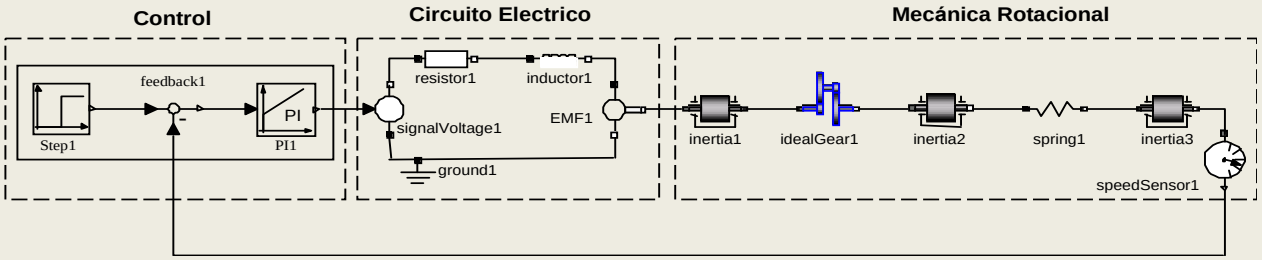
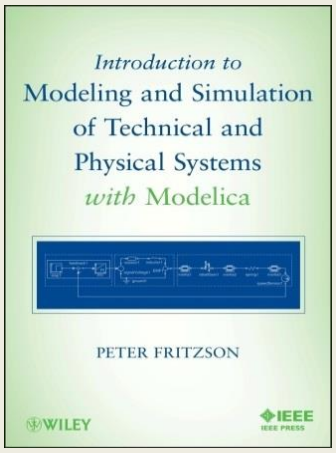
# Introducción al Modelado y Simulación de Sistemas Técnicos y Físicos con Modelica


## Peter Fritzson

Para información adicional, incluyendo archivos de ejercicios y soluciones descargables, visite: la página web del libro en http://www.openmodelica.org, la página web de Modelica Association http://www.modelica.org, la página web del autor http://www.ida.liu.se/~petfr, la página web del grupo de investigación en Modelica del autor http://www.ida.liu.se/labs/pelab/modelica o escriba un correo electrónico al autor del libro peter.fritzson@liu.se. Para información adicional de la versión en español escriba un correo a José Luis Villa Ramírez a jvilla@unitecnologica.edu.co

**Traducción**

La versión original de este libro ha sido traducida de la versión original en inglés al español por:

Sebastián Dormido Bencomo
Catedrático de Universidad de Ingeniería de Sistemas y Automática
Universidad Nacional de Educación a Distancia
Madrid, España

Alfonso Urquía Moraleda
Profesor Titular de Universidad de Ingeniería de Sistemas y Automática
Universidad Nacional de Educación a Distancia
Madrid España

Esta versión es una versión revisada y actualizada por:

José Luis Villa Ramírez
Profesor Titular
Universidad Tecnológica de Bolívar
Cartagena, Colombia

**Derechos reservados**

De la edición en inglés:
*Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*.
ISBN: 978-1-1180-1068-6, Wiley-IEEE Press, September 2011.


De la edición traducida al español:

ISBN: 978-91-7685-924-7

ninguna responsabilidad de patente con respecto al uso de la información aquí contenida. Aunque se han tomado todas las precauciones en la preparación de este libro, el editor no asume responsabilidad alguna por los errores u omisiones. Tampoco se asume ninguna responsabilidad por los daños que puedan resultar del uso de la información aquí contenida.

**Diseño de cubierta**

Parte del material del Tutorial de Modelica, la Especificación del Lenguaje Modelica, y documentación y código de las librerías de Modelica que están disponibiles en http://www.modelica.org han sido reproducidas en este libro con permiso de Modelica Association bajo la Modelica Licence 2 Copyright © 1998-2015, Modelica Association. Vea las condiciones de la licencia (incluyendo la restricción de la garantía) en http://www.modelica.org/modelica-legal-documents/ModelicaLicense2.html. Licenciado por Modelica Association bajo la Modelica License 2.

**No Asunción de Responsabilidades de la Licencia de Modelica**

El software (fuentes, binarios, etc.) en su forma original o modificada se proporciona "tal como está" y los propietarios de los derechos de autor no asumen ninguna responsabilidad por su contenido cualquiera que sea. No se asume ninguna garantía expresa o implícita, incluidas entre otras, las garantías implícitas de comercialización y de adecuación a un fin en particular. *En ningún caso* los propietarios de los derechos de autor, o cualquiera que modifique y/o redistribuya el paquete, *es responsable* de daño alguno directo, indirecto, incidental, especial, ejemplar, o consiguiente, que surjan de cualquier forma de uso de este software, incluso si se indica la posibilidad de tal daño.

**Nombres de marcas registradas**

Modelica® es una marca registrada de Modelica Assocation. . Dymola® es una maca registrada de Dassault Systèmes. MATLAB® y Simulink® son marcas registradas de MathWorks Inc. Java™ es una marca de Oracle Inc. Mathematica® y Wolfram SystemModeler® son marcas registradas de Wolfram Research Inc.

# Tabla de Contenido

## CAPÍTULO 2 UN RECORRIDO RÁPIDO POR MODELICA ..... 27

## Prefacio

Este libro enseña los conceptos básicos del modelado y simulación, y proporciona una introducción al lenguaje Modelica que es adecuada para personas que están familiarizadas con conceptos básicos de programación. Da una introducción básica acerca de los conceptos de modelado y simulación, así como de los fundamentos del modelado basado en componentes orientado a objetos adecuada para aquellos que inician en el tema. El libro tiene los siguientes objetivos:

- Ser un texto útil en cursos introductorios sobre modelado y simulación.
- Ser fácilmente accesible para las personas que no han tenido experiencia previa en modelado, simulación y programación orientada a objetos.
- Proveer una introducción básica de los conceptos del modelado físico, modelado orientado a objetos, y modelado basado en components.
- Proveer ejemplos de demostración de modelamiento en algunas áreas de aplicación seleccionadas.

El libro contiene ejemplos de modelos en diferentes dominios de aplicación, así como ejemplos que combinan varios dominios.

Todos los ejemplos y ejercicios en este libro están disponibles en un material electrónico de autoestudio llamado DrModelica, basado en este libro y en el libro mas extenso Principles of Object-Oriented Modeling of Simulation with Modelica 2.1 Fritzson (2004), para el cual se ha planeado una version mas actualizada. DrModelica guía gradualmente al lector desde ejemplos y ejercicios introductorios simples a unos más avanzados. Parte de este material de enseñanza puede ser descargado de manera gratuita del sitio web del libro, `www.openmodelica.org`, donde se puede encontrar material de enseñanza adicional relacionado con este libro.

## Agradecimientos

A los miembros de la Modelica Association que crearon el lenguaje Modelica, y han contribuido con muchos ejemplos de código Modelica en los documentos *Modelica Language Rationale* y *Modelica Language Specification* (ver http://www.modelica.org), algunos de los cuales son

usados en este libro. Los miembros que contribuyeron a varias versiones de Modelica se mencionan más adelante.

En primer lugar, agradezco a mi esposa, Anita, quien me ha apoyado y me ha soportado durante este esfuerzo de redacción de este libro.

Un agradecimiento especial a Peter Bunus por su ayuda con ejemplos de modelos, algunas figuras, formateado de MicroSoft Word, y por muchas discusiones inspiradoras. Muchas gracias a Adrian Pop, Peter Aronsson, Martin Sjölund, Per Östlund, Adeel Asghar, Mohsen Torabzadeh-Tari, y muchas otras personas que contribuyeron al esfuerzo de OpenModelica por su gran trabajo en el compilador y el sistema de OpenModelica, y también a Adrian por hacer que la herramienta OMNotebook finalmente funcione. Muchas gracias a Hilding Elmqvist por compartir la visión acerca de un lenguaje de modelado declarativo, por iniciar el esfuerzo para diseñar Modelica invitando a investigadores e ingenieros para formar un grupo de diseño, por servir como el primer presidente de la Asociación Modelica, y por el entusiasmo y las muchas contribuciones al diseño del lenguaje incluyendo la presión por un concepto de clase unificada. También agradezco por su inspiración en cuanto a material de presentación incluyendo la búsqueda de ejemplos clásicos de ecuaciones.

Muchas gracias a Martin Otter por servir como el segundo presidente de la Asociación Modelica, por su entusiasmo y energía, y sus contribuciones al diseño de la librería Modelica, por dos de las tablas y parte del texto en el Capítulo 5 sobre las librerías de Modelica a partir de las especificaciones del lenguaje Modelica, así como inspiración en cuanto a material de presentación. Gracias a Jakob Mauss quien hizo la primera version del glosario, y a varios miembros de la Asociación Modelica por las mejoras en el mismo.

Muchas gracias a Eva-Lena Lengquist Sandelin y Susanna Monemar por ayudar con los ejercicios, y por preparar la primera version del material de enseñanza desarrollado como cuaderno interactivo DrModelica, el cual hace que los ejemplos en este libro sean más accesibles para el aprendizaje interactivo y la experimentación.

Gracias a Peter Aronsson, Adrian Pop, Jan Brugård, Hilding Elmqvist, Vadim Engelson, Dag Fritzson, Torkel Glad, Pavel Grozman, Emma Larsdotter Nilsson, Håkan Lundvall, y Sven-Erik Mattsson por comentarios constructivos en diferentes partes del libro. Gracias a Hans Olsson y Martin Otter quien editó las recientes versiones de la especificación Modelica. Gracias por sus comentarios y recomendaciones a todos los miembros de PELAB y a los empleados de MathCore Engineering.

Linköping, Septiembre de 2003 y Mayo de 2015

Peter Fritzson

# Prólogo

Sin ningún género de dudas puede considerarse la Tesis Doctoral "Dymola: A structured model language for large continuous systems" leída en 1978 por Hilding Elmqvist y realizada en el Lund Institute of Tecnology (Suecia) bajo la supervisión del Prof. K. J. Åström el punto de partida del posterior nacimiento de Modelica.

Las ideas fundamentales que incorporaba el lenguaje de modelado Dymola fue la utilización de ecuaciones en general (no sentencias de asignación), una metodología orientada a objetos y un esquema nuevo de conexionado entre componentes. Esto permitió por primera vez que los desarrolladores de modelos contemplasen el proceso de modelado desde una perspectiva física y no matemática. Una consecuencia de todo lo dicho es que el lenguaje de modelado es acausal, es decir la causalidad computacional se establece cuando se traduce el modelo completo y es una propiedad global de todo el modelo y no de los componentes que lo constituyen. Esta propiedad de acausalidad es un requisito esencial para la reusabilidad de los componentes y es una de las grandes ventajas de este nuevo paradigma de modelado.

Sin embargo uno de los grandes problemas con todas las herramientas de modelado que iban surgiendo tomando como base estas ideas era la falta de transportabilidad de los modelos desarrollados que eran dependientes del entorno utilizado. A mediados de la década de los noventa Hilding Elmqvist reconociendo este hecho inicia un esfuerzo unificador que va a dar como resultado el desarrollo del lenguaje de modelado Modelica. La definición de Modelica se hace con la participación de muchos expertos de diferentes dominios de la ingeniería así como de la gran mayoría de desarrolladores de los lenguajes de modelado orientado a objetos que habían ido surgiendo.

El objetivo de Modelica fue crear un lenguaje de modelado capaz de expresar la conducta de modelos de un amplio abanico de campos de la ingeniería y sin limitar a los modelos a una herramienta comercial en particular. Se puede pues considerar a Modelica no solo como un lenguaje de modelado sino como una especificación de dominio público que permite el intercambio de modelos.

Modelica es así un lenguaje de modelado que no tiene propietario y su nombre es una marca registrada de la "Modelica Association" que es la responsable de la publicación de la especificación del lenguaje Modelica que entre otras ofrece a los desarrolladores de modelos las siguientes características:

- *Encapsulación del conocimiento*. El modelador debe ser capaz de codificar todo el conocimiento relacionado a un objeto particular en una forma compacta y con puntos de interfaz bien definidos con el exterior.
- *Capacidad de interconexión topológica*. El modelador debe ser capaz de interconectar objetos de una forma topológica, poniendo juntos modelos de componentes mediante un proceso similar a como un experimentador conecta equipos reales en un laboratorio. Este requisito entraña que las ecuaciones que describen los modelos deben tener una naturaleza declarativa o lo que es equivalente deben ser acausales.
- *Modelado jerárquico*. El modelador debe ser capaz de declarar a los modelos interconectados como nuevos objetos, haciéndolos indistinguibles desde el exterior de los modelos basados en ecuaciones. Se puede así construir modelos con una estructura jerárquica bien definida.
- *Instanciación de objetos*. El modelador debe tener la posibilidad para describir clases de objetos genéricos e instanciar objetos actuales de estas definiciones de clases mediante un mecanismo de invocación de modelos.
- *Herencia de clases*. Una característica muy útil es la herencia de clases ya que permite el encapsulamiento del conocimiento incluso por debajo del nivel de los objetos físicos. El conocimiento así encapsulado puede entonces distribuirse a través del modelo por un mecanismo de herencia que asegura que el mismo conocimiento no tendrá que ser codificado separadamente algunas veces en lugares diferentes del modelo.
- *Capacidad de interconexión generalizada*. Una característica útil de un entorno de modelado es su capacidad de interconectar modelos a través de sus *puertos* en la interfaz. Los puertos son diferentes de los modelos regulares (objetos) ya que ofrecen a los modelos un número variable de conexiones.

"Introducción al Modelado y Simulación de Sistemas Técnicos y Físicos con Modelica" escrito por Peter Fritzon profesor en la Universidad de Linköping (Suecia) trata de mostrar a sus lectores los fundamentos del modelado y la simulación y servir al mismo tiempo como una introducción básica al lenguaje Modelica.

Es de destacar el papel desempeñado por el grupo del Prof. Fritzson en la definición y divulgación del lenguaje Modelica y el esfuerzo que está haciendo con su grupo PELAB por desarrollar un compilador de Modelica de código abierto.

Estamos seguros que esta traducción de su texto al español será bienvenida por la cada vez más creciente comunidad de usuarios que han adoptado la metodología orientada a objetos en sus trabajos de modelado y simulación.

Madrid, julio de 2006
Sebastián Dormido

## Prólogo a la versión actualizada en español

Esta versión actualizada y ampliada del libro Introducción al Modelado y Simulación de Sistemas Técnicos y Físicos es en sí mismo el reflejo de la escalabilidad del lenguaje Modelica, así como del intenso trabajo del grupo de desarrollo del PELAB. En cuanto a la escalabilidad del lenguaje Modelica es claro que las actualizaciones y evolución del lenguaje de descripción de sistemas conservan los mismos principios básicos que permiten construir complejos sistemas a partir de componentes basados en principios físicos, y a la vez conservar una definición de clase unificada. En cuanto al intenso trabajo del grupo de desarrollo del PELAB es importante reconocer la evolución que han tenido las herramientas de OpenModelica en cuanto a estabilidad y poder de compilación y simulación.

Esta nueva versión incluye un ejemplo detallado del modelamiento de un Motor de Corriente Continua a partir de componentes predefinidos en el Capítulo 4, y se ha introducido un nuevo capítulo que describe la Librería Estándar de Modélica.

Desde el lado de las herramientas, esta versión del libro incluye en la sección de los Apéndices una completa introducción al Modelamiento Textual con OpenModelica Notebook (OMNotebook) y DrModelica. Adicionalmente se presenta un ejemplo detallado de modelamiento con la herramienta OpenModelica Connection Editor (OMEdit). Tanto OMNotebook como OMEdit son las herramientas de compilación y simulación mas avanzadas desarrolladas por el consorcio OpenModelica, que es el ambiente de código abierto de modelamiento y simulación basado en Modelica.

Esperamos que con esta nueva versión del libro la comunidad de usuarios de Modelica y OpenModelica crecerá con mayor velocidad en los países de habla hispana.

Cartagena de Indias, Septiembre de 2015
José Luis Villa

# Capítulo 1

# Conceptos Básicos

Con frecuencia se dice que los computadores están revolucionando la ciencia y la ingeniería. Utilizando computadores somos capaces de construir complejos diseños de ingeniería, tales como transbordadores espaciales. Podemos calcular las propiedades de cómo era el universo una fracción de segundo después del "big bang". Nuestros retos son cada vez más exigentes. Queremos crear diseños aun más complejos, tales como mejores vehículos espaciales, automóviles, medicinas, sistemas telefónicos móviles computarizados, etc. Deseamos comprender aspectos más profundos de la naturaleza. Estos son simplemente unos pocos ejemplos de modelado y simulación soportado por computadores. Se necesitan herramientas y conceptos más potentes para ayudarnos a manejar esta complejidad creciente, que es precisamente de lo que trata este libro.

Este libro presenta un enfoque basado en componentes orientados a objetos para el modelado matemático y la simulación asistidos por computador, mediante el potente lenguaje Modelica y sus herramientas asociadas. Puede considerarse que Modelica es un enfoque casi universal para el modelado y simulación computacional de alto nivel, al ser capaz de representar un abanico de áreas de aplicación y proporcionar una notación general, así como abstracciones poderosas e implementaciones eficientes. La parte introductoria de este libro, compuesta por los dos primeros capítulos, da una visión panorámica de los dos temas principales de este texto:

- Modelado y simulación.
- El lenguaje Modelica.

Los dos temas se presentan juntos porque forman un todo. A través del texto, Modelica se utiliza como un vehículo para explicar diferentes aspectos del modelado y la simulación. En forma

complementaria, una serie de conceptos del lenguaje Modelica son presentados mediante ejemplos de modelado y simulación. Este primer capítulo introduce conceptos básicos tales como *sistema*, *modelo*, y *simulación*. El segundo capítulo constituye un rápido recorrido por el lenguaje Modelica. Contiene una serie de ejemplos, entremezclados con presentaciones de temas tales como el modelado matemático orientado a objetos. El tercer capítulo presenta una introducción al concepto de clase de Modelica, en tanto que el cuarto capítulo presenta la metodología de modelado para sistemas continuos, discretos, e híbridos. El quinto capítulo presenta un corta descripción de la *librería estándar de Modelica* y de algunas librerías de modelos actualmente disponibles en Modelica para diferentes dominios de aplicación. Finalmente, en dos de los apéndices, se presentan algunos ejemplos usando modelamiento textual a través de la herramienta de libro electrónico OMNotebook de OpenModelica, así como varios ejemplos simples de modelamiento gráfico.

## 1.1 Sistemas y Experimentos

¿Qué es un sistema? Ya hemos mencionado algunos sistemas tales como el universo, un transbordador espacial, etc. Un sistema puede ser casi cualquier cosa. Un sistema puede contener subsistemas, que a su vez también son sistemas. Una posible definición de sistema podría ser:

- Un sistema es un objeto o colección de objetos cuyas propiedades deseamos estudiar.

Nuestro deseo de estudiar ciertas propiedades seleccionadas de los objetos es central en esta definición. El aspecto de “estudio” es importante, a pesar del hecho de que es subjetivo. La selección y definición de lo que constituye un sistema es algo arbitrario y debe estar guiado por el uso que se va a hacer del sistema.

¿Qué razones puede haber para estudiar un sistema? Hay muchas respuestas a esta pregunta, pero cabe destacar dos grandes motivaciones:

- Estudiar un sistema para comprenderlo, con la finalidad de construirlo. Este es el punto de vista de la ingeniería.
- Satisfacer la curiosidad humana, por ejemplo, para comprender más acerca de la naturaleza—el punto de vista de las ciencias naturales.

### 1.1.1 Sistemas Naturales y Artificiales

De acuerdo con la definición que hemos dado anteriormente, un sistema puede tener un origen natural, como por ejemplo el universo, puede ser artificial, como un transbordador espacial, o puede ser una mezcla de ambos. Por ejemplo, la casa mostrada en la Figura 1-1, que posee un sistema de agua caliente por radiación solar, es un sistema artificial. Es decir, está fabricado por los seres humanos. Si incluimos el sol y las nubes en el sistema, se transforma en una combinación de componentes naturales y artificiales.

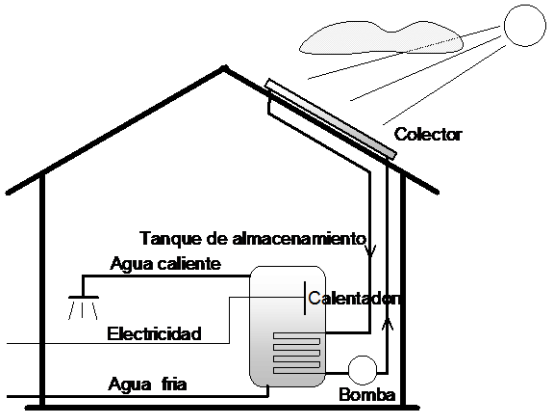



**Figura 1-1.** Un sistema: una casa con un sistema de agua caliente por radiación solar, junto con las nubes y el sol.

Incluso si un sistema ocurre naturalmente, su definición es siempre fuertemente selectiva. Esto se hace muy aparente en la siguiente cita de Ross Ashby (Ashby, 1956):

> *Llegado este punto, debemos tener claro cómo se define un sistema. Nuestro primer impulso es apuntar al péndulo y decir "el sistema es aquella cosa que está allí". Este método, sin embargo, tiene una desventaja fundamental: cada objeto material contiene no menos de una infinidad de variables, y por lo tanto, de posibles sistemas. El péndulo real, por ejemplo, no tiene solo longitud y posición; también tiene masa, temperatura, conductividad eléctrica, estructura cristalina, impurezas químicas, algo de radioactividad, velocidad, potencia reflejada, estrés estructural, una capa superficial de humedad, contaminación bacterial, absorción óptica, elasticidad, forma, gravedad, etc. Pretender estudiar todos estos hechos no es realista, y de hecho nunca se hace. Lo*

*necesario es que escojamos y estudiemos los hechos que son relevantes para nuestro propósito específico.*

Incluso si el sistema es completamente artificial, tal como el sistema de telefonía móvil representado en la Figura 1-2, debemos ser altamente selectivos en su definición, dependiendo de qué aspectos en concreto deseamos estudiar en ese momento.

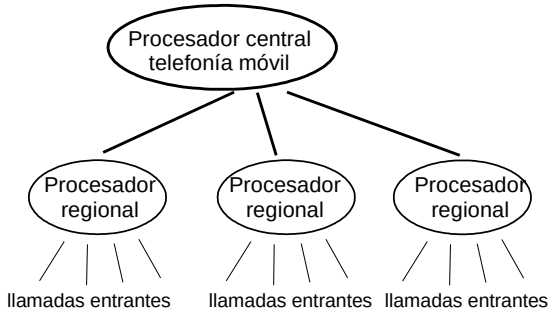



**Figura 1-2.** Un sistema de telefonía móvil, que contiene un procesador central y procesadores regionales para manejar llamadas entrantes.

Una propiedad importante de los sistemas es que deben ser *observables*. Algunos sistemas, en los que no se incluyen los grandes sistemas naturales como el universo, son también *controlables* en el sentido de que podemos influir en su comportamiento a través de ciertas entradas. Es decir:

- Las *entradas* de un sistema son las variables del entorno que influyen sobre el comportamiento del sistema. Estas entradas pueden o no ser controlables por nosotros.
- Las *salidas* de un sistema son variables que son determinadas por el sistema y que pueden influir sobre el entorno que le rodea.

En muchos sistemas, las mismas variables actúan tanto como *entradas y* como *salidas*. Hablamos de comportamiento *acausal* si las relaciones o influencias entre las variables no tienen una dirección causal, que es el caso para las relaciones descritas por ecuaciones. Por ejemplo, en un sistema mecánico las fuerzas del entorno influyen sobre el desplazamiento de un objeto, pero por otra parte el desplazamiento del objeto afecta a las fuerzas entre el objeto y el entorno. En este caso, la decisión de qué es entrada y qué es salida es fundamentalmente una elección del observador, que está guiada por el interés en particular del estudio, en lugar de ser una propiedad del propio sistema.

### 1.1.2 Experimentos

De acuerdo a nuestra definición de sistema, la observabilidad es esencial si se desea estudiar un sistema. Debemos al menos ser capaces de observar algunas de las salidas del sistema. Podremos aprender más acerca del sistema si somos capaces de excitarlo controlando sus entradas. Este proceso se llama *experimentación*. Es decir:

- Un *experimento* es el proceso de extraer información de un sistema excitando sus entradas.

Para poder realizar un experimento sobre un sistema, éste debe de ser tanto controlable como observable. Aplicamos un conjunto de condiciones externas a las entradas accesibles y observamos la reacción del sistema al medir aquellas de sus salidas que son accesibles.

Una de las desventajas del método experimental es que para un gran número de sistemas muchas entradas no son ni accesibles ni controlables. Estos sistemas están bajo la influencia de entradas inaccesibles, algunas veces llamadas *entradas de perturbación*. Igualmente, a menudo muchas de las salidas que son realmente útiles no están accesibles, con lo que no pueden ser medidas. Éstas suelen denominarse e*stados internos* del sistema. Surgen también una serie de problemas prácticos asociados con la realización de un experimento, por ejemplo:

- El experimento podría ser demasiado *costoso*: investigar la durabilidad de los barcos construyéndolos y haciéndolos colisionar es un método muy caro de obtener información.
- El experimento podría ser demasiado *peligroso*: no es aconsejable entrenar a los operarios de las plantas nucleares acerca de cómo solventar situaciones peligrosas, poniendo para ello el reactor en estados de funcionamiento de riesgo.
- El *sistema* sobre el que se desea experimentar podría *no existir aún*. Esto sucede típicamente con sistemas que están aun por diseñar o fabricar.

Las desventajas del método experimental nos llevan al concepto de modelo. Si hacemos un modelo de un sistema y el modelo es lo suficientemente realista, entonces podemos emplear este modelo para investigar y responder a muchas preguntas referentes al sistema real.

## 1.2 El Concepto de Modelo

Dadas las definiciones previas de sistema y experimento, podemos ahora intentar definir la noción de modelo:

- Un *modelo* de un sistema es cualquier cosa a la que se puede aplicar un "experimento", con el fin de responder a preguntas respecto del *sistema*.

Esto implica que puede usarse un modelo para responder preguntas acerca de un sistema, *sin* realizar experimentos sobre el sistema *real*. En su lugar, realizamos "experimentos" simplificados sobre el modelo. El modelo, a su vez, puede considerarse como un sistema simplificado que refleja las propiedades del sistema real. En el caso más simple, el modelo puede ser simplemente cierta información que se usa para responder a preguntas acerca del sistema.

Dada esta definición, cualquier modelo también constituye un sistema. Los modelos, al igual que los sistemas, son por naturaleza jerárquicos. Si se separa una parte del modelo, se obtiene un nuevo modelo, que es válido para un subconjunto de los experimentos para los que a su vez el modelo original también es válido. Un modelo está siempre relacionado con el sistema al que representa y con los experimentos a los que puede estar sujeto. Una afirmación del tipo "el modelo del sistema no es válido" carece de sentido si no se indica cuál es el sistema asociado y el experimento. Un modelo de un sistema podría ser válido para un cierto experimento en el modelo y no serlo para otro. El término "*validación* del modelo" (ver la Sección 1.5.3) siempre se refiere a un experimento o a una clase de experimento a realizar.

Hablamos de diferentes clases de modelos dependiendo de cómo se representa:

- Modelo *mental*—una sentencia como "esta persona es fiable" nos ayuda a responder a preguntas acerca del comportamiento de esa persona en diversas situaciones.
- Modelo *verbal*—esta clase de modelo se expresa mediante palabras. Por ejemplo, la frase "si se aumenta el límite de velocidad, entonces ocurrirán más accidentes" es un ejemplo de un modelo verbal. Los sistemas expertos son una técnica para formalizar modelos verbales.
- Modelo *físico*—se trata de un objeto físico que reproduce algunas propiedades de un sistema real, para ayudarnos a responder preguntas del sistema. Por ejemplo, en la fase de diseño de edificios, aeroplanos, etc., frecuentemente se construyen pequeños modelos físicos con la misma forma y apariencia como los objetos reales a estudiar, por ejemplo con respecto a sus propiedades aerodinámicas y estéticas.
- Modelo *matemático*— es una descripción de un sistema donde las relaciones entre las variables del sistema se expresan de forma matemática. Las variables pueden ser cantidades medibles, tales como el tamaño, la longitud, el peso, la temperatura, el nivel de desempleo, el flujo de información, la velocidad medida en bits por segundo, etc. La mayoría de las leyes de la naturaleza son modelos matemáticos en este sentido. Por ejemplo, la Ley de Ohm describe la relación entre la corriente y la caída de tensión en una

resistencia; las Leyes de Newton describen relaciones entre la velocidad, la aceleración, la masa, la fuerza, etc.

Fundamentalmente, las clases de modelos que trataremos en este libro son modelos matemáticos representados de diversas formas, tales como ecuaciones, funciones, programas de computador, etc. Los sistemas representados mediante modelos matemáticos en un computador se denominan a menudo *prototipos virtuales*. El proceso de construir e investigar tales modelos es el prototipado virtual. Algunas veces, el término *modelado físico* se emplea también para el proceso de construir modelos matemáticos de sistemas físicos en el computador. Esto es así cuando el proceso de estructuración y síntesis del modelo matemático es el mismo que el usado para la construcción de los modelos físicos reales.

## 1.3 Simulación

En la sección previa mencionamos la posibilidad de efectuar "experimentos" sobre modelos, en lugar de sobre los sistemas reales que corresponden a los modelos. Este es realmente uno de los usos principales de los modelos, y se denota por el término *simulación*, del Latín *simulare*, que significa pretender ser. Definimos el término simulación como sigue:

- Una *simulación* es un experimento efectuado sobre un modelo.

En analogía con nuestra definición previa de *modelo*, esta definición de simulación no impone que el modelo deba ser representado matemáticamente o mediante un programa de computador. Sin embargo, en el resto de este texto nos concentraremos en los *modelos matemáticos*, fundamentalmente en aquellos que tienen una forma representable en un computador. Lo que sigue son algunos ejemplos de tales experimentos o simulaciones:

- Una simulación de un proceso industrial, tal como la fabricación de acero o de pulpa en la industria papelera, cuya finalidad es aprender acerca del comportamiento del proceso bajo diferentes condiciones de operación, con el objetivo final de mejorar el proceso.
- Una simulación del comportamiento de un vehículo, por ejemplo de un automóvil o un avión, con la finalidad de proporcionar entrenamiento realista a los pilotos.
- Una simulación de un modelo simplificado de una red de computadores de paquetes conmutados, para conocer su comportamiento para diferentes cargas, con el objetivo de mejorar su rendimiento.

Es importante darse cuenta de que las dos partes de la simulación, es decir, la *descripción del experimento y del modelo*, son entidades conceptualmente separadas. Por otra parte, si bien estos dos aspectos de una simulación son independientes, van unidos. Por ejemplo, un modelo es válido

sólo para una cierta clase de experimentos. Puede ser útil definir un *marco experimental* asociado con el modelo, que defina qué condiciones debe satisfacer un experimento para ser válido.

Si el modelo matemático se representa en forma ejecutable en un computador, las simulaciones se pueden realizar mediante *experimentos numéricos*, o en casos no numéricos mediante *experimentos computados*. Esta es una forma simple y segura de efectuar experimentos, con la ventaja añadida de que esencialmente todas las variables del modelo son observables y controlables. Sin embargo, el valor de los resultados de simulación es completamente dependiente de cuan bien el modelo representa al sistema real con respecto a las cuestiones a las que la simulación tiene que responder.

Excepto por la experimentación, la simulación es la única técnica que es aplicable de forma general para el análisis del comportamiento de sistemas arbitrarios. Las técnicas analíticas son mejores que la simulación, pero normalmente se aplican sólo bajo un conjunto de hipótesis simplificadoras, las cuales frecuentemente no pueden ser justificadas. Por otra parte, es bastante usual combinar técnicas analíticas con simulaciones. Es decir, la simulación no se usa sola, sino combinada con técnicas analíticas o semianalíticas.

### 1.3.1 Razones para la Simulación

Existe una serie de buenas razones que justifican realizar simulaciones, en lugar de experimentos sobre los sistemas reales:

- Los experimentos son demasiado *costosos*, demasiado *peligrosos*, o el sistema que se desea investigar *no existe aún*. Éstas son las principals dificultades de la experimentación con sistemas reales, mencionadas previamente en la Sección 1.1.2.
- La *escala de tiempos* de la dinámica del sistema no es compatible con la del experimentador. Por ejemplo, lleva millones de años observar pequeños cambios en el desarrollo del universo, mientras que cambios similares se pueden observar rápidamente en una simulación por computador del universo.
- Las variables pueden ser *inaccesibles*. En una simulación todas las variables pueden ser estudiadas y controladas, incluso aquellas que son inaccesibles en el sistema real.
- Fácil *manipulación* de modelos. Utilizando simulación, es fácil manipular los parámetros del modelo de un sistema, incluso fuera del rango admisible de un sistema físico particular. Por ejemplo, la masa de un cuerpo en un modelo de simulación basado en computador se puede aumentar de 40 a 500 Kg. pulsando una tecla, mientras que este cambio podría ser difícil de realizar en el sistema físico.

- Supresión de *perturbaciones*. En una simulación de un modelo es posible suprimir perturbaciones que podrían ser inevitables en las medidas del sistema real. Esto puede permitirnos aislar determinados efectos en particular y, por lo tanto, mejorar nuestra comprensión acerca de dichos efectos.
- Supresión de *efectos de segundo-orden*. A menudo, las simulaciones se realizan porque permiten la supresión de efectos de segundo orden, tales como pequeñas no linealidades u otros detalles de ciertos componentes del sistema. La supresión de los efectos de segundo orden puede ayudarnos a comprender mejor los efectos principales.

### 1.3.2 Peligros de la Simulación

La facilidad de uso de la simulación es también su desventaja más seria: es bastante fácil para el usuario olvidar las limitaciones y condiciones bajo las que una simulación es válida, y por lo tanto sacar conclusiones erróneas de la simulación. Para reducir estos peligros, se debería intentar siempre comparar al menos algunos de los resultados de la simulación del modelo con los resultados experimentales medidos del sistema real. También ayuda el ser conscientes de las siguientes tres fuentes más comunes de problemas cuando se emplea simulación:

- Enamorarse del modelo—el efecto Pygmalion[1]. Es fácil entusiasmarse en exceso con un modelo y olvidarse de su marco experimental. Es decir, el modelo no es el sistema real y sólo representa al sistema real bajo ciertas condiciones. Un ejemplo es la introducción de zorros en el continente Australiano, que se hizo para resolver el problema de los conejos, sobre la hipótesis de que los zorros cazan conejos, lo cual es verdad en muchas otras partes del mundo. Desgraciadamente, los zorros encontraron que la fauna indígena era mucho más fácil de cazar y en gran medida ignoraron a los conejos.
- Forzar a que la realidad encaje dentro de las restricciones del modelo—el efecto Procrustes[2]. Un ejemplo es la conformación de nuestras sociedades siguiendo ciertas teorías económicas de moda, que tienen una visión simplista de la realidad, e ignoran muchos otros aspectos importantes de la conducta humana, de la sociedad y la naturaleza.

[1] Pygmalion fue un rey mítico de Chipre, que era también escultor. El rey se enamoró de una de sus obras, una escultura de una mujer joven, y pidió a los dioses que convirtieran su escultura en un ser vivo.

[2] Procrustes es un conocido ladrón de la mitología Griega. Se conoce por la cama donde torturaba a los viajeros que caían en sus manos: si las víctimas eran demasiado pequeñas, les estiraba los brazos y las piernas hasta que se ajustaban a la longitud de la cama y si eran demasiado altas, les cortaba la cabeza y parte de las piernas.

- Olvidar el nivel de precisión del modelo. Todos los modelos tienen hipótesis simplificadoras y, para interpretar los resultados correctamente, hay que tenerlas en cuenta.

Por estas razones, aunque las técnicas analíticas son en general más restrictivas, ya que tienen un dominio de aplicación más estrecho, tales técnicas son más potentes cuando se aplican. Un resultado de simulación es válido sólo para un conjunto particular de datos de entrada. Se necesitan muchas simulaciones para llegar a comprender el sistema aún de manera aproximada. Si las técnicas analíticas son aplicables, deberían usarse en lugar de la simulación o como un complemento a ésta.

## 1.4 Construcción de Modelos

Puesto que la simulación es útil para estudiar el comportamiento de los sistemas, ¿cómo podemos construir los modelos de los sistemas? Este es el tema central de la mayor parte de este libro y del lenguaje Modelica, el cual ha sido creado para simplificar la construcción de modelos y para facilitar su reutilización una vez que han sido construidos.

En principio, existen dos fuentes principales de conocimiento general relacionado con sistemas para la construcción de modelos matemáticos:

- La *experiencia general* reunida en los dominios relevantes de la ciencia y la tecnología, que se encuentra disponible en la literatura y que poseen los expertos en estas áreas. Esto incluye las *leyes de la naturaleza*, por ejemplo, las leyes de Newton para los sistemas mecánicos, las leyes de Kirchhoff para los sistemas eléctricos, las relaciones aproximadas que se aplican a sistemas no técnicos basados en teorías económicas o sociológicas, etc.
- El propio *sistema*, es decir, observaciones y experimentos realizados sobre el sistema que necesitamos modelar.

Además del conocimiento anterior referente al sistema, existe también un conocimiento especializado acerca de la construcción de modelos en dominios y aplicaciones específicos, así como mecanismos genéricos para manejar hechos y modelos. Por ejemplo:

- *Experiencia en la aplicación*—dominar el área y las técnicas de la aplicación, lo cual permite usar todos los hechos relativos a una aplicación de modelado específica.
- *Ingeniería del software y del conocimiento* —conocimiento genérico acerca de cómo definir, manejar, usar, y representar modelos y software. Por ejemplo, la orientación a objetos, las técnicas de componentes de sistemas, la tecnología de los sistemas expertos, etc.

¿Cuál es entonces el *proceso* de análisis y síntesis apropiado que se debe usar para aplicar estas fuentes de información para construir modelos de los sistemas? Generalmente, primero intentamos identificar los componentes principales de un sistema, y las clases de interacción existentes entre estos componentes. Cada componente se divide en subcomponentes hasta que cada parte se ajusta a la descripción de un modelo existente en alguna librería de modelos, o podemos usar leyes apropiadas de la naturaleza u otras relaciones para describir la conducta de ese componente. A continuación, se definen las interfases del componente y una formulación matemática de las interacciones entre los componentes del modelo.

Ciertos componentes podrían tener parámetros del modelo y coeficientes desconocidos o parcialmente conocidos. A menudo, éstos pueden estimarse ajustando datos de medidas experimentales del sistema real al modelo matemático usando *identificación de sistemas*, que en casos simples se reduce a técnicas básicas como ajuste de curvas y análisis de regresión. Sin embargo, versiones más avanzadas de identificación de sistemas pueden incluso determinar la forma del modelo matemático, que se escoge entre un conjunto de estructuras básicas de modelos.

## 1.5 Análisis de Modelos

La simulación es una de las técnicas más comunes de utilizar los modelos para contestar a preguntas acerca de los sistemas. Sin embargo, también hay otros métodos de analizar los modelos, tales como el análisis de la sensibilidad y el diagnóstico basado en modelos, o técnicas matemáticas analíticas, aplicables en aquellos casos donde se pueden encontrar soluciones cerradas en forma analítica.

### 1.5.1 Análisis de la Sensibilidad

El análisis de la sensibilidad trata la cuestión de cuan *sensible* es el comportamiento del modelo a *cambios* en sus parámetros. Esta es una cuestión común en diseño y análisis de sistemas. Por ejemplo, incluso en dominios de aplicación con especificaciones precisas, tales como los sistemas eléctricos, los valores de una resistencia en un circuito tienen una precisión de sólo entre el 5% y el 10%. Si hay una gran sensibilidad en los resultados de las simulaciones a pequeñas variaciones en los parámetros del modelo, se debería sospechar de la validez del modelo. En tales casos, pequeñas variaciones aleatorias en los parámetros del modelo puede conducir a grandes variaciones aleatorias en su comportamiento.

Por otra parte, si la conducta simulada no es muy sensible a pequeñas variaciones en los parámetros del modelo, hay una gran probabilidad de que el modelo refleje de forma bastante precisa la conducta del sistema real. Tal robustez en la conducta es una propiedad deseable cuando se diseñan nuevos productos, porque de otra forma podría ser costoso de fabricar, puesto que ciertas tolerancias deberán mantenerse muy pequeñas. Sin embargo, hay también una serie de ejemplos de sistemas reales que son muy sensibles a variaciones de determinados parámetros específicos del modelo. En esos casos, esta sensibilidad debería también reflejarse en los modelos de aquellos sistemas.

### 1.5.2 Diagnóstico Basado en Modelos

El *diagnóstico* basado en modelos es una técnica que está en parte relacionada con el análisis de sensibilidad. Se trata de analizar el modelo del sistema con el objetivo de encontrar las causas de cierto comportamiento del sistema. En muchos casos, es necesario determinar cuáles son las causas de ciertos comportamientos problemáticos y erróneos. Por ejemplo, consideremos un automóvil, que es un sistema complejo compuesto de muchas partes que interactúan entre sí, tales como un motor, un sistema de ignición, un sistema de transmisión, un sistema de suspensión, ruedas, etc. Bajo un conjunto de condiciones operativas bien definidas, puede considerarse que cada una de estas partes exhibe un comportamiento correcto siempre que el valor de algunas de sus magnitudes se encuentre dentro de ciertos intervalos de valores especificados. Un valor medido o computado fuera de tal intervalo podría indicar un error en ese componente, o en otra parte que influye sobre ese componente. Esta clase de análisis se llama diagnóstico basado en modelos.

### 1.5.3 Verificación y Validación de Modelos

Previamente hemos comentado los peligros que tiene la simulación, por ejemplo, cuando un modelo no es válido para un sistema considerando el objetivo de la simulación. ¿Cómo podemos verificar que el modelo es un modelo bueno y fiable, es decir, que es válido para el fin al que se destina? Esto puede ser muy difícil, y algunas veces podemos confiar sólo en obtener una respuesta parcial a esta pregunta. Sin embargo, las técnicas siguientes son útiles para verificar, al menos parcialmente, la validez de un modelo:

- Revisar críticamente las hipótesis y aproximaciones que hay detrás del modelo, incluyendo la información disponible sobre el dominio de validez de estas hipótesis.
- Comparar, en casos especiales, simplificaciones del modelo con soluciones analíticas.

- Comparar con resultados experimentales, en los casos donde esto sea posible.
- Efectuar análisis de sensibilidad del modelo. Si los resultados de la simulación son relativamente insensibles a pequeñas variaciones de los parámetros del modelo, entonces tenemos fundadas razones para confiar en la validez del modelo.
- Comprobar la consistencia interna del modelo. Por ejemplo, verificar que las dimensiones o unidades son compatibles en las ecuaciones. Así, en la ecuación de Newton $F = m\,a$, la unidad [N] en el lado izquierdo es consistente con [kg m $s^{-2}$] en el lado derecho.

En el último caso, sería posible verificar automáticamente en las herramientas de modelado que las dimensiones son consistentes, siempre que estén disponibles los atributos de unidades para las magnitudes del modelo. Esta funcionalidad aun no está disponible en la mayoría de las herramientas de modelado actuales.

## 1.6 Clases de Modelos Matemáticos

Las diferentes clases de modelos matemáticos pueden ser caracterizados por diferentes propiedades que reflejan el comportamiento de los sistemas que se modelan. Un aspecto importante es si el modelo incorpora propiedades *dinámicas*, dependientes del tiempo, o si es *estático*. Existe otra línea divisoria entre los modelos que evolucionan en el tiempo de manera *continua* y aquellos que cambian en instantes *discretos* de tiempo. Existe una tercera línea de separación entre los modelos *cuantitativos* y los *cualitativos*.

Ciertos modelos describen la *distribución espacial* de las magnitudes, por ejemplo, de la masa. Por el contrario, otros modelos son *concentrados*, en el sentido de que la cantidad distribuida espacialmente es aproximada concentrándola y representándola mediante una única variable, por ejemplo, una masa puntual.

Algunos fenómenos naturales son descritos convenientemente mediante procesos estocásticos y distribuciones de probabilidad, por ejemplo, las transmisiones de radio ruidosas o la física cuántica a nivel atómico. Tales modelos se denominan modelos *estocásticos* o basados en *probabilidad*, y en ellos el comportamiento solo puede ser representado en términos estadísticos. Por el contrario, los modelos *deterministas* permiten representar el comportamiento sin incertidumbres. Sin embargo, incluso los modelos estocásticos pueden ser simulados de forma "determinista" utilizando un computador, puesto que las secuencias de números aleatorios, que a menudo se usan para representar las variables estocásticas, pueden ser generadas tantas veces como se desee de manera idéntica partiendo de los mismos valores semilla.

A menudo, un mismo fenómeno puede ser modelado como estocástico o como determinista, dependiendo del nivel de detalle con el que se estudie. Ciertos aspectos de un nivel son abstraídos

o promediados en el nivel superior. Por ejemplo, considere el modelado de gases con diferentes niveles de detalles, comenzando por el nivel de partículas elementales de la mecánica cuántica, donde las posiciones de las partículas son descritas mediante distribuciones de probabilidad:

- Partículas elementales (orbitales)— modelos estocásticos.
- Átomos (modelo del gas ideal)— modelos deterministas.
- Grupos de átomos (mecánica estadística)— modelos estocásticos.
- Volúmenes de gas (presión y temperatura)— modelos deterministas.
- Gases reales (turbulencia)— modelos estocásticos.
- Mezclador ideal (concentraciones)— modelos deterministas.

Es interesante observar que el tipo de modelo cambia, entre estocástico y determinista, en función de qué aspectos necesitamos estudiar. Los modelos estocásticos detallados se pueden promediar como modelos deterministas cuando son aproximados en el nivel inmediatamente superior de la jerarquía. Por otra parte, conductas estocásticas, tales como la turbulencia, se puede introducir en los niveles macroscópicos como resultado de fenómenos caóticos causados por partes deterministas que interactúan.

### 1.6.1 Clases de Ecuaciones

Los modelos matemáticos normalmente contienen ecuaciones. Hay básicamente cuatro clases principales de ecuaciones, de cada una de las cuales damos un ejemplo a continuación.

Las *ecuaciones diferenciales* contienen derivadas tales como $\frac{dx}{dt}$, denotada por $\dot{x}$. Por ejemplo:

$$\dot{x} = a \cdot x + 3 \tag{1-1}$$

Las *ecuaciones algebraicas* no incluyen ninguna variable diferenciada:

$$x^2 + y^2 = L^2 \tag{1-2}$$

*Las ecuaciones en derivadas parciales* contienen también derivadas con respecto a otras variables distintas del tiempo:

$$\frac{\partial a}{\partial t} = \frac{\partial^2 a}{\partial z^2} \tag{1-3}$$

Las *ecuaciones en diferencias* expresan relaciones entre variables, por ejemplo, en diferentes instantes de tiempo:

$$x(t+1) = 3x(t) + 2 \qquad (1\text{-}4)$$

### 1.6.2 Modelos Dinámicos vs. Estáticos

Todos los sistemas, tanto los naturales como los hechos por el hombre, son dinámicos en el sentido de que existen en el mundo real, que evoluciona en el tiempo. Los modelos matemáticos de tales sistemas serían naturalmente considerados como *dinámicos*, en el sentido de que evolucionan en el tiempo y por lo tanto lo incorporan. Sin embargo, con frecuencia es útil hacer la aproximación de ignorar la dependencia respecto al tiempo en el modelo de un sistema. Tal modelo del sistema se llama *estático*. Podemos definir los conceptos de modelo dinámico y modelo estático como sigue:

- Un modelo *dinámico* incluye el *tiempo* en el modelo. La palabra *dinámico* proviene de la palabra Griega *dynamis*, que significa fuerza y potencia, siendo *dinámico* la interrelación entre fuerzas (dependiente del tiempo). El tiempo se puede incluir explícitamente, como una variable en una formula matemática, o estar presente indirectamente, por ejemplo a través de la derivada respecto del tiempo de una variable, o como eventos que ocurren en ciertos instantes de tiempo.
- Un modelo *estático* se puede definir *sin* involucrar al *tiempo*. La palabra *estático* proviene del Griego *statikos*, que significa algo que crea equilibrio. Los modelos estáticos se suelen utilizar para describir sistemas en estado estacionario o situaciones de equilibrio, donde la salida no cambia si la entrada permanece constante. Sin embargo, los modelos estáticos pueden mostrar un comportamiento dinámico cuando son alimentados por señales de entrada dinámicas.

Es usual que el comportamiento de un modelo dinámico dependa de la historia *previa* de su simulación. Por ejemplo, la presencia de una derivada respecto al tiempo en un modelo matemático significa que esta derivada se debe integrar para resolver la correspondiente variable cuando se simula el modelo. La *integración* toma en cuenta la historia temporal previa. Por ejemplo, este es el caso de los modelos de los condensadores: la tensión en el condensador es proporcional a la carga acumulada en el condensador, esto es, a la integral/acumulación de la corriente en el condensador. Al diferenciar esa relación, se obtiene que la derivada temporal de la tensión del condensador es proporcional a la corriente en el condensador. Podemos estudiar la tensión en el condensador, la cual aumenta a lo largo del tiempo con una velocidad que es proporcional a la corriente, como se muestra en la Figura 1-3.

Otra manera en la cual un modelo puede depender de su historia previa es permitir que los eventos precedentes influyan en el estado actual. Por ejemplo, en el modelo de un sistema ecológico, donde el número de presas en el sistema estará influido por eventos tales como el nacimiento de los depredadores. Por otra parte, un modelo dinámico como es un generador sinusoidal se puede modelar mediante una fórmula que incluya explícitamente el tiempo y que no considere la historia temporal previa.

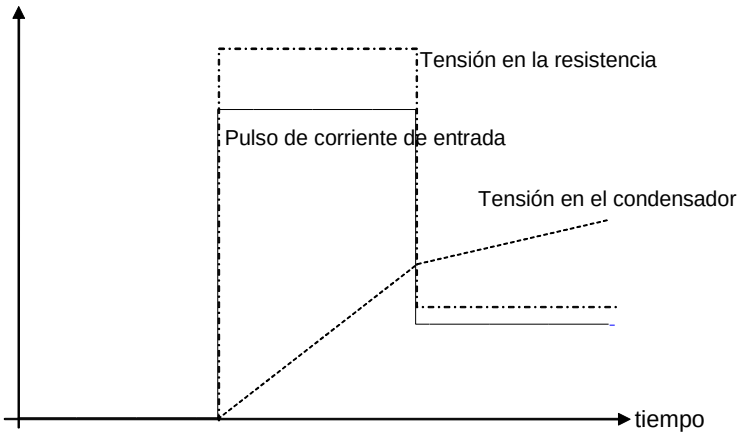



**Figura 1-3.** Una resistencia es un sistema estático, donde la tensión es directamente proporcional a la corriente con independencia del tiempo, mientras que un condensador es un sistema dinámico, donde la tensión es dependiente de la historia anterior en el tiempo.

Una resistencia es un ejemplo de un modelo estático, que se puede formular sin incluir el tiempo. La caída de tensión en una resistencia es directamente proporcional a la corriente que circula a través de la resistencia (vea la Figura 1-3), con independencia del tiempo o de la historia previa.

### 1.6.3 Modelos Dinámicos de Tiempo Continuo vs. de Tiempo Discreto

Hay dos clases principales de modelos dinámicos: los modelos de tiempo continuo y los de tiempo discreto. La clase de modelos de tiempo continuo se puede caracterizar como sigue:

- Los modelos de *tiempo continuo* evolucionan los valores de sus variables de manera continua en el tiempo.

En la Figura 1-4 se representa una variable de un modelo de tiempo continuo denominado A. La formulación matemática de los modelos de tiempo continuo incluye ecuaciones diferenciales con derivadas respecto al tiempo de algunas de las variables del modelo. Muchas leyes de la naturaleza, por ejemplo tal como se expresan en la física, se formulan como ecuaciones diferenciales.

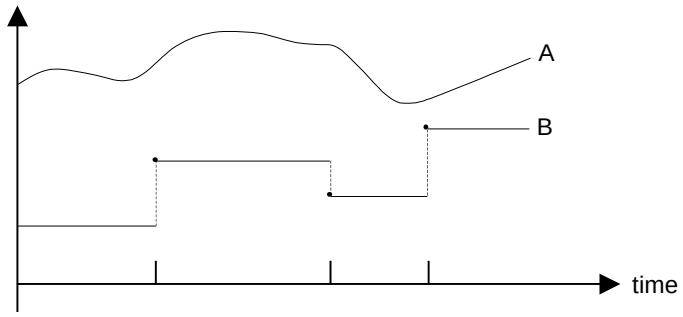



**Figura 1-4.** Un sistema de tiempo discreto B cambia sus valores sólo en ciertos instantes de tiempo, mientras que los valores de los sistemas de tiempo continuo como A evolucionan de manera continua.

La segunda clase de modelos matemáticos es la de los modelos de tiempo discreto, tales como B en la Figura 1-4, donde las variables cambian de valor solo en ciertos instantes de tiempo:

- Los modelos de *tiempo discreto* modifican sus variables en instantes discretos de tiempo.

