Dynamic Systems Library
User's tutorial

Ivan Martinez
Department of Automation
Technical University of Denmark

October 3, 2000

Contents

List of Figures

1 Introduction

DSLib (Dynamic Systems Library) is a C++ library for dynamic systems implementation in RTLinux 2.X operating system. The library intends to be an interface between the block diagrams design and the realtime programing, so knowledge of operating system services is not required. Complex dynamic systems running in real-time can be implemented by only using standard C and C++. With DSLib, dynamic systems are defined as block diagrams, providing a more intuitive and well-known way of working. DSLib consists of a set of C++ classes which define components of such diagrams. By creating instances of the classes, setting their properties, and relating them, it is possible to define any kind of dynamic system. The operations performed in the blocks of the diagram are defined in C code. When the dynamic system is defined with DSLib, it can automatically generate the source code for a kernel module containing its implementation in realtime. If the dynamic system is incorrect, the user will be informed about it. If the dynamic system is right, the generated code will be error-free.

The classes that are used as components of the block diagrams are:

- CDynamicSystem
- CLoop
- CFunction
- CFunctionInput
- CFunctionOutput
- CFunctionVariable
- CLoopVariable
- CSystemVariable
- CLoggingFifo
- CSettingFifo
- CReportingFifo

- CCommandingFifo

Following sections explain how to use each of these components to define dynamic systems. The explanation will have a practical approach and an example of a dynamic system will be created in the tutorial.

2 Tutorial's example

To learn how to use DSLib, a simple dynamic system will be created in the tutorial. The system is shown in figure 1. Since it covers all topics, it is usefull to learn how to use DSLib. Apart from that, it does not represent any real system and its variables may tend to infinite.

The following components may be distinguished:

- **Functions**: These calculate output values from input values. There are four functions represented in blue color and labeled as *FunctionAA*, *FunctionAB*, *FunctionBA* and *FunctionBB*. The mathematical operations that produce the outputs from the inputs can be seen inside the blue boxes.
- **Function variables**: They are internal variables used in functions for temporary storage of values. There are two local variables represented as yellow dotted boxes inside their function boxes:
  - *FuncVarAB* will contain real values.
  - *FuncVarBB* will contain integer values.
- **Function inputs**: These are the variables that contain the input values for the functions. Inputs are represented as a green label next to an arrow entering a function box. There are four inputs, one for each function:
  - *InputAA* will contain integer values.
  - *InputAB* will contain real values.
  - *InputBA* will contain integer values.
  - *InputBB* will contain real values.

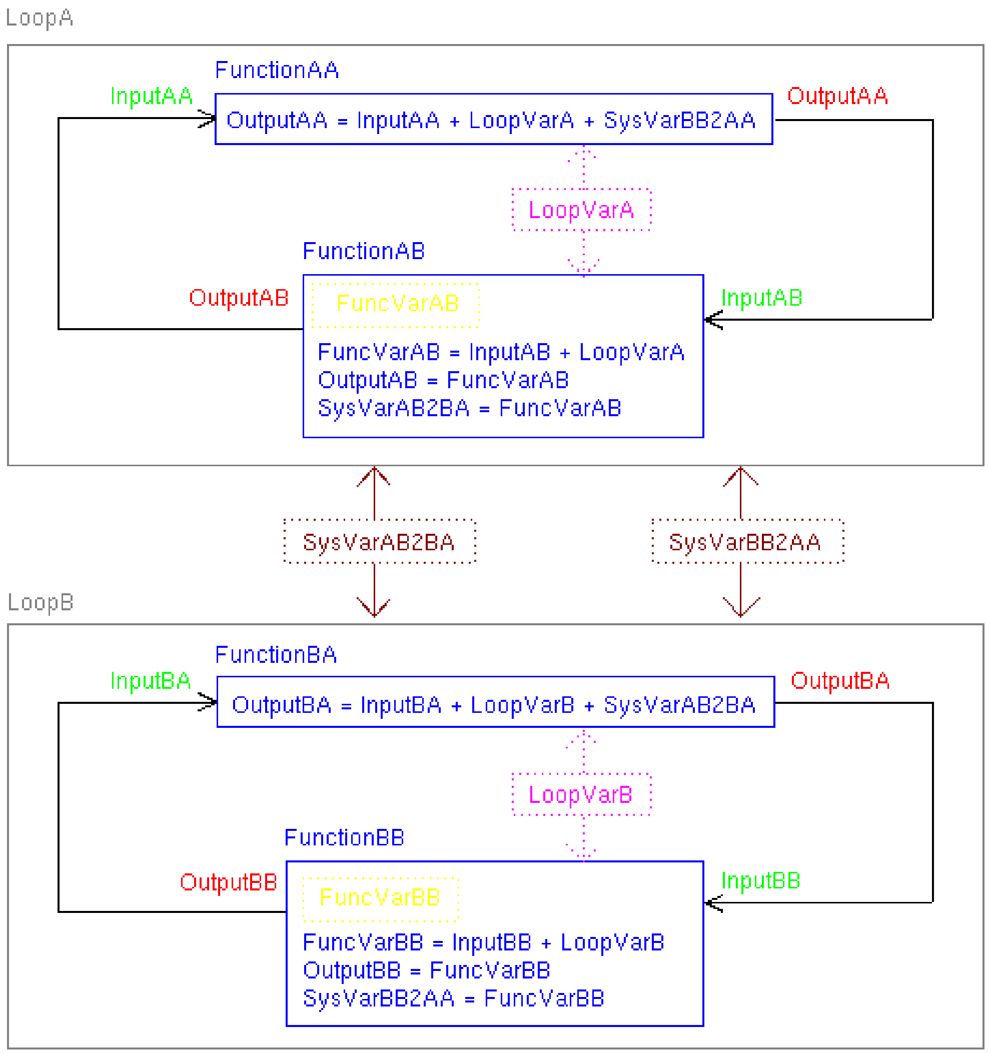

Figure 1: Dynamic system implemented in the tutorial

- **Function outputs**: These are the variables that store the output values of the functions. Outputs are represented as a red label next to an arrow leaving a function box. There are four outputs, one in each function:
  - *OutputAA* will contain real values.
  - *OutputAB* will contain integer values.
  - *OutputBA* will contain real values.
  - *OutputBB* will contain integer values.
- **Connections**: Every function input is connected to an output, so the input obtain its value from the output connected to it. Connections are represented with black arrows.
- **Loops**: A set of functions connected through inputs and outputs forms a loop. There are two loops represented as gray boxes containing their functions. The time base of the system is discrete: this means that the operation of their functions will only be performed at certain instants of time. In the example, each loop has a different time base:
  - *LoopA* is time-based, its operations will be performed once each three seconds.
  - *LoopB* is event-based, its operations will be performed once each time a certain event occurs.
- **Loop Variables**: These are an alternative way of comunicating functions in the same loop. There are two loop variables represented in magenta:
  - *LoopVarA* will contain integer values.
  - *LoopVarB* will contain real values.
- **System variables**: These are the only way to communicate between functions of different loops. There are two system variables represented in brown:
  - *SysVarAB2BA* will contain real values.
  - *SysVarBB2AA* will contain integer values.

Following sections describe one by one all the components available to define dynamic systems and how to use them to implement this or any other dynamic system.

3 Defining a dynamic system

The dynamic system component represents the entire dynamic system, and defines its general properties. First thing which is to be done is to create this component from its correspondant class:

```
CDynamicSystem* pDynamicSystem = new CDynamicSystem();
```

The dynamic system component defines the maximum execution time for the system. This is done by calling its member function *SetTime*:

```
bool b_NoError = true;
bNoError = pDynamicSystem->SetTime(1000000000) && bNoError;
bNoError = pDynamicSystem->SetTime(5 * DynamicSystem->Time()) && bNoError;
```

As the argument of *SetTime* is the time in nanoseconds, limits the execution time to 5 seconds. The execution can be stopped before that time, how to do this will be explained later.

The execution time was set in two steps by using *Time* function, which returns the last specified execution time. This is made to avoid using the integer constant 50000000000, which may be wrongly parsed by GCC compiler. In DSLib, the name of the member functions that set values start with the word "Set". For each of these functions, there is always another one that returns the value and whose name is the same but without the word "Set". This is the case of *SetTime* and *Time*. In this tutorial, the correspondant function that returns the value is only mentioned if necessary, but it must be remembered that it always exists.

It may be noticed the use of *bNoError* variable. Every member function that makes a modification in any component of the dynamic system returns a boolean value telling if the modification was succesful. If the returned value is "false", a message telling what was wrong can be obtained. This will be seen this later.

4 Defining loops

The loop component represents a set of function blocks. Function blocks must be grouped in loops and there can be several loops in the same dynamic system. Wether two blocks are in the same loop or not determines which kind of connections can be stablished between the blocks. A loop also defines an execution scheduling which will be the execution scheduling for all its blocks.

In the current example there are two loops, *LoopA* and *LoopB*. *LoopA* could be created in the following way:

```
CLoop* pLoopA = new CLoop();
bNoError = pLoopA->SetName("LoopA") && bNoError;
```

SetName defined a name for the loop. Most of DSLib's components need an unique name which other components may use to refer to them. The name must be in C-identifier format.

The following explains how to define the execution scheduling of the loop and thus the execution scheduling of its function blocks. The type of scheduling is defined with *SetScheduling* function:

```
bNoError = pLoopA->SetScheduling(CLoop::PERIOD) && bNoError;
```

CLoop::PERIOD means time-based scheduling. The other possible value is *CLoop::INTERRUPT* that will be used *LoopB* and means event-based scheduling. If *SetScheduling* function is not used, the default scheduling type is time-based. As the loop is time-based, the period must be set:

```
bNoError = pLoopA->SetPeriod(300000000) && bNoError;
```

That sets three seconds between executions of the loop. In this tutorial when it is said "execution of a loop" it means the execution of all the operations defined in its function blocks. More about blocks, their operations, how they are executed will be seen later.

Every loop is executed as a realtime task. There can be scheduling conflicts between realtime tasks, so at certain moments two or more are supposed to be executed at the same time. These conflicts are solved with a priority value, so the task with the highest priority is executed first and the others have to wait until it is finished. Execution priority of loops is defined with *SetPriority*:

```
bNoError = pLoopA->SetPriority(0) && bNoError;
```

As the default priority value is 0, calling last function did not make any difference. For more info about the scheduling, read the RTLinux documentation [1].

Floating point operations can be enabled or disabled in realtime tasks. If the code of a realtime task only works with integer variables, disabling floating point operations improves the efficiency of its execution. This must be used very carefully, if the task performs floating point operations and they are disabled the execution of the task will fail and stop. In case of doubt, leaving such operations enabled is safe. In the example, both loops work with real variables, so floating point operations must be enabled:

```
bNoError = pLoopA->SetFloatingPoint(true) && bNoError;
```

Floating point operations are enabled by default, so calling last function did not make any difference and it is not necessary to call it for *LoopB*. To disable floating point operations the function must be called with *false* as the argument.

