Génération de code pour Scicos/Nsp par évaluation partielle

LMCS 2015 - Rennes le 24 novembre 2015

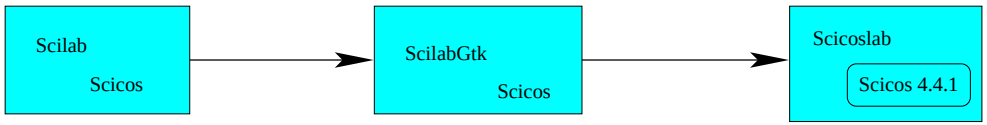
J.Ph Chancelier, Ramine Nikoukhah, Pierre Weis

CERMICS-ENPC, ALTAIR

Résumé

- Un générateur de code pour des diagrammes Scicos basé sur l'évaluation partielle
  - Décrire la sémantique des blocs dans un langage de script.
  - Un sous ensemble du langage Nsp.
- Un travail plus large sur la conception de compilateurs pour les langages de blocs (François Delebecque, Clément Franchini, Alan Layec and Pierre Weis).
- Participation à un projet FUI pour la génération de code embarqué.

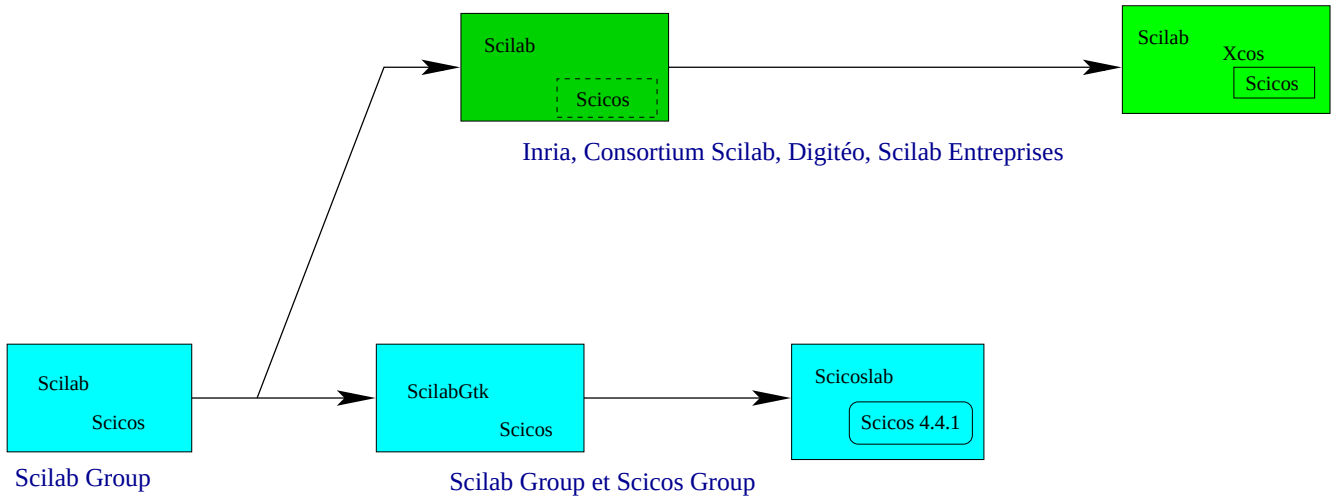
Scilab/Scicoslab/Nsp et Scicos/Xcos



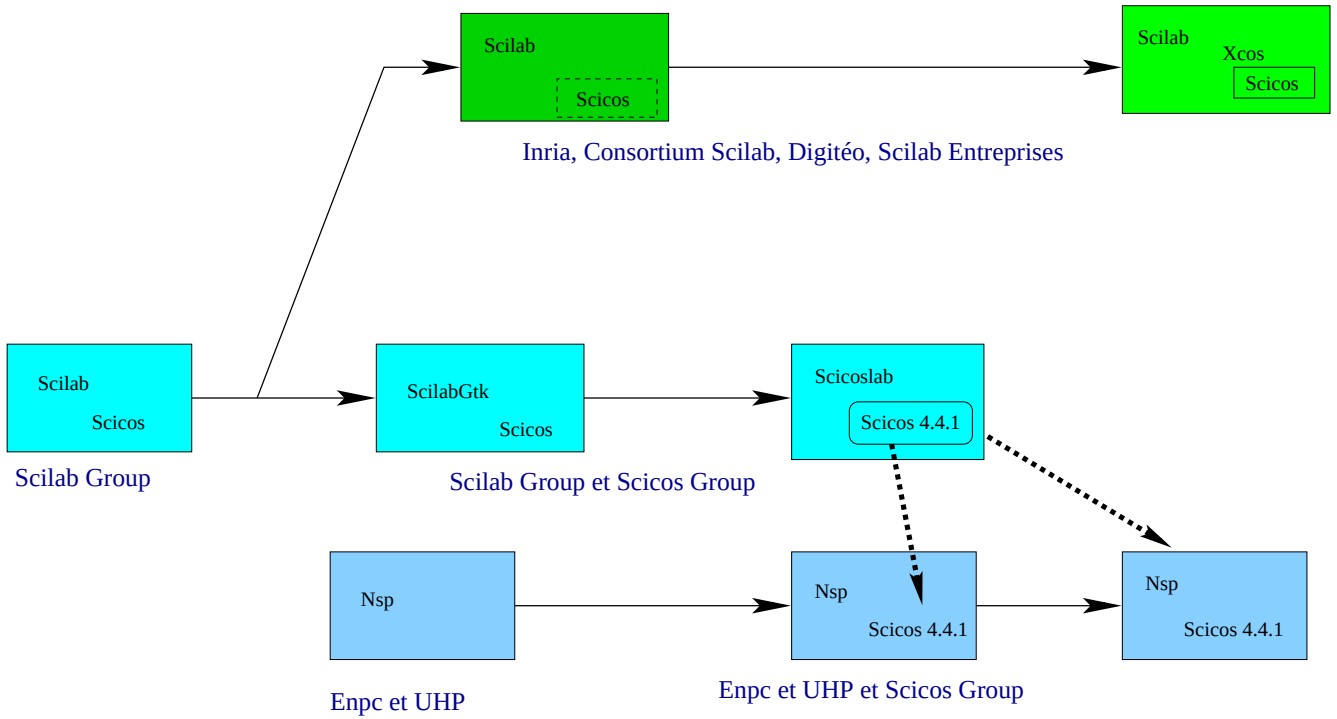
Scilab Group

