https://doi.org/10.31449/inf.v43i2.2019 Informatica **43** (2019) 263–279 **263**

A CLR Virtual Machine Based Execution Framework for IEC 61131-3 Applications

Salvatore Cavalieri and Marco Scroppo
University of Catania, Department of Electrical Electronic and Computer Engineering (DIEEI), Italy
E-mail: salvatore.cavalieri@unict.it, marcostefano.scroppo@dieei.unict.it

Keywords: IEC61131-3, PLC, CLR VM, real-time industrial applications

Received: November 13, 2017

The increased need of flexibility of automation systems and the increased capabilities of sensors and actuators paired with more capable bus systems, pave the way for the reallocation of IEC 61131-3 applications away from the field level into so-called compute pools. Such compute pools are decentralised with enough compute power for a large number of applications, while providing the required flexibility to quickly adapt to changes of the applications requirements. The paper proposes a framework able to deploy IEC 61131-3 applications to multiple computing platforms based on CLR VM; it uses C# language as intermediate code. The software solution proposed by the authors does not require any modifications of the IEC 61131-3 applications. Current literature does not provide solutions like that here presented; due to the spread current use of C# language in the development of industrial applications, adoption of the proposed solution seems very attractive. The paper will deeply describe the software implementation and will also present an analysis about the capability of the proposed framework to respect real-time constraints of the industrial processes, mainly focusing on the periodic ones.

Povzetek: Prispevek predlaga okvir, ki omogoča uporabo aplikacij IEC 61131-3 za več računalniških platform, ki temeljijo na CLR VM.

1 Introduction

Programmable Logic Controllers (PLCs) are widely used for the control of automation systems. The standard IEC 61131-3 defines the execution model as well as programming languages for such systems [1]. According to IEC 61131-3, software development becomes independent of process mapping and device specific configuration files. Programmers can focus on the algorithm and control development. Device specific knowledge is outsourced into the block library and can be substituted, every time a new target PLC device should be programmed.

During these last years, the need to deploy IEC 61131-3 – based applications addressing multiple target platforms (also different from PLCs, e.g. based on general purpose computing architectures) became more and more urgent for the reason explained in the following. In a common factory automation scenario, actuators and sensors connect to the PLCs via automation buses; traditionally, bus based systems dominated the automation industry. Nowadays, more powerful and flexible automation networks appear and allow the connection of thousands of actuators and sensors to the same network, while still obtaining the required timing performance; interested readers are referred to [2] for a detailed overview.

Those changes in the communication technologies opens possibilities of computation further away from the field level, compared to how it is done in today's automation systems. On the other hand, many sensors and actuators are equipped with small microcontrollers, allowing them to do basic data processing; furthermore, they are able to connect directly to the new bus technologies.

Having basic data processing done at the lowest level (i.e., at the field level directly on sensors and actuators) and a connection to capable networks, allows the reallocation of applications away from the field level into so-called compute pools [3] [4]. Such compute pools are decentralised with enough compute power for a large number of applications, while providing the required flexibility to quickly adapt to changes of the applications requirements. This has several benefits. Changing control applications becomes merely a problem of reconfiguration in the compute pool. Costs will decrease as well as the need for physical PLCs will be decreased; in this new scenario, the PLC is migrated to the computing pool and can be also realised by general purpose computer architectures (e.g., a server or a cluster of servers).

Several requirements must be satisfied in order to reach this goal. The first one is the guarantee of the total compliance with IEC 61131-3; it is clear that moving an IEC 61131-3 application on a compute pool must be realised without any changes in the same application. Then, respect of real-time constraints of the IEC 61131-3 control applications must occur when migrating the application to the compute pool. Particular cares must be reserved for real-time applications requiring periodic executions; in these cases, executions of each process must occur exactly with the requested period.

Current literature presents several solutions in the direction just pointed out. For example, in [5], the use of the Java Virtual Machine (JVM) to deploy IEC 61131-3 applications to embedded devices has been proposed. In [6] different levels of an automation process are proposed and a cloud-based solution is presented. An example of virtual PLC is given by [7], where PLC systems are executed as applications within a legacy OS. Finally, [3] [4] present the use of a multi-core high performance computing architecture to realise the compute pool.

Based on what said, the aim of the paper is to contribute to find solutions able to deploy IEC 61131-3 – based applications to multiple computing platforms, mainly focusing on general purpose computer systems (e.g., single server or cluster of servers with common operating systems).

In the last years, the domain of factory and process automation features intense usage of languages (e.g., Java, C#) based on Virtual Machines (VMs), like JVM or Common Language Runtime (CLR) VM, as pointed out in [8]. A VM has some clear benefits: portability, security, Just-in-time Compiler to boost performance in time, ease of development in conjunction with a garbage collector, multi-threading and others; reader may refers to [8] in order to achieve a complete survey on this subject. On this basis, the authors believe that one of right possible directions to reach the aim of the paper is that to adopt languages supporting VMs for the deployment of IEC 61131-3 application on common computing platforms. This idea was already pointed out in [5], which proposed the deployment of IEC 61131-3 applications using Java bytecode as a common intermediate format, although the deployment was limited to the embedded devices.

To the best of authors' knowledge, literature does not provide solutions aimed to deploy IEC 61131-3 applications using languages based on CLR VM, like C# language, as intermediate code. Due to the spread current use of C# language in the development of industrial applications, adoption of C# language based on CLR VM to deploy IEC 61131-3 applications on computing platforms seems attractive. Typical candidate platforms are those based on general purpose computing architecture (on which CLR VM allows the use of common operating systems like Linux and Windows), but also all the embedded systems supporting a CLR VM may be considered.

For all the previous reasons, the authors propose a novel software solution made up by different features. First of all, it is able to translate a generic IEC 61131-3 application into C# code which could be executed in a general purpose CLR VM-based platform. Furthermore, the solution here proposed includes the definition of a framework which is able to realise the deployment of IEC 61131-3 applications on a compute poll based on CLR VM, using the C# code as intermediate one. The proposed solution does not require any modifications to the native IEC 61131-3 applications; all additional overhead is handled by the framework here defined. Applications in the automation domain often come with real-time requirements; in order to better allow their respect, the proposed framework features the use of a CLR VM on the top a real-time operating system who is in charge to schedule time-critical applications. Finally, the last feature of the proposed software solution is the use of open source environments; in particular, the implementation presented in the paper is based on the use of MONO [9] as CLR VM and a real-time Xenomai co-kernel [10] alongside a common Linux kernel. Choice of real-time Xenomai co-kernel has been based on a performance evaluation whose main results will be shown in the paper.

The paper will deeply describe the proposed software solution pointing out the main features. Then, results of a performance evaluation aimed to analyse its capability to respect real-time constraints of typical periodic industrial applications will be presented and discussed.

Some of the very preliminary results achieved at the first stages of the research carried out by the authors have been subject of publication [11][12]. This paper presents the full results of the research and gives a very deep analysis of the implementation realised and of the outcomes achieved by the authors.

2 PLC and IEC 61131-3 main features

The main feature of a PLC is the use of cyclic loops for the execution of programs; each loop is called Program Scan. As shown by Figure 1, in each Program Scan, PLC reads the real inputs copying them into an internal memory area called I. Then, PLC execute one or more programs and finally it updates all the output values found in the memory area Q into the real output devices. The program/s executed inside the Program Scan may use internal memory, called area M, for the temporary storage of information. Each program may have a task associated, whose main aim is to control the execution of the program itself; the most common task is the periodic one, triggering the program in such a way it should be iterated after a certain fixed time interval (i.e., the period of the task). Tasks may feature priorities and the execution of a program inside the Program Scan may be pre-empted by another program whose task associated features a higher priority. Generally, reading and writing operations shown by Figure 1, cannot be interrupted by other programs.

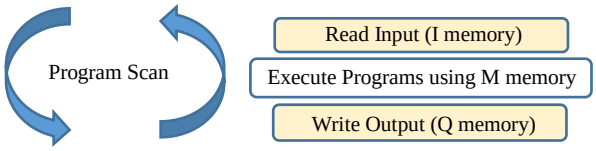

Figure 1: Program Scan.

IEC 61131-3 is the vendor independent standardised programming language for factory automation [1]. IEC 61131-3 allows users to write programs, choosing among five programming languages: Ladder Diagram (LD), Sequential Function Charts (SFC), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL).

IEC 61131-3 software development is independent of process mapping and device specific configuration files. IEC 61131-3 application is typically deployed on a PLC

device, whose specific knowledge is outsourced into the block library and can be substituted, every time a new target PLC device should be programmed.

In order to make the program itself independent on the device on which it must be deployed, the structure of an application written according to IEC 61131-3 standard is made up by at least two separate sections: Program and Configuration.

An IEC61131-3 Program provides a large re-usable software component. It is defined by a program type definition (starting with PROGRAM and ending with END_PROGRAM keywords) which has input, output and internal variables declaration and a body which contains software describing the behaviour of the program itself. As said before, one of the five languages specified above can be used to describe the program.

A Configuration defines the software for a complete PLC and will always include at least one but, in some cases, many Resources. A Configuration is specific to a particular type of PLC product and the arrangement of the relevant PLC hardware. It can be used to create software for another PLC if the hardware is identical. Configuration is introduced with the keyword CONFIGURATION and terminate with END_CONFIGURATION keyword.

A Resource describes a processing facility inside a PLC type that is able to execute an IEC 61131-3 Program. A Resource is defined within the Configuration using the keyword RESOURCE followed by an identifier and the type of the processor on which the Resource will be loaded (keyword ON is used before the type of processor). In the real cases, for each type of PLC a detailed description of the hardware and software features is associated (e.g., firmware version, number of inputs/outputs, internal memory). The resource definition contains a list of Global Variable declarations and task definitions that can be assigned to Programs. It terminates with the keyword END_RESOURCE.

