3.4. Gérer les états continus et les ports

Nous allons maintenant examiner les fonctions d'interfaçage et de simulation d'un bloc qui calcule l'intégrale de son vecteur d'entrée et dont il est possible optionnellement d'autoriser la réinitialisation sur événement. Il s'agit d'une version simplifiée du bloc `INTEGRAL_m` de la palette SYSTÈMES À TEMPS CONTINU. Ce bloc a la particularité de comprendre un vecteur d'état continu et un nombre de ports d'entrées variable selon qu'il peut ou non être réinitialisé.

La fonction d'interfaçage

Exemple 3.15 : *La section "define" de la fonction d'interfaçage* `IntegralInt`

```
x0=0;
reinit=0;
model = scicos_model();
model.state = x0;
model.sim = list("IntegralSim",4);
model.intyp = 1;
model.outtyp = 1;
model.in = 1; model.in2 = 1; //une entrée scalaire
model.out = 1; model.out2 = 1;//une sortie scalaire
model.dep_ut = [%f %t];
exprs = [sci2exp(x0);sci2exp(reinit)];
sciblk = standard_define([2 2],model,exprs);
style_properties = ["blockWithLabel";
                    "verticalLabelPosition=middle";
                    "verticalAlign=middle";
                    "displayedLabel=$\int$"];
sciblk.graphics.style = strcat(style_properties,";")
```

Les paramètres initiaux de ce bloc correspondent à un simple intégrateur d'une entrée scalaire avec une condition initiale nulle. Cette condition initiale est spécifiée dans le champ `state` de la structure *`model`*. Cette spécification fixe la taille de l'état associé à cette instance du bloc. Cette taille doit rester constante durant une simulation.

La définition des états discrets pour les blocs qui en comportent peut se faire de façon similaire par l'intermédiaire des champs `dstate` ou `odstate` de la structure *`model`*. Le champ `dstate` permet de définir un état constitué d'un vecteur de flottants tandis que le champ `odstate` est une liste pouvant contenir des matrices de flottants ou des matrices d'entiers.

Exemple 3.16 : *La section "set" de la fonction d'interfaçage* `IntegralInt`

```
exprs = sciblk.graphics.exprs;
[ok,x0,reinit,exprs]=scicos_getvalue(...
                     _("Integral block parameters"),...
                     [_("Initial Condition");
                    _("With re-intialization (1:yes, 0:no)")],...
                     list("vec",-1,"vec",1),exprs)
if ~ok then return;end
n = size(x0,'*');
out = [n, 1];
outype = 1;
evtout = []; //aucun port de sortie d'activation
if reinit~= 0 then
  //deux entrées régulières de dimension n x 1
  intype = [1, 1];
  in = [n, 1; n, 1];
  //un port d'entrée d'activation
  evtin = 1;
  reinit = 1;
else
  //une entrée régulière de dimension n x 1
  intype = 1;
  in = [n, 1];
  //aucun port d'entrée d'activation
  evtin = [];
end
[model,graphics] = set_io(sciblk.model, sciblk.graphics,...
                     list(in, intype), list(out, outype),...
                     evtin, evtout)
model.ipar = reinit;
model.state = x0(:);
sciblk.model = model;
graphics.exprs = exprs;
sciblk.graphics = graphics;
```

Le point important à noter ici est l'usage de la fonction `set_io`. Cette fonction met à jour les valeurs de `model.intype`, `model.in`, `model.in2`, `model.outtype`, `model.out`, `model.out2` que l'on a déjà vus dans les exemples précédents mais aussi `model.clkin` et `model.clkout` qui permettent de déclarer l'existence de ports d'entrées et de sorties d'activation.

Cette fonction assure aussi la gestion des liens reliés au bloc dans le cas où le nombre de ports du bloc est modifié par l'utilisateur.

Les caractérisations des ports d'entrées et de sorties sont fournies à la fonction sous forme de listes à deux éléments, le premier est un tableau à deux colonnes dont chaque ligne donne les dimensionnalités du signal attendu sur le port correspondant et le second le vecteur des types des signaux attendus sur chacun des ports selon les correspondances données dans le Tableau 3.2.

Note *› À la différence des signaux réguliers, les liens d'activation ne sont porteurs que de signaux scalaires, les valeurs de* `model.clkin` `model.clkout` *doivent donc être des vecteurs ne contenant que des valeurs 1.*

La fonction de simulation

Exemple 3.17 : *Le fichier* `IntegralSim.c` *de la fonction de simulation*

```
#include <math.h>
#include "scicos_block4.h"
void  IntegralSim(scicos_block *Cblk, scicos_flag flag)
{
  int i = 0;
  int n = GetInPortRows(Cblk,1);

  if (flag == DerivativeState){ /* Evaluation de la fonction C */
    double *in = GetRealInPortPtrs(Cblk,1);
    double *xd = GetDerState(Cblk);
    for(i=0; i<n; ++i) xd[i] = in[i];
  }
  else if (flag == OutputUpdate || ...
           flag == ReInitialization){ /* Evaluation de la fonction F */
    double *x = GetState(Cblk);
    double *out = GetRealOutPortPtrs(Cblk,1);
    for(i=0; i<n; ++i) out[i] = x[i];
  }
  else if (flag == StateUpdate){/* Evaluation de la fonction Dx*/
    double *x = GetState(Cblk);
    double *in2 = GetRealInPortPtrs(Cblk,2);
    for(i=0; i<n; ++i) x[i] = in2[i];
  }
}
```

Le bloc ayant un état interne, une entrée et une sortie d'activation doit implémenter :

- l'évaluation de la fonction F calculant la sortie du bloc. L'évaluation de cette fonction est réalisée lorsque la fonction de simulation est appelée avec la variable `flag` égale à `OutputUpdate` ou à `ReInitialization` . Cette dernière valeur de `flag` est utilisée par Xcos au démarrage de la simulation ou lors du traitement d'une discontinuité pour déterminer par une procédure de point fixe la valeur de l'ensemble des signaux du système.