Los modelos de tiempo discreto se suelen representar por conjuntos de ecuaciones en diferencias. También mediante programas de computador que transformen el estado del modelo en un instante de tiempo al estado en el siguiente instante de tiempo.

En ingeniería aparecen frecuentemente modelos de tiempo discreto, especialmente en los sistemas controlados por computador. Un caso especial muy común es el de los sistemas muestreados, donde un sistema de tiempo continuo es medido en intervalos de tiempo regulares y es aproximado por un modelo de tiempo discreto. Tales modelos muestreados normalmente interaccionan con otros sistemas de tiempo discreto como computadores. Modelos de tiempo discreto pueden también ocurrir naturalmente. Por ejemplo, una población de insectos que cría durante un período corto, una vez al año. En este caso, el período de discretización sería de un año.

### 1.6.4 Modelos Cuantitativos vs. Cualitativos

Todas las diferentes clases de modelos matemáticos analizados previamente en esta sección son de una naturaleza cuantitativa—los valores de la variable se pueden representar numéricamente de acuerdo con una escala medible cuantitativamente.

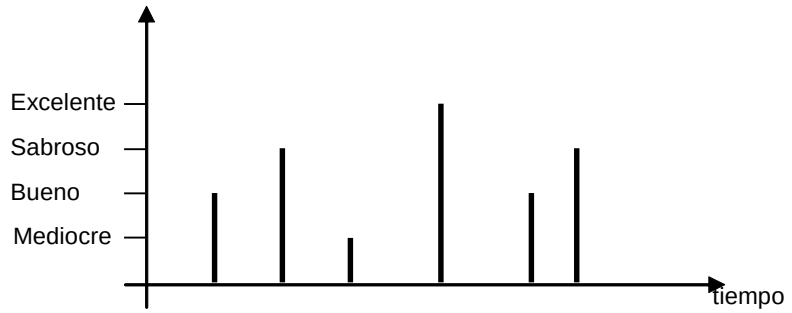



**Figura 1-5.** Calidad del alimento en un restaurante según inspecciones en instantes irregulares.

A otros modelos, como los denominados modelos *cualitativos*, les falta esa clase de precisión. Lo más en lo que podemos confiar es en una clasificación aproximada dentro de un conjunto finito de valores. Por ejemplo, como en el modelo de la calidad de los alimentos representado en la Figura 1-5. Los modelos cualitativos son por naturaleza de tiempo discreto y las variables dependientes están también discretizadas. Sin embargo, incluso si los valores discretos se representan por números en el computador (por ejemplo, mediocre—1, bueno—2, sabroso —3, excelente—4), tenemos que tener en cuenta el hecho de que los valores de las variables en ciertos modelos cualitativos no están necesariamente de acuerdo con una escala medible lineal. Por ejemplo, sabroso no tiene por qué ser tres veces mejor que mediocre.

## 1.7 Utilización del Modelado y la Simulación en el Diseño de Productos

¿Qué papel desempeña el modelado y la simulación en el diseño y el desarrollo de productos industriales? De hecho, las explicaciones previas han tratado ya brevemente este tema. Construir modelos matemáticos en el computador, mediante los denominados *prototipos virtuales*, y simularlos es una forma de determinar y optimizar rápidamente las propiedades de los productos, sin necesidad de construir prototipos físicos costosos. Con frecuencia, este enfoque puede reducir

drásticamente el tiempo de desarrollo y el tiempo de puesta en el mercado, a la vez que aumenta la calidad del producto diseñado.

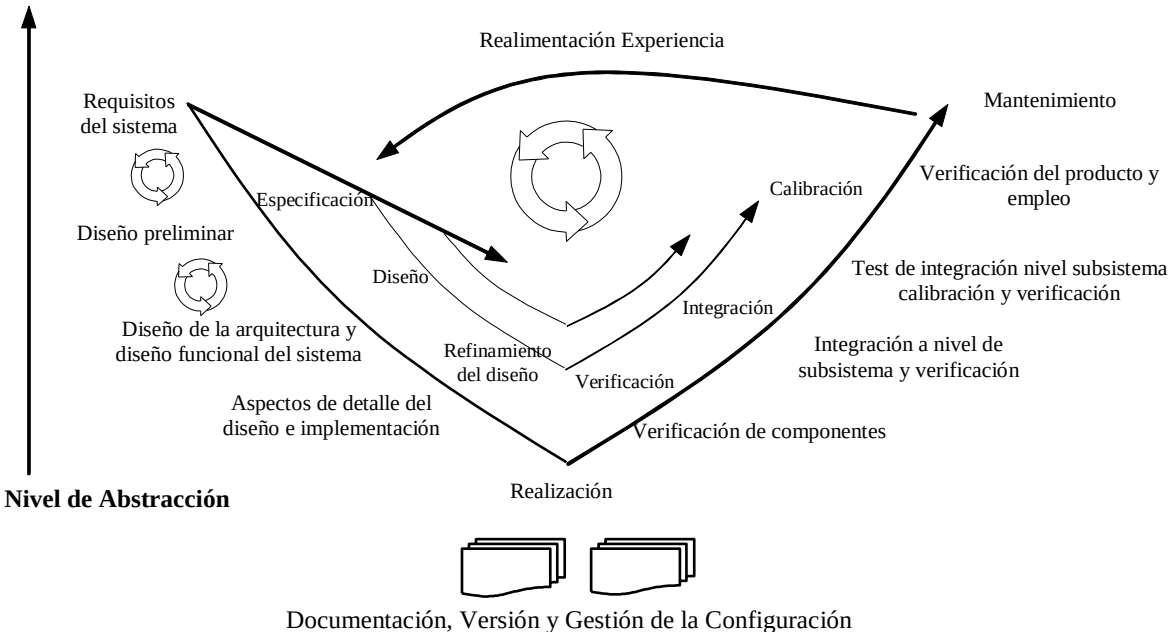



**Figura 1-6.** El diseño-V del producto.

El llamado *diseño-V* de producto, representado en la Figura 1-6, incluye todas las fases estándar del desarrollo del producto:

- Análisis de requisitos y especificación.
- Diseño del sistema.
- Refinamiento del diseño.
- Realización e implementación.
- Verificación y validación de subsistemas.
- Integración
- Calibración del sistema y validación del modelo.
- Empleo del producto.

¿Dónde encaja el modelado y la simulación dentro de este proceso de diseño?

En la primera fase, de *análisis de los requisitos*, se especifican los requisitos funcionales y no funcionales. En esta fase, se identifican los parámetros de diseño importantes y se especifican

requisitos sobre sus valores. Por ejemplo, cuando se diseña un automóvil se pueden plantear requisitos sobre la aceleración, el consumo de combustible, las emisiones máximas, etc. Esos parámetros del sistema también serán parámetros en nuestro modelo del producto diseñado.

En la *fase de diseño del sistema*, especificamos la arquitectura del sistema, es decir, los componentes principales en el sistema y sus interacciones. Si disponemos de una librería de componentes del modelo de simulación, podemos usar estas librerías de componentes en la fase de diseño. En caso contrario, podemos crear nuevos componentes que se adapten al producto diseñado. Este proceso de diseño aumenta de manera iterativa el nivel de detalle en el diseño. Una herramienta de modelado que soporte el modelado y la descomposición jerárquica puede ayudar a manejar la complejidad del sistema.

La *fase de implementación* realizará el producto como un sistema físico y/o como un modelo de prototipo virtual en el computador. Un prototipo virtual se puede realizar antes de que se construya un prototipo físico, normalmente por una pequeña fracción del costo.

En la *fase de verificación y validación de subsistemas*, se verifica el comportamiento de los subsistemas del producto. Se simulan los prototipos virtuales de los subsistemas usando el computador, así como los modelos corregidos si hubiera problemas.

En la *fase de integración* se conectan los subsistemas. En un modelo del sistema para su simulación por computador, se conectan los modelos de los subsistemas entre sí de manera apropiada. Posteriormente, puede simularse el sistema completo y se pueden corregir ciertos problemas de diseño en base a los resultados de la simulación.

En la *fase de calibración y validación* del sistema y del modelo se valida el modelo empleando medidas tomadas de prototipos físicos apropiados. Se calibran los parámetros de diseño y el diseño se suele *optimizar* en cierta medida, de acuerdo con lo especificado en los requisitos originales.

Durante la última fase, de *empleo del producto*, que usualmente sólo aplica a la versión física del producto, éste se envía al cliente con el fin de obtener sus comentarios acerca del producto. En ciertos casos, esto puede también aplicarse a prototipos virtuales, que pueden entregarse y ponerse en un computador que está interactuando con el resto del sistema físico del cliente en tiempo real, lo que se ha llamado simulación HWIL (hardware-in-the-loop).

En la mayoría de los casos, la realimentación de la experiencia se usa para sintonizar tanto los modelos como los productos físicos. Todas las fases del proceso de diseño interactúan continuamente con la base de datos de diseño y del modelo, tal como se muestra en la Figura 1-6.

## 1.8 Ejemplos de Modelos de Sistemas

En esta sección, presentamos brevemente algunos ejemplos de modelos matemáticos de tres áreas de aplicación diferentes, con el fin de ilustrar la potencia del modelado matemático que tiene Modelica y la tecnología de simulación que se describe en el resto de este libro:

- Un sistema termodinámico—parte de un modelo de una turbina de gas industrial GTX100
- Un sistema mecánico 3D con una descomposición jerárquica—un robot industrial.
- Una aplicación bioquímica—parte del ciclo del citrato (ciclo TCA).

En la Figura 1-8 se muestra el diagrama de conexiones del mecanismo de corte de la potencia de la turbina de gas GTX100, y en la Figura 1-7 se muestra dicha turbina de gas.

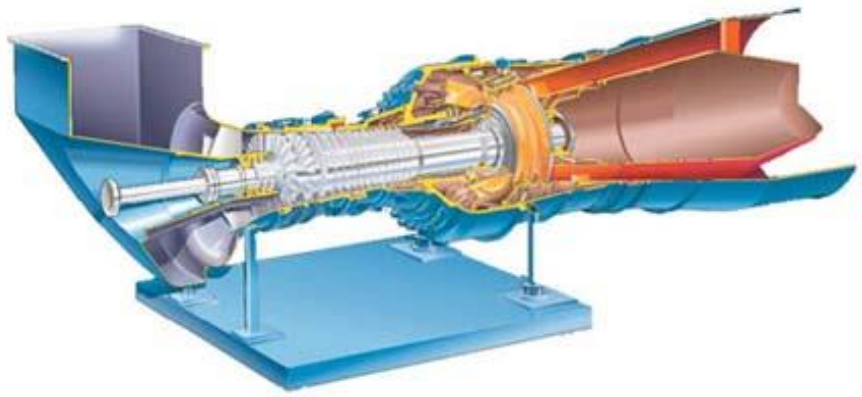


**Figura 1-7**. Un esquema de la turbina de gas GTX100. Cortesía de Alstom Industrial Turbines AB, Finspång, Suecia.

El diagrama de conexiones mostrado en la Figura 1-8 podría no parecer un modelo matemático, pero detrás de cada icono del diagrama hay un modelo del componente, que contiene las ecuaciones que describen su comportamiento.

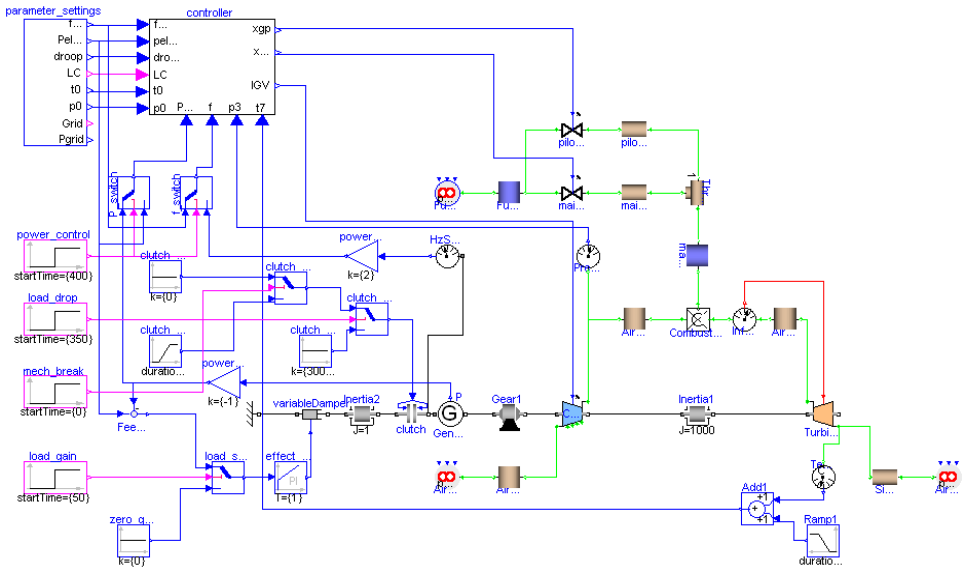



**Figura 1-8**. Detalle del mecanismo de corte de la potencia en un modelo de turbina de gas 40 MW GTX100. Cortesía de Alstom Industrial Turbines AB, Finspång, Suecia.

En la Figure 1-9 se muestran algunas gráficas de las simulaciones de la turbina de gas, que ilustran cómo puede usarse un modelo para investigar las propiedades de un sistema dado.

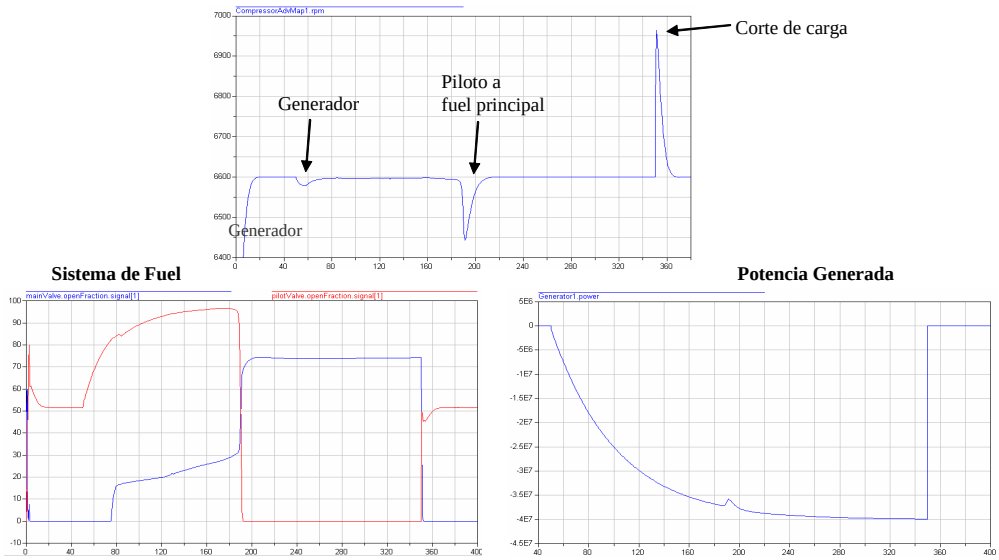



**Figure 1-9**. Simulación del mecanismo de corte del sistema de potencia de la turbina de gas GTX100. Cortesía de Alstom Industrial Turbines AB, Finspång, Suecia

El segundo ejemplo, el robot industrial, ilustra la potencia de la descomposición jerárquica del modelo. El robot tri-dimensional, que se muestra a la derecha de la Figura 1-10, está representado por un diagrama de conexión 2D (en el medio). Cada parte en el diagrama de conexiones puede ser un componente mecánico tal como un motor, una articulación, un sistema de control para el robot, etc. Los componentes pueden consistir en otros componentes, que a su vez se pueden descomponer. En el fondo de la jerarquía tenemos las clases de modelos que contienen las ecuaciones.

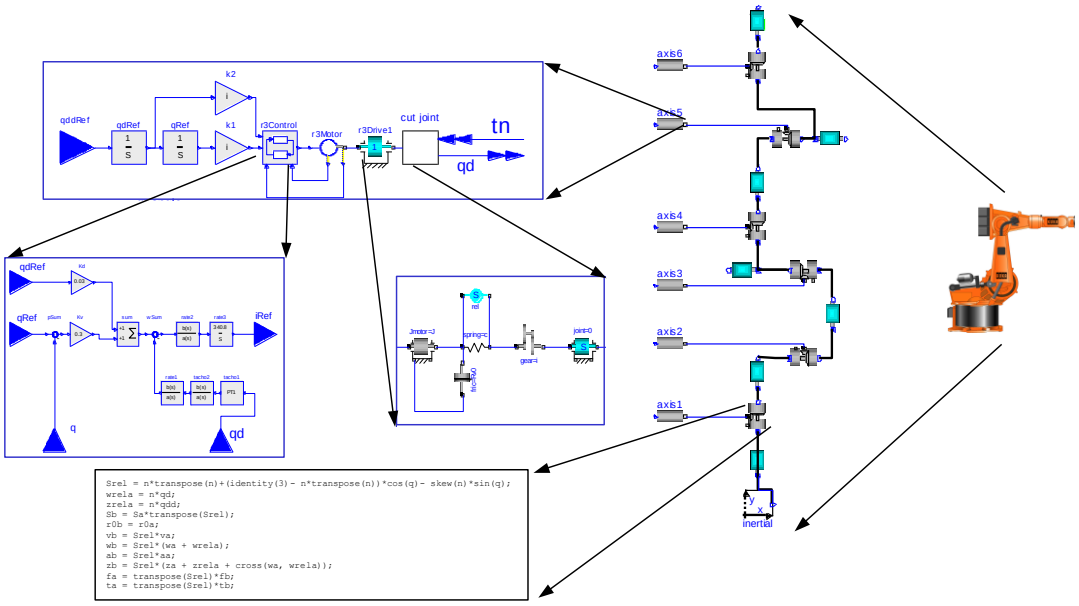



**Figura 1-10.** Modelo jerárquico de un robot industrial. Cortesía de Martin Otter.

El tercer ejemplo es de un dominio completamente diferente—relaciones bioquímicas que describen las reacciones entre reactivos, en este caso particular describen parte del ciclo del citrato (ciclo TCA), tal como está representado en la Figura 1-11.

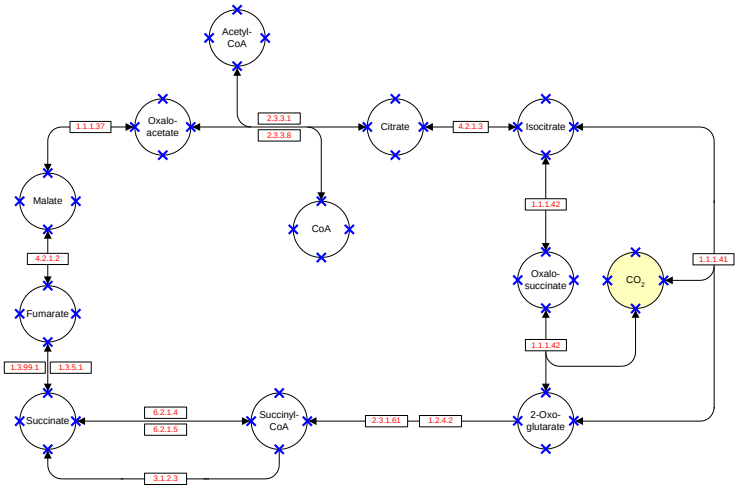



**Figura 1-11**. Modelo de las relaciones bioquímicas de parte del ciclo del citrato (ciclo TCA).

## 1.9 Resumen

Hemos presentado brevemente conceptos importantes tales como sistema, modelo, experimento y simulación. Los sistemas se pueden representar mediante modelos, que pueden ser sometidos a experimentos, es decir, simulación. Ciertos modelos se pueden representar usando las matemáticas, son los llamados modelos matemáticos. Este libro trata de la tecnología, basada en componentes orientados a objetos, para construir y simular dichos modelos matemáticos. Hay diferentes clases de modelos matemáticos: modelos estáticos versus modelos dinámicos, modelos de tiempo continuo versus modelos de tiempo discreto, etc., dependiendo de las propiedades del sistema modelado, la información disponible acerca del sistema y las aproximaciones hechas en el modelo.

## 1.10 Referencias

Cualquier libro sobre modelado y simulación necesita definir conceptos fundamentales, tales como sistema, modelo, y experimento. Las definiciones de este capítulo están generalmente disponibles en textos de modelado y simulación, que incluyen (Ljung y Glad 1994; Cellier 1991).

La cita sobre las definiciones de sistema, con referencia al péndulo, es de (Ashby 1956). El ejemplo de diferentes niveles de detalles en modelos matemáticos de gases, presentado en la Sección 1.6, se presenta en (Hyötyniemi 2002). El proceso de diseño-V de productos, mencionado en la Sección 1.7, se describe en (Stevens et al. 1998; Shumate y Keller 1992). La relación bioquímica del ciclo del citrato de la Figura 1-11 se modeló a partir de la descripción en (Allaby, 1998)

# Capítulo 2

# Un recorrido rápido por Modelica

Modelica es esencialmente un lenguaje de modelado que permite la especificación de modelos matemáticos de sistemas complejos naturales o hechos por el hombre, por ejemplo, con el objetivo de simular en un computador sistemas dinámicos cuyo comportamiento evolucione en función del tiempo. Modelica es también un lenguaje de programación orientado a objetos basado en ecuaciones, orientado hacia aplicaciones computacionales de gran complejidad que requieren elevado rendimiento. Las cuatro características más importantes de Modelica son:

- Modelica se basa fundamentalmente en ecuaciones en lugar de sentencias de asignación. Esto permite el modelado acausal, que posibilita una mayor reutilización de las clases puesto que las ecuaciones no especifican una cierta dirección del flujo de los datos. Así, una clase en Modelica puede adaptarse a más de un contexto de flujo de datos.
- Modelica tiene capacidades de modelado multi-dominio, lo que significa que se pueden describir y conectar componentes de modelos correspondientes a objetos físicos de algunos dominios diferentes, tales como aplicaciones eléctricas, mecánicas, termodinámicas, hidráulicas, biológicas y de control.
- Modelica es un lenguaje orientado a objetos con un concepto de clase general que unifica clases, genéricos —conocidas como plantillas en C++ —y subtipos generales en una única construcción del lenguaje. Esto facilita la reutilización de los componentes y el desarrollo de los modelos.

- Modelica tiene un modelo de componentes de software, con construcciones para crear y conectar los componentes. Resulta muy apropiado como un lenguaje de descripción de la arquitectura de sistemas físicos complejos y en cierta medida de sistemas de software.

Estas son las propiedades principales que hacen que Modelica sea potente y fácil de usar, especialmente para el modelado y la simulación. Comenzaremos con una introducción sencilla a Modelica desde el principio.

## 2.1 Comenzando con Modelica

Los programas en Modelica se construyen a partir de clases, también llamadas modelos. A partir de la definición de una clase es posible crear cualquier número de objetos, que se conocen como instancias de esa clase. Piense en una clase como una colección de planos e instrucciones usadas por una fábrica para crear objetos. En este caso, el compilador de Modelica y el sistema de ejecución serían la fábrica.

Una clase en Modelica contiene elementos, entre los cuales la declaración de las variables y las secciones *equation*, que contienen ecuaciones, son los principales. Las variables contienen datos que pertenecen a las instancias de la clase; constituyen el almacenamiento de datos de la instancia. Las ecuaciones de una clase especifican el comportamiento de las instancias de esa clase.

Es una larga tradición que el primer programa de demostración en cualquier lenguaje de computador es un programa trivial que imprime la cadena `"Hola Mundo"`. Como Modelica es un lenguaje basado en ecuaciones, imprimir una cadena de caracteres no tiene mucho sentido. En lugar de esto, nuestro programa en Modelica Hola Mundo resuelve una *ecuación diferencial* trivial:

$$\dot{x} = -a \cdot x \tag{2-1}$$

La variable $x$ en esta ecuación es una variable dinámica (también es una variable de estado) que puede cambiar su valor con el tiempo. La derivada en el tiempo $\dot{x}$ es la derivada respecto de t de `x`, representada en Modelica como `der(x)`. Como todos los programas en Modelica, llamados *modelos*, consisten en declaraciones de clases, el programa `HolaMundo` se declara como una clase usando la palabra reservada *class*:

```
class HolaMundo
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = -a*x;
end HolaMundo;
```

Utilice su editor de texto favorito o el entorno de programación de Modelica para escribir este código[3] Modelica. También puede abrir el documento electrónico DrModelica, que contiene todos los ejemplos y ejercicios de este libro. Luego, ejecute la orden de simulación en su entorno de Modelica. Éste compilará el código de Modelica a algún código intermedio, normalmente código C, que a su vez se compilará a código máquina y se ejecutará junto con un solucionador numérico de ecuaciones diferenciales ordinarias (ODE) o de ecuaciones diferenciales algebraicas (DAE) para producir una solución de x como función del tiempo. La siguiente orden permite obtener en los entornos MathModelica u OpenModelica la solución entre t = 0 y t = 2:

```
simulate[4](HolaMundo,stopTime=2)
```

Como la solución para `x` es una función del tiempo, se puede representar ejecutando una orden plot:

```
plot[5](x)
```

(o en formato más largo `plot(x,xrange={0,2})` que especifica el eje-x), que da la curva de la Figura 2-1:

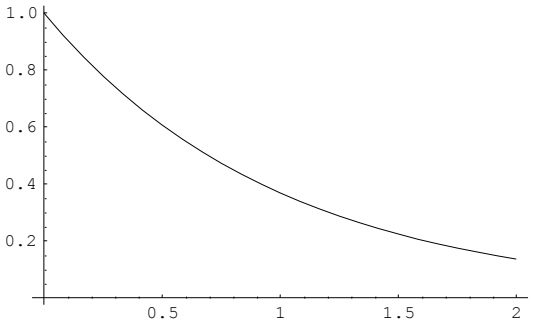



**Figura 2-1.** Gráfica de una simulación del modelo sencillo `HolaMundo`.

---

[3] Hay un ambiente Modelica de código abierto OpenModelica que se puede descargar de www.openmodelica.org, Wolfram System Modeler es de Wolfram Research y MathCore (www.mathcore.com or www.wolfram.com), Dymola es de Dassault Systémès (www.dymolca.com). Antes de simular un modelo existente, usted necesita abrirlo o cargarlo, e.g. `loadModel(HelloWorld)` en OpenModelica.

[4] `simulate` es la orden para simulación en OpenModelica. Esta misma orden en Wolfram System Modeer con estilo Mathematica sería `M1=WSMSimulate["HolaMundo",{0,2}]`, y en Dymola sería `simulateModel("HolaMundo", stopTime=2)`.

[5] `plot` es la orden para graficar resultados de simulación en OpenModelica. Esta misma orden en Wolfram SystemModeler con estilo Mathematica sería `WSMPlot[M1, "x"]` y en Dymola sería `plot({"x"})`.

Ya tenemos un pequeño modelo en Modelica que hace algo, ¿pero qué significa realmente? El programa contiene una declaración de una clase llamada `HolaMundo`, que tiene dos variables y una única ecuación. El primer atributo de la clase es la variable `x`, que se inicializa con un valor inicial de 1 en el instante en que la comienza simulación. Todas las variables en Modelica tienen un atributo `start` con un valor por defecto que normalmente se fija a 0. Puede asignarse valor a este atributo para fijar un valor inicial diferente, incluyéndolo entre paréntesis después del nombre de la variable. Esta es una ecuación de modificación, fija el valor del atributo start a 1 y reemplaza la ecuación por defecto original para el atributo.

El segundo atributo es la variable `a`, que es una constante que se inicializa a 1 en el comienzo de la simulación. Tal constante lleva como prefijo la palabra clave `parameter` para indicar que es constante durante la simulación, pero que es un parámetro del modelo que se puede cambiar entre simulaciones. Por ejemplo, mediante una orden en el entorno de simulación se puede volver a ejecutar la simulación con un valor diferente de `a`.

Observe también que, cuando se declara la variable, ésta tiene un tipo que precede a su nombre cuando se declara la variable. En este caso, tanto la variable `x` como la "variable" `a` son del tipo `Real`.

La única ecuación en este ejemplo `HolaMundo` especifica que la derivada de `x` es igual a la constante `-a` multiplicada por `x`. En Modelica, el signo igual = siempre significa igualdad. Es decir, establece una ecuación, y no una asignación como en muchos otros lenguajes. La derivada con respecto al tiempo de una variable se representa mediante la pseudofunción `der( )`.

Nuestro segundo ejemplo es ligeramente más complicado, ya que en lugar de una ecuación contiene cinco (2-2):

$$\begin{aligned} m\dot{v}_x &= -\frac{x}{L}F \\ m\dot{v}_y &= -\frac{y}{L}F - mg \\ \dot{x} &= v_x \\ \dot{y} &= v_y \\ x^2 + y^2 &= L^2 \end{aligned} \qquad (2\text{-}2)$$

Este ejemplo es realmente un modelo matemático de un sistema físico: un péndulo planar como el representado en la Figura 2-2.

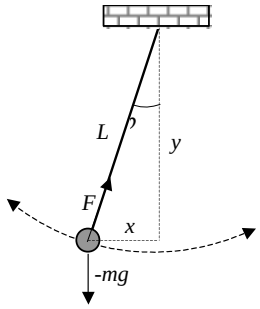



**Figura 2-2.** Un péndulo planar.

Estas ecuaciones corresponden a las ecuaciones de movimiento de Newton para la masa del péndulo bajo la influencia de la gravedad junto con una restricción geométrica: la 5ª ecuación. Esta ecuación, $x^2 + y^2 = L^2$, especifica que la posición de la masa (*x*,*y*) debe estar en una circunferencia de radio *L*. Las variables $v_x$ y $v_y$ son las velocidades de la masa en las direcciones *x* e *y* respectivamente.

Este modelo posee una propiedad interesante. Es el hecho de que la 5ª ecuación es de una clase diferente. Es una *ecuación algebraica* que sólo contiene fórmulas algebraicas en las que intervienen las variables, pero no sus derivadas. Las cuatro primeras ecuaciones de este modelo son ecuaciones diferenciales, como en el ejemplo `HolaMundo`. Los sistemas de ecuaciones que contienen ecuaciones diferenciales y algebraicas se llaman *sistemas de ecuaciones algebraico diferenciales* (DAEs). El siguiente es un modelo en Modelica del péndulo:

```
class Pendulo  "péndulo planar"
  constant  Real PI=3.141592653589793;
  parameter Real m=1, g=9.81, L=0.5;
  Real F;
  output    Real x(start=0.5),y(start=0);
  output    Real vx,vy;
equation
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-m*g;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end Pendulo;
```

Simulamos el modelo `Pendulo` y representamos la coordenada-x, mostrada en la Figura 2-3:

```
simulate(Pendulo, stopTime=4)
plot(x);
```

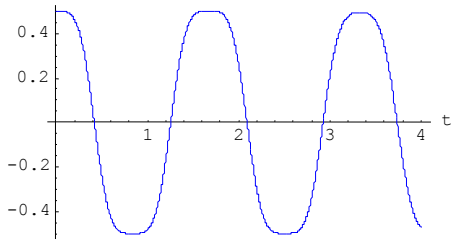



**Figura 2-3.** Gráfica de la simulación del modelo DAE (ecuaciones algebraico diferenciales) del Péndulo.

También pueden escribirse sistemas de ecuaciones DAE sin significado físico, con ecuaciones que contengan fórmulas seleccionadas más o menos aleatoriamente, como en la clase `DAEejemplo` mostrada a continuación:

```
class DAEejemplo
  Real x(start=0.9);
  Real y;
equation
  der(y) + (1+0.5*sin(y))*der(x) = sin(time);
  x-y = exp(-0.9*x)*cos(y);
end DAEejemplo;
```

Esta clase contiene una ecuación diferencial y una algebraica. Simúlelo y represéntelo gráficamente ¡para ver si aparece alguna curva razonable!

Finalmente, una observación importante con respecto a los modelos en Modelica:

- ¡El número de *variables* debe ser igual al número de *ecuaciones*!

Esta afirmación es cierta para los tres modelos vistos hasta el momento, y se cumple para todos los modelos en Modelica que son resolubles. Por variables entendemos aquellas magnitudes que pueden variar, es decir, no las constantes ni los parámetros descritos en la Sección 2.1.3.

### 2.1.1 Variables y Tipos Predefinidos

Este ejemplo muestra un modelo algo más complicado, que describe un oscilador de Van der Pol[6]. Obsérvese que se emplea la palabra clave `model` en lugar de `class` con prácticamente el mismo significado.

[6] Balthazar van der Pol fue un ingeniero eléctrico Holandés que inició la dinámica experimental moderna en el laboratorio durante 1920s y 1930s. Van der Pol investigó circuitos eléctricos empleando tubos de vacío y encontró

```
model VanDerPol  "Modelo del oscilador de Van der Pol"
  Real x(start = 1) "Cadena descriptiva para x"; // x comienza en 1
  Real y(start = 1) "Cadena descriptiva para y"; // y comienza en 1
  parameter Real lambda = 0.3;
equation
  der(x) = y;                          // Esta es la primera ecuación
  der(y) = -x + lambda*(1 - x*x)*y; /*La 2ª ecuación diferencial */
end VanDerPol;
```

Este ejemplo contiene la declaración de dos variables dinámicas (que son también variables de estado): `x` e `y`. Ambas son del tipo `Real` y toman el valor 1 al comienzo de la simulación, que se produce normalmente en el instante de tiempo 0. En la siguiente línea, se encuentra la declaración del parámetro constante lambda, que es un parámetro del modelo.

La palabra clave `parameter` especifica que la variable es constante durante la ejecución de una simulación, pero que se puede inicializar antes del inicio de una simulación o entre simulaciones. Esto significa que `parameter` es una clase especial de constante, que se implementa como una variable estática que se inicializa una única vez y nunca cambia su valor durante una determinada ejecución. Un `parameter` es una variable constante que facilita al usuario la modificación de la conducta de un modelo. Por ejemplo, cambiando el parámetro lambda se influye mucho en la conducta del oscilador de Van der Pol. Por el contrario, una constante fija en Modelica declarada con el prefijo `constant` nunca cambia y se puede sustituir por su valor siempre que ocurra.

Finalmente presentamos declaraciones de tres variables ficticias, simplemente para mostrar variables de diferentes tipos de datos además del tipo `Real`. La variable booleana `bb` tiene un valor inicial por defecto de `false` si no se especifica nada más. La variable de tipo String `dummy`, que es siempre igual a `"dummy string"`. La variable de tipo entero `fooint`, que es siempre igual a `0`.

```
Boolean bb;
String dummy  = "dummy string";
Integer fooint = 0;
```

Modelica tiene tipos de datos “primitivos” predefinidos: números en coma flotante, enteros, booleanos y cadenas de caracteres. Existe también el tipo Complex para calculos con numerous complejos, el cual está predefinido en una librería. Estos tipos primitivos contienen datos que Modelica comprende directamente, en contraste con las clases de tipos definidas por

---

que tenían oscilaciones estables, que ahora se llaman ciclos límites. El oscilador de van der Pol es un modelo que desarrolló para describir la conducta de circuitos no lineales de tubos de vacío.

programadores. El tipo de cada variable debe declararse explícitamente. Los tipos de datos primitivos de Modelica son:

| | |
|---|---|
| `Boolean` | `true` o `false` |
| `Integer` | corresponde con el tipo de dato int de C, normalmente 32-bit en complemento a 2 |
| `Real` | corresponde con el tipo de dato double de C, normalmente 64-bit en coma flotante |
| `String` | cadena de caracteres de texto |
| `enumeration(...)` | tipo enumeración de literales enumerados |
| `Complex` | Para calculus con numerous complejos, un tipo básico predefinido en una librería |

Finalmente, hay una sección de ecuaciones que comienza con la palabra clave `equation`, que contiene dos ecuaciones mutuamente dependientes que definen la dinámica del modelo.

Para ilustrar el comportamiento del modelo, ejecutamos una orden para simular el oscilador de Van der Pol durante 25 segundos, comenzando en el instante de tiempo 0:

```
simulate(VanDerPol, stopTime=25)
```

Un diagrama en el plano de fase de las variables de estado para el modelo del oscilador de Van der Pol (véase la Figura 2-4):

```
plotParametric(x,y, stopTime=25)
```

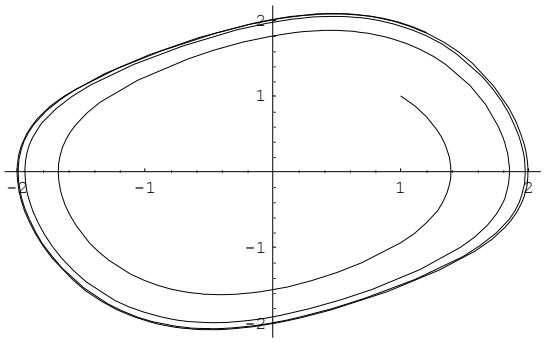



**Figura 2-4.** Representación paramétrica de una simulación del oscilador de Van der Pol.

Los nombres de variables, funciones, clases, etc. se denominan identificadores. Hay dos maneras de construir los identificadores en Modelica. La forma más común es que el identificador comience con una letra, seguida por letras o dígitos, por ejemplo, `x2`. La segunda forma es

comenzar con una comilla simple, seguida por cualesquiera caracteres, y terminar con otra comilla simple. Por ejemplo, `'2nd*3'`.

### 2.1.2 Comentarios

Los comentarios del código son el texto descriptivo arbitrario, por ejemplo escrito en castellano, que se inserta en el programa de computador. Modelica tiene tres estilos de comentarios, tal como se ilustra en el ejemplo previo `VanDerPol`.

Los comentarios permiten escribir texto descriptivo junto con el código, lo que facilita al usuario la utilización del modelo o a los programadores comprender en el futuro el código escrito. Ese programador puede ser usted mismo meses o años después. Usted ahorrará esfuerzo en el futuro comentando su propio código. También, con frecuencia, uno mismo encuentra errores en su propio código cuando escribe los comentarios, ya que al explicarlo uno se ve forzado a pensar en el programa una vez más.

La primera clase de comentarios es una cadena de caracteres entre comillas. Por ejemplo, "`un comentario`". Opcionalmente, aparecen después de las declaraciones de variables o al comienzo de las declaraciones de clases. Aquellos son "comentarios de definición" que se procesan para que sean utilizados por el entorno de programación de Modelica. Por ejemplo, para aparecer en menús o como textos de ayuda para el usuario. Desde un punto de vista sintáctico, no son realmente comentarios, ya que forman parte de la sintaxis del lenguaje. En el ejemplo previo, tales comentarios de definición aparecen para la clase `VanDerPol` y para las variables `x` e `y`.

Los otros dos tipos de comentarios son ignorados por el compilador de Modelica. Simplemente están para beneficio de los programadores de Modelica. El compilador ignora el texto que sigue a `//` hasta el final de la línea. También ignora el texto `/* y el texto */`. Este último tipo de comentarios se puede utilizar cuando hay secciones grandes de texto, que ocupan varias líneas.

Finalmente, debemos mencionar una construcción llamada `annotation`. Se trata de un tipo de "comentario" estructurado, que puede almacenar información conjuntamente con el código y que se describe en la Sección 2.17.

### 2.1.3 Constantes

Las constantes en Modelica pueden ser números enteros, tales como 4, 75, 3078; números en coma flotante, como 3.14159, 0.5, 2.735E-10, 8.6835e+5; cadenas de caracteres, como `"hola mundo"`, `"rojo"`; y valores enumerados, tales como `Colores.rojo`, `tallas.grande`.

Los programadores prefieren las constantes con nombre por dos razones. Una razón es que el nombre de la constante es una forma de documentación, que se puede emplear para describir para qué se usa ese valor en particular. La otra, que quizás es incluso una razón más importante, es que una constante con nombre se define en un único lugar en el programa. Cuando se necesita cambiar o corregir la constante se hace solamente en un lugar, simplificando así el mantenimiento del programa.

Las constantes con nombre en Modelica se crean utilizando en las declaraciones uno de los prefijos `constant` o `parameter`, y proporcionando una ecuación de declaración como parte de dicha declaración. Por ejemplo:

```
constant  Real    PI = 3.141592653589793;
constant  String  colorrojo = "rojo";
constant  Integer uno = 1;
parameter Real    masa = 22.5;
```

Las constantes del tipo parámetro pueden ser declaradas sin una ecuación de declaración, puesto que su valor se puede definir, por ejemplo, leyendolas de un archivo antes de que la simulación comience. Por ejemplo:

```
parameter Real masa, gravedad, longitud;
```

### 2.1.4 Variabilidad

Hemos visto que algunas variables pueden cambiar su valor en cualquier punto en el tiempo, mientras que las constants con nombre son mas o menos constants. De hecho, hay un concepto general de cuatro niveles de variabilidad de variables y expresiones en Modelica:

- Expresiones o variables con *variabilidad continua en el tiempo*, las cuales pueden cambiar en cualquier punto en el tiempo.
- *Variabilidad en Tiempo Discreto* significa que los cambios en los valores pueden ocurrir solamente en lo que llamamos eventos; ver la sección 2.15
- *Variabilidad de Parámetros* significa que el valor se puede cambiar en la inicialización antes de la simulación, pero es fija durante la simulación.
- *Variabilidad Constante* significa que el valor es fijo siempre.

### 2.1.5 Valores Iniciales por Defecto

Si se declara una variable numérica sin asignarle un valor o inicializarla a través del modificador `start` en su declaración, entonces se inicializa por defecto a cero al inicio de la simulación. Las variables booleanas tienen `false` como valor por defecto de `start`, y las variables del tipo cadena tienen la cadena vacía `""` si no se especifica nada.

Son excepciones a esta regla los *resultados* de las funciones y las variables *locales* de las funciones, en los cuales el valor inicial por defecto en la llamada a la función está *indefinido*.

## 2.2 Modelado Matemático Orientado a Objetos

Los lenguajes de programación orientados a objetos tradicionales como Simula, C++, Java, y Smalltalk, así como los lenguajes procedurales tales como Fortran o C, permiten la programación de operaciones sobre los datos almacenados. Los datos almacenados del programa incluyen valores de variables y datos de objetos. El número de objetos a menudo cambia dinámicamente. El punto de vista de Smalltalk de la orientación a objetos enfatiza el envío de mensajes entre los objetos creados (dinámicamente).

La filosofía de Modelica respecto a la orientación a objetos es diferente, puesto que el lenguaje Modelica se basa en el modelado matemático *estructurado*. La orientación a objetos se puede ver como un concepto de estructuración, que se utiliza para manejar la complejidad que entraña la descripción de los sistemas grandes. Un modelo en Modelica es fundamentalmente una descripción matemática declarativa, que simplifica mucho el análisis posterior. Las propiedades de los sistemas dinámicos se expresan de una forma declarativa mediante ecuaciones.

El concepto de programación *declarativa* está inspirado en las matemáticas, donde es común decir o declarar qué es lo que se *mantiene*, en lugar de dar un *algoritmo* detallado paso a paso de *cómo* lograr el objetivo deseado, tal como se precisa cuando se usan los lenguajes procedurales. Esto libera al programador de la tarea de mantener el control de estos detalles. Más aun, el código se hace más conciso y fácil de modificar sin introducir errores.

Así pues, la visión de la orientación a objetos declarativa de Modelica, desde el punto de vista del modelado orientado a objetos, se puede resumir como sigue:

- La orientación a objetos se usa fundamentalmente como un concepto de *estructuración*, que emplea la estructura declarativa y reutilización de modelos matemáticos. Nuestras tres formas de estructuración son jerarquía, conexiones de componentes y herencia.

- Las propiedades de los modelos dinámicos se expresan de una manera declarativa mediante *ecuaciones*[7].
- Un objeto es una colección de *instancias* de variables y ecuaciones que comparten un conjunto de datos.

Sin embargo:

- La orientación a objetos en el modelado matemático *no* se ve como un paso de mensajes dinámicos.

La forma declarativa orientada a objetos de describir los sistemas y su comportamiento que ofrece Modelica está en un nivel de abstracción más elevado que el que es usual en la programación orientada a objetos, puesto que se pueden omitir algunos detalles de implementación. Por ejemplo, no se necesita escribir explícitamente código para transferir datos entre objetos a través de código de sentencias de asignación o paso de mensajes. Tal código se genera automáticamente por el compilador de Modelica, basándose en las ecuaciones dadas.

De la misma forma que en los lenguajes orientados a objetos ordinarios, las clases son como plantillas para crear objetos. Tanto las variables como las ecuaciones pueden ser heredadas entre clases. También, pueden heredarse las definiciones de las funciones. Sin embargo, la especificación del comportamiento se hace básicamente mediante ecuaciones, en lugar de vía métodos. Existen también mecanismos para expresar código algorítmico incluyendo funciones en Modelica, pero esto es más la excepción que la regla. En el Capítulo 3 se incluye la explicación de ciertos conceptos relativos a la orientación a objetos.

## 2.3 Clases e Instancias

Modelica, como cualquier lenguaje de computador orientado a objetos, proporciona las nociones de clases y objetos, también llamadas instancias, como una herramienta para resolver problemas de modelado y programación. Cada objeto en Modelica tiene una clase que define sus datos y su comportamiento. Una clase tiene tres tipos de miembros:

- Las variables de datos asociadas con una clase y sus instancias. Las variables representan los resultados de los cálculos obtenidos por la resolución de las ecuaciones de una clase conjuntamente con las ecuaciones de otras clases. Durante la resolución numérica de los

[7] También se permiten los algoritmos, pero de una forma que hace possible considerar una sección algoritmo como un sistema de ecuaciones.

problemas dependientes del tiempo, las variables almacenan los resultados del proceso de solución en el instante de tiempo actual.

- Las ecuaciones que especifican la conducta de una clase. La forma en que las ecuaciones interactúan con ecuaciones de otras clases determina el proceso de solución, es decir, la ejecución del programa.
- Las clases pueden ser miembros de otras clases.

Aquí está la declaración de una clase simple, que podría representar un punto en un espacio tri-dimensional:

```
class Punto "Punto en un espacio tri-dimensional"
 public
  Real x;
  Real y, z;
end Punto;
```

La clase `Punto` tiene tres variables que representan las coordenadas `x`, `y`, y `z` de un punto y no tiene ecuaciones. Una declaración de clase como esta es como una plantilla que define cómo son las instancias que se crean a partir de ella, así como las instrucciones, en la forma de ecuaciones, que definen el comportamiento de esos objetos. Puede accederse a los miembros de una clase utilizando la notación punto (`.`). Por ejemplo, si se considera la instancia `miPunto` de la clase `Punto`, se puede acceder a la variable `x` escribiendo `miPunto.x`.

Los miembros de una clase pueden tener dos niveles de visibilidad. La declaración `public` de `x`, `y`, y `z`, que es la declaración por defecto, significa que cualquier código con acceso a una instancia de `Punto` puede referenciar a estos miembros. El otro nivel posible de visibilidad, especificado mediante la palabra clave `protected`, significa que sólo se permite el acceso a estos miembros desde el código interno de la clase, así como desde el código de las clases que hereden a esta clase.

Observe que la aparición de una de las palabras claves `public` o `protected` quiere decir que todas las declaraciones de elementos que siguen a esa palabra clave asumen la visibilidad correspondiente hasta otra aparición de una de esas palabras, o hasta que se alcance el final de la clase que contiene las declaraciones.

### 2.3.1 Creación de Instancias

En Modelica, los objetos se crean implícitamente simplemente declarando las instancias de las clases. Esto es diferente en los lenguajes orientados a objetos, como Java o C++, donde la creación de objetos se especifica utilizando la palabra clave new. Por ejemplo, para crear tres

instancias de nuestra clase `Punto`, simplemente declaramos tres variables del tipo `Punto` en una clase, que aquí es la clase `Triangulo`:

```
class Triangulo
  Punto  punto1;
  Punto  punto2;
  Punto  punto3;
end Triangulo;
```

Sin embargo, queda un problema. ¿En qué contexto debería instanciarse Triangulo y cuándo debería interpretarse como una clase incluida en una librería, que no se debe instanciar hasta que realmente se utilice?

Este problema se resuelve considerando la clase *superior* de la jerarquía de instanciación en el programa Modelica que se va a ejecutar como un tipo de clase "main", que siempre se instancia implícitamente, implicando que sus variables se instancian, y que las variables de aquellas variables se instancian, etc. Por tanto, para instanciar `Triangulo`, o bien se hace que la clase `Triangulo` sea la clase "superior" o se declara una instancia de `Triangulo` en la clase "main". En el ejemplo que sigue, la clase `Triangulo` y la clase `Foo1` ambas se instancian.

```
class Foo1
  ...
end Foo1;

class Foo2
  ...
end Foo2;
...

class Triangulo
  Punto  punto1;
  Punto  punto2;
  Punto  punto3;
end Triangulo;

class Main
  Triangulo  pts;
  Foo1       f1;
end Main;
```

Las variables de las clases de Modelica se instancian para cada objeto. Esto significa que una variable en un objeto es distinta de la variable con el mismo nombre en cada uno de los demás objetos de esa clase que han sido instanciados. Muchos lenguajes orientados a objetos permiten variables de la clase. Tales variables son específicas a una clase, en oposición a las instancias de

la clase, y se comparten entre todos los objetos de esa clase. La noción de variables de clase no está aún disponible en Modelica.

### 2.3.2 Inicialización

Otro problema es la inicialización de variables. Como se ha mencionado previamente en la Sección 2.1.5, si no se especifica nada diferente, el valor inicial por defecto de todas las variables numéricas es cero, con la excepción de los resultados de las funciones y sus variables locales, para los cuales no está definido un valor por defecto en el momento de invocacion. Es posible especificar otros valores iniciales diferentes usando el atributo `start` de las variables instanciadas. Observe que el valor `start` sólo es una sugerencia para el valor inicial—el solucionador puede escoger un valor diferente, a menos que el atributo `fixed` sea true para esa variable. En el ejemplo que se muestra a continuación, se especifica esto para la clase `Triangulo`:

```
class Triangulo
  Punto  punto1(start={1,2,3});
  Punto  punto2;
  Punto  punto3;
end Triangulo;
```

Alternativamente, el valor `start` de `punto1` puede especificarse cuando se instancia `Triangulo`, tal como se muestra a continuación:

```
class Main
  Triangulo  pts(punto1.start={1,2,3});
  foo1       f1;
end Main;
```

Una forma más general de inicializar un conjunto de variables teniendo en cuenta algunas restricciones, es especificar un sistema de ecuaciones que debe ser resuelto con el fin de obtener los valores iniciales de estas variables. Este método está soportado en Modelica mediante la construcción `initial equation`.

A continuación se muestra un ejemplo. Se trata de un controlador de tiempo continuo que es inicializado en estado estacionario, para lo cual se impone que la derivada sea nula.

```
model Controlador
  Real y;
equation
  der(y) = a*y + b*u;
```

```
initial equation
  der(y)=0;
end Controlador;
```

Este modelo tiene la siguiente solución en el instante inicial:

```
der(y) = 0;
y = -(b/a)*u;
```

### 2.3.3 Clases Especializadas

El concepto de clase es fundamental en Modelica y se emplea para diferentes propósitos. Prácticamente cualquier cosa en Modelica es una clase. Sin embargo, con el fin de hacer el código de Modelica más fácil de leer y mantener, se han introducido palabras claves especiales para determinados usos específicos del concepto clase. Las palabras claves `model`, `connector`, `record`, `block`, `type`, `package` y `function` pueden ser usadas en lugar de `class` bajo las condiciones apropiadas, llamadas restricciones. Algunas de las clases especializadas también tienen funcionalidades adicionales, llamadas mejoramientos. Por ejemplo, una clase `function` tiene el mejoramiento de que puede ser llamado, mientras que una clase `record` es una clase utilizada para declarar una estructura de datos tipo registro y tiene la restricción de que no puede contener ecuaciones.

```
record Persona
  Real   edad;
  String nombre;
end Persona;
```

Un `model` es lo mismo que una clase, es decir, estas palabras clave son completamente intercambiables. Un `block` es una clase cuya causalidad computacional es fija. Esto significa que se especifica, para cada variable de la clase, si tiene una causalidad de entrada o de salida. Así, en la declaración de cada variable de la interfaz en una clase `block` debe emplearse una palabra clave, que fija su causalidad como `input` o `output`.

Una clase `connector` se usa para declarar la estructura de "puertos" o puntos de interfaz de un componente, y no puede contener ecuaciones. Pero tiene la propiedad adicional de permitir el uso de la función `connect(…)` a otras instancias de clase `connector`. Un `type` es una clase que puede ser un alias o una extensión a un tipo, registro o arreglo predefinido. Por ejemplo:

```
type vector3D = Real[3];
```

La idea de clases especializadas es beneficiosa, puesto que el usuario no tiene que aprender más que un concepto: el *concepto de clase*. La noción de clases especializadas permite al usuario

expresar de forma más precisa cuál es el objetivo de la clase, y requiere que el compilador de Modelica compruebe que las restricciones de estos usos realmente se satisfacen. Afortunadamente, la noción es bastante uniforme puesto que todas las propiedades básicas de una clase, tales como la sintaxis y semántica de la definición, instanciación, herencia, y propiedades genéricas, son idénticas para todos los tipos de clases especializadas. Más aún, la construcción de traductores de Modelica se simplifica, porque sólo se tiene que implementar la sintaxis y la semántica del concepto de clase, junto con algunas comprobaciones adicionales sobre las clases especializadas.

