Introducción a Modelica.

Ernesto Kofman

Laboratorio de Sistemas Dinámicos
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Índice general

Capítulo 1

Principios Básicos de Modelica

1.1. Conceptos Generales

Modelica es un lenguaje orientado a objetos desarrollado para describir de manera sencilla modelos de sistemas dinámicos eventualmente muy complejos.

Además de las características básicas de todo lenguaje orientado a objetos, Modelica contiene herramientas específicas que permiten describir las relaciones constitutivas de los distintos componentes de cada modelo y las relaciones estructurales que definen la interacción entre dichos componentes.

De esta manera, el lenguaje permite asociar cada *componente* de un sistema a una *instancia* de una *clase* de Modelica.

Adicionalmente, los componentes típicos de los sistemas de distintos dominios de la física y de la técnica pueden agruparse en librerías de clases para ser reutilizados. De hecho, existe una librería estándar de clases de Modelica, que contiene los principales componentes básicos de sistemas eléctricos, mecánicos (traslacionales, rotacionales y multicuerpos), térmicos, state graphs, y diagramas de bloques. Otras librerías (disponibles en la web) contienen componentes de sistemas hidráulicos, bond graphs, redes de petri, etc.

Por otro lado, las herramientas que provee Modelica para expresar relaciones estructurales de un modelo permiten construir la estructura del mismo de una manera totalmente gráfica, lo que a su vez permite describir un sistema mediante un diagrama muy similar al del Sistema Físico Idealizado.

Como con todo lenguaje, para poder simular un modelo descripto en Modelica es necesario utilizar un *compilador*. Actualmente existen tres compiladores más o menos completos de Modelica: *Dymola*, *MathModelica* y *OpenModelica*. Los dos primeros son herramientas comerciales que cuentan con interfaces gráficas para construir los modelos. *OpenModelica* es una herramienta libre, de código abierto, que aún está en desarrollo y que no cuenta en principio con interface gráfica (aunque puede utilizarse con una versión gratuita de *MathModelica*

denominada *MathModelica Lite*).

A modo de ejemplo, veamos en primer lugar un modelo relativamente simple de Modelica. Consideremos un circuito RLC serie conectado a una fuente de tensión que impone un escalón de entrada.

Este modelo, utilizando las clases que provee la librería estándar de Modelica, tiene la siguiente representación:

```
model RLC_Circuit
  Modelica.Electrical.Analog.Basic.Ground Ground1
  Modelica.Electrical.Analog.Basic.Resistor Resistor1
  Modelica.Electrical.Analog.Basic.Inductor Inductor1
  Modelica.Electrical.Analog.Basic.Capacitor Capacitor1
  Modelica.Electrical.Analog.Sources.ConstantVoltage ConstantVoltage1
equation
  connect(Resistor1.n, Inductor1.p)
  connect(Inductor1.n, Capacitor1.p)
  connect(Capacitor1.n, Ground1.p)
  connect(ConstantVoltage1.p, Resistor1.p)
  connect(ConstantVoltage1.n, Ground1.p)
end RLC_Circuit;
```

Las primeras 5 líneas de código declaran los componentes Ground1, Resistor1, etc. como instancias de ciertas clases definidas en la librería estándar. Como puede verse, el modelo también tiene una sección de ecuaciones (tras el comando **equation**), donde en este caso se describen las conexiones entre los componentes. Demás está decir, las *ecuaciones* que definen la función **connect** no son ni más ni menos que las *relaciones estructurales* del modelo.

La Figura 1.1 muestra el modelo del circuito RLC tal como se ve en la interface gráfica de Dymola.

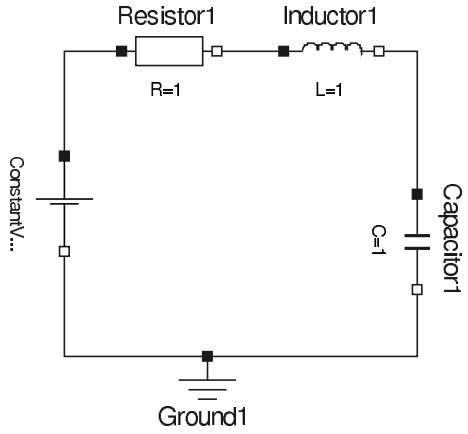

Figura 1.1: Representación en Dymola del circuito RLC

Siguiendo con la idea del ejemplo anterior, a partir del mismo podemos construir un sistema más complejo, reemplazando el capacitor por un conversor electromecánico (motor de cc), y acoplándolo con una inercia, y con un sistema piñón cremallera, desde la cual se acopla una masa, sujeta a una pared a través de un resorte y un amortiguador.

El modelo en Modelica, utilizando los componentes de la librería estándar, es el que sigue:

```
model Motor1
  Modelica.Electrical.Analog.Basic.Ground Ground1
  Modelica.Electrical.Analog.Basic.Resistor Resistor1
  Modelica.Electrical.Analog.Basic.Inductor Inductor1
  Modelica.Electrical.Analog.Sources.ConstantVoltage ConstantVoltage1
  Modelica.Electrical.Analog.Basic.EMF EMF1
  Modelica.Mechanics.Rotational.Inertia Inertia1
  Modelica.Mechanics.Rotational.IdealGearR2T IdealGearR2T1
  Modelica.Mechanics.Translational.SlidingMass SlidingMass1
  Modelica.Mechanics.Translational.SpringDamper SpringDamper1
  Modelica.Mechanics.Translational.Fixed Fixed1
equation
  connect(Resistor1.n, Inductor1.p)
  connect(ConstantVoltage1.p, Resistor1.p)
  connect(ConstantVoltage1.n, Ground1.p)
  connect(Inductor1.n, EMF1.p)
  connect(EMF1.n, Ground1.p)
  connect(EMF1.flange_b, Inertia1.flange_a)
  connect(Inertia1.flange_b, IdealGearR2T1.flange_a)
  connect(IdealGearR2T1.flange_b, SlidingMass1.flange_a)
  connect(SlidingMass1.flange_b, SpringDamper1.flange_a)
  connect(SpringDamper1.flange_b, Fixed1.flange_b)
end Motor1;
```

El modelo, en la interface gráfica de Dymola, aparece como se muestra en la Figura 1.2.

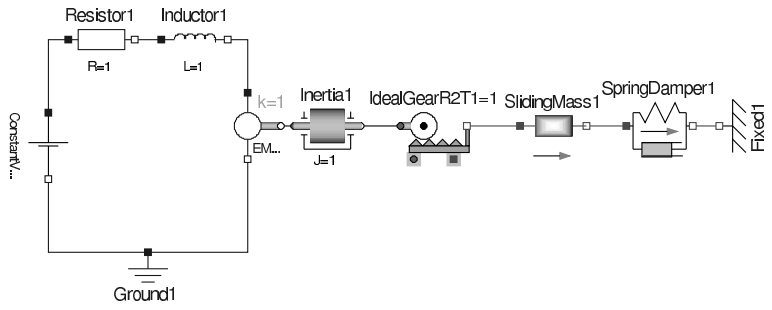

Figura 1.2: Representación en Dymola de un sistema electromecánico

En ambos ejemplos, podemos ver claramente la cercanía que pueden tener los modelos representados en Modelica con los diagramas típicos que representan los sistemas físicos idealizados.

