RTAI-based Execution Environments for Function Block Based Control Applications

G. Doukas¹, A. Brusaferri², M. Colla³, K. Thramboulidis¹

¹ University of Patras
Elec. & Comp. Engineering
26500, Patras, Greece,
gdoukas@ee.upatras.gr
thrambo@ee.upatras.gr

² Institute of Industrial Technologies and Automation – CNR
20131 Milan, Italy
a.brusaferri@itia.cnr.it

³ CIMSI Institute
SUPSI - University of Applied Sciences
6928 Manno, Switzerland
marco.colla@icimsi.ch

Abstract

The IEC 61499 standard has been recently adopted to promote a more flexible development process in the control and automation domain. The standard mainly deals with modeling issues leaving a lot of model execution details open. Different research groups are already working for the development of execution environments for the defined component model, proposing implementations with different execution semantics. This paper discusses the semantics of the execution environment and presents two execution environments, which although independently developed share a similar view of IEC61499 execution semantics and are both implemented over RTAI (Real Time Application Interface), a real-time Linux-based platform.

1. Introduction

The International Electrotechnical Commission, to address the increasing demand for a more flexible development process in control and automation systems, has defined the IEC61499 standard [14]. This standard attempts to introduce current software engineering practices, such as object technology and component based development into the control and automation domain. It uses the Function Block concept as its main control application building construct, and is considered as an evolution of the in industry widely used IEC 61131 standard.

Even though a significant research to the direction of utilizing the IEC61499 in practice has been already done by different research groups from academy and industry, execution environments for the component model defined by the standard are still under development. FBRT [6], Corfu and Archimedes execution environments [13] [10], and ISaGRAF [7] are the most known works to this direction until now.

particularly two IEC 61499 execution environments, the RTAI-AXE and the CEC, although independently developed in different countries and contests, are based on the same real-time operating system and adopt quite similar interface mechanisms to the physical input/output and communication systems.

The CEC execution environment has been designed and developed inside a currently running European project [1] by a joint work of Swiss and Italian research institutes, with the goal to apply it to a real transport line small-scale plant. For this environment the hard real time operating system RTAI [2] runs on standard PCs, while the interface to the I/Os is realized using Comedi drivers.

On the other hand the RTAI-AXE has been designed and implemented as a part of Archimedes System Platform [10]. The RTAI Linux hard real-time operating system was also utilized along with the Comedi acquisition infrastructure, for a prototype implementation of RTAI-AXE. The kernel-space implementation has been avoided through the use of LXRT, which is the user-space real-time programming interface that RTAI provides.

Even though the semantics of an execution model are independent from the interface to the underlying services, it is interesting to examine how these two 61499-compliant execution environments, starting from a common base, solved the implementation dependent and by the standard intentionally undefined issues.

The remainder of this paper is organized as follows: In the next section some background information of this work is given. In section 3, the FB execution model is discussed and its implementation on the two runtime environments is presented. In section 4, implementation issues regarding FB network diagrams are presented for both execution environments. Section 5 describes the interface layer to the mechanical process as a significant
part of an IEC61499 execution environment. Finally, the paper is concluded in the last section.

2. Background work

2.1. IEC 61499 execution environments

The FBRT [6] is the first execution environment for IEC61499 FB based control applications. This is a Java implementation of a FB runtime environment with no real-time characteristics.

ICS Triplex ISaGRAF [7], a well known commercially available toolset that supports the IEC61131 function block, includes in its latest version support for IEC61499. The proposed execution environment, even though not completely compliant with the standard, provides the first commercially available tool.

The increasing interest of the scientific community for IEC61499 standard has been expressed through European or National projects, such as TORERO (http://www.uni-magdeburg.de/iaf/cvs/torero/), and uCrons (http://www.microns.org/), with each of them proposing a different solution for IEC61499-compliant application execution.

Other researchers also address the need for appropriate runtime environment to support the execution of FB-based applications. In [8], details of the Fuber runtime environment, an open-source Java-based FB runtime implementation, are presented, while in [15] an approach for executing FB networks utilizing object interaction via the port concept is proposed.

