsiguientes (por la restricción 30) hasta encontrarse con el que tiene asociado mediante **finalBarrido**, no podremos volver a ejecutar el experimento *afExp*, por lo que es necesario definir el proceso *ft2Proc*. El mismo, será asociado en el extremo de la asociación **finalBarrido** de *tBarrierMed*. Como se puede ver en la figura 4.11, *tBarrierMed* ejecuta una SecuenciaDePulsos *invRecSec*, estudiada en la sección 2.1, la cual tiene asociadas los objetos de la clase Propiedad: *tIR*, que será sobre la cual se realizará el barrido, y *p90IR* y *p180IR*, que representan los tiempos que deben durar los pulsos de π y de $\pi/2$, respectivamente.

La relación entre el barrido *barridoIR* y la propiedad *tIR* tiene asociada un objeto de la clase ValoresBarrido que contiene los valores que toma *tIR* a lo largo del barrido. Algo a destacar, es que *p90IR* toma el mismo valor que *p90FID*, lo que es muy lógico ya que tienen el mismo significado, aunque son usadas en distintas secuencias de pulsos.

En el diagrama de objetos de la figura 4.11, se obviaron las propiedades de frecuencia y fase para facilitar la comprensión. Las mismas se muestran en la figura 4.12.

En esta medición no se batirá la frecuencia, lo que significa que *frec0IR* y *frec1IR* toman el mismo valor.

En la figura 4.13 se muestra la relación agregada para que se actualice la frecuencia antes de realizar la primera medición del barrido *tBarrierMed*.

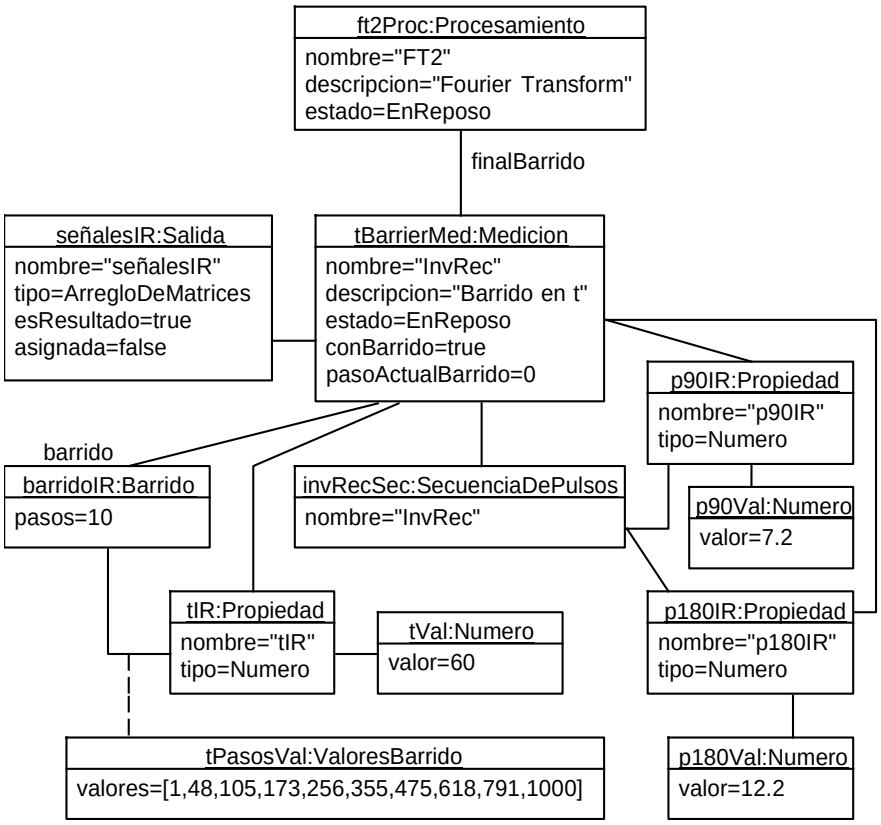

Figura 4.11: Objeto *tBarrierMed* y sus relaciones con los demás objetos (más relaciones en figura 4.12)