Scilab Group et Scicos Group

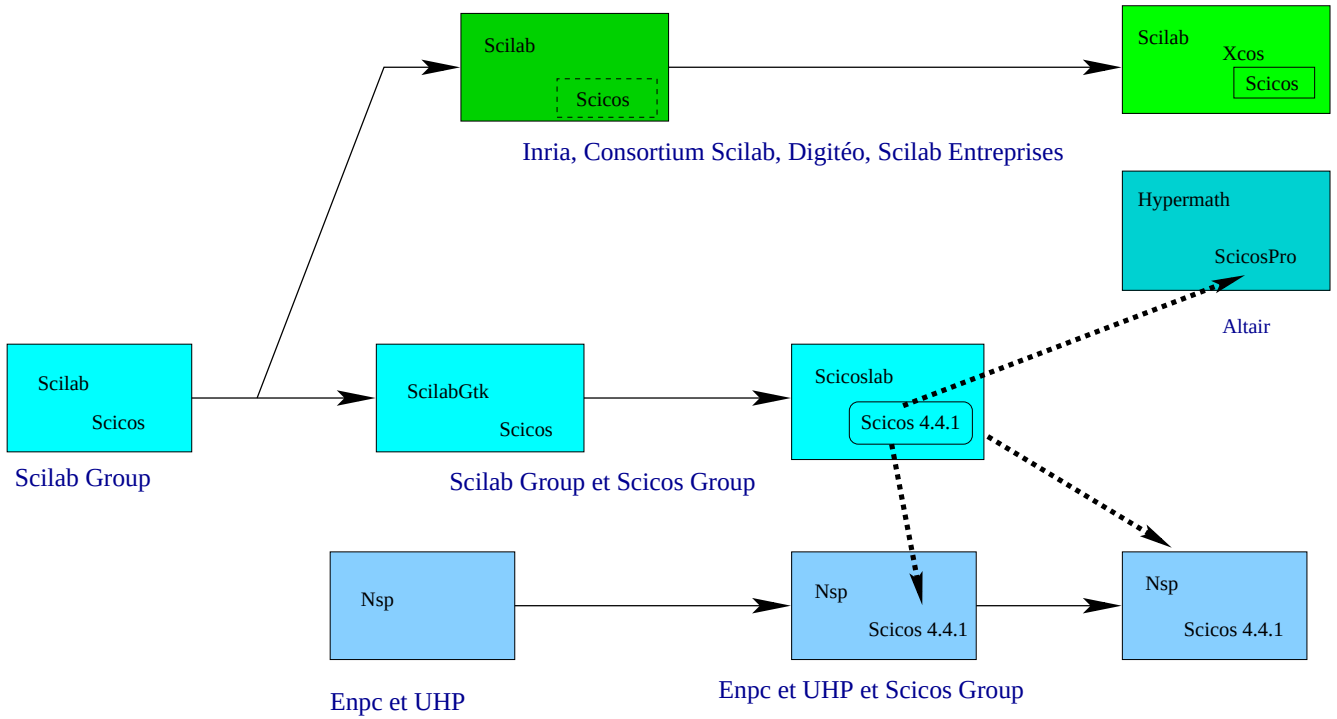
Scilab/Scicoslab/Nsp et Scicos/Xcos



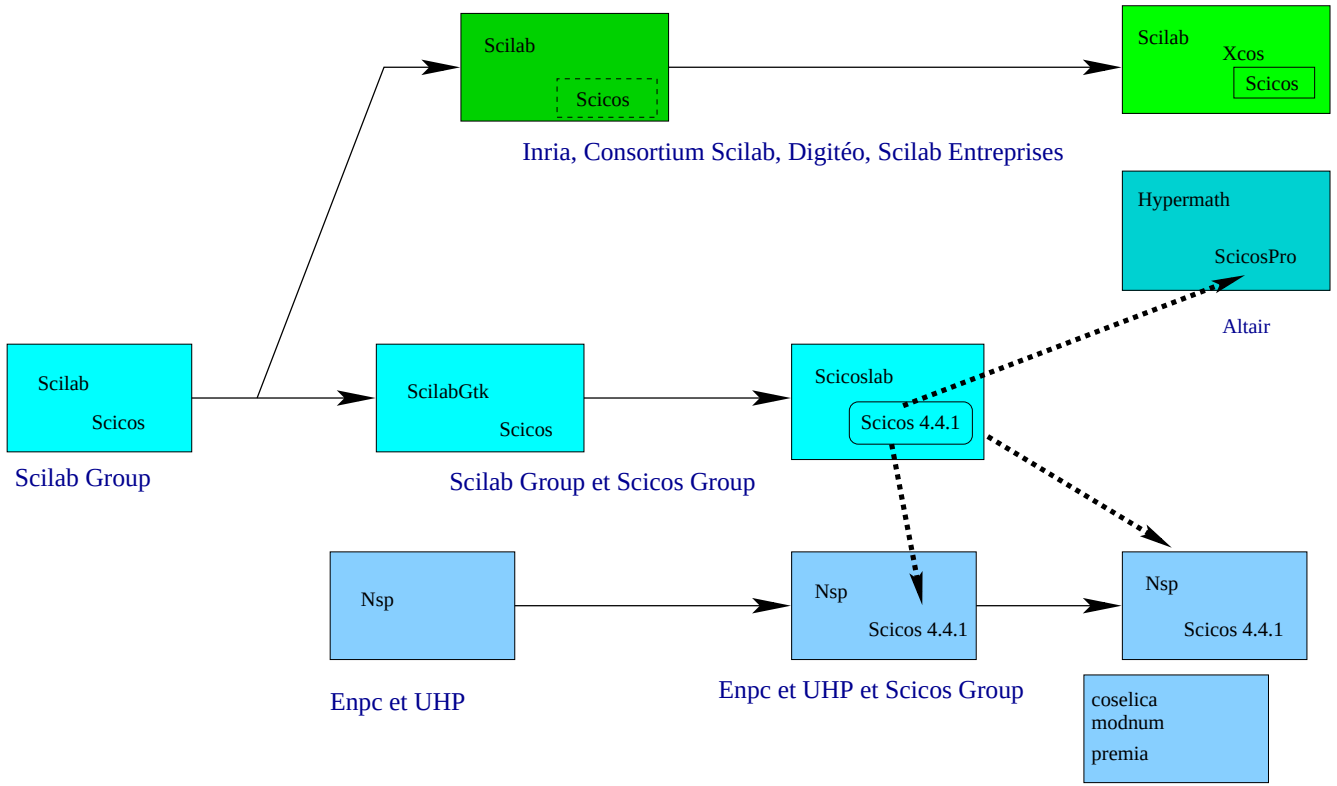
Scilab/Scicoslab/Nsp et Scicos/Xcos



Scilab/Scicoslab/Nsp et Scicos/Xcos



Scilab/Scicoslab/Nsp et Scicos/Xcos



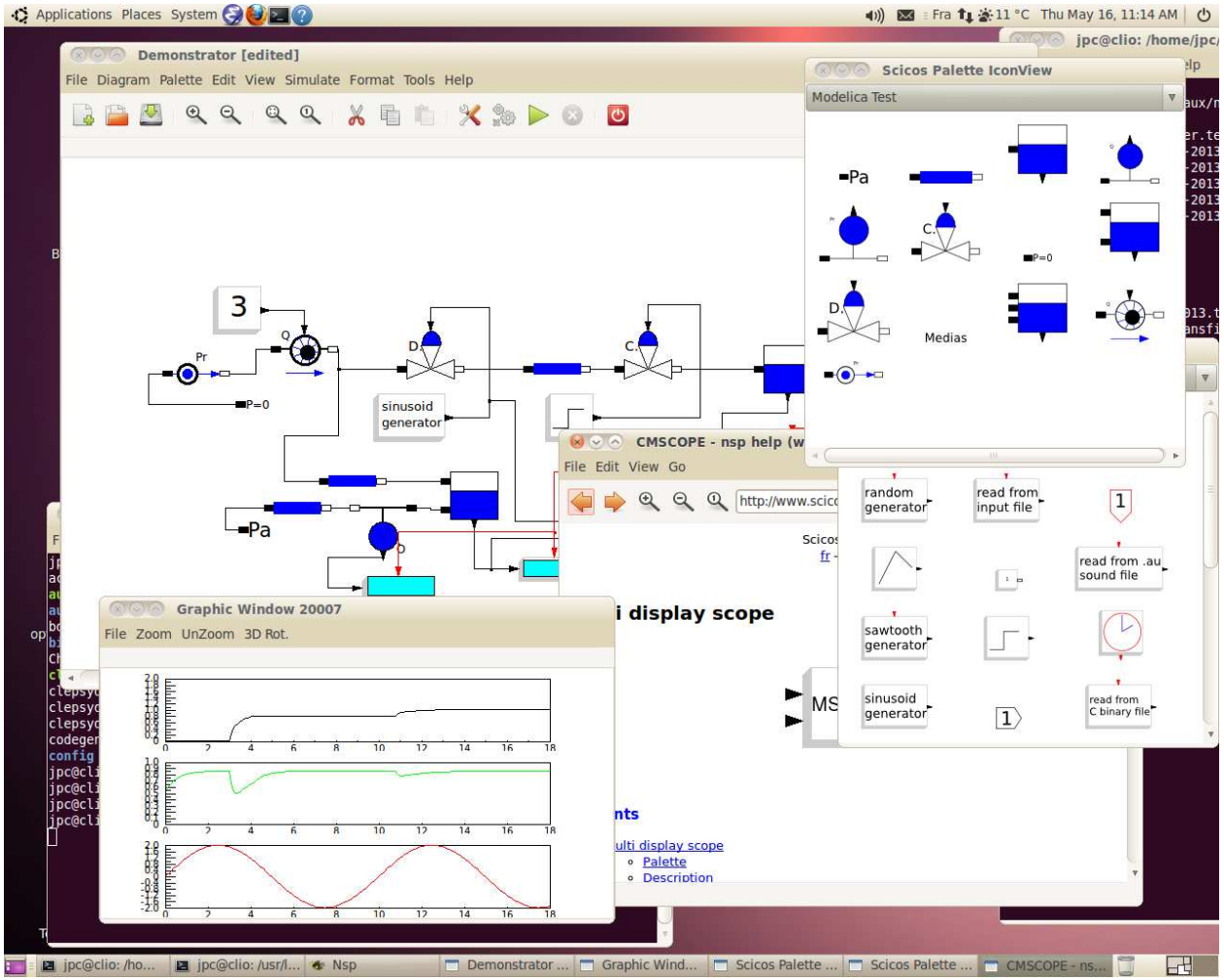
Nsp: Un langage de script à la Matlab

- Langage de script spécialisé pour le calcul numérique et GPL.
- Syntaxe très voisine de la syntaxe Scilab.
- Interprète en C
- Structures de données internes (système de classes écrit en C) et librairies associées (runtime pour un compilateur ou utilisation externe).
  - Langage d'interfaçage simple et étendu.
  - Possibilité de créer de nouveaux objets.
  - Facilite la création de boîtes à outils

