SynDEx v6 TUTORIAL and USER MANUAL

Quentin Quadrat

January 5, 2006

Contents

Chapter 1

Example 7: edition of the source code associated with an operation

In the previous Example 6, we have learnt that a m4 file, called with the name of the application plus the m4x extension, must be manually written. It contains all the source code associated with all operations present in a SynDEx application. For example, in the **example6** application, the **conv** function increments of one the value of the input and stores the result in the output. Thus in the **example6.m4x** file, we write the following code:

```
define(‘conv’,‘ifelse(
    MGC,‘INIT’,‘‘’’,
    MGC,‘LOOP’,‘‘$2[0] = $1[0] + 1’’,
    MGC,‘END’,‘‘’’),)
```

where **$1** and **$2** correspond respectively to the input port name **in** and the output port name **out** of the **conv** function.

Handwriting this kind of code is not very easy, for several reasons:

- The port number may change by inserting or removing a port or parameter, following the SynDEx's rule of port numeration. For example, after inserting a parameter **P** in the function **conv**, **$1** will not refer to the input port name `in` but it will refer to the new added parameter **P**. All numbers are now brought, thus we must modify the code and replace the arguments **$1** by **$2** and **$2** by **$3**;
- This task is quite repetitive when an application contains many operations,
- It is easy to make a mistake in the m4 syntax. Great knowledges in m4 syntax (two different kind of quotes, `ifelse` ...) and SynDEx m4 macros (`MGC`) are required.

It should be more convenient to write `@OUT(out)[0] = @IN(in)[0] + @PARAM(P)` and let SynDEx interpret it and generate the associated m4x file that to write the specification with the m4 syntax. SynDEx (version $\geq$ 6.8.4) is able to do that thanks the *Code Editor* which is a tool integrated in the graphical user interface (GUI). In the following example, we will show how to use this tool.

1.1 Adding parameters to an already defined operation

In this example, we show how to modify some functions by adding parameters in order to expand parameters into m4 arguments.

To open the example6.sdx application

1. In the SynDEx principal window: click on the menu bar **File**.
2. In the menubar **File**: select the menu button **Open**.
3. In the **Open** window: find the **example6.sdx** file and open it.

To add parameters to the conv function

1. In the main algorithm: right click on the **conv** blue box.
2. In the popup menu: select the menu button **Modify**.
3. In the **Modify** window: in the entry text, write `conv_ref <2;3>` instead of `conv`. Close the window [1].
4. In the **conv_ref** blue box: double left click:
5. In the **Algorithm Function conv**: open the menu bar **Edit**.
6. In **Edit** menu: select the menu button **Parameters Names**.
7. In the **Parameters Names** window: in the text area, write `P T`. Close the window.

To verify that the parameters have been stored in the function

We have three solutions :

1. In the main algorithm: put the mouse on the **conv_ref** box.
2. In the SynDEx principal window: read the printed informations.

Or:

1. In the main algorithm: double left click on the **conv_ref** blue box.
2. In the **Algorithm Function conv**: open the menu bar **Edit**.
3. In the menu bar **Edit**: select the menu button **Port Order**.

Or by activating the bubbles which are yellow tool tips that appears when the mouse cursor points on a SynDEx operation box.

1. In the main algorithm: double left click on the **conv_ref** blue box.
2. In the **Algorithm Function conv**: open the menu bar **Edit**.
3. In the menu bar **Options**: select the menu button.
4. Target a box with your mouse cursor. **Show Info Bubbles**.

1.2 Edition of the code associated with an operation

1.2.1 In the case of a generic processor

To open the Code Editor

We need to launch the Code Editor of the selected operation. Let us consider the case of the **conv** function. We have to do the following operations:

1. In the main algorithm: double left click on the **conv** blue box.
2. In the menu bar **Edit**: select the menu button **Edition of the associated source code**. This opens the Code Editor. It looks like a window with three push-buttons and an editable text area. Each push-button corresponds to one specific phase of three (init, loop and end phase). When one of the three buttons is pressed, the corresponding phase is activated and the text area shows the associated source code.

[1] Note, that you will have only to ignore the error messages printed in the SynDEx principal window if you have inverted (3) and (7).

3. In the Code Editor window: press on the **Init phase** button and write the following C language code in the text area (which is empty). This code is understood as a generic code:

```
printf("Init phase of function conv for default type of processor.\n");
```

4. Do the same thing for the **Loop phase** [2].

```
@OUT(out)[0]=@IN(in)[0]*@PARAM(T)+@PARAM(P);
printf("Loop phase of function conv for default processor = %i.\n", @OUT(out)[0]);
```

5. And for the **End phase**.

```
printf("End phase of function conv for default processor.\n");
```

6. In the Code Editor window: open the menu bar **Edit**.
7. In he menu bar **Edit**: select the menu button **Apply to All phases**. It saves all buffers of all edited phases.
8. In the Code Editor window: close the window with the OK button (which also saves all buffers).

Note the following points:

- You can not launch the Code Editor from a main algorithm or a read-only operation (definition coming from libraries).
- You can write a code associated with a super-operation meanwhile it will not be present in the **applicationName_sdc.m4x** file (the background color of the text area of the Code Editor is grey).
- It is important to recall that the code is common to all the references of the same operator. Only the values of parameters are specific of each instance.

To show that code is common to all references

We create a new reference of the **conv** function.