As said, a task may be associated to a Program controlling its execution. Task may be single or periodic; in this last case, a period is specified for its execution. A priority value is assigned to each task in order to determine the order of their executions. Tasks are defined inside the RESOURCE section, as said. A Task declaration is introduced using the keyword TASK followed by the task identifier and optional values for the following parameters: SINGLE (if the task is not periodic), INTERVAL (period, if the task is periodic), PRIORITY (task priority value). After its definition, Task is associated to an instance of a Program using the keywords WITH.

Figure 2 shows a very simple IEC 61131-3 application using ST language; this same example will be used in the remainder of this paper.

As it can be seen, the simple IEC 61131-3 application is made up by only one PROGRAM section called MyProgram, which contains the definition of the local (i.e., VAR) and external (or global, i.e., VAR_EXTERNAL) variables. Furthermore, it contains the algorithm coded into ST language; it is made up by only two assignments, the first is relevant to a global variable (StepSizeVar) and the other to the local variable ComputedResult. CONFIGURATION section is named MyConfiguration and is made up by only one resource called MyResource; the type of PLC chosen for the execution of the software has been called PLC1 in the example. The RESOURCE section contains the declaration of the global variables used by the program (StepSizeVar, MaxValueVar and MinValueVar); the declaration includes the mapping of these variables into the internal PLC memory (called M memory, as shown by Figure 1) at the addresses 200, 204 and 208, respectively. RESOURCE section also contains the definition of two periodic Tasks, named MyTask1 and MyTask2; they differ for the period and the priority values. According to IEC 61131-3 standard, low priority values refer to high priority tasks.

```
PROGRAM MyProgram
  VAR
    ComputedResult : REAL;
  END_VAR
  VAR_EXTERNAL
    StepSizeVar : REAL;
    MaxValueVar : REAL;
    MinValueVar : REAL;
  END_VAR

  StepSizeVar := MaxValueVar-MinValueVar;
  ComputedResult := StepSizeVar;
END_PROGRAM

CONFIGURATION MyConfiguration
  RESOURCE MyResource ON PLC1
    VAR_GLOBAL
      StepSizeVar AT  %MD200 : REAL;
      MaxValueVar AT  %MD204 : REAL;
      MinValueVar AT  %MD208 : REAL;
    END_VAR
    TASK MyTask1(INTERVAL := T#100ms, PRIORITY := 1);
    TASK MyTask2(INTERVAL := T#150ms, PRIORITY := 2);
    PROGRAM MyInstance1 WITH MyTask1: MyProgram;
    PROGRAM MyInstance2 WITH MyTask2: MyProgram;
  END_RESOURCE
END_CONFIGURATION
```

Figure 2: IEC 61131-3 ST-based Program relevant to a simple algorithm.

Finally, two instances of the MyProgram Program are defined into the RESOURCE section; they are called MyInstance1 and MyInstance2 and are featured by the tasks MyTask1 and MyTask2 associated, respectively, controlling their execution.

3 Overview of Xenomai

The Xenomai project has the aim of providing real-time support for user applications [10].

It is a real-time development framework that cooperate with the Linux kernel in order to make possible the real-time management of tasks on any hardware with a Linux-based operating system. The project has a strong focus on embedded systems, although Xenomai can also be used over common desktop and server architectures. Xenomai has two modes of use:

- as co-kernel extension for a patched version of the original Linux kernel. This is the solution adopted in the paper.
- as libraries for native Linux kernel (features added in the version 3.0 in 2015)

In both modes, it is possible to use the Native Xenomai C language-based API functions to run real-time tasks [13].

To create and run a simple real-time task, three steps are needed:

1. Creation of the task and setting of its properties (e.g., priority) using the *rt_create_task()* API function. If the task is periodic, the *rt_task_set_periodic()* API function will be also used, in order to allow Xenomai to have knowledge of the task periodicity.
2. Creation of a C language-based procedure that the task will perform during its execution. If the task is not-periodic there are not particular constraints for the structure of this procedure. But, for periodic task, the C-language-based procedure must be featured by an infinite while loop inside which the *rt_task_wait_period()* API function must be present; it allows the procedure to be stopped after its conclusion and to resume its regular running after the task period previously set by *rt_task_set_periodic()* API function. Figure 3 shows how the C language-based procedure (called *task_function()* in the figure), must be written in the case of periodic task.
3. Association of the C language-based procedure to the Xenomai task created at the step 1 and starting the task using the *rt_task_start()* API function; in particular, the entry point of the C language-based procedure is passed to this function.

```
void task_function(){
  while (true) {
    //Code in C Language to be executed
    rt_task_wait_period();
  }
}
```

Figure 3: Structure of a C language-based procedure to be assigned to a periodic task.

4 Running C# programs over Xenomai

This section plays a strategic role inside the paper. Introduction pointed out that the aim of the paper is that to propose a framework able to translate an IEC61131-3 application into C# program; furthermore, the framework is able to allow real-time execution of the C# program using a Xenomai co-kernel.

Before the framework defined may be presented, this section has to point out how a C# program may be executed over a Xenomai real-time co-kernel and, most important, if execution of a C# program may actually exploit the real-time features of the Xenomai co-kernel.

The software solution presented in Figure 4 has been defined to allow execution of a C# program over Xenomai co-kernel. It is based on the use of a MONO Virtual Machine running on the top of a Linux OS with Xenomai co-kernel [9].

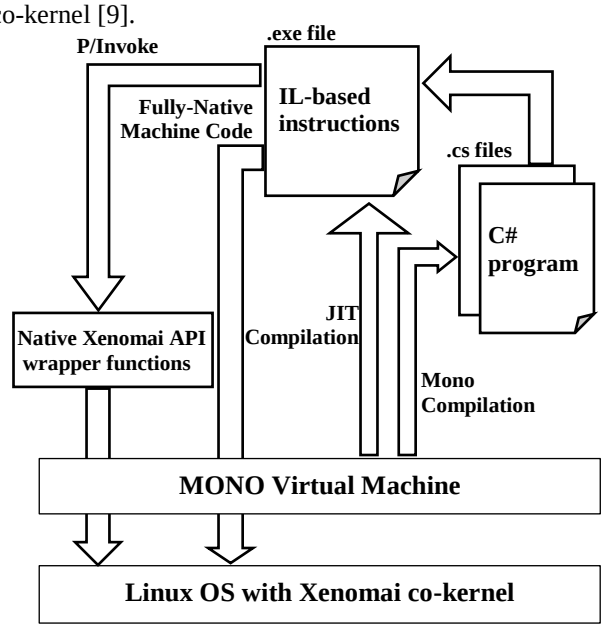

Figure 4: Software solution adopted for the Xenomai-based C# program execution.

As said in the previous section, Xenomai offers a set of native API functions to realise real-time mechanisms [13]; these API functions are callable inside a program written in C language. In order to allow a C# language-based program to call a particular real-time Xenomai API function, suitable wrapper functions had to be defined. Each wrapper function maps a C# function call to a particular Xenomai API function; this happens through the definition of a C function containing the call to the native Xenomai API. All the wrapper functions are pre-compiled and realise a run-time library named in the figure "Native Xenomai API wrapper functions". One of the following subsections will give an overview of the wrapper functions defined in the research here present.

C# programs (written inside .cs files) are compiled by Mono producing .exe files containing Intermediate Language (IL)-based instructions. At run-time, for each IL-based executable file, Just-In-Time (JIT) compilation is realised producing binary code. Native machine code is executed directly by Linux/Xenomai kernel. In order to execute the Native Xenomai API wrapper functions, P/Invoke procedure allows to call the unmanaged code produced by the compilation of the "Native Xenomai API wrapper functions". The unmanaged code is mapped on the Xenomai real-time system calls as the wrapper functions contains the calls to Xenomai API, as said before.

4.1 Native Xenomai API wrapper functions

The Xenomai wrapper functions defined according to the goal of the research here presented, are detailed in the following.

rt_task_create_wrap(). It calls the native Xenomai API *rt_task_create()* belonging to the Task Management Services. This service creates a new real-time task which is left in an innocuous state until it is actually started by the Xenomai service *rt_task_start()*. Among the parameters passed to the native Xenomai API function, the wrapper function specifies the priority of the new task in the range from [0 .. 99], where 0 is the lowest effective priority.

rt_task_set_periodic_wrap(). This wrapper function calls the native Xenomai API *rt_task_set_periodic()*, which makes a real-time task periodic, by programing its first release point and its period in the processor time line.

rt_task_start_wrap(). It allows to start execution of a Xenomai task that has been previously created. This wrapper calls the native Xenomai API *rt_task_start()*, which releases the target task from the dormant state. Among parameters passed to the native Xenomai API *rt_task_start()*, the wrapper function specifies the address of the procedure to be execute when the task is running.

rt_task_wait_period_wrap(). It makes the Xenomai task wait for the next periodic release point in the processor time line. A rescheduling of the task always occurs, unless the current release point has already been reached. In the latter case, the current task immediately returns from this service without being delayed.

rt_task_sleep_wrap(). It suspends the calling process for a certain amount of milliseconds passed as argument. This function calls the Xenomai *rt_task_sleep()* API function.

rt_sem_create_wrap(). It allows to create a Xenomai real-time semaphore, fully handled by Xenomai itself. It wraps the Native Xenomai API *rt_sem_create()*.

rt_sem_p_wrap(). It is used to acquire the semaphore or put on hold his release if already occupied. It is directly mapped to the native Xenomai API *rt_sem_p()*, which acquires a semaphore unit. If the semaphore value is greater than zero, it is decremented by one and the service immediately returns to the caller. Otherwise, the caller is blocked until the semaphore is either signalled or destroyed, unless a non-blocking operation has been required. Among the parameters passed to the native API function, there is the descriptor address of the affected semaphore.

rt_sem_delete_wrap(). It directly maps to the Xenomai API *rt_sem_delete()*, which destroys a semaphore and release all the tasks currently pending on it.

rt_sem_v_wrap(). This function allows to call the native Xenomai API *rt_sem_v()* inside a C# program. This service releases a semaphore unit; the parameters passed to the native Xenomai API function, specify the descriptor address of the affected semaphore.