2.2. Motivation for execution environments

Among the most important goals for the development of control and automation software we discriminate are the provisioning of good real-time performance and high level of modularity, re-configurability and maintainability. The accomplishment of these goals was the motivating force in the development of both RTAI based execution environments.

The CEC approach mainly focuses on real-time performance and automatically generates the target executable from the application design model. This executable is composed of the CEC execution environment and the control application. Neither configuration tables nor services are present in the runtime environment, because it was decided to resolve such issues at design time for the whole application [3].

This choice allowed to speed up the execution of the application by generating it in ANSI C language [4], eliminating the need for object oriented artifacts for supporting features like dynamic download and instantiation of single FBs.

Following a different philosophy the RTAI-AXE focuses on re-configurability and maintainability, while achieving good performance as illustrated in [13]. The runtime environment, implemented in C++, is application independent and when executed on networked devices it offers a distributed framework that supports the deployment and execution of any FB based application. Appropriate tools, bound to an Engineering Support System (ESS) such as Corfu FBDK [9] or Archimedes ESS [10], can support the automatic generation of FB types, in the form of dynamic loading libraries, which will eventually be downloaded to the control devices and utilized, by means of appropriate configuration commands, to form an executable FB application.

2.3. RTAI Linux

Linux, an open-source Unix-like operating system introduced by Linus Torvalds in 1991, was initially designed for general purpose computing systems. So, it was optimized for maximum throughput and average performance of all processes and could not cover requirements of real-time applications. The idea of extending Linux to support such applications gave birth to real-time Linux variants. RTAI (Real-Time Application Interface) [2], one of these variants, is released under GPL as an open source real time extension of Linux. RTAI is provided as a minimal Linux kernel patch and a collection of Linux kernel modules, which result in a modular architecture that makes RTAI flexible, upgradeable and very attractive for use in the control and automation domain. RTAI is based on a hard RT micro kernel that runs Linux as its lowest priority task. It uses a virtual interrupt mechanism that handles the interrupt enable/disable requests on behalf of the Linux kernel, thus the RT kernel can never be pre-empted by Linux. The same mechanism is also utilized by the RT kernel to forward the hardware interrupts to Linux. This technique allows the RT micro kernel to run along with Linux in a completely transparent way.

3. Function Block execution model

The core element of an IEC61499-compliant runtime environment is the FB’s execution model. The standard distinguishes between basic and composite FBs as well as between FB types and FB instances. According to the standard, basic FB instances of a given type provide a specific interface of event/data I/O, while their behavior and functionality are defined by means of the ECC and encapsulated algorithms. Composite FB instances of a specific type also provide a specific interface and are actually an encapsulation of interconnected FBs (FB network). Although composite FBs could be considered as design time artifacts, the need for appropriate runtime modeling has been identified [11]. The rest of this section describes the FB execution models of the proposed execution environments.

3.1. CEC FB execution model

In the CEC environment, a Basic Function Block type is modeled as a set of specific procedures, one of which
implements a dedicated Execution Control Chart (ECC), which in turn directly calls actions for executing algorithms and emitting events without interacting with external schedulers. The basic FB type contains also an appropriate data structure with pointers to input, output and internal data variables.

The basic FB instance is represented by a variable of the above defined FB type. This variable is passed as parameter to the FB type procedures. A basic FB instance is always passive and assigned to a single, active container that manages the event flow by lining up the received output events and calling the interested FBs with the related input event. The ECC is activated and the corresponding algorithms are executed. Generated events are pushed to the container’s event queue for further processing, as shown in figure 1.

Figure 1: CEC FB execution semantics

In order to avoid unpredictable behavior, each basic FB cannot be reentrant. However, events that arise when a FB is running are never lost; they are simply stacked in the event queue of the FB container.

This event management approach will be better clarified in the Function Block Network execution model paragraph of the corresponding section.

The basic FB execution has been implemented according to the single run principle adopted in [5]. This means that the input event can be used at most once for clearing the first transition, and then is canceled. Nevertheless, if the successive transitions are expressed using just guard conditions, several states may be traversed during a single FB call.

Thank to this approach, and if the guard conditions are mutually exclusive, the evaluation order of transitions should always be deterministic.