Por último, antes de comenzar a ver los principios del lenguaje, veremos un último ejemplo de un modelo Bond Graphs correspondiente a un RLC serie, o cualquier modelo análogo. Utilizando los elementos de la libería BondLib (una librería que se puede descargar gratuitamente de la red, y que fue desarrollada por François Cellier), el modelo puede especificarse como sigue:

```
model BondRLC
  BondLib.Bonds.Bond Bond1
  BondLib.Junctions.J1p4 J1p4_1
  BondLib.Passive.R R1
  BondLib.Passive.C C1
  BondLib.Passive.I I1
  BondLib.Sources.Se Se1
  BondLib.Bonds.Bond Bond2
  BondLib.Bonds.Bond Bond3
  BondLib.Bonds.Bond Bond4
equation
  connect(Se1.BondCon1, Bond1.BondCon1)
  connect(Bond1.BondCon2, J1p4_1.BondCon1)
  connect(Bond3.BondCon1, J1p4_1.BondCon4)
  connect(C1.BondCon1, Bond3.BondCon2)
  connect(Bond2.BondCon2, I1.BondCon1)
  connect(J1p4_1.BondCon3, Bond4.BondCon1)
  connect(Bond4.BondCon2, R1.BondCon1)
end BondRLC;
```

La Figura 1.3 muestra el modelo tal como se ve en la interface gráfica de Dymola.

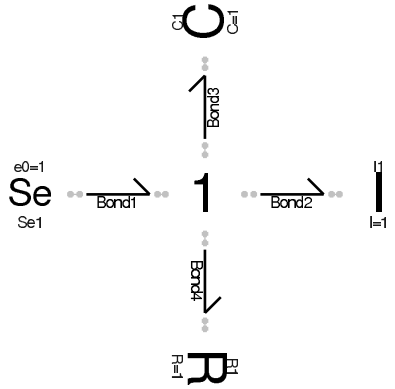

Figura 1.3: Representación en Dymola de un Bond Graph

1.2. Clases y Funciones Elementales

Como mencionamos anteriormente, Modelica es un lenguaje orientado a objetos. Por lo tanto, la especificación de modelos se basa en la definición e instanciación de *clases*.

Por ejemplo, la clase **VanDerPol** que presentamos a continuación, es la representación en Modelica de la ecuación de Van der Pol:

```
class VanDerPol
  Real x(start = 1);
  Real y(start = 1);
  parameter Real lambda = 0.3;
equation
  der(x) = y;
  der(y) = - x + lambda*(1 - x*x)*y;
end VanDerPol;
```

Como podemos ver (y similar a los ejemplos introductorios ya vistos), esta clase tiene dos partes, una en la que se declaran los componentes de la misma (en este caso dos variables reales, **x** e **y**, y un parámetro real **lambda**) y otra, tras el comando **equation**, donde se describen las ecuaciones que representan el sistema.

La declaración **Real x(start = 1)** dice que la variable **x** es una instancia de la clase **Real**, un tipo predefinido en Modelica. La clase **Real** tiene varios atributos, siendo uno de ellos el valor inicial de la variable (**start**). El valor de cada atributo de una clase puede elegirse al momento de instanciar dicha clase. En este caso, en ambas variables, se dio valor 1 al atributo **start** de la clase.

Por otro lado, la declaración **parameter Real lambda = 0.3** utiliza al especificador **parameter** para indicar que lo que sigue es la declaración de un parámetro, es decir, una constante. Entonces, además de indicar la clase a la que pertenece el parámetro (en este caso la clase **Real**), hay que darle un valor.

Luego sigue la sección de ecuaciones. La función **der** es una de las funciones predefinidas en Modelica que indica la derivada respecto al tiempo de una expresión (aunque en general utilizaremos derivadas de variables).

Las ecuaciones en Modelica son (en general) acausales. Por lo tanto, el siguiente modelo **VanderPol2** es equivalente al anterior.

```
class VanDerPol2
  Real x(start = 1);
  Real y(start = 1);
  parameter Real lambda = 0.3;
equation
  der(x) - y = 0;
  der(y) + x - lambda*(1 - x*x)*y = 0;
end VanDerPol2;
```

Para que un modelo esté completo y pueda ser simulado, debe haber tantas ecuaciones como variables. En el caso de la clase **VanDerPol**, esto se cumple ya que hay dos variables y dos ecuaciones.

1.3. Reutilización de Clases

Si bien con lo visto en el punto anterior podríamos en principio representar cualquier sistema de ecuaciones algebraico diferenciales, esto no nos alcanza para construir modelos matemáticos a partir de los componentes y de la estructura de un sistema. Para este objetivo, es necesario utilizar conceptos un poco más avanzados, que involucren la reutilización de clases y el conexionado de submodelos.

Comencemos entonces con un ejemplo, en el que nos proponemos construir un modelo en Modelica de la cascada de dos filtros pasabajos, cada uno de ellos caracterizado por la ecuación:

$$T \cdot \dot{y}(t) + y(t) = u(t) \tag{1.1}$$

Para ello, comenzamos creando una nueva clase **LowPassFilter**.

```
model LowPassFilter
  parameter Real T=1;
  Real u;
  Real y;
equation
  T*der(y)+y=u;
end LowPassFilter;
```

Notar que esta clase tiene dos variables y una única ecuación. Por lo tanto, no podemos simular este modelo aislado. Es claro en este caso que falta especificar quien es la entrada.

Notar además que utilizamos **model** en lugar de **class** para la definición. Además de la clase general **class**, Modelica cuenta con 7 clases restringidas: **block**, **connector**, **function**, **model**, **package**, **record** y **type**. Cada una de estas clases restringidas permite declarar clases más específicas que iremos viendo a lo largo del curso. Por supuesto, cualquiera de ellas puede reemplazarse por **class**, pero siempre es mejor especificar de que tipo de clase se trata para mejorar la legibilidad y facilitar la depuración de código.

Como siguiene paso, armaremos el modelo compuesto por dos filtros en cascada, el primero con constante de tiempo **T**=1 (el valor por defecto en la clase **LosPassFilter**) y el otro con constante de tiempo **T**=2. Además, consideraremos una entrada senoidal al primer filtro y supondremos que el segundo filtro tiene condición inicial $y = 1$.

```
model Filter2
```

```
  LowPassFilter F1;
  LowPassFilter F2(T=2,y(start=1));
equation
  F1.u=sin(time);
  F2.u=F1.y;
end Filter2;
```

Como vemos, el modelo **Filter2** contiene dos componentes, **F1** y **F2**, que son instancias de la clase **LowPassFilter** que definimos antes.

El primer filtro, **F1**, mantiene todos los atributos por defecto de la clase **LowPassFilter**, mientras que el segundo tiene cambiado el parámetro **T**, y el atributo **start** de la variable real **y**.

En cuanto a las ecuaciones, la primera iguala la variable **u** del primer filtro con la función **sin(time)**. La función **sin** es otra función predefinida en el lenguaje Modelica, mientras que **time** es una variable global de Modelica que contiene el tiempo.

La segunda ecuación expresa la *conexión* en cascada de los filtros. En total, el modelo **Filter2** contiene 4 ecuaciones (2 propias y 1 dentro de cada modelo **F1** y **F2**) y 4 variables (**F1.u**, **F1.y**, **F2.u** y **F2.y**). Por lo tanto, este modelo puede simularse sin problemas.

Como ya dijimos, las ecuaciones son acausales. Por lo tanto, es completamente válido escribir el siguiente modelo de Modelica, donde esta vez forzamos la *salida* **F2.y** a un determinado valor (esto sería equivalente a realizar una inversión causal).

```
model Filter2b
  LowPassFilter F1;
  LowPassFilter F2(T=2,y(start=1));
equation
  F2.y=sin(time);
  F2.u=F1.y;
end Filter2b;
```

En este caso, sin embargo, el valor inicial de **F2.y** no es conservado en $t = t_0^+$ (hay una discontinuidad en **F2.y** cuando $t = t_0$).