1. In the main algorithm: open the menu bar **Edit**.
2. In the menu bar **Edit**: select the menu button **Create references**.
3. In the **Create references** window: double click on **conv**.
4. In the window that appears: in the entry text, write `conv_ref_bis<8;9>`.
5. Open the Code Editor of this new box: the code is the same. To show that values of parameters are specific to the reference function (and not to the function definition), we must generate the C code thanks the compilation of the GNUMakefile show in the previous example 6.

To link the new function.

1. In the main algorithm: first remove the link between, the **conv_ref** and **mul** boxes. Second, link the **conv_ref** output to the **conv_ref_bis** input. Third, link the **conv_ref_bis** output to the **mul** input.
2. In the main algorithm: select the menu bar **window**.
3. In the menu bar **window**: select the menu button **Auto position**.
4. In the **Space between vertices** window: in the two entry texts, write 120. Close this window. The main algorithm window should look like this (confer fig. **??**).

[2] Macros of the Code Editor as @IN, @OUT, @PARAM, ... are explained in the User Manual.

1.2.2 In the case of an architecture with heterogeneous processors

Sometimes it is interesting for an operation to have different source codes depending on the type of processor. In deed we can want to have a heterogeneous architecture where each processor has his own language but a difficulty appears: during the adequation, an operation can be executed on a first type of processor but after changing his duration, it can be executed on another processor. Because of the fact the processor no longer recognizes the language.

To resolve this problem, the user is obliged to associated the code in specific syntax. To facilitate this work, the Code Editor associates a window for each triplet (phase, processor, operation).

To include a new processor type

1. In the SynDEx principal window: open the menu bar **File**.
2. In the **File** menu: select the **Include Libraries**.
3. In the **Include Libraries** menu: check the button **C40**.

To create a new architecture

1. In the SynDEx principal window: open the menu bar **Architecture**.
2. In the **Architecture** menu: select the **New local Architecture**.
3. In the **New local Architecture** window: in the entry text, write `archi2`. Click OK.
4. In the **archi2** window: create an operator reference **C40** named `root`. And set it as a main operator.

To replace the conv_ref_bis box by another reference

1. Create a new definition of a function, as learnt in the first tutorial. The definition function is named **bar**, it has one input called **in** and one output called **out**.
2. In the **Edit Main Algorithm** window: delete the **conv_ref_bis** box.
3. In the **Edit Main Algorithm** window: create a reference **bar_ref** box to the **bar** function definition.
4. Link the boxes as the fig. **??**.

To insert code for C40 processor type for the bar function

1. In the **Edit Main Algorithm** window: in the **bar_ref** box, right click and open the `Code Editor`.
2. In the **bar Code Editor** window: select the menu bar **Type processor**.
3. In the menu bar **Type processor**: select the **C40** menu button.
4. In the **bar Code Editor** window: click on the **Init phase** button and write in the text area:

```
/* Hi, I am bar function, in init phase for C40 processor */
```

5. In the **bar Code Editor** window: click on the **Loop phase** button and write in the text area:

```
@OUT(out)[0] = @IN(in)[0];
```

6. In the **bar Code Editor** window: click on the **Phase phase** button and write in the text area:

```
/* Bye, I am bar function, in end phase for C40 processor */
```

7. In the **bar Code Editor** window: select the menu bar **Type processor**.

To insert code for U processor type for the bar function

1. In the menu bar **Type processor**: select the **U** menu button.
2. In the **bar Code Editor** window: click on the **Init phase** button and write in the text area:

```
/* Hi, I am bar function, in init phase for U processor */
```

3. In the **bar Code Editor** window: click on the **Init phase** button and write in the text area:

```
@OUT(out)[0] = @IN(in)[0] * 42;
```

4. In the **bar Code Editor** window: click on the **Init phase** button and write in the text area

```
/* Bye, I am bar function, in loop phase for U processor */
```

To modify the durations

1. In the **Edit Main Algorithm** window: open the **Durations** of all reference boxes that do not came from libraries (user's function). Write `C40 = 1` at the end of the text area.

1.2.3 Learn the macros of the Code Editor

SynDEx grant les noms des ports et des paramtres, aussi utiliss dans le code utilisateur dont il ne connat pas la syntaxe, un jeu de macros spcifique l'diteur de Code a t cr pour faciliter l'criture du code spcifique de l'utilisateur que SynDEx ne peut pas automatiser.

La premire macro est `@NAME(nom_port_ou_param)` prend en paramtre un nom de port ou de paramtre associ l'opration et le transforme en macro m4 (le caractre `$` suivi d'un entier positif ou nul. La numrotation, comme nous l'avons dit en introduction dans le chapitre prcdent, dpend du type de port (entre, sortie, entre–sortie) ainsi que l'ordre où ils sont dclars. Toute erreur avec des noms inconnus est collecte puis affiche lors de la gnration de code.

Comme SynDEx accepte que des ports de types diffrents aient des noms identiques alors l'diteur de Code peut se tromper. Des macros comme `@IN()` ou `@OUT()` ou `@INOUT()` ou `@PARAM()` permettent de spcifier le type. Toute erreur sur un nom connu mais de mauvais type produit une erreur lors de la gnration de code.

Par dfaut, le code associ est entour par deux niveaux de guillemets (confre l'exemple **??** de l'introduction). Un utilisateur peut vouloir retirer ou ajouter un niveau de guillemets. Cela est possible grce aux macros `@TEXT(texte)` et `@QUOTE(texte)`, où `texte` est du texte de taille arbitrairement grande et qui peut inclure d'autres macros.