The composite FB type is also modeled as a set of procedures and a data structure with pointers to input and output data and to the variables which represent the contained FBs.

Similar to a basic FB instance, a composite FB instance is a variable of the FB type, which is passed as parameter when calling one of the composite FB procedures. A composite FB is always called, like a basic FB, with a single event in input.

In the CEC execution environment, a composite is always an active container, or thread, which manages the event flow of the contained FBs network through its own queue and in turn is assigned to a single active container. As a composite can contain other composites, the execution environment is automatically separated in a vertical hierarchy of threads, one for each level of composition, with crescent priority while descending the hierarchy.

This way the behavior of a composite is predictable, because when is already running it cannot receive new events from its container and cannot be reentrant.

3.2. The RTAI-AXE FB execution model

In RTAI-AXE the FB instances are not considered as instances of the FB type class. The implementation scenario proposed in [12], where two different classes are utilized to map FB types and FB instances, is adopted. The FB type class encapsulates the common behavior and functionality of FBs of the same type, namely the ECC and Algorithms. The FB instance class provides the means of creating FB instances that refer to FB type instances and encapsulate FB input, output, and internal variables and FB state. The ECC is not implemented as a function, but as a state chart structure with appropriate references to FB type constructs, such as actions (that is pairs of algorithms and output events) and transition conditions. FB algorithms are mapped to methods of FB type class that can act upon a FB instance that is passed as a parameter. ECC transition conditions are also implemented as FB type class methods that return a Boolean value. The invocation of the ECC actually maps to a call of an appropriate method, that traverses the ECC structure taking into account the current FB instance state and incoming events and decides the actions to be executed.

FBs are passive objects, so they are not executed by their own thread. The concept of FB container, which is an active object, has been introduced to handle FB execution as it will be discussed in the next section.

On early versions of RTAI-AXE composite FBs were modeled as part (sub-network) of the whole application FB network, thus the FB network execution semantics applied to composite FB execution. Later on, the need of providing means to model the input and output variables of composite FBs interface had been identified. Currently the execution of a composite FB is supported by a mechanism that wraps the event and data interface of the composite FB and enforces the operation of the FB instances of composite FB on the same set of data, thus making it behave as an autonomous component.
The actual execution of the encapsulated FBs follows the execution principles of the rest FB network through the use of FB containers.

4. Function Block Network execution model

There are several different schemes of FB network execution regarding the number of threads used and the thread-to-FB allocation policy, as examined in [11]. Three possible scenarios are distinguished:

a) the whole FB network is executed by a single thread of execution,
b) each FB instance of the FB network has its own (private) thread of execution, and
c) a single thread of execution may execute a variable number of FB instances.

Furthermore, the third scenario can be differentiated by the number of threads allowed to execute a single FB instance, thus two new alternatives are considered: 1) each FB instance is allowed to be executed by only one thread, and 2) a FB instance is allowed to be executed by more than one thread (in different time instances).

In both RTAI-AXE and CEC the application FB network is executed by multiple threads, but each FB instance is not allowed to be executed by different threads, as imposed by the former of the last two alternative scenarios (c.1). For the assignment of FB instances to threads, both environments define and utilize specific constructs, such as the “FB container” for the case of RTAI-AXE, and the “active container” for the CEC.

4.1. FBN execution model for RTAI-AXE

In RTAI-AXE, FB instances are injected into FB containers (FBCs) which handle the execution of those FB instances. The FBC accepts input events and dispatch them to its injected FB instances enforcing their execution, i.e., the execution of ECC and corresponding algorithms. Generated output events are also handled by the FBC and are either routed to FBC’s queue if the target FB instance belongs to the same FBC, or to the Event Connection Manager (ECM) of the device [13]. This approach does not impose synchronization issues on the access of FBs. Each FBC is independent in both aspects of execution and (re)configuration and can communicate with other FBCs through simple communication mechanisms, i.e. the Event Connection Manager (ECM) and the Data Connection Manager (DCM), responding to events without imposing complicated synchronization. FBCs are instances of the FBContainer class and are generated during start-up so that all the required execution resources are pre-allocated and thus can be utilized during configuration and reconfiguration of the system.