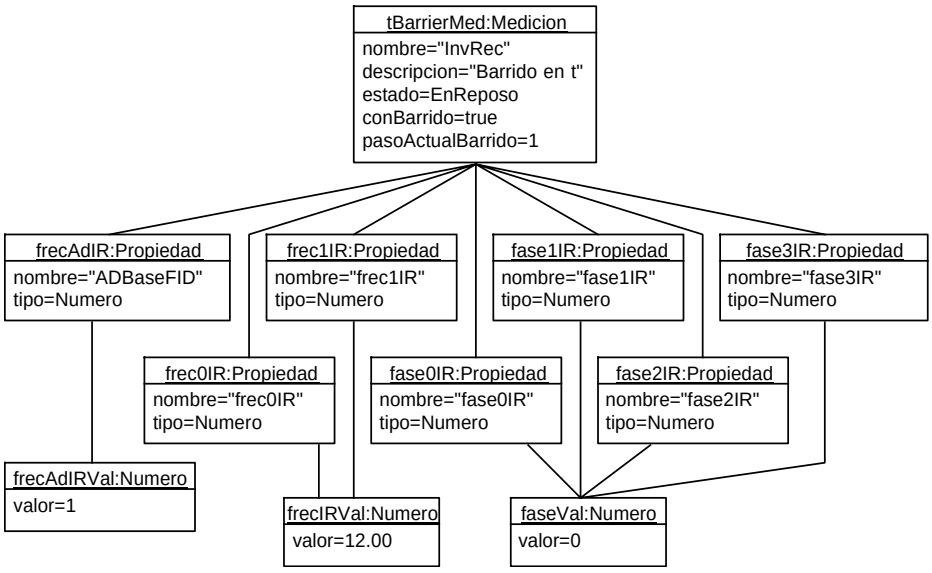

Figura 4.12: Propiedades de *tBarrierMed* (completa el diagrama de la figura 4.11)

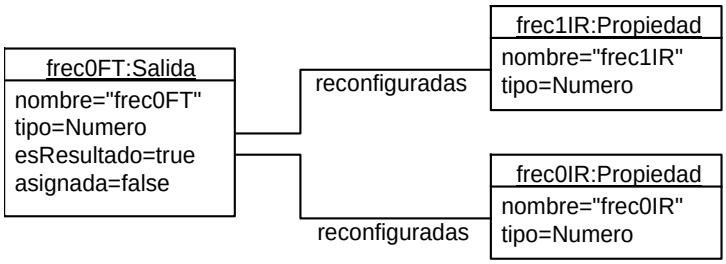

Figura 4.13: Relación de *reconfiguracion* de las frecuencias del objeto de tipo Medicion *tBarrierMed*

Notemos que *ft2Proc* tiene la misma estructura que *ftProc*, con la diferencia que, en vez de tomar la señal de una FID, toma un arreglo de señales de Inversión-Recuperación, por lo que se debe reformular el procedimiento asociado a *ft2Proc* para que tome como entrada un arreglo de señales y analice la última señal adquirida. Esto se puede ver en la figura 4.14.