LoopB is event-based, which means that the execution of the loop is triggered by an interrupt. The loop is created and its scheduling set to event-based in the following way:

```
CLoop *pLoopB = new CLoop();
bNoError = pLoopB->SetName("LoopB") && bNoError;
bNoError = pLoopB->SetScheduling(CLoop::INTERRUPT) && bNoError;
```

Since it is event-based the IRQ number that will trigger the execution of the loop must be set in the following way:

```
bNoError = pLoopB->SetIRQNumber(5) && bNoError;
```

Each time an interrupt is triggered in line 5, the loop will be executed.

So far three components of the dynamic system have been defined: the dynamic system and two loops. They are not related in any way, there is nothing that make them know about the existance of the others. Following sections explains how two join them to start constructing the dynamic system.

[1] RTLinux documentation can be downloaded from http://www.rtlinux.org

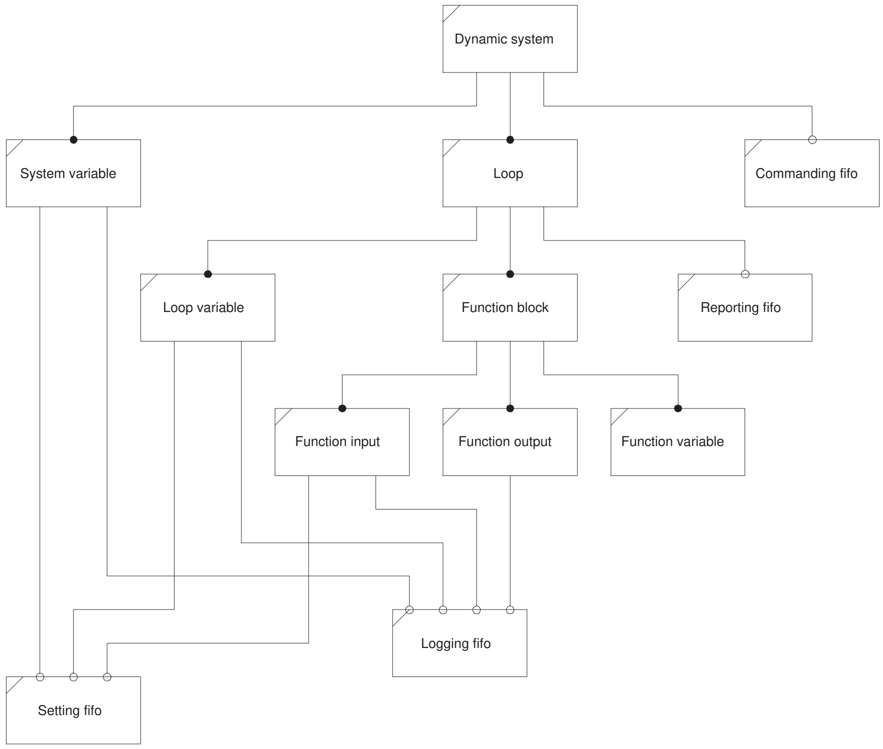

Figure 2: Organization of DSLib components in dynamic system definitions
Lines represent container-contained relationships, where the containers are above their contained components. An empty circle means that only one instance of the component can be contained, a filled circle means that many can be contained.

5 Containers and components: adding loops

With DSLib, dynamic systems are defined by sets of related components. This section explains how to join them in a dynamic system definition.

When a dynamic system is being defined, components must be added to other components in a tree structure of container-contained componets. Every component can contain several other components, but it can only have one container. Each type of component has a set of suitable container types, so not any component can be added to any other. Obviously there must be a component which is the root container, the one that is not contained by other components. This is the role of the dynamic system component. The tree structure of valid container-contained relationships is represented in figure 2.

The only suitable container for the loops is the dynamic system. Components are added to other components in the following way:

```
bNoError = pDynamicSystem->AddComponent(pLoopA) && bNoError;
bNoError = pDynamicSystem->AddComponent(pLoopB) && bNoError;
```

Following sections explain the rest of components and which are their suitable containers.

6 Defining and adding function blocks

As it has been stated in section 1, DSLib intends to provide a block-diagram design approach to define dynamic systems. One of the components of DSLib is the function block, which follows the same concept than Simulink's function blocks. A function block consist of a set of operations to calculate some outputs from some inputs. Inputs, outputs and operations can be freely defined by the user.

The function block component for *FunctionAA* can be created and named like this:

```
CFunction* pFunctionAA = new CFunction();
bNoError = pFunctionAA->SetName("FunctionAA") && bNoError;
```

The operations to be performed by the block are defined in C code as the argument of *SetCode* function:

```
bNoError = pFunctionAA->SetCode(
  "OutputAA = InputAA + LoopVarA + SysVarBB2AA;") && bNoError;
```

Any C code is valid as the operations of the function block, not only mathematical operations. However, there are some restrictions. The operations of the block will be executed as a realtime task, and most system services can not be requested from such tasks in RTLinux. For example, file input/output or dynamic memory allocation cannot be performed. Thus, the operations that are forbidden for realtime tasks are also forbiden for function blocsk. See the RTLinux documentation [2] for more info about the restrictions for realtime tasks.

[2]RTLinux documentation can be downloaded from http://www.rtlinux.org

Function blocks are contained by loops, as it will be seen at the end of this section. The operations of the function block are performed once at the instants defined by the scheduling of its loop, but iterative loops can be part of this operations.

SetCode only accepts a single string as its arguments, but many times the operations of a function block must be performed in several C lines. In this case, the lines must be separated by the end-of-line C symbol, "\n". This happens in *FunctionAB*'s operations:

```
CFunction* pFunctionAB = new CFunction();
bNoError = pFunctionAB->SetName("FunctionAB") && bNoError;
bNoError = pFunctionAB->SetCode(
  "FuncVarAB = InputAB + LoopVarA;\n"
  "OutputAB = (int)FuncVarAB;\n"
  "SysVarAB2BA = FuncVarAB;") && bNoError;
```

The last line does not need to be ended with "\n". The rest of function blocks of the example can be created and defined in the following way:

```
CFunction* pFunctionBA = new CFunction();
bNoError = pFunctionBA->SetName("FunctionBA") && bNoError;
bNoError = pFunctionBA->SetCode(
  "OutputBA = InputBA + LoopVarB + SysVarAB2BA;") && bNoError;
CFunction* pFunctionBB = new CFunction();
bNoError = pFunctionBB->SetName("FunctionBB") && bNoError;
bNoError = pFunctionBB->SetCode(
  "FuncVarBB = (int)InputBB + (int)LoopVarB;\n"
  "OutputBB = FuncVarBB;\n"
  "SysVarBB2AA = FuncVarBB;") && bNoError;
```

Function blocks must be added to loops. It was stated before that the scheduling of the loop will be the scheduling of its functions blocks. Also, it will be seen later that function blocks transmit values to each other in different ways depending on if they are in the same loop or in different ones. Scheduling and communication must be taken into consideration when deciding how to group function blocks in loops. In our example, *FunctionAA* and *FunctionAB* must be added to *LoopA* and *FunctionBA* and *FunctionBB* must be added to *LoopB* in the following way:

```
bNoError = pLoopA->AddComponent(pFunctionAA) && bNoError;
bNoError = pLoopA->AddComponent(pFunctionAB) && bNoError;
bNoError = pLoopB->AddComponent(pFunctionBA) && bNoError;
bNoError = pLoopB->AddComponent(pFunctionBB) && bNoError;
```

The operations of the defined function blocks make use of variables that have not been defined. These variables are function inputs, function outputs, loop variables and system variables. Following chapters explain how to define them.

7 Defining and adding outputs

Function outputs store the output values of the function blocks that contains them. From the operations of the function block, function outputs are referred as C variables. In the dynamic system definition, function blocks are components that must be created. The following code creates the function output component for *OutputAA*:

```
CFunctionOutput* pOutputAA = new CFunctionOutput();
bNoError = pOutputAA->SetName("OutputAA") && bNoError;
```

The name of the component will be the name of the correspondant variable in the operations. This variable also has a type which must be specified with *SetType* function. In the example *OutputAA* stores real values so the suitable C type is *float*:

```
bNoError = pOutputAA->SetType("float") && bNoError;
```

The definition of *OutputAA* will produce the following implicit declaration in *FunctionA*'s operations:

```
float OutputAA;
```

This declaration is local to the function block, so the variable can't be refered in other functions. The variable has an undetermined value at the begining of each execution of the function block's operations, so it should be read only if it has been asigned a value before. The value stored in the variable at the end of the execution will be the value of the output.

The rest of outputs can be created and defined in the following way:

```
CFunctionOutput* pOutputAB = new CFunctionOutput();
bNoError = pOutputAB->SetName("OutputAB") && bNoError;
bNoError = pOutputAB->SetType("int") && bNoError;
CFunctionOutput* pOutputBA = new CFunctionOutput();
bNoError = pOutputBA->SetName("OutputBA") && bNoError;
bNoError = pOutputBA->SetType("float") && bNoError;
CFunctionOutput* pOutputBB = new CFunctionOutput();
bNoError = pOutputBB->SetName("OutputBB") && bNoError;
bNoError = pOutputBB->SetType("int") && bNoError;
```

Naturally, function outputs must be added to their correspondant function blocks:

```
bNoError = pFunctionAA->AddComponent(pOutputAA) && bNoError;
bNoError = pFunctionAB->AddComponent(pOutputAB) && bNoError;
bNoError = pFunctionBA->AddComponent(pOutputBA) && bNoError;
bNoError = pFunctionBB->AddComponent(pOutputBB) && bNoError;
```

8 Defining and adding inputs

Function inputs have very similar features to function outputs, the main difference is that they store the input values of the function blocks that contains them. From the operations of the function block, function inputs are referred as C variables. In the dynamic system definition, function blocks are components that must be created. This section assumes that section 7 has been read. Function inputs are created and defined as if they were function outputs. For the example of the tutorial it could be done in the following way:

```
CFunctionInput* pInputAA = new CFunctionInput();
bNoError = pInputAA->SetName("InputAA") && bNoError;
bNoError = pInputAA->SetType("int") && bNoError;
CFunctionInput* pInputAB = new CFunctionInput();
bNoError = pInputAB->SetName("InputAB") && bNoError;
bNoError = pInputAB->SetType("float") && bNoError;
CFunctionInput* pInputBA = new CFunctionInput();
bNoError = pInputBA->SetName("InputBA") && bNoError;
```

```
bNoError = pInputBA->SetType("int") && bNoError;
CFunctionInput* pInputBB = new CFunctionInput();
bNoError = pInputBB->SetName("InputBB") && bNoError;
bNoError = pInputBB->SetType("float") && bNoError;
```

In the example of the tutorial each function has a single input and a single output, but functions blocks can have any number of function inputs and function outputs, it is just a question of creating, defining and adding them. Function blocks with neither inputs nor outputs make sense because they can use other ways of transmiting values to each other and they can communicate with hardware from their operations.