Los conceptos de clases especializadas `package` y `function` en Modelica tienen mucho en común con el concepto de clase, pero también tienen propiedades adicionales, llamadas mejoramientos. Especialmente las clases tipo `function` tienen un número importante de mejoramientos, por ejemplo, pueden ser llamadas con una lista de argumentos, pueden ser instanciadas en tiempo de ejecución, etc. Una clase `operator` es similar a una clase `package` excepto que puede contener solamente declaraciones de clases tipo `function` y su finalidad es que el usuario pueda definir operadores sobrecargados (Ver sección 2.14.4).

### 2.3.4 Reutilización de Clases a través de Modificaciones

El concepto de clase es la clave para la reutilización del conocimiento de modelado en Modelica. Las capacidades de Modelica para expresar adaptaciones o modificaciones de las clases, a través de los denominados modificadores, facilitan la reutilización. Por ejemplo, suponga que se desea conectar en serie dos modelos de un determinado filtro que tienen constantes de tiempo diferentes.

En lugar de crear dos clases diferentes de filtro, es mejor definir una única clase de filtro y crear dos instancias de la clase modificadas adecuadamente, las cuales están conectadas. A continuación, se muestra una clase que modela un filtro paso baja, y posteriormente se muestra el modelo que representa la conexión de los dos filtros pasa bajo modificados:

```
model FiltroPasoBajo
  parameter Real T=1  "Constante de tiempo del filtro";
  Real u, y(start=1);
equation
  T*der(y) + y = u;
end FiltroPasoBajo;
```

Esta clase tipo `model` se puede usar para crear dos instancias del filtro, con constantes de tiempo diferentes, y "conectarlas" entre sí mediante la ecuación `F2.u = F1.y` tal como sigue:

```
model FiltrosEnSerie
  FiltroPasoBajo F1(T=2), F2(T=3);
equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltrosEnSerie;
```

En este modelo hemos empleado modificadores. Es decir, ecuaciones de atributo tales como `T=2` y `T=3`, para modificar la constante de tiempo del filtro paso-baja cuando se crean las instancias `F1` y `F2`. La variable de tiempo independiente se denota `time`. Si el modelo FiltrosEnSerie se emplea para declarar variables en un nivel jerárquico superior, por ejemplo, `F12`, las constantes de tiempo pueden todavía adaptarse usando una modificación jerárquica, tal como se hace con `F1` y `F2` a continuación:

```
model FiltrosModificadosEnSerie
  FiltrosEnSerie F12(F1(T=6), F2.T=11);
end FiltrosModificadosEnSerie;
```

### 2.3.5 Clases Predefinidas y Atributos

Las clases de tipos predefinidos de Modelica se corresponden con los tipos primitivos `Real`, `Integer`, `Boolean`, `String` y `enumeration`(...). Poseen la mayoría de las propiedades que tiene una clase, por ejemplo, pueden heredarse, modificarse, etc. Solamente el atributo `value` puede ser modificado en tiempo de ejecución y se accede a él a través del propio nombre de la variable, no mediante la notación punto. Es decir, para acceder al valor se usa `x`, en lugar de `x.value`. Se accede a los demás atributos empleando la notación punto.

Por ejemplo, una variable `Real` tiene un conjunto de atributos por defecto tales como unidad de medida, valor inicial, valores mínimo y máximo. Estos atributos por defecto se pueden cambiar cuando se declara una nueva clase, por ejemplo:

```
class Voltaje = Real(unit= "V", min=-220.0, max=220.0);
```

## 2.4 Herencia

Una de las grandes ventajas de la orientación a objetos es la facilidad para extender el comportamiento y las propiedades de una clase existente. La clase original, conocida como la *superclase* o *clase base*, se extiende para crear una versión más especializada de esa clase,

conocida como la *subclase* o *clase derivada*. En este proceso, el comportamiento y las propiedades de la clase original, en la forma de declaraciones de variables, ecuaciones y otros contenidos, son reutilizados o heredados por la subclase.

A modo de ejemplo, consideremos la extensión de una clase sencilla usando Modelica, por ejemplo, la clase `Punto` introducida previamente. En primer lugar, introducimos dos clases llamadas `DatosColor` y `Color`. En `DatosColor` se definen las variables que representan el color, y esta clase es heredada por `Color`. Además, se incluye en la clase `Color` una ecuación como una restricción. Finalmente, la nueva clase `PuntoColoreado` hereda de clases múltiples, es decir, usa herencia multiple para obtener las variables de la posición de la clase `Punto`, y las variables de color en conjunto con la ecuación de la clase `Color`.

```
record DatosColor
  Real rojo;
  Real azul;
  Real verde;
end DatosColor;

class Color
  extends DatosColor;
equation
  rojo + azul + verde = 1;
end Color;

class Punto
 public
  Real x;
  Real y, z;
end Punto;

class PuntoColoreado
  extends Punto;
  extends Color;
end PuntoColoreado;
```

En la Sección 3.7 se volverán a tratar la herencia y la reutilización.

## 2.5 Clases Genéricas

En muchas situaciones resulta ventajoso poder describir patrones genéricos de modelos o de programas. En lugar de escribir muchos fragmentos similares de código, con esencialmente la misma estructura, se puede ahorrar una cantidad considerable de programación y de

mantenimiento de software expresando la estructura general del problema y proporcionando los casos especiales como valores de *parámetros*.

Tales construcciones genéricas están disponibles en algunos lenguajes de programación. Por ejemplo, las plantillas en C++, los genéricos en Ada y los parámetros tipo en los lenguajes funcionales tales como Haskell o Standard ML. En Modelica, la construcción `class` es lo suficientemente general para manejar el modelado y la programación genérica, además de la funcionalidad usual de la clase.

Hay esencialmente dos casos de parametrización de las clases genéricas en Modelica. Los *parámetros de clase* pueden ser o bien *parámetros de instancia*, cuyos valores son instancias (componentes), o bien *parámetros de tipo*, cuyos valores son tipos. Observe que, en este contexto, el término parámetro de clase no hace referencia a los parámetros de los modelos que se declaran anteponiendo la palabra clave `parameter`, aun cuando tales "variables" son también una forma de parámetro de clase. En lugar de ello, este término designa a los *parámetros formales de clase*. Tales parámetros formales llevan como prefijo la palabra clave `replaceable`. El caso especial de las funciones locales reemplazables es aproximadamente equivalente a los métodos virtuales existentes en algunos lenguajes de programación orientados a objetos.

### 2.5.1 Parámetros de Clase que son Instancias

En primer lugar, presentamos el caso en el cual los parámetros de clase son variables. Es decir, declaraciones de instancias (las instancias a menudo se denominan componentes). En el ejemplo que se muestra a continuación, la clase `C` tiene tres parámetros de clase, que están *señalados* mediante la palabra clave `replaceable`. Estos parámetros de clase son componentes (variables) de la clase `C`, que se declaran con los siguientes tipos por defecto: `ClaseVerde`, `ClaseAmarilla` y `ClaseVerde` respectivamente. Hay también una declaración de un objeto rojo, que no es reemplazable y por lo tanto no es un parámetro de clase (véase la Figura 2-5).

Seguidamente se muestra la clase `C`, que posee tres parámetros de clase (`pobj1`, `pobj2` y `pobj3`) y una variable (`obj4`) que no es un parámetro de clase:

```
class C
  replaceable ClaseVerde    pobj1(p1=5);
  replaceable ClaseAmarilla pobj2;
  replaceable ClaseVerde    pobj3;
  ClaseRojo    obj4;
equation
  ...
end C;
```

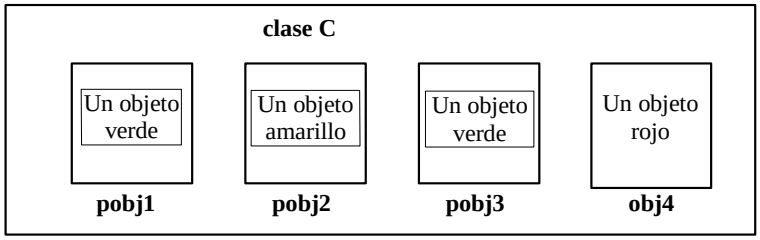



**Figura 2-5.** Tres parámetros de clase, `pobj1`, `pobj2`, y `pobj3`, que son instancias (variables) de la `clase C`. Son esencialmente cajas que contienen objetos de diferentes colores.

Ahora, se declara otra clase, llamada `C2`, donde se proporcionan dos declaraciones de `pobj1` y `pobj2` como argumentos actuales de la clase `C`. Estos son `rojo` y `verde` respectivamente, que sustituyen a los valores por defecto, que son `verde` y `amarillo`. Para que se produzca el cambio de tipo, es imprescindible que la palabra clave `redeclare` preceda al argumento actual del parámetro formal de la clase. El requisito de tener que usar una palabra clave para la redeclaración en Modelica se introdujo con el fin de evitar el cambio accidental del tipo de un objeto a través de un modificador estándar.

En general, el tipo de un componente de una clase no se puede cambiar si no se declara como `replaceable` y se proporciona una redeclaración. Una variable en una redeclaración puede reemplazar a la variable original si el tipo reemplazante es un subtipo del tipo original, o si satisface cierta restricción impuesta sobre el tipo reemplazante. Es posible imponer restricciones sobre el tipo reemplazante, si bien esta capacidad no se muestra aquí.

```
class C2 = C(redeclare ClaseRoja pobj1, redeclare ClaseVerde pobj2);
```

Por supuesto, obtener la clase `C2` mediante la redeclaración de `pobj1` y `pobj2` es equivalente a definir directamente `C2` sin reutilizar la clase `C`, como se muestra debajo.

```
class C2
  ClaseVerde pobj1(p1=5);
  ClaseVerde pobj2;
  ClaseVerde pobj3;
  ClaseVerde obj4;
equation
  ...
end C2;
```

### 2.5.2 Parámetros de Clase que son Tipos

Un parámetro de clase puede también ser un tipo. Esto es útil para cambiar la clase de varios objetos simultáneamente. Por ejemplo, al definir el parámetro de tipo `ClaseColoreada` en la clase `C`, que se muestra a continuación, es fácil cambiar el color de todos los objetos del tipo `ClaseColoreada`.

```
class C
  replaceable class ClaseColoreada = ClaseVerde;
  ClaseColoreada            obj1(p1=5);
  replaceable ClaseAmarilla obj2;
  ClaseColoreada            obj3;
  ClaseRoja                 obj4;
equation
  ...
end C;
```

La Figura 2-6 representa cómo el valor del tipo del parámetro de la clase `ClaseColoreada` se propaga a los miembros de las declaraciones de los objetos `obj1` y `obj3`.

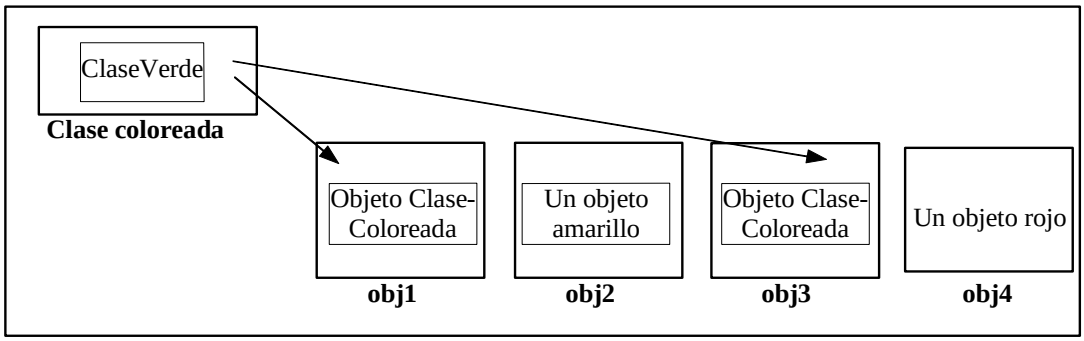



**Figura 2-6.** El parámetro de la clase `ClaseColoreada` es un parámetro de tipo que se propaga a las declaraciones de las instancuas de dos miembros de los objetos obj1 y obj3.

Creamos una clase `C2`, dando al parámetro tipo `ClaseColoreada` de la clase `C` el valor `ClaseAzul`.

```
class C2 = C(redeclare class ClaseColoreada = ClaseAzul);
```

Esto es equivalente a la siguiente definición de `C2`:

```
class C2
  ClaseAzul      obj1(p1=5);
  ClaseAmarilla obj2;
  ClaseAzul     obj3;
  ClaseRoja     obj4;
equation
  ...
end C2;
```

## 2.6 Ecuaciones

Como se ha indicado anteriormente, Modelica es fundamentalmente un lenguaje basado en ecuaciones, en contraste con los lenguajes de programación ordinarios, donde proliferan las sentencias de asignación. Las ecuaciones son más flexibles que las asignaciones, ya que no imponen una cierta dirección del flujo de datos u orden de ejecución. Esta es la clave de las capacidades para el modelado físico y la potencial reutilización que poseen las clases de Modelica.

Pensar en ecuaciones resulta un poco inusual para la mayoría de los programadores. En Modelica se cumple lo siguiente:

- Las sentencias de asignación en los lenguajes convencionales se suelen representar normalmente como ecuaciones en Modelica.
- Las asignaciones de atributos se representan como ecuaciones.
- Las conexiones entre objetos generan ecuaciones.

Las ecuaciones son más potentes que las sentencias de asignación. Por ejemplo, consideremos la ecuación de una resistencia: la resistencia `R` multiplicada por la corriente `i` es igual a la caída de tensión `v`:

```
R*i = v;
```

Esta ecuación puede emplearse de tres formas, que corresponden a tres posibles sentencias de asignación: para calcular la corriente a partir de la tensión y la resistencia, para calcular la tensión a partir de la resistencia y la corriente, o para calcular la resistencia a partir de la tensión y la corriente. Esto se expresa mediante las tres sentencias de asignación siguientes:

```
i := v/R;
v := R*i;
R := v/i;
```

Las ecuaciones en Modelica pueden clasificarse informalmente en cuatro grupos diferentes, que dependen del contexto sintáctico en el que ocurren:

- *Ecuaciones normales*, que se localizan en las secciones de ecuaciones, incluyendo la ecuación connect, que es una forma especial de ecuación.
- *Ecuaciones de declaración*, que son parte de las declaraciones de variables o constantes.
- *Ecuaciones de modificación*, que se utilizan normalmente para modificar atributos.
- *Ecuaciones iniciales*, que son especificadas en las secciones initial equation o son asignadas al atributo start. Estas ecuaciones se usan para resolver el problema de inicialización en el instante inicial de ejecución de la simulación.

Como ya hemos visto en algunos ejemplos, las ecuaciones normales aparecen en las secciones de ecuaciones que comienzan por la palabra clave `equation` y finalizan mediante alguna otra palabra clave permitida:

```
equation
  ...
  <ecuaciones>
  ...
<alguna otra palabra clave permitida>
```

La ecuación anterior de la resistencia es un ejemplo de ecuación normal, que se puede colocar en una sección equation. Las ecuaciones de declaración se suelen dar como parte de declaraciones de constantes o parámetros a los que se asignan valores:

```
constant Integer uno = 1;
parameter Real masa = 22.5;
```

Una ecuación siempre se cumple, lo cual quiere decir que la masa en el ejemplo anterior nunca cambia de valor durante la simulación. Es también posible especificar una ecuación de declaración para una variable normal, por ejemplo:

```
Real velocidad = 72.4;
```

Sin embargo, hacer esto no tiene mucho sentido, puesto que restringirá a la variable a tener el mismo valor a lo largo de todo el cálculo, comportándose así como una constante. Por lo tanto, una ecuación de declaración es algo diferente de una inicialización de una variable en otros lenguajes.

Por lo que respecta a la asignación de atributos, estos se especifican típicamente utilizando ecuaciones de modificación. Por ejemplo, si necesitamos especificar un valor inicial para una

variable, es decir, asignarle un valor en el instante inicial de la simulación, entonces asignamos al atributo start de la variable una ecuación de atributo. Por ejemplo:

```
Real velocidad(start=72.4);
```

### 2.6.1 Estructuras de Ecuaciones Repetitivas

Antes de leer esta sección sería conveniente que mirara la Sección 2.13, que trata de los arreglos, y la Sección 2.14.2, en la que se explican los ciclos for de sentencias y algorítmicos.

En ocasiones surge la necesidad de expresar en una forma conveniente conjuntos de ecuaciones que tienen una estructura regular, es decir, repetitiva. A menudo, las ecuaciones con estructura repetitiva pueden expresarse como ecuaciones en forma de arreglos, incluyendo referencias a los elementos de los arreglos utilizando la notación de corchetes. Sin embargo, para el caso más general de ecuaciones con estructura repetitiva, Modelica proporciona una construcción de ciclo. Observe que esto no es un ciclo en el sentido algorítmico de la palabra—simplemente es una notación abreviada para expresar un conjunto de ecuaciones.

Por ejemplo, consideremos una ecuación para una expresión polinomial:

```
y = a[1]+a[2]*x + a[3]*x^2 + ... + a[n+1]*x^n
```

La ecuación polinomial se puede expresar como un conjunto de ecuaciones con estructura regular en Modelica, con `y` igual al producto escalar de los vectores `a` y `xpowers`, ambos de longitud `n+1`:

```
xpowers[1] = 1;
xpowers[2] = xpowers[1]*x;
xpowers[3] = xpowers[2]*x;
...
xpowers[n+1] = xpowers[n]*x;
y = a * xpowers;
```

El conjunto regular de ecuaciones que incluye `xpowers` se puede expresar de forma más conveniente utilizando un ciclo `for`:

```
for i in 1:n loop
  xpowers[i+1] = xpowers[i]*x;
end for;
```

En este caso particular, una ecuación vectorial proporciona incluso una notación más compacta:

```
xpowers[2:n+1] = xpowers[1:n]*x;
```

Aquí los vectores `x` y `xpowers` tienen longitud `n+1`. La notación dos puntos `2:n+1` significa extraer un vector de longitud `n`, comenzando desde el elemento 2 hasta incluir al elemento `n+1`.

### 2.6.2 Ecuaciones Diferenciales Parciales

Las ecuaciones diferenciales parciales (de forma abreviada EDP) contienen derivadas con respecto a otras variables distintas del tiempo. Por ejemplo, coordenadas espaciales cartesianas tales como *x* e *y*. Los modelos de fenómenos tales como el flujo de calor o el flujo de fluidos contienen típicamente EDP. En el momento actual, el lenguaje Modelica no incorpora oficialmente funcionalidades para tratar EDP, si bien se está trabajando para incluirlas.

## 2.7 Modelado Físico No Causal

El modelado No Causal es un estilo de modelado declarativo. Esto quiere decir que es un modelado basado en ecuaciones, no en sentencias de asignación. Las ecuaciones no especifican qué variables son entradas y cuáles son salidas, mientras que en las sentencias de asignación las variables en el lado izquierdo son siempre salidas (resultados) y las variables en el lado derecho son siempre entradas. Así, la causalidad de los modelos basados en ecuaciones no está especificada y se fija solamente cuando se resuelve el correspondiente sistema de ecuaciones. Esto se llama *modelado no causal*. El término *modelado físico* refleja el hecho de que el *modelado no causal* está muy bien adaptado para representar la *estructura física* de los sistemas modelados.

La ventaja principal del modelado no causal es que la solución de la dirección de las ecuaciones se adaptará al contexto del flujo de datos en el cual se calcula la solución. El contexto del flujo de datos se define expresando cuáles variables se necesitan como *salidas*, y cuáles son *entradas* externas al sistema simulado.

La no causalidad de las clases de la librería de Modelica las hace más reutilizables que las clases tradicionales, las cuales contienen sentencias de asignación donde la causalidad `entrada-salida` está fijada.

### 2.7.1 Modelado Físico vs. Modelado Orientado a Bloques

Para ilustrar la idea del modelado físico no causal mostramos un ejemplo de un circuito eléctrico sencillo (vea la Figura 2-7). El diagrama de conexiones[8] del circuito eléctrico muestra cómo se conectan los componentes. Se puede dibujar colocando los componentes de manera que se correspondan aproximadamente a la estructura física del circuito eléctrico en una tarjeta de circuito impreso. Las conexiones físicas en el circuito real corresponden a las conexiones lógicas en el diagrama. Por lo tanto el *término modelado físico* resulta bastante apropiado.

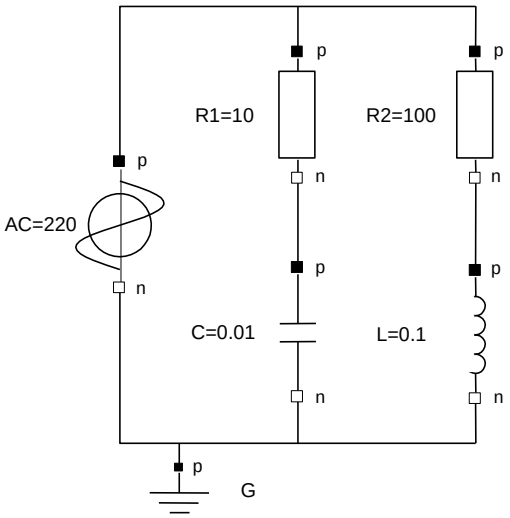



**Figura 2-7.** Diagrama de conexiones del modelo no causal del circuito simple.

A continuación, se muestra el modelo `CircuitoSimple` escrito en Modelica. Corresponde directamente con el circuito representado en el diagrama de conexiones de la Figura 2-7. Cada objeto gráfico en el diagrama corresponde a una instancia declarada en el modelo del circuito simple. El modelo es no causal, puesto que no se especifica ningún flujo de señal, es deci, no se ha especificado aún un flujo causa-efecto. Las conexiones entre objetos se especifican utilizando ecuaciones `connect`, que es una forma sintáctica especial de ecuación que se examinará

[8] Un diagrama de conexiones enfatiza las conexiones entre componentes de un modelo, mientras que un diagrama de composición especifica de qué componentes se compone un modelo, sus subcomponentes, etc. Un diagrama de clases normalmente representa herencia y relaciones de composición.

posteriormente. Las clases `Resistor`, `Capacitor`, `Inductor`, `VSourceAC`, y `Ground` se explicarán con detalle en las Secciones 2.11 y 2.12.

```
model CircuitoSimple
  Resistor  R1(R=10);
  Capacitor  C(C=0.01);
  Resistor  R2(R=100);
  Inductor     L(L=0.1);
  VSourceAC    AC;
  Ground       G;
equation
  connect(AC.p, R1.p);    // Malla del condensador
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);    // Malla del inductor
  connect(R2.n, L.p);
  connect(L.n,  C.n);
  connect(AC.n, G.p);     // Conexión a tierra
end CircuitoSimple;
```

A modo de comparación, en la Figure 2-8 se muestra el mismo modelo del circuito, pero modelado utilizando el modelado causal orientado a bloques. En este caso, se pierde la topología física —la estructura del diagrama no tiene una correspondencia simple con la estructura de la tarjeta del circuito físico. Este modelo es causal, puesto que el flujo de señal ha sido deducido y se muestra claramente en el diagrama. Incluso para este ejemplo simple, el análisis para convertir el modelo físico intuitivo a un modelo causal orientado a bloques no es trivial. Otra desventaja es que las representaciones de las resistencias son dependientes del contexto. Por ejemplo, las resistencias `R1` y `R2` tienen definiciones diferentes, lo cual hace difícil la reutilización de los componentes de la librería de modelos. Más aun, tales modelos de sistemas son normalmente difíciles de mantener puesto que incluso pequeños cambios en la estructura física pueden dar como resultado grandes cambios en el correspondiente modelo del sistema orientado a bloques.

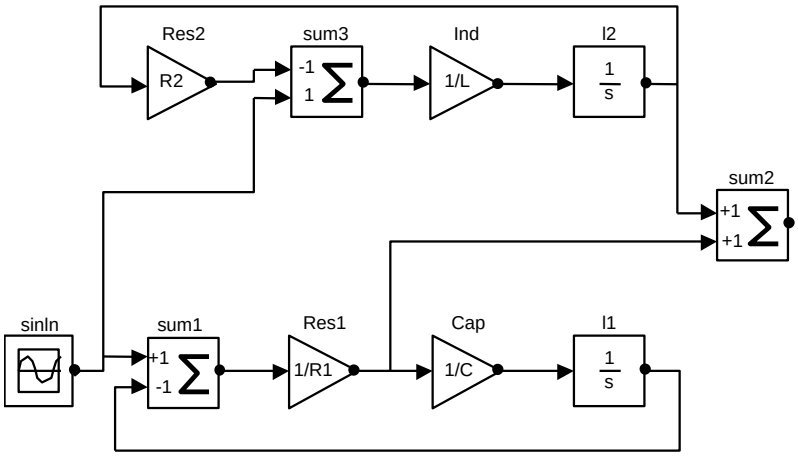



**Figure 2-8** Modelo del circuito sencillo utilizando modelado causal orientado a bloques, en el cual el flujo de señal está explícito.

## 2.8 El Modelo de Componentes de Software de Modelica

Durante mucho tiempo, los desarrolladores de software han mirado con cierta envidia a los constructores de sistemas de hardware, al considerar la aparente facilidad con la que empleaban componentes de hardware reutilizables para construir sistemas complicados. Con el software muy a menudo parece que se tiene la tendencia a desarrollar desde el principio en lugar de reutilizar componentes. Los primeros intentos en componentes de software incluyen las librerías de procedimientos, que desafortunadamente tienen una aplicabilidad muy limitada y una flexibilidad baja. La aparición de la programación orientada a objetos ha estimulado el desarrollo de formalismos de componentes de software tales como CORBA, el modelo de objeto de componentes de Microsoft COM/DCOM, y JavaBeans. Estos modelos de componentes tienen un éxito considerable en ciertas áreas de aplicación, pero queda todavía un largo camino por recorrer para alcanzar el nivel de reutilización y estandarización de los componentes de la industria del hardware.

El lector podría preguntarse qué tiene que ver todo esto con Modelica. De hecho, Modelica ofrece un modelo de componentes de software bastante potente, que es bastante similar a los sistemas de componentes de hardware en flexibilidad y potencial para su reutilización. La clave de este aumento en la flexibilidad se debe al hecho de que las clases de Modelica se basan en

ecuaciones. ¿Qué es un modelo de componentes de software? Debería incluir los tres elementos siguientes:

1. Componentes
2. Un mecanismo de conexión
3. Una estructura para los componentes

Los componentes se conectan mediante el mecanismo de conexión, que puede ser visualizado usando los diagramas de conexiones. La estructura de componentes hace efectivo los componentes y las conexiones, y asegura que la comunicación funciona y que las restricciones se mantienen a lo largo de las conexiones. Para sistemas compuestos de componentes no causales, el compilador deduce automáticamente la dirección del flujo de datos (es decir, la causalidad) en tiempo de composición.

### 2.8.1 Componentes

Los componentes son simplemente instancias de clases en Modelica. Esas clases deberían tener interfaces bien definidas, que permitan la comunicación y el acoplamiento entre el componente y el mundo exterior. En ocasiones, estas interfaces se denominan puertos. En Modelica se llaman conectores.

Un componente se modela independientemente del entorno donde se utiliza, lo que es esencial para su reusabilidad. Esto significa que en la definición del componente, incluidas sus ecuaciones, se pueden emplear solamente las variables locales y las variables de los conectores. No se debería permitir ningún medio de comunicación entre un componente y el resto del sistema que no se produjera mediante los conectores. Sin embargo, en Modelica es también posible acceder a datos de los componentes empleando la notación punto. Un componente puede estar internamente compuesto por otros componentes conectados, es decir, modelado jerárquico.

### 2.8.2 Diagramas de Conexiones

Los sistemas complejos consisten en gran número de componentes conectados. De estos componentes, muchos podrán descomponerse jerárquicamente en otros componentes a través de varios niveles. Para captar toda esta complejidad, es importante disponer de una representación gráfica de los componentes y de sus conexiones. Los diagramas de conexiones proporcionan una representación gráfica de este tipo, como la representación esquemática que se muestra a modo de

ejemplo en la Figura 2-9. Anteriormente, en la Figura 2-7, se representó el diagrama de conexiones de un circuito sencillo.

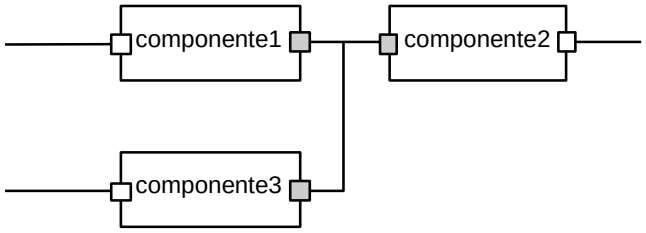



**Figura 2-9.** Representación esquemática de un diagrama de conexión para componentes.

Cada rectángulo en el diagrama de ejemplo representa un componente físico. Por ejemplo, una resistencia, un condensador, un transistor, un engranaje mecánico, una válvula, etc. Las conexiones representadas en el diagrama mediante líneas corresponden a conexiones físicas reales. Por ejemplo, las conexiones se pueden realizar usando cables eléctricos, conexiones mecánicas, tuberías para fluidos, mediante el intercambio de calor entre los componentes, etc. Los conectores (puntos de interfaz) se representan en el diagrama como pequeños cuadrados sobre el rectángulo. Las variables en tales puntos de interfaz definen la interacción entre el componente representado por el rectángulo y otros componentes.

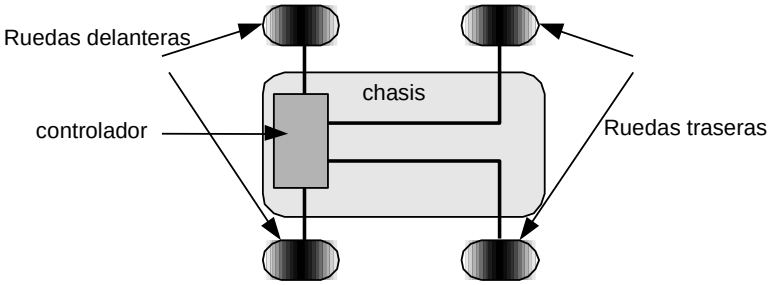



**Figura 2-10.** Un diagrama de conexión para un modelo de carro sencillo.

En la Figura 2-10 se muestra un ejemplo de diagrama de conexión. Se trata de un carro sencillo para una aplicación en el dominio mecánico.

El modelo del carro sencillo mostrado a continuación incluye variables que son subcomponentes, tales como ruedas, chasis y unidad de control. Una cadena de "comentario" después del nombre de la clase la describe brevemente. Las ruedas se conectan al carro y al controlador. Las ecuaciones de conexión están presentes, pero no se muestran en este ejemplo parcial.

```
class Carro  "una clase coche que combina componentes del carro"
  Rueda             w1,w2,w3,w4   "Ruedas de una a cuatro";
  Chasis            chasis        "Chasis";
  ControladorCarro  controlador   "Controlador del carro";
  ...
end Carro;
```

### 2.8.3 Conectores y Clases connector

Los conectores de Modelica son instancias de clases connector, que definen las variables que son parte de la interfaz de comunicación que se especifica mediante un conector. Así, los conectores especifican interfaces externas para la interacción.

Por ejemplo, `Pin` es una clase de tipo connector. Se puede utilizar para representar las interfaces externas de los componentes eléctricos (vea la Figura 2-11) que tienen pines. Los tipos `Voltage` y `Current` utilizados dentro de `Pin` son lo mismo que `Real`, pero con unidades asociadas diferentes. Desde el punto de vista del lenguaje Modelica, los tipos `Voltage` y `Current` son similares a `Real` y se consideran que tienen tipos equivalentes. La comprobación de la compatibilidad de unidades dentro de las ecuaciones es opcional.

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
```

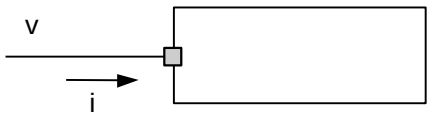



**Figura 2-11.** Un componente con un conector eléctrico `Pin`.

La clase del conector `Pin` que se muestra a continuación contiene dos variables. El prefijo `flow` en la segunda variable indica que esta variable representa una magnitud de tipo flow. Esto tiene una significación especial para las conexiones, tal como se explica en la próxima sección.

```
connector Pin
  Voltage       v;
  flow Current  i;
end Pin;
```

### 2.8.4 Conexiones

Las conexiones entre componentes pueden establecerse entre conectores de tipo equivalente. Modelica soporta conexiones no causales basadas en ecuaciones, lo que significa que las conexiones son traducidas a ecuaciones. Para conexiones no causales, no se necesita conocer la dirección del flujo de datos en la conexión. Adicionalmente, las conexiones causales se pueden establecer conectando un conector con un atributo `output` a un conector declarado como `input`.

Se pueden establecer dos tipos de acoplamiento mediante conexiones, dependiendo de si las variables que se conectan son o no de tipo flow (por defecto no lo son). Son de tipo flow aquellas variables que se declaran utilizando el prefijo `flow`:

1. Acoplamiento de igualdad, para variables de tipo no flow, de acuerdo con la primera ley de Kirchhoff.
2. Acoplamiento de suma igual a cero, para variables de tipo flow, de acuerdo con la ley de corrientes de Kirchhoff.

Por ejemplo, la palabra clave `flow` para la variable `i`, que es del tipo `Current` en la clase de conector `Pin`, indica que la suma de todas las corrientes en los terminales conectados es igual a cero, de acuerdo con la ley de corrientes de Kirchhoff.

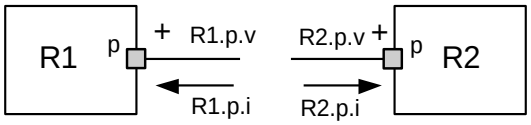



**Figura 2-12.** Conectando dos componentes que tienen pines eléctricos.

Las ecuaciones de conexión se utilizan para conectar instancias de la clase connector. Una ecuación de conexión `connect(R1.p,R2.p)`, donde `R1.p` y `R2.p` son objetos de la clase de connector `Pin`, conecta los dos pines (vea la Figura 2-12) formando un nodo del circuito. Esto produce dos ecuaciones, que son:

```
R1.p.v = R2.p.v
R1.p.i + R2.p.i = 0
```

La primera ecuación dice que los voltajes de los extremos de los cables conectados son idénticos. La segunda ecuación corresponde a la segunda ley de Kirchhoff, lo que quiere decir que la suma de las corrientes que entran al nodo es igual a cero (suponiendo valor positivo cuando fluyen hacia dentro del componente). Las ecuaciones de suma a cero se generan cuando se usa el prefijo

`flow`. Leyes similares se aplican a los flujos en redes de tuberías, y a fuerzas y pares en sistemas mecánicos.

### 2.8.5 Conexiones Implícitas con las palabras clave Inner/Outer

Hasta ahora nos hemos enfocado en conecciones explícitas entre conectores donde cada conexión es explicitamente representada por una ecuación tipo `connect` y una línea correspondiente en un diagrama de conexiones. Sin embargo, cuando se está modelando cierta clase de modelos grandes con muchos componentes interactuantes, esta forma de hacerlo se vuelve algo torpe debido al gran número de conexiones potenciales – puede llegar a requerirse una conexión entre cada par de componentes. Esta situación es particularmente verdadera en el caso de un modelo de un sistema que involucra *campos de fuerza*. Los cuales requieren un máximo de *n* `x` *n* conexiones entre los n componentes influenciados por el campo de fuerza o 1 `x` *n* conexiones entre un objeto central y *n* componentes si se desprecia la interacción intercomponententes.

Para el caso de 1 `x` *n* conexiones, en vez de usar un número grande de conexiones explícitas, Modelica provee un mecanismo conveniente para *conexiones explícitas* entre un objeto y *n* de sus componentes. Para ello Modelica implemente los prefijos de declaración `inner` y `outer`.
Una clase mas común de interaxiónes implícitas es donde un *atributo compartido* de un objeto de ambiente único se *accesa* por un número determinado de componentes con ese ambiente. Por ejemplo, se podría tener un ambiente que inclua is componentes de una casa, cada uno puede acceder una temperature de ambiente compartida. Otro ejemplo puede ser un ambiente de una placa de un circuito con componentes electrónicos que accesan a una temperatura de placa.
A continuación se muestra un modelo de ejemplo con un componente de ambiente de Modelica basado en estos conceptos. En este caso una variable compartida de temperatura de ambiente `T0` se declara como una *declaración de definición* de `T0` marcada por un prefijo `outer` en los componentes comp1 y comp2

```
model Ambiente
  import Modelica.Math.sin;
  inner Real T0;
     //Definición de una temperatura de ambiente actual T0
  Componente comp1,comp2;
     // Busca por una coincidencia comp1.T0 = comp2.T0 = T0
  parameter Real k=1;
equation
  T0 = sin(k*time);
```

```
end Ambiente;

model Componente
  outer Real T0
    //Referencia a la temperature T0 definida en los ambientes
  Real T;
equation
  T = T0
end Componente;
```

### 2.8.6 Conectores Expandibles para Buses de Información

En ingeniería es común tener lo que se ha denominado buses de información. Los buses de información tienen el propósito de transporter información entre varios componentes de un sistema, por ejemplo, sensors, actuadores y unidades de control. Aunque algunos buses se encuentran estandarizados (e.g. por IEEE), lo mas común es que estos son genéricos de tal manera que permitan muchas clases de componentes diferentes.
Esta es la idea clave detrás de la construcción conector expandible que corresponde a las palabras clave `expandable connector` en Modelica. Un conector expandible actúa como un bus de información ya que pretende conectar a muchas clases de componentes. Para hacerlo posible, el conector expandible expande automáticamente su tipo para acomodar todos los componentes conectados a él con sus diferentes interfaces. Si un elemento con un cierto nombre y un cierto tipo no se encuentra presenta sencillamente es adicionado.
Todos los campos en un conectos expandible se ven como instancias tipo connector aún si no se declaran como tales, es decir, por ejemplo es posible conectar a una variable tipo `Real`.
Aún mas, cuando dos conectores expandibles se conectan, cada uno se expande con las variables que están declaradas solamente en el otro conector expandible. Esto se repite hasta que todas las instancias de los conectores expandibles han coincidido con todas las variables. Es decir, que cada una de las instancias del conector se expanden hasta llegar a ser la unión de todas las variables del conector. Si una variable aparece como una entrada en un conector expandible, esta no debería aparecer como una entrada en al menos una instancia de otro conector expandible en el conjunto conectado. A continuación se presenta un pequeño ejemplo:

```
expandable connector BusMotor
end BusMotor
```

```
block Sensor
  RealOutput velocidad;
end Sensor

block Actuador
  RealInput velocidad;
end Actuador

model Motor
  BusMotor bus;
  Sensor sensor;
  Actuador actuador;
equation
  connect(bus.velocidad, sensor.velocidad)
    // Con esto se establece la no entrada
  connect(bus.velocidad, actuador.velocidad);
end Motor;
```

Hay muchos mas elementos a considerar cuando se usan conectores expansibles; por ejemplo, ver Modelica(2010) y Fritzson(2011).

### 2.8.7 Conectores tipo Stream

En termodinámica ecisten aplicaciones con fluídos donde los flujos de material pueden ser bidireccionales con ciertas cantidades asociadas. En estos casos sucede que los dos tipos de variables básicas de flujo en un conector, variables de potencial/no flujo y variables de flujo, no son suficientes para describir los modelos que resultan en un acercamiento numéricamente sólido. Típicamente, tales aplicaciones tienen flujos bidireccionales de materia con transporte convective de cantidades específicas, tales como entalpía específica y composición química.
Si se desea usar conectores convencionales con variables de flujo y no flujo, los modelos correspondientes podrían incluir sistemas no lineales de ecuaciones con variables desconocidas booleanas par alas direcciones de flujo y singularidades alrededor del flujo cero. En general, tales sistemas de ecuaciones no puden ser resueltos de manera confinable. Las formulaciones del modelo se pueden simplificar cuando se formulan dos ecuaciones de balance diferentes par las dos posibles direcciones del flujo. Sin embargo, esto no es possible usando solamente variables de flujo y no flujo.
Este problema fundamental se ha direccionado en Modelica introduciendo un tercer tipo de variable conector llamado variable *stream*. Declarada con el prefijo `stream`. Una variable tipo

stream describe una cantidad que es transportada por una variable de flujo, esto es, un fenómeno de trasnsporte ouramente convectivo.
Si al menos una variable en un conector tiene el prefijo `stream`, entonces se le llama al conector un *stream connector* y a la variable correspondiente se le llama una *stream variable*. Por ejemplo:

```
connector PuertoFluido
…
   flow Real m.flow
     "Flujo de material; m.flow>0 si enra flujo al componente";
   stream Real h.outflow
     "Variable específica en componente si m.flow<0"
end PuertoFluido
model SistemaFluido
…
  ComponenteFluido m1, m2, …, mN;
  PuertFluido      c1, c2, …, cM;
equation
   connect(m1.c, m2.c)
   …
   connect(m1.c, cM)
   …
end SistemaFluido
```

Para mas deatlles y explicaciones adicionales ver Modelica(2010) y Fritzson(2011).

## 2.9 Clases Parciales

Una propiedad que tienen en común bastantes componentes eléctricos es que poseen dos pines. Por ello, resulta útil definir una clase de modelo, por ejemplo llamada `DosPines`, que represente esta propiedad común. De esto se trata una *clase parcial*, puesto que no contiene las suficientes ecuaciones como para especificar completamente el comportamiento físico del componente. Para ello, se define anteponiendo la palabra reservada `partial`. En otros lenguajes orientados a objetos, las clases parciales se denominan *clases abstractas*.

```
partial class DosPines[9] "Superclase de componentes con dos pines
eléctricos"
```

[9] Esta clase `DosPines` forma parte de la librería estándar de Modelica, si bien recibe el nombre `Modelica.Electrical.Analog.Interfaces.OnePort` , puesto que éste es el nombre comúnmente usado por los expertos en modelado de sistemas eléctricos. En este texto se emplea el nombre `DosPines`, que

```
  Pin      p, n;
  Voltage  v;
  Current  i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end DosPines;
```

La clase `DosPines` contiene dos pines, `p` y `n`, una magnitud `v` que representa la caída de tensión en el componente, y una magnitud `i` que representa la corriente que entra por el pin `p`, que circula a través del componente, y sale por el pin `n` (véase la Figura 2-13). Aunque desde el punto de vista de la física del sistema real no existe ninguna diferencia entre ambos pines, resulta útil asignarles nombres diferentes, por ejemplo `p` y `n`, y representarlos gráficamente de manera diferente, por ejemplo mediante un cuadrado lleno y vacío respectivamente, de modo que los signos de `v` e `i` queden bien definidos.

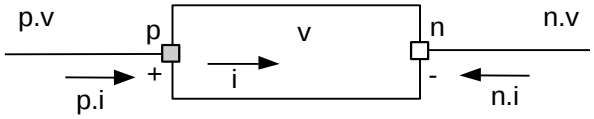



**Figura 2-13.** Clase genérica `DosPines`, que describe la estructura general de los componentes eléctricos simples de dos pines.

Las ecuaciones de la clase `DosPines` definen relaciones genéricas entre las magnitudes de los componentes eléctricos simples. Para obtener un modelo útil, es necesario añadir además una ecuación constitutiva que especifique las características físicas del componente.

### 2.9.1 Reutilización de las Clases Parciales

Dada la clase parcial `DosPines`, resulta sencillo definir la clase `Resistencia` añadiendo la ecuación constitutiva de la resistencia eléctrica, que es la siguiente:

```
R*i = v;
```

resulta más intuitivo debido a que la clase se usa para componentes con dos puertos físicos y no uno. El nombre `OnePort` resulta más fácilmente comprensible si se considera que representa puertos compuestos de dos subpuertos.

Esta ecuación describe las características físicas específicas, es decir, la relación entre la caída de tensión y la corriente en la resistencia (véase la Figura 2-14).

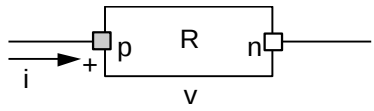



**Figura 2-14.** Un componente resistencia eléctrica.

```
class Resistencia "Resistencia eléctrica ideal"
  extends DosPines;
  parameter Real R(unit="Ohm") "Resistencia";
equation
  R*i = v;
end Resistencia;
```

De manera similar, una clase que modele un condensador eléctrico también puede reutilizar `DosPines`. En este caso, debe añadirse la ecuación constitutiva del condensador (véase la Figura 2-15).

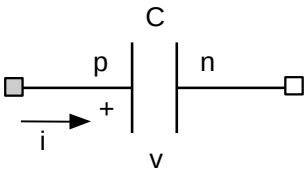



**Figura 2-15.** Un componente condensador eléctrico.

```
class Condensador "Condensador eléctrico ideal"
  extends DosPines;
  parameter Real C(Unit="F") "Capacitancia";
equation
  C*der(v) = i;
end Condensador;
```

Durante la simulación del modelo, los valores de estas variables `i` y `v`, definidas en el componente anterior, van cambiando en función del tiempo. El solucionador de ecuaciones diferenciales calcula los valores de $v(t)$ e $i(t)$ (donde *t* es el tiempo), de modo que se satisfaga $C \cdot \dot{v}(t) = i(t)$ para todos los valores de *t*. Es decir, de modo que se satisfaga la ecuación constitutiva del condensador.

## 2.10 Diseño y Uso de una Librería de Componentes

De manera similar a como se han creado los modelos de los componentes resistencia y condensador, pueden crearse los modelos de otros componentes eléctricos y ser agrupados formando una librería sencilla de componentes eléctricos, que puede ser usada para componer modelos tales como el modelo `CircuitoSimple`. Las librerías de componentes reutilizables son realmente la clave para poder modelar de manera eficaz los sistemas complejos.

## 2.11 Ejemplo: Librería de Componentes Eléctricos

A continuación, se muestra un ejemplo de diseño de una pequeña librería de componentes eléctricos. Se han considerado los componentes necesarios para modelar un circuito sencillo: `CircuitoSimple`. Asimismo, se muestran las ecuaciones que pueden extraerse del modelo de cada uno de los componentes.

### 2.11.1 Resistencia

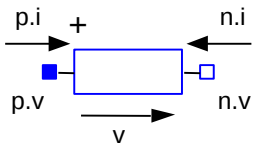



**Figura 2-16.** Componente resistencia.

Pueden extraerse cuatro ecuaciones del modelo de la resistencia eléctrica, que está representado en la Figura 2-14 y en la Figura 2-16. Las tres primeras provienen de la clase `DosPines`, que ha sido heredada, mientras que la última es la ecuación constitutiva de la resistencia.

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
v = R*i
```

### 2.11.2 Condensador

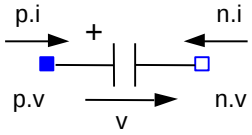



**Figura 2-17.** Componente condensador.

El modelo del condensador, representado en la Figura 2-15 y en la Figura 2-17, está compuesto por las cuatro ecuaciones mostradas a continuación. La última es la ecuación constitutiva del condensador.

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
i = C * der(v)
```

### 2.11.3 Inductor

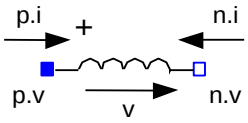



**Figura 2-18.** Componente inductor.

En la Figura 2-18 se representa esquemáticamente un inductor eléctrico ideal. El código del modelo se muestra a continuación.

```
class Inductor "Inductor eléctrica ideal"
  extends DosPines;
  parameter Real L(unit="H") "Inductancia";
equation
  v = L*der(i);
end Inductor;
```

A continuación, se muestran las cuatro ecuaciones que componen el modelo. Al igual que en los anteriores componentes, las tres primeras ecuaciones provienen de la clase `DosPines` y la última es la ecuación constitutiva del inductor.

```
0 = p.i + n.i
```

```
v = p.v - n.v
i = p.i
v = L * der(i)
```

### 2.11.4 Fuente de tensión

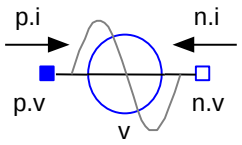



**Figura 2-19.** Componente fuente de tensión `VfuenteAC`, donde `v(t) = VA*sin(2*PI*f*time)`.

En la Figura 2-19 se representa la fuente de tensión que forma parte del circuito eléctrico que estamos desarrollando como ejemplo. A continuación, se muestra el código del modelo de esta fuente. Este modelo, al igual que otros modelos en Modelica, representa un comportamiento (el valor de la tensión en la fuente) que varía con el tiempo. Obsérvese que el tiempo se representa mediante la variable predefinida `time`. Por simplicidad, en el modelo se asigna valor explícitamente a la constante `PI`, cuando lo más habitual es importarla del package de constantes de la librería estándar de Modelica.

```
class VfuenteAC "Fuente de tensión sinusoidal"
  extends DosPines;
  parameter Voltage  VA           = 220 "Amplitud";
  parameter Real     f(unit="Hz") = 50  "Frecuencia";
  constant  Real     PI           = 3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VfuenteAC;
```

De este modelo, que también está basado en el modelo `DosPines`, pueden extraerse cuatro ecuaciones. Las tres primeras son heredadas de `DosPines`:

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
v = VA*sin(2*PI*f*time)
```

### 2.11.5 Tierra

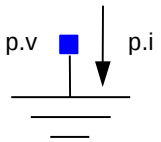



**Figura 2-20.** Componente tierra.

Finalmente, se define una clase que modela los nodos de tierra del circuito. La tensión en estos puntos (de valor cero) puede ser instanciado como un valo de referencia para el cálculo de la tensión en los restantes nodos del circuito. Esta clase posee sólo un pin (véase la Figura 2-20).

```
class Tierra "Tierra"
  Pin p;
equation
  p.v = 0;
end Tierra;
```

Puede extraerse una única ecuación de la clase `Tierra`:

```
p.v = 0
```

## 2.12 El Modelo de un Circuito Sencillo

Los componentes eléctricos modelados anteriormente pueden emplearse para componer el modelo del circuito eléctrico mostrado en la Figura 2-21.

En la declaración de las dos instancias de la clase `Resistencia`, `R1` y `R2`, se modifica el valor del parámetro resistencia eléctrica. Igualmente, la instancia `C` de la clase `Condensador` y la instancia `L` de la clase `Inductor` son declaradas modificando los valores de la capacidad y la inductancia respectivamente. Las instancias de la fuente de tensión `AC` y de la tierra `G` no poseen modificadores. Las conexiones entre los componentes del circuito han sido descritas empleando sentencias connect.

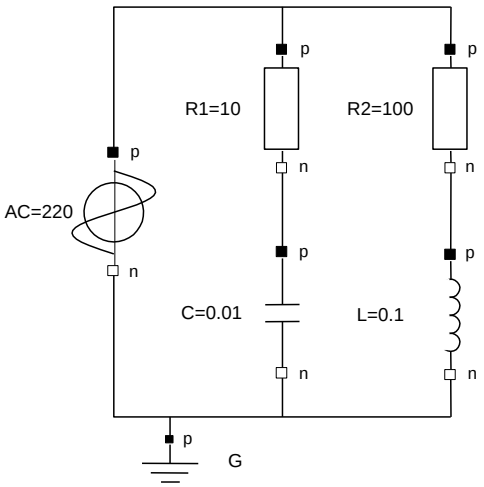



**Figura 2-21.** Modelo de un circuito eléctrico sencillo.

```
class CircuitoSimple
  Resistencia  R1(R=10);
  Condensador  C(C=0.01);
  Resistencia  R2(R=100);
  Inductor     L(L=0.1);
  VfuenteAC    AC;
  Tierra       G;
equation
  connect(AC.p, R1.p);     // Malla del condensador
  connect(R1.n, C.p);      // Cable 2
  connect(C.n, AC.n);      // Cable 3
  connect(R1.p, R2.p);     // Malla del inductor
  connect(R2.n, L.p);      // Cable 5
  connect(L.n,  C.n);      // Cable 6
  connect(AC.n, G.p);      // Conexión a tierra
end CircuitoSimple;
```

## 2.13 Arrays

Un array es una colección de variables, donde todas ellas son de un mismo tipo. El acceso a los elementos de un array se realiza a través de sus índices. Un índice es un número entero que representa la posición del elemento en una determinada dimensión del array. El índice puede tomar un valor comprendido entre 1 y el número de elementos que tenga el array en esa dimensión. Una variable de tipo array puede declararse añadiendo paréntesis cuadrados tras el nombre de la clase, como se hace en el lenguaje Java, o bien añadiendo los paréntesis tras el nombre de la variable, como se hace en el lenguaje C. Por ejemplo:

```
Real[3]     vectorPosicion = {1,2,3};
Real[3,3]   matrizIdentidad = {{1,0,0}, {0,1,0}, {0,0,1}};
Real[3,3,3] arr3d;
```

Las sentencias anteriores declaran un vector de tres componentes, una matriz cuadrada con tres filas y tres columnas, y un array con tres dimensiones. De forma completamente equivalente, estos tres arrays pueden declararse indicando la dimensión tras el nombre de la variable:

```
Real  vectorPosicion[3] = {1,2,3};
Real  matrizIdentidad[3,3] = {{1,0,0}, {0,1,0}, {0,0,1}};
Real  arr3d[3,3,3];
```

En las declaraciones de los arrays `vectorPosicion` y `matrizIdentidad` se indican los valores que toman sus componentes. Para ello, se ha empleado el constructor de arrays: las llaves {}. Para hacer referencia a un determinado componente de un array `A` se escribe `A[i,j,...]`. El índice de la dimensión *k*-ésima puede tomar un valor entero comprendido entre 1 y `size(A,k)`, que es el número de elementos del array en la dimensión *k*-ésima. Pueden formarse submatrices empleando la notación `:` para los rangos de índices. Por ejemplo, `A[i1:i2, j1:j2]`, donde el rango `i1:i2` incluye todos los números de índice mayores o igual que `i1` y menores o igual que `i2`.