  L'instruction `double *x = GetState(Cblk);` permet d'obtenir le pointeur sur le vecteur d'état du bloc. Si nécessaire, nous aurions pu obtenir la taille de ce vecteur d'état grâce à `GetNstate(Cblk)` ;

- l'évaluation de la fonction C calculant la dérivée de l'état. L'évaluation de cette fonction est réalisée lorsque la fonction de simulation est appelée quand l'argument `flag` est égal à `DerivativeState` .

  L'instruction `double *xd = GetDerState(Cblk);` permet d'obtenir le pointeur sur le vecteur devant contenir la valeur de la dérivée de l'état du bloc ;

- l'évaluation de la fonction D_x calculant le changement de l'état continu lors de l'activation. L'évaluation de cette fonction est réalisée lorsque le bloc a été activé par son port d'activation. La fonction de simulation est alors appelée lorsque l'argument `flag` est égal à `StateUpdate>` .

  Dans le cas d'un bloc possédant plusieurs ports d'entrées d'activation, la fonction D_x peut dépendre de la manière dont le bloc a été activé. L'instruction `int nevprt = GetNevIn(Cblk);` permet de retourner l'entier `nevprt` qui caractérise l'activation du bloc. La valeur de `nevprt` est donnée par l'expression $\sum_{j=1}^{n} i_j 2^{j-1}$ où n est le nombre de ports d'entrées d'activation, i_j est égal à 1 si une activation a été reçue sur le port i et 0 sinon. Ainsi pour un bloc ayant trois ports d'entrées d'activation, si le bloc est activé par le troisième port on aura `nevprt==4` et si les ports 1 et 2 sont activés simultanément par deux signaux synchrones, on aura `nevprt==3`.

États discrets

La gestion des états discrets est très similaire à celle des états continus, sans toutefois la contrainte d'être stockés dans des tableaux double précision. Si le bloc ne comprend qu'un seul vecteur d'état de type flottant, la valeur initiale de l'état peut être définie dans la fonction de d'interfaçage dans la structure `model.dstate`. Par contre, il est aussi possible de gérer des blocs comprenant plusieurs groupes d'états, chaque groupe pouvant être un tableau de nombres flottants, un tableau d'entiers codés sur 8, 16 ou 32 bits. Dans ce cas, les états initiaux sont assignés sous forme d'une liste Scilab dans la structure `model.odstate`. Chaque élément de la liste correspondant à un groupe d'états.

La fonction de simulation peut alors accéder à ces états discrets grâce aux fonctions récapitulées au Tableau 3.4.

Tableau 3.4 : Récapitulatif des fonctions d'accès aux états discrets

Fonction	Type	Rôle
États définis dans la structure de données `model.odstate`		
`GetNoz(Cblk)`	`int`	
`GetOzType(Cblk, i)`	`int`	Retourne le type du tableau correspondant au $i^{ième}$ groupe d'états.
`GetOzSize(Cblk, i, k)`	`int`	pour `k` variant de 1 à 2, retourne la $k^{ième}$ dimension du tableau correspondant au $i^{ième}$ groupe d'états.
`GetOzPtrs(Cblk, i)`	`void *`	Retourne le pointeur du tableau correspondant au $i^{ième}$ groupe d'états.
`GetRealOzPtrs (Cblk, i)`	`double *`	Retourne le pointeur sur la partie réelle du tableau de flottants correspondant au $i^{ième}$ groupe d'état.
`GetImagOzPtrs (Cblk, i)`	`double *`	Retourne le pointeur sur la partie imaginaire du tableau de flottants correspondant au $i^{ième}$ paramètre groupe d'état s'il est complexe et sinon retourne `NULL`.
`Getint8OzPtrs (Cblk, i)`	`char *`	Retourne le pointeur sur le tableau d'entiers (stockés sur 8 bits) correspondant au $i^{ième}$ groupe d'états.
`Getint16OzPtrs (Cblk, i)`	`short *`	Retourne le pointeur sur le tableau d'entiers (stockés sur 16 bits) correspondant au $i^{ième}$ groupe d'états.
`Getint32OzPtrs (Cblk, i)`	`int *`	Retourne le pointeur sur le tableau d'entiers correspondant au $i^{ième}$ groupe d'états.
`Getuint8OzPtrs (Cblk, i)`	`unsigned char *`	Retourne le pointeur sur le tableau d'entiers non signés (stockés sur 8 bits) correspondant au $i^{ième}$ groupe d'états.
`Getuint16OzPtrs (Cblk, i)`	`unsigned short *`	Retourne le pointeur sur le tableau d'entiers de type `unsigned short` correspondant au $i^{ième}$ groupe d'états.
`Getuint32OzPtrs (Cblk, i)`	`unsigned int *`	Retourne le pointeur sur le tableau d'entiers de type `unsigned long` correspondant au $i^{ième}$ groupe d'états.
États définis dans le vecteur des états flottants `model.dstate`		
`GetNdstate(Cblk)`	`int`	Retourne le nombre de valeurs flottantes stockées dans le vecteur `model.dstate`.
`GetDstate(Cblk)`	`double *`	Retourne le pointeur sur le vecteur des états flottants.