The function inputs of the example can be added to their function blocks in the following way:

```
bNoError = pFunctionAA->AddComponent(pInputAA) && bNoError;
bNoError = pFunctionAB->AddComponent(pInputAB) && bNoError;
bNoError = pFunctionBA->AddComponent(pInputBA) && bNoError;
bNoError = pFunctionBB->AddComponent(pInputBB) && bNoError;
```

The correspondant variables will store the value of the input at every execution of the operations of their blocks. The variables can be writen but their values will be overwritten at the begining of the next execution.

The utility is not the only difference between function inputs and function outputs, function inputs must be connected to outputs and an initial value can be defined for them. Conections will be seen in the following section and initializations in section 16.

9 Connecting inputs to outputs

It has been stated that function inputs store the value of the input at the begining of each execution of their function block's operations. The question now is the origin of that value. As it may be guessed, the value is obtained from a function output. Function inputs and function outputs can be connected, which stablishes an assignation of the value of the output to the input.

The connections between function inputs and outputs must follow a set of rules:

1. Every connection is between a function input and a function output, it is not possible to connect outputs to outputs or inputs to inputs.

2. The connections must be made between variables of the same C data type, it is not possible to connect a "float" to an "integer" for example.

3. Connection must be made between function inputs and function outputs of blocks contained in the same loop. Blocks of different loops must use other ways of transmiting values that will be explained.

4. Several function inputs can be connected to the same output, but different function outputs cannot be connected to the same input.

5. All function inputs must be connected, but function outputs can be left unconnected.

The connections must be stablished in the function input components by using their *Connect* function. In the example, thay can be made in the following way:

```
bNoError = pInputAA->Connect(pOutputAB) && bNoError;
bNoError = pInputAB->Connect(pOutputAA) && bNoError;
bNoError = pInputBA->Connect(pOutputBB) && bNoError;
bNoError = pInputBB->Connect(pOutputBA) && bNoError;
```

It may be noticed that the connections have been made after adding the function inputs to their function blocks, that means, the properties of the components have been modified after adding them to their correspondant containers. To explain why this is possible we must think of the components as the C++ objects that they are. **DSLib works with pointers to the original objects**, it never makes copies of them. Adding an object means adding a pointer to the object, not a copy of the object. Once the objects are created, setting their properties and adding them to their containers can be made in any order. In the other hand, if a pointer to an object is added to any component and then the object is deleted, DSLib will work with a pointer to a no-existing object and will produce an error.

10 Defining and adding function variables

The operations of the function blocks can make use of variables for temporary storage of values that will not be tranmited to other function blocks. The function variable component is intended for that. It defines a local variable that will not have a determined value at the begining of the execution of the operations and will loose its value at the end. Thus, they cannot be used to transmit values between blocks. In the example, *FunctionAB* uses *FuncVarAB* to store the result of adding *LoopVarA* to *InputAB* before assigning it to *OutputAB* and *SysVarAB2BA*. It has been determined that it would store real values.

This section assumes that section 7 has been read. Function variables are created, defined and added to function blocks like function outputs. This could be done for *FuncVarAB*:

```
CFunctionVariable* pFuncVarAB = new CFunctionVariable();
bNoError = pFuncVarAB->SetName("FuncVarAB") && bNoError;
bNoError = pFuncVarAB->SetType("float") && bNoError;
bNoError = pFunctionAB->AddComponent(pFuncVarAB) && bNoError;
```

Using function variable components has the same effect that making local declarations in the operations of the function block. In the case of *FuncVarAB*, doing the following instead of using the function variable component does not make any difference:

```
bNoError = pFunctionAB->SetCode(
  "float FuncVarAB;\n"
  "FuncVarAB = InputAB + LoopVarA;\n"
  "OutputAB = (int)FuncVarAB;\n"
  "SysVarAB2BA = OutputAB;") && bNoError;
```

It may be wondered what is the reason for using function variable components. DSLib checks that the names of the variables are not repeated in the scope of the operations. If a local declaration is used, that variable cannot be considered in the checking. If the local declaration repeats an identifier in the scope, the implementation of the dynamic system generated by DSLib will fail in compilation. Using function variables components does not produce any redundant code in the implementation of the dynamic system, so **users are encouraged to always use function variables instead of local declarations**.

FuncVarBB, which stores integer values, can be added to its function block in the same way:

```
CFunctionVariable* pFuncVarBB = new CFunctionVariable();
bNoError = pFuncVarBB->SetName("FuncVarBB") && bNoError;
bNoError = pFuncVarBB->SetType("int") && bNoError;
bNoError = pFunctionBB->AddComponent(pFuncVarBB) && bNoError;
```

11 Defining and adding loop variables

It has been mentioned that there are alternative ways to transmit values between function blocks apart from inputs and outputs. Loop variable components are one of them. These components define variables whose scope is the operations of all the function blocks of a certain loop. The variables can be referred in the code of any function of the loop, either for setting as well as for getting their value.

This section assumes that section 7 has been read. Loop variables are created and defined like any other component that defines a variable. In the case of *LoopVarA*:

```
CLoopVariable* pLoopVarA = new CLoopVariable();
bNoError = pLoopVarA->SetName("LoopVarA") && bNoError;
bNoError = pLoopVarA->SetType("int") && bNoError;
```

This will produce the following implicit declaration in all the functions of the loop:

```
int LoopVarA;
```

Loop variables have the special property of keeping their value between executions of the loop. If the operations of a function block depend on some results of its previous executions, loop variable components must be used. Overwriting the variable is the only way of modifying its value.

LoopVarB can be created and defined in the following way:

```
CLoopVariable* pLoopVarB = new CLoopVariable();
```

```
bNoError = pLoopVarB->SetName("LoopVarB") && bNoError;
bNoError = pLoopVarB->SetType("real") && bNoError;
```

Another special feature of loop variable components is that the must be added to loops:

```
bNoError = pLoopA->AddComponent(pLoopVarA) && bNoError;
bNoError = pLoopB->AddComponent(pLoopVarB) && bNoError;
```

12 Defining and adding system variables

System variables are another way of transmiting values between function blocks. They are similar to loop variables, but their scope is the entire dynamic system. They can be referred in the operations of any of its function blocks, for setting as well as for getting their value.

System variables are created and defined as any other component that defines a variable. Again, this has been learned in section 7. For *SysVarAB2BA*, it could be done in the following way:

```
CSystemVariable* pSysVarAB2BA = new CSystemVariable();
bNoError = pSysVarAB2BA->SetName("SysVarAB2BA") && bNoError;
bNoError = pSysVarAB2BA->SetType("float") && bNoError;
```

In this case the implicit declaration will be in the operations of all the function blocks of the dynamic system:

```
float SysVarAB2BA;
```

Like loop variables, system variables keep their value between executions, so it is only modified when it is overwritten.

The creation and definition of *SysVarBB2AA* could be like this:

```
CSysVarBB2AA* pSysVarBB2AA = new CSystemVariable();
bNoError = pSysVarBB2AA->SetName("SysVarBB2AA") && bNoError;
bNoError = pSysVarBB2AA->SetType("int") && bNoError;
```

System variables are added to the dynamic system component:

```
bNoError = pDynamicSystem->AddComponent(pSysVarAB2BA) && bNoError;
bNoError = pDynamicSystem->AddComponent(pLoopVarBB2AA) && bNoError;
```

13 When to use function input and outputs, loop variables or system variables

Three ways of transmiting values between function blocks have been described, but not when to use each of them. This chapter discusses this topic.

Loop variables or connections between function inputs and outputs will be considered first. Since loop variables can be read and written from different functions blocks, they can be used to communicate them as function inputs and function outputs do. However, loop variables are not intended to communicate functions blocks. In the execution of a DSLib loop, the operations of every function block are executed after its inputs are updated by its previous blocks. DSLib looks for an ordination of function blocks that acomplishes this, depending on the connections between them. DSLib cannot know which function blocks write the loop variables and which read them, so using loop variables instead of connections produces a random ordination of function blocks with the risk of unexpected behaviour of the dynamic system.

The aim of loop variables is to store values that must be kept between executions of the loop, such as state values or parameter values of the loop. These values may be read by several function blocks, and using function inputs and function outputs to transmit them to each other can make the definition of the dynamic system too complex. Loop variables are supposed to be written by one or few function blocks of the dynamic system. Loop variables can work as parameters because their value can be set from outside the dynamic system during its execution, as it will be seen later.

Regarding system variables, their purpose is to transmit values between function blocks of different loops, as well as store state or parameter values of the system. The criteria about when to use them is clear. If the variable must be referred by functions blocks of different loops, it can only be a system variable. If the variable is only referred in functions of the same loop, another type of variable component should be used. This is because system variables are protected against concurrent access. Since the variable can be referred from different loops, and thus different realtime task, one task could interrupt an writing operation on the variable and then read an wrong value from it. Fortunately read and writing operations on system variables cannot be interrupted, but this may produce delays on higher priority tasks. Consequently, system variables may reduce the performance of the execution of the dynamic system.

14 Next step: Arranging the dynamic system

When the definition of the dynamic system is finished, *ArrangeAll* function of the dynamic system component must be called. *ArrangeAll* performs some internal operations and values updates that are unrelevant for the user.

Considering how *bNoError* variable has been used, it should be *false* if one or more of the functions returned false. If things have been made as explained, at this point its value is *true* meaning that all the operations succeeded. The dynamic system component and all its loops, function blocks and variables have been defined, so it seems to be the moment for the final arrange:

```
bNoError = pDynamicSystem->ArrangeAll() && bNoError;
```

ArrangeAll must be called after the last modification of the dynamic system and before generating its realtime implementation. Once *ArrangeAll* has been called, the source can be generated several times without calling it again. However, if the definition of the dynamic system is modified in some way, *ArrangeAll* must be called again before generating the realtime implementation.

At this point *bNoError* should be *false*, which means that the arrange operation failed. Next section explains how to know what the problem is.

15 Reading the errors report

It has been mentioned in the introduction that DSLib can detect errors in the definition of the dynamic system and inform the user about them. This is one of the most important features of DSLib, and this section explains how to make use of it.

Every function of DSLib that may make any modification in the dynamic system returns a boolean value. The value is *true* if the operation succeeded, and *false* if failed. These functions are *AddComponent*, *ArrangeAll* and all the functions whose name start with the word "Set". They can fail because of many reasons: the operation could not be performed, the value of the argument is not valid, the operation would produce an wrong dynamic system defintion, etcetera. DSLib is supposed to detect all the possible errors in the definition of a dynamic system. If DSLib did not detect any error, it will generate a realtime implementation that will compile and run without any problem. **There is an exception for this rule**. DSLib does not analize any C code. Any argument of a function that is supposed to be C code is included in the realtime implementation as it is. This happens with *SetCode*, for example. If the C code has errors, or is not C code at all, DSLib will not detect it but the realtime implementation will be wrong. *SetType* checks if the argument is a C identifier, but not if it has been defined.