1.4. Paquetes y Librerías

Otra de las clases restringidas de Modelica es definida por la palabra **package**. Esta clase sirve para mantener ordenados los modelos y las demás clases que se utilizan en forma conjunta, a modo de librerías. Por ejemplo, las clases **LowPassFilter**, **Filter2** y **Filter2b** conviene empaquetarlas dentro de una única clase, de manera que puedan guardarse en un único archivo, y que en otro contexto, podamos utilizar otro modelo llamado **Filter2**.

Para esto, podemos hacer lo siguiente:

```
package Filtros

model LowPassFilter
  parameter Real T=1;
  Real u;
  Real y;
equation
  T*der(y)+y=u;
end LowPassFilter;

model Filter2
  LowPassFilter F1;
  LowPassFilter F2(T=2,y(start=1));
equation
  F1.u=sin(time);
  F2.u=F1.y;
end Filter2;

model Filter2b
  LowPassFilter F1;
  LowPassFilter F2(T=2,y(start=1));
equation
  F2.y=sin(time);
  F2.u=F1.y;
end Filter2b;

end Filtros;
```

Está permitido también colocar paquetes dentro de paquetes, y en general, colocar declaraciones de cualquier clase restringida (no sólo **model**) dentro de un paquete.

1.5. Conexiones

En el último ejemplo de los filtros, resolvimos la conexión entre los dos filtros agregando la ecuación **F2.u=F1.y**. Sin embargo, en sistemas complejos esto no es muy práctico.

En la mayor parte de los dominios, la conexión entre componentes involucra relaciones estructurales entre más de una variable. Por ejemplo, al conectar varios dipolos eléctricos a un mismo punto, estamos dando lugar a varias ecuaciones, unas que igualan el potencial de todos los conectores y una ecuación adicional que iguala a cero la suma de las corrientes.

Para tomar en cuenta estas particularidades de las conexiones, Modelica cuenta con una clase restringida **connector** y con una función predefinida **connect**.

Veamos entonces como se utilizan estos conceptos en un ejemplo sencillo. La siguiente clase (**Pin**) puede utilizarse para definir un conector eléctrico.

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

Como vemos, la clase **Pin** contiene dos componentes del tipo **Real**. El primero de ellos (**v**) sería la tensión y el segundo (**i**) la corriente. Además, la corriente está definida con el modificador **flow**.

Un conector puede contener muchos componentes (de distintas clases, no necesariamente de tipo **Real**), y Modelica distingue entre componentes tipo *across* y tipo *through*. Los primeros son los que se declaran por defecto, mientras que los segundos son los que se declaran con el modificador **flow**. Al conectar (luego veremos como) dos o mas conectores entre sí, se agregarán ecuaciones $v_1 = v_2 = \cdots$ para los componentes tipo *across* y una ecuación del tipo $i_1 + i_2 + \cdots = 0$ para los componentes tipo *through* (es decir, las declaradas con el modificador **flow**).

En Modelica se trabaja con la convención de que las variables tipo **flow** son positivas cuando tienen un sentido entrante al conector.

Utilizando el conector **Pin**, podemos entonces derivar el modelo de un resistor como sigue:

```
model Res
  Pin p; //conector positivo
  Pin n; //conector negativo
  Real v; //voltaje
  Real i; //corriente
  parameter Real R=1; //resistencia
equation
  v=p.v-n.v;
  i=p.i;
  p.i+n.i=0;
  v-i*R=0;
end Res;
```

Como podemos ver, la clase **Res** cuenta con dos conectores (**p**, **n**), dos variables reales (**v**, **i**), y un parametro real (**R**). En la sección de ecuaciones, encontramos las relaciones constitutivas del resistor. Las 3 primeras vinculan las corrientes y tensiones de los conectores con la tensión y corriente de la resistencia, mientras que la tercera expresa la ley de Ohm.

Con la misma idea, podemos hacer el modelo de un capacitor:

```
model Cap
  Pin p; //conector positivo
  Pin n; //conector negativo
```

```
  Real v; //voltaje
  Real i; //corriente
  parameter Real C=1; //capacitancia
equation
  v=p.v-n.v;
  i=p.i;
  p.i+n.i=0;
  i-C*der(v)=0;
end Cap;
```

y el modelo de una puesta a tierra (potencial de referencia)

```
model Ground
  Pin p;
equation
  p.v=0;
end Ground;
```

Luego, utilizando estas tres clases, podemos armar un primer circuito RC de la siguiente forma:

```
model RC
  Cap C1(v(start=1));
  Res R1(R=10);
  Ground G1;
equation
  connect(R1.p,C1.p);
  connect(R1.n,C1.n);
  connect(R1.n,G1.p);
end RC;
```

Notar que en este caso dimos una condición inicial a la tensión del capacitor, y utilizamos una resistencia de 10Ω. Además de los tres componentes (capacitor, resistor y tierra), el modelo contiene tres *ecuaciones*. En efecto, cada comando **connect** no hace sino agregar un sistema de ecuaciones que representa relaciones estructurales del sistema.

Cada comando **connect** vincula dos conectores y símplemente agrega, como dijimos antes, ecuaciones del tipo $v_1 = v_2$ para las variables tipo *across*, y agrega términos a las ecuaciones del tipo $i_1 + i_2 + \cdots = 0$ de las variables tipo through.

1.6. Herencia y Modelos Parciales

Si comparamos las clases **Cap** y **Res** que desarrollamos para el ejemplo anterior, veremos que tienen muchas cosas en común. Ambas clases contienen dos variables y dos conectores idénticos, y las 3 primeras ecuaciones son también idénticas.

Esta coincidencia se debe al hecho que tanto el capacitor como la resistencia son dipolos eléctricos. De hecho, cualquier modelo que hagamos de un dipolo eléctrico tendrá en común todos estos elementos.

Una manera de explotar estas *coincidencias*, que son de hecho muy comunes en todos los dominios, para desarrollar modelos más simples es utilizando el concepto de *herencia* de la orientación a objetos.

Una manera de hacer esto en Modelica es definiendo un *modelo parcial* que contenga todos los elementos comunes y luego derivando distintas clases a partir del mismo que heredan dichos componentes.

Para el caso de un dipolo eléctrico, podemos utilizar el siguiente modelo:

```
partial model OnePort
  Pin p; //conector positivo
  Pin n; //conector negativo
  Real v; //voltaje
  Real i; //corriente
equation
  v=p.v-n.v;
  i=p.i;
  p.i+n.i=0;
end OnePort;
```

El modificador **partial** es sólo para indicar que el modelo está incompleto y prevenir a un usuario de incluirlo directamente dentro de un modelo.

Utilizando la clase **OnePort** podemos ahora sí construir el modelo de un resistor de una manera mucho más simple:

```
model Resistor
  extends OnePort;
  parameter Real R=1;
equation
  v=i*R;
end Resistor;
```

El comando **extends** declara que la clase actual extiende la clase **OnePort** y por lo tanto hereda todos los componentes públicos de dicha clase. De esta manera, la clase **Resistor** contiene también las variables **i**, **v**, los conectores **p**, **n**, y las 3 ecuaciones de la clase **OnePort**. En otras palabras, la clase **Resistor** será idéntica a la clase **Res** definida antes.