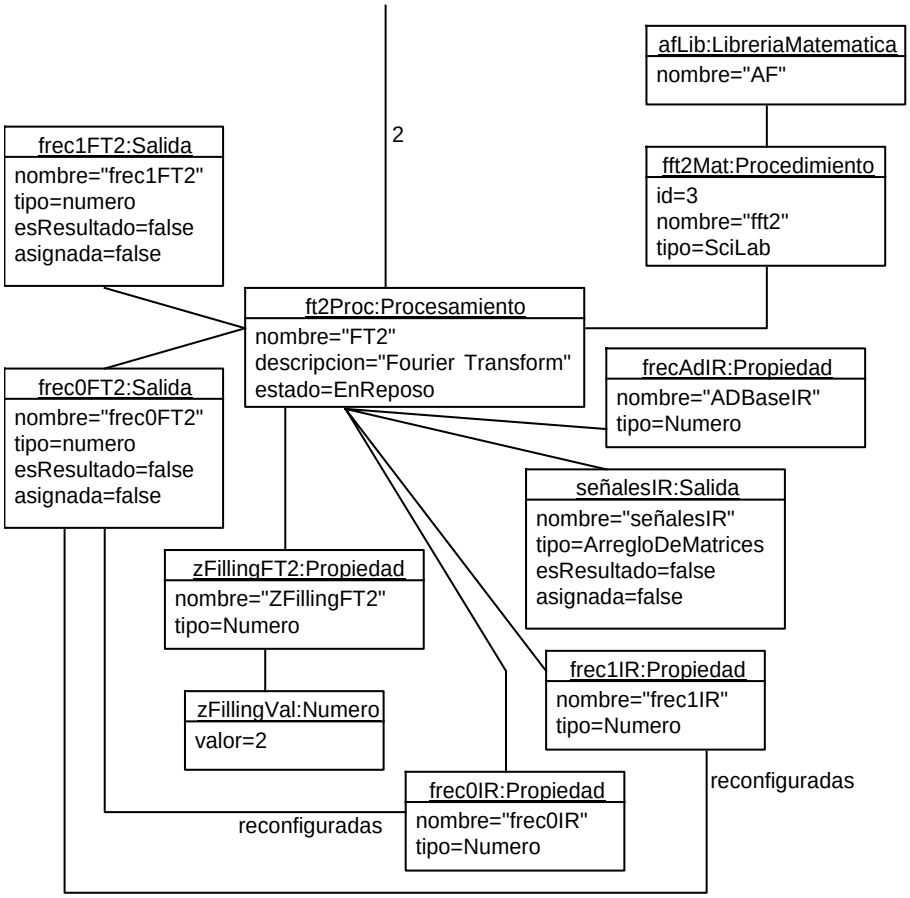

Figura 4.14: Objeto *ft2Proc* y sus relaciones con los demás objetos

Ahora bien, debido a la longitud de una ejecución del proyecto *InvRec*, no mostraremos la misma con el nivel de detalle con el cual se mostró la ejecución del ajuste de frecuencia en la sección anterior. Nos concentraremos en analizar los dos cambios de estado más interesantes que ocurren en esta ejecución. El resto no tiene mayores complicaciones que las analizadas en la ejecución completa del ajuste de frecuencia.

El primer cambio de estado que veremos, será el que ocurre luego de la primera ejecución de *tBarrierMed*. Notemos que antes de esto, los únicos cambios observables a nivel de objetos son el estado de las salidas de los procesos del experimento *afExp*, las cuales están todas asignadas, y el atributo **valor** de *frecIRVal*. Cuando *tBarrierMed* entra en estado *EnEjecucion*, cambia el valor de su atributo **pasoActualBarrido** a 1 y cambia el valor del atributo **valor** de *tVal* a 1 (que es el primero del atributo **valores** de *tPasosVal*). Realizada la primera medición del barrido de *tBarrierMed*, este proceso, en vez de cambiar al estado *EnReposo*, cambiará al estado *Pausado*. Además, la salida *señalesIR* cambiará el valor de su atributo **asignada** a verdadero y se le asociará un valor, de tipo ArregloDeMatrices, el cual tendrá asociado el objeto de tipo Matriz que contiene la señal adquirida. Este estado se puede ver en la figura 4.15.