1.3 Generation of m4x files

Before performing the Code Generation we have to perform the Adequation.

1. In the SynDEx principal window: open the menu bar **Adequation**.
2. In the menu bar **Adequation**: select the menu button **Flatten**.
3. In the SynDEx principal window: open the menu bar **Code**.
4. In the menu bar **Code**: check the menu box **Generate m4x Files** option.
5. In the menu bar **Code**: select the menu button **Generate Executive**.
6. In the menu bar **Code**: select the menu button **View Code**.

Two cases are possible:

- The check box `Generate m4x Files` has not been checked. This generate for each operator of the main architecture the code in a file (called the name of processor with the m4 extension), and an architecture description (file **exemple6.m4**) as explained previously.
- The check box `Generate m4x Files` has been checked. Then, two new files (**exemple6.m4x** and **exemple6_sdc.m4x**) are generated in the same directory as the application.

What are these two files ? They constitute the *Applicative kernel*:

- The **MonApplication_sdc.m4x** file is the most important of the two files because it contains all the m4 macro codes associated to all operations used in a SynDEx application. To each generation, the new file overwrites the old one.
- The **MonApplication.m4x** file is an user editable file in the aim to allow the user to complete the executive kernel if needed. During the generation, if this file does not physically exist then SynDEx creates a generic file (including the **MonApplication_sdc.m4x** file plus some other features) else SynDEx does not overwrite it.

The body of **exemple6_sdc.m4x** file has the following code:

```
divert(-1)
#
# (c)INRIA 2001
# SynDEx v6 executive macros specific to application example6
#
#
# WARNING: DO NOT EDIT THIS FILE - it is generated by SynDEx
#
divert(0)

define(‘conv’,‘ifelse(
    processorType_,processorType_,‘ifelse(
        MGC,‘INIT’,‘‘printf("Init phase of function $0\n");’’,
        MGC,‘LOOP’,‘‘$4[0]=$3[0]*$2+$1;
                    printf("Loop phase of function $0\n return = %i\n", $4[0]);’’,
        MGC,‘END’,‘‘printf("End phase of function $0\n");’’)’)’)

define(‘bar’,‘ifelse(
    processorType_,‘U’,‘ifelse(
        MGC,‘INIT’,‘‘/* Hi, I am $0 function, in init phase for U processor */’’,
        MGC,‘LOOP’,‘‘$2[0] = $1[0] * 42;’’,
        MGC,‘END’,‘‘/* Bye, I am $0 function, in loop phase for U processor */’’)’,
    processorType_,‘C40’,‘ifelse(
        MGC,‘INIT’,‘‘/* Hi, I am $0 function, in init phase for C40 processor */’’,
        MGC,‘LOOP’,‘‘$2[0] = $1[0];’’,
        MGC,‘END’,‘‘/* Bye, I am $0 function, in loop phase for C40 processor */’’)’)’)
```

The body of **exemple6.m4x** file has the following code:

```
divert(-1)
#
# (c)INRIA 2001
# SynDEx v6 executive macros specific to application example6
#
#
# Edit what do you want in this file. It is not overwritten by SynDEx
#
divert(0)
```

```
define(‘dnldnl’,‘//’)
define(‘NOTRACEDEF’)
define(‘NBITERATIONS’,5)

include(‘example6_sdc.m4x’)

#include <stdio.h> /* for printf */
```

Explications about the work of the m4 macro `ifelse` are given in the SynDEx User Manual and the GNU m4 Manual.

To see the difference between executable codes

Set the **archi** as main architecture and run the GNUmakefile compilation as explained in Example 6. You obtain a root.c file. Open it, you see that there is no trace of U codes but only C40 codes.

Set the **archi2** as main architecture and do as previous, you see that there is no trace of C40 codes but only U codes.

Chapter 2

Example 7: A complete Application: Algorithm Architecture Adequation, Code Generation, Code Compilation and Execution

2.1 The aim of the example

In the seven previous examples we have learnt how to use SynDEx's GUI to create architectures, algorithms, launch adequation, obtain executive files... Now, we have sufficient knowledge to perform a simple automatic control application that will be executed on a multiprocessor architecture.

First the application is described and the system is defined in Scicos (the block diagram editor of the Scilab software[1]). Second the corresponding SynDEx application is created (using the Example 1 to 3 of the tutorial). This need the generation of some C code following the method discussed in Example 7. Finally, we compile the application to obtain executable for several processors as it has been shown in Examples 5 to 7. SynDEx generates the code necessary to the communication between the processors.

2.2 The model

We consider a system of two cars. The second car $\mathcal{C}_2$ follows the car $\mathcal{C}_1$ trying to maintain the distance l while the acceleration and the deceleration of $\mathcal{C}_1$. We call: $x_1(t)$ the position of the first car; $x_2(t)$ the position of the second car plus l; $\dot{x_1}(t)$ and $\dot{x_2}(t)$ the speeds of two cars. We denote k_1 and k_2 the inverse of the car masses. We call $r(t)$ the reference speed chosen by the first driver. We suppose that we are able to observe the speed of the first car and the distance between the cars.

We have the following fourth order (four degrees of liberty) system:

$$\begin{aligned} \ddot{x_1} &= k_1 u_1 \\ \ddot{x_2} &= k_2 u_2 \\ y_1 &= \dot{x_1} \\ y_2 &= x_1 - x_2 \end{aligned} \tag{2.1}$$

We will decompose the system into mono-input mono-output system $S_1(u_1, y_1)$ and $S_2(u2, y2)$. Denoting by uppercase letter the Laplace transform of the variables, we have $Y_1 = k_1 U_1/s$ and $Y_2 = (k_1 U_1 - k_2 U_2)/s^2$ where U_1 is seen as a perturbation that we want reject in the second system.