Con la misma idea, podemos desarrollar modelos de un capacitor, de un inductor y de una fuente constante como sigue:

```
model Capacitor
  extends OnePort;
  parameter Real C=1;
equation
  C*der(v)=i;
```

```
end Capacitor;

model Inductor
  extends OnePort;
  parameter Real L=1;
equation
  v=L*der(i);
end Inductor;

model ConstVolt
  extends OnePort;
  parameter Real V=1;
equation
  v=V;
end ConstVolt;
```

Luego, podemos armar un circuito RLC serie similar al de la sección introductoria como sigue:

```
model RLCserie
  Resistor R1;
  Capacitor C1;
  Inductor I1;
  ConstVolt S1;
  Ground G;
equation
  connect(S1.p,R1.p);
  connect(R1.n,I1.p);
  connect(I1.n,C1.p);
  connect(C1.n,S1.n);
  connect(S1.n,G.p);

end RLCserie;
```

1.7. Tipos de Datos y Redundancia

La clase predefinida **Real** tiene varios atributos, como el valor inicial (**start**) –que ya vimos como modificar– y otros que veremos luego. Uno de los atributos de la clase real es la unidad de medida (**unit**), que es una cadena con valor por defecto " ".

Pueden entonces crearse clases derivadas de la clase **Real** que contengan otras unidades, lo que es muy útil para la legibilidad de los modelos, interpretación de resultados y para tener información redundante al depurar.

Lo que sigue, es un ejemplo de su uso en una librería elemental de sistemas mecánicos traslacionales en una dimensión.

```
package Mecanicos
  type Velocity=Real(unit="m/s");
  type Position=Real(unit="m");
  type Force=Real(unit="N");
  type Length=Real(unit="m");
  type Distance=Real(unit="m");
  type Mass=Real(unit="Kg");

  connector Flange
    Position s;
    flow Force f;
  end Flange;

  partial model Rigid
    Flange a;
    Flange b;
    Position s;
    parameter Length L=0;
  equation
    s=a.s+L/2;
    b.s=a.s+L;
  end Rigid;

  partial model Compliant
    Flange a;
    Flange b;
    Distance srel;
    Force f;
  equation
    srel=b.s-a.s;
    a.f=-f;
    b.f=f;
  end Compliant;

  model SlidingMass
    extends Rigid;
    Velocity v;
    parameter Mass m=1;
  equation
    a.f+b.f=m*der(v);
    der(s)=v;
  end SlidingMass;

  model Spring
    extends Compliant;
    parameter Real k(unit="N/m")=1;
```

```
    parameter Length srel0=0;
  equation
    f=k*(srel-srel0);
  end Spring;

  model Damper
    extends Compliant;
    parameter Real bf(unit="N.m/s")=1;
  equation
    f=bf*der(srel);
  end Damper;

  model Fixed
    Flange a;
    parameter Position s0=0;
  equation
    a.s=s0;
  end Fixed;

  model SDM
    SlidingMass M(s(start=1));
    Spring S;
    Damper D;
    Fixed F;
  equation
    connect(S.a,F.a);
    connect(D.a,F.a);
    connect(S.b,M.a);
    connect(D.b,M.a);
  end SDM;
end Mecanicos;
```

Notar que en el amortiguador y en el resorte, los parámetros los declaramos de la clase **Real** pero modificando el atributo **unit** en la propia declaración.

Capítulo 2

Modelado Avanzado con Modelica

2.1. Bloques y Causalidad

Los diagramas de bloques son una herramienta de modelado muy útil en diversos dominios y aplicaciones. Son una manera sencilla de expresar relaciones matemáticas entre variables ampliamente utilizada en control, comunicaciones, procesamiento de señales, etc. De hecho, en un ejemplo del capítulo anterior, armamos el sistema de dos filtros en cascada como si fuera un diagrama de bloques.

Sin embargo, en dicho ejemplo, vimos que podíamos invertir causalmente el sistema forzando la salida del segundo filtro a un valor predefinido. Si bien esta flexibilidad puede ser una ventaja en ciertos problemas, el modelo que entonces construimos no era realmene un diagrama de bloques ya que evidentemente no respetaba las relaciones causales entre las variables.

En muchos casos, ya sea por cuestiones de ventaja computacional, para prevenir conexiones incorrectas, o para facilitar la legibilidad, es importante establecer y respetar las relaciones causales. Por esto, Modelica cuenta con clases y modificadores que permiten restringir y/o predefinir la causalidad con que se calculan las ecuaciones.

Para ilustrar esta funcionalidad de Modelica, construiremos una librería de bloques para modelar subsistemas como diagramas de bloques. Para esto, comenzaremos definiendo conectores causales.

```
connector InPort = input Real;
connector OutPort = output Real;
```

Notar que estamos definiendo los conectores como una extensión de la clase **Real**. Además, el modificador **input** dice que la variable en cuestión es una entrada, es decir, se calcula de manera externa al componente al que pertenece.

Recíprocamente, el modificador **output** expresa lo contrario, es decir, que la variable se calcula en el propio componente.

Utilizando estos conectores, podemos crear un modelo parcial para un bloque SISO:

```
partial block SISO
  InPort u;
  OutPort y;
end SISO;
```

Notar que utilizamos **block** en lugar de **model** para definir la clase. La clase **block** sirve para restringir los modelos de forma tal que sean causales, es decir, que tengan definidas las variables como entradas y salidas. En este caso, los conectores juegan el rol de entrada y salida, ya que están definidos con el modificador **input** y **output**.

Extendiendo el bloque SISO, podemos fácilmente crear bloques ganancia, integradores, etc.:

```
block Gain
  extends SISO;
  parameter Real K=1;
equation
  y=K*u;
end Gain;

block Integrator
  extends SISO;
equation
  der(y)=u;
end Integrator;
```

Podemos también crear, de manera más directa, un bloque tipo fuente que calcule una entrada constante:

```
block ConstSource
  OutPort y;
  parameter Real U=1;
equation
  y=U;
end ConstSource;
```

Finalmente, utilizando estos bloques podemos crear un Diagrama de Bloques de un oscilador de una manera muy sencilla:

```
model Oscil
  Gain G1(K=-1);
  Integrator I1(y(start=1));
  Integrator I2;
```

```
equation
  connect(G1.y,I1.u);
  connect(I1.y,I2.u);
  connect(I2.y,G1.u);
end Oscil;
```

2.2. Matrices y Vectores

En muchos dominios y aplicaciones, es conveniente expresar ciertas variables mediante magnitudes vectoriales. Más aún, en algunos casos, puede ser conveniente contar con arreglos de componentes (por ejemplo, para algunos sistemas de parámetros distribuidos).

Para estos casos, Modelica permite definir tanto vectores como matrices de una manera relativamente simple. Como veremos, los elementos de dichos vectores y matrices pueden en principio pertenecer a cualquier clase.

Para explicar las principales características de Modelica en el manejo de vectores y matrices, continuaremos con el ejemplo de la sección anterior, construyendo ahora bloques MIMO (Multiple Input, Multiple Output).

Comenzaremos entonces definiendo un bloque parcial de tipo MIMO:

```
partial block MIMO
  parameter Integer nin=1;
  parameter Integer nout=1;
  InPort u[nin];
  OutPort y[nout];
end MIMO;
```

Como vemos, el bloque MIMO cuenta con dos parámetros (**nin** y **nout**)que expresan el número de entradas y de salidas del bloque. La declaración **InPort u[nin]** dice que **u** es un vector de **nin** objetos de la clase **Inport**.

De esta forma, el bloque MIMO cuenta con un vector de puertos de entrada de dimensión **nin** y un vector de puertos de salida de dimensión **nout**.

A partir de este bloque parcial, construiremos un bloque que calcule un modelo de Espacio de Estados:

```
block StateSpace
  parameter Real A[:, size(A, 1)]=identity(2);
  parameter Real B[size(A, 1), :]=[1; 1];
  parameter Real C[:, size(A, 1)]=[1, 1];
  parameter Real D[size(C, 1), size(B, 2)]=zeros(size(C, 1), size(B, 2));
  extends MIMO(final nin=size(B, 2), final nout=size(C, 1));
  output Real x[size(A, 1)];

equation
  der(x)=A*x+B*u;
  y=C*x+D*u;
```

```
end StateSpace;
```

Como vemos, el bloque **StateSpace** tiene cuatro parámetros. El primero es una matriz llamada **A**. La función **size(A, 1)** devuelve el número de filas de la matriz **A**. Por lo tanto, estamos diciendo que **A** puede tener cualquier cantidad de filas (por el operador **:**) pero en número de columnas debe ser igual al número de filas. En otras palabras, estamos obligando a que **A** sea una matriz cuadrada.