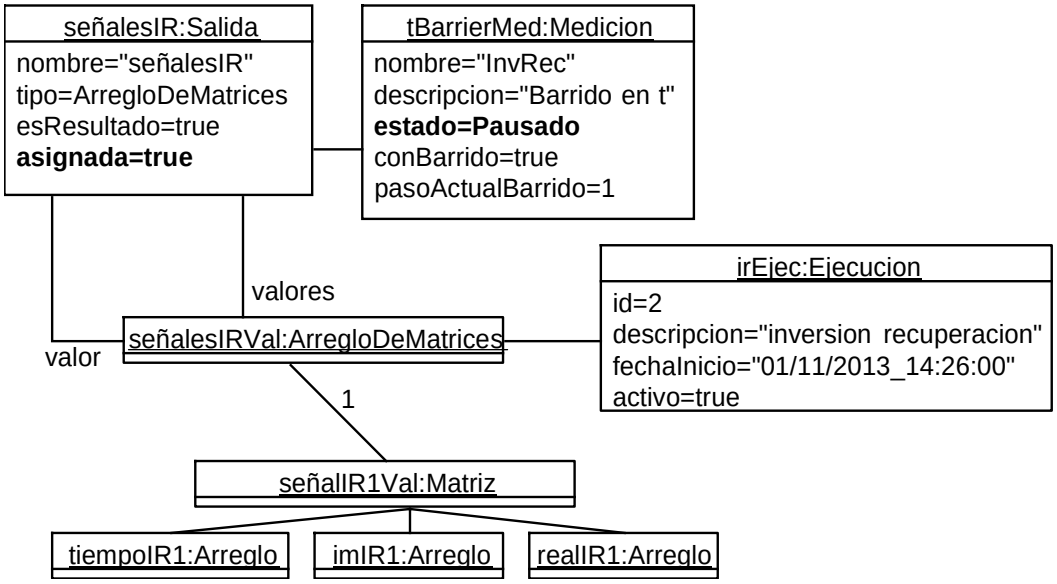

Figura 4.15: Cambio de estado de *tBarrierMed* luego de su primer ejecución

Luego de esto, se ejecuta *ft2Proc*, el cual ajusta la frecuencia. Notemos que, una vez se ejecutó *ft2proc*, sus salidas quedan asignadas, por lo que las restricciones impuestas al modelo no nos permiten ejecutar nuevamente *tBarrierMed*. Para poder ejecutarlo, será necesario pasar a falso el atributo **asignada** de cada una de las salidas de *ft2Proc* (por la restricción 30). Luego, se vuelve a ejecutar *tBarrierMed,* con la sutileza de que se incrementa en uno el valor del atributo **pasoActualBarrido**. Esto se repite de manera iterativa hasta que el valor del atributo **pasoActualBarrido** de *tBarrierMed* es igual al atributo **pasos** de *barridoIR*. Una vez concluida esta medición, *tBarrierMed* pasará al estado *EnReposo*. Luego de esto, se ejecuta una última vez *ft2Proc* y se finaliza la ejecución de manera normal.

Para que la ejecución de este proyecto sea idéntica a una ejecución del experimento Inversion-Recovery del *Condor Lite*, visto en la sección 3.1.2, se debe añadir un último proceso en el experimento *invRecExp* que se ejecute al finalizar el barrido, tenga como entrada el arreglo de señales *señalesIR* y como salida una señal que será el resultado del preprocesamiento de un barrido inversión-recuperación típico de la TD-NMR. Llamemos a este

proceso *irPreProc*. Al añadir *irPreProc* al experimento *invRecExp* en el orden 3, lo que lo vuelve el último proceso del experimento, ocurre que, aunque *ft2Proc* se ejecuta luego de cada medición del barrido de tBarrierMed, irPreProc se ejecutará una única vez, luego ejecutar ft2Proc, una vez que la medición *tBarrierMed* haya vuelto al estado *EnReposo.*

Con este ejemplo, se termina de mostrar el poder de representación que provee NMR_CORE y cómo se comportan sus proyectos a lo largo de una ejecución.

Un comportamiento del modelo que no se menciona en estos ejemplos es que, si bien las restricciones impuestas limitan la ejecución a ejecutar los procesos de manera ordenada, no restringen la cantidad de ejecuciones que se realizan de un mismo proceso. Esto permite que se pueda ejecutar, por ejemplo en el caso de este experimento, diez veces seguidas *ft2Proc*. Esto deberá ser evitado en las implementaciones que se realicen del modelo, ya que puede traer complicaciones a la hora de determinar los valores que tienen las salidas en una ejecución.