A first proportional feedback $U_1 = \rho_1(R - Y_1)$ will insure the first car to follow the reference speed. The second controller will be proportional derivative $U_2 = \rho_2 Y_2 + \rho_3 s Y_2$ (in fact we will suppose in the

[1] http://www.scilab.org

following diagram that the derivative of y_2 is also observed). The coefficient ρ_1 is obtained by placing the pole of the first loop:

$$Y_1 = \rho_1 k_1 R/(s + \rho_1 k_1).$$

The coefficient ρ_2 and ρ_3 are obtained by placing the pole of the transfer from U_1 to Y_2 in the closed loop system which is given by:

$$Y_2 = U_1 k_1/(s^2 + k_2\rho_3 s + k_2\rho_2).$$

2.3 Building controllers

The purpose of the controller of the $\mathcal{C}_1$ car is to follow the reference in speed given the first driver. It stabilizes the $\mathcal{C}_1$ speed around its reference speed by using pole placement, for example, gains are respectively: (0, -5, 0, 0, -5). The controller of second car stabilizes the distance between the two cars. It stabilizes y_2 around 0 by pole placement. For example, gains are respectively: (4, 4,-4,-4).

The controler of $\mathcal{C}_2$ knows these informations and send them electronically to $\mathcal{C}_1$. This remark is available for $\mathcal{C}_1$.

2.3.1 Block diagrams of controllers

Our controllers are simple. They are represented in figures 2.2 and 2.4 in SynDEx and figures 2.1 and 2.3 in Scicos.

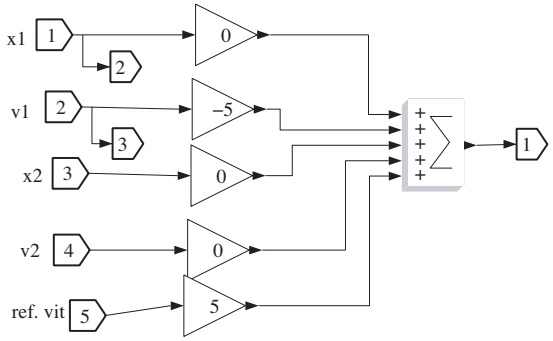

Figure 2.1: The controler of the first car.

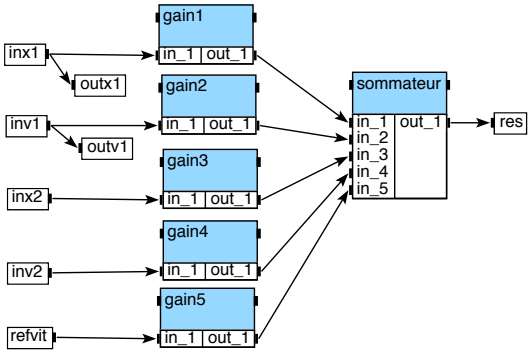

Figure 2.2: The controler of the first car..

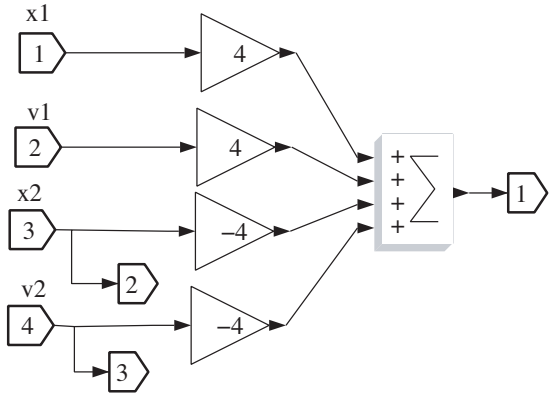

Figure 2.3: The controler of the second car.

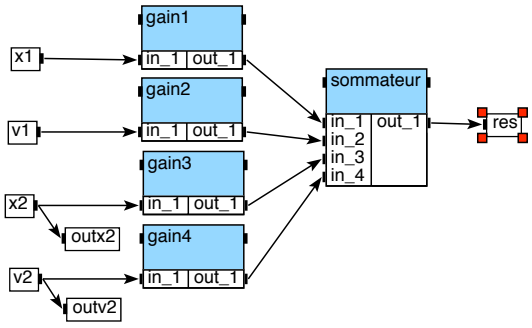

Figure 2.4: The controler of the second car.

2.3.2 Edition of the source code associated with the functions

We attach C source code to each function definition: gain and nary-sums. The code is inserted for the default processor.

Gain

A gain is a function that multiplies its input by a coefficient given as a parameter, named `GAIN`. Open the Code Editor of the gain definition and write the following code in the `loop` phase of the default processor.

```
@OUT(out_1)[0] = @IN(in_1)[0] * @PARAM(GAIN);
```

Sum

We have three different forms of sum depending of its arity: – a sum with five input ports (figure 2.2); – a sum with four input ports (figure 2.4).

