Modelica

Un langage de modélisation objet

Présentation Cirad - UMR System

13 février 2003

Ph. Reitz - LIRMM

reitz@lirmm.fr

Plan

- Motivations
  - Pourquoi Modelica
  - Pourquoi présenter Modelica dans cette UMR
- Modéliser avec Modelica
  - Principe général
  - Le langage
- Conclusions
  - Les apports potentiels
  - Les limites / manques

Motivations : pourquoi Modelica

(1/2)

- Un langage déclaratif, orienté objet
- Un langage spécialement adapté à la modélisation de systèmes
  - Applicable à tous les domaines
  - Modelica = UML pour les modélisateurs ?
- Un langage pour décrire des modèles hybrides
  - Hybride = mélange discret - continu

Motivations : pourquoi Modelica

(2/2)

- Vers un langage de modélisation normalisé
  - Les spécifications sont normalisées
  - Les outils informatiques (compilateurs) ont pour seule obligation de se conformer à ces spécifications
- Domaines d'application à ce jour
  - Mécatronique, Electronique, Systèmes de puissance, Hydraulique, Aérodynamique

Motivations : pourquoi présenter Modelica ici (1/2)

- Tout modèle ou toute simulation sur la plate-forme doit pouvoir être exprimé dans un langage à la fois
  - compréhensible par le modélisateur
  - traitable informatiquement
- La définition d'un langage impose de caractériser proprement tous les objets manipulés (sémantique)

Motivations : pourquoi présenter Modelica ici (2/2)

- Modelica est issu d'une communauté de modélisateurs, familière de l'approche objet (école scandinave)

Message : ce n'est pas Modelica en soit qui est intéressant, mais **la démarche** des modélisateurs qui le développent : simuler, c'est programmer $\Rightarrow$ recherche d'un langage de programmation adapté à leurs besoins

Modéliser / simuler avec Modelica : processus général (1/2)

1. définition d'un modèle du système
   - orienté composant
     - variables d'état discrètes et/ou continues
     - dynamique (temps continu) décrite par
       - équation différentielle
       - programme
     - héritage entre composants
       - composants réutilisables (bibliothèques de composants)
   - modèle = assemblage de composants
     - éditeurs graphiques

Modéliser / simuler avec Modelica : processus général (2/2)

2. définition des paramètres d'une simulation
3. compilation du modèle paramétré
   - mise à plat du modèle objet
   - production d'un système d'EDA hybride
4. exécution du programme de simulation
   - exploitation de solveurs d'EDA hybrides adaptés au problème

Modelica : le langage

- Concepts clés
  - une liste de constructions de types de base
    - Real, Integer, String, Boolean
    - Array, Enumeration
  - une notion générale de classe (**`class`**)
  - des formes de classes particulières

    `model`, **`record`**, **`block`**, **`connector`**, **`type`**, **`package`**, **`function`**
  - dynamique : le temps évolue continûment. Il est partagé par tous les composants d'un modèle

Modelica : les classes

Une classe définit les propriétés communes aux composants qui y sont rattachés

- reprise de propriétés de classes existantes (héritage multiple)
  - telles quelles
  - avec spécialisation (types plus spécifiques, valuation de paramètres)

- variables d'état
  - variabilité
    - **`constant`** : valeur imposée dans le modèle (immuable pour toute simulation)
    - **`parameter`** : valeur libre devant être spécifiée en début de simulation (immuable pour cette simulation)
    - **`discrete`** : valeur réelle pouvant changer lors d'événements discrets
  - causalité
    - **`input`** : valeur devant provenir d'une variable output
    - **`output`** : valeur produite
  - flux
    - **`flow`** : variable du genre flux (voir *connecteurs*)

- équations
  - liste d’équations liant les variables d’état
    - définissent la dynamique des variables continues
- algorithme
  - description d ’un programme
    - permet de calculer les valeurs de variables discrètes

Modelica : classes particulières

- **`type`** : définit un type, par extension d'un type prédéfini par exemple

Exemple :

```
type Voltage = Real(quantity="Voltage", unit="V");
```

- **`connector`** : définit un connecteur
  - des variables d'état sont des potentiels
    - lorsque deux connecteurs sont reliés, la connexion impose que les variables potentiels sont égales
  - des variables d'état sont des flux (**`flow`**)
    - lorsque deux connecteurs sont reliés, la connexion impose que les variables flux ont leur somme égale à 0
  - pas d'équations associées

Exemple :

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

- **`model`** : classe ne pouvant pas être exploitée comme connecteur

Exemples :

un modèle abstrait :

```
partial model TwoPins
    Pin     p, n;
    Voltage u;
    Current i;
equation
    0 = p.i + n.i;
    u = p.v - n.v;
    i = p.i;
end TwoPins;
```

des modèles concrets :

```
model Resistor
    extends TwoPins;
    parameter Resistance R;
equation
    u = R*i;
end Resistor;

model Capacitor
    extends TwoPins;
    parameter Capacitance C;
equation
    C*der(u) = i;
end Capacitor;
```