5 Conclusiones

En este trabajo, se define un modelo de datos para representar experimentos en TD-NMR y se analiza su comportamiento en experimentos reales, ampliamente utilizados por los especialistas en la materia. Concluida esta etapa, hemos observado propiedades del modelo. A continuación, haremos un breve resumen de estas propiedades y propondremos un trabajo a futuro que extienda la funcionalidad del modelo.

A grandes rasgos podemos observar que la clara separación de esquemas que brinda la arquitectura ANSI/SPARC permite ver el problema tanto desde el punto de vista teórico como del desarrollo, cuestión indispensable en un trabajo que está orientado tanto a personas del ámbito de la investigación en la técnica RMN como a futuros desarrolladores de aplicaciones en el ámbito científico.

En cuanto al diseño generado, podemos decir que el mismo permite representar un variado conjunto de experimentos de TD-NMR de manera simple y acorde a la manera en la cual se realizan los mismos en la práctica, lo que cumple con los objetivos de buscados.

El gran poder de expresividad que brinda el modelo, exige fijar invariantes de integridad fuertes. El chequeo de los invariantes definidos, requerirá de algunos sacrificios en cuanto a eficiencia en las distintas implementaciones que se puedan realizar.

El nivel de acoplamiento que poseen las distintas clases de este modelo se considera dentro de lo estándar. Si bien se podría intentar disminuir esta cualidad del modelo en el futuro, no se considerará una tarea imprescindible.

Además, es importante mencionar que se desarrollo una plataforma de experimentación en TD-NMR, llamada *MagLab*, que brinda soporte a este modelo. Esta plataforma, propiedad del cliente, permite generar y ejecutar proyectos de NMR_CORE en una amplia gama de espectrómetros de RMN y soporta procedimientos escritos en el leguaje SciLab. Debido al éxito de la misma, el cliente ha decidido migrar varias aplicaciones comerciales para que soporten proyectos de NMR_CORE, lo que comprueba la importancia de este trabajo, y le da a NMR_CORE el respaldo buscado.

5.1 Trabajo a futuro

Normalmente, hay ciertas las mediciones en los equipos de TD-NMR que demoran tiempos considerables. Nuestro modelo permite solamente la ejecución secuencial de procesos, por lo que, mientras el equipo ejecutar una medición, las aplicaciones que den soporte a este modelo permanecen en espera de una respuesta. Dar soporte a la paralelización de los procesos sincronizada debería ser el próximo desafío.

Al haber diseñado NMR_CORE de manera que pueda ser utilizado en cualquier equipo del cliente, no existen restricciones sobre los valores que pueden tomar las propiedades de las mediciones. Esto dar lugar a que se puedan ejecutar mediciones que configuren las

propiedades de un equipo con valores por fuera de los límites soportados por el hardware, lo que derivaría en la rotura de componentes de alto costo del mismo. Para prevenir esto, es necesario extender NMR_CORE para que contemple las especificaciones de los equipos y restringa los valores de las propiedades de las mediciones a los definidos en la especificación correspondiente al equipo con el cual se está trabajando. Sobre esto también se deberá trabajar en el futuro.