To support runtime reconfiguration, the RTAI-AXE implements event connections through an active element, the Event Connection Manager (ECM). This entity, having its own thread of execution, receives generated events on an event queue and dispatches them to the appropriate target FB containers/instances. For this reason the ECM maintains an event connection table and offers appropriate services to allow its configuration and reconfiguration. A similar, light-weight version of this mechanism is also embedded in FBCs to allow the dispatching of events within the FB instances of a single FBC. To enhance real-time capabilities an alternative version of ECM is under consideration. In this version ECM may process events through multiple queues to allow a prioritization of event dispatching.

Data connections are realized through a non-buffered, single storage mechanism. In this scheme a single storage area is reserved for representing a data connection between FBs, while every FB contains pointers to such area. A lock mechanism allows protecting accesses from FBs allocated to different containers.

On a distributed run-time environment, means should be provided to allow the implementation of event and data connections between FBs that reside on different devices. On an early attempt to implement such a distributed FB control application execution environment, the Industrial Process Control Protocol (IPCP) layer was introduced [12]. This layer offers a set of services to allow the realization of inter-device event and data connections based on a publish-subscribe mechanism. Although a CORBA middleware was utilized for the initial prototype implementation of the IPCP layer, the need to provide a more efficient implementation is identified. The RTAI linux operating system offers real-time networking capabilities through RTnet (http://www.rts.uni-hannover.de/rtnet/), an open source hard real-time network protocol stack that works over standard Ethernet hardware and implements protocols such as UDP/IP, ICMP and ARP deterministically. Currently, distributed FB network execution is supported through a simple UDP based publish-subscribe mechanism, implemented over RTnet.

Inter-device data connection is implemented by the same publish-subscribe mechanism as inter-device event connections. Two possible publication schemes are supported to allow the publication of data variables either on creation or along with the associated generated events.

4.2. FBN execution model for CEC

The CEC execution environment manages FBNs through elements called active containers. An active container is a thread that manages the event flow by lining up the received FBs output events, which are posted to its own queue, and by directly calling the contained FBs with the related input event. As already explained in Section 3.1, a composite, being a FBN container, is implemented as an active container.

Moreover, the CEC design environment allows splitting the FBN of an Application to Resources, as
stated in the IEC 61499 standard. In the execution environment a Resource is hence represented also by an active container, or thread, with its own event queue. This way different event paths, merged, split or crossed, can be executed by different threads. Every FB instance, either basic or composite, is assigned to just a single, active container.

This way the designer can choose a horizontal separation of the application in multiple threads, while the execution environment is automatically separated in a vertical hierarchy of threads, as already mentioned.

A hierarchy of active containers, namely resources and composites, is shown in figure 2.

This solution permits to avoid the rerunning of an already running FB, independently of its type, location and composition level of its instance, assuring that during each run a FB executes a single external input event.

Event connections are implemented at generation time. An output event is a message posted from the emitting FB, basic or composite, to the queue of its own active container element. An input event is a parameter the active container pass to the contained FB when calling it. When source and destination FBs are mapped on different active containers, the message representing the output event has to be routed from the source to the destination active container. The event processing is based on the FIFO queues of the active containers, either composite FBs or resources; while no extensions for priority mechanisms have been previued at design level. This way, the services of the underlying middleware are used in a flexible and portable way.

If the FBs are contained in the same resource, a single memory area is reserved for the data connection; else, it is duplicated at the ends of the connection and the data are sent with the event message among resources.

At the end, the CEC design environment allows assigning Resources to the Devices, as stated in the IEC 61499 standard. In the execution environment an extra thread is created for each Device. This thread is responsible for creating, initializing and controlling the Resource threads, as also for handling the local runtime, middleware and I/O management services.

The inter–device events and data exchange is managed by a dedicated thread called Communication Manager. Such a run–time process is responsible for the access to the communication port and interacts with the device resources by means of:

- operating system messages for the events
- specific memory areas for the data.