Pueden escribirse expresiones en las que intervienen arrays empleando los operadores aritméticos +, -, * y /. Estos operadores pueden operar escalares, vectores, matrices o arrays multi-dimensionales (cuando sean aplicables), cuyos elementos sean del tipo `Real` o `Integer`. El operador multiplicación * tiene el significado de producto escalar cuando se usa entre vectores, de producto de matrices cuando se usa entre matrices o entre una matriz y un vector, y de producto elemento a elemento cuando se usa entre un array y un escalar. Por ejemplo, el producto del vector vectorPosicion por el escalar `2` se expresa:

```
vectorPosicion*2
```

y da como resultado:

```
{2,4,6}
```

A diferencia de lo que sucede en Java, en Modelica los arrays de dimensionalidad > 1 son siempre rectangulares, al igual que sucede en Matlab o en Fortran.

Existe cierto número de funciones propias del lenguaje Modelica que operan sobre arrays. Algunas de ellas se muestran en la tabla siguiente:

| | |
|---|---|
| `transpose (A)` | Permuta las dos primeras dimensiones del array A. |
| `zeros (n1,n2,n3,...)` | Devuelve un array de dimensión $n_1$ x $n_2$ x $n_3$ x..., donde todos los componentes son de tipo entero e iguales a cero. |
| `ones(n1,n2,n3,...)` | Devuelve un array de dimensión $n_1$ x $n_2$ x $n_3$ x ..., donde todos los componentes son de tipo entero e iguales a uno. |
| `fill (s,n1,n2,n3, ...)` | Devuelve un array de dimensión $n_1$ x $n_2$ x $n_3$ x ... con todos los elementos igual el valor de la expresión escalar s. |
| `min (A)` | Devuelve el menor componente del array obtenido de evaluar la expresión A. |
| `max (A)` | Devuelve el mayor componente del array obtenido al evaluar la expresión A. |
| `sum (A)` | Devuelve la suma de todos los componentes del array obtenido de evaluar la expresión A. |

Las funciones escalares de Modelica, que tienen un argumento escalar, son igualmente aplicables a arrays. En este caso, se aplica la función a cada uno de los componentes del array. Por ejemplo, si A es un vector de números reales, entonces `cos(A)` es el vector obtenido de aplicar la función `cos` a cada uno de los componentes de A. Por ejemplo:

```
cos({1, 2, 3})  =  {cos(1), cos(2), cos(3)}
```

Los arrays pueden concatenarse entre sí empleando `cat(k,A,B,C,...)`, que concatena los arrays `A,B,C,...` en la dimensión `k`-ésima. Por ejemplo, `cat(1,{2,3}, {5,8,4})` da como resultado `{2,3,5,8,4}`.

Los casos especiales más comunes, que son la concatenación en la dimensión uno y dos, son soportados mediante una sintaxis especial: `[A;B;C;...]` y `[A,B,C,...]` respectivamente. Puede combinarse el uso de ambas formas sintácticas en una determinada expresión de concatenación. Con el fin de conseguir compatibilidad con la sintaxis para arrays de Matlab, que constituye un estándar de facto, los argumentos escalares y vectoriales de estos operadores especiales son convertidos en matrices antes de realizar la concatenación. Como resultado de ello, puede

construirse una matriz a partir de expresiones escalares separando las filas por punto y coma, y las columnas por comas. El ejemplo siguiente crea una matriz $m \times n$:

```
[expr11, expr12, ... expr1n;
expr21, expr22, ... expr2n;
…
exprm1, exprm2, ... exprmn]
```

Resulta instructivo seguir el proceso de creación de una matriz, a partir de expresiones escalares, usando estos operadores. Por ejemplo:

```
[1,2;
 3,4]
```

En primer lugar, cada argumento escalar es convertido en una matriz, obteniéndose:

```
[{{1}}, {{2}};
 {{3}}, {{4}}]
```

Puesto que [... , ...] (concatenación en la dimensión dos) tiene mayor prioridad que [... ; ...], (concatenación en la dimensión uno), el primer paso del proceso de concatenación produce el resultado:

```
[{{1, 2}};
 {{3, 4}}]
```

Finalmente, las matrices fila son concatenadas, obteniéndose la matriz $2 \times 2$ deseada:

```
 {{1, 2},
  {3, 4}}
```

El caso particular en el cual hay un único argumento escalar permite crear una matriz $1 \times 1$. Por ejemplo:

```
[1]
```

da como resultado la matriz:

```
{{1}}
```

## 2.14 Construcciones algorítmicas

Si bien las ecuaciones resultan especialmente adecuadas para el modelado de sistemas físicos, existen situaciones en las cuales también se requiere el uso de construcciones algorítmicas.

Típicamente, éste es el caso de los algoritmos: descripciones procedurales de cómo realizar determinados cálculos, consistentes normalmente en cierto número de sentencias que deben ser ejecutadas siguiendo un orden especificado.

### 2.14.1 Secciones de Algoritmos y Sentencias de Asignamiento

En Modelica, las sentencias algorítmicas sólo pueden aparecer dentro de las secciones de algoritmo. Estas secciones comienzan por la palabra reservada `algorithm`. Las secciones de algoritmo también se denominan ecuaciones algorítmicas. Esto es debido a que una sección de algoritmo puede interpretarse como un conjunto de ecuaciones, en las que intervienen una o más variables, que puede aparecer entre secciones de ecuaciones. Las secciones de algoritmo son delimitadas por la aparición de una de las siguientes palabras clave: `equation`, `public`, `protected`, `algorithm`, o `end`.

```
algorithm
  ...
  <sentencias>
  ...
  <alguna otra palabra clave>
```

Una sección algorithm puede estar embebida entre secciones equation, tal como se muestra en el ejemplo siguiente, en el cual la sección algoritmo contiene tres sentencias de asignación.

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y-5;
  x1 := x2+y;
equation
  u = x1+x2;
  ...
```

Obsérvese que en las ecuaciones algorítmicas, que componen el código de la sección algoritmo, se usa el valor de ciertas variables evaluadas fuera de la sección algoritmo. Estas variables se denominan *variables de entrada* al algoritmo—en este ejemplo `x`, `y`, y `z`. Análogamente, las variables que son evaluadas en la sección algoritmo se denominan *salidas del algoritmo*—en este ejemplo `x1` y `x2`. Esta terminología responde a la analogía existente entre una sección algoritmo y una función. De este modo, la sección algoritmo se corresponde con el cuerpo de una función.

Los parámetros formales `input` y `output` de la función se corresponden con las variables de entrada y salida del algoritmo respectivamente.

### 2.14.2 Sentencias

Además de las sentencias de asignación, que se usaron en el ejemplo anterior, en Modelica existen otros tres tipos de sentencias "algorítmicas": sentencias `if-then-else`, ciclos `for` y ciclos `while`. El código mostrado a continuación, que realiza una suma, contiene un ciclo while y una sentencia if , donde `size (a,1)` devuelve el número de componentes de la primera dimensión del array `a`. En general, las partes elseif y else de una sentencia if son opcionales.

```
suma := 0;
n := size(a,1);
while n>0 loop
  if a[n]>0 then
    suma := suma + a[n];
  elseif a[n] > -1 then
    suma := suma - a[n] -1;
  else
    suma := suma - a[n];
  end if;
  n := n-1;
end while;
```

Tanto los ciclos for como los while pueden finalizar inmediatamente cuando se ejecuta una sentencia break dentro del ciclo. Dicha sentencia consiste en la palabra clave `break` seguida de un punto y coma.

Considere de nuevo el cálculo del polinomio mostrado en la Sección 2.6.1, en la que se explicaron las estructuras repetitivas de ecuaciones.

```
y := a[1]+a[2]*x + a[3]*x^1 + ... + a[n+1]*x^n;
```

Cuando se describe mediante ecuaciones el cálculo del polinomio, es necesario emplear un vector auxiliar `xpowers` para almacenar las diferentes potencias de `x`. Alternativamente, puede realizarse el mismo cálculo como un algoritmo, empleando un ciclo `for` tal como se muestra a continuación. En este caso, el cálculo puede realizarse sin necesidad de usar un vector extra. Basta con usar una variable escalar `xpower` para almacenar la potencia de `x` calculada más recientemente.

```
algorithm
```

```
  y := 0;
  xpower := 1;
  for i in 1:n+1 loop
    y := y + a[i]*xpower;
    xpower := xpower*x;
  end for;
   ...
```

### 2.14.3 Funciones

Las funciones son una parte natural de cualquier modelo matemático. El lenguaje Modelica contiene cierto número de funciones matemáticas como `abs`, `sqrt`, `mod`, etc. Otras funciones, tales como `sin`, `cos`, `exp`, etc., están disponibles en la librería matemática estándar de Modelica `Modelica.Math`. Puede considerarse que los operadores aritméticos +, -, *, / son funciones, que son invocadas mediante una sintaxis conveniente de operadores. Asimismo, el lenguaje Modelica permite al usuario definir sus propias funciones. El cuerpo de una función en Modelica es una sección algoritmo, que contiene el código algorítmico procedural que debe ser ejecutado cuando se llama a la función. Las palabras clave `input` y `output` permiten definir los parámetros formales de entrada de la función y los resultados respectivamente. De esta forma, la sintaxis para la definición de las funciones es muy similar a la sintaxis de las clases tipo `block` de Modelica.

En Modelica, las funciones deben ser *funciones matemáticas*: sin efectos adicionales globales y sin memoria. Si se llama a una función repetidamente, usando en todos los casos unos determinados valores de los argumentos, debe obtenerse en todos los casos el mismo resultado. A continuación, se muestra el código de la función `evaluaPolinomio`, que evalúa un polinomio.

```
function evaluaPolinomio
  input  Real a[:];
   // Array, tamaño definido en el tiempo de llamada de la función
  input  Real x := 1.0;   // El valor por defecto de x es 1.0
  output Real y;
protected
  Real   xpower;
algorithm
  y := 0;
  xpower := 1;
  for i in 1:size(a,1) loop
    y := y + a[i]*xpower;
    xpower := xpower*x;
  end for;
```

```
end evaluaPolinomio;
```

Normalmente, la posición de los argumentos en la llamada a la función se emplea para asociar estos con los parámetros formales de la función. Por ejemplo, en la llamada que se muestra en el párrafo siguiente, el argumento `{1,2,3,4}` se asocia con el parámetro formal `a`, que es un vector, y `21` se asocia con el parámetro formal `x`. Los parámetros de las funciones en Modelica son de solo lectura, es decir, no se les puede asignar valor en el código de la función. Cuando una función es llamada usando asociación posicional de argumentos, es imprescindible que el número de argumentos y de parámetros sea el mismo. Además, el tipo de cada argumento debe ser compatible con el tipo declarado del correspondiente parámetro formal. Esto permite pasar arrays de longitud arbitraria como argumentos a funciones cuyos correspondientes parámetros formales son arrays de longitud indefinida, es decir, cuya longitud no ha sido especificada al declararlos en la función. Tal es el caso del parámetro formal de entrada `a` en la función `evaluaPolinomio`.

```
p = evaluaPolinomio({1, 2, 3, 4}, 21);
```

Como se muestra en el ejemplo siguiente, la misma llamada a la función `evaluaPolinomio` puede realizarse indicando el nombre de los parámetros, de modo que se use éste para realizar la asociación con los argumentos. Realizar las llamadas de esta forma tiene la ventaja de que se obtiene un código mejor auto documentado, así como más flexible frente a cambios en el código de la función.

Por ejemplo, si en la llamada a la función `evaluaPolinomio` se incluye el nombre de sus dos parámetros formales, entonces puede especificarse el argumento asignado a `a` y a `x` en cualquier orden. Asimismo, pueden añadirse nuevos parámetros formales en la definición de la función, que deberán tener valores por defecto asignados, sin que se produzca error de compilación en la llamada a la función. En la llamada a una función puede omitirse el argumento de aquellos parámetros que tengan valor por defecto, a no ser que se quiera modificar dicho valor.

```
p = evaluaPolinomio(a={1, 2, 3, 4}, x=21);
```

Las funciones pueden calcular múltiples resultados. Por ejemplo, la función `f` del párrafo siguiente calcula sus tres parámetros formales de salida: `r1`, `r2` y `r3`.

```
function f
  input Real x;
  input Real y;
  output Real r1;
  output Real r2;
  output Real r3;
  ...
end f;
```

Dentro de una sección algoritmo, solo pueden incluirse llamadas a funciones que calculen múltiples resultados en sentencias de asignamiento especiales, como la mostrada a continuación. En este caso se asigna a cada variable del lado izquierdo el correspondiente resultado de la llamada a la función.

```
(a, b, c) := f(1.0, 2.0);
```

En las ecuaciones, se emplea una sintaxis similar:

```
(a, b, c) = f(1.0, 2.0);
```

La ejecución de una función finaliza, y ésta devuelve su resultado, cuando se ejecuta la última asignación del cuerpo de la función o bien cuando se ejecuta una sentencia return.

### 2.14.4 Sobrecarga de Operadores y Números Complejos

La sobrecarga de las funciones y de los operadores permite definir varias funciones u operadores con un mismo nombre, pero con un conjunto diferente de tipos de parámetros formales de entrada para cada definición. Esta capacidad permite, por ejemplo, definir los operadores suma y producto de números complejos, usando los operadores ordinarios + y *, pero con nuevas definiciones. También, permite que coexistan varias definiciones de una función `solve`, que resuelva ecuaciones matriciales lineales, adecuadas para diferentes tipos de matrices, tales como matrices estándar densas, matrices sparse, matrices simétricas, etc.

De hecho, la sobrecarga de operadores ya existe predefinida con un limitado alcance para ciertos operadores en el lenguaje Modelica. Por ejemplo, el operador mas (+) para suma tiene varias definiciones diferentes dependiente del tipo de dato:

- 1+2 significa suma entera de dos enteros constants da un resultado entero, en este caso 3.
- 1.0+2.0 significa la suma en numerous de punto flotante de dos constants tipo `Real` dando como resultado un número en punto flotante, en este caso 3.0.
- “ab”+”2” significa concatenación de cadenas de dos constantes tipo `string` dando como resultado una cadena, en este caso “ab2”.
- {1,2}+{3,4} significa la suma de dos vectores constantes de enteros dando como resultado un vector, en este caso {4,6}.

La sobrecarga de operadores para tipos de datos definidos por el usuario pueden ser definidos usando las declaraciones `operator record` y `operator function`. A continuación se muestra parte de un ejemplo de un tipo de datos de números complejos:

```
Operador record Complex "Registro que define un número Complejo"
    Real re "Parte real del número complejo"
    Real im "Parte Imaginaria del número complejo"

    encapsulate operator 'Constructor'
      import Complex;

      function desdeReal
        input Real re;
        output Complex Resultado = Complex(re=re, im=0.0);
          annotation(Inline=true);
      end desdeReal
    end 'Constructor'

    encapsulate operator function '+'
      import Complex;
        input Complex c1;
        input Complex c2;
        output Complejo Resultado "Lo mismo como: c1 + c2";
          annotation(Inline=true);
      algorithm
        resultado := Complex(c1.re+c2.re, c1.im+c2.im);
    end '+';

end Complex;
```

Como es usual, en este ejemplo nosotros iniciamos con las declaraciones de la parte real y la parte imaginaria de los campos `re` e `im` de la declaración del registro del operador Complejo. Le sigue una definición tipo `constructor`, en este caso `desdeReal`, con un argumento de entrada solamente en vez de las dos entradas de un constructor `Complex` implícitamente definido a través de la definición del registro `Complex`. Le sigue la definición del operador de sobrecarga para '+'. Cómo se pueden utilizar estas definiciones? Para ello observe el siguiente pequeño ejemplo:

```
Real a;
Complex x;
Complex c = a + b;
  // Suma de un número Real a y un número Complejo b
```

La parte interesante está en la tercera línea, la cual contiene una adición `a + b` de un número tipo `Real a` y un número tipo `Complex b`. No existe un operador de suma intrínseco para números complejos, pero se tiene la definición del operador sobrecargado que se ha codificado arriba de

‘+’ para dos numerous complejos. Una suma de dos números complejos debería coincidir con esta definición en forma directa en el proceso de evaluación de la sentencia.
Sin embargo, en este caso se ha realizado una suma de un número real y un número complejo. Afortunadamente, el proceso de evaluación para el operador binario sobrecargado puede manejar también este caso si hay una función `constructor` en la definición del registro `Complex` que pueda convertir un número real a un número complejo. En este caso se tiene este constructor llamado `desdeReal`.
Obsérvese que `Complex` se encuentra predefinido en una librería de Modelica tal que pueda ser usada directamente.

### 2.14.5 Funciones Externas

En Modelica, es posible realizar llamadas a funciones definidas en lenguaje C o en Fortran. Si no se especifica el lenguaje de programación externo, se asume por defecto que la función está escrita en C. El cuerpo de una función externa es marcado por la palabra clave `external` en la declaración de la función externa en Modelica.

```
function log
  input Real x;
  output Real y;
external
end log;
```

La interfaz para las funciones externas soporta algunas capacidades avanzadas. Éstas incluyen: el uso de parámetros que sean simultáneamente de entrada y de salida, de arrays que son usados localmente en los cálculos por las funciones externas, asociación con los argumentos de la función externa mediante el orden en la llamada, especificación explícita de la forma en que deben almacenarse en memoria los arrays, etc. Por ejemplo, en la función `leastSquares` mostrada más abajo, el parámetro formal `Ares` corresponde a un parámetro que es simultáneamente de entrada y de salida de la función externa: su valor de entrada por defecto es `A` y se obtiene un valor diferente como resultado. Es posible controlar el orden y el uso de los parámetros de las funciones externas a Modelica. Esta capacidad se usa para pasar el tamaño de los arrays en la llamada a la rutina `dgels`. Algunas rutinas antiguas de Fortran, como `dgels`, necesitan que se les pase como parámetros el área de trabajo usada por la subrutina, es decir, el vector que usará la subrutina para realizar internamente sus cálculos. Este es el caso del vector `work`, que es definido como una variable local de la función en Modelica, en la sección `protected`.

```
function leastSquares "Resuelve un problema lineal de mínimos
cuadrados"
  input  Real A[:,:];
  input  Real B[:,:];
  output Real Ares[size(A,1),size(A,2)] := A;
     // Ares es sobrescrito,
     // obteniéndose como resultado la factorización
  output Real x[size(A,2),size(B,2)];
protected
  Integer lwork = min(size(A,1),size(A,2))+
                  max(max(size(A,1),size(A,2)),size(B,2))*32;
  Real work[lwork];
  Integer info;
  String transposed="NNNN"; // Truco para pasar datos CHARACTER
                            // a la rutina Fortran
  external "FORTRAN 77"
  dgels(transposed, 100, size(A,1), size(A,2), size(B,2), Ares,
        size(A,1), B, size(B,1), work, lwork, info);
end leastSquares;
```

### 2.14.6 Algoritmos Vistos como Funciones

El concepto de función es un bloque de construcción básico en la definición semántica o significado de la construcción de los lenguajes de programación. Algunos lenguajes de programación se basan completamente en el empleo de funciones matemáticas. Por ello, resulta útil intentar comprender y definir la semántica de las secciones algoritmo de Modelica en términos de funciones. Por ejemplo, considere la sección algoritmo mostrada a continuación, que aparece entre secciones equation:

```
algorithm
  y := x;
  z := 2*y;
  y := z+y;
  ...
```

Sin alterar su significado, este algoritmo puede ser transformado en una ecuación y una función, tal como se muestra a continuación. La ecuación iguala las variables de salida del algoritmo a los resultados de la función `f`. Los parámetros formales de entrada de la función `f` son las entradas a la sección algoritmo. Los parámetros de salida de la función son las salidas de la sección

algoritmo. El código algorítmico de la sección algorithm se ha transformado en el cuerpo de la función.

```
(y,z) = f(x);
...
function f
  input  Real x;
  output Real y,z;
algorithm
  y := x;
  z := 2*y;
  y := z+y;
end f;
```

## 2.15 Eventos Discretos y Modelado Híbrido

En general, los sistemas físicos macroscópicos evolucionan en el tiempo de manera continua, obedeciendo las leyes de la física. Tal es el caso de las partes en movimiento de los sistemas mecánicos, el valor de las corrientes y tensiones en un circuito eléctrico, las reacciones químicas, etc. Se dice que estos sistemas poseen dinámicas continuas.

Por otra parte, en algunos casos es provechoso hacer la aproximación de que ciertos componentes de un sistema muestran un comportamiento discreto. Esto es, suponer que los cambios en las variables del sistema ocurren instantáneamente y de manera discontinua, en instantes específicos de tiempo.

En los sistemas físicos reales, los cambios pueden producirse de manera muy rápida, pero nunca son instantáneos. Algunos ejemplos son las colisiones en los sistemas mecánicos, tal como una pelota que al rebotar cambia de manera casi instantánea su dirección, los interruptores en los circuitos eléctricos, que producen cambios muy rápidos en la tensión, las válvulas y bombas en las plantas químicas, etc. Se dice que estos componentes del sistema tienen una dinámica de tiempo discreto. El motivo para realizar esta aproximación de discretización es simplificar el modelo matemático del sistema, haciéndolo más fácilmente resoluble, y consiguiendo a menudo con ello que el tiempo necesario para simular el modelo se reduzca en varios órdenes de magnitud.

Por esta razón, Modelica permite emplear en los modelos variables con *variabilidad discreta en el tiempo*. El valor de cada una de estas variables cambia únicamente en instantes específicos de tiempo, denominados *eventos*, y se mantiene constante en el intervalo de tiempo entre dos eventos consecutivos (véase la Figura 2-22). Ejemplos de variables de tiempo discreto son las

variables `Real` declaradas con el prefijo `discrete`. También, las variables `Integer`, `Boolean` y `enumeration`, que siempre son de tiempo discrete y no pueden ser en tiempo continuo.

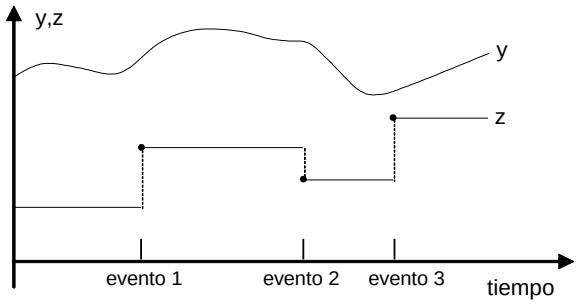



**Figura 2-22.** Las variables de tiempo discreto, como z, cambian de valor sólo en los instantes de evento, mientras que las variables de tiempo continuo, como y, pueden cambiar de valor tanto en los eventos como en los intervalos entre los eventos.

Puesto que la aproximación de tiempo discreto sólo es aplicable a ciertos subsistemas, a menudo el modelo del sistema está compuesto por componentes de tiempo discreto y componentes de tiempo continuo, interactuando todos ellos entre sí. Estos sistemas se denominan *sistemas híbridos* y las técnicas de modelado asociadas *modelado híbrido*. La introducción de los modelos matemáticos híbridos conlleva la aparición de nuevas dificultades en la resolución del modelo, si bien las ventajas de este tipo de modelos superan con creces a los inconvenientes.

Modelica proporciona dos tipos de cláusulas para describir los modelos híbridos. El primer tipo son las expresiones o ecuaciones condicionales, que permiten describir modelos discontinuos y condicionales. El segundo tipo son las ecuaciones tipo when, que permiten describir aquellas ecuaciones que son válidas únicamente en el instante en que se produce el evento. Este instante es aquel en el cual una determinada condición pasa de ser falsa a ser verdadera. Por ejemplo, las expresiones condicionales if-then-else permiten modelar fenómenos que son descritos mediante diferentes expresiones en diferentes regiones de operación. Tal es el caso de la siguiente ecuación, que modela un limitador:

```
y = if v > limite then limite else v;
```

El modelo de un diodo ideal constituye un ejemplo más completo de modelo condicional. En la Figura 2-23 se muestra la curva característica de un diodo real y en la Figura 2-24 la curva característica de un diodo ideal en forma parametrizada.

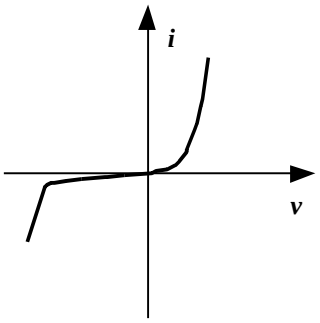



**Figura 2-23.** Curva característica de un diodo real.

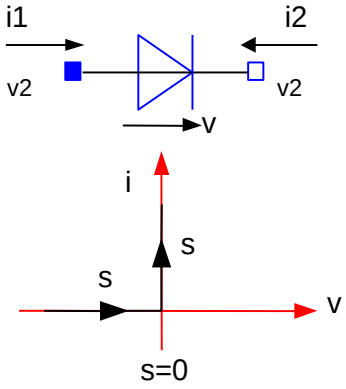



**Figura 2-24.** Curva característica de un diodo ideal.

Puesto que el voltaje del diodo ideal se hace infinito en un diagrama voltaje-corriente ordinario, resulta más apropiado realizar una descripción paramétrica, en la cual tanto el voltaje `v` como la corriente `i`, que es igual a `i1`, sean funciones del parámetro `s`. Cuando el diodo se encuentra en corte, la corriente que circula es cero y la caída de tensión es negativa. Cuando se encuentra en conducción, circula corriente y la caída de tensión es cero.

```
model Diodo "Diodo ideal"
  extends DosPines;
  Real s;
  Boolean off;
```

```
equation
  off = s < 0;
  if off
    then v=s;
    else v=0;   // ecuaciones condicionales
  end if;
  i = if off then 0 else s;   // expresión condicional
end Diodo;
```

La cláusula when se introdujo en Modelica con el fin de expresar ecuaciones instantáneas. Esto es, ecuaciones que son válidas únicamente en ciertos instantes puntuales de tiempo, a los que se denomina *eventos*. El evento se produce cuando determinada condición lógica pasa de ser falsa a ser verdadera. A continuación, se muestra la sintaxis de la cláusula when activada por un vector de condiciones. Las ecuaciones de la cláusula son válidas únicamente en el instante en que al menos una de las condiciones del vector pasa de valer falso a valer verdadero. Es decir, las ecuaciones permanecen activas solamente durante un instante de tiempo de duración cero. Además de un vector, la condición puede ser una única expresión lógica.

```
when {condición1, condición2, …} then
  <ecuaciones>
end when;
```

Una pelota que rebota es un buen ejemplo de un sistema híbrido, en cuyo modelo es apropiado usar la cláusula when. El movimiento de la pelota es caracterizado por dos variables: su altura desde el suelo, `altura`, y su velocidad vertical, `velocidad`. La pelota se mueve de manera continua entre un rebote y el siguiente, siendo en los instantes de los rebotes cuando ocurren los cambios discretos (véase la Figura 2-25). Cuando la pelota rebota contra el suelo, cambia el sentido de su velocidad. Una pelota ideal tiene un coeficiente de elasticidad igual a 1, con lo cual no pierde energía en el rebote. Una pelota más realista, como la modelada a continuación, tiene un coeficiente de elasticidad de 0.9, con lo cual sólo conserva el 90 por ciento de su velocidad tras el rebote.

El modelo de la pelota contiene las dos ecuaciones básicas del movimiento, que relacionan su altura, su velocidad y la fuerza de la gravedad. En el instante del rebote, se invierte instantáneamente el sentido de la velocidad y decrece su valor: `velocidad` (tras el rebote) = `-c*velocidad` (antes del rebote). Este cambio se describe mediante una ecuación instantánea de reinicialización, usando la sentencia `reinit(velocidad,-c*pre(velocidad))`. La sentencia `reinit` reinicializa el valor de la variable `velocidad`.

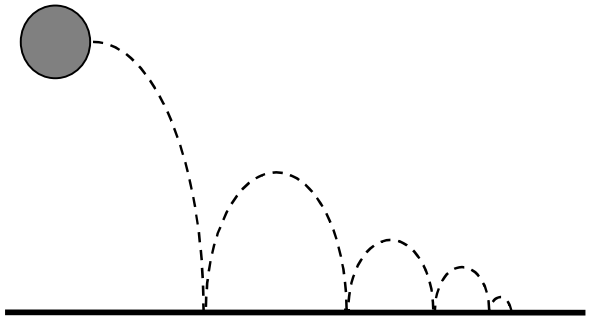


**Figura 2-25.** Una pelota que rebota.

```
model RebotePelota "Modelo sencillo de una pelota que rebota"
  constant  Real g = 9.81    "Constante de la Gravedad";
  parameter Real c = 0.9     "Coeficiente de restitución";
  parameter Real radio =0.1  "Radio de la pelota";
  Real altura(start = 1)     "Altura del centro de la pelota";
  Real velocidad(start = 0)  "Velocidad de la pelota";
equation
  der(altura) = velocidad;
  der(velocidad) = -g;
  when altura <= radio then
     reinit(velocidad,-c*pre(velocidad));
  end when;
end RebotePelota;
```

Obsérvese que las ecuaciones de una cláusula when están activas únicamente en el instante de tiempo en el que la condición de la cláusula when pasan de valer `false` a valer `true`. Por el contrario, las ecuaciones condicionales de una cláusula if están activas siempre que la condición de la cláusula if valga `true`.

Si se simula este modelo durante un tiempo suficientemente largo, la pelota terminará atravesando el suelo y moviéndose hacia valores cada vez más negativos de la altura. Este extraño comportamiento de la simulación se denomina castañeteo (chattering) o efecto Zeno. Se produce debido a la limitada precisión con la que el computador almacena los números en coma flotante y también debido al mecanismo de detección de los eventos que emplea el simulador. Ocurre en algunos modelos en los cuales los eventos son activados en instantes de tiempo infinitamente próximos entre sí. En este caso, el problema es que el modelo del impacto no es realista: la ley `velocidad_nueva = -c*velocidad` no es válida para velocidades muy pequeñas. Una forma

sencilla de solucionar el problema es detectar cuando la pelota cae a través del suelo y entonces conmutar a una ecuación que establezca que la pelota permanezca justo apoyada sobre el suelo. Una solución mejor, aunque más complicada, es conmutar a un modelo más realista de los materiales.

## 2.16 Paquetes

Los conflictos entre nombres son un problema a considerar cuando se desarrolla código reutilizable. Por ejemplo, librerías de clases en Modelica y funciones para varios dominios de aplicación. Por mucho cuidado que se ponga en escoger los nombres para las clases y las variables, es probable que alguna otra persona esté usando ese mismo nombre con otro propósito. Este problema se acentúa cuando se usan nombres descriptivos cortos, que son más cómodos de usar y por tanto bastante populares, con lo cual es bastante probable que sean usados en el código desarrollado por otra persona.

Una solución común para evitar conflictos de nombres es añadir un prefijo corto común a los elementos que están relacionados y agruparlos todos ellos dentro de un paquete. Por ejemplo, todos los nombres de la toolkit de X-Windows tienen el prefijo Xt, y el prefijo para la API de 32-bit de Windows es WIN32. Esto funciona relativamente bien para un número pequeño de packages, pero la probabilidad de que se produzcan colisiones de nombres aumenta a medida que crece el número de paquetes.

Algunos lenguajes de programación, como Java y Ada, así como Modelica, proporcionan un método más seguro y sistemático de evitar el conflicto de nombres a través del concepto de paquetes que en Modelica está sociado a la palabra clave `package`. Un paquete es simplemente un contenedor o espacio de nombres para los nombres de las clases, funciones, constantes y demás definiciones permitidas. El nombre del paquete se antepone como prefijo a todas las definiciones contenidas en el paquete, siguiendo la notación punto. Es posible *importar* definiciones dentro del espacio de nombres que constituye el paquete.

En Modelica, el concepto de paquete se introduce como una restricción y extensión del concepto clase. Así, puede emplearse la herencia para importar definiciones dentro del espacio de nombres de un paquete. Sin embargo, esto conlleva problemas conceptuales de modelado, ya que en este caso la finalidad de heredar sería importar, no especializar (que es el objetivo de la herencia). Para evitar esta dificultad, se ha introducido en Modelica el constructo de lenguaje `import` para los paquetes. En el ejemplo siguiente, se importa el tipo `Voltage`, junto con todas las demás definiciones contenidas en `Modelica.SIunits`, lo cual hace posible usarlo sin necesidad del prefijo al declarar la variable `v`. Por el contrario, en la declaración de la variable `i`

se usa el nombre completo `Modelica.SIunits.Ampere` del tipo `Ampere`, aun cuando también hubiera sido posible usar su versión corta. El nombre completo para `Ampere` permite encontrarlo de acuerdo con la organización jerárquica de la librería estándar de Modelica, la cual está situada conceptualmente en el nivel jerárquico superior.

```
package MiPaquete
  import Modelica.SIunits.*;

  class Foo;
    Voltage  v;
    Modelica.SIunits.Ampere  i;
  end Foo;

end MiPaquete;
```

Importar todas las definiciones de un paquete en otro, como se ha hecho en el ejemplo anterior, tiene la desventaja de que la introducción de nuevas definiciones en el paquete importado puede causar un conflicto de nombres con las definiciones contenidas en el paquete en el que se importa. Por ejemplo, si se introduce una definición llamada `v` en el package `Modelica.SIunits`, entonces se producirá un error de compilación en el package `MiPaquete`.

Existe una alternativa al problema de los nombres cortos, que no tiene la desventaja de la posible aparición de errores de compilación cuando se añaden nuevas definiciones a las librerías. Se trata de emplear alias como prefijos, en lugar de los prefijos obtenidos de aplicar la notación punto. Esto es posible usando la forma de renombrado de la sentencia `import`, como se muestra en el paquete `MpPaquete` mostrado abajo. En este caso, se introduce el alias `SI` en sustitución del nombre `Modelica.SIunits`, que es mucho más largo.

La forma en que se ha definido el package anterior presenta otra desventaja: se ha referenciado el tipo `Ampere` especificando su camino jerárquico completo, en lugar de emplear la sentencia `import`. Así pues, en el peor de los casos, para encontrar todas estas dependencias y las declaraciones a las que se refieren, será necesario:

- Inspeccionar visualmente el código fuente completo del paquete actual, lo que puede resultar largo.
- Buscar en todos los paquetes que contienen al paquete actual. Es decir, ascender en la jerarquía de paquetes, puesto que en un paquete pueden usarse los tipos y definiciones declarados en cualquier punto situado en un nivel jerárquico superior al actual.

En lugar de esto, un *paquete bien diseñado* debe especificar todas sus dependencias *explícitamente* mediante sentencias `import`, las cuales son fáciles de encontrar. Puede crearse un paquete de este tipo, por ejemplo el package `MiPaquete` mostrado abajo, anteponiendo la palabra

clave `encapsulated` a la palabra clave `package`. De esta forma, se evitan las búsquedas en niveles jerárquicos superiores al package encapsulado, puesto que todas las dependencias respecto a paquetes externos al actual deben declararse explícitamente mediante sentencias `import`. Este tipo de paquete encapsulado representa una unidad independiente de código y se corresponde más fielmente con el concepto de paquete encontrado en otros lenguajes de programación, tales como Java o Ada.

```
encapsulated package MiPaquete
  import SI = Modelica.SIunits;
  import Modelica;

  class Foo;
    SI.Voltage v;
    Modelica.SIunits.Ampere i;
  end Foo;
  ...
end MiPaquete;
```

## 2.17 Anotaciones

Una anotación en Modelica es información adicional que se asocia al modelo de Modelica. Esta información adicional es usada por los entornos de simulación de Modelica para, por ejemplo, soportar la documentación del modelo o su edición gráfica. La mayoría de las anotaciones no influyen en la ejecución de la simulación, es decir, se obtiene el mismo resultado si se elimina la anotación, si bien hay excepciones a esta regla. La sintaxis de una anotación es la siguiente:

**annotation**(*elementos_de_anotación*)

donde *elementos_de_anotación* es una lista de elementos de anotación, separados por comas, los cuales pueden ser cualquier tipo de expresión compatible con la sintaxis de Modelica. A continuación, se muestra una clase resistencia, en la cual se han incluido las propiedades gráficas de su icono como una anotación. Esta información deberá ser interpretada por el el editor de modelos gráficos.

```
model Resistencia
  annotation(Icon(coordinateSystem(
    preserveAspectRatio=true,
    extent={{-100,-100},{100,100}}, grid={2,2}),
  graphics = {Rectangle(
```

```
      extent={{-70, 30},{70, -30}},
      lineColor={0,0,255},fillColor={255,255,255},
      fillPattern=FillPattern.Solid),
      Line(points={{-90, 0},{-70, 0}},
       color={0,0,255})
        ...
  );
end Resistencia;
```

Otro ejemplo es la anotación predefinida `choices`, que es usada para generar menús en la interface gráfica de usuario:

```
annotation(choices(choice=1 "P",  choice=2 "PI",  choice=3 "PID"));
```

La anotación para funciones externas `arrayLayout` puede usarse para indicar explícitamente el formato de almacenamiento de los arreglos, por ejemplo, si se desea modificar la ordenación `rowMajor` y `columnMajor` para los lenguajes externos C y Fortran 77 respectivamente.

Este es uno de los raros casos en los cuales una annotation influye sobre los resultados de la simulación, puesto que un formato de almacenamiento erróneo para los arrays tendrá consecuencias en los cálculos matriciales. A continuación, se muestra un ejemplo:

```
annotation(arrayLayout = "columnMajor");
```

## 2.18 Convenciones de Nombres

Posiblemente, el lector se habrá dado cuenta de que en los ejemplos descritos en este capítulo se ha seguido un determinado criterio respecto a los nombres de las clases y de las variables. En efecto, se han seguido ciertas convenciones, que serán descritas a continuación. Estas convenciones de nombres han sido adoptados en la librería estándar de Modelica, haciendo el código más legible y, en cierta medida, reduciendo el riesgo de conflictos entre nombres. Las siguientes convenciones de nombres, que han sido aplicadss a lo largo de este libro, son recomendables para el código Modelica en general:

- Los nombres de los tipos y de las clases (pero normalmente no las funciones) comienzan siempre con mayúscula, por ejemplo, `Voltage`.
- Los nombres de las variables comienzan por minúscula , por ejemplo `cuerpo`, con la excepción de algunos nombres consistentes en una única letra, tal como T para la temperatura.

- En los nombres que consisten en varias palabras, la primera letra de cada palabra se escribe en mayúscula, con excepción de la primera palabra, a la que se aplican las reglas precedentes. Por ejemplo, `CorrienteElectrica` y `parteCuerpo`.
- El guión bajo sólo se emplea al final de los nombres, o al final de cada una de las palabras que compone el nombre, con el fin de indicar subíndices o superíndices. Por ejemplo, `cuerpo_bajo_arriba`.
- Los nombres recomendados para las instancias de los conectores en las clases (parciales) son: `p` y `n` para los conectores positivo y negativo en los componentes eléctricos y nombres conteniendo `a` y `b`, por ejemplo, `borde_a` y `borde_b`, para otros tipos de conectores, que son idénticos entre sí, y que aparecen a menudo en componentes con dos caras.

## 2.19 Librerías Estándar de Modelica

En gran medida, la gran potencialidad de Modelica para el modelado proviene de las facilidades que proporciona para la reutilización de clases de modelos. Las clases relacionadas con áreas en particular son agrupadas en paquetes, para facilitar su localización.

El paquete denominado `Modelica` es un paquete especial que, junto con el Lenguaje Modelica, es desarrollado y mantenido por la Modelica Association. Este paquete se conoce también como la *Librería Estándar de Modelica*. Contiene constantes, tipos, clases de conectores, modelos parciales y clases de modelos de componentes de varias áreas de aplicación, los cuales están agrupados en subpaquetes del paquete `Modelica`, que son denominados librerías estándar de Modelica.

A continuación, se enumera una parte del creciente conjunto de librerías estándar de Modelica que se encuentran disponibles en la actualidad:

| | |
|---|---|
| `Modelica.Constants` | Constantes matemáticas, físicas, etc. |
| `Modelica.Icons` | Definiciones de iconos gráficos que son usados en varios paquetes. |
| `Modelica.Math` | Definiciones de funciones matemáticas comunes. |
| `Modelica.SIUnits` | Definiciones de tipos basados en los nombres y las unidades estándar del sistema internacional (SI). |
| `Modelica.Electrical` | Modelos de componentes eléctricos comunes. |
| `Modelica.Blocks` | Bloques entrada/salida para su uso en diagramas de |

| | |
|---|---|
| | bloques. |
| `Modelica.Mechanics.Translational` | Componentes para mecánica traslacional 1D. |
| `Modelica.Mechanics.Rotational` | Componentes para mecánica rotacional 1D. |
| `Modelica.Mechanics.MultiBody` | Librería MBS —modelos mecánicos multi-cuerpo 3D. |
| `Modelica.Thermal` | Componentes para fenómenos térmicos, flujo de calor, etc. |
| ... | ... |

Están disponibles otras librerías en áreas de aplicación tales como la termodinámica, la hidráulica, los sistemas de potencia, comunicación de datos, etc.

La Librería Estándar de Modelica puede ser usada de forma gratuita, para fines comerciales y no comerciales, bajo las condiciones de *La Licencia de Modelica* que figura en las primeras páginas de este libro. La documentación completa de estas librerías, así como su código fuente, está disponible en el sitio web de Modelica.

Los modelos presentados hasta el momento han sido compuestos usando componentes de un único dominio de aplicación. Sin embargo, una de las mayores ventajas de Modelica es que posibilita la construcción de modelos multidominio, simplemente conectando entre sí componentes de librerías de diferentes dominios. El motor de corriente continua, mostrado en la Figura 2-26, es uno de los ejemplos más sencillos que permite ilustrar esta capacidad.

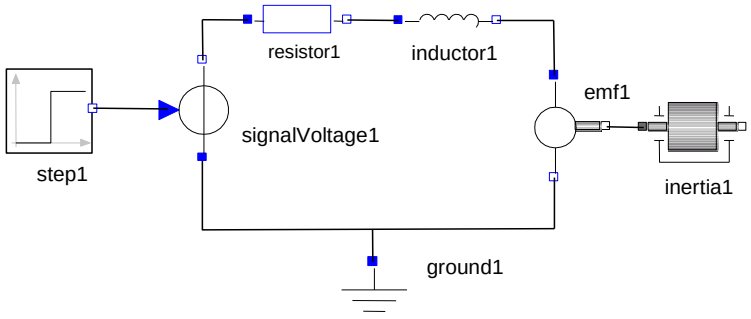



**Figura 2-26.** Modelo multidominio `DCMotorCircuit` compuesto por componentes mecánicos, eléctricos y bloques de señal.

Este modelo en particular contiene componentes de tres dominios: eléctrico, mecánico y bloques de señal, correspondientes a las librerías `Modelica.Mechanics`, `Modelica.Electrical` y `Modelica.Blocks`.

Los componentes de las librerías son particularmente fáciles de usar cuando se emplea un editor gráfico de modelos. A modo de ejemplo, en la Figura 2-27 se muestra el modelado del motor de corriente continua. La ventana de la izquierda muestra la librería `Modelica.Mechanics.Rotational`. El diseño gráfico del modelo puede hacerse pinchando y arrastrando con el ratón los iconos de los componentes desde la librería a la ventana central.

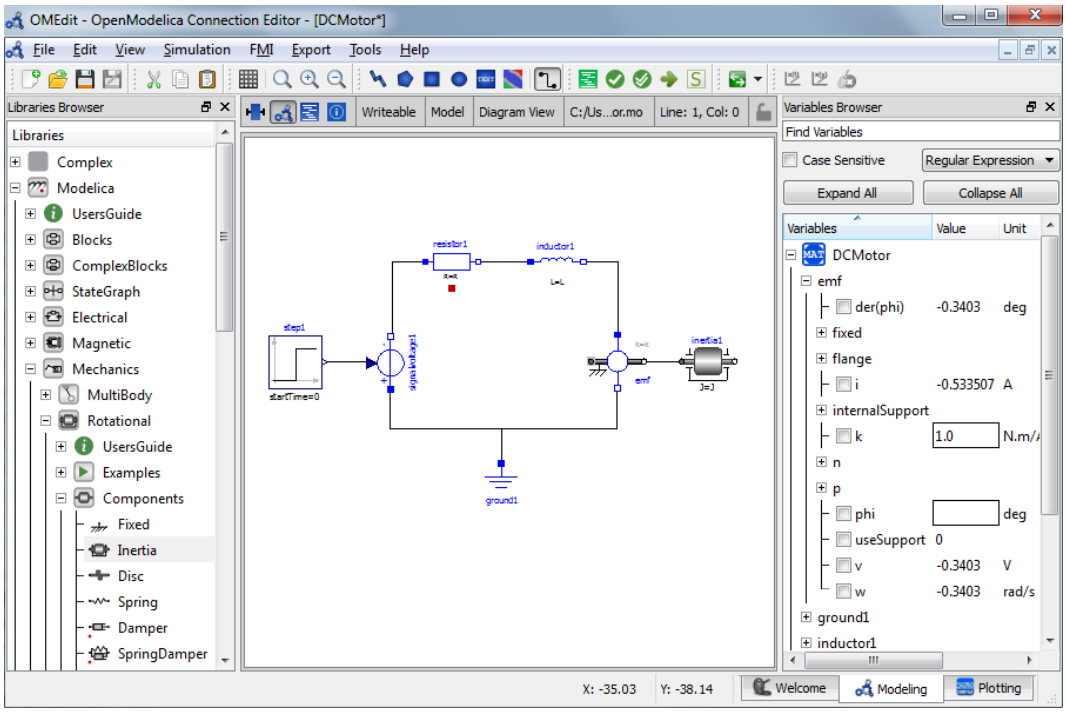



**Figura 2-27.** Edición gráfica de un modelo de un motor eléctrico de corriente continua, con los iconos de la librería `Modelica.Mechanics.Rotational` en la ventana izquierda.

## 2.20 Implementación y Ejecución de Modelica

Para comprender mejor cómo funciona Modelica, es útil examinar el proceso de traducción y ejecución de un modelo en Modelica. Dicho proceso aparece esquematizado en la Figura 2-28. En primer lugar, el código fuente escrito en Modelica es analizado y convertido a una representación

interna, normalmente a un árbol de sintaxis abstracta. Esta representación se analiza, se comprueban los tipos, las clases son heredadas y expandidas, se realizan las modificaciones e instanciaciones, se convierten las ecuaciones tipo `connect` a ecuaciones estándar, etc. El resultado de este proceso de análisis y traducción es un conjunto plano de ecuaciones, constantes, variables y definiciones de funciones. Con la única excepción de la notación punto en los nombres, no se conserva ninguna traza de la estructura orientada a objetos.

Después de obtener el modelo plano, todas las ecuaciones son ordenadas de acuerdo a las dependencias, respecto al flujo de datos, entre las ecuaciones. Las ecuaciones algebraico diferenciales (DAEs) no sólo son ordenadas, sino que también son manipuladas para transformar la matriz estructural del sistema a una forma triangular inferior por bloques. Esta transformación se denomina transformación BLT. A continuación, un módulo de optimización que contiene algoritmos para realizar simplificaciones algebraicas, métodos para la reducción de indices simbólicos, etc. elimina parte de las ecuaciones, conservando sólo aquel conjunto mínimo que deba ser resuelto numéricamente. Como un ejemplo trivial, si aparecen dos ecuaciones equivalentes sintácticamente, sólo se conserva una única copia de las dos ecuaciones. A continuación, aquellas ecuaciones independientes que tienen forma explícita son convertidas en sentencias de asignación. Esto es posible debido a que las ecuaciones han sido ordenadas y se ha establecido el orden de evaluación de las ecuaciones en conjunción con los pasos de integración del solucionador numérico. Si se obtiene un conjunto de ecuaciones fuertemente conectadas, este conjunto es transformado por un solucionador simbólico, que realiza cierto número de transformaciones algebraicas para simplificar la dependencia entre las variables. Algunas veces se resuelve un systema de ecuaciones diferenciales si se tienen una solución simbólica. Finalmente, se genera código C y se une con los solucionadores numéricos de ecuaciones, que resuelven este sistema de ecuaciones que ha sido simplificado en gran medida.

Los valores iniciales pueden tomarse de la definición del modelo o pueden ser especificados interactivamente por el usuario. Si es necesario, el usuario también especifica el valor de los parámetros. Un solucionador numérico de ecuaciones algebraico diferenciales (o, en casos sencillos, de ecuaciones diferenciales ordinarias) calcula el valor de las variables a lo largo del intervalo de simulación especificado $[t_0, t_1]$. El resultado de la simulación de un sistema dinámico es un conjunto de funciones del tiempo, tales como `R2.v(t)` en el modelo del circuito sencillo. Estas funciones pueden ser representadas gráficamente y/o salvadas en un archivo.

En la mayoría de los casos (pero no siempre), el rendimiento del código de simulación generado (incluyendo el solucionador) es similar al del código C escrito manualmente. A menudo, Modelica es más eficiente que el código C escrito directamente de manera manual, ya que aprovecha más oportunidades para optimizar simbólicamente el código de las que un programador humano podría manejar manualmente.

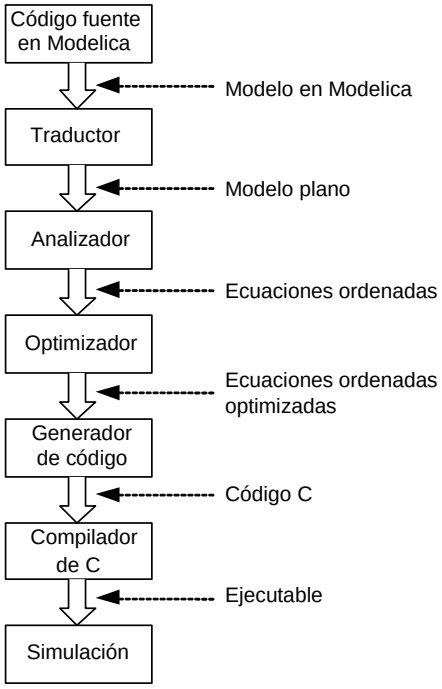



**Figura 2-28.** Etapas en la traducción y ejecución de un modelo en Modelica.

### 2.20.1 Traducción Manual del Modelo del Circuito Sencillo

Retomemos una vez más el modelo del circuito sencillo que se mostró en la Figura 2-7 y que se reproduce de nuevo en la Figura 2-29 para mayor comodidad del lector. Resulta instructivo traducir manualmente este modelo, con el fin de comprender el proceso.

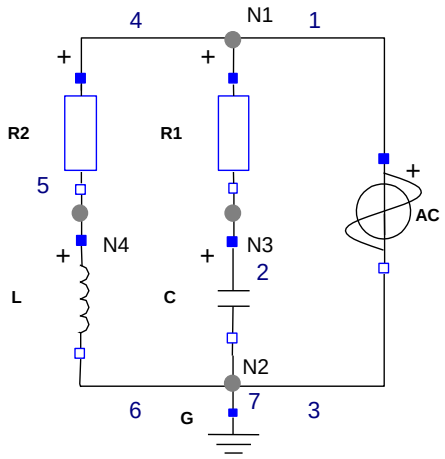



**Figura 2-29.** Una vez mas, modelo `CircuitoSimple` con indicación explícita de los nodos N1, N2, N3 y N4, y de los cables 1 a 7.

Las clases, instancias y ecuaciones son manipuladas, con el fin de expresarlas como un conjunto plano de ecuaciones, constantes y variables (vea las ecuaciones en la Tabla 2-1). Para ello, se siguen las reglas siguientes:

1. Para cada instancia de una clase, añadir una copia de todas las ecuaciones de esta instancia al sistema total de ecuaciones algebraico diferenciales (DAE) ó al sistema de ecuaciones diferenciales ordinarias (ODE). Ambas alternativas son posibles ya que en la mayoría de los casos un sistema DAE puede ser transformado en un sistema ODE.
2. Para cada conexión entre las instancias que aparece en el modelo, añadir las correspondientes ecuaciones de conexión al sistema DAE total. Estas ecuaciones de conexión consisten en igualar a cero la suma de las variables de tipo *flujo*, e igualar entre si las variables que son del tipo *no flujo*.

La ecuación `v=p.v-n.v` está definida en la clase `DosPines`. La clase `Resistencia` hereda la clase `DosPines`, incluyendo esta ecuación. La clase `CircuitoSimple` contiene la variable `R1`, que es de tipo `Resistencia`. En consecuencia, esta ecuación instanciada para `R1` es `R1.v=R1.p.v-R1.n.v`, la cual es añadida al sistema total de ecuaciones.

El cable etiquetado `1` se representa en el modelo como `connect (AC.p, R1.p)`. Las variables `AC.p` y `R1.p` son del tipo `Pin`. La variable `v` es una variable *no flujo* que representa el potencial de voltaje. Así pues, se genera la ecuación de igualdad `R1.p.v=AC.p.v`. La conexión de variables no flujo se traduce a ecuaciones de igualdad.

Observe que el otro cable (etiquetado como `4`) es conectada al mismo pin, `R1.p`. Esto se representa mediante una ecuación de conexión adicional: `connect(R1.p,R2.p)`. La variable `i` ha sido declarada como variable de flujo. Por ello, se genera la ecuación `AC.p.i+R1.p.i+R2.p.i=0`. La conexión entre variables de flujo se traduce a una ecuación que consiste en igualar a cero la suma de las variables. En este caso, esta ecuación corresponde con la segunda Ley de Kirchhoff.

El conjunto completo de ecuaciones (vea la Tabla 2-1) generado a partir de la clase `CircuitoSimple` consiste en 32 ecuaciones algebraico diferenciales. Éstas contienen 32 variables, además de la variable `time`, y de varios parámetros y constantes.

**Tabla 2-1.** Ecuaciones extraídas del modelo del circuito sencillo—un sistema DAE implícito.

| | | | |
|---|---|---|---|
| AC | 0 = AC.p.i+AC.n.i<br>AC.v = Ac.p.v-AC.n.v<br>AC.i = AC.p.i<br>AC.v = AC.VA*<br>sin(2*AC.PI*<br>AC.f*time); | L | 0 = L.p.i+L.n.i<br>L.v = L.p.v-L.n.v<br>L.i = L.p.i<br>L.v = L.L*der(L.i) |
| R1 | 0 = R1.p.i+R1.n.i<br>R1.v = R1.p.v-R1.n.v<br>R1.i = R1.p.i<br>R1.v = R1.R*R1.i | G | G.p.v = 0 |
| R2 | 0 = R2.p.i+R2.n.i<br>R2.v = R2.p.v-R2.n.v<br>R2.i = R2.p.i<br>R2.v = R2.R*R2.i | cables | R1.p.v = AC.p.v // cable 1<br>C.p.v = R1.n.v // cable 2<br>AC.n.v = C.n.v // cable 3<br>R2.p.v = R1.p.v // cable 4<br>L.p.v = R2.n.v // cable 5<br>L.n.v = C.n.v // cable 6<br>G.p.v = AC.n.v // cable 7 |
| C | 0 = C.p.i+C.n.i<br>C.v = C.p.v-C.n.v<br>C.i = C.p.i<br>C.i = C.C*der(C.v) | Flujo en los nodos | 0 = AC.p.i+R1.p.i+R2.p.i // N1<br>0 = C.n.i+G.p.i+AC.n.i+L.n.i // N2<br>0 = R1.n.i + C.p.i // N3<br>0 = R2.n.i + L.p.i // N4 |

En la Tabla 2-2 se muestran las 32 variables del sistema de ecuaciones, de las cuales 30 son algebraicas, puesto que no aparecen derivadas respecto al tiempo en el modelo. Dos variables, `C.v` y `L.i`, son variables dinámicas, puesto que sus derivadas respecto al tiempo si intervienen en el modelo. En este ejemplo sencillo, las dos variables dinámicas son variables de estado, por lo tanto el DAE puede reducirse a un ODE.

**Tabla 2-2.** Variables extraídas del modelo del circuito sencillo.

| | | | | |
|---|---|---|---|---|
| R1.p.i | R1.n.i | R1.p.v | R1.n.v | R1.v |
| R1.i | R2.p.i | R2.n.i | R2.p.v | R2.n.v |
| R2.v | R2.i | C.p.i | C.n.i | C.p.v |
| C.n.v | C.v | C.i | L.p.i | L.n.i |
| L.p.v | L.n.v | L.v | L.i | AC.p.i |
| AC.n.i | AC.p.v | AC.n.v | AC.v | AC.i |
| G.p.i | G.p.v | | | |

### 2.20.2 Transformación a la Forma de Espacio de Estados

El sistema de ecuaciones (sistema DAE) algebraico diferenciales implícitas, que se muestra en la Tabla 2-1, debe ser transformado y simplificado antes de aplicarle el solucionador numérico. El siguiente paso para ello es clasificar las variables que intervienen en el sistema DAE. Existen los siguientes cuatro grupos de variables:

1. Aquellas variables constantes que son parámetros del modelo son agrupadas en un vector de parámetros p. Son parámetros aquellas variables cuyo valor puede ser modificado entre sucesivas ejecuciones de la simulación y que han sido declaradas usando el prefijo `parameter`. Las demás constantes pueden ser reemplazadas por su valor, desapareciendo del modelo como variables con nombre.
2. Las variables que son declaradas con el atributo input (es decir, anteponiendo la palabra clave `input`) y que aparecen en las instancias del nivel jerárquico superior, son agrupadas en un vector de entradas *u*.
3. Las variables cuya derivada interviene en el modelo (variables dinámicas), es decir, aquellas variables sobre las que se ha aplicado el operador `der`( ), son agrupadas en un vector de estados *x*.
4. Todas las restantes variables, que son aquellas cuya derivada no interviene en el modelo, son agrupadas en un vector de variables algebraicas *y*.

Para el modelo del circuito simple, estos cuatro grupos de variables son los siguientes:

*p* = {R1.R, R2.R, C.C, L.L, AC.VA, AC.f}
*u* = {AC.v}
*x* = {C.v,L.i}
*y* = {R1.p.i, R1.n.i, R1.p.v, R1.n.v, R1.v, R1.i, R2.p.i, R2.n.i,