De manera similar, obligamos a que **B** y **C** tengan dimensiones coherentes con **A**, y que a su vez **D** tenga dimensión coherente con estas últimas.

Una vez definidos los parámetros, estamos heredando las propiedades del bloque parcial **MIMO** pero restringiendo el número de entradas y salidas de acuerdo a las dimensiones de **B** y **C**. En el comando **extends** además utilizamos el modificador **final**. Este sirve para que los parámetros correspondientes no puedan ser modificados en una instanciación subsecuente de la clase.

Luego, definimos el vector de estados **x** de manera que tenga dimensión acorde a la matriz **A**. Como estamos en un bloque, debemos además especificar que el estado es una salida (es decir, que se calcula en el propio bloque).

Por último, en la sección **equation** escribimos las ecuaciones correspondientes al bloque utilizando notación matricial compacta.

Cualquier usuario de Matlab o Scilab puede notar inmediatamente la similitud entre la notación matricial de Modelica con la de dichas herramientas. Sin embargo hay que tener cuidado con esto. Cuando en Modelica escribimos **X=[a,b]** estamos formando **X** como la matriz concatenación de los elementos **a** y **b**, lo que es una matriz de 2×1 (tiene 2 dimensiones). Esto no es lo mismo que un vector de dos componentes (que una sóla dimensión). Para especificar un vector lo correcto es escribir **X={a,b}**.

Utilizando entonces el bloque **StateSpace** y el bloque **ConstSource** de la sección anterior como entrada, podemos armar el siguiente Diagrama de Bloques

```
model DB1
  StateSpace SS1(A=[0,1;-1,-1],B=[0;1],C=[0,1;1,1]);
  ConstSource CS1;
equation
  connect(CS1.y,SS1.u[1]);
end DB1;
```

Notar que aunque en este caso el bloque **StateSpace** tiene una sola entrada, debimos realizar la conexión con la fuente explicitando que la conexión es entre la salida de la fuente y la primer componente de la entrada del bloque **StateSpace**. De otra forma tendríamos una incompatibilidad de tipos de dato.

Veamos ahora un caso más complejo de utilización de arreglos de componentes. En este caso, haremos una pequeña librería de componentes de sistemas mecánicos en 2D. Para esto, comenzaremos definiendo (como en el capítulo anterior) nuevos tipos de datos:

```
  type Acceleration=Real(unit="m/s/s");
  type Velocity=Real(unit="m/s");
```

```
  type Position=Real(unit="m");
  type Force=Real(unit="N");
  type Length=Real(unit="m");
  type Distance=Real(unit="m");
  type Mass=Real(unit="Kg");
  type Acceleration2D=Acceleration[2];
  type Velocity2D=Velocity[2];
  type Position2D=Position[2];
  type Force2D=Force[2];
```

Como vemos, además de redefinir los tipos del capítulo anterior para mecánica traslacional, hemos extendido los mismos obteniendo los tipos **Velocity2D**, **Force2D** y **Position2D**, que son vectores de dimensión 2.

Utilizando estas nuevas clases, podemos plantear un conector 2D como sigue

```
connector Flange2D
  Position2D s;
  flow Force2D f;
end Flange2D;
```

y a partir del mismo, modelos de un enganche fijo y de una masa puntual:

```
model Fixed2D
  Flange2D flange_a;
  parameter Position2D s0={0,0};
equation
  flange_a.s=s0;
end Fixed2D;

model PointMass2D
  Flange2D flange_a;
  Position2D s;
  Velocity2D v;
  parameter Mass m=1;
equation
  flange_a.f=m*der(v);
  der(s)=v;
  s=flange_a.s;
end PointMass2D;
```

Notar que en el enganche fijo definimos el vector **s0** utilizando llaves en lugar de corchetes. De otra manera, hubieramos definido una matriz de 2×1 en lugar de un vector lo que hubiera conllevado un error de tipo.

Como hicimos con los modelos traslacionales, podemos ahora también definir un modelo parcial **Compliant2D** para luego extenderlo a las clases **Damper2D** y **Spring2D**:

```
partial model Compliant2D
  Flange2D flange_a;
  Flange2D flange_b;
  Length s_rel;
  Force f;
  Force2D f2;
equation
  s_rel=sqrt((flange_b.s[1]-flange_a.s[1])^2+(flange_b.s[2]-flange_a.s[2])^2);
  f2[1]=f*(flange_b.s[1]-flange_a.s[1])/s_rel;
  f2[2]=f*(flange_b.s[2]-flange_a.s[2])/s_rel;
  flange_a.f=-f2;
  flange_b.f=f2;
end Compliant2D;

model Spring2D
  extends Compliant2D;
  parameter Length s_rel0=0;
  parameter Real k(unit="N/m")=1;
equation
  f=k*(s_rel-s_rel0);
end Spring2D;

model Damper2D
  extends Compliant2D;
  parameter Real b(unit="N.s/m")=1;
equation
  f=b*der(s_rel);
end Damper2D;
```

Podemos también plantear el modelo de una fuerza constante:

```
model ConstForce2D
  Flange2D flange_a;
  parameter Force2D F={1,1};
equation
  flange_a.f=-F;
end ConstForce2D;
```

Y se puede fácilmente hacer el modelo de una barra sin masa:

```
model Bar2D
  extends Compliant2D;
  parameter Length L=1;
equation
  s_rel=L;
end Bar2D;
```

Utilizando estos componentes, podemos fácilmente construir el modelo de un péndulo simple:

```
model Pendulum
  Fixed2D F;
  PointMass2D M(s(start={2,-10}));
  ConstForce2D G(F={0,-9.8});
  Bar2D B(L=10.2);
equation
  connect(F.flange_a,B.flange_b);
  connect(B.flange_a,M.flange_a);
  connect(G.flange_a,M.flange_a);
end Pendulum;
```

También podemos agregar elasticidad y fricción al cable:

```
model Pendulumb
  Fixed2D F;
  PointMass2D M(s(start={2,-10}));
  ConstForce2D G(F={0,-9.8});
  Spring2D S(s_rel0=10,k=10);
  Damper2D D;
equation
  connect(F.flange_a,S.flange_b);
  connect(S.flange_a,M.flange_a);
  connect(G.flange_a,M.flange_a);
  connect(F.flange_a,D.flange_a);
  connect(D.flange_b,M.flange_a);
end Pendulub;
```

Podemos también hacer un modelo de un doble péndulo como sigue:

```
model Pendulum2
  Fixed2D F;
  PointMass2D M1(s(start={2,-10}));
  PointMass2D M2(s(start={2,-20}));
  ConstForce2D G1(F={0,-9.8});
  ConstForce2D G2(F={0,-9.8});
  Spring2D S1(s_rel0=10,k=10);
  Spring2D S2(s_rel0=10,k=10);
  Damper2D D1;
  Damper2D D2;
equation
  connect(F.flange_a,S1.flange_b);
  connect(S1.flange_a,M1.flange_a);
  connect(G1.flange_a,M1.flange_a);
  connect(M1.flange_a,S2.flange_a);
  connect(S2.flange_b,M2.flange_a);
  connect(G2.flange_a,M2.flange_a);
  connect(F.flange_a,D1.flange_a);
```

```
  connect(D1.flange_b,M1.flange_a);
  connect(M1.flange_a,D2.flange_a);
  connect(D2.flange_b,M2.flange_a);
end Pendulum2;
```

El paralelo mecánico masa-amortiguador puede también utilizarse como un modelo único:

```
model SpringDamper2D
  Flange2D flange_a;
  Flange2D flange_b;
  Spring2D S;
  Damper2D D;
equation
  connect(S.flange_a,flange_a);
  connect(D.flange_a,flange_a);
  connect(S.flange_b,flange_b);
  connect(D.flange_b,flange_b);
end SpringDamper2D;
```

lo que puede utilizarse a su vez para simplificar el modelo anterior.