Scicos: Un langage de blocs pour les systèmes hybrides

- Bloc Édition Simulation Génération de code pour des systèmes dynamiques hybrides
  - ´Éditeur hiérarchique pour la construction de systèmes par bloc diagrammes.
  - Compilateur (ordonnancement de l'exécution des blocs)
  - Simulateur de systèmes hybrides (ODE/DAE) avec détection d'évènements (traversée de zéros)
  - Compilateur vers C de sous schémas Modelica (schémas non orientés)
  - Compilateur vers C de schémas Scicos
  - Ensemble de palettes.
  - Rajout possible de boîtes à outils (coselica, modnum).
  - Comportement des blocs décrit en C, Nsp ou Modelica.

Scicos: l'éditeur dans Nsp



Génération de code

- Utilisation
  - la génération de code pour des systèmes embarqués.
  - pour des simulations hors Nsp/Scicos
  - construire de nouveaux blocs à partir d'un diagramme.
- Limitations
  - Les blocs sont souvent décrits en C.
  - Le générateur de code voit chaque bloc comme une entité atomique
  - Les blocs ne sont pas toujours spécialisés et peuvent contenir des switch (type et dimensions).
  - Certains blocs sont en Modelica.
  - Les utilisateurs sont plus habitués à Nsp qu'à Modélica.

Nsp comme langage de script pour définir les blocs

- La sémantique d'un bloc scicos est souvent voisine de la sémantique d'une primitive Nsp.
- La surcharge d'opérateurs dans Nsp simplifie l'adaptation du code à des types divers
- Après évaluation partielle on aura du code C spécialisé pour les types et dimensions spécifique au schéma traduit.

Exemple: Le bloc summation

```
function block=P_SUMMATION(block,flag)
 if flag==1 then
   vars=block.io;
   nin= (length(vars)) - 1;
   sgns=block.params.p2
   code_insert('annotation',"Sum block begins with "+string(nin)+" inputs.")
   if nin == 1 then
     code_insert('annotation',"Using the sum function.")
     if sgns(1)==-1 then out=-sum(vars(1))
     elseif sgns(1)==+1 then out=sum(vars(1))
     else error('wrong sign: "+string(sgns(1)))
     end
   else
     if sgns(1)==-1 then out=-vars(1)
     elseif sgns(1)==+1 then out=vars(1)
     else error('wrong sign: "+string(sgns(1)))
     end
     for i=2:nin
      if sgns(i)==-1 then out=out-vars(i)
      elseif sgns(i)==+1 then out=out+vars(i)
      else error('wrong sign: "+string(sgns(i)))
      end
     end
   end
   block.io($)=out
 end
endfunction
```