Boolean values telling if the operations succeeded are not very meaningful, it is necessary to know what the problem are. This information can be get in the following way:

```
CErrors Errors;
pDynamicSystem->ReportErrors(Errors);
```

CErrors very similar to a STL vector class [3] . Some of the functions of the STL vector class are implemented in *CError*, but their names use hungarian notation [4]

ReportErrors is defined in every DSLib component. The function takes a *CErrors* vector as argument and after clearing it, the function may add *CError* objects to the vector. Everyone of that objects contains information about an error produced in the component or in any of its contained components. Calling *ReportErrors* in the *CDynamicSystem* component will return the errors produced in any of the components of the dynamic system.

ReportErrors returns *true* if any error has been reported, and *false* if not.

In *CError* objects, *Number* function returns the error number and *Message* a short explanation about the error. Since *CErrors* works like a STL vector, the errors report can be read in the following way:

```
CErrors::ConstIterator ErrorsIterator;
for (ErrorsIterator = Errors.Begin();
   ErrorsIterator != Errors.End();
   ErrorsIterator++)
{
```

[3]Standard Template Library is a C++ library that defines data structures, such as vector.

[4]Hungarian notation defines a way of naming functions and variables. It is the notation used in this library.

```
    cout << (*ErrorsIterator).Number() << endl;
    cout << (*ErrorsIterator).Message() << endl;
}
```

This should produce the following outuput:

```
502
Block couldn't be ordered
502
Block couldn't be ordered
```

Still, the information obtained about the errors is not enough. It is also possible to know where the errors have been produced by using the function *Location* of *CError* This function returns a pointer to the component that produced the error. The pointer is a pointer to a *CCodeGenerator* class, which is the base class for all the DSLib components. *CCodeGenerator* and all the components have a *Class* function that returns the actual class identifier for the component. Using this function makes it possible to know which type-casting is possible for the value of the pointer.

The following code checks for every error if *Location* points to the class of the function block component (*CFunction*), and in that case performs a type-casting to display the name of the component:

```
CFunction* pFunction;
for (ErrorsIterator = Errors.Begin();
    ErrorsIterator != Errors.End(); ErrorsIterator++)
  if (((*ErrorsIterator).Location())->Class() == "CFunction")
  {
    pFunction = (CFunction*)((*ErrorsIterator).Location());
    cout << pFunction->Name() << endl;
  }
```

The output should be:

```
FunctionAA
FunctionAB
```

So functions *FunctionAA* and *FunctionAB* could not be ordered. It will be seen in next section that they could not be ordered because no input variable has been initalized.

There is an important thing about *ReportErrors* that must be known: It may not report all the errors in the definition of the dynamic system, but it always report some of them. Every time a function call to modify the definition of a dynamic system fails, *ReportErrors* must be called, the reported errors fixed, and the operation that failed repeated. Repeating this process until no more errors are reported is the way of finding and fixing all the errors in a definition of a dynamic system.

16 Initializing inputs: Blocks ordination

It is possible to set the initial value of function inputs. This a very important step in the definition of a dynamic system, since the set of initialized function inputs cannot be freely chosen. DSLib depends on the set of initialized function inputs to find an efficient execution ordination of function blocks. This section explains how to find a right set.

Initialization function of the function input component defines the initialization for the variable. Such initialization is defined in C code as the argument of function.

The initial value of the function inputs was not determined in section 2, so it is supposed that *InputAA* and *InputAB* should start with 1. This can be defined in the following way:

```
pInputAA->SetInitialization("InputAA = 1;");
pInputAB->SetInitialization("InputAB = 1;");
```

The initalization can be defined with any C code, with the restrictions for realtime code described in section 6. The opeations of the initializations are performed once before the execution of the dynamic system is started.

It may be noticed that the boolean value returned by *SetInitialization* has been ignored. This is possible because *ReportErrors* returns a boolean value telling if there were errors reported. In case things are done in this way, *ReportErrors* must be always called to know if a function failed. The dynamic system can be arranged and the errors report displayed in the following way:

```
pDynamicSystem->ArrangeAll();
if (pDynamicSystem->ReportErrors(Errors))
{
  CFunctionInput* pFunctionInput;
  for (ErrorsIterator = Errors.Begin();
    ErrorsIterator != Errors.End(); ErrorsIterator++)
  {
    cout << (*ErrorsIterator).Number() << endl;
    cout << (*ErrorsIterator).Message() << endl;
    if (((*ErrorsIterator).Location())->Class() == "CFunctionInput")
    {
      pFunctionInput = (CFunctionInput*)((*ErrorsIterator).Location());
      cout << pFunctionInput->Name() << endl;
    }
  }
}
```

The output of this should be:

```
1401
Redundant initialization in input variable
InputAB
```

First the problem was that no function input was initialized, now the problem is that too many have been initialized. The following is an explanation of how to find the right set.

It has been mentioned that the operations of a function block are performed after its inputs are updated by the previous function blocks. When the dynamic system is arranged, DSLib looks for an execution ordination of the function blocks that acomplishes this. The algorithm to find the ordination in a certain loop could be the following:

1. Look for a function block that acomplishes:
    - All its inputs are connected to updated outputs.
    - The function block is not in the ordination yet.

If it cannot be found, the algorithm **failed** and stops.

2. Mark the found block as the next one in the ordination.

3. If all the blocks are in the ordination, the algorithm **succeeded** and stops.

4. Mark the outputs of the found function block as updated.

5. Go back to step 1.

This algorithm would never work, because at the begining all the function outputs are marked as not updated and there is not a function block whose inputs are connected to updated outputs. The problem could be solved with an algorithm that also accepts initialized inputs:

1. Look for a function block that acomplishes:

   - Every one of its inputs is initilized or connected to an updated output.
   - The function block is not in the ordination yet.

   If it cannot be found, the algorithm **failed** and stops.

2. Mark the found block as the next one in the ordination.

3. If all the blocks are in the ordination, the algorithm **succeeded** and stops.

4. Mark the outputs of the found function block as updated.

5. Go back to step 1.

There is still another problem with this version. If all the inputs of every function block are initialized, in every iteration of the algorithm any function block that is not in the ordination yet is a candidate to be the next one. The ordination of function blocks would be random, so their operations could be performed before their input are updated by the previous function blocks. The solution, then, is to obligate users to define a minimum set of initializations that makes the algorithm succeed. All the initializations must be necessary to find an ordination of function blocks:

1. Look for a function block that acomplishes:

   - Every one of its inputs is initilized or connected to an updated output.

- The function block is not in the ordination yet.

   If it cannot be found, the algorithm **failed** and stops.

2. Mark the found block as the next one in the ordination.

3. If all the blocks are in the ordination, the algorithm **succeeded** and stops.

4. Mark the outputs of the found function block as updated.

5. If any of these outputs is connected to an initialized input, the initialization is redundant. The algorithm **failed** and stops.

6. Go back to step 1.

In the example, *ArrangeAll* first failed because neither *InputAA* nor *InputAB* were initialized, so the algorithm could not find the first block.

When both inputs were initialized, the algorithm could start with *FunctionAA* or *FunctionAB*. It actually started with *FunctionAA*, and marked it as the first function block in the ordination. *OutputAA* was marked as updated, but then the algorithm found that it was connected to an initialized input, *InputAB*. The initialization is redundant, because *FunctionAB* could be choosen in the next iteration regardless of it. The algorithm failed and the error "Redundant initialization in input variable" was generated.

If the initialization of *InputAB* is removed, the algorithm can only start with *FunctionAA* and mark *OutputAA* as updated. In the second iteration, *InputAB* would be connected to an updated output, so the next block in the ordination would be *FunctionAB*. In this case the algorithm managed to put all the blocks of the loop in the ordination and would succeed.

The initialization of *InputAB* can be removed in the following way:

```
bNoError = true;
bNoError = pInputAB->SetInitialization("") && bNoError;
```

As it has been mentioned, next step after fixing errors is to repeat the operation that failed. In this case the operation is *ArrangeAll*, but if the system is arranged again it would fail because the set of initializations in *LoopB* is still wrong. No error about this was reported before, but if the errors report is generated again it would

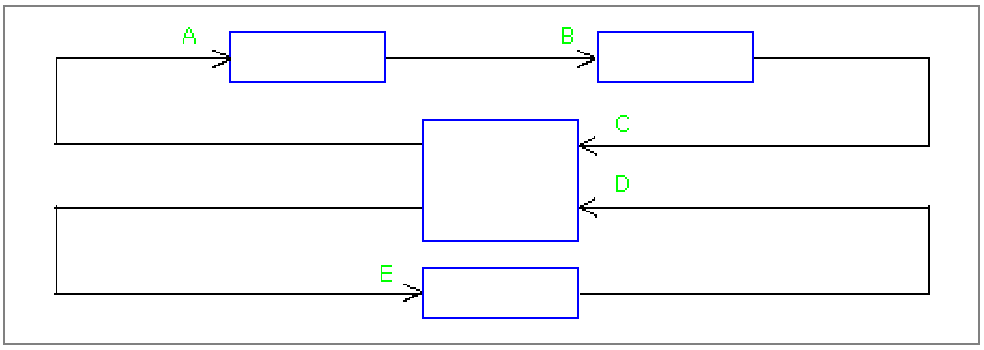

Figure 3: Function inputs initialization example
The valid sets of initializations are $\{A, D\}$, $\{A, E\}$, $\{B, D\}$, $\{B, E\}$, $\{C, D\}$ and $\{C, E\}$.

be reported now. This confirms that *ReportErrors* does not necessarily report all the errors, but it always reports some of them. The following fixes the problem:

```
bNoError = pInputBB->SetInitialization("InputBB = 1;") && bNoError;
```

If *ArrangeAll* is called now, it would succeed and return *true*, meaning that the initialization problems were solved. However, the dynamic system will not be arranged yet because more modifications will be made in the following sections.

An example of two singled-input function blocks is not very instructive to understand the problem of the right set of initializations. Figure 3 shows a more complicated one. Its valid sets of initializations are $\{A, D\}$, $\{A, E\}$, $\{B, D\}$, $\{B, E\}$, $\{C, D\}$ and $\{C, E\}$.

17 Initializing loop and system variables

It is possible to define the initial value of loop and system variables at the begining of the execution of the dynamic system. This is very similar to define the initial value of function inputs, but with the important difference of that any set of initializations can be used. This section describes how to define these initializations.