2.3. Funciones y Algoritmos

Otra clase restringida de Modelica es la correspondiente a las funciones. Las funciones de Modelica son similares a las funciones de cualquier lenguaje de programación: tienen un cierto número de argumentos de entrada y devuelven un cierto número de argumentos de salida. El proceso de cálculo de las salidas en tanto, se lleva a cabo mediante un algoritmo especificado a partir de la palabra clave **algorithm**.

Para ilustrar el uso de funciones, agregaremos una función que calcule la distancia entre dos posiciones de la librería Mecanicos2D que creamos en la sección anterior.

```
function Distance2D
  input Position2D s1;
  input Position2D s2;
  output Distance d;
algorithm
  d:=sqrt((s1[1]-s2[1])^2+(s1[2]-s2[2])^2);
end Distance2D;
```

Notar que la función tiene dos posiciones 2D (vectores reales de dos elementos) como entradas (**s1** y **s2**) y una distancia (real escalar) como salida.

En lugar de **equation**, la función tiene una sección **algorithm**. Dentro del algoritmo, en lugar de ecuaciones tenemos asignaciones, explicitadas por el símbolo **:=**.

Utilizando esta función, podemos redefinir el modelo **Compliant2D** como sigue:

```
partial model Compliant2D
  Flange2D flange_a;
  Flange2D flange_b;
  Length s_rel;
  Force f;
  Force2D f2;
equation
  s_rel=Distance2D(flange_a.s,flange_b.s);
  f2[1]=f*(flange_b.s[1]-flange_a.s[1])/s_rel;
  f2[2]=f*(flange_b.s[2]-flange_a.s[2])/s_rel;
  flange_a.f=-f2;
  flange_b.f=f2;
end Compliant2D;
```

Un detalle importante a tener en cuenta es que si bien la función **Distance2D** contiene relaciones causales entre **d**, **s1** y **s2**, estas relaciones causales son internas a la función. La ecuación **s_rel=Distance2D(flange_a.s,flange_b.s)** no implica a priori ninguna relación causal entre las variables involucradas.

Las funciones de Modelica pueden tener algoritmos más complejos. Por ejemplo, la siguiente función permite evaluar un polinomio arbitrario en un valor real arbitrario:

```
function PolyEval
  input Real a[:];
  input Real x;
  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 PolyEval;
```

Como podemos ver, esta función tiene dos argumentos: un vector real (**a**) y un número real **x**. La salida es un número real (**y**). Además, hay una variable auxiliar **xpower** declarada en la sección **protected**. Los componentes declarados en la sección **protected** son elementos privados de la clase, y que por lo tanto no se pueden acceder de manera externa (esto vale también para las clases **model** y **block**).

Luego, el algoritmo calcula de manera simple el valor del polinomio **y=a[1]+a[2]*x+a[3]*x*x+...**.

Utilizando esta función **PolyEval** podemos construir un modelo de amortiguador más general cuya fuerza se calcule como $F_{roce} = b_1 + b_2 v + b_3 v^2 + \ldots$. Un posible modelo es el siguiente:

```
model Damper2Db
  extends Compliant2D;
  parameter Real b[:]={0,1};
equation
  f=PolyEval(b,der(s_rel));
end Damper2Db;
```

Notar que utilizamos como valor por defecto de b el correspondiente a un rozamiento lineal con coeficiente unitario.

Por último, cabe mencionar que la estructura **for ... loop** que utilizamos en el algoritmo para evaluar el polinomio puede también usarse dentro de una sección de ecuaciones (en una clase tipo **model**) para generar conjuntos de ecuaciones.

Por ejemplo, si queremos conectar un arreglo de conectores de un modelo con un arreglo de conectores de otro modelo, lo más simple es utilizar una estructura **for ... loop**.

2.4. Campos Físicos y Conexiones Virtuales

Un detalle un poco molesto del ejemplo del doble péndulo es que para cada una de las dos masas debimos conectar una fuente con la fuerza de gravedad. Este detalle sería crítico si en lugar de dos hubiéramos tenido un sistema con muchas masas.

Una alternativa simple para evitar esto es definir directamente la aceleración de la gravedad **g={0,-9.8}** dentro del modelo de la masa puntual y luego sumar la fuerza **m*g** en la ecuación que describe la segunda ley de Newton.

Si bien esta alternativa funciona, nuestro nuevo modelo de masa puntual funcionará sólamente suponiendo que la gravedad es de 9.8. Una solución más genérica es tener en cuenta que la masa está dentro de un campo gravitatorio que impone una determinada aceleración **g**, determinada por el modelo al que pertenece la masa.

Esto se resuelve en Modelica con los modificadores **inner** y **outer**. Cuando un objeto se declara con el modificador **outer**, su definición debe encontrarse en la clase superior mediante el modificador **inner**.

Por ejemplo, el modelo de masa puntual con la gravedad definida por el entorno quedaría como sigue:

```
model PointMass2Db
  Flange2D flange_a;
  Position2D s;
```

```
  Velocity2D v;
  outer parameter Acceleration2D g;
  parameter Mass m=1;
equation
  flange_a.f+m*g=m*der(v);
  der(s)=v;
  s=flange_a.s;
end PointMass2Db;
```

Luego, el modelo de los dos péndulos sería el que sigue

```
model Pendulum2b
  inner parameter Acceleration2D g={0,-9.8};
  Fixed2D F;
  PointMass2Db M1(s(start={2,-10}));
  PointMass2Db M2(s(start={2,-20}));
  SpringDamper2D SD1(S(s_rel0=10,k=10));
  SpringDamper2D SD2(S(s_rel0=10,k=10));
equation
  connect(F.flange_a,SD1.flange_b);
  connect(SD1.flange_a,M1.flange_a);
  connect(M1.flange_a,SD2.flange_a);
  connect(SD2.flange_b,M2.flange_a);
end Pendulum2b;
```

Supongamos que queremos ahora agregar la presencia de rozamiento con el aire. Una forma de hacerlo es conectar a cada masa un elemento de un solo conector que calcule una fuerza **flange_a.f=b*der(flange_a.s)**. Esto nos obligaría a agregar un componente por cada masa que tengamos en el sistema.

Otra manera de hacerlo sería agregar la fuerza de roce dentro del modelo de la masa puntual. Sin embargo, esto restringiría el modelo de la masa puntual a un dado modelo de fricción.

Podríamos también proceder como con la gravedad, y decir que la fricción es una fuerza que se calcula de manera externa a la masa, utilizando una función con el modificador **outer**, y declarando dicha función en el modelo correspondiente mediante **inner**.

Haremos algo de esto, pero para no tener que hacer tantas declaraciones en el modelo del péndulo, utilizaremos un modelo auxiliar que llamaremos **world** donde tendremos declaradas tanto la función que calcula la fricción como la aceleración de la gravedad.