```
R2.p.v, R2.n.v, R2.v, R2.i, C.p.i, C.n.i, C.p.v, C.n.v, C.i, L.n.i,
L.p.v, L.n.v, L.v, AC.p.i, AC.n.i, AC.p.v, AC.n.v, AC.i, AC.v, G.p.i,
G.p.v}
```

Se desea expresar el problema mediante el sistema de ecuaciones diferenciales ordinarias (ODE) de menor dimensión posible (en el caso más general, será un sistema DAE), y calcular el valor de las restantes variables a partir de la solución de este problema de dimensión mínima. Preferiblemente, el sistema de ecuaciones estará expresado en la forma explícita de espacio de estados, que es el que se muestra a continuación.

$$\dot{x} = f(x,t) \tag{2-3}$$

Esto es, la derivada $\dot{x}$ con respecto al tiempo del vector de estados *x* es igual a una función del vector de estados *x* y del tiempo. Usando un método de solución iterativo numérico para resolver este sistema de ecuaciones diferenciales ordinarias, en cada paso de iteración, se calcula la derivada del vector de estados a partir del valor del vector de estados en ese instante de tiempo.

Para el modelo del circuito simple, se obtiene lo siguiente:

$x$ = {C.v,L.i}, $u$ = {AC.v} (con constantes: R1.R,R2.R,C.C,L.L,
$\dot{x}$ = {**der**(C.v), **der**(L.i)} AC.VA,AC.f,AC.PI) (2-4)

### 2.20.3 Método de Solución

Emplearemos un método de solución numérico iterativo. Primero, se asume que se tiene un valor estimado del vector de estados $x$ ={C.v,L.i} en el instante inicial de la simulación, t=0. Se emplea una aproximación numérica para la derivada $\dot{x}$ (es decir, der(*x*)) en el instante de tiempo t, por ejemplo:

**der**(*x*) = (*x*(t+h) - *x*(t))/h (2-5)

obteniéndose de la expresión anterior una aproximación para el valor de *x* en el instante t+h:

*x*(t+h) = *x*(t) + **der**(*x*)*h (2-6)

De esta forma, en cada instante t se calcula el valor del vector de estado para el instante siguiente, t+h, suponiendo que sea posible calcular der(x) en el instante t en tiempo de simulación. Ahora bien, la derivada der(x) del vector de estados puede calcularse a partir de $\dot{x} = f(x,t)$. Es decir, seleccionando las ecuaciones en las que interviene der(x), y despejando las derivadas de las variables del vector *x* en función de otras variables, como se muestra a continuación:

**der**(C.v) = C.i/C.C (2-7)

```
der(L.i) = L.v/L.L
```

Es necesario emplear otras ecuaciones del sistema DAE para calcular las variables desconocidas `C.i` y `L.v`, que intervienen en las ecuaciones anteriores. Empezando con C.i, mediante manipulaciones algebraicas y sustituciones se derivan las ecuaciones (2-8) hasta (2-10).

```
C.i = R1.v/R1.R
usando: C.i = C.p.i = -R1.n.i = R1.p.i = R1.i
          = R1.v/R1.R
```
(2-8)

```
R1.v = R1.p.v - R1.n.v = R1.p.v – C.v
usando: R1.n.v = C.p.v = C.v+C.n.v
            = C.v + AC.n.v
            = C.v + G.p.v = C.v + 0 = C.v
```
(2-9)

```
R1.p.v  = AC.p.v = AC.VA*sin(2*AC.f*AC.PI*t)
usando: AC.p.v = AC.v+AC.n.v = AC.v+G.p.v =
           = AC.VA*sin(2*AC.f*AC.PI*t)+0
```
(2-10)

De manera análoga, se obtienen las ecuaciones (2-11) y (2-12) mostradas a continuación:

```
L.v  = L.p.v - L.n.v = R1.p.v - R2.v
usando: L.p.v = R2.n.v = R1.p.v-R2.v
y: L.n.v = C.n.v = AC.n.v = G.p.v = 0
```
(2-11)

```
R2.v = R2.R*L.p.i
usando: R2.v = R2.R*R2.i = R2.R*R2.p.i
          = R2.R*(-R2.n.i) = R2.R*L.p.i
          = R2.R*L.i
```
(2-12)

Reuniendo las cinco ecuaciones:

```
C.i     = R1.v/R1.R
R1.v    = R1.p.v - C.v
R1.p.v  = AC.VA*sin(2*AC.f*AC.PI*time)
L.v     = R1.p.v - R2.v
R2.v    = R2.R*L.i
```
(2-13)

Escribiendo las ecuaciones en ordende dependencia de datos, y convirtiéndo las eecuaciones en sentencias de asignación, estos es possible ya que todos los valores de la variables se pueden

calcular ahora en orden, se obtiene el conjunto de asignaciones mostrado a continuación. Estas asignaciones deben ser evaluadas en cada iteración, con lo cual se obtienen los valores de `C.v`, `L.i`, y `t` en la misma iteración.

```
R2.v        := R2.R*L.i
R1.p.v      := AC.VA*sin(2*AC.f*AC.PI*time)
L.v         := R1.p.v - R2.v
R1.v        := R1.p.v - C.v
C.i         := R1.v/R1.R
der(L.i)    := L.v/L.L
der(C.v)    := C.i/C.C
```

Estas sentencias de asignación pueden ser convertidas a código escrito en algún lenguaje de programación, por ejemplo C, y ejecutadas junto con un solucionador apropiado de ODE. Normalmente, se emplean mejores aproximaciones para las derivadas y estrategias más sofisticadas para la elección del tamaño del paso de integración que las empleadas por el *método de integración de Euler* que se ha aplicado en el ejemplo anterior. Las transformaciones algebraicas y el procedimiento de ordenación, realizadas fastidiosamente a mano en el ejemplo del circuito sencillo, pueden ser ejecutadas de manera automática. Este procesos de transformación es conocido como *transformación BLT*. Dicha transformación consiste en transformar la matriz de coeficientesdel sistema de ecuaciones en una forma triangular inferior por bloques (véase la Figura 2-30).

| | | R2.v | R1.p.v | L.v | R1.v | C.i | L.i | C.v |
|---|---|---|---|---|---|---|---|---|
| R2.v = R2.R*L.i | | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| R1.p.v = AC.VA*sin(2*AC.f*AC.PI*time) | = | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| L.v = R1.p.v - R2.v | | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| R1.v = R1.p.v - C.v | | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| C.i = R1.v/R1.R | | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **der**(L.i) = L.v/L.L | | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **der**(C.v) = C.i/C.C | | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

**Figura 2-30.** Matriz estructural BLT del ejemplo `SimpleCircuito`.

Las restantes 26 variables algebraicas que aparecen en el sistema de ecuaciones del modelo del circuito, y que no son parte del kernel minimal de 7 variables del sistema ODE resuelto arriba,

pueden calcularse a voluntad en aquellos instantes de tiempo en que se precise conocer su valor, ya que no son necesarias para resolver el kernel del sistema ODE.

Es necesario recalcar que el ejemplo del circuito eléctrico sencillo simulado en la Figura 2-31 es trivial. Los modelos de simulación realistas a menudo contienen decenas de cientos de ecuaciones, ecuaciones no lineales, modelos híbridos, etc. Las transformaciones simbólicas y las reducciones de sistemas de ecuaciones realizadas por un compilador de Modelica real son mucho más complicadas de lo que se ha mostrado en este ejemplo. Por ejemplo, incluyen la reducción del índice de las ecuaciones y el desgloce de los subsistemas de ecuaciones, ver Fritzon (2004; 2014). Para la reducción de índice, se realiza una diferenciación simbólica del sistema de ecuaciones con el objetivo de transformarlo a la forma de una ecuación algebráico diferencial de índice 1 (una ecuación diferencia ordinaria) que es numéricamente mas estable para ser resuelta con la mayoría de solucionadores.

```
simulate(CircuitoSimple,stopTime=5)]
plot(C.v, xrange={0,5})
```

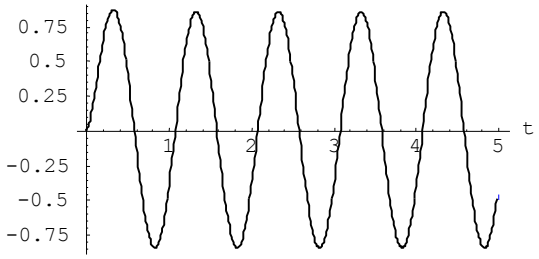



**Figura 2-31.** Simulación del modelo `CircuitoSimple` con un gráfico de la evolución del voltaje `C.v` entre los terminales del condensador.

## 2.21 Historia

En septiembre de 1996, un grupo de diseñadores de herramientas, expertos en aplicaciones y científicos de la computación unieron sus fuerzas para trabajar juntos en el área de la tecnología y las aplicaciones del modelado orientado a objetos. El grupo incluía a los desarrolladores de los lenguajes de modelado orientados a objetos Dymola, Omola, ObjectMath, NMF (Neutral Modeling Format), Allan-U.M.L, SIDOPS+ y Smile, si bien no todos ellos pudieron acudir a la primera reunión. El objetivo inicial fue escribir un libro blanco acerca de la tecnología de los lenguajes orientados a objetos, incluyendo una posible unificación de los lenguajes de modelado

existentes. Esta tarea fue parte de una acción en el proyecto ESPRIT Simulation in Europe Basic Research Working Group (SiE-WG).

Sin embargo, el trabajo pronto se centró en un objetivo más ambicioso: la creación de un lenguaje de modelado orientado a objetos, nuevo y unificador, basado en la experiencia adquirida en los diseños precedentes. Los diseñadores realizaron un esfuerzo de unificación de los conceptos para crear un lenguaje común, iniciando el diseño desde cero. Este nuevo lenguaje se llamó *Modelica*.

Pronto, el grupo se establecido como un comité técnico (Technical Committee 1) dentro de EuroSim, y como el Technical Chapter on Modelica dentro de la Society for Computer Simulation International (SCS). En febrero de 2000, se constituyó la Modelica Association como una organización internacional independiente, sin ánimo de lucro, cuyo objetivo es desarrollar y promover el desarrollo y difusión del lenguaje Modelica y de las Librerías Estándar de Modelica.

La primera descripción del lenguaje Modelica, que es la versión 1.0, fue puesta en la web en septiembre de 1997, por un grupo de ingenieros orgullosos con el trabajo realizado, tras un cierto número de reuniones de diseño intensas. En 2011, que es la fecha en que se escribe este documento, está disponible la especificación de Modelica 3.2. Ésta es el resultado de una gran cantidad de trabajo, incluyendo 34 reuniones de diseño, de tres días de duración cada una. Actualmente existen siete herramientas comerciales bastante completes que soportan el diseño textual y gráfico de modelos con Modelica, así como una implementación de código abierto bastante completa, y varios prototipos parciales desarrollados en universidades. También está disponible la amplia y creciente librería Estándar de Modelica. El lenguaje está difundiéndose rápidamente tanto en el ámbito industrial como en el académico.

Mirando con cierta perspectiva la tecnología de Modelica, uno se da cuenta de dos hechos relevantes:

- Modelica incluye *ecuaciones*, lo cual es inusual en la mayoría de los lenguajes de programación.
- La tecnología de Modelica soporta la edición *gráfica* para el diseño de modelos, basada en componentes predefinidos.

De hecho, respecto al primer punto, las ecuaciones fueron usadas muy pronto en la historia humana—ya en el tercer milenio A.C. En aquel momento, el bien conocido signo de igualdad empleado en las ecuaciones (es decir, "=") no había sido todavía inventado. Esto ocurrió mucho más tarde: el signo de la ecuación fue introducido por Robert Recorde en 1557, en la forma mostrada en la Figura 2-32.

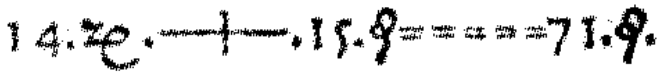
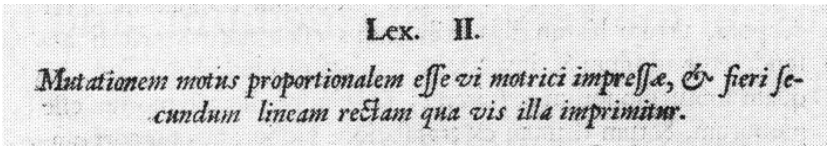


**Figura 2-32.** Signo de ecuación inventado por Robert Recorde en 1557. Reproducido a partir de la Figura 4.1-1 de la página 81 en Gottwald et al. (1989), cortesía de Thompson Inc.

Sin embargo, esta invención tardó cierto tiempo en difundirse por Europa. Incluso cien años más tarde, Newton (en su *Principia*, vol. 1, 1686) aun escribió su famosa ley del movimiento como un texto en Latín, como se muestra en la Figure 2-33. Traducido al español, esta ley puede expresarse como: "El cambio en el movimiento es proporcional a la fuerza ejercida que lo motiva".



Lex. II.

*Mutationem motus proportionalem esse vi motrici impressæ, & fieri secundum lineam rectam qua vis illa imprimitur.*

**Figure 2-33.** Famosa segunda ley del movimiento de Newton escrita en Latín. Traducida al español, dice "El cambio en el movimiento es proporcional a la fuerza ejercida que lo motiva". Reproducida de la Figura "Newton's laws of motion" de la página 51 en Fauvel et al. (1990), cortesía de Oxford University Press.

En la sintaxis matemática moderna, la ley de Newton del movimiento se escribe como sigue:

$$\frac{d}{dt}(m \cdot v) = \sum_i F_i \tag{2-14}$$

Este es un ejemplo de una ecuación diferencial. Los primeros simuladores capaces de resolver estas ecuaciones eran analógicos. La idea era modelar el sistema en términos de ecuaciones diferenciales ordinarias y entonces construir un dispositivo físico que se comportara de acuerdo a estas ecuaciones. Los primeros simuladores analógicos eran dispositivos mecánicos, si bien a partir de la década de 1950 predominaron los simuladores analógicos electrónicos. Estos estaban compuestos por bloques electrónicos sumadores, multiplicadores, integradores, etc. que eran conectados mediante cables, tal como se muestra en la Figura 2-34.

Acerca de los desarrollos posteriores, durante largo tiempo el soporte a ecuaciones fue bastante raro en los lenguajes de computación. Se desarrollaron las primeras versiones de los sistemas Lisp y de los sistemas algebraicos computacionales. Sin embargo, estaban fundamentalmente orientados a la manipulación de fórmulas, en lugar de estar orientados directamente a la simulación.

No obstante, pronto comenzaron a aparecer cierto número de lenguajes de simulación para los computadores digitales. La primera herramienta de modelado basada en ecuaciones fue Speed-Up, un paquete para ingeniería química que apareció en 1964. Algo más tarde, en 1967, apareció Simula 67—el primer lenguaje de programación orientado a objetos, que tuvo una profunda influencia en los lenguajes de programación y algo más tarde en los lenguajes de modelado. Ese mismo año, el informe CSSL (Continuous System Simulation Language) unificó la sintaxis existente para la descripción de los modelos de simulación de sistemas continuos. También, introdujo un formato común para describir las "ecuaciones" causales. A continuación se muestra un ejemplo:

$$\begin{aligned} &\textit{variable} = \textit{expression} \\ &v = \mathrm{INTEG}(F) / m \end{aligned} \tag{2-15}$$

La segunda ecuación es una variante de la ecuación del movimiento: la velocidad es igual a la integral de la fuerza dividida por la masa. Estas ecuaciones no son ecuaciones generales en el sentido matemático, puesto que la causalidad es de derecha a izquierda. Es decir, no está permitida una ecuación de la forma *expresión = variable*. Pese a esta limitación, esto supuso un paso adelante muy importante hacia una representación más general de los modelos matemáticos basados en ecuaciones, cuya finalidad era ser ejecutados en un computador. ACSL apareció en 1976 y fue un sistema de simulación bastante común, que inicialmente se basó en el estándar CSSL.

Un lenguaje de modelado pionero, e importante antecesor de Modelica, fue Dymola (*Dynamic modeling language*—no la herramienta Dymola actual, que significa Dynamic modeling laboratory). El lenguaje Dymola fue descrito en la Tesis Doctoral de Hilding Elmqvist en 1978. Este fue el primer trabajo en el cual se reconoció la importancia de realizar los modelos usando ecuaciones no causales y empleando submodelos estructurados jerárquicamente, así como de disponer de métodos para la manipulación simbólica automática que soporten la resolución de las ecuaciones. El sistema GASP-IV en 1974, seguido por GASP-V en 1979, introdujo la simulación integrada de modelos con parte continua y parte discreta. El lenguaje Omola (1989) es un lenguaje de modelado completamente orientado a objetos que incluye herencia, así como simulación híbrida. El lenguaje Dymola fue posteriormente (1993) ampliado para soportar la herencia, así como con mecanismos para el manejo de eventos discretos y métodos más eficientes para la solución de sistemas de ecuaciones simbólicas y numéricas.

Otros lenguajes pioneros en el modelado no causal orientado a objetos son NMF (Natural Model Format, 1989), usado inicialmente para simulació Allan-U.M.L, SIDOPS+ que soporta el modelado mediante grafos de enlace (bond graph) y Smile (1995)—que está influenciado por

Objective C. Otros dos lenguajes importantes que merecen ser citados son ASCEND (1991) y gPROMS (1994).

La experiencia del autor de este libro con el modelado y la solución de problemas basados en ecuaciones comenzó en 1975, resolviendo la ecuación de Schrödinger aplicada a un problema muy específico de la física del estado sólido, usando la aproximación del seudo potencial. Más tarde, en 1989, inicié junto con mi hermano Dag Fritzson el desarrollo de un nuevo lenguaje de modelado orientado a objetos, llamado ObjectMath. Éste fue uno de los primeros sistemas de simulación y algebra computacional orientado a objetos. Estaba integrado con Mathematica y soportaba el concepto de clase genérica parametrizable, así como la generación de código C++ para realizar eficientemente la simulación de aplicaciones industriales. La cuarta versión de ObjectMath fue completada en otoño de 1996, momento en el que decidí unirme al esfuerzo de Modelica en lugar de continuar con la quinta versión de ObjectMath. Mas tarde, en 1998, hicimos la rpimera especificación ejecutable formal de parte del lenguaje Modelica el cual eventualmente se convirtió en el esfuerzo de código abierto OpenModelica. En diciembre de 2007, Yo inicié la creación del Open Source Modelica Consortium con siete (7) miembros inicialmente, los cuáles se han expandido a mas de 35 miembro a junio de 2011.

La Figura 2-34 cuenta una historia interesante respecto al segundo aspecto mencionado anteriormente: la especificación gráfica de los modelos de simulación.

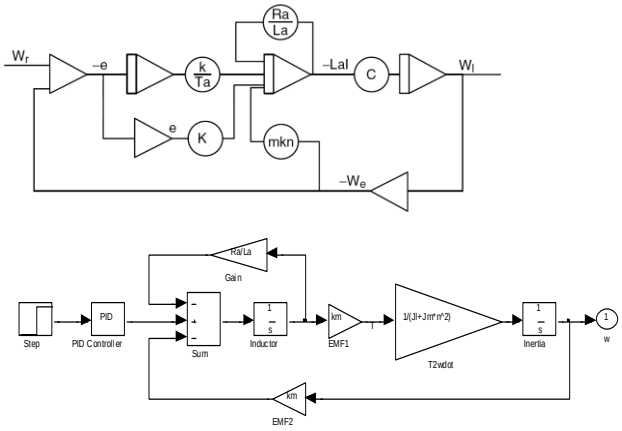



**Figura 2-34.** Computación analógica vs. modelado gráfico basado en diagramas de bloques realizado en los computadores digitales modernos. Cortesía de Karl-Johan Åström y Hilding Elmqvist.

La parte superior de la Figura 2-34 muestra la circuitería de un computador análogo con sus bloques constitutivos conectados mediante cables. La parte de abajo de la figura es un diagrama de bloques con una estructura muy similar, directamente inspirado en el paradigma de la computación analógica, pero ejecutado en un computador digital. Tales diagramas de bloques se construyen típicamente empleando herramientas actuales como Simulink o SystemBuild. Los diagramas de bloques representan ecuaciones causales, puesto que hay una dirección del flujo de los datos específicada.

Los diagramas de conexiones para el modelado gráfico usando Modelica incluyen conexiones entre las instancias de las clases, que contienen ecuaciones no causales, tal como se exploró primeramente en el sistema de Hibliz. Esta es una generalización inspirada en los diagramas de los circuitos empleados en la computación analógica causal y en los diagramas de bloques. Los diagramas de conexión de Modelica tienen la ventaja de que soportan el modelado físico de manera natural, puesto que la topología del diagrama de conexiones se corresponde directamente con la estructura y descomposición del sistema físico modelado.

## 2.22 Resumen

Este capítulo ha proporcionado una rápida visión general de los conceptos y construcciones más importantes del lenguaje Modelica. También, se han definido conceptos importantes tales como el modelado matemático orientado a objetos y el modelado físico no causal, y se han presentando brevemente los conceptos y constructos del lenguaje Modelica para definir componentes, conexiones y conectores. El capítulo finaliza con un ejemplo detallado de la traducción y ejecución de un modelo sencillo, y con una breve historia, desde los tiempos antiguos hasta la actualidad, de las ecuaciones y de los lenguajes para el modelado matemático, incluyendo Modelica.

## 2.23 Literatura

Muchos libros sobre lenguajes de programación están organizados de acuerdo a un patrón bastante bien establecido. En primer lugar, presentan una visión general rápida del lenguaje, seguida por una presentación más detallada acerca de los conceptos más importantes del lenguaje y sus estructuras sintácticas. Este libro no constituye una excepción a esta regla: este capítulo proporciona una primera visión general. Como en muchos otros textos, comenzamos con un ejemplo `HolaMundo`, como en el libro (Arnold y Gosling 1999) acerca del lenguaje de

programación Java, pero con contenidos diferentes, ya que imprimir el mensaje "Hola mundo" no es demasiado relevante para un lenguaje basado en ecuaciones.

El documento de referencia más importante para este capítulo es el tutorial de Modelica (Modelica Association 2000), cuya primera versión incluyó una lógica de diseño (Modelica Association 1997) fue editada en su mayor parte por Hilding Elmqvist. Varios ejemplos, fragmentos de código y de texto de este capítulo están basados en otros ejemplos similares incluidos en dicho tutorial. Por ejemplo, el modelo `CircuitoSimple` con los componentes eléctricos sencillos, el modelo `EvaluadorPolinomio`, el filtro pasa bajo, el `Diodo` ideal, el modelo de la pelota que rebota `RebotePelota`, etc. La Figure 2-8, referente al modelado orientado a bloques, está también extraída del tutorial. Otro importante documento de referencia para este capítulo es la especificación del lenguaje Modelica (Modelica Association 2010). En este capítulo se reusan algunas formulaciones de la especificación Modelica concernientes a la sobrecarga de operadores y a los conectores tipo `stream` con el objetivo de establecer las mismas semánticas.

La traducción manual del modelo del circuito simple está inspirado en uno similar, pero menos elaborado, contenido en una serie de artículos de Martin Otter (1999). La historia reciente de los lenguajes para el modelado matemático es descrita con cierto detalle en Åström et al. (1998), mientras que en Gottwald et al. (1989) pueden encontrarse algunas partes de la historia de la invención y uso de las ecuaciones. La fotografía de la segunda ley de Newton escrita en Latín puede encontrarse en Fauvel et al. (1990). Los primeros trabajos acerca de la combinación de la simulación continua y discreta (GASP-IV) son descritos en Pritsker (1974), seguido por Cellier (1979) en el sistema GASP-V. El primer trabajo de simulación del autor de este libro, centrado en resolver una aplicación de la ecuación de Schrödinger a un caso particular, es descrito en Fritzson y Berggren (1976).

Los predecesores del lenguaje Modelica incluyen: Dymola entendido como Dynamic Modeling Language (Elmqvist 1978; Elmqvist et al. 1996); Omola (Mattsson et al. 1993; Andersson 1994); ObjectMath: (Fritzson et al. 1992, 1995; Viklund y Fritzson 1995); NMF: (Sahlin et al. 1996), y Smile: (Ernst et al. 1997).

Speed-Up, que fue la primera herramienta de simulación basada en ecuaciones, es presentada en Sargent y Westerberg (1964), mientras que Simula-67—el primer lenguaje de programación orientado a objetos—es descrito en Birtwistle et al. (1974). La primera especificación del lenguaje CSSL es descrita en Augustin et al. (1967), mientras que el sistema ACSL es descrito en Mitchell y Gauthier (1986). El sistema Hibliz para un acercamiento gráfico jerárquico del modelado es presentado en Elmqvist y Mattsson (1982 y 1989).

Los sistemas de componentes software son presentados en Assmann (2002) y en Szyperski (1997).

El sistema Simulink para el modelado orientado a bloques es descrito en MathWorks (2001), mientras que el lenguaje y la herramienta MATLAB son descritos en MathWorks (2002).

El cuaderno electrónico DrModelica, con los ejemplos y ejercicios de este libro, está inspirado en DrScheme (Felleisen et al. 1998) y DrJava (Allen et al. 2002), así como por Mathematica (Wolfram 1997), un libro electrónico para la enseñanza de las matemáticas (Davis et al. 1994), y el entorno MathModelica (Fritzson, Engelson y Gunnarsson 1998; Fritzson, Gunnarsson y Jirstrand 2002). La primera versión de DrModelica es descrita en Lengquist-Sandelin y Monemar (2003a, 2003b).

Algunos artículos en general sobre Modelica y libros son: (Elmqvist and Mattsson 1997; Fritzson y Engelson 1998; Elmqvist et al. 1999), una serie de 17 artículos (en alemán) de los cuales (Otter 1999) es el primero, (Tiller 2001; Fritzson y Bunus 2002; Elmqvist et al 2002; Fritzson 2004; Fritzson 2014).

Las memorias de las siguientes conferencias, también como algunos que no se listan aquí, contienen un buen número de artículos relacionados con Modelica: la Scandinavian Simulation Conference (Fritzson 1999) y especialmente de la International Modelica Conference (Fritzson 2000; Otter 2002; Fritzson 2003, Schmitz 2005, Kral and Haumer 2006; Bachmann 2008; Casella 2009; Clauβ 2001) .

## 2.24 Ejercicios

**Ejercicio 2-1**:

*Pregunta*: ¿Qué es una clase?

*Creación de una clase*: Cree una clase, `Suma`, que calcule la suma de dos parámetros enteros (`Integer`) de valores dados.

**Ejercicio 2-2**:

*Pregunta*: ¿Qué es una instancia?

*Creación de instancias:*

```
class Perro
  constant  Real   patas  = 4;
  parameter String nombre = "Dummy";
end Perro;
```

- Cree una instancia de la clase `Perro`.

- Cree otra instancia y asigne al perro el nombre "Tim".

**Ejercicio 2-3**:

Escriba una función, `Promedio`, que devuelva el valor medio de dos valores reales (`Real`).

Llame a la función `Promedio` con los valores 4 y 6.

**Ejercicio 2-4**:

*Pregunta*: ¿Qué significado tienen los términos `partial`, `class` y `extends`?

**Ejercicio 2-5**:

*Herencia:* Considere la clase `Bicicleta` mostrada a continuación.

```
record Bicicleta
  Boolean tiene_ruedas = true;
  Integer numDeRuedas  = 2;
end Bicicleta;
```

Defina un record, `BicicletaInfantil`, que herede de la clase `Bicicleta` y represente una bicicleta para niños. Asigne valores a las variables.

**Ejercicio 2-6**:

*Ecuaciones de Declaración y Ecuaciones Normales:* Escriba la clase, `AgnoNacimiento`, que calcula el año de nacimiento a partir de dos datos: el año actual y la edad de la persona.

Señale las ecuaciones de declaración y las ecuaciones normales.

*Ecuación de Modificación:* Escriba una instancia de la clase `AgnoNacimiento` anterior. La clase, llamada `AgnoNacimientoDeMartin`, calculará el año de nacimiento de Martín, sabiendo que tiene 29 años. Señale la ecuación de modificación.

Compruebe su respuesta, por ejemplo, escribiendo lo siguiente[10]

```
val(AgnoNacimientoDeMartin.AgnoNacimiento,0)
```

[10] Usando la interface de línea de comando de OpenModelica o los comandos OMNotebook, la expresión `val(AgnoNacimientoDeMartin.AgnoNacimiento,0)` significa el valor de AgnoNacimiento en el `time = 0`, al inicio de la simulación. También, es posible en muchos casos introducir interactivamente una expresión como `AgnoNacimientoDeMartin.AgnoNacimiento` y obtener el resultado sin necesidad de especificar el valor del argumento tiempo.

**Ejercicio 2-7**:

*Clases:*

```
class Ptest
  parameter Real x;
  parameter Real y;
  Real z;
  Real w;
equation
  x + y = z;
end Ptest;
```

*Pregunta*: ¿Qué está mal en esta clase? ¿Falta algo?

**Ejercicio 2-8**:

Cree un record que contenga varios vectores y matrices:

- Un vector que contenga dos valores `Boolean`: `true` y `false`.
- Un vector con cinco valores `Integer` de su elección.
- Una matriz con tres filas y cuatro columnas que contenga valores `String` de su elección.
- Una matriz de una fila y cinco columnas que contenga diferentes `Real` de su elección.

**Ejercicio 2-9**:

*Pregunta*: ¿Puede realmente escribirse una sección algorithm dentro de una sección equation?

**Ejercicio 2-10**:

*Escritura de una Sección Algorithm:* Cree la clase que calcula el valor medio de dos números enteros, `Promedio`, usando una sección algorithm.

Cree una instancia de la clase y asígnele varios valores.

Simule y entonces compruebe el resultado de la clase instanciada escribiendo `variableDeInstancia.variableDeClase`.

**Ejercicio 2-11**: (Un ejercicio un poco más difícil)

Escriba una clase, `PromedioAmpliada`, que calcule el promedio de 4 variables (`a, b, c` y `d`).

Cree una instancia de la clase y asígnele algunos valores.

Simule y entonces compruebe el resultado de la clase instanciada escribiendo `variableDeInstancia.variableDeClase`

**Ejercicio 2-12**:

*Ecuación If:* Escriba la clase `Luces`, que asigna a la variable interruptor (de tipo integer) el valor uno si las luces están encendidas y cero si están apagadas.

*Ecuación When:* Escriba la clase `ConmutadorDeLaLuz`, que describe un conmutador que inicialmente está apagado y que es encendido en el instante de tiempo igual a 5.

*Ayuda:* `sample(start, interval)` devuelve true y dispara eventos en determinados instantes, y `rem(x, y)` devuelve el resto de la división entera `x/y`, de modo que `div(x,y) * y + rem(x, y) = x`.

**Ejercicio 2-13**:

*Pregunta*: ¿Qué es un paquete?

*Creación de un paquete*: Cree un paquete que contenga una función, que realice la división de dos números reales (`Real`), y que contenga una constante `k = 5`.

Cree una clase que contenga la variable `x`. Dicha variable obtiene su valor a partir de una llamada a la función división, contenida en el paquete, para dividir 10 por 5.

# Capítulo 3

# Clases y Herencia

En Modelica, la unidad fundamental para el modelado es la clase. Las clases proporcionan la estructura para los objetos, también llamados instancias, y sirven de plantilla para su crear objetos a partir d las definiciones de las clases. Las clases pueden contener ecuaciones, que son la base del código ejecutable usado para los cálculos en Modelica. Código algorítmico convencional puede ser también parte de las clases. La interacción entre los objetos de clases bien estructuradas en Modelica se produce normalmente a través de los conectores, que pueden ser considerados como los puertos de acceso a los objetos. Todos los objetos de datos en Modelica son instanciados a partir de clases, incluyendo los tipos básicos de datos—`Real`, `Integer`, `String`, `Boolean`—y los tipos enumerados, que son clases predefinidas o esquemas de clases.

En Modelica, una clase es esencialmente equivalente a un tipo. Las declaraciones son los constructos sintácticos necesarios para introducir las clases y los objetos.

## 3.1 Contrato entre el Diseñador y el Usuario de la Clase

Los lenguajes de modelado orientado a objetos tratan de separar *qué* es un objeto, de los detalles relativos a *cómo* es implementado y especificado su comportamiento en detalle. Normalmente, el "qué" de un objeto en Modelica es descrito a través de su documentación, que incluye gráficos e iconos, y a través de los posibles conectores públicos, variables, y otros elementos públicos, y su semántica asociada. Por ejemplo, el qué de un objeto de la clase `Resistencia` es la

documentación que indica que modela una resistencia ideal "realista", también el hecho de que la interacción con el mundo exterior se produce a través de dos conectores `n`, `p` del tipo `Pin`, y también su semántica. Esta combinación—documentación, conectores y otros elementos públicos, y la semántica—se describe a menudo como un *contrato* entre el diseñador de la clase y el modelador que la emplea para realizar sus modelos. La parte "qué" del contrato especifica al usuario qué representa la clase, mientras que el "cómo" proporcionado por el diseñador de la clase implementa las propiedades y el comportamiento requeridos.

Una suposición que se realiza comúnmente, pero que es incorrecta, es que los conectores y los otros elementos públicos de la clase (su "firma") especifican por completo el contrato. Esto es incorrecto, ya que la semántica de la clase es también parte del contrato, aún anque solo pueda ser descrita públicamente en la documentación y modelada internamente mediante ecuaciones y algoritmos. Por ejemplo, dos clases, un `Resistor` y un `Resistor` dependiente de la temperatura, pueden tener la misma firma en términos de sus conectores, pero aun así no ser equivalentes puesto que tienen diferente semántica. El contrato de uma clase incluye en conjunto tanto su firma, como la parte apropiada de su semántica.

El cómo de un objeto es definido por su clase. La implementación del comportamiento de la clase se define en términos de ecuaciones y también posiblemente a través de código algorítmico. Cada objeto es una instancia de una clase. Muchos objetos son compuestos, es decir, están compuestos por instancias de otras clases.

## 3.2 Ejemplo de una Clase

Las propiedades básicas de una clase están determinadas por:

- Los datos contenidos en las variables declaradas en la clase.
- El comportamiento especificado mediante ecuaciones y también posiblemente mediante algoritmos.

A continuación se muestra la clase `CuerpoCeleste`. Puede ser usada para almacenar datos relacionados con los cuerpos celestes, tales como la tierra, la luna, los asteroides, los planetas, los cometas y las estrellas:

```
class CuerpoCeleste
  constant  Real   g = 6.672e-11;
  parameter Real   radio;
  parameter String nombre;
  Real             masa;
```

```
end CuerpoCeleste;
```

La declaración de una clase comienza con una palabra clave tal como class o model, seguida por el nombre de la clase. Una declaración de una clase crea un tipo de nombre en Modelica. Esto posibilita crear variables de ese tipo, también conocidas como objetos o instancias de la clase, simplemente anteponiendo el nombre del tipo al nombre de la variable:

```
CuerpoCeleste luna;
```

Esta declaración establece que `luna` es una variable que contiene un objeto del tipo `CuerpoCeleste`. La declaración crea el objeto, es decir, reserva la cantidad de memoria requerida para almacenarlo. Esto no es lo que ocurre en un lenguaje como Java, donde la declaración de un objeto únicamente crea una referencia al objeto.

La primera versión de `CuerpoCeleste` no está demasiado bien diseñada. Esto es intencionado. A lo largo de este y de los dos siguientes capítulos demostraremos la utilidad de ciertas capacidades del lenguaje, aplicándolas para mejorar esta clase.

## 3.3 Variables

Las variables de una clase también se denominan campos de registro o atributos. Las variables `radio`, `nombre` y `masa` de `CuerpoCeleste` son algunos ejemplos. Cada objeto del tipo `CuerpoCeleste` tiene sus propias instancias de estas variables. Puesto que cada objeto contiene una instancia diferente de las variables, esto significa que cada objeto tiene su propio estado único. Cambiar el valor de la variable `masa` en un objeto `CuerpoCeleste` no afecta a las variables `masa` de otros objetos `CuerpoCeleste`.

Algunos lenguajes de programación, por ejemplo Java y C++, permiten definir variables estáticas, también llamadas variables de clase. Estas variables son compartidas por todas las instancias de la clase. Sin embargo, Modelica no dispone de este tipo de variable.

La declaración de una instancia de una clase, por ejemplo de `luna`, que es una instancia de `CuerpoCeleste`, reserva memoria para el objeto e inicializa sus variables a los valores apropiados. Tres de las variables de la clase `CuerpoCeleste` tienen un status especial. La *constante* gravitatoria `g` es una constante que no cambia (`constant`) y por ello puede ser sustituida por su valor. Los parámetros de la simulación `radio` y `nombre` son ejemplos de un tipo especial de “constante”, que es definida en Modelica mediante la palabra clave `parameter`. Se asignan valores al inicio de la simulación a estas constantes parámetro de simulación y éstos se mantienen constantes durante toda la simulación.

En Modelica, las variables almacenan los resultados de los cálculos realizados al resolver las ecuaciones de una clase junto con las ecuaciones de las otras clases. Durante la solución de problemas dependientes del tiempo, las variables almacenan los resultados del proceso de solución en el instante actual de tiempo.

El lector se habrá dado cuenta de que usamos de manera intercambiable, es decir con el mismo significado, los términos *objeto* e *instancia*. También, usamos los términos *campo de registro*, *atributo* y *variable* de manera intercambiable. En algunas ocasiones, el *término* variable se usa intercambiablemente con *instancia* u *objeto*, puesto que una variable en Modelica siempre contiene una instancia de alguna clase.

### 3.3.1 Nombres de Variables Duplicados

En las declaraciones de las clases no se permite tener nombres duplicados. El nombre de cada elemento declarado, por ejemplo una variable o una clase local, debe ser diferente de los nombres de los restantes elementos declarados en la clase. Por ejemplo, la clase siguiente es ilegal:

```
class DuplicacionIlegal
  Real    duplicado;
  Integer duplicado;    // Error! Nombre de variable duplicado
end DuplicacionIlegal;
```

### 3.3.2 Nombres de Variables Idénticos y Nombres de Tipos

El nombre de una variable no puede ser idéntico al nombre de su tipo. Considere la siguiente clase errónea:

```
class IlegalTipoComoVariable
  Voltage Voltage;   // Error! El nombre de la variable debe ser
                     // diferente del nombre de su tipo
  Voltage voltage;   // Bien! Voltage y voltage son nombres
                     // diferentes
end IlegalTipoComoVariable;
```

La primera declaración de variable es ilegal, puesto que el nombre de la variable es idéntico al nombre del tipo de la declaración. El motivo por el cual esto constituye un problema es que cuando se busque el tipo `Voltage` desde la segunda declaración, éste quedará oculto por la variable con el mismo nombre. La segunda declaración de variable es legal, puesto que el nombre en minúsculas `voltage` es diferente del nombre de tipo en mayúsculas `Voltage`.

### 3.3.3 Inicialización de Variables

En caso de que no se especifiquen explícitamente los valores iniciales para las variables, los valores iniciales por defecto sugeridos (el solucionador puede escoger otros, si no se asignan usando la plabra clave `fixed`) son los siguientes (vea la Sección 2.3.2):

- El valor cero es el valor inicial por defecto para las variables numéricas.
- La cadena de caracteres vacía `""` para las variables String.
- El valor `false` para las variables tipo `Boolean`.
- Para las variables enumeradas, el valor inferior de la enumeración en su tipo enumerado.

Sin embargo, en caso de que no se les asignen explícitamente valores por defecto, las *variables locales* de las funciones tienen valores iniciales *sin especificar*. Los valores iniciales pueden ser indicados explícitamente asignando valor a los atributos `start` de las variables de instancia, o proporcionando asignaciones de inicialización cuando las instancias son variables locales o parámetros formales de funciones. Por ejemplo, en la clase `Cohete` que se mostrará en la siguiente sección, se asignan explícitamente valores iniciales a las variables `masa`, `altitud` y `velocidad`.

## 3.4 Comportamiento como Ecuaciones

Si bien se dispone de algoritmos y de funciones, la forma más básica de especificar el comportamiento de una clase en Modelica es mediante ecuaciones. La forma en que las ecuaciones interactúan con las ecuaciones de las otras clases determina el proceso de cálculo de la solución. Es decir, la ejecución del programa en el cual se calculan los sucesivos valores que van tomando las variables en el tiempo. Esto es exactamente lo que sucede durante la simulación de un sistema dinámico. Durante la solución de los problemas dependientes del tiempo, las variables almacenan los resultados del proceso de solución en el instante actual de tiempo.

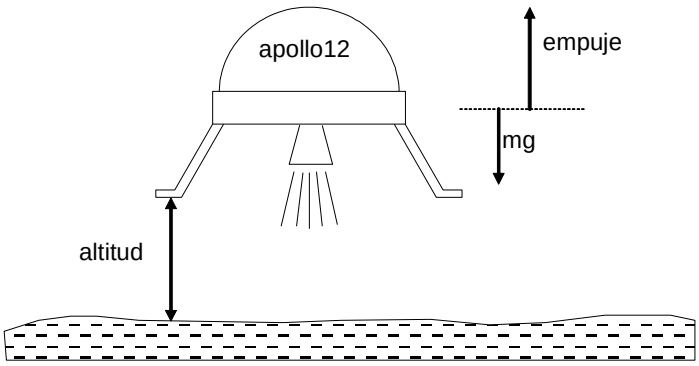



**Figura 3-1.** Cohete Apollo12 aterrizando en la luna.

La clase `Cohete` contiene las ecuaciones que describen el movimiento vertical para un cohete (por ejemplo, como se muestra en la Figura 3-1). Dicho movimiento está influido por un campo gravitatorio externo `gravedad` y por la fuerza `empuje`, producida por el motor del cohete, que actúa en sentido opuesto a la fuerza gravitatoria. Esto se muestra en la siguiente expresión para la aceleración:

$$aceleracion = \frac{empuje - masa \cdot gravedad}{masa}$$

Las tres ecuaciones siguientes son ecuaciones diferenciales de primer orden. Describen leyes del movimiento bien conocidas, que relacionan la altitud, la velocidad vertical y la aceleración:

$$masa' = -coefPerdidaMasa \cdot abs(empuje)$$

$$altitud' = velocidad$$

$$velocidad' = aceleracion$$

Todas estas ecuaciones aparecen en la clase `Cohete`, mostrada debajo, donde la notación matemática (') para la derivada ha sido reemplazada en Modelica por la seudo función `der`(). La derivada de la `masa` del cohete es negativa, puesto que la velocidad con la que se consume el combustible es proporcional al `empuje` ejercido por el motor del cohete.

```
class Cohete "clase cohete"
  parameter String nombre;
  Real masa(start=1038.358);
  Real altitud(start= 59404);
```

```
  Real velocidad(start= -2003);
  Real aceleracion;
  Real empuje;     // Fuerza de empuje del cohete
  Real gravedad;   // Aceleración del campo gravitatorio
  parameter Real coefPerdidaMasa=0.000277;
equation
  (empuje-masa*gravedad)/masa = aceleracion;
  der(masa)  = -coefPerdidaMasa * abs(empuje);
  der(altitud) = velocidad;
  der(velocidad) = aceleracion;
end Cohete;
```

La siguiente ecuación especifica la intensidad del campo de fuerza gravitatorio. Forma parte de la clase `Alunizaje`, mostrada en la siguiente sección, ya que depende tanto de la masa del cohete y como de la masa de la luna:

$$gravedad = \frac{g_{luna} \cdot masa_{luna}}{\left(altitud_{apollo} + radio_{luna}\right)^2}$$

La cantidad del empuje que debe ser aplicado por el motor del cohete es específico a una clase particular de alunizaje, y por ello pertenece a la clase `Alunizaje`.

$$\begin{aligned} empuje = {} & if \ \left(time < timeDecrementoEmpuje\right) \ then \\ & fuerza1 \\ & else \ if \ \left(time < timeFinEmpuje\right) \ then \\ & fuerza2 \\ & else \ 0 \end{aligned}$$

## 3.5 Control de Acceso

Los miembros de las clases en Modelica tienen dos niveles de visibilidad: `public` o `protected`. En caso de no especificarse nada, la visibilidad por defecto es `public`. Por ejemplo, con relación a las variables `fuerza1` y `fuerza2` de la clase `Alunizaje` mostrada más abajo, la declaración como `public` de `fuerza1`, `fuerza2`, `apollo`, y `luna` significa que cualquier código que pueda acceder a una instancia de `Alunizaje` puede leer o modificar estos valores.

El otro posible nivel de visibilidad, especificado mediante la palabra clave `protected` (por ejemplo, para las variables `timeFinEmpuje` y `timeDecrementoEmpuje`), significa que sólo

puede accederse a estas variables desde el código *de dentro* de la clase y desde el código de las clases que heredan a ésta. El código de dentro de la clase incluye al código de las clases locales. Sin embargo, desde el código de la clase sólo puede accederse a la *misma* instancia de la variable protegida—las clases que extiendan la clase accederán a otra instancia de la variable protegida, puesto que las declaraciones son "copiadas" en la herencia. Esto es diferente a las correspondientes reglas de acceso en Java.

Observe que la aparición de una de las palabras clave `public` o `protected` implica que todas las declaraciones siguientes, hasta que vuelva a aparecer una de estas palabras clave, poseen el correspondiente nivel de visibilidad.

Las variables `empuje`, `gravedad` y `altitud` pertenecen a la instancia `apollo` de la clase `Cohete` y por ello son referenciadas anteponiendo el prefijo `apollo`, por ejemplo `apollo.empuje`. La constante gravitatoria `g`, la `masa` y el `radio` pertenecen a un cuerpo celeste en particular, llamado `luna`, sobre cuya superficie está posándose el cohete `apollo`.

```
class Alunizaje
  parameter Real fuerza1 = 36350;
  parameter Real fuerza2 = 1308;
  protected
  parameter Real timeFinEmpuje = 210;
  parameter Real timeDecrementoEmpuje = 43.2;
public
  Cohete         apollo(nombre="apollo12");
  CuerpoCeleste  luna(nombre="luna",masa=7.382e22,radio=1.738e6);
equation
  apollo.empuje = if (time< timeDecrementoEmpuje) then fuerza1
                  else if (time< timeFinEmpuje)   then fuerza2
                  else 0;
  apollo.gravedad = luna.g * luna.masa / (apollo.altitud +
                    luna.radio)^2;
end Alunizaje;
```

## 3.6 Simulación del Ejemplo del Alunizaje

Simulemos el modelo `Alunizaje` durante el intervalo de tiempo `{0,230}`. Para ello, usamos la siguiente orden del entorno de simulación OpenModelica:

```
simulate(Alunizaje, stopTime=230)
```

Puesto que la solución para la altitud del cohete Apollo es una función del tiempo, puede ser representada en un diagrama (ver Figura 3-2). Tiene una altitud igual a 59404m (no mostrada en el diagrama) en el instante cero, la cual va reduciéndose gradualmente hasta tocar la superficie lunar, cosa que sucede cuando la altitud vale cero.

```
plot(apollo.altitud, xrange={0,208})
```

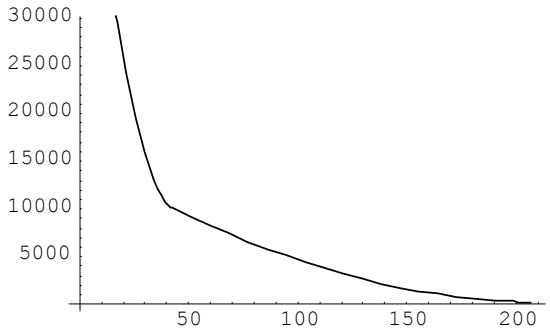



**Figura 3-2.** Altitud del cohete Apollo sobre la superficie lunar.

La fuerza de empuje del cohete es inicialmente alta, pero se reduce a un nivel bajo a partir del instante 43.2 segundos (que es el valor del parámetro `timeDecrementoEmpuje`), tal como se muestra en la Figura 3-3.