Open the Code Editor of a sum with the five input ports and write the following code in the `loop` phase of the default processor.

```
@OUT(out_1)[0] = @IN(in_1)[0] + @IN(in_2)[0] + @IN(in_3)[0]
                              + @IN(in_4)[0] + @IN(in_5)[0];
```

Open the Code Editor of a sum with the four input ports and write the following code in the `loop` phase of the default processor.

```
@OUT(out_1)[0] = @IN(in_1)[0] + @IN(in_2)[0] + @IN(in_3)[0] + @IN(in_4)[0];
```

2.4 Building the complete model

In a real application, our job stops with the SynDEx's adequation of the two controllers on their associated architectures. Nevertheless, for pedagogic reasons, we will simulate the whole system (with the dynamics of the cars) in the aim to verify that our application does the same job that Scicos.

After inserting the reference speed `ref_vit` (which we define it as a square wave generator) and two kinds of output (`vitesse` ($\dot{x_1}$), the distance between the two cars), the applications looks like the figure 2.5 in SynDEx. In the following sections, we study the inner of hierarchical boxes `voiture1` and `voiture2`.

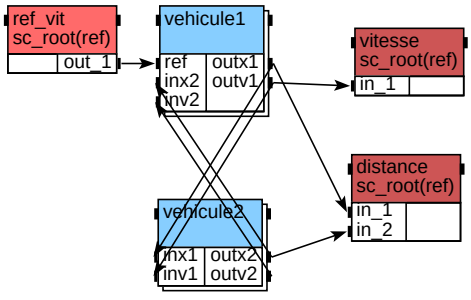

Figure 2.5: The SynDEx's main algorithm

2.4.1 The cars and their controllers

In the following diagrams (from 2.6 to 2.9), the blocks (operations denoted by **meca** are the car dynamics. Let us get the controllers of the two cars.

The $\mathcal{C}_1$ car and its controller

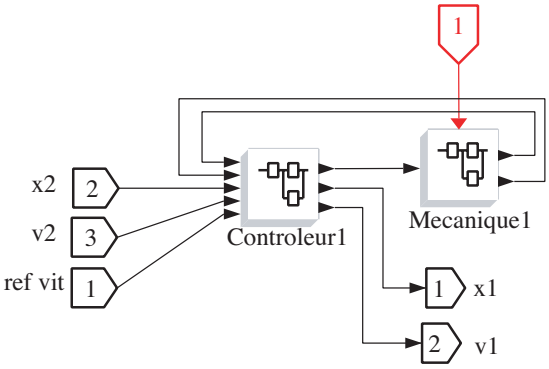

Figure 2.6: The $\mathcal{C}_1$ car dynamics and its controller in Scicos.

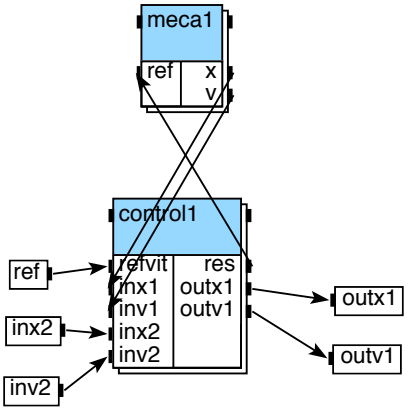

Figure 2.7: The $\mathcal{C}_1$ car dynamics and its controller in SynDEx.

The $\mathcal{C}_2$ car and its controller

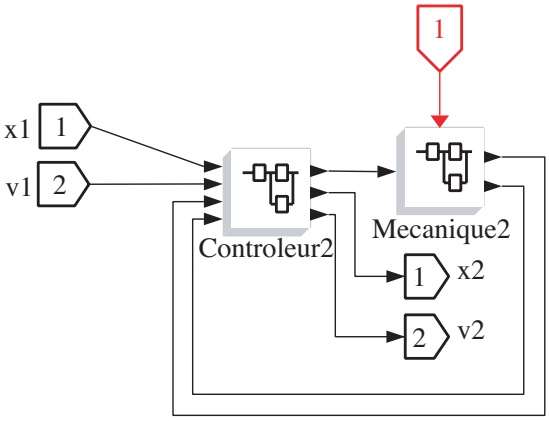

Figure 2.8: The $\mathcal{C}_2$ car dynamics and its controller in Scicos.

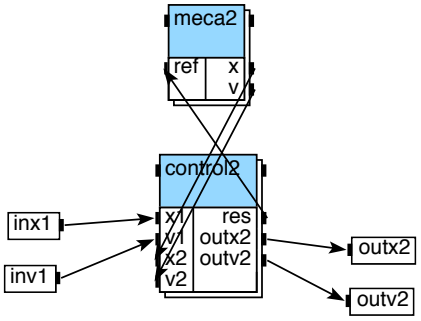

Figure 2.9: The $\mathcal{C}_2$ car dynamics and its controller in SynDEx.

2.4.2 Car dynamics

The car dynamics is given with Scicos (SynDEx) block diagrams (operations) in the figure 2.10 and figure 2.11, where the input 1 (`ref`) is the acceleration of the car. The first integral gives the speed of the car and the second its position.

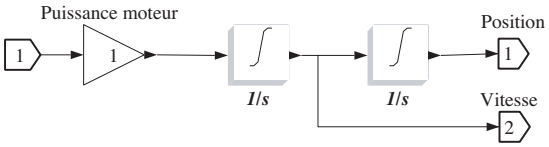

Figure 2.10: Car dynamics with Scicos block diagrams (continuous time).

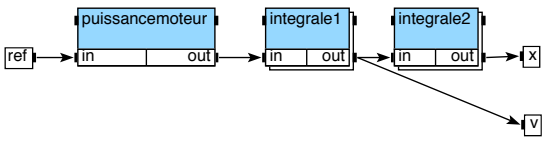

Figure 2.11: Car dynamics with SynDEx operations.