4.2 Analysis of the real-time capabilities

Evaluation of the capability of the software solution shown by Figure 4 to respect real-time constraints of a generic C# program was considered of primary importance. Real-time feature has been evaluated observing the capability of a particular C# program to promptly react to a rising event; real-time capabilities have been measured checking that all rising events have been caught with the lowest delay.

The analysis has been carried out on an embedded system. Choice of an embedded system compared with a general purpose computing device like a server or a personal computer, had the advantage to allow an easier use of an oscilloscope to analyse the output produced upon the occurrence of an event realised by a digital input.

The embedded system is made up by a MPC8309 PowerQUICC processor [14] running at 333Mhz with 256MB RAM, a microcontroller PIC32MX, and two Serial Peripheral Interface (SPI) acquisition boards (each featuring 4 channels at 16 bit, sampling at 125 μs).

Figure 5 shows the general architecture of the embedded system. The PIC32MX receives the samples from SPI, forwarding them to MPC8309 through the MISO (Master data In/Slave data Out) bus. The SYNC line is used by MPC8309 processor to advise the PIC32MX that it is ready to start the acquisition of samples. After reception of this synchronization signal, the PIC32MX will start transmission of samples received from SPI, synchronizing them with a DRDY signal with a duration of 15 μs, sent with a period of 5ms.

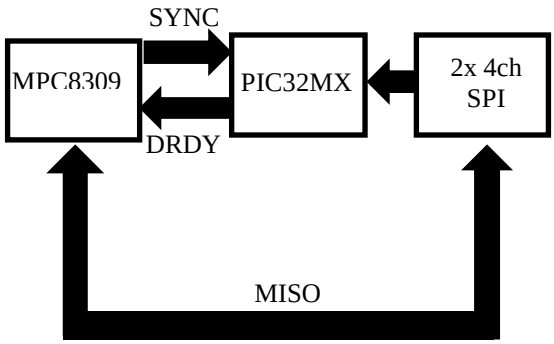

Figure 5: Architecture of the Embedded System.

A Linux Kernel 3.8.13 with co-kernel Xenomai version 2.6.4 has been installed in the MPC8309 embedded system. A Mono framework version 3.2.6 has been also installed.

A huge set of tests has been performed in order to explore the capability featured by the Xenomai-based software solution shown by Figure 4 to meet real-time constrains of a C# program running inside the MPC8309. A C# program realising the flow-chart described by Figure 6 has been defined. It reacts to the DRDY activation; on the receipt of this signal, the C# program set a particular General Purpose I/O, the GPIO #1, and maintains the value ON for 1 ms; this is achieved using the rt_task_sleep_wrap() describe before. After the sleep interval has passed, the GPIO #1 is put OFF. It is important to recall that DRDY is activated each 5 ms, as said at the beginning of this section.

Two other C# programs have been defined; they both realise the flow chart shown by Figure 7. Each program waits for the setting of the GPIO #1 (by the program shown by Figure 6); when this occurs, the GPIO #2 is set and suddenly reset. Then, each program calls a sleep function with a duration of 2 ms; one C# program realises

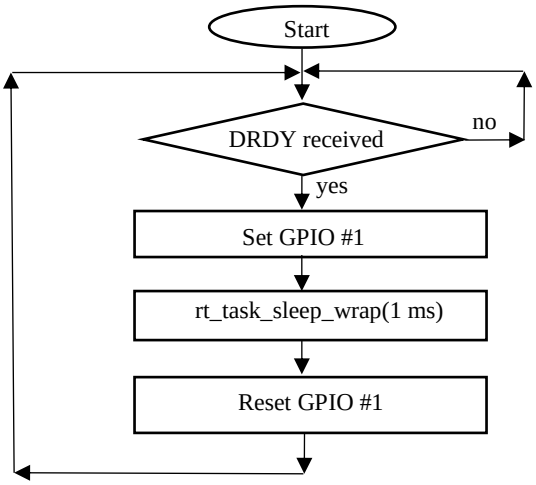

Figure 6: C# language-based program acting on receipt of DRDY and setting GPIO #1.

this call though the function rt_task_sleep_wrap(), whilst the other C# program uses the C# Thread.sleep(). The only difference between the two C# programs is that the first one foresees the real-time management of the sleep by Xenomai co-kernel, whilst the other one does not exploit the real-time features of Xenomai co-kernel, as the management of the sleep of the process is given to Linux OS.

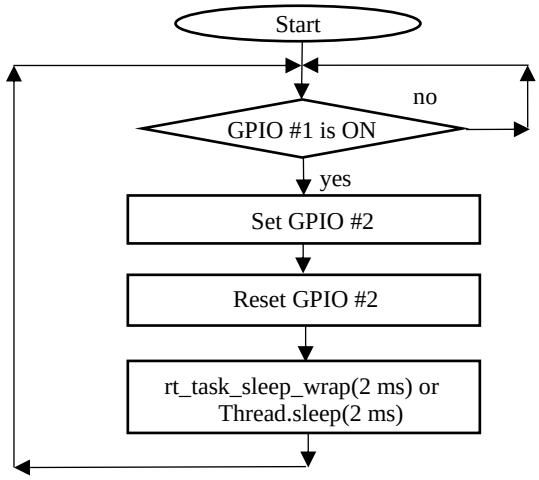

Figure 7: C# language-based programs acting on receipt of GPIO #1 signal.

Figure 8 points out the behaviour of the C# program described by Figure 7 using the C# Thread.sleep(). The signal number 1 (on the top) refers to the setting of the GPIO #1 done by the C# program shown by Figure 6; it easy to verify that the period of this signal is 5ms as it is synchronised with DRDY. Signal number 2 (on the bottom) refers to the GPIO #2 and is set/reset by the C# program shown by Figure 7 when C# Thread.sleep() is used.

Figure 9 refers to the C# program described by Figure 7 when rt_task_sleep_wrap() is used. Again, the signal number 1 (on the top) refers to the setting of the GPIO #1 done by the first C# program shown by Figure 6. Signal number 2 (on the bottom) refers to the GPIO #2 and is

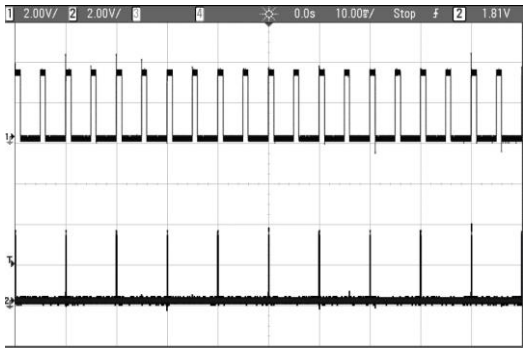

Figure 8: Performance achieved using C# Thread.sleep().

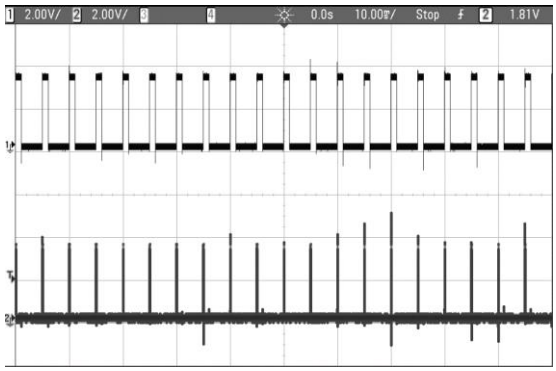

Figure 9: Performance using rt_task_sleep_wrap().

set/reset by the C# program shown by Figure 7 when rt_task_sleep_wrap() is used.

Comparison of the two Figures 8 and 9 points out that the C# Thread.sleep is not able to wake-up the process in time to catch each single setting/resetting of the GPIO #2. The use of Xenomai API allows total respect of real-time requirements here presented.

A huge set of other tests not shown here for space limitation, allowed to reach the same conclusions just pointed out: use of Native API Xenomai wrapper functions here defined according to the software solution shown by Figure 4, allows to fully exploit the real-time capabilities offered by Xenomai co-kernel and allows the respect of time-critical constraints. For these reasons, the software solution presented in Figure 4 will be used in the remainder of this paper.

5 Overview of the proposed framework

As pointed out in the Introduction, the main aim of this paper is that to present a framework based on the use of CLR-based virtual machine, able to deploy an IEC 61131-3 application on a computing system supporting CLR VM. The framework is made up by the two modules shown in Figure 10 with the grey coloured backgrounds: *C# Translator* and *PLC Framework*.

C# Translator is in charge to process a generic IEC 61131-3 application in order to produce C# language-based classes (contained in .cs files). These classes include all the information relevant to the different sections of the IEC 61131-3 application (e.g., program, configuration,

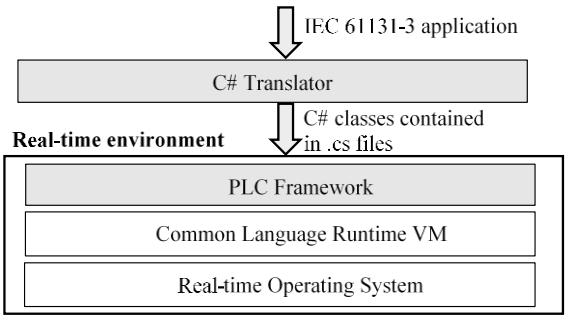

Figure 10: Architecture of the Framework proposed.

resource, including task definitions with periods and priorities). These classes will be used by the real-time environment shown by Figure 10, as explained in the remainder of this section.

A very important feature to be pointed out is that the C# Translator may be used stand-alone, i.e. not linked to the real-time environment shown by Figure 10. C# Translator allows to achieve a C# code that in principle may be executed by a common CLR-based platform, using a common C# language-based integrated development environment (IDE).