```
plot(apollo.empuje, xrange={0,208})
```

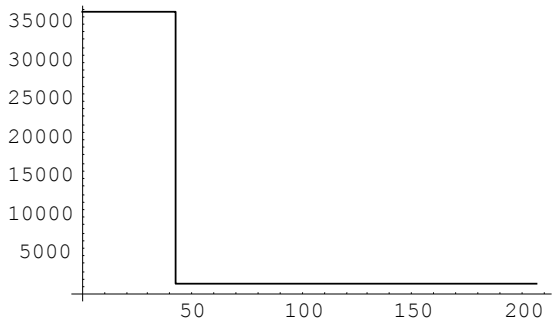



**Figura 3-3.** Empuje del motor del cohete, con un empuje inicial grande `fuerza1` seguido por un empuje menor `fuerza2`.

La masa del cohete decrece debido al consumo de combustible, desde un valor inicial de 1038.358 kg hasta aproximadamente 540 kg (vea la Figura 3-4).

```
plot(apollo.masa, xrange={0,208})
```

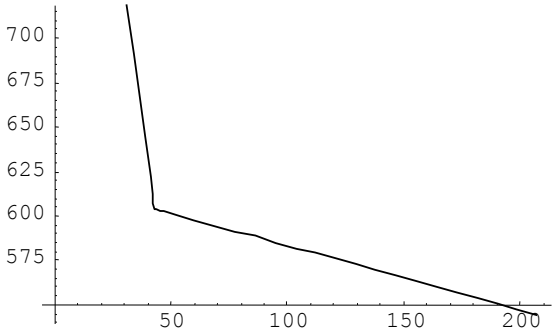



**Figura 3-4**. La masa del cohete decrece a medida que se consume el combustible.

El campo gravitatorio aumenta a medida que el cohete se aproxima a la superficie lunar, tal como se muestra en la Figura 3-5, donde la gravedad ha aumentado hasta 1.63 N/kg después de 200 segundos.

```
plot(apollo.gravedad, xrange={0,208})
```

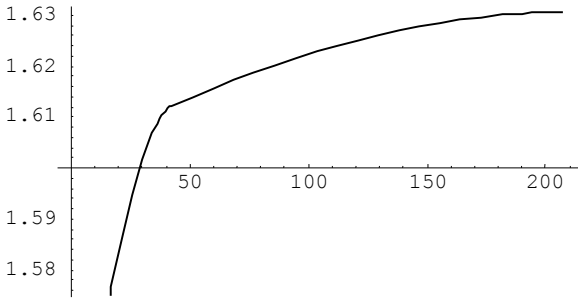



**Figura 3-5.** Incremento gradual de la gravedad a medida que el cohete se aproxima a la superficie lunar.

Al iniciarse la aproximación a la superficie lunar, el cohete tiene una velocidad negativa alta. Ésta va reduciéndose, hasta que vale cero al producirse el contacto. Así pues, el alunizaje es suave, como se muestra en la Figura 3-6.

```
plot(apollo.velocidad, xrange={0,208})
```

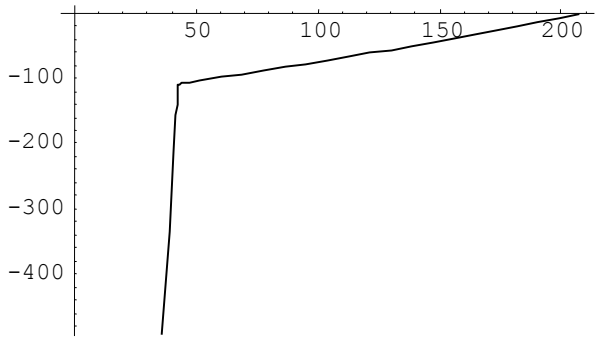



**Figura 3-6.** Velocidad vertical relativa a la superficie lunar.

Cuando el lector experimente con el modelo `Alunizaje`, debe darse cuenta de que el modelo no reproduce la física del alunizaje en al menos una cuestión importante. Si se permite que la simulación prosiga tras el contacto con la luna, es decir, tras el instante en el cual la velocidad se ha reducido a cero, entonces la velocidad crecerá de nuevo y el cohete se acelerará hacia el centro de la luna. Esto sucede porque no se ha modelado la fuerza de contacto que ejerce la superficie de la luna sobre el cohete una vez que este ha alunizado, la cual evitaría que esto suceda. Se propone como ejercicio introducir esta fuerza, ejercida por el suelo, en el modelo `Alunizaje`.

## 3.7 Herencia

Veamos un ejemplo de cómo extender una clase sencilla en Modelica. Por ejemplo, la clase `ColorDatos` que fue introducida en la Sección 2.4. Se muestran dos clases, llamadas `ColorDatos` y `Color`. La clase derivada (subclase) `Color` hereda de su clase base (superclase) `ColorDatos` las variables para representar el color y añade una ecuación que es una restricción sobre el valor de los colores.

```
record ColorDatos
  Real   rojo;
  Real   azul;
  Real   verde;
end ColorDatos;

class Color
  extends ColorDatos;
equation
```

```
  rojo + azul + verde = 1;
end Color;
```

Los datos y el comportamiento de la superclase, que toman la forma de declaraciones de variables y atributos, ecuaciones y otros contenidos, son copiados en la subclase en el proceso de herencia. Sin embargo, como ya mencionamos anteriormente, antes de realizar la copia se hacen sobre las definiciones heredadas ciertas expansiones de tipos, comprobaciones y operaciones de modificación. La clase `Color`, una vez expandida, es equivalente a la clase siguiente:

```
class ColorExpandida
  Real rojo;
  Real azul;
  Real verde;
equation
  rojo + azul + verde = 1;
end ColorExpandida;
```

### 3.7.1 Herencia de Ecuaciones

En la sección anterior, se dijo que las ecuaciones heredadas son copiadas desde la superclase o clase base y son insertadas en la subclase o clase derivada. ¿Qué sucede si ya existe una *ecuación idéntica* que ha sido declarada localmente en la clase derivada? En tal caso habrán dos ecuaciones identicas, hacienda que el sistema sea sobredeterminado e imposible de resolver

```
class Color2
  extends Color;
equation
  rojo + azul + verde = 1;
end Color2;
```

La clase expandida `Color2` es equivalente a la siguiente clase:

```
class Color2Expandida
  Real rojo;
  Real azul;
  Real verde;
equation
  rojo + azul + verde = 1.0;
  rojo + azul + verde = 1.0;
end Color2Expandida;
```

### 3.7.2 Herencia Múltiple

Modelica permite la herencia múltiple, es decir, varias sentencias `extends`. Esto resulta útil cuando una clase quiere incluir varios tipos ortogonales de comportamiento y de datos. Por ejemplo, combinar geometría y color.

Por ejemplo, la nueva clase `PuntoColoreado` hereda de múltiples clases, es decir, usa herencia múltiple. Obtiene las variables *posición* de la clase `Punto`, y las variables color y la ecuación de la clase `Color`.

```
class Punto
  Real x;
  Real y, z;
end Punto;

class PuntoColoreado
  extends Punto;
  extends Color;
end PuntoColoreado;
```

En algunos lenguajes de programación orientados a objetos, la herencia múltiple causa problemas cuando se hereda la misma definición dos veces a través de diferentes clases intermedias. Un caso bien conocido es la denominada herencia en diamante (ver Figura 3-7):

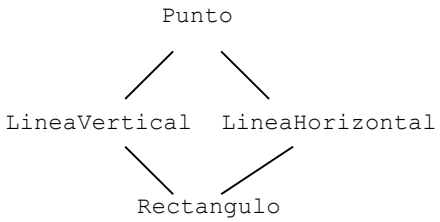



**Figura 3-7.** Herencia en diamante.

La clase `Punto` contiene las coordenadas de una posición, definida por las variables `x` e `y`. La clase `LineaVertical` hereda `Punto`, pero también añade la variable `vlongitud` para la longitud de la línea. Análogamente, la clase `LineaHorizontal` hereda las variables posición `x`, `y` y añade la longitud horizontal. Finalmente, se pretende que la clase `Rectangulo` posea las variables que definen la posición `x` e `y`, la longitud vertical y la longitud horizontal.

```
class Punto
  Real x;
  Real y;
```

```
end Punto;

class LineaVertical
  extends Punto;
  Real vlongitud;
end LineaVertical;

class LineaHorizontal
  extends Punto;
  Real hlongitud;
end LineaHorizontal;

class Rectangulo
  extends LineaVertical;
  extends LineaHorizontal;
end Rectangulo;
```

El problema potencial es que se tiene una herencia en diamante, ya que la coordenada de posición definida mediante las variables `x` e `y`, es heredada dos veces: una vez de `LineaVertical` y otra de `LineaHorizontal`. ¿Qué variables deberán usarse, las que provienen de `LineaVertical` o las que provienen de `LineaHorizontal`? ¿Existe alguna forma de resolver este problema?

Sí, existe una forma de resolverlo. En Modelica la herencia en diamante no es un problema, puesto que existe una regla que establece que si se heredan varias declaraciones o ecuaciones idénticas, entonces sólo una de ellas se mantiene. Así pues, habrá un único conjunto de variables posición en la clase `Rectangulo`, de modo que el conjunto total de variables de la clase es el siguiente: `x`, `y`, `vlongitud` y `hlongitud`. Lo mismo aplica para las clases `Rectangulo2` y `Rectangulo3` mostradas debajo.

```
class Rectangulo2
  extends Punto;
  extends LineaVertical;
  extends LineaHorizontal;
end Rectangulo2;

class Rectangulo3
  Real x, y;
  extends LineaVertical;
  extends LineaHorizontal;
end Rectangulo3;
```

El lector quizá puede pensar que habrá situaciones en las que el resultado dependerá del orden relativo de las cláusulas `extends`. Sin embargo, esta situación no puede darse en Modelica, como se explicará en la Sección 3.7.4.

### 3.7.3 Procesado de los Elementos de la Declaración y Uso Previo a la Declaración

La búsqueda y el análisis de los elementos de las declaraciones dentro de una clase se realiza de la manera descrita a continuación. Con ello, se garantiza que los elementos declarados puedan ser usados antes ser declarados y que el proceso no dependa del orden en que son declarados.

1. Se buscan los *nombres* de las clases, variables y otros atributos declarados localmente. Además, se juntan los modificadores con las declaraciones de los elementos locales y se aplican las redeclaraciones.
2. Se procesan las *cláusulas extends*, buscando y expandiendo las clases heredadas. El contenido de éstas es expandido e insertado en la clase actual. La búsqueda de las clases heredadas debería realizarse *independientemente*, es decir, el análisis y expansión de una cláusula `extends` no debería depender de las demás.
3. Se expanden todas las declaraciones de elementos y se comprueban los tipos.
4. Se comprueba que todos los elementos que tienen el mismo nombre son idénticos.

La razón por la cual lo primero que se hace es buscar los nombres de las tipos, variables y otros atributos locales es es possible que un elemento haya sido *usado* antes de ser *declarado*. Por ello, deben conocerse los nombres de los elementos de la clase antes de realizar las expansiones o el análisis de los tipos.

Por ejemplo, en la clase C2 se usan las clases `Voltage` y `Lpin` antes de ser declaradas:

```
class C2
  Voltage v1, v2;
  Lpin    pn;

  class Lpin
    Real p;
  end Lpin;
  class Voltage = Real(unit="kV");
end C2;
```

### 3.7.4 Orden de la Declaración de las Cláusulas `extends`

Como se ha indicado en el Capítulo 2 y en las secciones previas, en Modelica el uso de los elementos declarados es independiente del orden en el cual son declarados, con la excepción de los parámetros formales de las funciones y los campos de registro de las funciones (variables). Así

pues, las variables y las clases pueden ser usadas antes de ser declaradas. Esto también es válido para las cláusulas `extends`. El orden en que están escritas las cláusulas `extends` dentro de una clase no influye en las declaraciones y ecuaciones heredadas mediante dichas cláusulas `extends`.

### 3.7.5 El Ejemplo del `Alunizaje` Usando Herencia

En el ejemplo `Alunizaje` de la sección 3.4 se repite la declaración de ciertas variables, tales como `masa` y `nombre`, en las clases `CuerpoCeleste` y `Cohete`. Esta repetición puede evitarse, declarando estas variables en una clase llamada `Cuerpo`, que represente un cuerpo genérico, y reusando esta clase heredándola en `CuerpoCeleste` y `Cohete`. Se muestra a continuación la clase `Alunizaje` reestructurada de esta forma. Se ha reemplazado la palabra clave `class` por la palabra clave más restrictiva `model`, que tiene la misma semántica que `class` aparte del hecho de que no puede ser usada en sentencias de conexión. La razón de ello es que es más frecuente emplear la palabra clave `model` para propósitos de modelado que la palabra clave `class`. El primer modelo es la clase `Cuerpo`, que ha sido diseñada para ser heredada por otros tipos de cuerpo más especializados.

```
model Cuerpo  "cuerpo genérico"
  Real   masa;
  String nombre;
end Cuerpo;
```

La clase `CuerpoCeleste` hereda de la clase `Cuerpo` y constituye, de hecho, una versión especializada de `Cuerpo`. Compare ésta con la versión sin herencia de `CuerpoCeleste` presentada en la sección 3.4.

```
model CuerpoCeleste "cuerpo celeste"
  extends Cuerpo;
  constant  Real g = 6.672e-11;
  parameter Real radio;
end CuerpoCeleste;
```

La clase `Cohete` también hereda de la clase `Cuerpo` y puede ser considerada otra versión especializada de `Cuerpo`.

```
model Cohete "clase genérica cohete"
  extends Cuerpo;
  parameter Real coefPerdidaMasa=0.000277;
  Real altitud(start= 59404);
  Real velocidad(start= -2003);
```

```
  Real aceleracion;
  Real empuje;
  Real gravedad;
equation
  empuje - masa * gravedad = masa * aceleracion;
  der(masa)  = -coefPerdidaMasa * abs(empuje);
  der(altitud) = velocidad;
  der(velocidad) = aceleracion;
end Cohete;
```

La clase `Alunizaje` mostrada a continuación es idéntica a la presentada en la sección 3.4, aparte del hecho de que se ha sustituido la palabra clave `class` por `model`.

```
model Alunizaje
  parameter Real fuerza1 = 36350;
  parameter Real fuerza2 = 1308;
  parameter Real timeFinEmpuje = 210;
  parameter Real timeDecrementoEmpuje = 43.2;
  Cohete    apollo(nombre="apollo12", masa(start=1038.358) );
  CuerpoCeleste luna(masa=7.382e22,radio=1.738e6,nombre="luna");
equation
  apollo.empuje = if (time< timeDecrementoEmpuje) then fuerza1
                  else if (time< timeFinEmpuje) then fuerza2
                       else 0;
  apollo.gravedad = luna.g*luna.masa /(apollo.altitud+luna.radio)^2;
end Alunizaje;
```

Simulamos este modelo `Alunizaje` reestructurado durante el intervalo de tiempo {0, 230}:

```
simulate(Alunizaje, stopTime=230)
```

Como era de esperar, el resultado es idéntico al de `Alunizaje` mostrado en la sección 3.5. Vea, por ejemplo, la altitud del cohete Apollo en función del tiempo, graficada en la **Figura 3-8**.

```
plot(apollo.altitud, xrange={0,208})
```

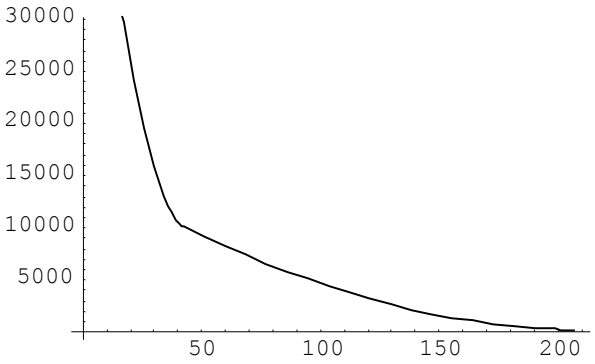



**Figura 3-8.** Altitud del cohete Apollo sobre la superficie lunar.

## 3.8 Resumen

Este capítulo se ha enfocado en el concepto estructural mas importante en Modelica: el concepto de clase. Se inició con una introducción en forma de tutorial de la idea de contacto entre el diseñador y el usuario en conjunto con lo básico de las clases de Modelica, como se ejemplificó con el modelo de alunizaje en la luna.

## 3.9 Literatura

Un documento de referencia importante para este capítulo es la especificación de lenguaje de Modelica por la Modelica Asociation (2010). Varios ejemplos de este capítulo están basados en ejemplos similares que se encuentran enel mismo document anterior y en Modelica Asociation (2000). Al comienzo de este capítulo se mencionó la idea de un contrato entre diseñadores de software y usuarios. Esto es parte de los métodos y principios para el modelamiento y diseño de software orientado a objetos descrito por Rumbaugh et al. (1991), Booch (1991,1994), y Meyer (1997). El ejemplo de alunizaje en la luna está basado en la misma formulación de las ecuaciones de Newton como está mostrado en el ejemplo del alunizaje en la luna en Cellier (1991)

# Capítulo 4

# Metodología de Modelamiento de Sistemas

Hasta el momento, en este texto se han explicado los principios del modelado matemático orientado a objetos, cierto número de construcciones del lenguaje Modelica, que posibilitan una representación de alto nivel del modelo y un alto grado de reutilización del mismo, y algunos ejemplos de modelos que demuestran el uso de estas construcciones del lenguaje.

Sin embargo, hasta ahora no hemos presentado un método sistemático para el modelado de los sistemas bajo estudio. Tampoco hemos presentado hasta el momento, de manera detallada, las ecuaciones matemáticas en el formato del espacio de estados que deberán emplearse. Estos son los temas que se tratarán en este capítulo. No obstante, las explicaciones acerca de la representación en el espacio de estados cubren sólo el caso sencillo de los sistemas continuos.

## 4.1 Construcción de Modelos de un Sistema

Una pregunta básica es: ¿de qué manera pueden obtenerse modelos matemáticos adecuados de los sistemas que queremos estudiar, y posiblemente simular? Es decir, ¿qué es un proceso de modelado eficaz?

El dominio de aplicación es de importancia capital en todos los tipos de modelado. Se habla de *modelado físico* cuando los sistemas a modelar vienen descritos por las leyes naturales de la física, la química, la biología, la mecánica, la ingeniería eléctrica, etc., y estas leyes pueden ser representadas directamente en el modelo matemático que se está formulando. Sin embargo, lo importante no es si el dominio de aplicación es “físico” o no. Existen leyes que gobiernan los

sistemas económicos, de comunicación de datos, de procesamiento de la información, etc., que pueden ser expresadas más o menos directamente en el modelo que estamos construyendo cuando se usa una aproximación de modelado de alto nivel. Todas estas leyes vienen dadas en el universo en el que existimos (si bien son formuladas por los seres humanos), y pueden ser consideradas como leyes básicas de la naturaleza.

Al comenzar la tarea de modelado, en primer lugar es preciso identificar qué dominios de aplicación están involucrados en el sistema que deseamos modelar, y para cada uno de estos dominios es preciso encontrar qué leyes influyen, de manera relevante, en los fenómenos que deseamos estudiar.

Para poder manejar la complejidad de los modelos de grandes dimensiones, y reducir el esfuerzo de modelado reutilizando modelos de componentes, es bastante útil aplicar la descomposición jerárquica y las técnicas basadas en componentes orientadas a objetos, tales como las tratadas en este texto. Para clarificar este punto, compararemos brevemente el enfoque tradicional al modelado físico con el enfoque basado en el modelado basado en componentes orientado a objetos.

Sin embargo, debe tenerse en cuenta que incluso el enfoque tradicional al modelado físico es de "más alto nivel" que algunos otros enfoques, tales como el modelado orientado a bloques o el programar directamente el modelo usando algún lenguaje imperative común. Estos últimos requieren que el usuario debe convertir manualmente las ecuaciones en sentencias de asignación o bloques y debe reestructurar manualmente el código, para adaptar el contexto de los flujos de datos y señales al determinado uso específico que se va a dar al modelo.

### 4.1.1 Modelado Deductivo vs. Modelado Inductivo

Hasta el momento, hemos tratado casi exclusivamente la aproximación al modelado denominada *modelado deductivo*, también conocida como *modelado físico*, donde el comportamiento del sistema es *deducido* al aplicar las leyes naturales, que son expresadas en un modelo del sistema. Los modelos de este tipo son creados en base al conocimiento existente de los procesos "físicos" o "artificiales" que rigen el sistema en cuestión, de ahí el nombre de "modelado físico".

Sin embargo, en algunos casos, especialmente en sistemas biológicos y económicos, puede que no se disponga del conocimiento preciso sobre sistemas complejos y de sus procesos internos, que hasta cierta medida sería necesario para soportar el modelamiento físico. En estas áreas de aplicación es frecuente emplear un procedimiento de modelado completamente diferente. Se realizan observaciones del sistema bajo estudio y se trata de ajustar un modelo matemático hipotético a los datos observados, adaptando el modelo, típicamente mediante el cálculo de

valores de determinados coeficientes desconocidos del mismo. Este procedimiento se denomina *modelado inductivo*.

Los modelos inductivos se basan directamente en los valores medidos. Esto hace que estos modelos sean difíciles de validar más allá de los valores observados. Por ejemplo, podríamos querer acomodar algunos mecanismos en un modelo del sistema de tal forma que nos permitieran predecir desastres, y en consecuencia, posiblemente prevenirlos. Sin embargo, esto sería imposible sin haber observado previamente un desastre real (que quisiéramos evitar a toda costa) en el sistema. Esta es una desventaja clara de los modelos inductivos.

Obsérvese también que, añadiendo un número suficientemente grande de parámetros a un modelo inductivo, es posible ajustar virtualmente cualquier modelo a virtualmente cualquier conjunto de datos. Esta es una de las desventajas más fuertes de los modelos inductivos, ya que de esta manera uno puede engañarse fácilmente acerca de la validez del modelo.

En el resto de este libro se tratará principalmente la aproximación deductiva al modelado o modelado físico, con excepción de algunos pocos ejemplos de aplicación de sistemas biológicos, en los cuales se emplearán modelos que en parte son inductivos y en parte tienen una motivación física.

### 4.1.2 Enfoque Tradicional

La metodología tradicional para el modelado físico puede, a grosso modo, dividirse en tres fases:

1. Establecer la estructura básica en términos de las variables.
2. Plantear las ecuaciones y las funciones.
3. Convertir el modelo al formato de espacio de estados.

La primera fase incluye decidir qué variables son de interés. Por ejemplo, en base al uso que se va a dar al modelo y a los papeles que desempeñarán estas variables. ¿Qué variables deben ser consideradas como *entradas*, por ejemplo, las señales externas, como *salidas* o como variables de *estado* internas? ¿Qué cantidades son especialmente importantes para describir lo que sucede en el sistema? ¿Cuáles dependen del tiempo, y cuáles son aproximadamente *constantes*? ¿Qué variables influyen sobre otras variables?

La *segunda* fase consiste en plantear las ecuaciones básicas y las fórmulas del modelo. Hay que buscar qué leyes del dominio de aplicación son relevantes para el modelo. Por ejemplo, las ecuaciones de conservación de magnitudes del *mismo tipo*, tales como la potencia entrante relacionada con la potencia saliente, la tasa de flujo entrante relacionado con la tasa del flujo saliente, y la conservación de cantidades tales como la energía, la masa, la carga, la información,

etc. Asimismo, hay que formular *ecuaciones constitutivas* que relacionen magnitudes de *diferente tipo*. Por ejemplo, el voltaje y la corriente en una resistencia, los flujos de entrada y salida de un tanque, paquetes de entrada y de salida para un enlace de comunicaciones, etc. También puede ser preciso formular relaciones, por ejemplo, acerca de propiedades de los materiales y otras propiedades del sistema. A la hora de plantear estas relaciones, debe tenerse en cuenta un intermedio entre el nivel de precisión requerido y el grado de aproximación apropiado.

La *tercera* fase consiste en convertir el modelo desde su formato actual, consistente en un conjunto de variables y un conjunto de ecuaciones, a una representación del sistema mediante ecuaciones en el *espacio de estados*, que se adecue al solucionador numérico a emplear. Es necesario escoger un conjunto de variables de estado, expresar sus derivadas temporales en función de las variables de estado, para variables dinámicas, y las variables de entrada (para el caso de ecuaciones en el espacio de estados en forma explícita), y expresar las variables de salida en función de las variables de estado y las variables de entrada. Si se incluyen algunas variables de estado innecesarias no causa ningún perjuicio, sólo conlleva realizar cálculos innecesarios. Esta fase es completamente innecesaria cuando se usa Modelica, puesto que la conversión al formato del espacio de estados es realizada de manera automática por el compilador de Modelica.

### 4.1.3 Enfoque Basado en Componentes Orientados a Objetos

Cuando se aplica el enfoque basado en componentes orientados a objetos, en primer lugar tratamos de entender la estructura del sistema y de descomponerlo jerárquicamente, de forma descendente. Cuando los componentes del sistema y las interacciones entre estos componentes han sido identificadas a grosso modo, podemos aplicar las dos primeras fases del modelado tradicional, consistentes en identificar las variables y ecuaciones de estos componentes. El enfoque orientado a objetos consta de las fases siguientes:

1. *Definir* el sistema brevemente: ¿Qué tipo de sistema es? ¿Qué hace?
2. *Descomponer* el *sistema* en sus *componentes* más importantes. Esbozar clases de modelos para esos componentes, o emplear clases ya existentes pertenecientes a las librerías apropiadas.
3. Definir la *comunicación*, es decir, determinar las interacciones y los caminos de comunicación entre los componentes.
4. Definir las *interfaces*, es decir, determinar los puertos/*connectors* externos de cada componente para la comunicación con los otros componentes. Formular clases de

conectores apropiadas, que permitan un alto grado de conectividad y reusabilidad, a la vez que permitan un grado apropiado de comprobación de los tipos de las conexiones.

5. *Descomponer* recursivamente los *modelos de los componentes* de "alta complejidad", en un conjunto de subcomponentes "más pequeños", comenzando de nuevo en la fase 2, hasta que todos los componentes han sido definidos como instancias de tipos predefinidos, clases pertenecientes a librerías o nuevas clases definidas por el usuario.
6. *Formular nuevas clases de modelos* cuando sea preciso, tanto clases base como clases derivadas:
    a. Declarar nuevas clases de modelos para todos los modelos componentes que no sean instancias de clases existentes. Cada nueva clase debe incluir las variables, ecuaciones, funciones y formulas que sean relevantes para definir el comportamiento del componente, de acuerdo con los principios ya descritos en las dos primeras fases del enfoque tradicional al modelado.
    b. Declarar las *clases base* posibles, que favorezcan la reutilización y faciliten el mantenimiento, extrayendo la funcionalidad en común y las similitudes estructurales de clases de componentes que tengan propiedades similares.

Para entender mejor cómo funciona en la práctica el enfoque del modelado orientado a objetos, en la Sección 4.2 aplicaremos este método al modelado de un sistema de tanques sencillo.

### 4.1.4 Modelado Descendente vs. Modelado Ascendente

El proceso de definición de la estructura del modelo puede enfocarse de dos maneras relacionadas entre sí:

- *Modelado descendente*. Es útil cuando se conoce el área de aplicación bastante bien y se dispone de una librería de modelos de componentes. Se comienza definiendo el modelo del sistema al más alto nivel, descomponiéndolo gradualmente en subsistemas, hasta que se llega a los subsistemas que corresponden con los modelos de componentes de las librerías.
- *Modelado ascendente*. Este enfoque se usa típicamente cuando la aplicación es menos conocida o cuando no se dispone de una librería de componentes. En primer lugar, se formulan las ecuaciones básicas y se diseñan pequeños modelos experimentales de los fenómenos más importantes, con el fin de intentar comprender el área de aplicación. Típicamente, se comienza con modelos muy simplificados, y posteriormente se van añadiendo más fenómenos. Tras alguna experimentación, se va adquiriendo cierto

conocimiento acerca del área de aplicación, y entonces es posible reestructurar estos fragmentos de modelos en un conjunto de componentes de modelos. Al emplearlo en diferentes aplicaciones, el conjunto de componentes puede dar problemas, en cuyo caso deberá ser estructurado tantas veces como sea preciso. De esta forma, se van gradualmente construyendo modelos de aplicación más complejos basados en estos componentes, hasta que finalmente se obtiene el modelo de aplicación deseado.

A continuación se presentan algunos ejemplos de modelamiento descendente y de modelamiento ascendente. En la sección 4.3 se presenta el modelamiento de un motor de corriente continua a partir de componentes predefinidos, el cual es un ejemplo típico de un modelamiento descendente. El pequeño ejemplo del modelado de un tanque descrito en la Sección 4.2 posee ciertas características del modelado ascendente, dado que se comienza con un modelo plano sencillo de un tanque antes de crear las clases de componentes y formar el modelo del tanque. Estos ejemplos van creciendo gradualmente hasta formar un conjunto de componentes que son usados para componer los modelos de aplicación finales.

### 4.1.5 Simplificación de Modelos

En ocasiones, los modelos no son lo suficientemente precisos como para describir de manera adecuada el fenómeno. Esto puede ser debido a la realización de aproximaciones demasiado simplificadoras en determinadas partes del modelo.

También puede darse la situación contraria. Aunque se construya un modelo razonable siguiendo la metodología anterior, no es infrecuente que algunas partes del modelo sean demasiado complejas, lo cual conlleva problemas tales como:

- Simulaciones que tardan demasiado en ejecutarse.
- Inestabilidades numéricas.
- Dificultad a la hora de interpretar los resultados, debido a que el modelo contiene demasiados detalles de bajo nivel.

En conclusión, con frecuencia existen buenas razones para plantearse simplificar el modelo. En ocasiones, es difícil encontrar en el modelo el equilibrio adecuado entre simplicidad y precisión. Se trata más de un arte que de una ciencia y requiere una experiencia considerable. Sin embargo, la mejor manera de adquirir esa experiencia es mediante el diseño de modelos, y el análisis y evaluación de su comportamiento. A continuación, se dan algunos consejos acerca de cómo simplificar un modelo, por ejemplo, reduciendo el número de sus variables de estado:

- *Despreciar* efectos pequeños que no sean importantes para el fenómeno que se esté modelando.
- *Agregar* las variables de estado en un número menor de variables. Por ejemplo, las temperaturas en diferentes puntos de una varilla pueden, en ocasiones, representarse mediante la temperatura media en toda la varilla.

*Enfocar* el modelado en aquellos fenómenos cuyas constantes de tiempo están en el rango de interés, es decir:

- Aproximar por constantes aquellos subsistemas que tengan una dinámica muy lenta.
- Aproximar aquellos subsistemas que tengan una dinámica muy rápida mediante relaciones estáticas. Es decir, relaciones en las que no interviene la derivada de estas variables de estado que cambian rápidamente.

Una ventaja de no modelar las dinámicas muy rápidas y las muy lentas de un sistema es que se reduce el orden del modelo. Es decir, el número de sus variables de estado. Aquellos modelos cuyos componentes tienen constantes de tiempo con el mismo orden de magnitud son numéricamente más simples y más eficientes de simular. Por otra parte, ciertos sistemas poseen la propiedad intrínseca de que sus constantes de tiempo son muy dispares. Estos sistemas dan lugar a sistemas de ecuaciones diferenciales difíciles, los cuales requieren para su simulación del empleo de solucionadores numéricos adaptivos.

## 4.2 Modelado de un Sistema de un Tanque

Como ejercicio para ilustrar la metodología de modelado, consideremos el ejemplo sencillo de un sistema con un tanque, que contiene además un sensor de nivel y un controlador, el cual controla una válvula por medio de un actuador (Figura 4-1).

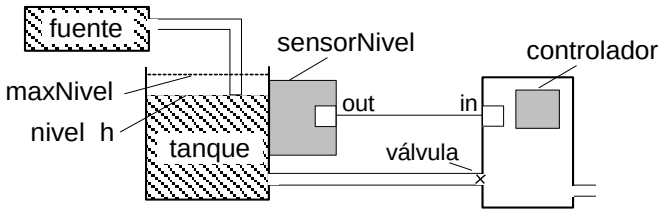



**Figura 4-1.** Un sistema con un tanque, una fuente de líquido y un controlador.

El nivel de líquido `h` en el tanque debe mantenerse tan próximo como sea posible a un determinado nivel de referencia. El líquido entra en el tanque, a través de una tubería, procedente de una fuente, y sale del tanque a través de una tubería con un caudal que es controlado por una válvula.

### 4.2.1 Usando del Enfoque Tradicional

En primer lugar, mostraremos el resultado de modelar el sistema de un tanque usando el enfoque tradicional. En la próxima sección, se muestra el modelo plano con las variables y ecuaciones del sistema de un tanque. En este punto, no explicaremos cómo han sido deducidas las ecuaciones ni que significado tienen ciertas variables. Esto será descrito en detalle en las siguientes secciones, en las que se presentará el modelado orientado a objetos de este sistema tanque.

#### 4.2.1.1 Modelo Plano del Sistema de Tanque

El modelo `TanquePlano` es un modelo "plano" del sistema de un tanque, el cual no evidencia una "estructura del sistema" jerárquica interna. Es decir, se trata solo de una colección de variables y ecuaciones que modelan la dinámica del sistema. La estructura interna del sistema, que consiste en componentes, interfaces, acoples entre los componentes, etc., no se refleja en este modelo.

```
model TanquePlano
  // Variables y parámetros relacionadas con el tanque
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2")       =1;
  parameter Real flowGain(unit="m2/s") =0.05;
  Real      h(start=0,unit="m")   "Nivel en el tanque";
  Real      qInflow(unit="m3/s")  "Flujo en la válvula de entrada";
  Real      qOutflow(unit="m3/s") "Flujo en la válvula de salida";
  // Variables y parámetros relacionadas con el controlador
  parameter Real K=2                   "Ganancia";
  parameter Real T(unit="s")= 10       "Constante de tiempo";
  parameter Real minV=0, maxV=10;  // Limites del flujo de salida
  Real   ref = 0.25  "Nivel de referencia para el control";
  Real   error       "Desviación respecto al nivel de referencia";
  Real   outCtr      "Señal de control sin limitador";
  Real   x;          "Variable de estado del controlador";
equation
  assert (minV>=0,"minV debe ser mayor o igual a cero");//
  der(h) = (qInflow-qOutflow)/area;   // Balance de masa
```

```
  qInflow  = if time>150 then 3*flowLevel else flowLevel;
  qOutflow = LimitValue(minV,maxV,-flowGain*outCtr);
  error  = ref-h;
  der(x) = error/T;
  outCtr = K*(error+x);
end TanquePlano;
```

En el modelo se emplea una función para reflejar que hay unos flujos mínimo y máximo a través de la válvula de salida:

```
function LimitValue
  input  Real pMin;
  input  Real pMax;
  input  Real p;
  output Real pLim;
algorithm
  pLim := if p>pMax then pMax
          else if p<pMin then pMin
          else p;
end LimitValue;
```

Simulando el modelo plano del sistema de tanque y graficando la variable nivel se obtiene el resultado mostrado en la Figura 4-2.

```
simulate(TanquePlano, stopTime=250)
plot(h, stopTime=250)
```

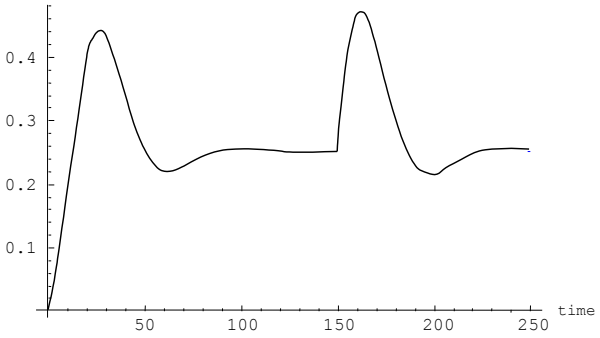



**Figura 4-2.** Simulación del modelo `TanquePlano` y gráfica del nivel en el tanque.

### 4.2.2 Usando el Enfoque Basado en Componentes Orientado a Objetos

Al aplicar el enfoque de modelado basado en componentes orientado a objetos para modelar, en primer lugar se analiza la estructura interna del sistema de un tanque. ¿Se descompone el sistema, de manera natural, en ciertos componentes? La respuesta es sí. En el diagrama del sistema de la Figura 4-3 pueden distinguirse varios componentes. Por ejemplo, el tanque, la fuente de líquido, el sensor de nivel, la válvula y el controlador.

Así pues, aparentemente tenemos cinco componentes. Sin embargo, se van a emplear representaciones muy sencillas del sensor de nivel y de la válvula: simplemente una variable escalar para cada uno de estos dos componentes. Por este motivo, estos dos componentes serán definidos como variables tipo `Real` en el modelo del tanque, en lugar de definir dos nuevas clases conteniendo una única variable cada una. Así pues, tendremos tres componentes, que deberán ser modelados explícitamente como instancias de nuevas clases: el tanque, la fuente y el controlador.

La siguiente etapa es determinar las interacciones y los caminos de comunicación entre estos componentes. Es bastante obvio que el fluido circula desde la fuente hasta el tanque a través de una tubería. El fluido abandona el tanque a través de una salida controlada por una válvula. El controlador necesita medidas del nivel de líquido, que provienen del sensor. Así pues, es necesario establecer un camino de comunicación desde el sensor del tanque hasta el controlador.

Los caminos de comunicación necesitan ser conectados en algún sitio. Por ello, es necesario crear instancias de conectores para aquellos componentes que son conectados, y declarar las clases connector necesarias. De hecho, el modelo del sistema debería ser diseñado de modo que la única comunicación entre un componente y el resto del sistema se produjera a través de conectores.

Finalmente, deberíamos pensar acerca de la reutilización y la generalización de ciertos componentes. ¿Esperamos que sean necesarias varias variantes de los componentes? En ese caso, es útil agrupar la funcionalidad básica en una clase base y hacer que cada variante sea una especialización de esa clase base. En el caso del sistema de un tanque, van a emplearse varias variantes del controlador, comenzando con un controlador continuo proporcional e integral (PI). Así pues, resulta útil crear una clase base para los controladores del sistema de un tanque.

### 4.2.3 Sistema de un Tanque con un Controlador continuo tipo PI

La estructura del sistema de un tanque, desarrollada aplicando el modelado basado en componentes orientado a objetos, es claramente visible en la Figura 4-3.

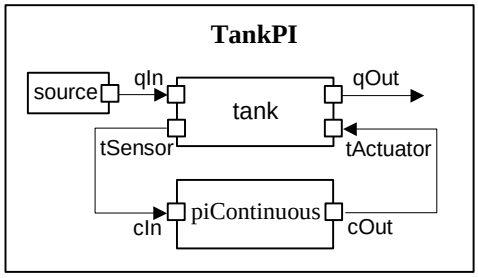



**Figura 4-3.** Un sistema con un tanque, un controlador PI continuo y una fuente de líquido. Aunque por motivo de una mayor claridad se han usado flechas, no es necesario asignar dirección a las señales—sólo hay conexiones físicas representadas mediante ecuaciones.

Los tres componentes del sistema de tanque—el tanque, el controlador PI y la fuente de líquido—son explícitos en la Figura 4-3 y en la declaración de la clase `TankPI` que se muestra a continuación.

```
model TankPI
  LiquidSource           source(flowLevel=0.02);
  PIcontinuousController piContinuous(ref=0.25);
  Tank                   tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, piContinuous.cOut);
  connect(tank.tSensor, piContinuous.cIn);
end TankPI;
```

Las instancias de los tanques son conectadas a los controladores y a las fuentes de líquido a través de conectores. El tanque tiene cuatro conectores: `qIn` para el flujo de entrada, `qOut` para el flujo de salida, `tSensor` para proporcionar medidas del nivel de líquido y `tActuator` para fijar la posición de la válvula situada en la salida del tanque. La ecuación central que regula el comportamiento del tanque es la ecuación del *balance de masa*, que se ha formulado de manera simplificada, asumiendo que la presión es constante. El flujo de salida se relaciona con la posición

de la válvula mediante el parámetro `flowGain`, y mediante un limitador que garantiza que el flujo no excede las correspondientes posiciones abierto/cerrado de la válvula.

```
model Tank
  ReadSignal  tSensor "Conector, sensor leyendo el nivel en el tanque
(m)";
  ActSignal   tActuator "Conector, actuador controlando el flujo de
entrada input flow";
  LiquidFlow qIn  "Conector, flujo (m3/s) a través de la válvula de
entrada";
  LiquidFlow  qOut  "Conector, flujo (m3/s) a través de la válvula de
salida";
  parameter Real area(unit="m2")       = 0.5;
  parameter Real flowGain(unit="m2/s") = 0.05;
  parameter Real minV=0, maxV=10; // Límites para el flujo de salida a
través de la válvula
  Real h(start=0.0, unit="m") "Nivel en el tanque";
equation
  assert (minV>=0,"minV - el nivel mínimo de la válvula debe ser >= 0
");//
  der(h)      = (qIn.lflow-qOut.lflow)/area;   // Balance de masa
equation
  qOut.lflow  = LimitValue(minV,maxV,-flowGain*tActuator.act);
  tSensor.val = h;
end Tank;
```

Como se ha indicado anteriormente, el tanque tiene cuatro conectores. Son instancias de las siguientes tres clases de conector:

```
connector ReadSignal  "Lectura del nivel de líquido"
  Real val(unit="m");
end ReadSignal;

connector ActSignal   "Señal enviada al actuador para fijar la
posición de la válvula"
  Real act;
end ActSignal;

connector LiquidFlow  "Flujo de líquido en las entradas y salidas"
  Real lflow(unit="m3/s");
end LiquidFlow;
```

El fluido que entra en el tanque debe provenir de algún sitio. Por ello, tenemos un componente en el sistema de tanque que representa la fuente de líquido. El flujo aumenta abruptamente en el

instante time=150 triplicándose el flujo existente hasta ese momento, lo cual crea un problema de control interesante que el controlador del tanque debe manejar.

```
model LiquidSource
  LiquidFlow qOut;
  parameter  flowLevel = 0.02;
equation
  qOut.lflow = if time>150 then 3*flowLevel else flowLevel;
end LiquidSource;
```

Es preciso especificar el controlador. Inicialmente, escogeremos un controlador PI, que posteriormente remplazaremos por otros tipos de controladores. El comportamiento de un controlador continuo PI (*proporcional e integral*) viene definido fundamentalmente por las dos ecuaciones siguientes:

$$\frac{dx}{dt} = \frac{error}{T}$$
$$outCtr = K*(error + x) \tag{4-1}$$

Donde *x* es la variable de estado del controlador, *error* es la diferencia entre el nivel de referencia y el valor actual del nivel obtenido del sensor, *T* es la constante de tiempo del controlador, *outCtr* es la señal de control enviada al actuador para controlar la posición de la válvula y *K* es el factor de ganancia. Estas dos ecuaciones se incluyen en la clase que modela el controlador `PIcontinuousController`, la cual extiende a la clase base `BaseController`, que será definida más adelante.

```
model PIcontinuousController
  extends BaseController(K=2,T=10);
  Real  x  "Variable de estado del controlador PI continuo";
equation
  der(x) = error/T;
  outCtr = K*(error+x);
end PIcontinuousController;
```

Integrando la primera ecuación se obtiene x. Sustituyendo en la segunda ecuación, se obtiene la expresión siguiente para la señal de control, que contiene dos términos que son directamente proporcionales a la señal de error y a la integral de la señal de error respectivamente. De ahí el nombre de controlador proporcional e integral (PI).

$$outCtr = K*(error + \int \frac{error}{T} dt) \tag{4-2}$$

Tanto el controlador PI como el controlador proporcional, integral, derivativo (PID), que será definido más adelante, heredan la clase controlador parcial `BaseController`, que contiene los parámetros en común, las variables de estado y dos conectores: uno para leer el sensor y otro para controlar el actuador de la válvula.
De hecho, la clase `BaseController` también puede ser reutilizada para definir controladores PI y PID discretos en este mismo ejemplo del sistema de un tanque en la Figura 4-3. Los controladores discretos muestrean repetidamente el nivel de liquid y producen una señal de control cuyo valor cambia en instantes de tiempo discretos con una periodicidad igual a `Ts`.

```
partial model BaseController
  parameter Real Ts(unit="s")=0.1 "Periodo de tiempo entre muestras
discretas";
  parameter Real K=2              "Ganancia";
  parameter Real T=10(unit="s")   "Constante de tiempo";
  ReadSignal     cIn    "Entrada: nivel del sensor, connector";
  ActSignal      cOut   "Control enviado al actuador, connector";
  parameter Real ref    "Nivel de referencia";
  Real           error  "Desviación respecto al nivel de referencia";
  Real           outCtr "Señal de control";
equation
  error    = ref-cIn.val;
  cOut.act = outCtr;
end BaseController;
```

Simulando el modelo `TankPI` se obtiene la misma respuesta que para el modelo `FlatTank`, lo cual no es sorprendente dado que ambos modelos tienen las mismas ecuaciones básicas (Figura 4-4).

```
simulate(TankPI, stopTime=250)
plot(tank.h)
```

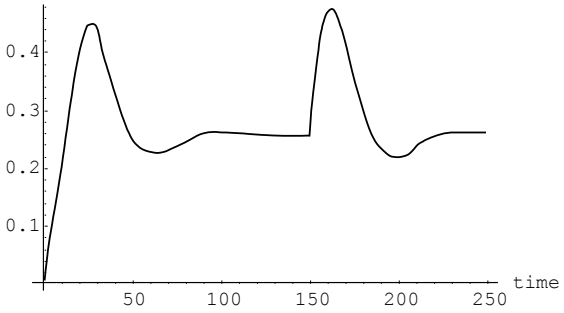



**Figura 4-4.** Respuesta del nivel del tanque para el sistema `TankPI`, que contiene un controlador `PI`.

### 4.2.4 Tanque con Controlador PID Continuo

Definimos el sistema `TankPID` de la misma manera que `TankPI`, pero con la diferencia de que el controlador PI ha sido reemplazado por un controlador PID. Aquí vemos una clara ventaja del enfoque basado en componentes orientado a objetos frente al enfoque tradicional: los componentes del sistema pueden reemplazarse de manera sencilla y ser cambiados de la forma plug-and-play (vea la Figura 4-5).

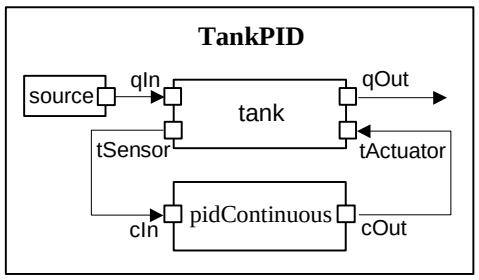



**Figura 4-5**. El sistema de un tanque en el que el controlador `PI` ha sido reemplazado por un `PID`.

La declaración en Modelica de la clase `TankPID` es como sigue:

```
model TankPID
  LiquidSource            source(flowLevel=0.02);
  PIDcontinuousController pidContinuous(ref=0.25);
  Tank                    tank(area=1);
equation
```

```
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, pidContinuous.cOut);
  connect(tank.tSensor, pidContinuous.cIn);
end TankPID;
```

Puede construirse el modelo del controlador PID (proporcional, integral, derivativo) de manera similar a como se hizo con el controlador PI. Los controladores PID reaccionan más rápidamente a los cambios instantáneos que los controladores PI, debido al término que contiene la derivada. Por otra parte, el controlador PI pone un mayor énfasis en compensar los cambios de variación lenta. Las ecuaciones básicas de un controlador PID son las siguientes:

$$\begin{aligned} \frac{dx}{dt} &= \frac{error}{T} \\ y &= T\frac{d\,error}{dt} \\ outCtr &= K*(error + x + y) \end{aligned} \tag{4-3}$$

Creamos una clase `PIDcontinuousController` en Modelica, que contiene las tres ecuaciones anteriores:

```
model PIDcontinuousController
  extends BaseController(K=2,T=10);
  Real  x; // Variable de estado del controlador PID continuo
  Real  y; // Variable de estado del controlador PID continuo
equation
  der(x) = error/T;
  y      = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

Integrando la primera ecuación, y sustituyendo *x* e *y* en la tercera ecuación, obtenemos una expresión para la señal de control que contiene los tres términos siguientes: proporcional a la señal de error, proporcional a la integral de la señal de error y proporcional a la derivada de la señal de error. Por ello, se llama controlador integral, proporcional, derivativo (PID).

$$outCtr = K*(error + \int \frac{error}{T}dt + T\frac{d\,error}{dt}) \tag{4-4}$$

Simulamos de nuevo el modelo del tanque, pero esta vez incluyendo el controlador PID (vea la Figure 4-6):

```
simulate(TankPID, stopTime=250)
```

```
plot(tank.h)
```

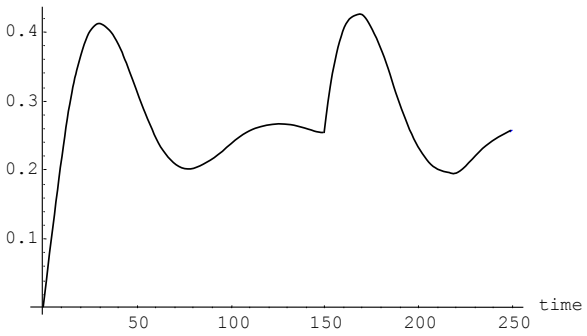



**Figure 4-6.** Respuesta del nivel del tanque para el sistema `TankPID`, que contiene un controlador PID.

La respuesta del nivel del tanque obtenida es muy similar a la que se obtuvo empleando el controlador PI, si bien el control para restaurar el nivel de referencia, tras los cambios en el flujo de entrada, es más rápido.

Con el fin de permitir establecer una comparación, en el diagrama de la Figura 4-7 se muestran los resultados de la simulación de `TankPI` y `TankPID`.

```
simulate(compareControllers, stopTime=250)
plot({tankPI.h,tankPID.h})
```

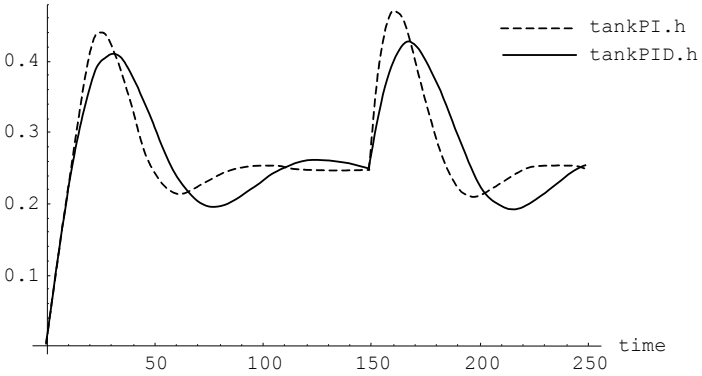



**Figura 4-7.** Comparación de las simulaciones de `TankPI` y `TankPID`.

### 4.2.5 Dos Tanques Interconectados

Las ventajas del enfoque de modelado basado en componentes orientado a objetos resultan incluso más aparentes cuando se combinan varios componentes de formas diferentes, como se muestra en la Figura 4-8, en la que se han conectado dos tanques en serie, lo cual no es infrecuente en la industria de procesos.