Primero entonces definimos la función de fricción como una función convencional:

```
function Friction2D
  input Velocity2D v;
  output Force2D f;
```

```
  parameter Real b(unit="N.s/m")=1;
algorithm
  f:=b*v;
end Friction2D;
```

Luego, armamos nuestra clase **World2D** como sigue:

```
model World2D
  parameter Acceleration2D g={0,-9.8};
  function Friction=Friction2D;
end World2D;
```

Modificamos también el modelo de la masa puntual, declarando aquí el modelo **world** como una instancia de **World2D** definida de manera exterior:

```
model PointMass2Dc
  Flange2D flange_a;
  Position2D s;
  Velocity2D v;
  Force2D fric;
  outer World2D world;
  parameter Mass m=1;
equation
  fric=world.Friction(v);
  flange_a.f+m*world.g-fric=m*der(v);
  der(s)=v;
  s=flange_a.s;
end PointMass2Dc;
```

y por último, armamos el modelo del doble péndulo (esta vez quitamos los amortiguadores del cable):

```
model Pendulum2c
  inner World2D world(Friction(b=0.01));
  Fixed2D F;
  PointMass2Dc M1(s(start={2,-10}));
  PointMass2Dc M2(s(start={2,-20}));
  Spring2D S1(s_rel0=10,k=10);
  Spring2D S2(s_rel0=10,k=10);
equation
  connect(F.flange_a,S1.flange_b);
  connect(S1.flange_a,M1.flange_a);
  connect(M1.flange_a,S2.flange_a);
  connect(S2.flange_b,M2.flange_a);
end Pendulum2c;
```

Notar que fue necesario declarar **world** con el prefijo **inner** para que tenga correspondencia con el modificador **outer** de la masa puntual.

Si quisiéramos ahora cambiar la ley de rozamiento o la aceleración de la gravedad, símplemente debemos instanciar un modelo **world** distinto, sin necesidad de modificar para nada el modelo de masa puntual.

Este recurso de agregar un submodelo que define ciertas características del contexto se utiliza a menudo en la librería estándar de Modelica. De hecho, este es un recurso necesario en la librería de MultiBody Systems.

Capítulo 3

Discontinuidades y Sistemas Híbridos

Los sistemas continuos presentan a menudo discontinuidades en las ecuaciones que describen la dinámica. En muchos casos además, estas discontiuidades son producto de la interacción de un subsistema continuo con otro discreto (por ejemplo, en un sistema de control muestreado), lo que da lugar a los denominados *sistemas híbridos*.

En cualquiera de los casos, el problema requiere de un tratamiento especial en la etapa de modelado. Cuando ocurre una discontinuidad, los simuladores deben tener la información necesaria para advertirlo, ya que de otra manera cometerán errores numéricos inaceptables. Por esto, un lenguaje de modelado debe estar provisto de herramientas específicas para definir las situaciones en las que ocurren las mencionadas discontinuidades (eventos).

Por otra parte, en los sistemas híbridos, la descripción de los subsistemas discretos difiere en general de la de los subsistemas continuos, por lo que encontraremos también algunas diferencias nuevas en este sentido.

Finalmente, veremos algunas herramientas para describir sistemas de estructura variable, en los cuales los cambios producidos por las discontinuidades no solo afectan los parámetros sino también la estructura del sistema (causalidad y eventualmente el orden).

3.1. Eventos del Tiempo y del Estado

La estructura más simple para describir discontinuidades es la **if ... then ... else ...** . Como ejemplo simple de su utilización, comenzaremos describiendo el modelo de una pelotita que rebota contra el piso tal como se muestra en la Figura 3.1

```
model BBall
  Real y(start=1);
```

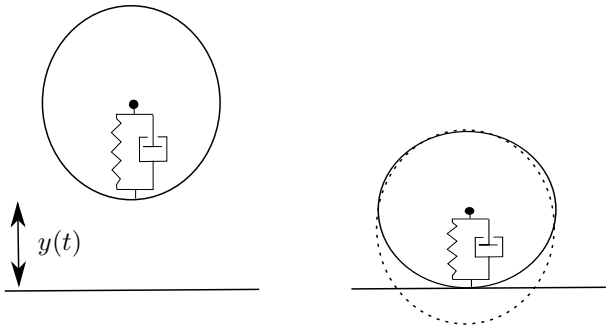

Figura 3.1: Esquema de la pelotita rebotando

```
  Real v;
  parameter Real m = 1;
  parameter Real k = 10000;
  parameter Real b = 10;
  parameter Real g = 9.8;
equation
  der(y)=v;
  m*der(v)=if y>0 then -m*g else -k*y-b*v-m*g;
end BBall;
```

Aunque el modelo se entiende inmediatamente, es importante aclarar que la condición **y>0** después del comando **if** es la que desencadena el evento de la discontinuidad. Por lo tanto, el simulador que utilicemos deberá garantizar que dará un paso exactamente en el instante en que se cumpla que **y=0**.

En este caso, la discontinuidad estuvo provocada por un *evento del estado*, ya que la condición del evento depende del estado **y**. Para representar eventos del tiempo, se puede proceder de manera totalmente análoga.

Por ejemplo, el siguiente sería el modelo de una fuente escalón para nuestra librería de Diagramas de Bloques desarrollada en el capítulo anterior:

```
block StepSource
   OutPort y;
   parameter Real T=1;
   parameter Real U=1;
equation
  y=if time<T then 0 else U;
end StepSource;
```

3.2. Sistemas de Tiempo y de Eventos Discretos

Los sistemas discretos se caracterizan porque la evolución de sus variables no es continua en el tiempo.

Modelica tiene algunas herramientas básicas para describir esta clase de sistemas: la estructura **when**, y las funciones **sample()** y **pre()**.

La estructura **when** contiene un conjunto de ecuaciones que están activas sólo cuando la condición de dicha estructura se torna verdadera. La función **sample**, en tanto, da como resultado **false** todo el tiempo, excepto una vez cada cierto período de tiempo (especificado en el argumento). Veamos como el uso combinado de estas dos estructuras permite hacer un bloque simple de muestreo:

```
block Sampler
  extends SISO;
  parameter Real T=1;
equation
  when sample(0,T) then
    y=u;
  end when;
end Sampler;
```

La función **sample** tiene dos argumentos. El segundo es el período, y el primero es el instante inicial de muestreo.

Las variables que se calculan dentro de la estructura **when** se interpretan como discretas. De hecho, es posible declararlas con el prefijo **discrete**. Una restricción importante es que dentro de la estructura **when** las ecuaciones deben estar definidas de manera causal. Esto es así ya que de otra manera se pueden producir indeterminaciones.

Un modelo un poco más complejo, es la versión discreta del bloque **StateSpace** visto en el capítulo anterior:

```
block DiscStateSpace
  parameter Real A[:, size(A, 1)]=identity(2);
  parameter Real B[size(A, 1), :]=[1; 1];
  parameter Real C[:, size(A, 1)]=[1, 1];
  parameter Real D[size(C, 1), size(B, 2)]=zeros(size(C, 1), size(B, 2));
  parameter Real T=1;
  extends MIMO(final nin=size(B, 2), final nout=size(C, 1));
  output Real x[size(A, 1)];
equation
  when sample(0,T) then
    x=A*pre(x)+B*u;
    y=C*pre(x)+D*u;
  end when;
end DiscStateSpace;
```

La función **pre** se refiere al valor de una variable antes de la ocurrencia del evento, y cumple un rol esencial en el modelado de cualquier sistema de tiempo o eventos discretos.

La condición dentro de la estructura **when** no tiene porque ser una función **sample**. Por ejemplo, el siguiente bloque representa un muestreador asíncrono, que toma muestras cada vez que el valor actual de la señal continua se separa de la última muestra en un valor fijo **dQ**:

```
block AsyncSampler
  extends SISO;
  parameter Real dQ=1;
equation
  when abs(y-u)>dQ then
    y=u;
  end when;
end AsyncSampler;
```

3.3. Componentes de Conmutación

Hasta aquí hemos visto como representar relaciones discontinuas entre variables a través de la estructura **if ... then ... else**. Aunque esto es simple en principio, su utilización dentro de componentes de distintos dominios físicos no es tan inmediata en muchos casos.