Introduction pointed out a main trend in automation, relevant to the reallocation of applications away from the field level into so-called compute pools. Real-time environment shown by Figure 10 is strictly linked to the concept of compute tools, as it will be shown better later. About C# Translator, no constraints exist for its installation; it may be installed in a compute poll or may be installed close to the system where the IEC 61131-3 development environment is running.

PLC Framework has the aim to realise a real-time environment implementing the same behaviour of a PLC. Mainly it allows the realisation of the Program Scan loop execution and allows the real-time scheduling of the tasks associated to the IEC 61131-3 Programs and the relevant executions.

On the basis of what said in the Introduction, and on the basis of the software solution shown by Figure 4, implementation of *PLC Framework* requires the presence of a CLR-based VM running on a real-time operating system, as shown by Figure 10. The CLR VM has been realised by MONO [9]. For the real-time operating system it has been assumed to adopt a Linux OS and a Xenomai co-kernel [10]; choice of Xenomai has been supported by the analysis of the relevant performances, shown in the previous section.

The remainder of this section will focus on the main technical details of the architecture shown by Figure 10.

5.1 C# translator

Given an IEC 61131-3 application, the main aim of the *C# Translator* module is to create a .cs file containing classes that can be used to execute a C# program which exactly behaves as the original IEC application. In the following, a detailed description of the procedure adopted by the *C# Translator* to map an IEC 61131-3 application to the C# classes, will be given.

Figure 11 shows the structure of a typical .cs file produced by *C# Translator*.

```
class ProgramName {
      public ProgramName(){}
      public void IECRoutine() {
      }
}

class ConfigurationName {
   class ResourceName {
         class GlobalDeclaration{
         }
         class TaskName {
               ProgramName instanceName;
               public TaskName(){
                  instanceName = new ProgramName();
               }
         }
   }
}

class ExternalProgramName {
}
```

Figure 11: File structure produced by *C# Translator*.

The .cs file produced contains three main classes: ProgramName, ConfigurationName and ExternalProgramName.

The class ProgramName is the translation of the PROGRAM section in the IEC61131-3 application; it mainly includes a method called IECRoutine() that translates the software describing the IEC61131-3 Program.

The class ConfigurationName is relevant to the CONFIGURATION section in the IEC61131-3 application and is made up by the class ResourceName, related to the RESOURCE section. Class ResourceName may include the declaration of Global Variables; in this case the class GlobalDeclaration is present. The other class contained in ResourceName, class TaskName, represents a single task (so many classes may be present if several IEC tasks have been defined); inside this class, the program to be associated with the task is specified.

Finally, class ExternalprogramName is a singleton class needed for the usage of the External Variables defined in the IEC61131-3 Program and declared as variables of the GlobalDeclaration class. This class has to contain a single instance of the GlobalDeclaration class, that IECRoutine can use so sharing the variables among all its instances. Furthermore, the class must implement suitable mechanisms able to guarantee that concurrent access to the shared variables must occur through use of critical section.

In order to clarify better the structure of the .cs file produced by the C# Translator, the IEC 61131-3 application shown by Figure 2 will be considered. After the processing operated by C# Translator, the .cs file shown by Figure 12 is achieved.

As it can be seen, the .cs file produced contains the three main classes: MyProgram, MyConfiguration and ExternalMyProgram.

The class MyProgram is the translation of the PROGRAM MyProgram section shown by Figure 2; it contains the local variable of the program, declared as

variables of the class (i.e., ComputedResult), and the method IECRoutine() that translates the algorithm of the PROGRAM MyProgram. Just like the IEC 61131-3 application, the C# algorithm uses global variables, as it will be explained in the following.

The class MyConfiguration refers to the IEC 61131-3 CONFIGURATION MyConfiguration section. As it occurs for the IEC 61131-3 application, it is made up by the class MyResource related to the RESOURCE section.

The class MyResource contains three classes: GlobalDeclaration, MyTask1 and MyTask2. Class GlobalDeclaration allows the definition of the global variables of the original IEC61131-3 application (i.e., StepSizeVar, MaxValueVar and MinValueVar). The classes MyTask1 and MyTask2 represent the IEC 61131-3 Tasks: inside these classes there are the variables that indicate the properties of the tasks as period and priority, a variable that indicates the program associated with the task (MyProgram) and a constructor with the aim of initializing these variables.

The class ExternalMyProgram contains a variable of type GlobalDeclaration (i.e., Gd), representing the External Variable of IEC61131-3 Program. This variable is used by the IECRoutine(), as shown by Figure 12. The ExternalMyProgram class implements a critical section using the lock mechanism, allowing a safe and unique instantiation of the single class instance and the safe concurrent access to it.

Again, it is important to point out that the classes produced by C# Translator could be directly used on a common CLR-based platform. They only requires a C# program using them; for example, execution of the IECRoutine() may be achieved through one or more instances of the class MyProgram.

In principle, implementation of *C# Translator* can be realised using whatever technology and language. In this work, the authors chosen to implement this module in C# as a CLR VM-based application. The implementation has been based on the use of the GOLD Parser system [15], by means of which parse tables have been created. In particular, its Builder component has been used to read a source grammar written in the GOLD Meta-Language and to produce the parse tables needed by the *C# Translator*.

5.2 PLC framework

During the design phase, it has been assumed that the *PLC Framework* had to comply with the following assumptions.

For each IEC 61131-3 periodic task, a Xenomai real-time task is created with the same period; priority value of the original IEC 61131-3 task is converted into the range 1 to 99, where 99 is given to the IEC 61131-3 tasks with the highest priority. It is important to recall that task priorities in Xenomai ranges from 0 to 99; as explained in the following, the value 0 has been reserved for a special purpose.

The native scheduling mechanism based on pre-emption adopted by Xenomai has been left unchanged; this means that execution of a Xenomai task is suspended when a higher priority Xenomai task has to be performed.

```
class MyProgram {
    double ComputedResult;
    public MyProgram(){}
    public void IECRoutine() {
      ExternalMyProgram.Gd.StepSizeVar=
                ExternalMyProgram.Gd.MaxValueVar-
                ExternalMyProgram.Gd.MinValueVar;
      ComputedResult = ExternalMyProgram.Gd.StepSizeVar;
    }
}

class MyConfiguration {
  class MyResource {
      class GlobalDeclaration{
          double StepSizeVar;
          double MaxValueVar;
          double MinValueVar;
      }
      class MyTask1 {
          double period;
          int priority;
          MyProgram MyInstance1;
          public MyTask1(){
            MyInstance1 = new MyProgram();
            period=100;
            priority=1;
          }
      }
      class MyTask2 {
          double period;
          int priority;
          MyProgram MyInstance2;
          public MyTask2(){
            MyInstance2 = new MyProgram();
            period=150;
            priority=2;
          }
      }
  }
}

class ExternalMyProgram {
    private static ExternalMyProgram instance = null;
    private static readonly object padlock = new object();
    private GlobalDeclaration Gd = new GlobalDeclaration();
    ExternalMyProgram() {}
    public static ExternalMyProgram Instance {
      get {
        lock (padlock) {
          if (instance == null) {
            instance = new ExternalMyProgram();
          }
          return instance;
        }
      }
    }
```

Figure 12: .cs file produced by *C# Translator* on the IEC 61131-3 application of Figure 2.

Program Scan loop has been realised through a Xenomai real-time task with the lowest priority, i.e. 0. This means that all the other tasks (to which priority values ranging from 1 to 99 have been assigned) can act pre-emption on the Program Scan task. This choice has been made in order to allow execution of a very urgent task, delaying the program scan loop.

The reading and writing operations shown by Figure 1, have been implemented in atomic way, in the sense that during their execution they cannot be interrupted by no other tasks. In order to make atomic the Program Scan execution, a semaphore-based mechanism has been

implemented. In particular, a Xenomai semaphore is created and locked when the Xenomai task implementing the Program Scan is started. The semaphore is deleted each time the Xenomai Program Scan task ends. All the other tasks, even if featuring higher priority cannot interrupt the Program Scan task if the Xenomai semaphore is locked.

Association of a C# program to a periodic Xenomai task has been realised according to the procedure described in Section 3. In particular, for each Xenomai task an instance of the task_function() method shown by Figure 13 is associated through the *rt_task_start_wrap()*.

```
class name {
// local variables
    void task_function () {
// local variables

        while (true) {
// local variables
            ScanProgram() or IEC61131Program();
            rt_task_wait_period_wrap (null);
        }
    }
}
```

Figure 13: task_function() method whose instance is associated to each Xenomai task.

Each task_function() method is made up by a while(true) cycle, inside which a particular C# code is defined; it may be the *ScanProgram()* or the *IEC61131Program()*, both described in the following. Then, the call to the wrapper function *rt_task_wait_period_wrap()* described before in this paper, is achieved. As said before, it forces the Xenomai task associated to the instance of task_function() to wait for the next periodic release point in the processor time line. Figure 13 points out that the local variables (if present) used by the *ScanProgram()* or by *IEC61131Program()*, could be defined at the three different scopes shown by the same figure; the definition of the proper scope will be discussed later in this paper. *ScanProgram()* is a C# program in charge to emulate the typical Program Scan of a PLC. It is shown by Figure 14, by means of a flow-chart graphical representation.

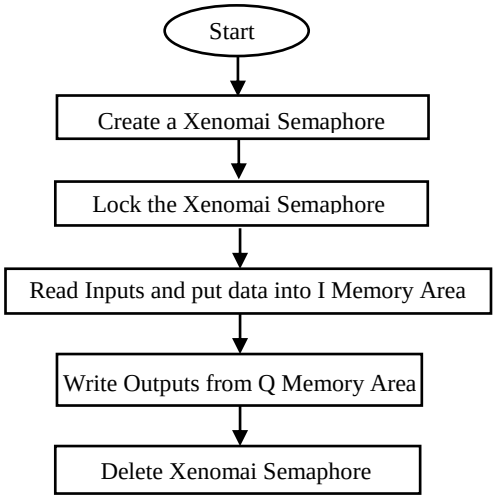

Figure 14: *ScanProgram()*.