Surcharge: Un nouveau type de variable de type bvar (bloc variable)

```
-nsp->x=symbolics(rand(1,2))     // a new variable of type bvar which is symbolic
x = "tmp_38",%t,Mat
-nsp->x.get_value[]              // what matters here is just size and type
ans   = r (1x2)

 |  0.9575  0.9965 |

-nsp->x.is_symbolic[]             // the variable is symbolic
ans =  b (1x1)

 | T |
-nsp->size(x)                     // the size
ans = r (1x2)

 |  1  2 |

-nsp->type(x.get_value[],'short') // the type of the attached value
ans = s (1x1)

  m
```

Surcharge: des fonctions pour les variables de type bvar

```
function out=plus_bvar_bvar(in1,in2)
  global overflow_option
  overflow_opt=overflow_option
  if isempty(overflow_opt) then overflow_opt="overflow";end
  if datatype(in1) <> datatype(in2) then error("Incompatible types"),end
  if ~is_sym(in1) && ~is_sym(in2) then
    out=(valueof(in1)+valueof(in2));
    return
  end
  if (prod(size(valueof(in1)))==1) & (prod(size(valueof(in2)))==1) then
    vin1= valueof(in1);vin2= valueof(in2);
    if "plus"=="dsl" && vin2==0 then vin2=vin2+1;end
    if "plus"=="dbs" && vin1==0 then vin1=vin1+1;end
    out=symbolics(vin1+vin2,getunique())
    rhs=expression("+",list(in1,in2),overflow_opt)
    gen_def(out,rhs)
  else
    out=Empty(valueof(in1)+valueof(in2))
    sz=size(valueof(out));sz1=size(valueof(in1));sz2=size(valueof(in2))
    for i=1:sz(1)
      for j=1:sz(2)
        out(i,j)=in1(min(i,sz1(1)),min(j,sz1(2)))+in2(min(i,sz2(1)),min(j,sz2(2)))
      end
    end
  end
endfunction
```

Surcharge: l'évaluation de fonctions produit du code par effet de bord

```
-nsp->x=symbolics(6);
-nsp->x+1.2;
-nsp->code
code -> code hobj
  code = l (1)
   (
    (1) = l (3)
     (
      (1) = "set" s (1x1)
      (2) = "tmp_2",%t,Mat
      (3) = h (3/11)
        exp = l (3)
         (
          (1) = "+" s (1x1)
          (2) = l (2)
           (
            (1) = "tmp_1",%t,Mat              (2) = 1.2 r (1x1)
           )
          (3) = s (1x1)
            overflow
         )
        type = "op" s (1x1)       tlist =  T  b (1x1)
    )
   )
```

Surcharge: génération de code

```
-nsp->declarations
declarations -> declarations hobj
  declarations = l (1)
   (
    (1) = l (3)
     (
      (1) = "ephemere" s (1x1)
      (2) = "tmp_2",%t,Mat
     )
   )
```

Surcharge: de nouvelles fonctions pour les variables standard et bvar

```
-nsp->If_exp([%t,%f],1:2,3:4)
ans = r (1x2)

 |  1  4 |
-nsp->Select_exp([1,3],[1,1],[2,3],[4,5])
ans = r (1x2)

 |  1  5 |
```

Surcharge: génération de code

A partir de pseudo code nous pouvons obtenir

- du pseudo code optimisé
- du code Nsp spécialise
- du code C.
- (Projet FUI) P pseudo code décrit pat un fichier xml.
  - Le P pseudo code peut être traduit en C ou Ada.