It is decided to initialize the loop and system variables of the example to 1. This section assumes that section 16 as been read. The initial value of the variables can be defined as if they were function inputs:

```
bNoError = pLoopVarA->SetInitialization("LoopVarA = 1;") && bNoError;
bNoError = pLoopVarB->SetInitialization("LoopVarB = 1;") && bNoError;
bNoError = pSysVarAB2BA->SetInitialization("SysVarAB2BA = 1;") && bNoError;
bNoError = pSysVarBB2AA->SetInitialization("SysVarBB2AA = 1;") && bNoError;
```

bNoError should be *true*.

18 Last step: Generating the module's code

The purpose of DSLib is to automatically generate the realtime implementation of a dynamic system. Finally, this section explains how to do this.

As it has been modified, it is necessary to arrange the dynamic system before generating the implementation.

```
bNoError = pDynamicSystem->ArrangeAll();
```

bNoError should be *true* now. This means that the dynamic system is correct and ready for code generation. The following code stores the realtime implementation of the dynamic system in a file named "rtmodule.c" of the current directory:

```
ofstream FileStream;
FileStream.open("rtmodule.c");
pDynamicSystem->GenerateSource(FileStream);
FileStream.close();
```

"rtmodule.c" should now be the C code for a Linux kernel module that is the realtime implementation of the dynamic system. "rtmodule.c" should be compiled to obtain binary of the kernel module. Since *ArrangeAll* returned *true*, the only compilation errors could be in the C code that defined the operations of the function blocks and the initialization of the variables. In this case it is right, so no compilation error should be produced.

After getting the module's binary, user must become *root* [5] and install it. Once it is installed, the only thing that may be noticed is that Linux runs slower. The realtime tasks that implement the dynamic system are actually running, but there

[5] "root" is the Linux administrator user, the one that has all the privileges in the operating system.

is no way to know what they are doing. This is because the dynamic system is not communicating with the "external" world. Following section explain how to add this communication.

19 Defining, adding and using logging fifos

A logging fifo is a Linux device file that logs the value of a variable during the execution of the dynamic system. Logging fifos can be read from Linux user tasks, so they are the tool to know how the dynamic system is performing.

Logging fifos can be used with function inputs, function outputs, loop variables and system variables. The following creates a logging fifo:

```
CLoggingFifo* pOutputAALogFifo;
pOutputAALogFifo = new CLoggingFifo();
```

Instead of a name, fifos need an unique number to identify them. The name can defined with function *SetNumber* in the following way:

```
bNoError = pOutputAALogFifo->SetNumber(0) && bNoError;
```

No other fifo can have the same number. The number will determine the name of the device file associated to the fifo, so the file always is "/dev/rtf*Number*". In this case it is "/dev/rtf0".

The size in bytes of the fifo must be defined. It is supposed that 20 values is enough capacity for the fifo. Since *OutputAA* is a *float*, the fifo will store *float* values. A *float* takes 4 bytes in the used operating system, so 80 bytes are needed:

```
bNoError = pOutputAALogFifo->SetSize(80) && bNoError;
```

Finally, the logging fifo is added to the variable that must be log, in this case *OutputAA*:

```
bNoError = pOutputAA->AddComponent(pOutputAALogFifo) && bNoError;
```

Suitable containers for logging fifos are function inputs, function outputs, loop variables and system variables.

The following explains how to use logging fifos during the execution of the dynamic system. A logging fifo is actually a RTLinux real time fifo, so it can be used in Linux user tasks as it would be done with any read-only realtime fifo. Since the dynamic system has been modified, it must be arranged and its source code generated as usual:

```
bNoError = pDynamicSystem->ArrangeAll();
FileStream.open("rtmodule.c");
pDynamicSystem->GenerateSource(FileStream);
FileStream.close();
```

When the compiled kernel module is installed, the following device file will be created:

```
/dev/rtf0
```

As it has been stated, the number after "rtf" is 0 because that is the number that was assigned to the fifo. To know the values that *OutputAA* is storing during the execution of the dynamic system, the device file must be read. The type of the values contained in the fifo is the type of the variable that it is being logged, in this case *float*. The fifo can be read and its values displayed with the following program:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
  int fd0;
  float real;
  fd0 = open("/dev/rtf0", O_RDONLY);
  while (read(fd0, &real, sizeof(real)))
    printf("/dev/rtf0 (OutputAA):\t%f\n", real);
  close (fd0);
  return 0;
}
```

This program should be run inmediately after installing the kernel module (when the fifos are created) and before removing it (when the fifos are destroyed). It should produce the following output:

```
/dev/rtf0 (OutputAA): 3.000000
/dev/rtf0 (OutputAA): 6.000000
/dev/rtf0 (OutputAA): 9.000000
/dev/rtf0 (OutputAA): 12.000000
/dev/rtf0 (OutputAA): 15.000000
/dev/rtf0 (OutputAA): 18.000000
/dev/rtf0 (OutputAA): 21.000000
/dev/rtf0 (OutputAA): 24.000000
/dev/rtf0 (OutputAA): 27.000000
/dev/rtf0 (OutputAA): 30.000000
/dev/rtf0 (OutputAA): 33.000000
/dev/rtf0 (OutputAA): 36.000000
/dev/rtf0 (OutputAA): 39.000000
/dev/rtf0 (OutputAA): 42.000000
/dev/rtf0 (OutputAA): 45.000000
/dev/rtf0 (OutputAA): 48.000000
/dev/rtf0 (OutputAA): 51.000000
/dev/rtf0 (OutputAA): 54.000000
```

The fifo is written with the value of *OutputAA* after each execution of *LoopA*. If 18 values have been read, that means that the operation of the function blocks of *LoopA* were performed 18 times. It will be checked now if the values are correct. As it has been seen in section *tutorial initializing inputs*, *FunctionAA* is executed before *FunctionAB* because *InputAA* is initialized. *SysVarBB2AA* keeps its initial vaule of 1 because *LoopB* is event-based and its interrupt is never triggered, so *LoopB* is never executed. *LoopVarA* is never written and keeps its initial value of 1. *InputAA* is initialized to 1. Thus:

```
OutputAA = InputAA + LoopVarA + SysVarBB2AA = 1 + 1 + 1 = 3;
FuncVarAB = InputAB + LoopVarA = OutputAA + LoopVarA = 3 + 1 = 4;
OutputAB = FuncVarAB = 4;
```

```
SysVarAB2BA = FuncVarAB = 4;
```

So *OutputAA* is 3 after the first execution. In the following one:

```
OutputAA = InputAA + LoopVarA + SysVarBB2AA =
  OutputAB + LoopVarA + SysVarBB2AA = 4 + 1 + 1 = 6;
FuncVarAB = InputAB + LoopVarA = OutputAA + LoopVarA = 6 + 1 = 7;
OutputAB = FuncVarAB = 7;
SysVarAB2BA = FuncVarAB = 7;
```

And so on.

As a RTLinux realtime fifo, reading the fifo from user tasks is not made in realtime. The user task can be delayed by the realtime tasks, so the writings can be made more frequently than the readings. In that case, the user task will be reading old values stored in the fifo, and not the current value of *OutputAA*. For more information about how real-time fifos work, read the RTLinux documentation [6].

Fifos to log the rest of variables can be created, defined and added in the following way:

```
CLoggingFifo* pOutputABLogFifo;
pOutputABLogFifo = new CLoggingFifo();
bNoError = pOutputABLogFifo->SetNumber(1) && bNoError;
bNoError = pOutputABLogFifo->SetSize(80) && bNoError;
bNoError = pOutputAB->AddComponent(pOutputABLogFifo) && bNoError;
CLoggingFifo* pOutputBALogFifo;
pOutputBALogFifo = new CLoggingFifo();
bNoError = pOutputBALogFifo->SetNumber(2) && bNoError;
bNoError = pOutputBALogFifo->SetSize(80) && bNoError;
bNoError = pOutputBA->AddComponent(pOutputBALogFifo) && bNoError;
CLoggingFifo* pOutputBBLogFifo;
pOutputBBLogFifo = new CLoggingFifo();
bNoError = pOutputBBLogFifo->SetNumber(3) && bNoError;
bNoError = pOutputBBLogFifo->SetSize(80) && bNoError;
```

[6] RTLinux documentation can be downloaded from http://www.rtlinux.org

```
bNoError = pOutputBB->AddComponent(pOutputBBLogFifo) && bNoError;
CLoggingFifo* pLoopVarALogFifo;
pLoopVarALogFifo = new CLoggingFifo();
bNoError = pLoopVarALogFifo->SetNumber(4) && bNoError;
bNoError = pLoopVarALogFifo->SetSize(80) && bNoError;
bNoError = pLoopVarA->AddComponent(pLoopVarALogFifo) && bNoError;
CLoggingFifo* pLoopVarBLogFifo;
pLoopVarBLogFifo = new CLoggingFifo();
bNoError = pLoopVarBLogFifo->SetNumber(5) && bNoError;
bNoError = pLoopVarBLogFifo->SetSize(80) && bNoError;
bNoError = pLoopVarB->AddComponent(pLoopVarBLogFifo) && bNoError;
CLoggingFifo* pSysVarAB2BALogFifo;
pSysVarAB2BALogFifo = new CLoggingFifo();
bNoError = pSysVarAB2BALogFifo->SetNumber(6) && bNoError;
bNoError = pSysVarAB2BALogFifo->SetSize(160) && bNoError;
bNoError = pSysVarAB2BA->AddComponent(pSysVarAB2BALogFifo) && bNoError;
CLoggingFifo* pSysVarBB2AALogFifo;
pSysVarBB2AALogFifo = new CLoggingFifo();
bNoError = pSysVarBB2AALogFifo->SetNumber(7) && bNoError;
bNoError = pSysVarBB2AALogFifo->SetSize(160) && bNoError;
bNoError = pSysVarBB2AA->AddComponent(pSysVarBB2AALogFifo) && bNoError;
```

Function inputs can also be logged, but this is not made because their connected outputs are being logged and it would report the same values.

Logging fifos of function inputs, function outputs and loop variables are written at the end of each execution of their loop. Logging fifos of system variables are written at the end of each execution of any loop in the dynamic system, so the number of values written in these fifos is the sum of the executions of every loop in the dynamic system. That is why it has been chosen a bigger size for fifos logging system variables.

20 Defining, adding and using setting fifos

A setting fifo is a Linux device fifo that sets the value of a variable during the execution of the dynamic system. Setting fifos can be written from user tasks, so they are the tool to transmit values to the dynamic system from outside of it. Setting fifos can only be used with function inputs, loop variables and system variables. This section explains how to do this.

This section assumes that section 19 has been read. A setting fifo can be created like a logging fifo:

```
CSettingFifo* pInputAASetFifo;
pInputAASetFifo = new CSettingFifo();
```

Like logging fifos, setting fifos need an unique number to identify them. As the range 0-7 has been already used for logging fifos, setting fifos must use other numbers:

```
bNoError = pInputAASetFifo->SetNumber(8) && bNoError;
```

Again, the number determines the name of the device file associated to the fifo, so it always is "/dev/rtf*Number*". The size in bytes of the fifo must also be set:

```
bNoError = pInputAASetFifo->SetSize(8) && bNoError;
```

Setting fifos are added to the variables that they will set. Their suitable containers are function inputs, loop variables and setting variables. This can be done in the following way:

```
bNoError = pInputAA->AddComponent(pInputAASetFifo) && bNoError;
```

Regarding usage, a setting fifo is a RTLinux realtime fifo. It can be used in Linux user tasks as any write-only realtime fifo. When the generated kernel module is installed, the following device file will be created:

```
/dev/rtf8
```

The number after "rtf" is the number that was assigned to the fifo, 8. Setting values in *InputAA* is made by writing this device file. The type of the values handled by the fifo is the type of the variable that it is being set, in this case *int*. From the user task, *InputAA* can be set to 3 in the following way:

```
int rtf8 = open("/dev/rtf8", O_WRONLY);
int integer = 3;
write(rtf8, &integer, sizeof(integer));
close(rtf8);
```

As a RTLinux realtime fifo, writing the fifo from user tasks is not made in realtime. The user task can be delayed by the realtime tasks, so it can take some time until the variable is set to the new value. Even it can never be set. For more information about how realtime fifos work, read the RTLinux documentation [7] .

Creating, defining and adding fifos to set the other variables of the dynamic system can be made in the following way:

```
CSettingFifo* pInputABSetFifo;
pInputABSetFifo = new CSettingFifo();
bNoError = pInputABSetFifo->SetNumber(9) && bNoError;
bNoError = pInputABSetFifo->SetSize(8) && bNoError;
bNoError = pInputAB->AddComponent(pInputABSetFifo) && bNoError;
CSettingFifo* pInputBASetFifo;
pInputBASetFifo = new CSettingFifo();
bNoError = pInputBASetFifo->SetNumber(10) && bNoError;
bNoError = pInputBASetFifo->SetSize(8) && bNoError;
bNoError = pInputBA->AddComponent(pInputBASetFifo) && bNoError;
CSettingFifo* pInputBBSetFifo;
pInputBBSetFifo = new CSettingFifo();
bNoError = pInputBBSetFifo->SetNumber(11) && bNoError;
bNoError = pInputBBSetFifo->SetSize(8) && bNoError;
bNoError = pInputBB->AddComponent(pInputBBSetFifo) && bNoError;
CSettingFifo* pLoopVarASetFifo;
pLoopVarASetFifo = new CSettingFifo();
bNoError = pLoopVarASetFifo->SetNumber(12) && bNoError;
bNoError = pLoopVarASetFifo->SetSize(8) && bNoError;
bNoError = pLoopVarA->AddComponent(pLoopVarASetFifo) && bNoError;
CSettingFifo* pLoopVarBSetFifo;
```

[7] RTLinux documentation can be downloaded from http://www.rtlinux.org

```
pLoopVarBSetFifo = new CSettingFifo();
bNoError = pLoopVarBSetFifo->SetNumber(13) && bNoError;
bNoError = pLoopVarBSetFifo->SetSize(8) && bNoError;
bNoError = pLoopVarB->AddComponent(pLoopVarBSetFifo) && bNoError;
CSettingFifo* pSysVarAB2BASetFifo;
pSysVarAB2BASetFifo = new CSettingFifo();
bNoError = pSysVarAB2BASetFifo->SetNumber(14) && bNoError;
bNoError = pSysVarAB2BASetFifo->SetSize(8) && bNoError;
bNoError = pSysVarAB2BASetFifo->AddComponent(pSysVarAB2BASetFifo) && bNoError;
CSettingFifo* pSysVarBB2AASetFifo;
pSysVarBB2AASetFifo = new CSettingFifo();
bNoError = pSysVarBB2AASetFifo->SetNumber(15) && bNoError;
bNoError = pSysVarBB2AASetFifo->SetSize(8) && bNoError;
bNoError = pSysVarBB2AASetFifo->AddComponent(pSysVarBB2AASetFifo) && bNoError;
```

It is intended to write the fifos in a frequency smaller than the execution of the loop, so a capacity of 2 values in each of them is enough.

21 Defining, adding and using reporting fifos

A reporting fifo is a Linux device fifo that reports the execution times of a loop during the execution of the dynamic system. Reporting fifos can be read from Linux user tasks, so they are the tool to know if the loop is being executed as scheduled. Reporting fifos can only be used with loops.

This section assumes that section 19 has been read. Logging fifos can be created like any other fifo:

```
CReportingFifo* pLoopARepFifo;
pLoopARepFifo = new CReportingFifo();
```

They also need an unique fifo number and a size in bytes. Values stored in reporting fifos are of type *hrtime_t*, which takes 8 bytes in the used operating system (read RTLinux documentation [8] for more information about *hrtime_t* type). To be able to store 40 values, 320 bytes are needed:

[8] RTLinux documentation can be downloaded from http://www.rtlinux.org

```
bNoError = pLoopARepFifo->SetNumber(16) && bNoError;
bNoError = pLoopARepFifo->SetSize(320) && bNoError;
```

The only suitable containers for reporting fifos are the loops. Reporting fifos are added to the loop whose execution times must be reported:

```
bNoError = pLoopA->AddComponent(pLoopARepFifo) && bNoError;
```

That is all about defining and adding logging fifos. Like any other fifo, a logging fifo is a RTLinux realtime fifo, so it can be used in user tasks as it would be done with any read-only realtime fifo. Again, the dynamic system must be aranged, and its source code generated and compiled. When the kernel module is installed the following device file will be created:

```
/dev/rtf16
```

To know the excution times of *LoopA*, the following program can be used:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <rtl_time.h>
int main()
{
  int rtf16;
  hrtime_t time;
  rtf16 = open("/dev/rtf16", O_RDONLY);
  while (read(rtf16, &time, sizeof(time)))
    printf("/dev/rtf16 (LoopA):\t%d\n", time);
  close (rtf16);
  return 0;
}
```

It should be run as soon as the kernel module is installed, for the reasons explained in section 19. It should produce the following output:

```
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 8448
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 23328
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 300020288
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 300045344
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 600018944
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 600043008
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 900024288
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 900047744
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 1200019648
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 1200043712
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 1500018784
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 1500042656
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 1800019072
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 1800043072
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): 2100020128
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): 2100043776
/dev/rtf16 (LoopA): 0
/dev/rtf16 (LoopA): -1894948000
```

```
/dev/rtf16 (LoopA): 1
/dev/rtf16 (LoopA): -1894924192
...
```

At the begining of each execution of the loop the fifo is writen twice: first, the value "0", and next, the current time. At the end of each execution of the loop the fifo is written twice: first, the value "1", and next, the current time. So the value "0" in the fifo means "next value is an starting time" and "1" means "next value is an ending time". Times are in nanosecons since the starting of the execution of the dynamic system.

It can be seen that the loop is performing the execution period of 0.3 seconds very accurately. At certain point, displayed execution times start to be negative. That is just a problem with the displaying of *hrtime_t* values, and the real values are correct. Only part of the output is reproduced here. The whole output indicates that the loop has been executed 18 times. As *OutputAA*'s logging fifo returned so many values as executions of the loop, that means the fifo is not losing any value. If a logging fifo returns less values than executions were reported by the reporting fifo, it means that the capacity of the logging fifo is too small.

As a RTLinux realtime fifo, reading the fifo from user tasks is not made in realtime. The user task can be delayed by the realtime tasks, so the writings can be made more frequently than the readings. In that case, the user task will be reading old values stored in the fifo, and not the current execution times. For more information about how realtime fifos work, read the RTLinux documentation [9] .

To report the execution times of *LoopB*:

```
CReportingFifo* pLoopBRepFifo;
pLoopBRepFifo = new CReportingFifo();
bNoError = pLoopBRepFifo->SetNumber(17) && bNoError;
bNoError = pLoopBRepFifo->SetSize(320) && bNoError;
bNoError = pLoopB->AddComponent(pLoopBRepFifo) && bNoError;
```

[9] RTLinux documentation can be downloaded from http://www.rtlinux.org

22 Defining, adding and using commanding fifos

A commanding fifo is a Linux device fifo that makes it possible to start and stop the execution of the dynamic system from user tasks. Commanding fifos can be written from user tasks, so they are the tool to control the execution of the dynamic system.

It may have been noticed that the execution of the dynamic system started as soon as the kernel module was installed. When the execution finishes, the module kept installed but there was no way to make it run again. A commanding fifo will allow to run and stop the system when needed:

Again, this section assumes that section 19 has been read. To create the commanding fifo:

```
CCommandingFifo* pCommandingFifo;
pCommandingFifo = new CCommandingFifo();
```

Like any other fifo, commanding fifos need an unique number to identify them:

```
bNoError = pCommandingFifo->SetNumber(18) && bNoError;
```

Again, the number determines the name of the device file associated to the fifo. The size in bytes of the fifo must also be set. Commanding fifos store *int* values, and it is considered that two of them will be enough:

```
bNoError = pCommandingFifo->SetSize(8) && bNoError;
```

Commanding fifos can only start and stop the whole dynamic system, and not individual loops. Their suitable container is the dynamic system component:

```
bNoError = pDynamicSystem->AddComponent(pCommandingFifo) && bNoError;
```

Regarding usage, a commanding fifo is also a RTLinux realtime fifo. that can be used in user tasks as a write-only realtime fifo. After arranging the dynamic system, generating and compiling its implementation, and installing the kernel module, the following device file will be created:

```
/dev/rtf18
```

To start the execution, the value "1" must be written in the commanding fifo. If the kernel module is installed, the following program starts the execution of the dynamic system:

```
#include <stdio.h>
#include <cntl.h>
#include <unistd.h>
int main()
{
  int rtf18;
  int command;
  rtf18 = open("/dev/rtf18", O_WRONLY);
  command = 1;
  write(rtf18, &command, sizeof(command));
  close (rtf18);
  return 0;
}
```

Writing "0" to the commanding fifo stops the execution. Running the following program makes this:

```
#include <stdio.h>
#include <cntl.h>
#include <unistd.h>
int main()
{
  int rtf18;
  int command;
  rtf18 = open("/dev/rtf18", O_WRONLY);
  command = 0;
  write(rtf18, &command, sizeof(command));
  close (rtf18);
  return 0;
}
```

It should be remembered that dynamic systems have a maximum execution time, so if it is not stopped with the commanding fifo the execution will stop when the maximum time is reached.

Again, as a RTLinux realtime fifo writing the fifo from user tasks is not made in realtime. The user task can be delayed by the realtime tasks, so it can take some time until the desired operation is performed or it can never be performed. For more information about how realtime fifos work, read the RTLinux documentation [10] .