At the beginning of the *ScanProgram()*, a semaphore is created by *rt_sem_create_wrap()*. Then, the C# program calls the *rt_sem_p_wrap()* in order to lock it. In this way, the *ScanProgram()* cannot be interrupted from this moment on; reading and writing operations are executed without pre-emption. Reading operations involve I memory, whilst writing operations are relevant to the Q memory (see Figure 1). When they are completed, the semaphore is deleted, by *rt_sem_delete_wrap()*; the relevant waiting queue on the same semaphore is deleted, so all the other task pending on it are released. These tasks may be scheduled for their execution by Xenomai co-kernel.

Figure 15 shows the algorithm implemented inside the *IEC61131Program()*. As it can be seen, it executes a program called IECRoutine(); during the description of the *C# Translator* module, it has been said that among the classes produced for each IEC 61131-3 application there is the class named ProgramName. As shown in Figure 11, this class contains a public void IECRoutine(). The program IECRoutine() executed inside *IEC61131Program* is made up by the same C# code extracted by the public void IECRoutine() contained in the class ProgramName. In the following, the extraction operation performed by the *PLC Framework* will be pointed out.

At the beginning, the *IEC61131Program()* checks the existence of Xenomai semaphore (by using the *rt_sem_p_wrap()*). As said before, this semaphore is created by the *ScanProgram()* and is deleted by the same routine when no more needed. If the *IEC61131Program()* does not find the semaphore, it runs the IECRoutine().

If semaphore exists, the *IEC61131Program()* must check if it is locked (e.g., by *ScanProgram()*). The check is again done using the *rt_sem_p_wrap()*, which locks the semaphore if it is found unlocked; this happens for example when the semaphore has been created by *ScanProgram()* but it was not already locked by it. In this case, the *IEC61131Program()* must suddenly unlock the semaphore (by using *rt_sem_v_wrap()*). This is needed as this task may be pre-empted by a higher priority task which must find the semaphore unlocked, otherwise it cannot be executed. Once the semaphore is unlocked, the

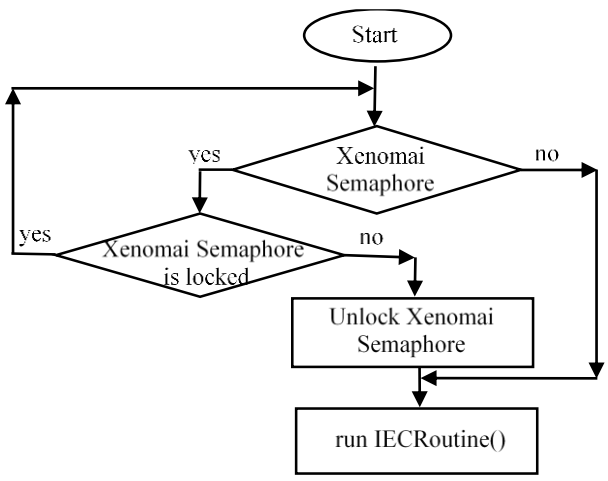

Figure 15: *IEC61131Program()*.

IEC61131Program() executes the IECRoutine(). Figure 15 points out that if the *IEC61131Program()* finds the semaphore locked, it will wait until it is deleted (by *ScanProgram*) or it is unlocked (e.g., by another Xenomai task).

On the basis of what said until know, for each IEC61131-3 application received by the *C# Translator* (in terms of C# classes contained by the .cs files already described), the *PLC Framework* has to performs two main activities.

The first is to produce the task_function() methods shown by Figure 13. One and only one method must contain the *ScanProgram()*, shown by Figure 14; each of the other task_function() methods must contain a *IEC61131Program()*, described by Figure 15.

The second important activity performed by *PLC Framework* is to associate a Xenomai task to each instance of task_function() method, and then activate them so they can be executed by Xenomai co-kernel.

These two activities are performed by two main modules running inside the *PLC Framework*: *ProgramsCreator* and *TasksCreator*. *ProgramsCreator* is in charge to produce the task_function() method on the basis of the C# classes received from the *C# Translator*. *TasksCreator* creates and activates the Xenomai tasks associated to these methods. Figure 16 shows them.

Figure 17 gives an overview of the main activities carried out by the *ProgramsCreator* on the reception of a .cs files produced by *C# Translator*. It analyses class ProgramName (see Figure 11) here contained. From this class, it extracts the class variables and the C# code contained in the IECRoutine() method; this method is placed into the *IEC61131Program()* as shown by Figure 15. Finally, the *ProgramsCreator* creates the class containing the task_function() method shown by Figure 13, placing the *IEC61131Program()* and placing the variables extracted as said before in one of the scopes shown by the same figure. Choice of the right scope will be discussed later in this paper, as said before. When no more ProgramName classes received from *C# Translator* are present, the *ProgramsCreator* will produce the class containing the task_function() method with the *ScanProgram()*, as shown by Figure 13.

The *TasksCreator* module has the main goal to create and run Xenomai tasks, starting from the task_function()

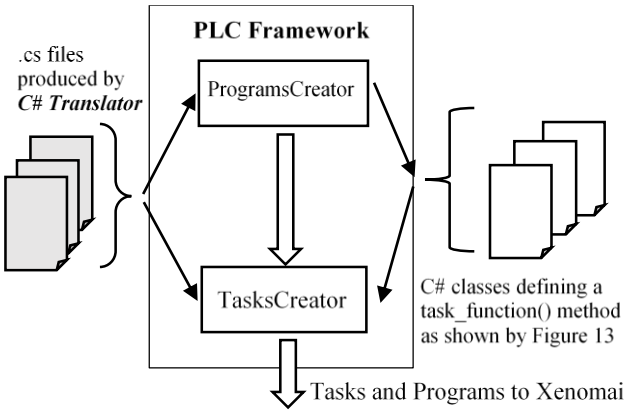

Figure 16: PLC Framework main components.

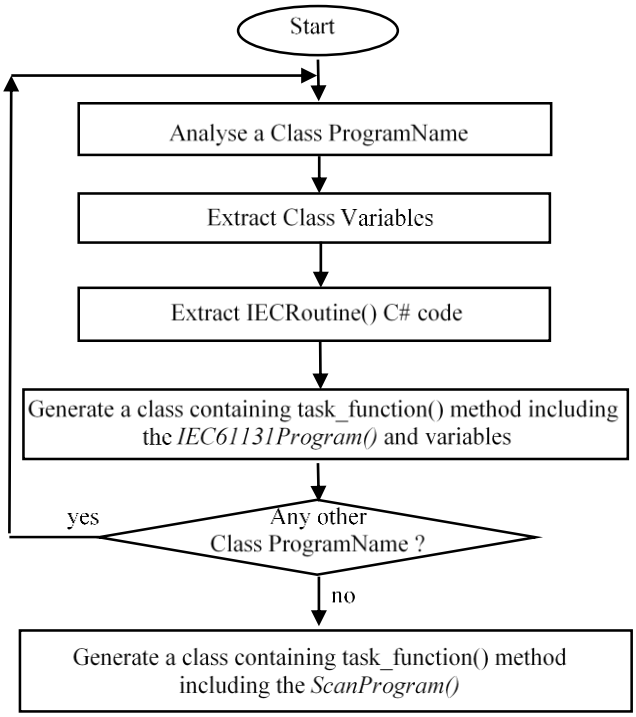

Figure 17: ProgramsCreator.

methods created by the *ProgramsCreator* module. Figure 18 shows the details of the algorithm implemented by this module.

At the beginning, the *TasksCreator* analyses each .cs file received from *C# Translator*. In particular, it focuses on the classes TaskName shown by Figure 11, extracting information (i.e., period and priority) of the entire set of IEC 61131-3 tasks. For each IEC 61131-3 task, a Xenomai real-time task is created through the *rt_task_create_wrap()*; a priority (ranging from 1 to 99) is specified for the Xenomai task to be created. As said

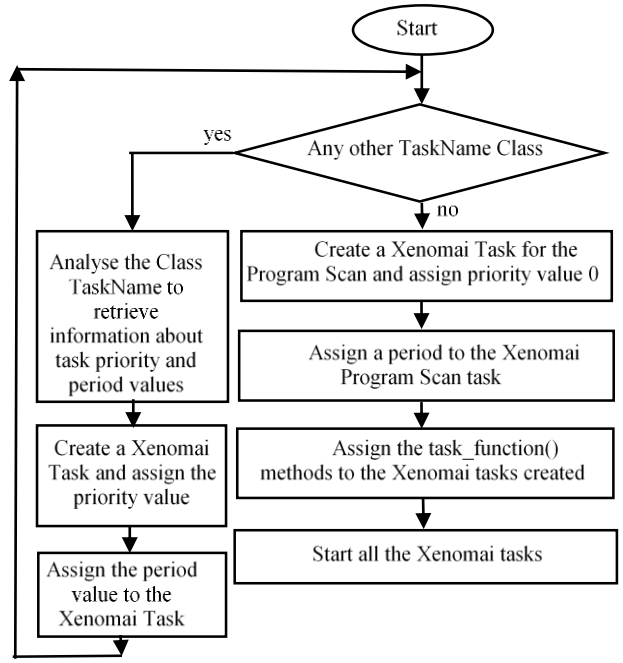

Figure 18: TasksCreator.

before, priority value is assigned according to the priority of the IEC 61131-3 task, mapping the highest priority IEC 61131-3 task with the highest Xenomai priority value (e.g., 99). Once a Xenomai task with a certain priority has been created, the same period of the relevant IEC 61131-3 task is assigned; this is done by the *rt_task_set_periodic_wrap()*.

When no more TaskName classes are present, a Xenomai task must be created in order to be subsequently associated to the task_function() method including the *ScanProgram()*. On the basis of the hypotheses explained in this section, the Xenomai task is created by using *rt_task_create_wrap()*, specifying the priority value 0 (the lowest Xenomai priority value). A period is assigned to this task according to the user settings; this is the period of the Program Scan loop the user has to apply. Period assignment is realised using again the *rt_task_set_periodic_wrap()*.