Exemple: traduction d'un fonction Nsp

```
function y=f(z);
 y=convert(z,"b");
endfunction;
```

Nous voulons obtenir une version spécialisée pour des matrice 2x2 de doubles.

```
function y=code_test_data()
  y=list(rand(2,2))
endfunction;
```

Exemple: Suite

La fonction traduite après évaluation partielle (non optimisé)

```
function [tmp_17]=testcode_internal(z)
   tmp_1=m2b(ones(2,2));
   tmp_5=m2b(ones(2,2));
   tmp_9=m2b(ones(2,2));
   tmp_13=m2b(ones(2,2));
   tmp_2=(z(1));
   tmp_3=m2b(tmp_2);
   tmp_5=tmp_1;
   tmp_5(1)=tmp_3;
   tmp_6=(z(2));
   tmp_7=m2b(tmp_6);
   tmp_9=tmp_5;
   tmp_9(2)=tmp_7;
   tmp_10=(z(3));
   tmp_11=m2b(tmp_10);
   tmp_13=tmp_9;
   tmp_13(3)=tmp_11;
   tmp_14=(z(4));
   tmp_15=m2b(tmp_14);
   tmp_17=tmp_13;
   tmp_17(4)=tmp_15;
endfunction
```

Exemple: Suite

La version C.

```
static void f(double *z,gboolean *tmp_17)
{
 gboolean tmp_1[4], tmp_3, tmp_5[4], tmp_7, tmp_9[4], tmp_11, tmp_13[4], tmp_15;
 double tmp_2, tmp_6, tmp_10, tmp_14;
    tmp_2=(z[0]);
    tmp_3=( tmp_2 != 0.0);
    memcpy(tmp_5,tmp_1,4*sizeof(gboolean));
    tmp_5[0]=tmp_3;
    tmp_6=(z[1]);
    tmp_7=( tmp_6 != 0.0);
    memcpy(tmp_9,tmp_5,4*sizeof(gboolean));
    tmp_9[1]=tmp_7;
    tmp_10=(z[2]);
    tmp_11=( tmp_10 != 0.0);
    memcpy(tmp_13,tmp_9,4*sizeof(gboolean));
    tmp_13[2]=tmp_11;
    tmp_14=(z[3]);
    tmp_15=( tmp_14 != 0.0);
    memcpy(tmp_17,tmp_13,4*sizeof(gboolean));
    tmp_17[3]=tmp_15;
};
```

Exemple: Suite

```
static int int_f(Stack stack, int rhs, int opt, int lhs)
{
  NspMatrix *z;
  NspBMatrix *tmp_17;
  CheckStdRhs(1,1);
  CheckLhs(0,1);
  if ((z = GetMat (stack, 1)) == NULLMAT) return RET_BUG;
  if ((tmp_17 = nsp_bmatrix_create(NVOID,2,2))== NULL ) return RET_BUG;
  (void) f(z->R,tmp_17->B);
  if ( lhs >= 1 ) {;
 MoveObj(stack,1, NSP_OBJECT(tmp_17));}
  else { nsp_bmatrix_destroy(tmp_17);;}
  return Max(lhs,0);
}
static OpTab bdl_func[]={
  {"bdlf",int_f},
  {(char *) 0, NULL}
};
int bdl_Interf(int i, Stack stack, int rhs, int opt, int lhs)
{
return (*(bdl_func[i].fonc))(stack,rhs,opt,lhs);
}
void bdl_Interf_Info(int i, char **fname, function (**f))
{
  *fname = bdl_func[i].name;
  *f = bdl_func[i].fonc;
}
```

La boîte à outils codegen

- La version Nsp
  - 8kloc
  - 20 blocs primitifs décrits en Nsp.

BDL: Un langage pour décrire la sémantique de blocs

- Motivation initiale: pouvoir décrire la sémantique d'une librairie de blocs (voire d'un schéma) Scicos dans un langage comme Nsp ;
  - La sémantique d'un bloc est souvent voisine d'une fonction du langage.
  - L'implémentation en C est souvent combinatoire pour tenir compte de la multiplicité des types
  - Intérêt de générer le code C spécialisé pour l'instance d'un bloc.
- Que cette sémantique soit précisément définie (typage fort, scope static, compilation possible) ;
- Qu'un bloc utilisateur puisse contenir une fonction Nsp compilable (EML) et que BDL couvre cette utilisation ;
- BDL est un langage utilisateur. Projet en développement.