SynDEx uses only in discrete time model (not continuous time) and is not able to manage implicit algebraic loop. That is, in SynDEx, any loop contains at least a delay $1/z$. Therefore, our application which is a continuous time dynamic system described in Scicos, must be discretized in time to be used in SynDEx.

The differential equation $\dot{x} = u$ is discretetized using the simplest way: the Euler scheme. Let us

denote by h the step of the discretisation and x_0 an arbitrary initial value, the discretized system can be written as:

$$x_{n+1} - x_n = uh \tag{2.2}$$

Finally, the system (2.2) is given in Scicos (SynDEx) in the figure 2.12 (2.13). Note that, the variable h is stored in the Scicos context, and used in the input of the gain and the clock definition. In SynDEx, h is defined as parameter in the definition of a gain and the clock definition is directly used in the source code associated with operations.

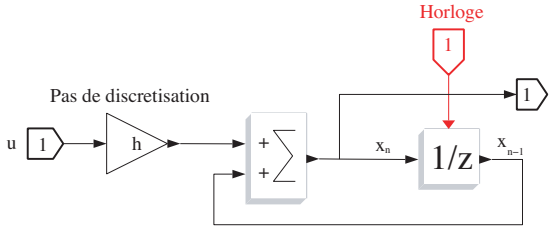

Figure 2.12: A integral discretized in Scicos (temps discret).

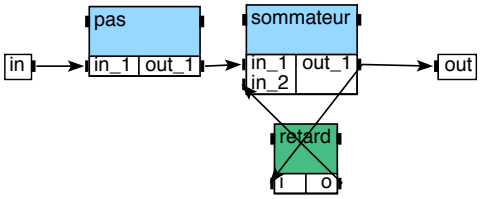

Figure 2.13: A integral discretized in SynDEx.

2.4.3 Edition of the source code associated with the functions

We attach C source code to each function definition: input and two kind of output.

Input

In our Scicos application an input is a square wave generator. As a rule, we will simulate a square wave generator by reading values in a text file (named **square_wave_generator.txt**). We will use the `fopen`, the `fclose` and the `fscanf` functions (`stdio.h` library). We will also use assertions (`assert.h` library) to ensure that the opening of a file has been successful.

For the moment, let suppose that it exists an array of `FILE*` (the structure returned by the `fopen` function) called `fd_array` and an variable called `timer` to simulate a pseudo-timer. Our sensor has a parameter called `POSI_ARRAY` to remember the position of the `FILE*` structure in the array.

Now, open the Code Editor of a sensor and write the following code in the `init` phase of the default processor.

```
timer = 0;

fd_array[@PARAM(POSI_ARRAY)] = fopen("ref_speed.txt", "r");
assert(fd_array@PARAM(POSI_ARRAY)] != NULL);
```

Because, in the Scicos application, we have defined the clock period of the square wave generator to the value 5 and defined the step of discretisation h to the value 0.001. We need, in the SynDEx application to send 5000 times the same value. To count, we use the variable `timer`. All the 5000-th times, we read a new value in the file.

Write the following code in the loop phase of the default processor.

```
timer = (timer + 1) % 5000;

if (timer == 1)
    fscanf(fd_array[@PARAM(POSI_ARRAY)], "%f\n", &data);
@OUT(out_1)[0] = data;
```

We need to free memory by closing the file. Write the following code in the end phase of the Code Editor.

```
fclose(fd_array[@PARAM(POSI_ARRAY)]);
```

Speed output

An output saves in a file the values of the system states. Thus, an output has a parameter called `POSI_ARRAY` to remember the position of the array where the stream has been saved. Open the Code Editor of an actuator and write the following code in the init phase of the default processor.

```
fd_array[@PARAM(POSI_ARRAY)] = fopen("actuator_@TEXT(@PARAM(POSI_ARRAY))", "w");
assert(fd_array[@PARAM(POSI_ARRAY)] != NULL);
```

The loop phase, allows to save the values.

```
fprintf(fd_array[@PARAM(POSI_ARRAY)], "%E\n", @IN(in_1)[0]);
```

We need to free the memory by closing the file. Write the following code in the end phase of the Code Editor.

```
fclose(fd_array[@PARAM(POSI_ARRAY)]);
```

Distance output

Contrary to the first type of output, this output has two input port but the `init` and `end` source code are identical. The `loop` phase differs. Open the Code Editor of an actuator and write the following code in the init phase of the default processor.

```
fprintf(fd_array[@PARAM(POSI_ARRAY)], "%E\n", (@IN(in_1)[0] - @IN(in_2)[0]));
```