Exemples (suite) :

```
model SinusSource
   extends   TwoPins;
   parameter Frequency F;
   parameter Voltage   A;
protected
   constant Real PI = 3.14159265358979;
equation
   u = A * sin(time * 2*PI*F);
end SinusSource;
```

- **`block`** : classe devant contenir au moins une variable causale (**`input`** ou **`output`**), non exploitable comme connecteur
- **`function`** : classe **`block`** ne contenant pas d'équation et contenant au plus un algorithme

```
function Distance
  parameter Integer N;
  input     Real[N] p1;
  input     Real[N] p2;
  output    Real    result;
algorithm
  result := 0;
  for i loop result := result +(p1[i]-p2[i])^2; end for;
  result := sqrt(result);
end Distance;
```

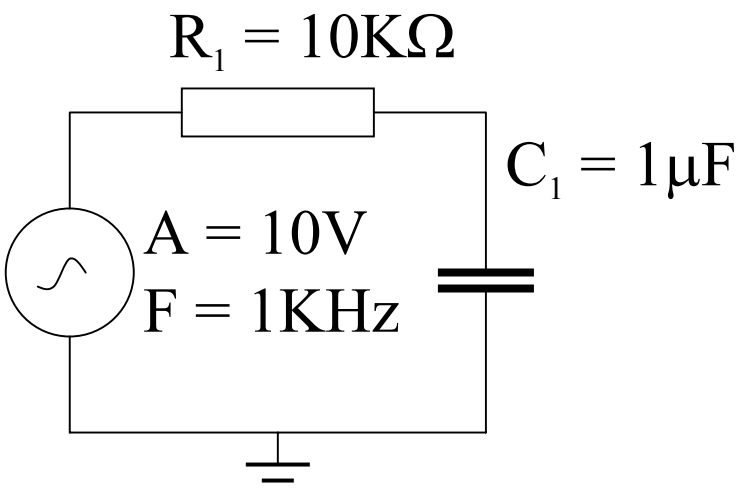
- **`record`** : classe sans équation, non exploitable comme connecteur

```
record Complex
  Real re, im;
end Complex;
```

- **`package`** : classe ne devant contenir que des définitions de classes ou de constantes

Un exemple de modèle

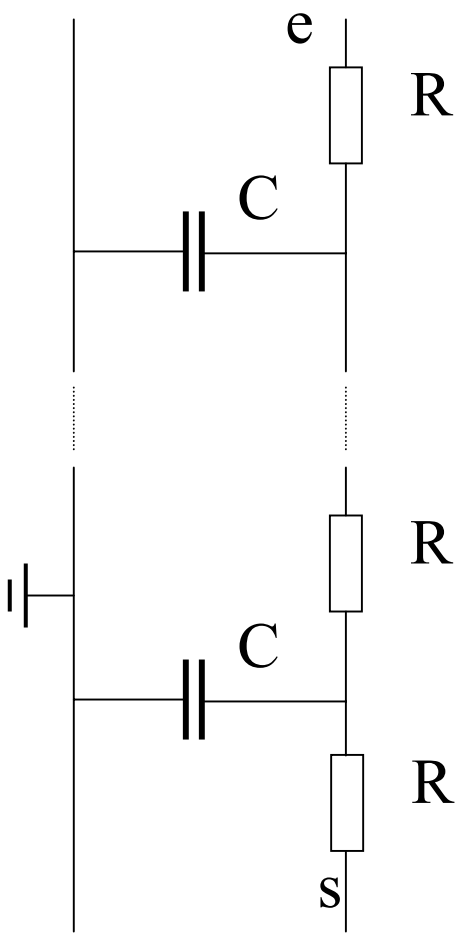
modèle de type EDO




```
model monCircuit
  Resistor    R1(R=1e4);
  Capacitor   C1(C=1e-6);
  SinusSource S(A=10, F=1000);
  Ground      G;
equation
  connect(S.p, R1.p);
  connect(R1.n, C1.p);
  connect(S.n, G);
  connect(C1.n, G);
end monCircuit;
```

Extension du même exemple

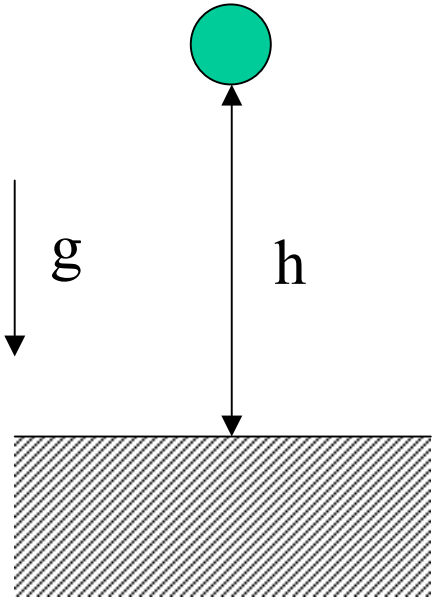
un nombre arbitraire de composants




```
model unEtageAtténuateur
  parameter Integer N;
  Pin             e, s;
  Resistor[N+1] R;
  Capacitor[N]  C;
  Ground          G;
equation
  connect(e, R[1].p);
  for i in 1:N loop
    connect(R[i].n, R[i+1].p);
    connect(R[i].n, C[i].p);
    connect(C[i].p, G);
  end for;
  connect(R[N+1].n, s);
end unEtageAtténuateur;
```