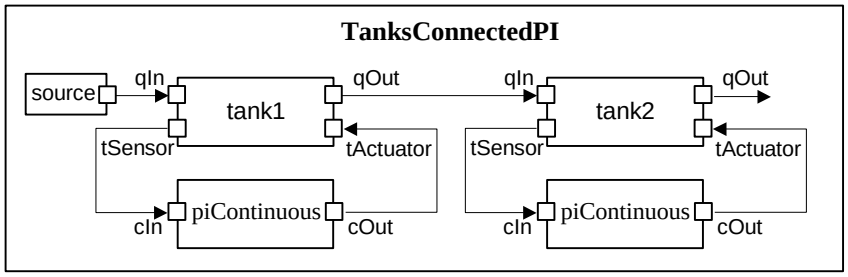



**Figura 4-8.** Dos tanques conectados, con controladores PI y una fuente de líquido conectada al primer tanque.

El modelo en Modelica `TanksConnectedPI`, correspondiente a la Figura 4-8, es el siguiente:

```
model TanksConnectedPI
  LiquidSource  source(flowLevel=0.02);
  Tank          tank1(area=1);
  Tank          tank2(area=1.3);
  PIcontinuousController piContinuous1(ref=0.25);
  PIcontinuousController piContinuous2(ref=0.4);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);
  connect(tank1.qOut,tank2.qIn);
  connect(tank2.tActuator,piContinuous2.cOut);
  connect(tank2.tSensor,piContinuous2.cIn);
end TanksConnectedPI;
```

Simulamos el sistema de dos tanques conectados. Se observan claramente las respuestas de los niveles de los tanques frente a cambios en el flujo de líquido proveniente de la fuente. Como era de esperar, la respuesta aparece primero en el tiempo en el primer tanque que en el segundo tanque (Figura 4-9).

```
simulate(TanksConnectedPI, stopTime=400)
```

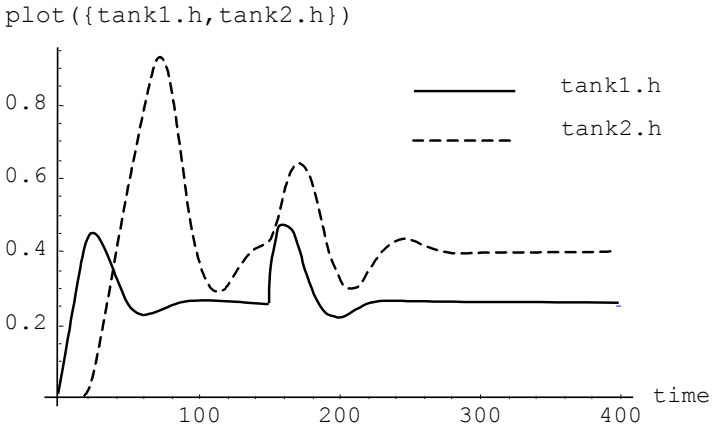



**Figura 4-9.** Respuesta de los niveles en los tanques para dos tanques conectados en serie.

## 4.3 Modelamiento descendente de un motor de corriente continua (CC) a partir de componentes predefinidos

En esta sección se ilustrará el proceso de modelado basado en componentes orientado a objetos cuando se usan clases de librerías predefinidas esquematizando el diseño de un modelo de un servo motor CC. No se entrará en detalles ya que el ejemplo previo del tanque ha sido lo suficientemente detallado.

### 4.3.1 Definición del Sistema

Que es un servo motor CC? Es un motor en el cual la velocidad puede ser controlada por alguna clase de regulador (ver la Figura 4-10). Ya que se trata de un servo, se necesita mantener una velocidad rotacional especificada a pesar de que esté sometido a una carga variable. Presumiblemente, contiene un motor eléctrico, una transmission y una carga rotacional dinámica, alguna clase de control para regular la velocidad rotacional, y algunos circuitos eléctricos ya que el sistema de control necesita conecciones eléctricas al resto del sistema, y hay partes eléctricas del motor. El lector puede haber notado que es difícil definer un sistema sin describer sus partes, es decir, ya estamos en la fase de descomposición del sistema.

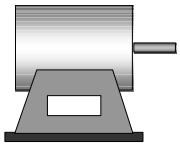


**Figura 4-10.** Un servo motor DC.

### 4.3.2 Descomposición en subsistemas y esquema de la comunicación

En esta fase se descompone el sistema en subsistemas mayors y se dibuja la comunicación entre estos subsistemas. Como ya se notó en la fase de definición del sistema, el sistema contiene partes mecánicas rotacionales que incluyen el motor y las cargas, un modelo de un circuito eléctrico que contiene las partes eléctricas del motor CC en conjunto con sus interface eléctrica, y un subsistema controllador que regula la velocidad del motor CC controlando la correinte que alimenta el motor. De esta manera, hay tres subsistemas como se esquematiza en la Figura 4-11: controlador, un circuito eléctrico, y un subsistema mecánico rotacional.

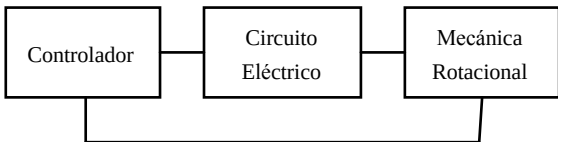



**Figura 4-11.** Subsistemas y sus conecciones.

Con respecto a las comunicaciones entre los subsistemas, el controlador debe estar conectado al circuito eléctrico ya que controla la corriente al motor. El circuito eléctrico también se debe conectar a las partes mecánicas rotacionales con el objetivo de que la energía eléctrica pueda convertirse a energía rotacional. Finalmente, se necesita un lazo de realimentación que incluya un sensor de la velocidad rotacional para que el controlador pueda hacer su tarea apropiadamente. Los enlaces de comunicación se dibujan en la Figura 4-11.

### 4.3.3 Modelamiento de los Subsistemas

La siguiente fase es modelar los subsistemas a través de descomposiciones adicionales. Se empieza por modelar el controlador y luego nos dirigimos a buscar clases en la librería de Modelica Standar para un nodo suma de realimentación y un controlador tipo PI. También se adiciona un bloque de función escalón como un ejemplo de una señal de referencia. Todas estas partes se muestran en la Figura 4-12.

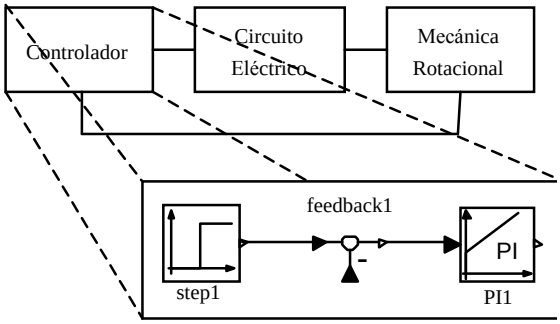



**Figura 4-12.** Modelamiento del Controlador.

El Segundo componente mayor a descomponer es la parte del circuito eléctrico del motor CC (ver Figura 4-13). Aquí se identificaron las partes estándar de un motor CC tales como un generador de voltaje eléctrico controlado por señal, una componente de tierra necesaria en todos los circuitos eléctricos, una resistencia, una inductancia que representa la bobina del motor, y una fuerza electromotriz (efm) para convertir energía eléctrica a movimiento rotacional.

El tercer subsistema, diagramado en la Figura 4-14, contiene tres cargas rotacionales con inercia: un engranaje ideal, un resorte rotacional, y un sensor de velocidad para medir la velocidad rotacional el cual se necesita como información para el controlador.

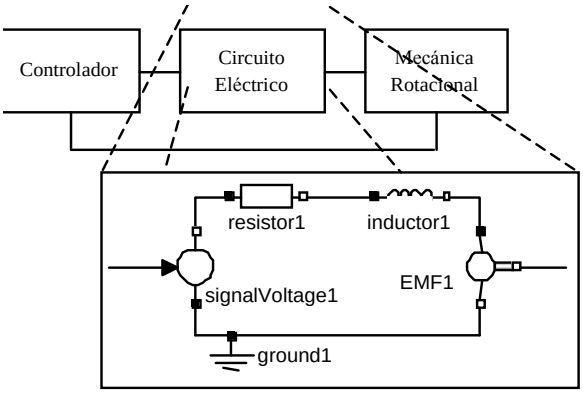



**Figura 4-13.** Modelamiento del circuito eléctrico.

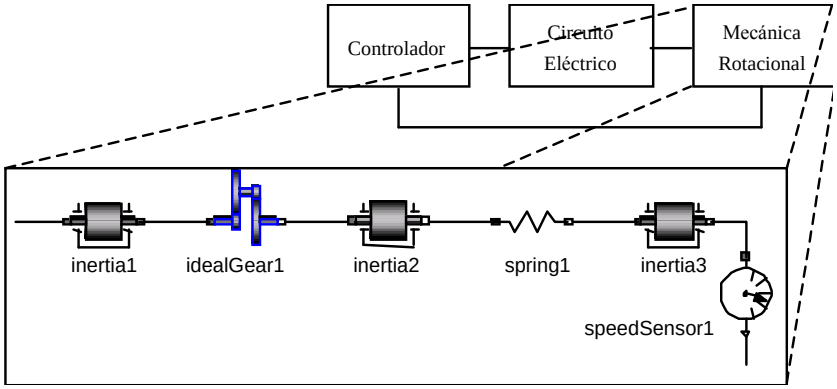



**Figura 4-14.** Modelamiento del subsistema mecánico incluyendo el sensor de velocidad.

### 4.3.4 Modelamiento de las Partes en los Subsistemas

Nos dirigimos a buscar todas las partes necesarias como modelos predefinidos en la librería de clases de Modelica. Si ese no fuera el caso, tendríamos la necesidad de definer clases apropiadas de modelos e identificar las ecuaciones para esas clases, como se ha esquematizado para las partes del subsistema de control en la Figura 4-15, el subsistema eléctrico en la Figura 4-16, y el subsistema mecánico rotacional en la Figura 4-17.

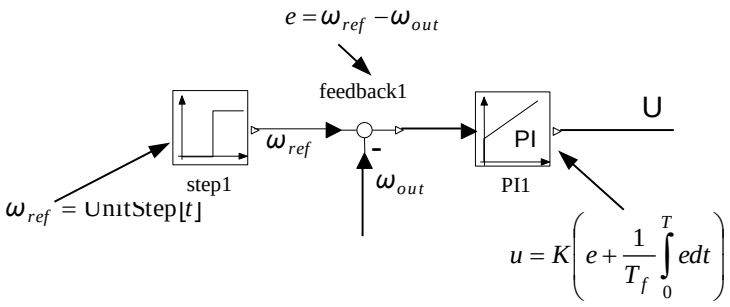



**Figura 4-15.** Ecuaciones Básicas y componentes en el subsistema de control.

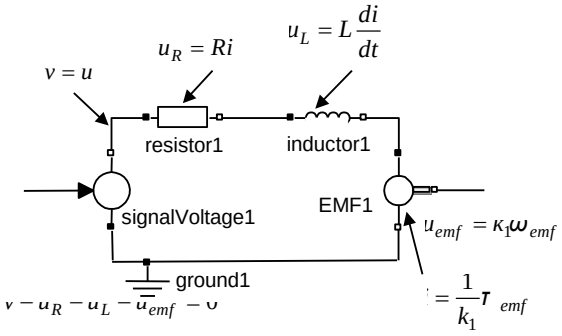



**Figura 4-16.** Definición de las clases y ecuaciones básicas para los componentes en el subsistema eléctrico.

El subsistema eléctrico representado en la Figura 4-16 contiene componentes eléctricos con ecuaciones básicas asociadas, por ejemplo, una resistencia, una inductancia, una fuente de señal de voltaje, y un componente emf.

El subsistema mecánico rotacional representado en la Figura 4-17 contiene un número de componentes tales como inercias, un engranaje, un resorte rotacional, y un sensor de velocidad.

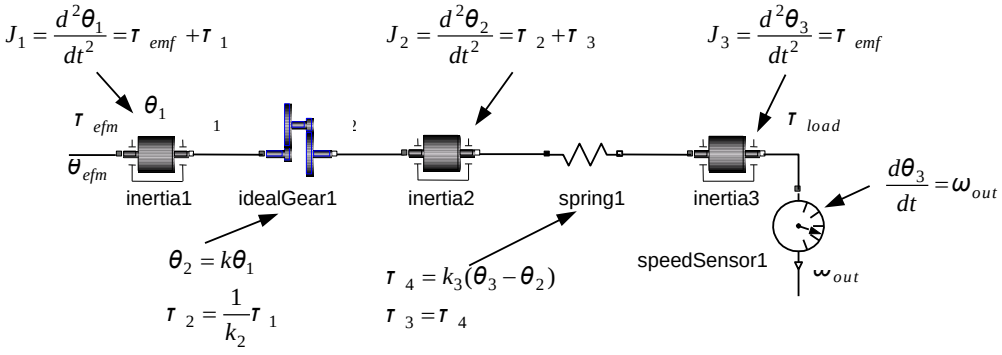



**Figura 4-17.** Definición de clases y ecuaciones básicas para componentes en el subsistema mecánico rotacional.

### 4.3.5 Definición de las interfaces y las conecciones

Cuando cada subsistema ha sido definido a partir de modelos predefinidos, se establecen las interfaces a los subsistemas a través de los conectores de estos componentes que interactúan con otros subsistemas. Cada subsistema debe definirse de tal manera que se habilite la comunicación con otros subsistemas de acuerdo a la estructura de comunicación previamente esquematizada. Esto requiereque se escojan cuidadosamente las clases de conectores para que sean de tipos compatibles. De hecho, la selección y definición de estas interfaces de conectores es uno de los pasos mas importantes en el diseño de un modelo.

El modelo completado del servo motor CC se representa en la Figura 4-18, la cual incluye los tres subsistemas y el lazo realimentado.

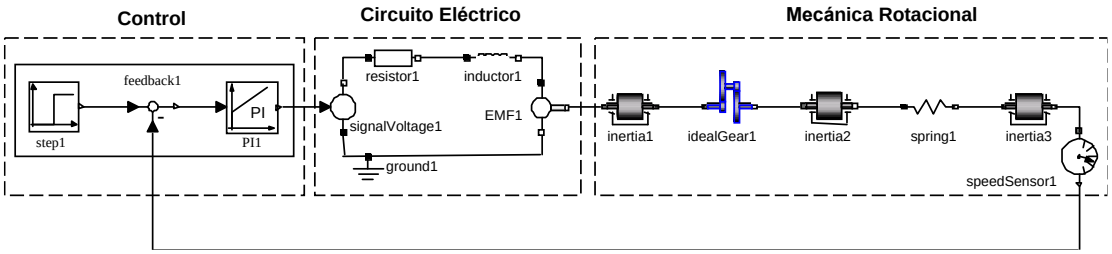



**Figura 4-18.** Finalización de las interfaces y conecciones entre los subsistemas, incluyendo el lazo de realimentación.

### 4.3.6 **Diseño de clases de interfaces-conectores**

Como en todo diseño de un sistema, una de las tareas mas importantes de diseño es definir las interfaces entre los componentes de un modelo del sistema, ya que esto pone los cimientos para la comunicación entre los componentes. De esta manera influencian fuertemente la descomposición del modelo.

Lo mas importante es identificar los *requerimientos* básicos detras del diseño de las interfaces de los componentes, es decir, las clases de conectores que influencian su estructura. Estos requerimientos puen establecerse brevemente como sigue:

- Debe ser *facil* y *natural* conectar componentes. Para interconectar a modelos de componentes físicos debe ser físicamente posible conectar estos componentes.
- Las interfaces de los componentes deben faciltar la *reutilización* de componentes de modelos existentes en las librerías de clases.

Se requiere de un diseño cuidadoso de las clases conector para alzanzar estos objetivos en forma satisfactoria. El número de clases conectores debe permanecer pequeño con el objetivo de evitar incompatibilidades innecesarias entre conectores debido a nombres y tipos diferentes de variables en los conectores.

La experiencia muestra que es sorprendentemente difícil diseñar clases conectores que satisfagan estos requerimientos. Hay una tendencia para que detalles superfluos que se crean durante el desarrollo del software (modelo) para diferentes necesidades computacionales entran sigilosamente a las interfaces, haciéndolas mas difíciles de usar e impidiendo el reuso de los componentes. Por lo tanto, las clases conectores deben permanecer tan simples como sea posible e intentar evitar introducir variables que no son realmente necesarias.

Una buena regla general cuando se diseñan clases conectores para modelos de componentes físicos es identificar las características de interacción (no causal) en el mundo físico real entre estos componentes. Las características de interacción deben ser simplificadas y abstraidas a un nivel apropiado y reflejarlas en el diseño de las clases conectores. Para componentes no físicos, por ejemplo, bloque de señal y componentes de software en general, se tiene que trabajar duro en encontrar el nivel de abstracción apropiado en las interfaces, e intentar probarlas en la práctica para obtener realimenteación en la facilidad de uso la reusabilidad. La librería estandar de Modelica contiene una gran número de clases conectores bien diseñados que pueden servir como inspiración a los diseñadores que estén diseñando en nuevas interfaces.

Hay básicamente tres clases diferentes, que reflejan tres situaciones de diseño:

1. Si hay alguna clase de interacción entre dos componentes *físicos* que involucran *flujo de energía*, se debe usar una variable de potencial y una variable tipo `flow` en el dominio apropiado para la clase conector.
2. Si se intercambia información o *señales* entre componentes, se deben usar variables de señal tipo `input/output` en la clase conector.
3. Para interacciónes complejas entre componentes, que involucren varias interacciones como las anteriores tipo 1 y 2, se diseña una clase de *conector compuesto* jerarquicamente estructurado del tipo apropiado 1, 2 o 3.

Cuando todos los conectores de un componente han sido diseñados de acuerdo a los tres principios citados arriba, la formulación del resto de la clase de componentes sigue parcialmente a partir de las restricciones implicadas por estos conectores. Sin embargo, estas pautas no deben seguirse ciegamente. Hay varios ejemplos de dominios con condiciones especiales que se desvían ligeramente de las reglas mencionadas.

## 4.4 Resumen

En este capítulo se ha presentado cómo llegar a modelos matemáticos de los sistemas en los cuáles estamos interesados usando un proceso basado en componentes orientado a objetos. El modelaje de sistemas se ilustró con un ejemplo de dos tanques, tanto usando el enfoque plano como el enfoque orientado a objetos.

## 4.5 Literatura

Los principios generales para el modelaje y el diseño orientado a bojetos son descritos en Rumbaugh (1991) y Booch (1991). Una discusión general de principios de diseño de modelos orientado a bloques se puede encontrar en Ljung y Glad (1994), mientras que en Anderson (1994) se describe el modelaje matemático orientado a objetos con alguna profundidad.

Muchos conceptos y términos en ingeniería de software y en modelamiento/simulación están descritos en el glosario de ingeniería de software estandar: (IEEE Std 610.12-1990) en conjunto con el glosario estandar IEEE de modelamiento y simulación (IEEE Std 610.3-1989).

# Capítulo 5

# La Librería Estandar de Modelica

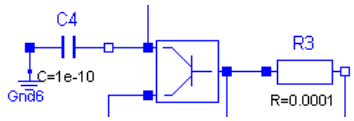
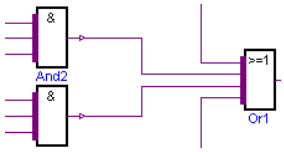
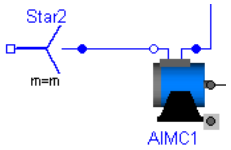
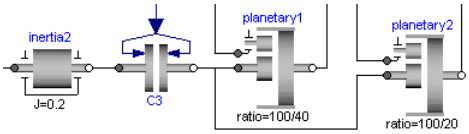
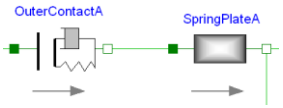
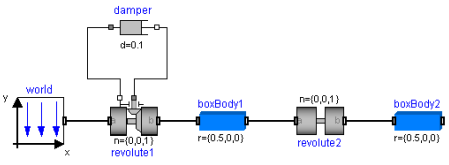
Gran parte de la potencia del modelado con Modelica proviene de la facilidad de reusar las clases de modelos. Las clases relacionadas en áreas particulares se agrupan en paquetes para que sean más fáciles de encontrar. En este capítulo se ofrece una descripción rápida de algunos paquetes communes en Modelica.

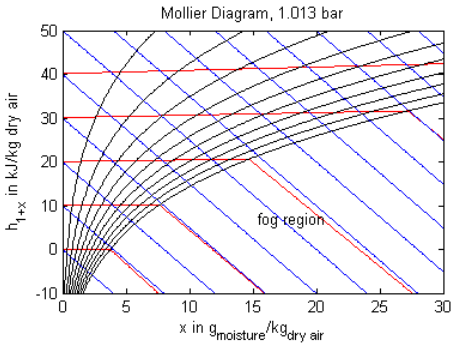
Un paquete especial, llamado `Modelica`, es un paquete predefinido estandarizado que junto con el lenguaje Modelica es desarrollado y mantenido por Modelica Association. Este paquete es también conocido como el *Modelica Standard Library*. Proporciona constantes, tipos, clases de conectores, modelos parciales, y clases de modelos de componentes de varias áreas de aplicación, las cuales son agrupadas en subpaquetes del paquete `Modelica`.

La librería Standar de Modelica puede ser usada libremente para fines comerciales y no comerciales bajo las condiciones de la Licencia de Modelica como se indicó en las primeras páginas de este libro. La documentación completa así como el código fuente de estas librerías aparece en la página web de Modelica: `http://www.modelica.org/library/` Estas librerías se incluyen con frecuencia en la distribución de herramientas de Modelica.

La version 3.1 de la librería estándar de Modelica de agosto de 2009 contiene cerca de 920 modelos y bloques, y 610 funciones en las sublibrerías listadas en la Tabla 5-1.

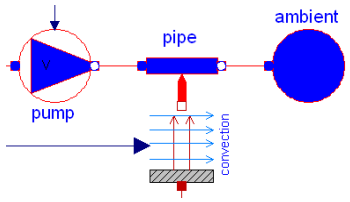
**Tabla 5-1.** Principales sublibrerías de la Librería Estándar de Modelica version 3.1.

| | |
|---|---|
|  | **Modelica.Electrical.Analog**<br>Componentes eléctricos análogos y electrónicos tales como resistencias, capacitancias, transformadores, diodos, transistores, líneas de transmisión, interruptores, fuentes y sensores. |
|  | **Modelica.Electrical.Digital**<br>Componentes electricos digitales basados en VHDL (IEEE 1999) con nueve valores lógicos. Contiene retardos, compuertas, fuentes, y convertidores entre 2-, 3-, 4-, y 9- valores lógicos. |
|  | **Modelica.Electrical.Machines**<br>Máquinas eléctricas no controladas, tales como motores y generadores síncronos, asíncronos y de corriente continua. |
|  | **Modelica.Mechanics.Rotational**<br>Sistemas mecánicos rotacionales en una dimensión, tales como dispositivos de transmission, engranaje planetario. Contiene inercias, amortiguaciones, caja de cambios, fricción en rodamientos, embrague, frenos, backlash, torque, otros. |
|  | **Modelica.Mechanics.Translational**<br>Sistemas mecánicos translacionales en una dimension tales como masa, topes, amortiguadores, backlash, fuerza. |
|  | **Modelica.Mechanics.MultiBody**<br>Sistemas mecánicos en tres dimensiones consistentes de articulaciones, cuerpos, elementos de fuerza y sensores, bodies, fuerzas and sensores. Las articulaciones pueden ser accionadas por los elementos de la librería Rotational. Todos los elementos tienen una animación por defecto. |

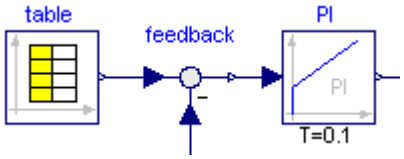



**Modelica.Media**
Una gran librería de medios para fluídos de sustancias simples y multiples con una o varias fases:

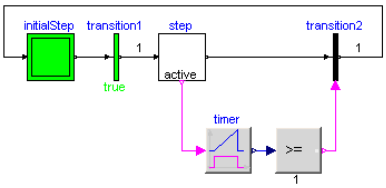
- Modelos de gas de alta precision basados en los coeficientes de Glenn de la NASA+mezcla entre estos modelos de gases.
- Modelos de agua simples y de alta precisión (IAPWS/IF97)
- Modelos de aire seco y húmedo.
- Medios incompresibles basados en tabla.
- Modelos simples de líquidos con compresibilidad lineal.




**Modelica.Thermal**
Flujo simple en tuberías para termofluídos, especialmente para sistemas de refrigeración con agua o con fluido de aire. Contiene tuberías, bombas, válvulas, sensors, Fuentes, otros. Además, agrupa los componentes de transferencia de calor que están presentes tales como condensador de calor, conductores térmicos, convección, radiación del cuerpo, otros.




**Modelica.Blocks**
Bloques de entrada/salida continuos y discretos. Contiene funciones de transferencia, sistemas en espacio de estado lineales, no lineales, matemáticos, lógicos, tablas, bloques de fuentes.




**Modelica.StateGraph**
Diagramas jeráquicos de estado con poder de modelado similar a Statecharts. Modelica es usado como un lenguaje de "acción" sincrono. El comportamiento determinístico está garantizado.

```
import Modelica.Math.Matrices;
A = [1,2,3;
```

**Modelica.Math.Matrices / Modelica.Utilities**
Funciones con operación de matrices, por ejemplo,

| | |
|---|---|
| `3,4,5;`<br>`2,1,4];`<br>`b = {10,22,12};`<br>`x = Matrices.solve(A,b);`<br>`Matrices.eigenValues(A);` | para resolver sistemas lineales y calcular valores propios y singulares. Por otra parte se proporcionan funciones para operar en cadenas, flujos y archivos. |
|  <br>`type Angle = Real (`<br>`final quantity = "Angle",`<br>`final unit = "rad",`<br>`displayUnit = "deg");` | **Modelica.Constants, Modelica.Icons, Modelica.SIunits**<br>Librería de utilidades para propocionar:<br>• constantes usadas a menudo como e, π, R<br>• una librería de íconos que puede ser usada en modelos.<br>• Cerca de 450 tipos predefinidos, como Masa, Angulo, Tiempo, basado en el sistema internacional de unidades. |

Un subpaquete denominado `Interfaces` pasa como parte de varios paquetes. Contiene las *definiciones de interface* en terminos de clases de conectores para el área de aplicación cubierta por el paquete en cuestión, así como *clases parciales comunmente usadas* para ser reusada por los otros subpaquetes dentro de ese paquete.

También, un subpaquete `Examples` contiene *modelos de ejemplo* sobre cómo usar las clases de los paquetes en cuestión que se presenta en muchas librerías. Algunas librerías con sublibrerías se muestran en la Table 5-2.

**Table 5-2.** Librerías seleccionadas con ejemplos de las sublibrerías `Interfaces` y `Examples`.

| | |
|---|---|
| `Modelica` | Librería estandar de Modelica Association incluyendo las siguientes sublibrerías: |
| `Blocks`<br>`Interfaces`<br>`Continuous`<br>`...` | Bloques de entrada/salida para usar en diagramas de bloques.<br>Sublibrería de Interfaces a Bloques.<br>Bloques de control continuo con estados internos |
| `Electrical` | Modelos de componentes eléctricos comunes |

| | |
|---|---|
| | Modelo de components eléctricos análogos |
| Analog | Sublibrerías eléctricas análogas. |
| Interfaces, Basic, Ideal, | Sublibrerías eléctricas análogas. |
| Sensors, Sources, | Sublibrerías eléctricas análogas. |
| Examples, | Sublibrerías eléctricas análogas. |
| Lines, Semiconductors | |
| | Componentes eléctricos digitales. |
| Digital | |
| ... | |
| | Librería mecánica general. |
| Mechanics | Modelos de componentes mecánicos rotacionales en 1D. |
| Rotational | Sublibrerías rotacional. |
| Interfaces, Sensors, | Componentes mecánico translacional en 1D. |
| Examples, ... | |
| | Sublibrerías translacional |
| Translational | Sistemas mecánicos en 3D – librería multicuerpo. |
| Interfaces, Sensors, | Sublibrerías multicuerpo. |
| Examples | Sublibrerías multicuerpo. |
| | Sublibrerías multicuerpo. |
| ... | ... |

Hay también un número de librerías gratis en Modelica que no son parte de la librería estándar, y no han sido "certificadas" por la Asociación de Modelica. La calidad de estas librerías es variable. Algunas están bien documentadas y probadas, pero este no es el caso de algunas otras librerías. El número de librerás disponibles en el sitio web de la Asociación de Modelica está creciendo rápidamente. La Tabla 5-3 es una muestra del panorama de estas librerías a septiembre de 2009.

**Tabla 5-3.** Selección de librerías adicionales gratis en Modelica..

| | |
|---|---|
| ModelicaAdditions: | Colección Antigua de librerías adicionales de Modelica: Blocks, PetriNets, Tables, HeatFlow1D, MultiBody. |
| Blocks | Bloques adicionales de entrada/salida. |
| Discrete | Bloques de entrada/salida con periodo de muestreo fijo. |

| | |
|---|---|
| Logical | Bloques booleanos de entrada/salida |
| Multiplexer | Combina y divide conectores de señal del tipo Real. |
| PetriNets | Redes de Petri y diagramas de transición de estados. |
| Tables | Tabla de consulta en una y dos dimensiones |
| HeatFlow1D | Flujo de calor en una dimensión. |
| MultiBody | Sistemas mecánicos en 3D – Antigua librería de MBS con algunas restricciones de conexión y la manipulación manual de lazos cinemáticos. |
| Interfaces, Joints, CutJoints | Antigua sublibrerías de MBS |
| Forces, Parts, Sensors, | Antigua sublibrerías de MBS |
| Examples | Antigua sublibrería de MBS |
| SPOT | Sistemas de potencia en modo transitorio y estado estacionario, 2007 |
| ExtendedPetriNets | Librería de redes de Petri extendidas, 2002 |
| ThermoFluid | Antigua librería (reemplazada) de termodinámica y termohidráulica, plantas de vapor, y sistemas de procesos. |
| SystemDynamics | Dinámica de sistemas a la J. Forrester, 2007. |
| QSSFluidFlow | Flujo de fluídos en estado cuasi estacionario. |
| Fuzzy Control | Librería de control fuzzy. |
| VehicleDynamics | Dinámica de chasis de vehículos (obsoleta), 2003 (reemplazado por una librería comercial) |
| NeuralNetwork | Modelos matemáticos de redes neuronales, 2006 |
| WasteWater | Plantas de tratamiento de aguas residuales, 2003 |
| ATPlus | Construcción, simulación y control (control fuzzy incluido), 2005 |
| MotorCycleDymanics | Dinámica y control de moyocicletas, 2009 |
| SPICElib | Algunas capacidades del simulador de circuitos eléctricos PSPICE, 2003 |
| BondLib | Modelado de sistemas físicos en gráfos de Bond, 2007 |
| MultiBondLib | Modelado de sistemas físicos con gráfos MultiBond, 2007 |

| | |
|---|---|
| `ModelicaDEVS` | Modelado a eventos discretos DEVS, 2006 |
| `External.Media Library` | Cálculo de propiedades externas de los fluídos, 2008 |
| `VirtualLabBuilder` | Implementación de laboratorios virtuales, 2007 |

Algunas librerías comerciales, por lo general no gratuitas también están disponibles.

Al desarrollar una aplicación o una librería en algunas areas de aplicación es aconsejable usar los tipos de cantidades disponibles en `Modelica.SIunits` y los conectores estándar disponibles en el correspondiente subpaquete `Interfaces`, por ejemplo, `Modelica.Blocks.Interfaces` de la Librería estándar de Modelica, para que los modelos basados en componentes de la misma abstracción física tengan interfaces compatibles y puedan ser conectados en conjunto.

En la Tabla 5-4 se definen clases de conectores elementales donde las variables de potencial son variables conectoras sin el prefijo `flow` y las variables de flujo tienen con el prefijo `flow`:

**Tabla 5-4.** Clases de conectores básicos para librerías de Modelica comunmente usadas.

| *Dominio* | *Variables de potencial* | *Variables de flujo* | *Definición del conector* | *Iconos* |
|---|---|---|---|---|
| análogo eléctrico | potencial eléctrico | corriente eléctrica | Modelica.Electrical.Analog.Interfaces .Pin, .PositivePin, .NegativePin |  |
| Polifásico electrico | vector de pines eléctricos | | Modelica.Electrical.MultiPhase.Interfaces .Plug, .PositivePlug, .NegativePlug |  |
| Fasor espacial eléctrico | 2 potenciales eléctricos | 2 corrientes eléctricas | Modelica.Electrical.Machines.Interfaces SpacePhasor |  |
| digital eléctrico | Entero (1..9) | --- | Modelica.Electrical.Digital.Interfaces .DigitalSignal, .DigitalInput, DigitalOutput |  |
| translacional | distancia | Fuerza de corte | Modelica.Mechanics.Translational. Interfaces.Flange_a, .Flange_b |  |
| Rotacional | angulo | Torque de Corte | Modelica.Mechanics.Rotational.Interfaces .Flange_a, .Flange_b |  |
| mecánica en 3 dimensiones | vector posición orientacion objeto | Vector de Fuerza de Corte Vector de Torque de | Modelica.Mechanics.MultiBody.Interfaces .Frame, .Frame_a, .Frame_b, .Frame_resolve |  |

| | | | | |
|---|---|---|---|---|
| | | Corte | | |
| flujo de fluidos simple | presión entalpía especifica | Rata de flujo másico rata de flujo entálpico | Modelica.Thermal.FluidHeatFlow.Interfaces .FlowPort, .FlowPort_a, .FlowPort_b |  |
| transferencia de calor | temperatura | rata de flujo calórico | Modelica.Thermal.HeatTransfer.Interfaces .HeatPort, .HeatPort_a, .HeatPort_b |  |
| Diagrama de bloques | Real, Entero, Booleano | | Modelica.Blocks.Interfaces .RealSignal, .RealInput, .RealOutput .IntegerSignal, .IntegerInput, .IntegerOutput .BooleanSignal, .BooleanInput, .BooleanOutput |  |
| Máquina de estados | Variables Booleanas (occupied, set, available, reset) | | Modelica.StateGraph.Interfaces .Step_in, .Step_out, .Transition_in, .Transition_out |  |
| Flujo de termofluido | presión | Velocidad del flujo másico | Modelica_Fluid.Interfaces .FluidPort, .FluidPort_a, .FluidPort_b |  |
| | *Variables de flujo* (si $m_{flow} < 0$): entalpia específica, Fracción de masa ($m_i/m$) Fracción de propiedad extra ($c_i/m$) | | | |
| Magnetico | Potencial magnetico | flujo magnetico | Magnetic.Interfaces .MagneticPort, .PositiveMagneticPort, .NegativeMagneticPort |  |

En todos los dominios dos conectores equivalentes son usualmente definidos, por ejemplo, `DigitalInput`—`DigitalOutput`, `HeatPort_a`—`HeatPort_b`, etc. Las declaraciones de las variables de estos conectores son *idénticas*, solamente los íconos son diferentes para que sea fácil distinguir los dos conectores del mismo dominio que están ligados al mismo componente del modelo.

## 5.1 Resumen

Este capítulo ha descrito muy brevemente la estructura de la librería de Modelica, version 3.1, tal como se presenta en la Especificación del Lenguaje Modelica 3.2 y en la página web de la Asociación Modelica al momento de este escrito, incluyendo la librería estándar Modelica. El conjunto de librerías disponibles está crecieno rápidamente, sin embargo, las actuales sublibrerías estándar de Modelica están bien probadas y hasta ahora la mayoría se han sometido a pequeñas mejoras evolutivas.

## 5.2 Literatura

Todas las librerías gratuitas de Modelica descritas aquí, incluyendo la documentación y el código fuente, se pueden encontrar en el sitio web de la Asociación Modelica, `www.modelica.org`. La documentación para varias librerías comerciales está también disponible en el sitio web de Modelica.

La referencia más importante para este capítulo es el Capítulo 19 en la especificación del lenguaje Modelica (Modelica Association 2010), del cual las Tabla 5-1 y Tabla 5-4 han sido reusadas. Estas tablas fueron creadas por Martin Otter.

# Apéndice A

# Glosario

**sección algoritmo**: parte de una definición de clase que consiste de la palabra clave `algorithm` seguida por una secuencia de instrucciones. Al igual que una ecuación, una sección algoritmo relaciona variables, es decir, restringe los valores que estas variables pueden tomar simultáneamente. En contraste con una sección ecuación, una sección algoritmo distingue las entradas de las salidas: una sección algoritmo especifica cómo calcular las variables de salida como una fucnión de las variables de entrada. (Ver Seccion 2.14).

**arreglo o variable tipo arreglo**: variable que contiene elementos tipo arreglo. Para un arreglo, el ordenamiento de sus elementos importa: el k-ésimo elemento en la secuencia de elementos de un arreglo x es el elemento del arreglo con índice `k`, denotado como `x[k]`. Todos los elementos de una arreglo tienen el mismo tipo. Un elemento del arreglo puede ser nuevamente un arreglo, es decir, los arreglos se pueden anidar. Un elemento de un arreglo es por lo tanto referenciado utilizando *n* índices en general, donde *n* es el número de dimensiones del arreglo. Los casos especiales son las matrices (*n* = 2) y vectores (*n* = 1). Los índices enteros del arreglo comienzan por uno, nunca por cero, es decir, el límite inferior es 1. (Ver la Sección 2.13)

**constructor de arreglos**: un arreglo puede construirse utilizando la función array – utiliando como comando simplificado los corchetes {a, b, ...}, y también puede incluir un iterador para construir un arreglo de expresiones. (Ver la Sección 2.13)

**elemento de un arreglo**: elemento contenido en un arreglo. Un elemento de un arreglo no tiene identificador. En su lugar los elementos de un arreglo se referencian a través de expresiones de acceso a un arreglo llamados índices que utilizan valores de la enumeración o valores de índice enteros positivos. (ver la Sección 2.13)

**asignamiento**: una declaración de la forma `x := expr`. La expresión `expr` no debe tener mayor variabilidad que x. (Ver la Sección 2.14.1)

**atributo**: una propiedad (o tipo de campo de registro) contenido en una variable escalar, tal como `min`, `max`, y `unit`. Todos los atributos están predefinidos y los valores de dichos atributos sólo se pueden definir mediante una modificación, como por ejemplo en `Real x(unit="kg")`. Los atributos no se puede acceder usando la notación de punto, y no está restringidos por secciones ecuación ó algoritmo. Por ejemplo, en `Real x(unit="kg") = y;` solamente los valores de `y` y `x` son declarados para que sean iguales, pero no sus atributos de unidad, ni ningún otro atributo de `x` ó `y`. (Ver Sección 2.3.1)

**clase base**: se llama a la Clase A una clase base de B, si la clase B extiende la clase A. Esta relación se especifica mediante una cláusula `extends` en B o en una de las clases base de B. Una clase hereda todos los elementos de sus clases base, y puede modificar todos los elementos no finales heredados de las clases base. (Ver sección 3.7)

**ecuación vinculante**: ya sea una ecuación de declaración o una modificación de elementos para el valor de la variable. Una variable con una ecuación vinculante tiene su valor fijado a alguna expresión. (Ver Sección 2.6)

**clase**: una descripción que genera un objeto llamado instancia. La descripción consiste en una definición de clase, un ambiente de modificación que modifica la definición de clase, una lista opcional de expresiones de dimensión, si la clase es una clase de arreglo, y una clase lexicamente cerrada para todas las clases. (Ver Sección 2.3)

**tipo clase** or **interface de herencia**: propiedad de una clase, que consiste en una serie de atributos y un conjunto de elementos públicos o protegidos constituidos por: nombre del elemento, tipo de elemento y los atributos de los elementos.

**asignamiento de declaración**: Asignación de la forma x := expresión definida por una declaración de variable en una función. Esto es similar a una ecuación de declaración, pero diferente puesto que una variable asignada, por lo general se puede asignar varias veces. (ver la Sección2.14.3)

**ecuación de declaración**: La ecuación de la forma x = expresión definida por una declaración de componentes. La expresión no debe tener mayor variabilidad que la componente declarada x. A diferencia de otras ecuaciones, una ecuación de declaración puede ser anulada (sustituida o eliminada) por una modificación del elemento. (Ver Sección 2.6)

**clase derivada** o **subclase**: a la clase B se le llama clase derivada de A, si B extiende a A. (Ver Sección 2.4)

**elemento**: parte de una definición de una clase, puede ser: una definición de la clase, una declaración de componentes, o una cláusula extendida. La declaración de componentes y las definiciones de clases se denominan elementos nombrados. Un elemento puede ser o heredado de una clase base o local.

**Modificación de un elemento**: parte de una modificación, anula la ecuación de declaración en la clase usada por la instancia generada por el elemento modificado. Ejemplo: `vcc(unit="V")=1000`. (Véanse las secciones 2.6 y 2.3.4)

**Redeclaración de elemento**: parte de una modificación, sustituye a uno de los elementos nombrados posiblemente utilizados para construir la instancia generada por el elemento que contiene la redeclaración. Ejemplo: `redeclare type Voltage = Real(unit="V")` remplaza `type Voltage`. (Ver Sección 2.5)

**encapsulado**: un prefijo de clase que hace que la clase no depende de dónde se coloca en el jerarquía de paquetes, ya que su búsqueda se detiene en el límite de la clase. (Ver la Sección 2.16).

**equación**: una relación de igualdad que forma parte de una definición de clase. Una ecuación escalar que relaciona variables escalares, es decir, limita los valores que estas variables pueden tomar al mismo tiempo. Cuando las n-1 variables de una ecuación que contiene n variables son conocidas, el valor de la n-ésima variable se puede inferir (resolver). En contraste con una declaración en una sección algoritmo, una ecuación no define para cuál de sus variables va a ser resuelta. Casos especiales son: ecuaciones iniciales, ecuaciones, ecuaciones instantáneas, ecuaciones de declaración. (Ver la Sección 2.6)

**evento**: algo que se produce de forma instantánea en un momento determinado o cuando ocurre una condición específica. Los eventos son, por ejemplo, definidos por la condición que se produce en una cláusula when, en una cláusula if, o en una expresión if. (Ver la Sección 2.15)

**expresión**: un término construido a partir de los operadores, las referencias de función, las variables / constantes con nombre o referencias variables (refiriéndose a variables) y literales. Cada expresión tiene un tipo y una variabilidad. (Vea las secciones 2.1.1 and 2.1.3)

**clausula de extensión**: un elemento sin nombre de una definición de clase que utiliza un nombre y una modificación opcional para especificar la herencia de una clase base de la clase definida mediante la definición de clase. (Ver Sección 2.4)

**aplanamiento**: computación que crea una clase aplanada de una clase dada, donde todas las herencias, modificaciones, etc han sido realizadas y resuelto todos los nombres, consiste en un

conjunto de ecuaciones planas, secciones algoritmo, declaraciones de componentes y funciones. (Ver la Sección 2.20.1)

**función**: una clase de la función de clase especializada. (Ver la Sección 2.14.3)

**subtipo de función**: la clase A es un subtipo de función de B si y sólo si A es un subtipo de B y los parámetros adicionales formales de la función A que no están en la función B se definen de tal manera que A se puede llamar en los lugares donde B es llamada (por ejemplo, parámetros formales adicionales que tienen valores por defecto). Para mas información, ver el Capítulo 3 de Fritzson (2004) o Fritzson (2012).

**identificador**: un nombre atómico (no compuesto). Ejemplo: `Resistor` (Ver sección 2.1.1 para nombres de variables)

**índice** o **subíndice**: Una expresión, tipicamemente del tipo entero o el símbolo de dos puntos (:), usado para referenciar un elemento (o rango de elementos) de un arreglo. (Ver la Sección 2.13)

**interface de herencia** o **tipo de clase**: propiedad de una clase, que consiste en una serie de atributos y un conjunto de elementos públicos o protegidos constituidos por un nombre de elemento, un tipo de elemento y los atributos de los elementos. (Ver la Sección 2.9 y la Sección 2.4)

**instancia**: el objeto generado por una clase. Una instancia contiene cero o más componentes (es decir, instancias), ecuaciones, algoritmos, y clases locales. Una instancia tiene un tipo. Básicamente, dos instancias tienen el mismo tipo, si sus atributos importantes son los mismos y sus componentes y clases públicas tienen identificadores y tipos igualmente comparables. Definiciones de tipos más específicos de equivalencia se dan por ejemplo, para las funciones. (Ver la Sección 2.3)

**instantanea**: una ecuación o enunciado es instantáneo si se cumple sólamente en los eventos, es decir, en puntos singulares en el tiempo. Las ecuaciones y las declaraciones de una cláusula tipo when son instantáneos. (Ver la Sección 2.15)

**literal**: un valor constante real, entero, booleano, enumeración o cadena, es decir, un literal. Se utiliza para construir expresiones. (Ver Sección 2.1.3)

**matríz**: un arreglo donde el número de dimensiones es 2. (Ver Sección 2.13)

**modification**: parte de un elemento. Modifica la instancia generada por ese elemento. Una modificación contiene modificaciones de elementos y redeclaraciones de elementos. (Ver la Sección 2.3.4)

**nombre**: Secuencia de uno o más identificadores. Se utiliza para hacer referencia a una clase o una instancia. Un nombre de clase se resuelve en el ámbito de una clase, que define un conjunto de clases visibles. Ejemplo nombre: "`Ele.Resistor`". (Ver Secciónes 2.16 y 2.18)

**registro de operación**: Un registro con operaciones definidas por el usuario, por ejemplo, la definición de multiplicación y adición. (Ver la Sección 2.14.4)

**parcial**: una clase que está incompleta y que no puede ser instanciada; por ejemplo, útil como una clase base (Ver Sección 2.9)

**tipo predefinido**: uno de los tipos `Real`, `Boolean`, `Integer`, `String` y tipos definidos como tipos `enumeration`. Las declaraciones de atributos de los tipos predefinidos, definen atributos tales como `min`, `max`, y `unit`. (Ver Sección 2.1.1)

**prefijo**: propiedad de un elemento de una definición de clase la cual pueda estar presente o no, por ejemplo `final`, `public`, `flow`.

**redeclaración**: un modificador con la palabra clave `redeclare` que cambia un elemento remplazable (Ver Sección 2.5)

**reemplazable**: un elemento que puede ser remplazado por un elemento diferente que tiene un tipo compatible. (Ver Sección 2.5)

**subtipo restringido**: un tipo A es un subtipo restringido del tipo B si y solo si A es un subtipo de B, y todos los componentes públicos presentes en A pero no en B deben ser connectables por defecto. Esto se usado para evitar introducir, a través de una redeclaración, un conector no conectado en el objeto/clase del tipo A en un nivel donde no es posible una conexión. (Ver la Sección 2.5.1 y el Capítulo 3 de Fritzson, 2004).

**escalar** o **variable escalar**: una variable que no es un arreglo. (Ver las secciones 2.1.1 y 2.13).

**tipo simple:** `Real`, `Boolean`, `Integer`, `String` y tipos de enumeración. (Ver la Sección 2.1.1).

**clase especializada**: uno de las siguientes clases: model, connector, package, record, operator record, block, function, operator function y type. La especialización de clases representa afirmaciones sobre el contenido de la clase y restring su uso en otras clases, asi como proporciona mejoras comparadas con el concepto de una clase básica. Por ejemplo, una clase

que tenga la especialización de clase package solamente puede contener claes y constantes. (Ver Sección 2.3.3)

**subtipo compatible**: relaciones entre tipos. A es un subtipo de B si y solo si un número de propiedades de A y B son las mismas y todos los elementos importantes de B tienen elementos correspondientes en A con los mismos nombres y sus tipos siendo subtipos del correspondiente tipo de elemento en B. (Ver el Capítulo 3 de Fritzson, 2004).

**supertipo**: relacion entre tipos. El inverso de subtipo. A es un subtipo de B significa que B es un supertipo o tipo base de A. (Ver la Sección 2.4)

**tipo**: propiedad de una instancia, expresión, que consiste de un número de atributos y un conjunto de elementos públicos que consisten de nombre del elemento, tipo de elemento, y atributos del elemento. Nota: el concepto de tipo de clase es una propiedad de una definición de clase y también incluye elementos protegidos. Un tipo de clase es utilizada en ciertas relaciones de subtipo, por ejemplo, con respecto a la herencia. (Ver el Capitulo 3 de Fritzson, 2004).

**variabilidad**: propiedad de una expresión, la cual puede tener uno de los siguientes cuatro valores:

- *continuous*: una expresión que puede cambiar su valor en algún punto en el tiempo.
- *discrete*: puede cambiar la simulación solamente en eventos durante la simulación.
- *parameter*: constante durante la simulación completa, pero se puede cambiar antes de cada simulación y aparece en los menus de la herramienta. Los valores por defecto de los parámetros de los modelos suelen ser no físicos y se recomienda cambiarlos antes de la simulacion.
- *constant*: constante durante toda la simulacion; puede ser usada y definida en un paquete.

Asignamientos x := expr y ecuaciones ligadas x = expr deben satisfacer una restricción de variabilidad: la expresión no debe tener una variabilidad más alta que la variable x. (Ver Sección 2.1.4)

**variable**: una instancia (objeto) generada por una variable o por la declaración de una constant. Formas especiales de variable son escalares, arreglos y atributos. (Ver Secciones 2.1.1 y 2.3.1)

**declaración de variable**: Un elemento de una definición de clase que genera una variable, parametro o constante. Una decalaración de variable especifica (1) un nombre de variable, por ejemplo un identificador, (2) la clase a ser aplanada para generar la variable, y (3) una expression opcional de un parámetro Booleano. La generación de la variable se suprime si este

parámetro de la expression se evalúa como falso. Una declaración de variable puede ser anulada por la redeclaración de un elemento. (Ver secciones 2.3.1, 2.1.1 y 2.1.3).

**referencia variable**: Una expression que contiene una secuencia de identificadores e indices. Una referencia variable de referencia es equivalente al objeto referenciado. Una referencia variable se resuelve (evalúa) en el ámbito de una clase (o expresión en el caso de una variable de iteración local). Un ámbito define un conjunto de variables y clases visibles. Ejemplo de referencia: `Ele.Resistor.u[21].r` (Ver Sección 2.1.1 y Sección 2.1.3)

**vector**: un arreglo donde el número de dimensiones es 1. (Ver Sección 2.13)

## Literatura

Este glosario ha sido ligeramente adaptado de la especificación del lenguaje Modelica 3.2 (Asociación Modelica 2010). La primera version del glosario fue desarrollada por Jakob Mauss. La versión actual contiene contribuciones de muchos miembros de la Asociación Modelica

# Apéndice B

# Los Comandos de OpenModelica y OMNotebook

Este anexo provee un breve resumen sobre los comandos de OpenModelica y una corta introducción al libro electrónico OMNotebook que puede ser usado para el modelado textual de Modelica.

## B.1 Libro Electronico Interactivo OMNotebook

Los libros electrónicos interactivos son documentos activos que pueden contener cálculos técnicos y texto al igual que gráficas. Por ello, éstos documentos son apropiados para enseñar y desarrollar experimentos, codificar simulaciones, documentar modelos y almacenar información, etc. OMNotebook es una implementación de código abierto de este tipo de libro electrónico, que pertenece al conjunto de herramientas de OpenModelica.

- OMNotebook y los documentos de DrModelica se instalan automáticamente cuando se instala OpenModelica. Para iniciar OMNotebook en Windows use el menú del programa `OpenModelica->OMNotebook`, o haga doble-click en el archivo `.onb` que desea abrir. El documento `DrModelica.onb` se abre automáticamente cuando inicia OMNotebook.
- Para evaluar una celda solo haga click en la celda específica y presione shift+enter. También puede evaluar una secuencia de celdas hacienda click en el marcador de la celda hacia la derecha y presionando shift+enter.
- Si finaliza un comando con un punto y coma (;), el valor del comando no será visualizado en una celda de salida.
- Cuando se usan o graban sus propios archivos, es útil cambiar previamente la ruta del directorio donde se encuentran ubicado sus archivos. Esto puede realizarse con el comando `cd()`.
- Para realizar una simulación, primero evalúe la celda(s) que contiene(n) el modelo al hacer click en ella y presionando shift+enter. Después, debe evaluar un comando de

simulación, por ejemplo, escribiendo "`simulate(modelname, startTime=0, stopTime=25 )`; en una celda de entrada de Modelica y presionando shift+enter.

- Se puede ahorrar escribir repetidamente una instrucción completa escribiendo la parte inicial de un comando, por ejemplo, sim para simulate, y pulsando la combinación de teclas shift-tab. El comando aparecerá automáticamente expandido y completado.
- Cuando se escribe código de Modelica, se debe usar una celda especial del tipo `ModelicaInput`.
- Se pueden crear nuevas celdas de ingreso usando el menú desplegable `Cell->Input Cell`, o usando la combinación de teclas ctrl-shift-I, mientras que una celda nueva con el mismo estilo de texto de la que se encuentra sobre ésta se puede crear con la combinación de teclas Alt+Enter.

Después de simular una clase es posible visualizarla o simplemente mirar los valores de las variables en la clase al evaluar el comando de plot o el comando val.

Los nombres de las variables que se dan a el commando plot corresponden al modelo más recientemente simulado – no es necesario suministrar el nombre del modelo como prefijo.
Para una explicación a modo de tutorial más completo sobre cómo usar un notebook, puede mirar el capítulo de notebook en la Guia de Usuario de OpenModelica, que se encuentra incluído en la instalación de OpenModelica y que puede ser accedido a través del menú del programa (en Windows) para OpenModelica.

OMNotebook: HelloWorld.onb*

File Edit Cell Format Insert Window Help

# First Basic Class

## 1 HelloWorld

The program contains a declaration of a class called `HelloWorld` with two fields and one equation. The first field is the variable x which is initialized to a start value 2 at the time when the simulation starts. The second field is the variable a, which is a constant that is initialized to 2 at the beginning of the simulation. Such a constant is prefixed by the keyword parameter in order to indicate that it is constant during simulation but is a model parameter that can be changed between simulations.

The Modelica program solves a trivial differential equation: `x' = - a * x.` The variable x is a state variable that can change value over time. The `x'` is the time derivative of x.