2.4.4 Generation of the mycar_sdc.m4x

SynDEx will generate the `mycar_sdc.m4x` file (as explained in example 7):

```
define(‘sommateur5_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
         MGC,‘INIT’,‘‘’’,
         MGC,‘LOOP’,‘‘$6[0] = $5[0] + $4[0] + $3[0] + $2[0] + $1[0];’’,
         MGC,‘END’,‘‘’’)’)’)

define(‘sommateur4_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
         MGC,‘INIT’,‘‘’’,
         MGC,‘LOOP’,‘‘$5[0] = $4[0] + $3[0] + $2[0] + $1[0];’’,
         MGC,‘END’,‘‘’’)’)’)

define(‘sommateur2_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
         MGC,‘INIT’,‘‘’’,
         MGC,‘LOOP’,‘‘$3[0] = $2[0] + $1[0];’’,
         MGC,‘END’,‘‘’’)’)’)

define(‘gain_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
         MGC,‘INIT’,‘‘’’,
         MGC,‘LOOP’,‘‘$3[0] = $2[0] * $1;’’,
         MGC,‘END’,‘‘’’)’)’)

define(‘vitesse_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
         MGC,‘INIT’,‘‘fd_array[$1] = fopen("actuator_’$1‘", "w");
                     assert(fd_array[$1] != NULL);’’,
         MGC,‘LOOP’,‘‘fprintf(fd_array[$1], "%E\n", $2[0]);’’,
```

```
        MGC,‘END’,‘‘fclose(fd_array[$1]);’’)’)’)


define(‘distance_def’,‘ifelse(
    processorType_,processorType_,‘ifelse(
        MGC,‘INIT’,‘‘fd_array[$1] = fopen("actuator_’$1‘", "w");
                    assert(fd_array[$1] != NULL);’’,
        MGC,‘LOOP’,‘‘fprintf(fd_array[$1], "%E\n", ($1[0] - $2[0]));’’,
        MGC,‘END’,‘‘fclose(fd_array[$1]);’’)’)’)

 define(‘ref_vit_def’,‘ifelse(
     processorType_,processorType_,‘ifelse(
        MGC,‘INIT’,‘‘k = 0;
                    fd_array[$1] = fopen("square_wave_generator.txt", "r");
                    assert(fd_array[$1] != NULL);’’,
        MGC,‘LOOP’,‘‘k = (k + 1) % 5000;
                    if (k == 1)
                        fscanf(fd_array[$1], "%f\n", &data);
                   $2[0] = data;’’,
        MGC,‘END’,‘‘fclose(fd_array[$1]);’’)’)’)
```

2.4.5 Handwrite the mycar.m4x file

You can not use directly the SynDEx generated **mycar.m4x** generic file because both the creation of local variable and the call of libraries is missing. You must upgrade it by handwriting the following code:

```
define(‘dnldnl’,‘‘// ’’)
define(‘NOTRACEDEF’)
define(‘NBITERATIONS’,‘‘20000’’)

define(‘proc_init_’,‘
    FILE *fd_array[10];
    float data;
    int timer;’)

include(‘mycar_sdc.m4x’)

divert
    #include <stdio.h> /* for printf */
    #include <assert.h>
divert(-1)
```

Where the macro `proc_init_` allows the local variable declaration to be declared because it inserts his source code between the `main` function and the operations initialization. Note that the main loop of the program is defined generically with a loop of `NBITERATIONS` where `NBITERATIONS` is initialized with the size of the input file (`quare_wave_generator.txt`). Finally, the call of libraries is inserted after the include of the `mycar_sdc.m4x` file.

2.5 Scicos simulation

Scicos software allows to simulate models in a window which looks like the figure 2.14, where the values of three states are plotted (ordinate axle) according to the time (abscissa axle). We have:

- The square wave generator is drawn at the bottom (red),
- the speed of the first car at the top (black),

- the distance between the two cars is seen in the middle of the figure (green).

Thanks the diagram, the system is stable (plots do not grow exponentially) and so it works. We do not continue to ameliorate the controllers job.

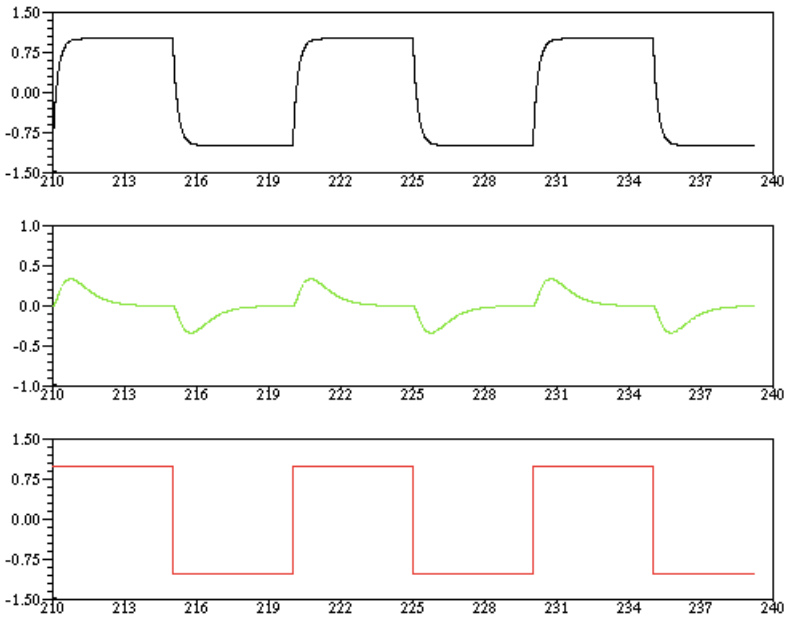

Figure 2.14: Scope window obtained with the values 0; -5; 0; 0; -5 for gains of the $\mathcal{C}_1$ controller and 4; 4; -4; -4 for the $\mathcal{C}_2$ controller.