Finally all the previous Xenomai tasks are started using the *rt_task_start_wrap()*. This wrapper function calls the native Xenomai API *rt_task_start()*, passing the address of the instance of the task_function() method, created by the ProgramsCreator, to be associated to each Xenomai task with priority ranging from 1 to 99. Address of the instance of the task_function() method containing the *ScanProgram()*, is passed to the Xenomai task with priority 0.

In order to better understand the main activities performed by the *ProgramsCreator* and *TasksCreators* modules, let us consider the .cs file shown by Figure 12.

ProgramsCreator will produce the C# code shown by Figure 19; it includes the IEC61131Program() which in turns is made up by the IECRoutine() given by Figure 20.

As shown by Figure 19, the ExternalMyProgram class contained in the .cs file of Figure 12 is imported. The ExternalMyProgram class is made up by the instance Gd of GlobalDeclaration class inside which the external variables of the IEC 61131-3 MyProgram program (StepSizeVar, MaxValueVar, MinValueVar) are defined. Figure 20 points out that IECRoutine() accesses these external variables through the instance Gd contained in the ExternalMyProgram class.

```
import ExternalMyProgram;

class MyProgram {
  //double ComputedResult;
      void task_function () {
      //double ComputedResult;
        while (true) {
          //double ComputedResult;
          IEC61131Program();
          rt_task_wait_period_wrap (null);
        }
      }
}
```

Figure 19: C# code produced by ProgramsCreator.

The *TaskCreator* module extracts from the MyTask1 and MyTask2 classes produced by the *C# translator* the information about Xenomai tasks to be created; this information is about periodicity, priority and about the Program to be associated. On the basis of the information contained in the .cs file shown by Figure 12, it is clear that two Xenomai tasks must be created. Their period values are 100ms and 150ms, respectively; priority values of the original IEC 61131-3 tasks are 1 and 2, which could be mapped to Xenomai priority values 99 and 98, respectively, as priority 1 is the highest priority value according to IEC 61131-3 and must be mapped to the highest Xenomai priority value (i.e., 99). Finally, according to the information contained in the MyTask1 and MyTask2 classes, two instances of the same task_function() method shown by Figure 19 is associated to these two Xenomai tasks.

```
public void IECRoutine(){
  ExternalMyProgram.Gd.StepSizeVar=
              ExternalMyProgram.Gd.MaxValueVar-
              ExternalMyProgram.Gd.MinValueVar;
  ComputedResult = ExternalMyProgram.Gd.StepSizeVar;
}
```

Figure 20: IECRoutine().

As said many times until now, in the Figure 19 declaration of the local variable of the IEC 61131-3 MyProgram program (ComputedResult) is not defined; the figure points out only the three different scopes where declaration may occur. The following section will definitely clarify the position of the declaration of local variables, through a deep analysis of the impact of the possible choices on the run-time performance of the system.

6 Performance evaluation

It is well known that execution of a generic C# application on a CLR VM may be delayed by the activation of the Garbage Collection. When a collection starts, it causes the stop of all the tasks associated to the C# program, including the Xenomai real-time tasks in the real-time environment architecture shown by Figure 10. The main consequence is the increase of each single execution time of C# programs; furthermore, the periodicity of one or more tasks could be not respected if the time interval needed by the Garbage Collector to conclude its work is higher than the task period.

It is clear that a performance evaluation is strongly required in order to point out if what written can actually occur in the framework here defined, and, in in this case, suitable mechanisms to prevent performance deterioration must be proposed and evaluated.

During the description of the *PLC Framework* it has been left unsolved the problem relevant to the declaration of the local variables of a IEC 61131-3 program inside the class containing the task_function() method, shown by Figure 13. The possible scopes where this declaration could occur have been highlighted but no indication about the right choice has been given. Actually, this choice seems to play a very strategic role from the performance point of view of the entire real-time environment proposed. In fact, each of the three possible scopes shown by Figure 13 may led to a very different impact of the

Garbage Collector on the overall behaviour of the IEC61131-3 programs execution.

On account of what said until now, performance evaluation was carried out by the authors with the main aim to analyse the impact of the Garbage Collector on the behaviour of the real-time environment depicted by Figure 10, considering the effect of the three different scopes of local variables pointed out by Figure 13. As it will be shown, this analysis allowed to find the best choice, able to minimise the impact of the Garbage Collector over the framework here defined.

The performance evaluation has been carried out on two different architectures: the embedded system described in Section 4.1 and a general purpose computer.

Only one IEC 61131-3 ST-based Program has been considered for the performance evaluation. Several periodic tasks were associated to this Program. This choice has been done in order to make simpler the analysis of the results, removing their dependence from possible differences in the program codes executed.

Figure 21 shows the IEC 61131-3 ST language-based implementation of the Goertzel Algorithm [16] considered for the performance evaluation.

The PROGRAM section features several variables. *Q0*, *Q1* and *Q2* are output arrays used for the per-sample processing; the samples are stored in the *sbuffer* array. The *coeff* array is another basic variable of the Goertzel Algorithm; it sores some of the precomputed constants foreseen by the algorithm, needed during processing [16]. The *magnitude* array is used to store the magnitude values of the signals coming from the channels and relevant to different harmonics. *ch* and *num_ch* variables refer to the number of channels, whilst *i* and *k_max* refer to the number of harmonics. The number of samples is represented by variables n and *j*.

The first FOR cycle present in the ST code of the PROGRAM section, iterates for all the samples; the second FOR cycle allows iteration for all the harmonics to be analysed. Finally, the last cycle refers to all the channels producing the samples. The boolean condition (j = n-1) indicates the completion of the analysis of all the samples; when this condition occurs the algorithm calculates the magnitude relevant to a specific channel and to a specific harmonic, given by the value of index.

Figure 21 also shows the CONFIGURATION and RESOURCE sections containing the global constants and variables and an example of periodic task associated to the Program Goertzel. In particular, TASK MainTask1 defines a periodic task with period 100 ms and priority value 1; an instance of Program Goertzel (called MainInst1) is associated to the MainTask1. As said before, it was assumed to associate a huge number of periodic tasks to the Program Goertzel shown in Figure 21; only for reason of space limits, the other tasks are not shown in the RESOURCE section of Figure 21.

```
PROGRAM Goertzel
  VAR_EXTERNAL CONSTANT
    k_max: INT :=12;
    num_ch: INT :=8;
  END_VAR
  VAR_EXTERNAL
    coeff : ARRAY [0..k_max] OF REAL;
  END_VAR
  VAR
    count : INT;
    i : INT;
    j : INT;
    ch : INT;
    index : INT;
    n : INT;
    sbuffer : ARRAY [0..num_ch*n] OF REAL;
    Q0 : ARRAY [0..num_ch*k_max] OF REAL;
    Q1 : ARRAY [0..num_ch*k_max] OF REAL;
    Q2 : ARRAY [0..num_ch*k_max] OF REAL;
    magnitude : ARRAY [0..num_ch*k_max] OF REAL;
  END_VAR
  FOR j:= 0 TO n-1 DO
    FOR i:= 1 TO k_max DO
      count:= i * 8 - 8;
      FOR ch:= 0 TO num_ch-1 DO
        index := count + ch;
        Q0[index] := (coeff[i - 1] * Q1[index]) –
                     (Q2[index] + sbuffer[j + ch * n]);
        Q2[index] := Q1[index];
        Q1[index] := Q0[index];
        IF j = n-1 THEN
          magnitude[index] := SQRT(Q1[index] *Q1[index] +
                Q2[index] * Q2[index] –
                (Q1[index] * Q2[index] * coeff[i - 1]));
        END_IF;
      END_FOR;
    END_FOR;
  END_FOR;
END_PROGRAM

CONFIGURATION Config
  RESOURCE Resource1
    VAR_GLOBAL CONSTANT
      k_max: INT :=12;
      num_ch: INT :=8;
    END_VAR
    VAR_GLOBAL
      coeff : ARRAY [0..k_max] OF REAL;
    END_VAR
    TASK MainTask1(INTERVAL :=T#100ms, PRIORITY := 1);
    PROGRAM MainInst1 WITH MainTask1 : Goertzel;
  END_RESOURCE
END_CONFIGURATION
```

Figure 21: IEC 61131-3 ST-based application relevant to the Goertzel Algorithm.

5.3 Performance Evaluation on Embedded System

Figures 22 and 23 show the C# code produced by *ProgramsCreator* on the basis of the Goertzel algorithm shown in Figure 21, considering the local variables inside the while(true) cycle. Figure 23 details the C# code realising Goertzel's algorithm inside the IECRoutine(). As already explained in the previous section, the external variables of the IEC61131-3 PROGRAM Goertzel are defined inside class GlobalDeclaration and are used though the access to the class ExternalGoertzel, which contains the instance Gd of GlobalDeclaration. The ExternalGoertzel class contained in the .cs file is imported, as shown by Figure 22.

It has been assumed to execute the Goertzel's code with a number of harmonics equals to 6 (i.e., ExternalGoertzel.Gd.k_max constant was set to 6).

```
import ExternalGoertzel;

class Goertzel {
  void task_function () {
      double[] sbuffer =
            new double[ExternalGoertzel.Gd.num_ch * n];
      double[] Q0 = new double[ExternalGoertzel.Gd.num_ch *
                        ExternalGoertzel.Gd.k_max];
      double[] Q1 = new double[ExternalGoertzel.Gd.num_ch *
                        ExternalGoertzel.Gd.k_max];
      double[] Q2 = new double[ExternalGoertzel.Gd.num_ch *
                        ExternalGoertzel.Gd.k_max];
      double[] magnitude =
            new double[ExternalGoertzel.Gd.num_ch *
                        ExternalGoertzel.Gd.k_max];
      UInt16 count, n, index, i, j, ch;
      while (true) {
          IEC61131Program();
          rt_task_wait_period_wrap (null);
      }
  }
}
```

Figure 22: C# code produced by ProgramsCreator.