6 Apéndice

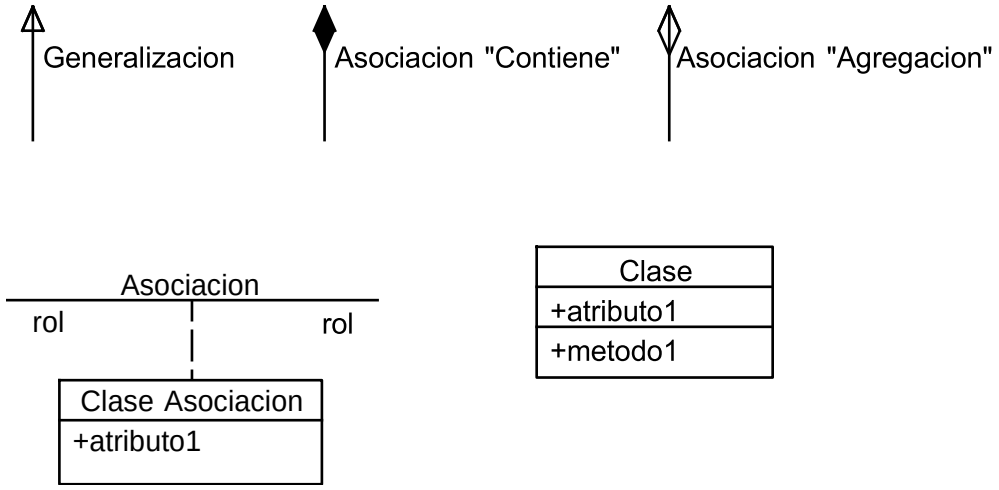

Figura 6.1: Elementos de un diagrama de clases UML

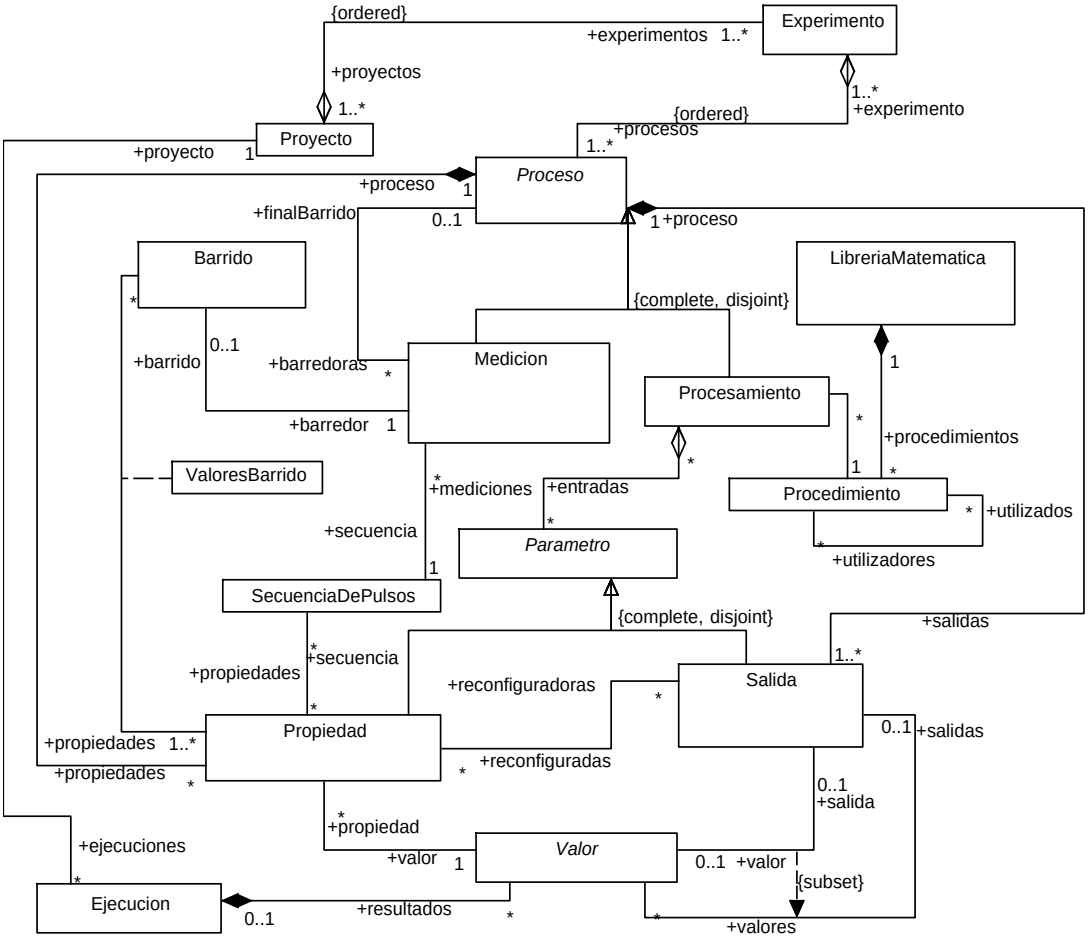

Figura 6.2: Esquema conceptual del modelo NMR_CORE

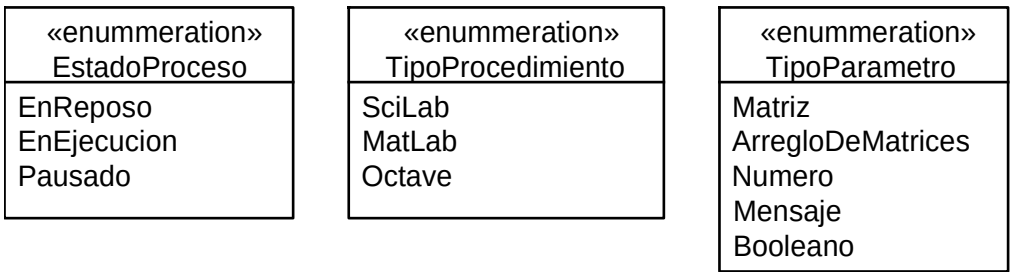

Figura 6.3: Enumeraciones usadas por el esquema lógico de NMR_CORE

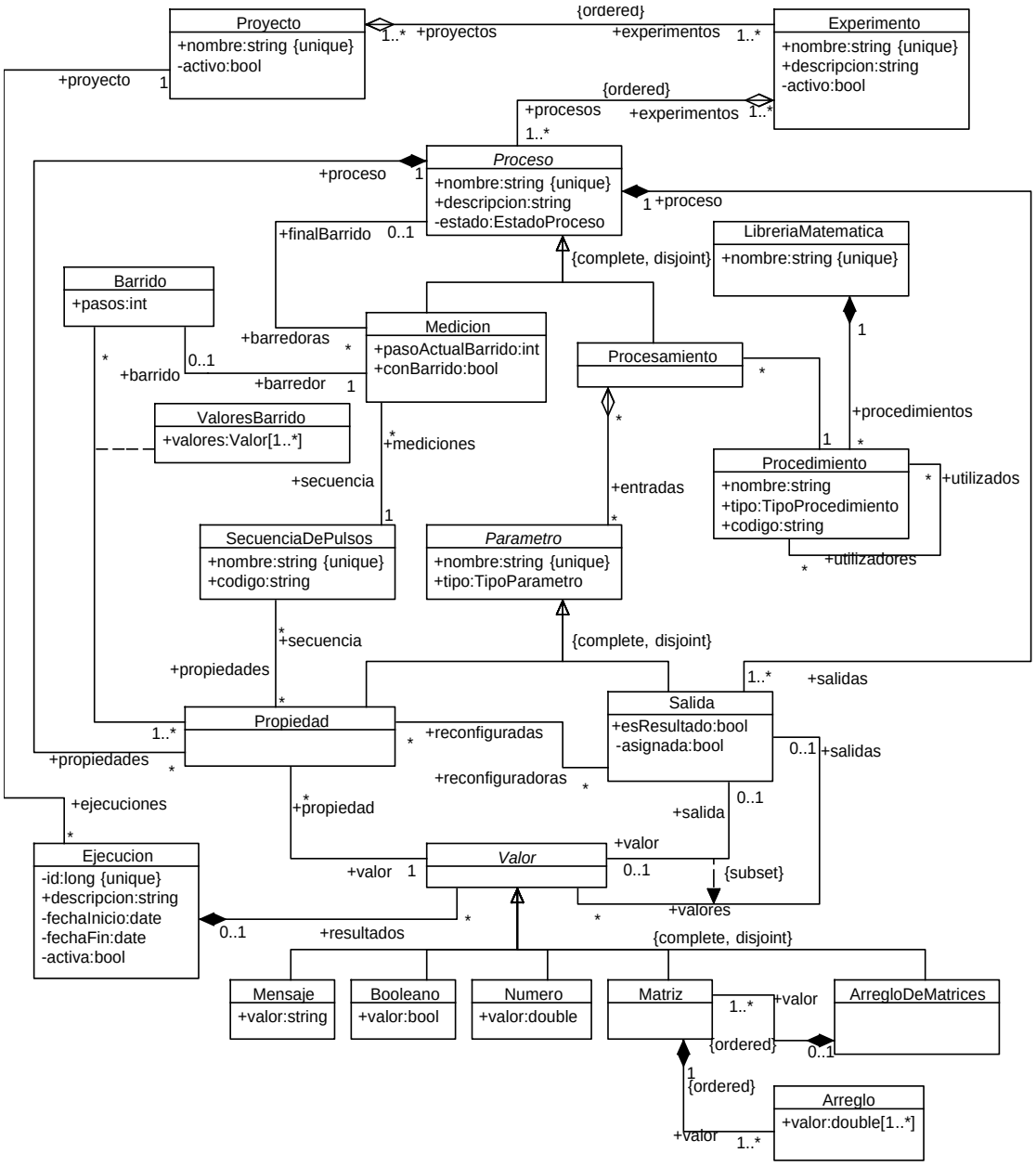

Figura 6.4: Esquema lógico del modelo NMR_CORE

7 Bibliografía

[1] R. Fogh, J. Ionides, E. Ulrich, W. Boucher, W. Vranken, J. Linge, M. Habeck, W. Rieping, T. Bhat, J. Westbrook, J. Thornton, G. Gilliland y H. Berman, «The CCPN project: an interim report on a data model for the NMR community,» 2002.

[2] «CCPN Official Site,» [En línea]. Available: http://www.ccpn.ac.uk/.