2.6 SynDEx simulation

2.6.1 In the case of a mono-processor architecture

In this subsection, we suppose that the architecture is constituted of an only operator named `root`. Launch the adequation and generate the executive and applicative files. We must generate manually a `GNUmakefile` where its body looks like:

```
A  = mycar
M4 = gm4

export Macros_Path = ./macros/Unix_C_TCP
export Libraries_Path = ./macros/libraries
export M4PATH = $(Macros_Path):$(Libraries_Path)

CFLAGS = -DDEBUG
VPATH = $(M4PATH)

.PHONY: all clean
all : $(A).mk $(A).run
clean ::
        $(RM) $(A).mk *~ *.o *.a

$(A).mk : $(A).m4 syndex.m4m U.m4m
        $(M4) $< >$@

root.libs =
```

```
p.libs =

include $(A).mk
```

Where:

- `A` is the name of your application (here `mycar`),
- the path about the generic *.m4? macro-files are stored in the exported shell variables `Macros_Path` and `Libraries_Path` then grouped into a new exported shell variable named `M4PATH`. The separator `:` means that a m4 macro-file will be search first in `Macros_Path` and then in `Libraries_Path` if is not found;
- a mix of this makefile and the informations stored in file named `mycar.m4m` will create another makefile called `mycar.mk` during the compilation.

Then, type the command `gmake` or `make` in a shell commands interpreter. The executable is compiled and launched. We obtain the files containing the values of the system states. It is easy to compare them with these obtained in Scicos.

2.6.2 In the case of a bi-processor architecture

Create an architecture with two operator: `root` (which is the main operator) and `pc1`. They linked together by a TCP/IP media.

Obtaining the executables is a little more difficult. The folder where the executive files have been generated, must contain the following files, in addition of the **mycar.sdx** file which contains the algorithm, architecture and contraints, and the **mycar.sdc** file which contains the source code associated with operation (both created during the saving of the application):

1. the applicative files **mycar.m4x** and **mycar_sdc.m4x** which contain all source code associated with operations in m4 code,
2. the architecture file **mycar.m4** which contains the definition of the architecture in m4 code,
3. the main processor **root.m4** which contains the distributed algorithm in m4 code,
4. the processor **pc1.m4** which contains the distributed algorithm in m4 code,
5. the **root.m4x** which plays the job of a symbolic link into the **mycar.m4m** file.
6. the makefile definition **mycar.m4m** file in m4 code,
7. the generic makefile **GNUmakefile** which will compile the application to obtain executables.

Unfortunately, **GNUmakefile**, **mycar.m4m** and **root.m4x** must be created manually.

Generation of the mycar.m4m file

Create a new file **mycar.m4m**, in which you set for each operator, except the main operator, the name of a workstation corresponding to this operator. Note that `root_hostname_` is substituted by default by the current hostname (so you do not need to, and should not, redefine it). The body of the file looks like:

```
define(‘pc1_hostname_’, agaetis)dnl
```

Where the `pc1_hostname_` will be substituted by the name of a workstation (in your case `agaetis`) granting rsh requests from `root_hostname_`. Type the command `uname -n` in a shell interpreter to know the name of the current workstation.

Generation of the root.m4x file

Because Windows does not support symbolic links we must create a new file **root.m4x** including the file mycar.m4m:

```
include(mycar.m4m)
```

Generation of the GNUmakefile file

We use the same makefile for the mono-processor architecture but we upgrade it. Insert the token `$(A).m4m` in the `$(A).mk` makefile rule: So, the line:

```
$(A).mk : $(A).m4 syndex.m4m U.m4m
        $(M4) $< >$@
```

is changed by:

```
$(A).mk : $(A).m4 syndex.m4m U.m4m $(A).m4m
        $(M4) $< >$@
```

Executable generation

To launch the execution, use a shell interpreter and type the command `gmake` or `make` in the folder of the example 6. The following actions occur:

- the `root.m4`, `pc1.m4` and `mycar.m4x` is expanded on the files `root.c` and `pc1.c`,
- the `mycar.mk` file is created,
- the executables are compiled on their respective workstation,
- all executables are automatically launched,
- their piece of information are compressed with the command `tar` before be sent on the network, thanks the script shell `rshcd` (in the `macros/Unix_C_TCP` folder).

To obtain the C files type the command `gmake root.c` and `gmake pc1.c`. To delete the file created during the compilation, type the command `gmake clean`.

2.6.3 In the case of a multi-processor architecture

To simulate the application the most realistic, we replace the old architecture by a new one containing five operators and communicating with a TCP/IP communication media (see figure 2.15). We have five computers because we have two controllers, two physic systems and a device generating and measuring the input and the output.

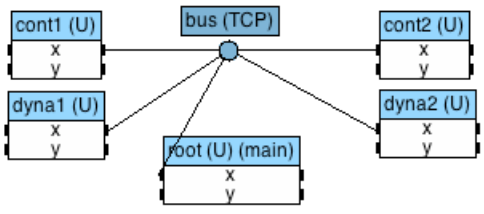

Figure 2.15: L'architecture avec cinq oprateurs.

As explain in the documentation, SynDEx uses a heuristic function to group the operations on a architecture. Meanwhile, it allows the user to force the placement of an operation on an operator thanks the software components. Let us create five, then let us:

- associate the controller of $\mathcal{C}_1$ with the operator named **cont1** and $\mathcal{C}_2$ with **cont2**,

- associate the car dynamics $\mathcal{C}_1$ with **dyna1** and $\mathcal{C}_2$ with **dyna2**,
- Put together the input and the outputs on a same operator named **root** (figure 2.5).

To launch the compilation and to execute the executable, we must upgrade the `mycar.m4m` by associating the operator names with the workstation host names.