In order to analyse the execution of the Goertzel Algorithm though the use of an oscilloscope, the GPIO #2 is set at the beginning of the execution of the IECRoutine(). The GPIO #2 is reset at the conclusion of the execution of the same code. Set and reset operations are not shown in the code of Figures 22 and 23.

Figure 24 points out the GPIO #2 values during the time; the time interval during which GPIO #2 is on represents the single execution time of the Goertzel Algorithm. As pointed out by Figure 24, results achieved show that duration of the each algorithm execution maintains about the same value in time. But the figure highlights that execution of the Goertzel algorithm does not occur with the same frequency; the rt_task_wait_period_wrap() called in the in the C# code of Figure 22 is not able to guarantee that execution of the Goertzel algorithm occurred after a deterministic time interval.

```
Public void IECRoutine() {
  for (j = 0; j < n; j++) {
    for (i = 1; i < ExternalGoertzel.Gd.k_max + 1; i++) {
      count = (UInt16)(i * 8 - 8);
      for (ch = 0; ch < ExternalGoertzel.Gd.num_ch; ch++) {
        index = (UInt16)(count + ch);
        Q0[index] = (ExternalGoertzel.Gd.coeff[i - 1] *
              Q1[index]) – (Q2[index] + sbuffer[j + ch *n]);
        Q2[index] = Q1[index];
        Q1[index] = Q0[index];
        if (j == n - 1) {
          magnitude[index] = Math.Sqrt(Q1[index] *
            Q1[index] + Q2[index] * Q2[index] –
            (Q1[index] * Q2[index] *
            ExternalGoertzel.Gd.coeff[i - 1]));
      }
      }
    }
  }
}
```

Figure 23: Details of C# Goertzel Algorithm Code contained in the IECRoutine().

Figure 24: Execution of the Goertzel Algorithm shown by Figure 22 with k_max=6.

Utilisation of the CPU has been increased considering a higher number of harmonics (setting k_max to 12). Figure 25 shows the results achieved, pointing out that now the behaviour of the system is completely unpredictable. Both duration of each execution and repetition of the execution occur in an arbitrary fashion.

The behaviours depicted by Figures 24 e 25 are due to the intervention of the Garbage Collector whose execution has been forced by the choice to define the local Goertzel variables inside the while(true) loop of Figure 22. This means that, for each loop execution, these variables are de-allocated and re-allocated, producing garbage that must be collected, causing the intervention of the Garbage Collector which stops the real-time tasks producing the bad behaviour depicted by Figures 24 and 25.

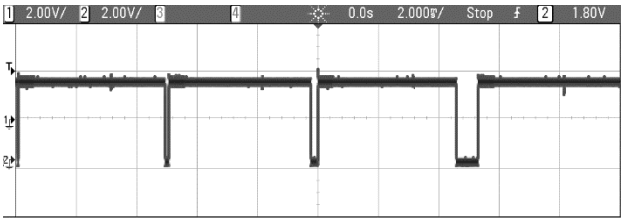

Figure 25: Execution of the Goertzel Algorithm shown by Figure 22 with k_max=12.

We proceeded to test the performance of the Goertzel algorithm by changing the scope of the local variables. The scope represented by Figure 26 has been considered; in this case, the Goertzel variable are global variable of the C# class Goertzel.

Figure 27 shows the executions of the algorithm during the time, considering a number of harmonics equal to 12 (k_max=12). As it is possible to see, now the duration of each execution is quite the same and the repetition in time of the Goertzel's algorithm is predictable because the impact of the Garbage Collector is much less than in the previous case. The variables are allocated only when the class is instantiated, and the Garbage Collector does not collect them until the deallocation of the class, that will occur only at the end of the associated task.

Another important performance improvement could be achieved defining the variable inside the task_function() as shown in Figure 28.

```
import ExternalGoertzel;

class Goertzel {
   double[] sbuffer = new double[ExternalGoertzel.Gd.num_ch * n];
   double[] Q0 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
   double[] Q1 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
   double[] Q2 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
   double[] magnitude = new double[ExternalGoertzel.Gd.num_ch
*
                                    ExternalGoertzel.Gd.k_max];
   UInt16 count, n, index, i, j, ch;

   void task_function () {
        while (true) {
             IEC61131Program();
             rt_task_wait_period_wrap (null);
        }
   }
}
```

Figure 26: C# code produced by ProgramsCreator.

In this case, it is well known that the variable access is faster than the scenario shown by Figure 16. In addition, the task_function() method is instanced only once before the creation and activation of the relevant task; this means that the so-defined variables are always active and never deallocated by Garbage Collector until the end of the task exactly like global variables. Figure 29 points out execution of the Goertzel algorithm in this scenario.

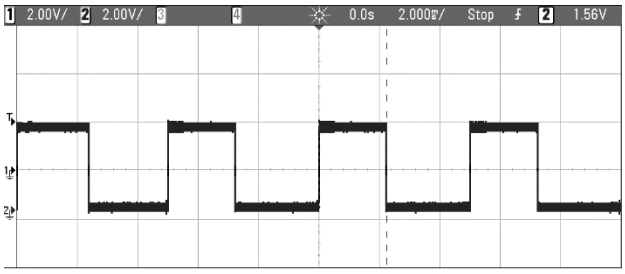

Figure 27: Execution of the Goertzel Algorithm shown by Figure 26 with k_max=12.

```
import ExternalGoertzel;

class Goertzel {
   void task_function () {
     while (true) {
        double[] sbuffer =
                  new double[ExternalGoertzel.Gd.num_ch * n];
        double[] Q0 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
        double[] Q1 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
        double[] Q2 = new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
        double[] magnitude=
                  new double[ExternalGoertzel.Gd.num_ch *
                                    ExternalGoertzel.Gd.k_max];
        UInt16 count, n, index, i, j, ch;

        IEC61131Program();
        rt_task_wait_period_wrap (null);
     }
   }
}
```

Figure 28: C# code produced by ProgramsCreator.

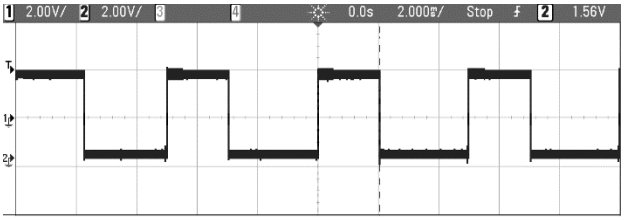

Figure 29: Execution of the Goertzel Algorithm shown by Figure 28 with k_max=12.

Comparing Figure 29 with Figure 27, it is possible to point out that the scope for the local variable considered in Figure 28 allows to improve performance of the system, as the execution time is now decreased.

5.4 Performance evaluation on general purpose computer

The algorithm shown by Figure 28 has been considered, as it allowed to achieve the best results in the performance evaluation on embedded system, as said before.

Performance evaluation has been carried out using a computer made up by a six-core Xeon processor (X5650 Intel) and 100 GB of RAM. The following software was installed on it: Ubuntu 16.04 (Kernel Linux 3.18.20), Xenomai co-kernel 3.0.2, and Mono 4.4.

Several periodic Xenomai tasks were associated to the task_function() method shown by Figure 28. It has been assumed to consider several groups of tasks; tasks belonging to each group share the same period and priority.

During execution of each Xenomai task, jitter values were measured. Figure 30 shows how jitter has been evaluated; each arrow represents a real execution of a Xenomai task.

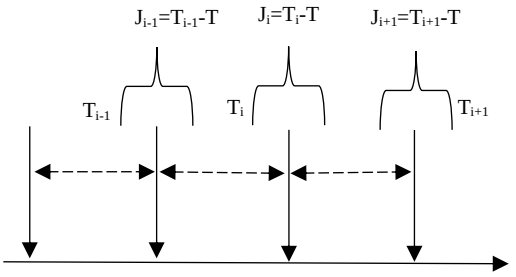

Figure 30: Jitter evaluation.

For each task, T_{i-1}, T_i, T_{i+1} are generic time intervals between consecutive Xenomai periodic task executions. Said T the period of the Xenomai task, J_{i-1}, J_i, J_{i+1} values shown in Figure 30 are the relevant jitter values. For each single task, the average absolute value of the jitters was calculated. It was said that tasks were divided into several group, each group sharing the same period and priority; for each group of tasks, the minimum and the maximum average absolute jitter values were pointed out.

Performance evaluation has been carried out considering different scenarios featured by different groups of tasks, different numbers of tasks for each group and different period values associated to a group. Scenarios were chosen in order to be comparable in terms

of bandwidth utilisation, otherwise comparison between their performances was meaningless.

For each group of tasks, the bandwidth utilisation has been defined by:

$$bandwidth\ utilisation = \frac{1}{T} * n * t \quad (1)$$

where *n* is the number of tasks belonging to the group and sharing the same period *T*, and *t* is the execution time of the Program associated to the task.

Use of the parameter given by (1) allowed to compare scenarios featured by the same bandwidth utilisation, during the performance evaluation carried out.

Tables 1 and 2 show two of the scenarios considered. Three groups of tasks have been considered for each scenario. Inside each scenario, the groups differs for the number of tasks and the relevant period. Comparing the different scenarios, they feature groups with the same bandwidth utilisation.

Tables 3 and 4 presents some of the results achieved. They show the minimum and maximum average absolute jitter values for each scenario and for each of the three groups. As it can be seen, the average absolute jitter values are always very close to zero.

Table 1: Scenario 1: Groups of Tasks with Period, Number of Tasks and Bandwidth Utilisation.

Group	Period (ms)	Number of Tasks	Bandwidth Utilisation
1	50	100	30%
2	30	100	50%
3	25	100	60%

Table 2: Scenario 2: Groups of Tasks with Period, Number of Tasks and Bandwidth Utilisation.