23 Clearing the memory

When the dynamic system implementation has been generated, the components used to define the dynamic system are not needed any longer. As pointers to all the components have been kept, it is possible to delete them one by one. However, there is an easier way to do this:

```
pDynamicSystem->DeleteObjects();
```

That should delete all the components contained in the dynamic system, except the dynamic system component itself. This component could be deleted in the following way:

```
delete pDynamicSystem;
```

An easier way of working is to declare the dynamic system as an object, not a pointer to an object:

```
CDynamicSystem DynamicSystem;
...
DynamicSystem.AddComponent(...);
...
DynamicSystem.DeleteObjects();
```

In this case deleting the dynamic system component is not necesary.

[10] RTLinux documentation can be downloaded from http://www.rtlinux.org

24 Resume: Tutorial's example implementation

The following is the complete DSLib implementation of the example:

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <fstream>
#include <iostream>
#include <stdlib.h>
#include "dynamicsystem.h"
#include "function.h"
#include <vector>
#include "functioninput.h"
#include "functionoutput.h"
#include "systemvariable.h"
#include "loopvariable.h"
#include "loggingfifo.h"
#include "settingfifo.h"
#include "reportingfifo.h"
#include "commandingfifo.h"
#include "functionvariable.h"

int main(int argc, char *argv[])
{
  // Defining a dynamic system
  CDynamicSystem* pDynamicSystem = new CDynamicSystem();
  bool bNoError = true;
  bNoError = pDynamicSystem->SetTime(1000000000);
  bNoError = pDynamicSystem->SetTime(5 * pDynamicSystem->Time());

  // Defining and adding loops
  CLoop* pLoopA = new CLoop();
  bNoError = pLoopA->SetName("LoopA") && bNoError;
  bNoError = pLoopA->SetScheduling(CLoop::PERIOD) && bNoError;
  bNoError = pLoopA->SetPeriod(300000000) && bNoError;
  bNoError = pLoopA->SetPriority(0) && bNoError;
  bNoError = pLoopA->SetFloatingPoint(true) && bNoError;
  CLoop* pLoopB = new CLoop();
  bNoError = pLoopB->SetName("LoopB") && bNoError;
  bNoError = pLoopB->SetScheduling(CLoop::INTERRUPT) && bNoError;
  bNoError = pLoopB->SetIRQNumber(5) && bNoError;
  bNoError = pDynamicSystem->AddComponent(pLoopA) && bNoError;
  bNoError = pDynamicSystem->AddComponent(pLoopB) && bNoError;

  // Defining and adding blocks
  CFunction* pFunctionAA = new CFunction();
  bNoError = pFunctionAA->SetName("FunctionAA") && bNoError;
  bNoError = pFunctionAA->SetCode(
    "OutputAA = InputAA + LoopVarA + SysVarBB2AA;") && bNoError;
  CFunction* pFunctionAB = new CFunction();
  bNoError = pFunctionAB->SetName("FunctionAB") && bNoError;
  bNoError = pFunctionAB->SetCode(
    "FuncVarAB = InputAB + LoopVarA;\n"
    "OutputAB = (int)FuncVarAB;\n"
    "SysVarAB2BA = FuncVarAB;") && bNoError;
  CFunction* pFunctionBA = new CFunction();
  bNoError = pFunctionBA->SetName("FunctionBA") && bNoError;
  bNoError = pFunctionBA->SetCode(
    "OutputBA = InputBA + LoopVarB + SysVarAB2BA;") && bNoError;
  CFunction* pFunctionBB = new CFunction();
  bNoError = pFunctionBB->SetName("FunctionBB") && bNoError;
  bNoError = pFunctionBB->SetCode(
    "FuncVarBB = (int)InputBB + (int)LoopVarB;\n"
    "OutputBB = FuncVarBB;\n"
    "SysVarBB2AA = FuncVarBB;") && bNoError;
  bNoError = pLoopA->AddComponent(pFunctionAA) && bNoError;
  bNoError = pLoopA->AddComponent(pFunctionAB) && bNoError;
  bNoError = pLoopB->AddComponent(pFunctionBA) && bNoError;
  bNoError = pLoopB->AddComponent(pFunctionBB) && bNoError;

  // Defining and adding outputs
  CFunctionOutput* pOutputAA = new CFunctionOutput();
  bNoError = pOutputAA->SetName("OutputAA") && bNoError;
  bNoError = pOutputAA->SetType("float") && bNoError;
  CFunctionOutput* pOutputAB = new CFunctionOutput();
  bNoError = pOutputAB->SetName("OutputAB") && bNoError;
  bNoError = pOutputAB->SetType("int") && bNoError;
  CFunctionOutput* pOutputBA = new CFunctionOutput();
  bNoError = pOutputBA->SetName("OutputBA") && bNoError;
  bNoError = pOutputBA->SetType("float") && bNoError;
```

```
CFunctionOutput* pOutputBB = new CFunctionOutput();
bNoError = pOutputBB->SetName("OutputBB") && bNoError;
bNoError = pOutputBB->SetType("int") && bNoError;
bNoError = pFunctionAA->AddComponent(pOutputAA) && bNoError;
bNoError = pFunctionAB->AddComponent(pOutputAB) && bNoError;
bNoError = pFunctionBA->AddComponent(pOutputBA) && bNoError;
bNoError = pFunctionBB->AddComponent(pOutputBB) && bNoError;

// Defining and adding inputs
CFunctionInput* pInputAA = new CFunctionInput();
bNoError = pInputAA->SetName("InputAA") && bNoError;
bNoError = pInputAA->SetType("int") && bNoError;
CFunctionInput* pInputAB = new CFunctionInput();
bNoError = pInputAB->SetName("InputAB") && bNoError;
bNoError = pInputAB->SetType("float") && bNoError;
CFunctionInput* pInputBA = new CFunctionInput();
bNoError = pInputBA->SetName("InputBA") && bNoError;
bNoError = pInputBA->SetType("int") && bNoError;
CFunctionInput* pInputBB = new CFunctionInput();
bNoError = pInputBB->SetName("InputBB") && bNoError;
bNoError = pInputBB->SetType("float") && bNoError;
bNoError = pFunctionAA->AddComponent(pInputAA) && bNoError;
bNoError = pFunctionAB->AddComponent(pInputAB) && bNoError;
bNoError = pFunctionBA->AddComponent(pInputBA) && bNoError;
bNoError = pFunctionBB->AddComponent(pInputBB) && bNoError;

// Connecting inputs to outputs
bNoError = pInputAB->Connect(pOutputAA) && bNoError;
bNoError = pInputAA->Connect(pOutputAB) && bNoError;
bNoError = pInputBA->Connect(pOutputBB) && bNoError;
bNoError = pInputBB->Connect(pOutputBA) && bNoError;

// Defining and adding function variables
CFunctionVariable* pFuncVarAB = new CFunctionVariable();
bNoError = pFuncVarAB->SetName("FuncVarAB") && bNoError;
bNoError = pFuncVarAB->SetType("float") && bNoError;
CFunctionVariable* pFuncVarBB = new CFunctionVariable();
bNoError = pFuncVarBB->SetName("FuncVarBB") && bNoError;
bNoError = pFuncVarBB->SetType("int") && bNoError;
bNoError = pFunctionAB->AddComponent(pFuncVarAB) && bNoError;
bNoError = pFunctionBB->AddComponent(pFuncVarBB) && bNoError;

// Defining and adding loop variables
CLoopVariable* pLoopVarA = new CLoopVariable();
bNoError = pLoopVarA->SetName("LoopVarA") && bNoError;
bNoError = pLoopVarA->SetType("int") && bNoError;
CLoopVariable* pLoopVarB = new CLoopVariable();
bNoError = pLoopVarB->SetName("LoopVarB") && bNoError;
bNoError = pLoopVarB->SetType("float") && bNoError;
bNoError = pLoopA->AddComponent(pLoopVarA) && bNoError;
bNoError = pLoopB->AddComponent(pLoopVarB) && bNoError;

// Defining and adding system variables
CSystemVariable* pSysVarAB2BA = new CSystemVariable();
bNoError = pSysVarAB2BA->SetName("SysVarAB2BA") && bNoError;
bNoError = pSysVarAB2BA->SetType("float") && bNoError;
CSystemVariable* pSysVarBB2AA = new CSystemVariable();
bNoError = pSysVarBB2AA->SetName("SysVarBB2AA") && bNoError;
bNoError = pSysVarBB2AA->SetType("int") && bNoError;
bNoError = pDynamicSystem->AddComponent(pSysVarAB2BA) && bNoError;
bNoError = pDynamicSystem->AddComponent(pSysVarBB2AA) && bNoError;

// Initializing inputs: Blocks ordenation
pInputAA->SetInitialization("InputAA = 1;");
pInputAB->SetInitialization("InputAB = 1;");
pDynamicSystem->ArrangeAll() && bNoError;
if (pDynamicSystem->ReportErrors(Errors))
{
  CFunctionInput* pFunctionInput;
  for (ErrorsIterator = Errors.Begin();
    ErrorsIterator != Errors.End(); ErrorsIterator++)
  {
    cout << (*ErrorsIterator).Number() << endl;
    cout << (*ErrorsIterator).Message() << endl;
    if (((*ErrorsIterator).Location())->Class() == "CFunctionInput")
    {
      pFunctionInput = (CFunctionInput*)((*ErrorsIterator).Location());
      cout << pFunctionInput->Name() << endl;
    }
  }
}
bNoError = true;
bNoError = pInputAB->SetInitialization("") && bNoError;
bNoError = pInputBB->SetInitialization("InputBB = 1;") && bNoError;
```

```
// Initalizing loop and system variables
bNoError = pLoopVarA->SetInitialization("LoopVarA = 1;") && bNoError;
bNoError = pLoopVarB->SetInitialization("LoopVarB = 1;") && bNoError;
bNoError = pSysVarAB2BA->SetInitialization("SysVarAB2BA = 1;") && bNoError;
bNoError = pSysVarBB2AA->SetInitialization("SysVarBB2AA = 1;") && bNoError;