Para ilustrar las dificultades, consideremos en primer lugar un diodo ideal, con una característica Volt-Ampère como la que muestra la Fig.3.2.

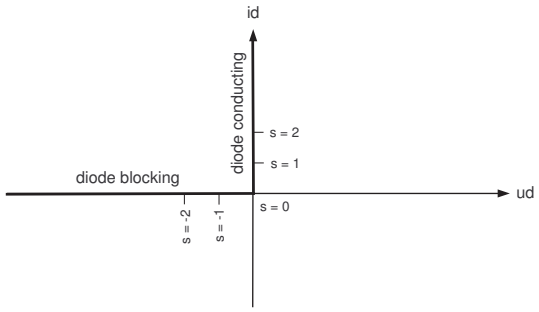

Figura 3.2: Característica Volt-Ampère de un diodo ideal.

El problema en este caso es que no podemos escribir una única ecuación que vincule las variables **v** e **i**.

El truco entonces es introducir una variable auxiliar **s** que juegue el rol de variable paramétrica como se muestra en la Fig.3.2. Con esta variable auxiliar, podemos plantear el siguiente modelo (utilizando como base la librería de componentes eléctricos que desarrollamos en el primer capítulo).

```
model IdealDiode
  extends OnePort;
protected
```

```
  Real s;
equation
  v=if s>0 then 0 else s;
  i=if s>0 then s else 0;
end IdealDiode;
```

Podemos entonces armar una fuente senoidal y construir un rectificador utilizando el modelo anterior como sigue:

```
model SinSource
  extends OnePort;
  parameter Real A=1;
  parameter Real f=1;
  constant Real pi=6*asin(0.5);
equation
  v=A*sin(2*pi*f*time);
end SinSource;

function asin
  input Real x;
  output Real y;
  parameter Integer n=10;
protected
  Real sk;
algorithm
  sk:=x;
  y:=x;
  for k in 3:2:2*n-1 loop
    sk:=sk*x^2*(k-2)^2/(k-1)/k;
    y:=y+sk;
  end for;
end asin;

model Rectif
  Resistor R1;
  Resistor R2(R=10);
  Capacitor C1;
  IdealDiode D;
  SinSource S1;
  Ground G;
equation
  connect(S1.p,R1.p);
  connect(R1.n,D.p);
  connect(D.n,C1.p);
  connect(C1.n,S1.n);
  connect(S1.n,G.p);
```

```
  connect(C1.p,R2.p);
  connect(C1.n,R2.n);
end Rectif;
```

Notar que de paso aprovechamos para introducir una nueva función (más compleja que las que vimos en el capítulo anterior) **asin** que calcula la inversa del seno a través de una serie.

Si bien este modelo funciona perfectamente, la siguiente variante (sin resistencia en la serie entre la fuente y el diodo) tendrá problemas debido a la singularidad estructural provocada por las conmutaciones causales:

```
model Rectif2
  Resistor R2(R=10);
  Capacitor C1;
  IdealDiode D;
  SinSource S1;
  Ground G;
equation
  connect(S1.p,D.p);
  connect(D.n,C1.p);
  connect(C1.n,S1.n);
  connect(S1.n,G.p);
  connect(C1.p,R2.p);
  connect(C1.n,R2.n);
end Rectif2;
```

Para evitar estos inconvenientes, el modelo de *diodo ideal* de Modelica no es tan ideal, sino que contiene una resistencia de conducción (muy baja) y otra de apagado (muy alta). Esto trae problemas de rigidez al sistema, por lo que en general deben utilizarse métodos implícitos para simular (de hecho, tanto Dymola como OpenModelica tienen a DASSL como método de integración por defecto).

Volviendo ahora a la librería de componentes mecánicos del primer capítulo, podemos también utilizar las ideas vistas para plantear un modelo de contacto elástico (con resorte y amortiguador) discontinuo como el mostrado en la Figura 3.3.

```
model ElastoGap
  extends Compliant;
  Boolean Contact;
  parameter Real k=1;
  parameter Real bf=1;
  parameter Real srel_0=0;
equation
  Contact=srel<srel_0;
  f=if Contact then bf*der(srel)+k*(srel-srel_0) else 0;
end ElastoGap;
```

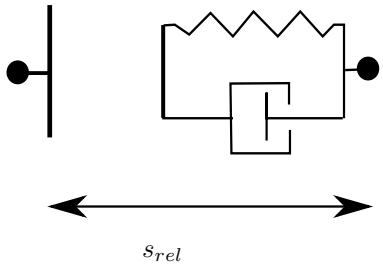

Figura 3.3: Contacto elástico discontinuo

Notar que en este modelo usamos una variable tipo **Boolean** (lo hicimos para mejorar la legibilidad, en realidad podríamos haber evitado su uso).

Utilizando este nuevo componente, es ahora muy sencillo hacer un modelo de la pelotita que rebota contra el piso de la Fig.3.1.

```
model BBall
  SlidingMass M(s(start=1));
  ElastoGap EG(k=100000,bf=10);
  ConstForce FG(F=-9.8);
  Fixed F;
equation
  connect(FG.a,M.b);
  connect(EG.b,M.a);
  connect(EG.a,F.a);
end BBall;
```

3.4. Sistemas de Estructura Variable

Consideremos ahora nuevamente el modelo del péndulo simple del capítulo anterior, en el paquete que construimos **Mecanicos2D**.

Agreguemos ahora la hipótesis adicional de que el cable que sostiene la masa puede romperse cuando la fuerza que hace es mayor que cierto parámetro **fmax**. En este caso, al romperse el cable, hay un cambio de estructura en el modelo (que incluye no sólo una conutación un cambio de orden). El modelo con el cable roto es más simple que el modelo **Pendulum** que vimos, ya que simplemente consiste de una masa y de la fuerza de gravedad:

```
model MassG
  PointMass2D M;
  ConstForce2D G(F={0,-9.8});
equation
  connect(G.flange_a,M.flange_a);
end MassG;
```

Una manera entonces de hacer un modelo que combine ambos modelos según si el cable esté roto o no es la siguiente:

```
model BreakingPend
  Position2D s;
  Boolean Broken(start=false);
  Pendulum P;
  MassG MG;
  parameter Force fmax=10;
equation
  Broken=P.B.f>fmax or pre(Broken);
  s=if Broken then MG.M.s else P.M.s;
  when Broken then
    reinit(MG.M.s,P.M.s);
    reinit(MG.M.v,P.M.v);
  end when;
end BreakingPend;
```

Notar que en este caso sólo nos interesa la posición del péndulo **s**. Cuando el cable está sano, esta posición coincide con la del modelo **Pendulum** mientras que cuando está roto coincide con la del modelo **MassG**.

Por otro lado, utilizamos el comando **reinit** que reinicializa el valor de una variable. Así, al romperse el cable, forzamos que la posición y velocidad de **MassG**, coincida con la del péndulo.

Bibliografía

[1] Modelica Association. *Tutorial - Modelica - A Unified Object-Oriented Language for Physical Systems Modeling.* Linköping, Sweden, 2000. Disponible en www.modelica.org.

[2] Modelica Association. *Language Specification - Modelica - A Unified Object-Oriented Language for Physical Systems Modeling.* Linköping, Sweden, 2005. Disponible en www.modelica.org.

[3] Peter Fritzon. *Tutorial - Introduction to Object-Oriented Modeling and Simulation with OpenModelica.* Linköping, Sweden, 2006. Disponible en www.modelica.org.