Un autre exemple

modèle de type EDA




```
model BalleRebondissante
  parameter Real e = 0.7;
  parameter Real g = 9.81;
            Real h (start=1);
            Real v;
equation
  der(h) = v;
  der(v) = -g;
  when h <= 0 then
    reinit(v, -e*pre(v));
  end when;
end BalleRebondissante;
```

Encore un exemple

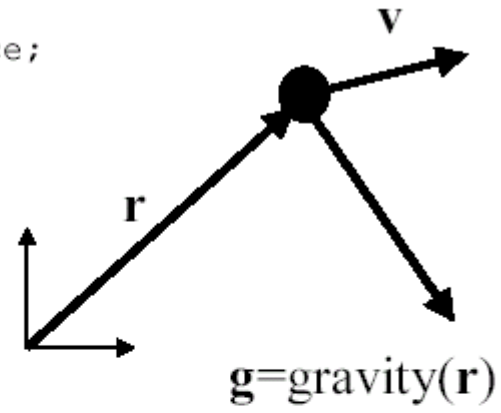
modélisation d'un champ (partage de variable)

```
model unObjet
  outer Real T0;
        Real T;
equation
  T = T0;
end unObjet;
```

```
model unEnvironnement
  inner      Real T0;
  parameter Real a=1;
equation
  T0 = sin(a*time);
end unEnvironnement;
```

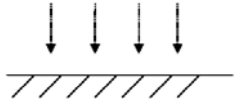
```
model monEnvironnement
  extends unEnvironnement(a=5);
  unObjet   o1, o2, o3;
     // o1.T = o2.T = o3.T = T0 = sin(5*time)
end monEnvironnement;
```

```
model Particle
   parameter Real m = 1;
   outer function gravity = gravityInterface;
   Real r[3](start = {1,1,0}) "position";
   Real v[3](start = {0,1,0}) "velocity";
equation
     der(r) = v;
   m*der(v) = m*gravity(r);
end Particle;
```

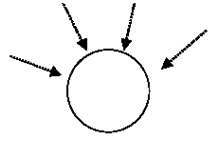



```
partial function gravityInterface
  input  Real r[3] "position";
  output Real g[3] "gravity acceleration";
end gravityInterface;
```

```
function uniformGravity
  extends gravityInterface;
algorithm
  g := {0, -9.81, 0};
end uniformGravity;
```



```
function pointGravity
   extends gravityInterface;
   parameter Real k=1;
protected
   Real n[3]
algorithm
   n := -r/sqrt(r*r);
   g := k/(r*r) * n;
end pointGravity;
```



```
model Composite1
   inner function gravity =
                  pointGravity(k=1);
   Particle p1, p2(r(start={1,0,0}));
end Composite1;
```

```
model Composite2
   inner function gravity =
                  uniformGravity;
   Particle p1, p2(v(start={0,0.9,0}));
end Composite2;
```

Travaux similaires
(non exhaustif...)

- Simulation
  - Scilab - Scicos (Inria)
    - libre et gratuit
  - gPROMS (Imperial College)
- Vérification
  - HyTech (Berkeley)

Conclusions (1/2)

- Nombreuses extensions prévues
  - Etendre le langage aux EDP
  - Eléments finis
  - Systèmes à structure variable
- Langage permettant de spécifier des séries de simulations (expériences)
  - Calibration
  - Tests de sensibilité

Conclusions (2/2)

- Avancées sur le plan algorithmique
  - Etude des systèmes hybrides (automates hybrides)
  - Définition de solveurs à large spectre
  - Caractériser des classes de modèles décidables
    - Sûreté (prouver qu'un état ne sera jamais atteint)
    - Contrôlabilité (prouver qu'un état pourra toujours être atteint à partir d'un état donné)

Outils existants

- Editeurs graphiques libres
- Bibliothèques de composants dans divers domaines
  - aucun en agronomie ☹
- Compilateurs
  - commerciaux pour l'instant
  - un projet *open-source* en cours

Informations utiles

- Source principale

  `http://www.Modelica.org`

- Abréviations
  - EDA = Equation Différentielle et Algébrique

    DAE = Differential and Algebraic Equation
  - EDO = Equation Différentielle Ordinaire

    ODE = Ordinary Differential Equation
  - EDP = Equation aux Dérivées Partielles

    PDE = Partial Differential Equation