// Defining, adding and using logging fifos
CLoggingFifo* pOutputAALogFifo;
pOutputAALogFifo = new CLoggingFifo();
bNoError = pOutputAALogFifo->SetNumber(0) && bNoError;
bNoError = pOutputAALogFifo->SetSize(80) && bNoError;
bNoError = pOutputAA->AddComponent(pOutputAALogFifo) && bNoError;
CLoggingFifo* pOutputABLogFifo;
pOutputABLogFifo = new CLoggingFifo();
bNoError = pOutputABLogFifo->SetNumber(1) && bNoError;
bNoError = pOutputABLogFifo->SetSize(110) && bNoError;
bNoError = pOutputAB->AddComponent(pOutputABLogFifo) && bNoError;
CLoggingFifo* pOutputBALogFifo;
pOutputBALogFifo = new CLoggingFifo();
bNoError = pOutputBALogFifo->SetNumber(2) && bNoError;
bNoError = pOutputBALogFifo->SetSize(120) && bNoError;
bNoError = pOutputBA->AddComponent(pOutputBALogFifo) && bNoError;
CLoggingFifo* pOutputBBLogFifo;
pOutputBBLogFifo = new CLoggingFifo();
bNoError = pOutputBBLogFifo->SetNumber(3) && bNoError;
bNoError = pOutputBBLogFifo->SetSize(130) && bNoError;
bNoError = pOutputBB->AddComponent(pOutputBBLogFifo) && bNoError;
CLoggingFifo* pLoopVarALogFifo;
pLoopVarALogFifo = new CLoggingFifo();
bNoError = pLoopVarALogFifo->SetNumber(4) && bNoError;
bNoError = pLoopVarALogFifo->SetSize(140) && bNoError;
bNoError = pLoopVarA->AddComponent(pLoopVarALogFifo) && bNoError;
CLoggingFifo* pLoopVarBLogFifo;
pLoopVarBLogFifo = new CLoggingFifo();
bNoError = pLoopVarBLogFifo->SetNumber(5) && bNoError;
bNoError = pLoopVarBLogFifo->SetSize(150) && bNoError;
bNoError = pLoopVarB->AddComponent(pLoopVarBLogFifo) && bNoError;
CLoggingFifo* pSysVarAB2BALogFifo;
pSysVarAB2BALogFifo = new CLoggingFifo();
bNoError = pSysVarAB2BALogFifo->SetNumber(6) && bNoError;
bNoError = pSysVarAB2BALogFifo->SetSize(160) && bNoError;
bNoError = pSysVarAB2BA->AddComponent(pSysVarAB2BALogFifo) && bNoError;
CLoggingFifo* pSysVarBB2AALogFifo;
pSysVarBB2AALogFifo = new CLoggingFifo();
bNoError = pSysVarBB2AALogFifo->SetNumber(7) && bNoError;
bNoError = pSysVarBB2AALogFifo->SetSize(170) && bNoError;
bNoError = pSysVarBB2AA->AddComponent(pSysVarBB2AALogFifo) && bNoError;
bNoError = pDynamicSystem->ArrangeAll() && bNoError;
FileStream.open("/home/ivan/project/dsl/rtmodule/rtmodule.c");
pDynamicSystem->GenerateSource(FileStream);
FileStream.close();

// Defining, adding and using setting fifos
CSettingFifo* pInputAASetFifo;
pInputAASetFifo = new CSettingFifo();
bNoError = pInputAASetFifo->SetNumber(8) && bNoError;
bNoError = pInputAASetFifo->SetSize(8) && bNoError;
bNoError = pInputAA->AddComponent(pInputAASetFifo) && bNoError;
CSettingFifo* pInputABSetFifo;
pInputABSetFifo = new CSettingFifo();
bNoError = pInputABSetFifo->SetNumber(9) && bNoError;
bNoError = pInputABSetFifo->SetSize(8) && bNoError;
bNoError = pInputAB->AddComponent(pInputABSetFifo) && bNoError;
CSettingFifo* pInputBASetFifo;
pInputBASetFifo = new CSettingFifo();
bNoError = pInputBASetFifo->SetNumber(10) && bNoError;
bNoError = pInputBASetFifo->SetSize(8) && bNoError;
bNoError = pInputBA->AddComponent(pInputBASetFifo) && bNoError;
CSettingFifo* pInputBBSetFifo;
pInputBBSetFifo = new CSettingFifo();
bNoError = pInputBBSetFifo->SetNumber(11) && bNoError;
bNoError = pInputBBSetFifo->SetSize(8) && bNoError;
bNoError = pInputBB->AddComponent(pInputBBSetFifo) && bNoError;
CSettingFifo* pLoopVarASetFifo;
pLoopVarASetFifo = new CSettingFifo();
bNoError = pLoopVarASetFifo->SetNumber(12) && bNoError;
bNoError = pLoopVarASetFifo->SetSize(8) && bNoError;
bNoError = pLoopVarA->AddComponent(pLoopVarASetFifo) && bNoError;
CSettingFifo* pLoopVarBSetFifo;
pLoopVarBSetFifo = new CSettingFifo();
bNoError = pLoopVarBSetFifo->SetNumber(13) && bNoError;
bNoError = pLoopVarBSetFifo->SetSize(8) && bNoError;
bNoError = pLoopVarB->AddComponent(pLoopVarBSetFifo) && bNoError;
```

```
  CSettingFifo* pSysVarAB2BASetFifo;
  pSysVarAB2BASetFifo = new CSettingFifo();
  bNoError = pSysVarAB2BASetFifo->SetNumber(14) && bNoError;
  bNoError = pSysVarAB2BASetFifo->SetSize(8) && bNoError;
  bNoError = pSysVarAB2BA->AddComponent(pSysVarAB2BASetFifo) && bNoError;
  CSettingFifo* pSysVarBB2AASetFifo;
  pSysVarBB2AASetFifo = new CSettingFifo();
  bNoError = pSysVarBB2AASetFifo->SetNumber(15) && bNoError;
  bNoError = pSysVarBB2AASetFifo->SetSize(8) && bNoError;
  bNoError = pSysVarBB2AA->AddComponent(pSysVarBB2AASetFifo) && bNoError;

  // Defining, adding and using reporting fifos
  CReportingFifo* pLoopARepFifo;
  pLoopARepFifo = new CReportingFifo();
  bNoError = pLoopARepFifo->SetNumber(16) && bNoError;
  bNoError = pLoopARepFifo->SetSize(320) && bNoError;
  bNoError = pLoopA->AddComponent(pLoopARepFifo) && bNoError;
  CReportingFifo* pLoopBRepFifo;
  pLoopBRepFifo = new CReportingFifo();
  bNoError = pLoopBRepFifo->SetNumber(17) && bNoError;
  bNoError = pLoopBRepFifo->SetSize(320) && bNoError;
  bNoError = pLoopB->AddComponent(pLoopBRepFifo) && bNoError;
  bNoError = pDynamicSystem->ArrangeAll() && bNoError;
  FileStream.open("/home/ivan/project/dsl/rtmodule/rtmodule.c");
  pDynamicSystem->GenerateSource(FileStream);
  FileStream.close();

  // Defining, adding and using a commanding fifo
  CCommandingFifo* pCommandingFifo;
  pCommandingFifo = new CCommandingFifo();
  bNoError = pCommandingFifo->SetNumber(18) && bNoError;
  bNoError = pCommandingFifo->SetSize(100) && bNoError;
  bNoError = pDynamicSystem->AddComponent(pCommandingFifo) && bNoError;

  // Arranging the dynamic system
  pDynamicSystem->ArrangeAll();

  // Generating the module's code
  ofstream FileStream;
  FileStream.open("/home/ivan/project/dsl/rtmodule/rtmodule.c");
  pDynamicSystem->GenerateSource(FileStream);
  FileStream.close();

  // Clearing the memory
  pDynamicSystem->DeleteObjects();
  delete pDynamicSystem;

  return EXIT_SUCCESS;
}
```

25 DSLib's user API

The following is the list of classes and functions that may be used to define dynamic systems.

25.1 CDynamicSystem

- `bool AddComponent(CCodeGenerator*);`
- `bool ArrangeAll(void* = NULL);`
- `string Class() const;`
- `void GenerateSource(ostream&, void* = NULL) const;`
- `bool SetTime(long long);`

- long long Time() const;

25.2 CLoop

- bool AddComponent(CCodeGenerator*);
- string Class() const;
- bool FloatingPoint() const;
- unsigned int IRQNumber() const;
- const string& Name() const;
- long long Period() const;
- unsigned int Priority() const;
- enum eScheduling Scheduling() const;
- bool SetFloatingPoint(bool);
- bool SetIRQNumber(unsigned int);
- bool SetName(const string&);
- bool SetPeriod(long long);
- bool SetPriority(unsigned int);
- bool SetScheduling(eScheduling);

25.3 CFunction

- bool AddComponent(CCodeGenerator*);
- string Class() const;
- const string& Code() const;
- const string& Name() const;
- bool SetCode(const string&);
- bool SetName(const string&);

25.4 CFunctionInput

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `bool Connect(COutput*);`
- `const COutput* ConnectedOutput() const;`
- `const string& Initialization() const;`
- `const string& Name() const;`
- `bool SetInitialization(const string&);`
- `bool SetName(const string&);`
- `bool SetType(const string&);`
- `const string& Type() const;`

25.5 CFunctionOutput

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Initialization() const;`
- `const string& Name() const;`
- `bool SetInitialization(const string&);`
- `bool SetName(const string&);`
- `bool SetType(const string&);`
- `const string& Type() const;`

25.6 CFunctionVariable

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Name() const;`
- `bool SetName(const string&);`
- `bool SetType(const string&);`
- `const string& Type() const;`

25.7 CLoopVariable

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Initialization() const;`
- `const string& Name() const;`
- `bool SetInitialization(const string&);`
- `bool SetName(const string&);`
- `bool SetType(const string&);`
- `const string& Type() const;`

25.8 CSystemVariable

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Initialization() const;`
- `const string& Name() const;`
- `bool SetInitialization(const string&);`

- `bool SetName(const string&);`
- `bool SetType(const string&);`
- `const string& Type() const;`

25.9 CLoggingFifo

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Name() const;`
- `unsigned int Number() const;`
- `bool SetName(const string&);`
- `bool SetNumber(unsigned int);`
- `bool SetSize(int);`
- `int Size() const;`

25.10 CSettingFifo

- `bool AddComponent(CCodeGenerator*);`
- `string Class() const;`
- `const string& Name() const;`
- `unsigned int Number() const;`
- `bool SetName(const string&);`
- `bool SetNumber(unsigned int);`
- `bool SetSize(int);`
- `int Size() const;`

25.11 CReportingFifo

- bool AddComponent(CCodeGenerator*);
- string Class() const;
- const string& Name() const;
- unsigned int Number() const;
- bool SetName(const string&);
- bool SetNumber(unsigned int);
- bool SetSize(int);
- int Size() const;

25.12 CCommandingFifo

- bool AddComponent(CCodeGenerator*);
- string Class() const;
- const string& Name() const;
- unsigned int Number() const;
- bool SetName(const string&);
- bool SetNumber(unsigned int);
- bool SetSize(int);
- int Size() const;

25.13 CErrors

- typedef vector<Type>::iterator Iterator;
- typedef vector<Type>::const_iterator ConstIterator;
- void Append(const TDerivableVector<Type>&);

- `ConstIterator Begin() const;`
- `Iterator Begin();`
- `void Clear();`
- `bool Empty() const;`
- `ConstIterator End() const;`
- `Iterator End();`
- `void PushBack(Type);`
- `unsigned int Size() const;`
- `Type operator[](unsigned int);`
- `Type operator[](unsigned int) const;`

25.14 CError

- `unsigned int Number() const;`
- `const CCodeGenerator* Location() const;`
- `const string Message() const;`