Group	Period (ms)	Number of Tasks	Bandwidth Utilisation
1	25	50	30%
2	15	50	50%
3	12.5	50	60%

Table 3: Scenario 1: Minimum and Maximum Average Absolute Jitters.

Group	Period (ms)	Min (ms)	Max (ms)
1	50	6.58 E-05	3.40 E-04
3	30	5.03 E-05	3.59 E-04
4	25	5.89 E-05	6.45 E-04

Table 4: Scenario 2: Minimum and Maximum Average Absolute Jitters.

Group	Period (ms)	Min (ms)	Max (ms)
1	25	5.27 E-05	1.89 E-04
2	15	5.39 E-05	3.41 E-04
3	12.5	5.86 E-05	6.96 E-04

These results seem to demonstrate that Garbage Collector does not affect at all the performance of the system. They confirm the same results achieved through the experiments carried out by the embedded system and shown in the previous subsection.

In order to verify this result, in the following an analysis will be presented in order to point out what could occur when the Garbage Collector intervenes. The C# code shown by Figure 22 has been considered. As said, in this scenario the local variables are mapped inside the while(true) cycle; this means that the entire set of variables are re-located for each cycle. This affects the heap memory capacity, going to fill it and forcing the Garbage Collector to intervene to free the unused variables, as each cycle uses another set of the same local variables.

Tables 5 and 6 show the minimum and maximum average absolute jitter values for the same scenarios seen before. It is important to point out the higher values of the jitter. Furthermore, it is important to compare the maximum average absolute jitter values with the period of each group; in some cases, the values are close to the same periods, pointing out the very bad performance achieved.

Time instants of each Xenomai task execution have been recorded during the performance evaluation, considering again the C# code shown by Figure 22. The entire set of the execution times for the tasks belonging to each group has been carefully analysed. Analysis pointed out that Xenomai task executions sometimes featured the behaviour depicted by Figure 31. Each vertical arrow in the figure represents a real execution; T_i is the time interval between two consecutive executions, and the dotted vertical arrows represents the instant at which a periodic execution is expected but does not occur.

Table 5: Scenario 1: Minimum and Maximum Average Absolute Jitters.

Group	Period (ms)	Min (ms)	Max (ms)
1	50	9.38	11.30
2	30	5.62	11.12
3	25	4.77	12.56

Table 6: Scenario 2: Minimum and Maximum Average Absolute Jitters.

Group	Period (ms)	Min (ms)	Max (ms)
1	25	2.12	2.56
2	15	1.30	2.49
3	12.5	1.10	2.62

As shown by Figure 31, during a task execution, jitter values greater than a multiple value of the task period may occur. It has been observed that the generic T_i may be greater than two or three times the task period, in the worst cases. The only event which could cause this behaviour is the running of Garbage Collector causing the stop of the

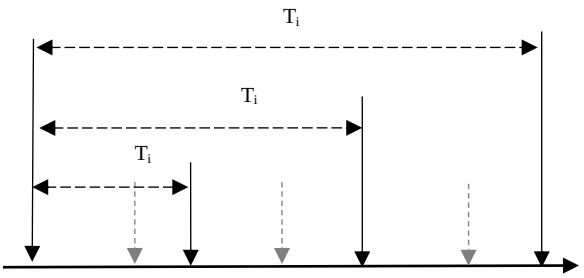

Figure 31: Xenomai task executions.

Xenomai tasks. Values of the time interval T_i clearly depends on the execution times of the Garbage Collector needed to collect all the garbage produced by the programs.

7 Final remarks

Paper has presented a software solution allowing the execution of IEC 61131-3 applications into computing systems based on a CLR VM. The software solution is made up by two main components.

The first component is a software able to realise translation of a generic IEC 61313-3 application into C# code. For each IEC 61131-3 application a .cs file is produced containing several classes relevant to the original IEC 61131-3 sections; these classes may be directly instantiated and used in a generic C# program running inside a CLR VM. Otherwise, the output produced may be passed to the second component here presented, which is a real-time execution environment. It is a framework able to realise the exact behaviour of a PLC (e.g., Program Scan loops and real-time task scheduling). It requires the presence of a CLR VM running on the top of a real-time operating system. The framework receives the C# classes produced by the first component described before, achieved for an IEC 61131-3 application. On the basis of these classes, it produces suitable C# programs and real-time tasks associated to the programs to be submitted to the underlying real-time operating system. In this paper, use of Xenomai real-time co-kernel has been presented.

After a description of both the software components, the paper focused on a performance evaluation of capability of the real-time environment to respect the periodic constraints of real-time tasks. As known, in a CLR VM-based environment, execution of a generic C# program may be delayed by the activation of the Garbage Collection. When a collection starts, it may cause the stop of all the tasks associated to the C# program and the increase of their execution time. The periodicity of one or more tasks could be not respected for the same reason. The results of the performance evaluation carried out by the authors, pointed out that although Garbage Collector may be a cause of performance degradation, its impact on the performance of the system may be drastically limited. This can be achieved by realising the right mapping between the IEC 61131-3 original local variables defined inside IEC 61131-3 PROGRAM section and the variables used by the C# classes generated by the real-time environment. Results presented in the paper, pointed out that the mapping choices operated by the authors avoid the intervention of the Garbage Collector. Under their adoption, performance evaluation allowed to demonstrate the capability of the real-time environment here presented to respect real-time constraints of periodic tasks.

To the best of authors' knowledge, current literature does not provide solutions aimed to deploy IEC 61131-3 applications on CLR VM, using C# language as intermediate code. Due to the spread current use of C# language in the development of industrial applications, adoption of the software solutions here presented seems attractive. Typical candidate platforms on which deployment may be achieved, are those based on general purpose computer architecture (on which CLR VM allows the use of common operating systems like Linux and Windows), but also all the embedded systems supporting a CLR VM may be considered.

Furthermore, the paper gives a contribution to a very spread research field currently present in literature; in particular it introduces a solution able to move computation further away from the field level into the so-called compute pools, which are decentralised and may be also realised inside cloud computing solutions. In the scenario proposed, the PLC is migrated to the compute pool which can be realised by a computer architectures based on CLR VM, as demonstrated by the research presented in the paper.

Although the paper has been presented considering Just-in-Time compilation, it is important to point out that the procedures presented in the paper and aimed to translate original IEC 61131-3 language-based programs may be applied also into the case the Ahead-of-Time (AOT) compilation was adopted.

References

[1] R.W.Lewis (1998). *Programming Industrial Control Systems Using IEC 1131-3*, IEE, ISBN-13: 978-0852969502, 1998.

[2] J. D. Decotignie (2009). The many faces of industrial ethernet [past and present]. *IEEE Industrial Electronics Magazine*, vol. 3, no. 1, pp. 8–19. https://doi.org/10.1109/mie.2009.932171

[3] M. Becker, K. Sandstrom, M. Behnam, T. Nolte (2015). A many-core based execution framework for IEC 61131-3. *Proceedings of 41st IEEE Annual Conference IECON 2015*, pp.4525-4530, https://doi.org/10.1109/IECON.2015.7392805.

[4] S. Mubeen, M. Becker, X. Zhao, L. Gan, M. Behnam, T. Nolte (2016). Towards automated deployment of IEC 61131-3 applications on multi-core systems. *Proceedings of IEEE World Conference on Factory Communication Systems (WFCS 2016)*, pp.1-4, https://doi.org/10.1109/WFCS.2016.7496531.

[5] M. Simros, S. Theurich and M. Wollschlaeger (2012). Programming Embedded Devices in IEC 61131-Languages with Industrial PLC Tools using PLCOPEN XML. *Proceedings of 10th Portuguese Conference on Automatic Control (2012)*, pp.51-56.

[6] O. Givehchi, H. Trsek, and J. Jasperneite (2013). Cloud computing for industrial automation systems - a comprehensive overview. *Proceedings of IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA 2013)*, pp.1–4. https://doi.org/10.1109/etfa.2013.6648080

[7] O. Givehchi, J. Imtiaz, H. Trsek, and J. Jasperneite (2014). Control-as-a-service from the cloud: A case study for using virtualized PLCs. *Proceedings of 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, pp.1–4. https://doi.org/10.1109/wfcs.2014.6837587

[8] I. Kühl and A. Fay (2011). A Middleware for Software Evolution of Automation Software. *Proceedings of IEEE 16th Conference on Emerging Technologies and Factory Automation (ETFA 2011)*, pp.1-9. https://doi.org/10.1109/etfa.2011.6059109

[9] Mono official website, available on http://www.mono-project.com/

[10] Xenomai official website, available on https://xenomai.org

[11] S. Cavalieri, L. Galvagno, M.S.Scroppo (2016). A Framework based on CLR Virtual Machine to deploy IEC 61131-3 programs. *Proceedings of 14th International Conference on Industrial Informatics (INDIN 2016)*, University of Poitiers, Poitiers, France,pp.126–131, https://doi.org/10.1109/INDIN.2016.7819146.

[12] S. Cavalieri, L. Galvagno, G. Puglisi, M.S. Scroppo (2016). Moving IEC 61131-3 applications to a computing framework based on CLR Virtual Machine. *Proceedings of 21th International Conference on Emerging Technologies and Factory Automation (ETFA 2016), Berlin, Germany*, pp.1-8, https://doi.org/10.1109/ETFA.2016.7733632.

[13] Xenomai official API reference website, available on https://xenomai.org/api-reference/

[14] Freescale Semiconductor, *MPC8309 (2014). PowerQUICC II Pro Integrated Communications Processor Family Hardware Specifications*, Data Sheet. Document Number MPC8309EC, Rev 4, 12/2014. Available on http://www.nxp.com/

[15] Gold parsing system official website, available on http://www.goldparser.org/

[16] G. Goertzel (1958). An algorithm for the evaluation of finite trigonometric series. *American Mathematical Monthly*, Vol. 65, No. 1, pp. 34-35, https://doi.org/10.2307/2310304