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = - a * x;
end HelloWorld;
```

```
Ok
```

## 2 Simulation of HelloWorld

```
simulate( HelloWorld, startTime=0, stopTime=4 );
```

```
[done]
```

```
plot( x );
```


Ready

El notebook DrModelica ha sido desarrollado para facilitar el aprendizaje del lenguaje de Modelica al igual que provee una introducción al modelamiento y la simulación orientada a objetos. Está basado y es material complementario del libro de Modelica: "Principles of Object-Oriented Modeling and Simulation with Modelica", (Fritzson 2004). Todos los ejemplos y ejercicios en DrModelica al igual que las paginas de referencia son de ese libro. La gran mayoría del texto en DrModelica también está basada en ese libro.

## B.2 Comandos Comunes y Pequeños Ejemplos

La sesión que se presenta a continuación en OpenModelica, permite mostrar la combinación común de comandos de simulación seguidos de un comando plot.

```
> simulate(myCircuit, stopTime=10.0)
> plot({R1.v})
```

Las secuencias de comandos se usan para simular, cargar y guardar clases, leer y almacenar datos, visualización de resultados y otras tareas más.

Los argumentos que se pasan a una función de secuencia de comandos deben seguir reglas sintácticas y de escritura para Modelica y para la función de secuencia de comandos en particular. En las siguientes tablas se indican brevemente los tipos de parámetros formales a las funciones por la siguiente notación:

- Argumento tipo `String`, por ejemplo `"hello"`, `"myfile.mo"`.
- `TypeName` – clase, paquete o nombre de la función. Por ejemplo `MiClase`, `Modelica.Math`.
- `VariableName` – nombre de la variable, por ejemplo, `v1`, `v2`, `vars1[2].x`, etc.
- Argumento tipo `Integer` o `Real`, por ejemplo, `35`, `3.14`, `xintvariable`.
- `options` – parámetros opcionales

Los comandos más comunes se muestran a continuación; el conjunto completo de comandos se presentan en la siguiente sección.

| | |
|---|---|
| **simulate**(*className*, *options*) | Traduce y simula un modelo, con tiempos de inicio y finalización opcionales y número opcional de intervalos de simulación o pasos para los cuales la simulación será computada. Muchas iteraciones proporcionará una mayor |

| | |
|---|---|
| | resolución de tiempo, pero ocupará más espacio en disco y tomará más tiempo en computar. El número de intervalos por defecto es 500.<br>*Entradas*: `TypeName` *`className`*`; Real` *`startTime`*`; Real` *`stopTime`*`; Integer` *`numberOfIntervals`*`; Real` *`outputInterval`*`; String` *`method`*`; Real` *`tolerance`*`; Real` *`fixedStepSize`*`;`<br>*Salidas*: `SimulationResult simRes;`<br>*Ejemplo 1*: `simulate(myClass);`<br>*Ejemplo 2*: `simulate(myClass, startTime=0, stopTime=2, numberOfIntervals=1000, tolerance=1e-10);.` |
| **`plot`**`(`*`variables`*`, options)` | Grafica una o varias variables del modelo más recientemente simulado, donde *variables* es un nombre simple o un vector de nombres de variables si diferentes variables se deben visualizar con una curva cada una. Los parámetros opcionales xrange y yrange permite especificar los intervalos de visualización en el diagrama.<br>*Entradas*: `VariableName` *`variables`*`; String` *`title`*`; Boolean` *`legend`*`; Boolean` *`gridLines`*`;`<br>`Real` *`xrange`*`[2]` i.e. `{xmin,xmax};`<br>`Real` *`yrange`*`[2]` i.e. `{ymin,ymax};`<br>*Salidas*: `Boolean res;`<br>*Ejemplo 1*: `plot(x)`<br>*Ejemplo 2*: `plot(x,xrange={1,2},yrange={0,10})`<br>*Ejemplo 3*: `plot({x,y,z})` `//` Grafica 3 curvas, para x, y, y z. |
| **`quit`**`()` | Deja y sale del ambiente OpenModelica |

## B.3 Lista completa de Commandos

A continuación se muestra la lista completa de la secuencia de comandos básicos de OpenModelica. Un conjunto adicional de comandos más avanzados para uso de clientes de software se describen en la documentación del sistema de OpenModelica.

Primero se muestra una sesión interactiva de OpenModelica usando algunos pocos comandos. La interfaz de línea de comandos interactiva puede usarse desde las herramientas de OpenModelica OMNotebook, OMShell o desde la interfaz de comandos en el plug-in para Eclipse OpenModelica MDT.

```
>> model Test Real y=1.5; end Test;
{Test}

>> instantiateModel(Test)
"fclass Test
Real y = 1.5;
end Test;
"

>> list(Test)
  "model Test
    Real y=1.5;
  end Test;
"

>> plot(y)

>> a:=1:10
  {1,2,3,4,5,6,7,8,9,10}

>> a*2
  {2,4,6,8,10,12,14,16,18,20}

>> clearVariables()
  true

>> clear()
  true

>> getClassNames()
  {}
```

A continuación se muestra la lista de los comandos básicos de OpenModelica

| | |
|---|---|
| **cd**() | Muestra el directorio actual como una cadena de texto.<br>*Salidas*: `String dir;` |
| **cd**(dir) | Cambia el directorio actual al directorio *dir* suministrado como cadena de texto.<br>*Entradas*: `String dir;`<br>*Salidas*: `Boolean res;` |

| | |
|---|---|
| | *Ejemplo*: `cd("C:\MyModelica\Mydir")` |
| **checkModel**(className) | Valida el modelo, optimiza las ecuaciones y reporta errores.<br>*Salidas*: `TypeName` *`className`*`;` *Outputs*: `Boolean` *`res`*`;`<br>*Ejemplo*: `checkModel(myClass)` |
| **clear**() | Borra todas las definiciones cargadas, incluyendo las variables y las clases.<br>*Salidas*: `Boolean` *`res`*`;` |
| **clearVariables**() | Borra todas las variables definidas por el usuario.<br>*Salidas*: `Boolean` *`res`*`;` |
| **clearClasses**() | Borra todas las definiciones de clase.<br>*Salidas*: `Boolean` *`res`*`;` |
| **clearLog**() | Borra el historial.<br>*Salidas*: `Boolean res;` |
| **closePlots**() | Cierra todas las ventanas de visualización.<br>*Salidas*: `Boolean res;` |
| **dumpXMLDAE**(*modelname*, ...) | Exporta una representación XML del modelo validado y optimizado, de acuerdo con varios parámetros opcionales. |
| **exportDAEtoMatlab**(*name*) | Exporta una representación en MATLAB del modelo. |
| **getLog**() | Entrega el historial en cadena de texto.<br>*Salidas*: `String log;` |
| **help**() | Muestra en pantalla el manual de ayuda de los comandos en cadena de texto. |
| **instantiateModel**(*modelName*) | Valida el modelo y muestra en pantalla la cadena de texto que contiene la definición de la clase validada.<br>*Entradas*: `TypeName className;`<br>*Salidas*: `String flatModel;` |
| **list**() | Muestra en pantalla una cadena de texto que contiene todas las definiciones de clase cargadas.<br>*Salidas*: `String classDefs;` |
| **list**(*className*) | Muestra en pantalla una cadena de texto que contiene la definición de la clase de la clase nombrada.<br>*Entradas*: `TypeName` *`className`*`;` |

| | |
|---|---|
| | *Salidas*: `String classDef;` |
| **listVariables**() | Muestra en pantalla un vector con los nombres de las variables que se encuentran definidas.<br>*Salidas*: `VariableNam[:] names;`<br>*Ejemplo*: `listVariables()  muestra {x,y, ...}` |
| **loadFile**(*fileName*) | Abre el archivo de Modelica (`.mo`) con el nombre *filename* entregado como cadena de texto.<br>*Entradas*: `String fileName` *Salidas*: `Boolean res;`<br>*Ejemplo*: `loadFile("../myLibrary/myModels.mo"` |
| **loadModel**(*className*) | Carga el archivo que corresponde a *className*, usando la clase de Modelica que corresponde con el nombre para localizar el archivo, buscando en la ruta indicada por la variable del entorno `OPENMODELICALIBRARY`<br>Nota: Si por ejemplo `loadModel(Modelica)` falla, es possible que `OPENMODELICALIBRARY` esté apuntando a una ubicación equivocada.<br>*Entradas*: `TypeName className`<br>*Salidas*: `Boolean res;`<br>*Ejemplo 1*: `loadModel(Modelica.Electrical)` |
| **plot**(*variables*, options) | Visualiza una o varias variables del modelo más recientemente simulado, donde *variables* es un nombre simple o un vector de nombres de variables si diferentes variables se deben visualizar con una curva cada una. Los parámetros opcionales xrange y yrange permiten especificar los intervalos de visualización en el diagrama.<br>*Entradas*: `VariableName variables; String title; Boolean legend;  Boolean gridLines;`<br>`Real xrange[2]` i.e. `{xmin,xmax};`<br>`Real yrange[2]` i.e. `{ymin,ymax};`<br>*Salidas*: `Boolean res;`<br>*Ejemplo 1*: `plot(x)`<br>*Ejemplo 2*: `plot(x,xrange={1,2},yrange={0,10})`<br>*Ejemplo 3*: `plot({x,y,z})  //` Grafica 3 curvas, para x, y, y z. |

| | |
|---|---|
| **plotParametric**(*variables1*,*variables2*, options) | Visualiza cada par de variables correspondientes del vector de variables o variables sencillas *variables1*, *variables2* como una gráfica paramétrica.<br>*Entradas*: VariableName *variables1*[:]; VariableName *variables2*[size(variables1,1)]; String *title*; Boolean *legend*; Boolean *gridLines*; Real *range*[2,2];<br>*Salidas*: Boolean *res*;<br>*Ejemplo 1*: plotParametric(x,y)<br>*Ejemplo 2*: plotParametric({x1,x2,x3},{y1,y2,y3}) |
| **Plot2**(variables, options) | Otra implementación (en Java) llamada plot2, que soporta muchas de las opciones de plot() |
| **plotParametric2**(*variables1*,*variables2*, options) | Otra implementación (en Java) llamada plotParametric2, que soporta muchas de las opciones de plotParametric() |
| **plotVectors**(v1, v2, options) | Visualiza los vectores v1 y v2 como una gráfica x-y.<br>*Entradas*: VariableName v1; VariableName v2;<br>*Salidas*: Boolean res; |
| **quit**() | Sale y cierra el entorno de OpenModelica. |
| **readFile**(*fileName*) | Abre el archivo entregado como la cadena de texto *fileName* y muestra en pantalla una cadena de texto que contiene los contenidos del archivo.<br>*Entradas*: String fileName; String matrixName; int nRows; int nColumns;<br>*Salidas*: Real res[nRows,nColumns];<br>*Ejemplo 1*: readFile("myModel/myModelr.mo") |
| **readMatrix**(*fileName*, *matrixName*) | Lee una matriz del archivo entregado *fileName* y *matrixName*.<br>*Entradas*: String *fileName*; String *matrixName*;<br>*Salidas*: Boolean matrix[:,:]; |
| **readMatrix**(fileName, matrixName, nRows, nColumns) | Lee una matrix a partir de un archivo, dado el nombre del archivo *(Filename)*, el nombre de la matrix *(matrixName)* # de filas *(nRows)* y # de Columnas *(nColumns)*.<br>*Entradas*: String fileName; String matrixName; int nRows; int nColumns; |

| | |
|---|---|
| | *Salidas*: `Real res[nRows,nColumns];` |
| **readMatrixSize**(*fileName*, *matrixName*) | Lee las dimensiones de una matrix de un archivo *(fileName)* dado el nombre de la matrix *(matrixName)*.<br>*Entradas*: `String fileName; String matrixName;`<br>*Salidas*: `Integer sizes[2];` |
| **readSimulation Result**( *fileName*, *variables*, *size*) | Lee el resultado de la simulación de una lista de variables y entrega una matriz de valores (cada columna como vector o valores de una variable). El tamaño del resultado (previamente obtenido al usar `readSimulation-ResultSize` se ingresa como entrada).<br>*Entradas*: `String fileName; VariableName variables[:]; Integer size;`<br>*Salidas*: `Real res[size(variables,1),size)];` |
| **readSimulation ResultSize**(*fileName*) | Lee el tamaño del resultado de una simulación, por ejemplo, el numero de los puntos de la simulación computados y almacenados de un vector de trayectoria, de un archivo.<br>*Entradas*: `String fileName;` *Salidas*: `Integer size;` |
| **runScript**(*fileName*) | Ejecuta el archivo de listado de comandos de un archivo de nombre definido como la cadena de texto *fileName*.<br>*Entradas*: `String fileName;` *Salidas*: `Boolean res;`<br>*Ejemplo*: `runScript("simulation.mos")` |
| **saveLog**(*fileName*) | Almacena el historial de mensajes de error de la simulación en un archivo.<br>*Entradas*: `String fileName;` *Salidas*: `Boolean res;` |
| **saveModel**(*fileName*, *className*) | Almacena el modelo/clase con el nombre *className* en el archivo definido por la cadena de texto *fileName*<br>*Entradas*: `String fileName; TypeName className`<br>*Salidas*: `Boolean res;` |
| **saveTotalModel**(*fileName*, *className*) | Almacena la definición total de clase en un archivo de una clase.<br>*Entradas*: `String fileName; TypeName className;`<br>*Salidas*: `Boolean res;` |
| **simulate**(*className*, *options*) | Traduce y simula un modelo, con un tiempo de inicio y final opcional, y un número opcional de intervalos o pasos de |

| | |
|---|---|
| | simulación para los cuales los resultados de la simulación serán computados. Muchas iteraciones proporcionarán una mayor resolución de tiempo, pero ocupará más espacio en disco y tomará más tiempo en computar. El número de intervalos por defecto es 500.<br>*Entradas*: `TypeName` *`className`*`; Real` *`startTime`*`; Real` *`stopTime`*`; Integer` *`numberOfIntervals`*`; Real` *`outputInterval`*`; String` *`method`*`; Real` *`tolerance`*`; Real` *`fixedStepSize`*`;`<br>*Salidas*: `SimulationResult simRes;`<br>*Ejemplo 1*: `simulate(myClass);`<br>*Ejemplo 2*: `simulate(myClass, startTime=0, stopTime=2, numberOfIntervals=1000, tolerance=1e-10);` |
| **`system`**(*`str`*) | Ejecuta *str* como comando de sistema (Shell) en el sistema operativo; entrega como resultado el valor entero de éxito. La salida en *stdout* desde una línea de comandos se coloca en la ventana de consola.<br>*Entradas*: `String` *`str`*`;` *Salidas*: `Integer res;`<br>*Ejemplo*: `system("touch myFile")` |
| **`timing`**(*`expr`*) | Evalúa la expresión *expr* y entrega el número de segundos (tiempo transcurrido) de la evaluación.<br>*Entradas*: `Expression` *`expr`*`;` *Salidas*: `Integer res;`<br>*Ejemplo*: `timing(x*4711+5)` |
| **`translateModel`**(*`className`*) | Valida el modelo, optimiza las ecuaciones y genera el código.<br>*Entradas*: `TypeName` *`className`*`;`<br>*Salidas*: `SimulationObject res;` |
| **`typeOf`**(*`variable`*) | Entrega el tipo de *variable* como cadena de texto.<br>*Entradas*: `VariableName` *`variable`*`;`<br>*Salidas*: `String res;`<br>*Ejemplo*: `typeOf(R1.v)` |
| **`val`**(*`variable, timepoint`*) | Entrega el valor de la *variable* resultado de la simulación evaluada o interpolada en el punto de tiempo *`timepoint`*. Se usan los resultados de la simulación más reciente. |

| | |
|---|---|
| | *Entradas*: `VariableName variable; Real timepoint;`<br>*Salidas*: `Real res;`<br>*Ejemplo 1*: `val(x,0)`<br>*Ejemplo 2*: `val(y.field,1.5)` |
| **writeMatrix**(*fileName*, *matrixName*, *matrix*) | Almacena una matrix en un archivo *(fileName)* definido el nombre de la matriz *(matrixName)* y la matriz *(matrix)*.<br>*Entradas*: `String fileName;`<br>`String matrixName; Real matrix[:,:];`<br>*Salidas*: `Boolean res;` |

## B.4 OMShell y Dymola

**OMShell**

OMShell es una interfaz de línea de comandos muy simple de OpenModelica. Note que OMNotebook es normalmente recomendado para principiantes dado que ofrece mas verificación de errores. OMShell posee las siguientes características para navegar entre los comandos, etc.

- Sale de OMShell presionando Ctrl-d
- Flecha-arriba – Muestra el comando anterior.
- Flecha-abajo – Muestra el siguiente comando.
- Tab – Completa el comando de los comandos existentes en OpenModelica.
- Circula a través de los comandos usando la tecla tab.

**Secuencia de Comandos en Dymola**

Dymola es una herramienta de modelamiento y simulación comercial de Modelica ampliamente usada. El lenguaje de escritura para Dymola es similar al de OpenModelica pero hay algunas diferencias, principalmente apreciables en el uso de cadenas de texto, por ejemplo `"Modelica.Mechanics"`, para los nombres de las clases y los nombres de las variables en lugar de usar los nombres directamente, por ejemplo, `Modelica.Mechanics`, como en la secuencia de comandos de OpenModelica.

A continuación se muestra un ejemplo de un archivo de secuenia de comandos de Dymola para el ejemplo `CoupledClutches` de la librería estándar de Modelica. Para una lista completa de los comandos de escritura de Dymola, consulte la guía de usuario de Dymola.

```
translateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches
")
experiment(StopTime=1.2)
simulate
plot({"J1.w","J2.w","J3.w","J4.w"});
```

## Literatura

Una revisión general de OpenModelica puede encontrarse en Fritzson et al. (2005). Programación documentada (Knuth 1984) es una forma de programación en donde los programas están integrados con la documentación en el mismo documento. Los cuadernos de Mathematica (Wolfran 1997) es uno de los primeros sistemas WYSIWYG (del acrónimo “lo que ves es lo que obtienes” en ingles) que soporta programación documentada. Estos cuadernos fueron usados inicialmente con Modelica, por ejemplo en entorno de simulación y modelamiento MathModelica, ver capítulo 19 (Fritzson 2004) y (Fritzson 2006). El cuaderno de DrModelica ha sido desarrollado para facilitar el aprendizaje del lenguaje de Modelica al igual que para proporcionar una introducción a la simulación y el modelamiento orientado a objetos. Está basado y es material complementario del libro de Modelica (Fritzson 2004). Dymola (Dasault Systemes 2011), es una herramienta industrial robusta para modelamiento y simulación. MathModelica (MathCore 2011) es una herramienta comercial más reciente para el modelamiento y la simulación de Modelica.

# Apéndice C

# Modelamiento textual con OMNotebook y DrModelica

Este anexo presenta algunos ejercicios de modelamiento textual con Modelica los cuales pueden usarse como ejemplos en un mini curso de modelamiento y simulación. En particular, es fácil correr los ejercicios en el libro electronico OMNotebook el cual es parte de OpenModelica, el cual puede descargarse desde el sitio web www.openmodelica.org. Después de la instalación, abra OMNotebook desde la opción del menu `OpenModelica->OMNotebook`. Puede encontrar un listado de comandos de OMNotebook y OpenModelica en el Apéndice B.

El anotador DrModelica se abre automáticamente cuando se inicia OMNotebook. Este anotador ha sido desarrollado para facilitar el aprendiza del lenguaje Modelica y a su vez proveer una introducción al modelamiento y simulación orientado a objetos. DrModelica está basado en el libro Modelica (Fritzson 2004) y a su vez puede ser considerado como material suplementario del mismo libro. Todos los ejemplos y ejercicios en DrModelica, así como las referencias, son tomadas del mismo libro. En forma similar, la mayor parte del texto en DrModelica se encuentra basado en el mismo libro.

El siguiente conjunto de ejercicios puede usarse con cualquier herramiento de Modelica, y puede descargarse desde el sitio de este libro en www.openmodelica.org. Si se usa OpenModelica, estos se pueden encontrar abriendo el documento `TextualModeling-Exercises.onb` en el directorio `testmodels` en la instalación de OpenModelica, e.g. haciendo doble click en el archivo o desplegando el menu de comandos `File->Open` en OMNotebook.

## C.1 HelloWorld

Simular y graficar el siguiente ejemplo en el anotador de ejercicios con una ecuación diferencial y y una condición inicial. Haga un ligero cambio en el modelo, vuelva a simular y a graficar.

```
model HelloWorld "A simple equation"
  Real x(start=1);
equation
  der(x)= -x;
end HelloWorld;
```

Introduzca parcialmente el comando de simulación, e.g. simul, en una celda de entrada (la cual puede ser creada presionando simultánemente las teclas ctrl-shift-I) en el anotador, presione las teclas shift-tab para completer el comando, complete con el nombre HelloWorld, y simulelo!

Antes de completar el comando:

```
simul
```

Después de completar el comando usando las teclas shift-tab:

```
simulate(modelname, startTime=0, stopTime=1, numberOfIntervals=500,
tolerance=1e-4)
```

Después de completar el comando en el archive HelloWorld

```
simulate(HelloWorld, startTime=0, stopTime=1, numberOfIntervals=500,
tolerance=1e-4)
```

Complete un comando plot en una celda de entrada (también se puede expandir usando shift-tab):

```
plot(x)
```

Observe el valor interpolado de la variable x en el tiempo time=0.5 usando la función `val(variableName, time)`:

```
val(x,0.5)
```

Igualmente revise el valor en el tiempo time=0.0:

```
val(x,0.0)
```

## C.2 Pruebe DrModelica con los Modelos de VanDerPol y DAEExample

Localice el modelo `VanDerPol` en DrModelica (siga el enlace de la sección 2.1 de DrModelica), córralo, haga un pequeño cambio, y vuelva a correrlo.

Cambie la el valor `stopTime` de la simulación a 10, simule y grafique.

Cambie el parámetro `lambda` en el modelo a 10, simule y grafique para 50 segundos. Explique el porqué del gráfico obtenido.

Localice el enlace DAEExample en DrModelica. Simule y grafique.

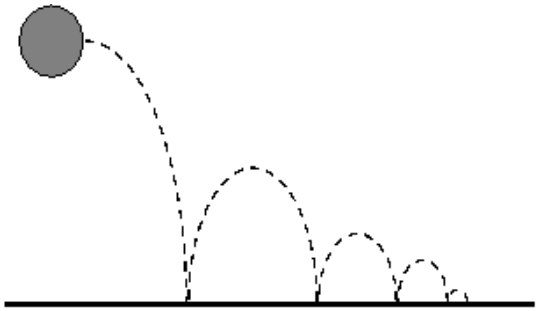
## C.3 Un sistema de Ecuaciones Simple

Construya un modelo en Modelica que resuelva el siguiente sistema de ecuaciones con sus respectivas condiciones iniciales, simule y grafique los resultados:

$$\dot{x} = 2 * x * y - 3 * x$$
$$\dot{y} = 5 * y - 7 * x * y$$
$$x(0) = 2$$
$$y(0) = 3$$

## C.4 Modelamiento Híbrido de una Bola que Rebota

Localice el modelo BouncingBall en una de las secciones de modelamiento híbrido de DrModelcia (e.g. en el enlace “when equations” de la sección 2.9), córralo, haga algunas modificaciones, vuelva a correrlo.



Una bola que rebota

## C.5 Modelamiento Híbrido con una muestra

Construya una señal cuadrada con un periodo de 1s y que empiece en t = 2.5. Note que esto se puede construir ya sea con la solución a una ecuación o con la solución de un algortimo. Pista: una forma fácil es usar el comando `sample`(...) para generar eventos, y defina una variable que cambie de signo en cada evento.

## C.6 Secciones de Funciones y de Algortimos

a) Escriba una función, `sum`, que calcule la suma de números reales, para un vector de tamaño arbitrario.

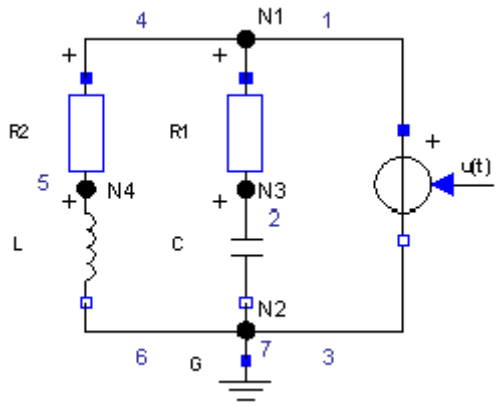
b) Escriba una funcion, `average`, que calcule el promedio de números reales, en un vector de tamaño arbitrario. La función `average` podría hacer uso de la función llamada `sum`.

## C.7 Adicionar un Component Conectado a un Circuito Existente

Adicionar un capacitor entre el componente R1 y el component R2, y un inductor entre el component R1 y el component de voltaje. Use el modelo `SimpleCircuit` descrito a continuación y la librería de componentes estándar de Modelica.

```
loadModel(Modelica);
```

```
model SimpleCircuit
  import Modelica.Electrical.Analog;
  Analog.Basic.Resistor  R1(R = 10);
  Analog.Basic.Capacitor C(C = 0.01);
  Analog.Basic.Resistor  R2(R = 100);
  Analog.Basic.Inductor  L(L = 0.1);
  Analog.Sources.SineVoltage AC(V = 220);
  Analog.Basic.Ground    G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end SimpleCircuit;
```




Este ejemplo ilustra que algunas veces es mas inconveniente usar modelamiento textual comparado con modelamiento gráfico. Si lo desea, puede verificar el resultado usando un editor de modelos gráficos como el descrito en el Anexo D.

## C.8 Modelamiento Detallado de un Circuito Eléctrico

Este ejercicio conssiste en construir un número determinado de componentes eléctricos. A continuación se describen las ecuaciones de cada componente. Puede saltar la subsección de ecuaciones si ya está familiarizado con las ecuaciones.

### C.8.1 Ecuaciones

El *element tierra*

$v_p = 0$

donde $v_p$ es el potencial del elemento tierra.

Una *resistencia*

$i_p + i_n = 0$

$u = v_p - v_n$

$u = R\, i_p$

donde $i_p$ y $i_n$ representan las corrientes en los pines (o puertos) positivo y negativo de la resistencia, $v_p$ y $v_n$ los correspondientes potenciales, *u* el voltaje aplicado a la Resistencia, y R el valor de la resistencia.

Una inductancia

$i_p + i_n = 0$

$u = v_p - v_n$

$u = L\, i'_p$

donde $i_p$ y $i_n$ representan las corrientes en los pines (o puertos) positivo y negativo de la inductancia, , $v_p$ y $v_n$ los correspondientes potenciales, u el voltaje aplicado a la inductancia, L el valor de la inductancia, y $i'_p$ la derivada de la corriente en el pin positivo.

Una *fuente de voltaje*

$i_p + i_n = 0$

$u = v_p - v_n$

$u = V$

donde $i_p$ y $i_n$ representan las corrientes en los pines (o puertos) positivo y negativo de la fuente de voltaje, $v_p$ and $v_n$ los correspondientes potenciales, u el voltaje aplicado a la fuente de voltaje, y V el voltaje constante.

### C.8.2 Implementación

Construya modelos en Modelica para los componentes del modelo arriba mencionados (elemento tierra, resistencia, inductancia, fuente de voltaje, etc.). Se debe definir una clase conector que represente un pin eléctrico ri4w5. Observe que las primeras dos ecuaciones que definen cada uno de los componentes eléctricos con dos pines definidos arriba son iguales. Utilice esta observación para definer un modelo parcial, llamado `TwoPin`, para ser usado en la definición de cualquier component eléctrico de dos pines. En este caso se deben construir seis componentes (`Pin`, `Ground`, `TwoPin`, `Resistor`, `Inductor`, y un `VoltageSource`).

Use los componentes definidos para construir un modelo de un diagrama circuital y simule el comportamiento del circuito.

#### Tipos definidos por el usuario

Primero defina los tipos Voltage y Current:

```
type Voltage = Real;
type Current = Real;
```

#### Pin

El Pin tiene un potencial *v*, y una variable de corriente *i*. De acuerdo a las leyes de Kirchhoff, los potenciales se escogen iguales y las corrientes suman cero en las conexiones. Por tanto, *v* es una variable de potencial de no flujo e *i* es una variable de flujo.

```
connector Pin
  ...
  ...
end Pin;
```

#### Ground

El componente Ground tiene un Pin positivo y una ecuación simple.

```
model Ground
  Pin p;
equation
  ...
end Ground;
```

### TwoPin

El elemento TwoPin tiene un `Pin` positivo y uno negativo, un voltaje `u` y una corriente `i` definida (la corrient `i` no aparece en las ecuaciones arriba y solo se introduce para simplificar la notación)

```
model TwoPin
  Pin p, n;
  ...
  ...
equation
  ...
  ...
  ...
end TwoPin;
```

### Resistor

Para definer el modelo de la resistencia se extiende el modelo parcial `TwoPin`, de esta manera solo adicionamos una declaración del parámetro R en conjunto con la ley de Ohm que relaciona el voltaje y la corriente entre sí.

```
model Resistor
  extends TwoPin;
  ...
equation
  ...
end Resistor;
```

Un modelo equivalente sin usar el modelo parcial podría parecerse al que se muestra a continuación:

```
model Resistor
  ...
  ...
```

```
  ...
  ...
  ...
equation
  ...
  ...
  ...
  ...
end Resistor;
```

*Nota*: la palabra clave `extends` en el lenguaje Modelica se puede interpreter como copiar y pegar información a partir del modelo parcial.

**Inductor**

La ecuación que relaciona el voltaje y la corriente para una inductancia con valor L se adiciona al modelo parcial

```
model Inductor
  ...
  ...
equation
  ...
end Inductor;
```

**VoltageSource**

En este caso el modelo parcial se extiende con la ecuación trivial que el voltaje entre los pines positivo y negativo de la fuente de voltaje permanece constante.

```
model VoltageSource
  ...
  ...
equation
  ...
end VoltageSource;
```

### C.8.3 Colocando Todo el Circuito en Conjunto

A continuación se muestra un ejemplo de un circuito simple donde se instancian los parámetros de los componentes a otros valores diferentes a los definidos por defecto:

```
model Circuit
  Resistor R1(R=0.9);
  Inductor L1(L=0.01);
  Ground G;
  VoltageSource EE(V=5);
equation
  connect(EE.p, R1.p);
  connect(R1.n, L1.p);
  connect(L1.n, G.p);
  connect(EE.n, G.p);
end Circuit;
```

### C.8.4 Simulación del Circuito

Para similar el circuito:

```
simulate(Circuit, startTime=0, stopTime=1)
```

Se pueden graficar varias señales, e.g. `R1.i` la cual es el corriente a través de la resistencia `R1`:

```
plot(R1.i)
```

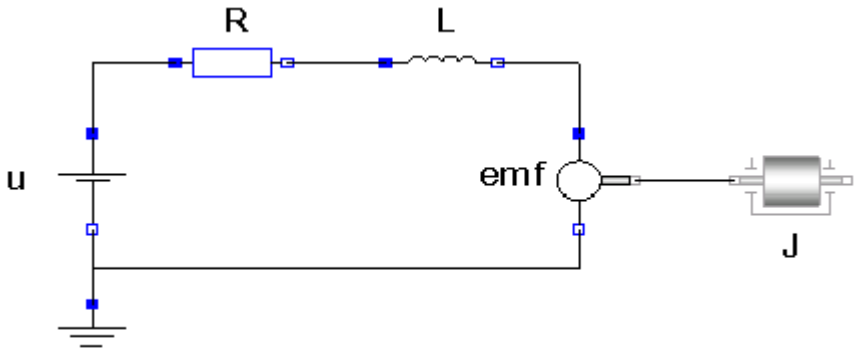
# Apéndice D

# Ejercicios De Modelamiento Gráfico

El siguiente pequeño ejemplo de modelamiento gráfico se puede usar con cualquier editor de modelos gráficos de Modelica.

## D.1 Motor DC Simple

Construya un motor DC simple, que tenga la siguiente estructura, usando la librería estándar de Modelica:

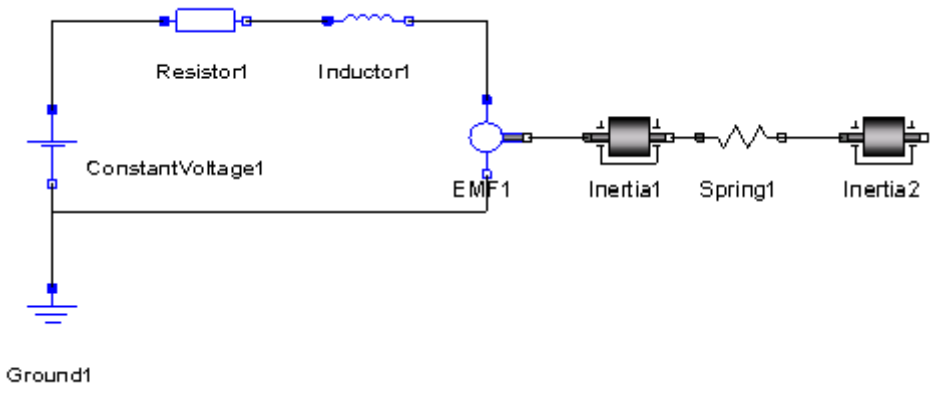



Guarde el modelo realizado en el editor gráfico, puede simularlo directamente desde el editor gráfico o cargarlo y simularlo (puede usar OMShell ó OMNotebook) para 15s, y grafique las variables de velocidad rotacional resultante en el eje de inercia y el voltaje en la fuente de voltaje (denotado u en la figura) en el mismo gráfico.

Pista 1: Revise los componentes de modelos en `Modelica.Electrical.Analogue.Basic`, `Modelica.Electrical.Analogue.Sources`, `Modelica.Mechanics.Rotational`, etc.
Pista 2: si usted tiene dificultades encontrando los nombres de las variables a graficar, puede desagregar el modelo usando el comando `instantiateModel` (en OMNotebook ó OMShell), el cual expone todos los nombres de las variables.

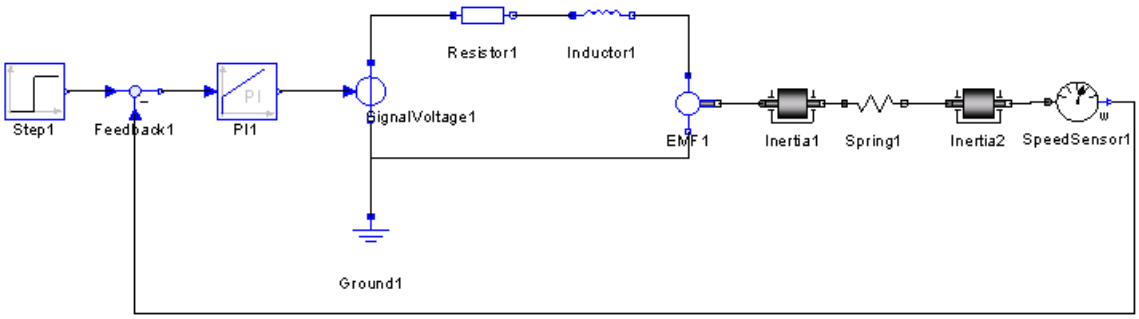
## D.2 Motor DC con Resorte e Inercia.

Adicione un resorte torsional entre el eje saliente y otro elemento de inercia. Simule otra vez y visualice los resultados. Ajuste los parámetros para obtener un resorte mas rígido.



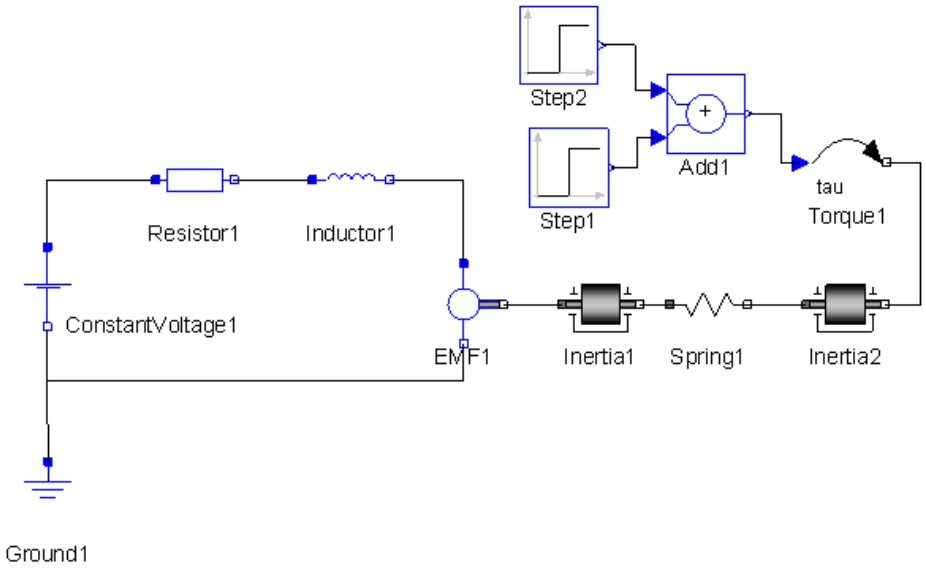

## D.3 Motor DC con Controlador

Adicione un controlador PI al sistema e intente controlar la velocidad rotacional del eje saliente. Verifique los resultados usando como entrada una señal escalón. Sintonice el controlador PI cambiando sus parámetros en el editor gráfico.




## D.4 Motor DC como un Generador

Qué se necesita si se desea hacer un motor DC híbrido, es decir, un motor DC que también puede actuar como un generador por un tiempo limitado? Hágalo trabajar como un motor DC durante los primeros 20s, en ese momento aplique un torque en sentido contrario en el eje saliente por los 20s siguientes, y finalmente apague el torque en sentido contrario, i.e. debería tener un pulso de torque comenzando en 20s y terminando en 40s. Dibuje el siguiente diagrama de conexiones en un editor de modelos gráficos, y ajuste los tiempos de inicio y la amplitud de señal para los modelos de señal Step1 y Step2 para dar el pulso de torque deseado.


Step2
Add1
Step1
tau
Torque1
Resistor1
Inductor1
ConstantVoltage1
EMF1
Inertia1
Spring1
Inertia2
Ground1

# Referencias


Allaby, Michael. Citric acid cycle. *A Dictionary of Plant Sciences*, 1998. http://www.encyclopedia.com/topic/citirc_acid.aspx

Allen, Eric, Robert Cartwright, and Brian Stoler. DrJava: A Lightweight Pedagogic Environment for Java. In Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002), Cincinnati, USA, Feb. 27–Mar. 3, 2002.

Andersson, Mats. Combined Object-Oriented Modelling in Omola. In Stephenson (ed.)*, Proceedings of the 1992 European Simulation Multiconference* (*ESM'92*), York, UK, Society for Computer Simulation International, June 1992.

Andersson, Mats. *Object-Oriented Modeling and Simulation of Hybrid Systems*. Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

Arnold, Ken, and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1999.

Ashby, W. Ross. *An Introduction to Cybernetics,* Chapman & Hall, London, 1956, p. 39.

Aström, K.-J., H. Elmqvist, and S.-E. Mattsson. Evolution of Continuous-Time Modeling and Simulation. In Zobel and Moeller (eds.). Proceedings of the 12th European Simulation Multiconference (ESM'98), pp. 9 – 18, Society for Computer Simulation International, Manchester, UK, 1998.

Augustin, Donald C., Mark S. Fineberg, Bruce B. Johnson, Robert N. Linebarger, F. John Sansom, and John C. Strauss. The SCi Continuous System Simulation Language (CSSL). *Simulation*, 9: 281–303, 1967.

Assmann, Uwe. *Invasive Software Composition*. Springer Verlag, 1993.

Bachmann, Bernard (ed.). Proceedings of the 6th International Modelica Conference. Bielefeld University, Bielefeld, Germany, March 3–4, 2008. Available at www.modelica.org.

Birtwistle, G.M., Ole Johan Dahl, B. Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN*. Auerbach Publishers, Inc., Boca Raton, FL, 1973.

Booch, Grady. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

Booch, Grady. *Object-Oriented Analysis and Design.* Addison-Wesley, 1994.

Brenan, K., S. Campbell, and L. Petzold. *Numerical Solution of Initial-Value Problems in Ordinary Differential-Algebraic Equations*. North Holland Publishing, co., New York, 1989.

Brück, Dag, Hilding Elmqvist, Sven-Erik Mattsson, and Hans Olsson. Dymola for Multi-Engineering Modeling and Simulation. In *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, Mar. 18–19, 2002.

Bunus, Peter, Vadim Engelson, and Peter Fritzson. Mechanical Models Translation and Simulation in Modelica. In *Proceedings of Modelica Workshop 2000*, Lund University, Lund, Sweden, Oct. 23–24, 2000.

Casella, Francesco (ed.). Proceedings of the 7th International Modelica Conference. Como, Italy, March 3–4, 2009. Available at www.modelica.org.

Cellier, Francois E. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. Ph.D. thesis, ETH, Zurich, 1979.

Cellier, Francois E., *Continuous System Modelling*, Springer Verlag, Berlin, 1991.

Clauβ, Christoph. Proceedings of the 8th International Modelica Conference. Dresden, Germany, March 20-22, 2011. Available at www.modelica.org

Davis, Bill, Horacio Porta, and Jerry Uhl. *Calculus & Mathematica Vector Calculus: Measuring in Two and Three Dimensions*. Addison-Wesley, Reading, MA, 1994.

Dynasim AB. *Dymola—Dynamic Modeling Laboratory, Users Manual*, Version 5.0. Dynasim AB, Lund, Sweden, Changed 2010 to Dassault Systemes Sweden. www.3ds.com,/products/catia/portfolio/dymola, 2003.

Elmqvist, Hilding. A Structured Model Language for Large Continuous Systems. Ph.D. thesis, TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

Elmqvist, Hilding, and Sven-Erik Mattsson. A Graphical Approach to Documentation and Implementation of Control Systems. In *Proceedings of the Third IFAC/IFIP Symposium on Software for Computer Control* (*SOCOCO'82*), Madrid, Spain. Pergamon Press, Oxford, U.K., 1982.

Elmqvist, Hilding, Francois Cellier, and Martin Otter. Object-Oriented Modeling of Hybrid Systems. In *Proceedings of the European Simulation Symposium* (*ESS'93*). Society of Computer Simulation, 1993.

Elmqvist, Hilding, Dag Bruck, and Martin Otter. *Dymola—User's Manual*. Dynasim AB, Research Park Ideon, SE-223 70, Lund, Sweden, 1996.

Elmqvist, Hilding, and Sven-Erik Mattsson. Modelica: The Next Generation Modeling Language—An International Design Effort. In *Proceedings of First World Congress of System Simulation*, Singapore, Sept. 1–3, 1997.

Elmqvist, Hilding, Sven-Erik Mattsson, and Martin Otter. Modelica—A Language for Physical System Modeling, Visualization and Interaction. In Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design (CACSD'99), Hawaii, Aug. 22–27, 1999.

Elmqvist, Hilding, Martin Otter, Sven-Erik Mattsson, and Hans Olsson. *Modeling, Simulation, and Optimization with Modelica and Dymola*. Book draft, 246 pages. Dynasim AB, Lund, Sweden, Oct. 2002.

Engelson, Vadim, Håkan Larsson, and Peter Fritzson. A Design, Simulation, and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in Modelica. In *Proceedings of the IEEE International Conference on Information Visualization*, pp. 188–193, London, July 14–16, 1999.

Ernst, Thilo, Stephan Jähnichen, and Matthias Klose. The Architecture of the Smile/M Simulation Environment. In Proceedings 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Vol. 6, Berlin, Germany, pp. 653–658. See also http://www.first.gmd.de/smile/smile0.html, 1997.

Fauvel, John, Raymond Flood, Michael Shortland, and Robin Wilson. *LET NEWTON BE! A New Perspective on His Life and Works. Oxford University Press*. Second Edition. Oxford University Press, Oxford, 1990.

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shiram Krishnamurthi. The DrScheme Project: An Overview. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, June 17–19, 1998.

Fritzson, Dag, and Patrik Nordling. Solving Ordinary Differential Equations on Parallel Computers Applied to Dynamic Rolling Bearing Simulation. In *Parallel Programming and Applications*, P. Fritzson, and L. Finmo (eds.), IOS Press, 1995.

Fritzson, Peter, and Karl-Fredrik Berggren. Pseudo-Potential Calculations for Expanded Crystalline Mercury, *Journal of Solid State Physics*, 1976.

Fritzson, Peter. *Towards a Distributed Programming Environment based on Incremental Compilation.* Ph.D. thesis, 161 pages. Dissertation no. 109, Linköping University, Apr. 13, 1984.

Fritzson, Peter, and Dag Fritzson. The Need for High-Level Programming Support in Scientific Computing—Applied to Mechanical Analysis. *Computers and Structures*, 45:2, pp. 387–295, 1992.

Fritzson, Peter, Lars Viklund, Johan Herber, and Dag Fritzson. Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis, In *Proc. of TOOLS EUROPE'92*, Dortmund, Germany, Mar. 30–Apr. 2. Prentice Hall, 1992.

Fritzson, Peter, Lars Viklund, Dag Fritzson, and Johan Herber. High Level Mathematical Modeling and Programming in Scientific Computing, *IEEE Software*, pp. 77–87, July 1995.

Fritzson, Peter, and Vadim Engelson. Modelica—A Unified Object-Oriented Language for System Modeling and Simulation. *Proceedings of the 12th European Conference on Object-Oriented Programming* (*ECOOP'98*), Brussels, Belgium, July 20–24, 1998.

Fritzson, Peter, Vadim Engelson, and Johan Gunnarsson. An Integrated Modelica Environment for Modeling, Documentation and Simulation. In Proceedings of Summer Computer Simulation Conference '98, Reno, Nevada, July 19–22, 1998.

Fritzson, Peter (ed.). Proceedings of SIMS'99—The 1999 Conference of the Scandinavian Simulation Society. Linköping, Sweden, Oct. 18–19, 1999. Available at www.scansims.org.

Fritzson, Peter (ed.). Proceedings of Modelica 2000 Workshop. Lund University, Lund, Sweden, Oct. 23–24, 2000. Available at www.modelica.org.

Fritzson, Peter, and Peter Bunus. Modelica—A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. Proceedings of the 35th Annual Simulation Symposium, San Diego, California, Apr. 14–18, 2002.

Fritzson, Peter, Peter Aronsson, Peter Bunus, Vadim Engelson, Henrik Johansson, Andreas Karström, and Levon Saldamli. The Open Source Modelica Project. In *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, Mar. 18–19, 2002.

Fritzson Peter, Mats Jirstrand, and Johan Gunnarsson. MathModelica—An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming. In Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany, Mar. 18–19, 2002. Available at www.ida.liu.se/labs/pelab/modelica/ and at www.modelica.org.

Fritzson, Peter (ed.). Proceedings of the 3rd International Modelica Conference. Linköping University, Linköping, Sweden, Nov 3–4, 2003. Available at www.modelica.org.

Fritzson, P. *Principles of Object Oriented Modeling and Simulation with Modelica 2.1* . Wiley-IEEE Press, Hoboken, NJ, 2004.

Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.2*, Wiley-IEEE Press. Expected publication during 2011.

Fritzson Peter, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In *Simulation News Europe*, 44/45, December 2005. See also: http://www.openmodelica.org

Fritzson, Peter. MathModelica - An Object Oriented Mathematical Modeling and Simulation Environment. Mathematica Journal, Vol 10, Issue 1. February. 2006.

Fritzson, Peter. Electronic Supplementary Material to Introduction to Modeling and Simulation of Technical ans Physical Systems with Modelica. www.openmodelica.org, July 2011.

Fritzson, Peter. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. 1250 pages. ISBN 9781-118-859124, Wiley IEEE Press, 2014.

Gottwald, S., W. Gellert (Contributor), and H. Kuestner (Contributor). *The VNR Concise Encyclopedia of Mathematics*. Second edition, Van Nostrand Reinhold, New York, 1989.

Hairer, E., S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer Series in Computational Mathematics, 2nd ed. 1992.

Hairer, E., and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics, 1st ed. 1991.

Hyötyniemi, Heikki. Towards New Languages for Systems Modeling. In *Proceedings of SIMS 2002*, Oulu, Finland, Sept. 26–27, 2002. Available at www.scansims.org,

IEEE. Std 610.3-1989. *IEEE Standard Glossary of Modeling and Simulation Terminology*, 1989

IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

Knuth, Donald E. Literate Programming. *The Computer Journal*, NO27(2), pp. 97–111, May 1984.

Kral, Christian and Anton Haumer. Proceedings of the 6th International Modelica Conference. Vienna, Austria, Sept 4 – 6, 2006. Available at www.modelica.org.

Kågedal, David, and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference'98*, Reno, Nevada, USA, July 19–22 1998.

Lengquist Sandelin, Eva-Lena, and Susanna Monemar. DrModelica—An Experimental Computer-Based Teaching Material for Modelica. Master's thesis, LITH-IDA-Ex03/3, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 2003.

Lengquist Sandelin, Eva-Lena, Susanna Monemar, Peter Fritzson, and Peter Bunus. DrModelica—A Web-Based Teaching Environment for Modelica. In Proceedings of the 44th Scandinavian Conference on Simulation and Modeling (SIMS'2003), Västerås, Sweden, Sept. 18−19, 2003. Available at www.scansims.org.

Ljung, Lennart, and Torkel Glad. *Modeling of Dynamic Systems*. Prentice Hall, 1994.

MathCore Engineering AB. Home page: www.mathcore.com. MathCore Engineering Linköping, Sweden, 2003.

MathWorks Inc. *Simulink User's Guide*, 2001.

MathWorks Inc. *MATLAB User's Guide*, 2002.

Mattsson, Sven-Erik, Mats Andersson, and Karl-Johan Åström. Object-Oriented Modelling and Simulation. In Linkens (ed.), *CAD for Control Systems*, Chapter 2, pp. 31–69. Marcel Dekker, New York, 1993.

Meyer, Bertrand. *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, Englewood Cliffs, 1997.

Mitchell, Edward E.L., and Joseph S. Gauthier. *ACSL: Advanced Continuous Simulation Language—User Guide and Reference Manual*. Mitchell & Gauthier Assoc., Concord, Mass, 1986.

Modelica Assocation. Home page: www.modelica.org. Last accessed 2010.

Modelica Association. *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial and Design Rationale Version 1.0*, Sept. 1997.

Modelica Association. *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial, Version 1.4.*, Dec. 15, 2000. Available at http://www.modelica.org

Modelica Association. *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.2.,* March 2010. Available at http://www.modelica.org

ObjectMath Home page: http://www.ida.liu.se/labs/pelab/omath.

OpenModelica page: http://www.openmodelica.org.

Otter, Martin, Hilding Elmqvist, and Francois Cellier. Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola, *Nonlinear Dynamics*, 9, pp. 91-112. Kluwer Academic Publishers, 1996.

Otter, Martin. Objektorientierte Modellierung Physikalischer Systeme, Teil 1: Übersicht. In *Automatisierungstechnik*, 47(1): A1–A4. 1999 (In German.) The first in a series of 17 articles. 1999.

Otter, Martin (ed.) Proceedings of the 2nd International Modelica Conference. Oberpfaffenhofen, Germany, Mar. 18–19, 2002. Available at www.modelica.org.

Otter, Martin, Hilding Elmqvist, and Sven-Erik Mattsson. The New Modelica MultiBody Library. *Proceedings of the 3rd International Modelica Conference*, available at www.modelica.org. Linköping, Sweden, Nov 3-4, 2003.

PELAB. Page on Modelica Research at PELAB, Programming Environment Laboratory, Dept. of Computer and Information Science, Linköping University, Sweden, www.ida.liu.se/labs/pelab/modelica, 2003.

Piela, P.C., T.G. Epperly, K.M. Westerberg, and A.W. Westerberg. ASCEND—An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language, *Computers and Chemical Engineering*, Web page: (http://www.cs.cmu.edu/~ascend/Home.html), 15:1, pp. 53–72, 1991.

Pritsker, A., and B. Alan. *The GASP IV Simulation Language*. Wiley, New York, 1974.

Rumbaugh, J.M., M. Blaha, W. Premerlain, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice-Hall,1991.

Sahlin, Per., and E.F. Sowell. A Neutral Format for Building Simulation Models. In *Proceedings of the Conference on Building Simulation*, IBPSA, Vancouver, Canada, 1989.

Sargent, R.W.H., and Westerberg, A.W. Speed-Up in Chemical Engineering Design, *Transaction Institute in Chemical Engineering*, 1964.

Schmitz, Gerhard (ed.). Proceedings of the 4th International Modelica Conference. Technical University Hamburg-Harburg, Germany, March 7–8, 2005. Available at www.modelica.org.

Shumate, Ken, and Marilyn Keller. *Software Specification and Design: A Disciplined Approach for Real-Time Systems*. Wiley, New York, 1992.

Stevens, Richard, Peter Brook, Ken Jackson, and Stuart Arnold. *Systems Engineering: Coping with Complexity*. Prentice-Hall, London, 1998.

Szyperski, Clemens. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Reading, MA,1997.

Tiller, Michael. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

Viklund, Lars, and Peter Fritzson. ObjectMath—An Object-Oriented Language and Environment for Symbolic and Numerical Processing in Scientific Computing, *Scientific Programming*, 4: 229–250, 1995.

Wolfram, Stephen. *The Mathematica Book*. Wolfram Media Inc, 1997.

Wolfram, Stephen. *A New Kind of Science*. Wolfram Media Inc, 2002.

# Indice