[3] R. Fogh, W. Vranken, W. Boucher, T. Stevens and E. Laue, "A nomenclature and data model to describe NMR experiments," 2006.

[4] [En línea]. Available: http://es.wikipedia.org/wiki/Resonancia_magn%C3%A9tica_nuclear.

[5] M. H. Levitt. Spin Dynamics: Basic of Nuclear Magnetic Resonance, 2nd Edition. John Wiley and Sons, 2009.

[6] E. Fukushima y S. B. W. Roeder. Experimental Pulse NMR: A Nuts and Bolts Approach. Westview Press, 1993.

[7] J. Keeler. Understanding NMR Spectroscopy. J. Wiley y Sons,2002.

[8] the European Process Industries STEP Technical Liaison Executive; West, Matthew, «Developing High Quality Data Models,» 2003.

[9] G. C. W. Graeme E. Simsion. Data Modeling Essentials (Third Edition), 2005.

[10] B. a. Schmidt, «Final Report of the ANSI/X3/SPARC Study Group on Database,» 1982.

[11] O. M. G. (OMG), «Unified Modeling Language,» 2001. [En línea]. Available: http://www.omg.org/spec/UML/2.4.

[12] G. B. I. J. J. R. Cris Kobryn, UML Distilled: A Brief Guide to the Standard Object Modeling Language (Third Edition), 2003.

[13] C. I. Dania, «MySQL4OCL: Un compilador de OCL a MySQL.,» *Universidad Complutense de Madrid,* 2011.

[14] G. Booch, J. Rumbaugh y I. Jacobson. The Unified Modeling Language - User Guide. Addison Wesley Longman Publishing Co., Inc., 1999.

[15] O. M. G. (OMG), «Object Constraint Language,» 2010. [En línea]. Available: http://www.omg.org/spec/OCL/2.2.