Two different communication mechanisms are supported. The event based communication implement a policy for which data are sent on the network each time the Communication Manager receive the respective event (OS message) from a resource. For the scan based communication policy, data are exchanged each fixed cycle time.

For the first implementation, the RTnet Ethernet based protocol has been chosen because of its openness.

![Diagram](image)

**Figure 2: active containers in the execution environment**

5. Interfacing to the mechanical process

A very important issue to be resolved in the implementation of execution environments is the interfacing of the control application to the mechanical process.

The control system software interacts with the process to be controlled using specific variables that are mapped to the input and output signals coming from the field by means of the I/O boards. In IEC 61131 compliant tools, such as the Isagraf toolset, these variables has to be defined in the dictionary of the control application and are available to the overall project. Such an approach is not feasible for the distributed object oriented approach specified by the IEC 61499 standard. For this reason the standard propose an alternative solution introducing a special type of function block called Service Interface Function Block (SIFB). Such kind of function blocks, proposed also for communication purposes, are modeled as sequences of Service primitives according to the ISO TR 8509 conventions instead of the Execution Control Chart.

Despite the specified SIFB modeling features, the implementation aspects required for the development of such function blocks, as well as for the communication interface of compliant IEC 61499 device, hasn’t already been provided with sufficient amount of details.

Furthermore the SIFB-based solution makes the design model implementation-platform dependent, since a direct access to I/O driver services is adopted for SIFB definition. For this reason an appropriate abstraction layer should be defined to hide the platform specific implementation issues of SIFBs.
5.1. The RTAI-AXE approach for process interfacing

Such an abstraction layer for the RTAI-AXE is the Mechanical Process Interface (MPI) layer, for which the architecture has been introduced in [12].

In this architecture, the MPI layer offers direct mapping of process parameters to IEC compliant event/data inputs and outputs within the application’s FBN, thus making SIFB obsolete. Of course, such an approach requires some extra constructs in the design space to substitute the SIFB construct. The concept of Mechanical Process Terminator (MPT) and Mechanical Process Parameter (MPP) was adopted to allow an implementation of this highly abstract process interface.

![Figure 3: the two different implementation approaches of the MPI layer with and without the need of SIFBs (MPIFBs)](image)

In the current implementation of the RTAI-AXE the MPI layer is implemented in a more standard compliant way. The MPP construct is preserved but is not directly mapped to FB inputs and outputs. This mapping is done by Mechanical Process Interface FBs (MPIFBs), that are SIFBs that use MPI layer services to refer to named MPPs to access process parameters in a platform independent way. The MPPs encapsulate all the platform-dependent operations that provide access to hardware I/O including the transformation of data from a hardware specific representation to an IEC compliant representation and vice-versa. The current implementation of MPI is based on the Comedi acquisition driver (http://www.comedi.org/), thus MPPs refer to Comedi device acquisition channels, and are configured during device start-up. A simple API is provided to allow MPP utilization. For instance, an output MPIFB can write a value to an analog process actuator by obtaining an appropriate reference of the corresponding named MPP through the API call getMPPAnalogOutputByName(“name”) and successively invoking the appropriate write method of the MPP instance.

Fig 3 shows the two different implementation approaches of the MPI layer with and without the need of MPIFBs.

5.2. The CEC approach for process interfacing

Starting from a different conceptual approach, that is to extend the currently available standard maintaining the SIFB modeling paradigm, a similar implementation solution has been considered within the CEC Tool, as introduced in [3].

Each I/O signal is defined as name and type within a simplified Service Interface Function Block and integrated in the control application at the resource level. Furthermore each simplified SIFB has the capability of asking for reading/writing a signal value by means of Operating System messages to a module called I/O Board Manager which has in charge the interaction with the physical I/O board. Such low level component is platform dependent and for the first version of the tool has been developed using Comedi drivers for National Instruments DIO-96 boards.

The associations between the signals names and the specific positions on the I/O boards are captured in a configuration file that is imported by the I/O Board Manager. Such a file contains the 3-uples (device name, sub-device name and channel name) that describe in a Comedi library style the location of each signal on the board, defined during the specific I/Os mapping.

By means of such two levels structured interface to the mechanical process, the control application hasn’t to contain particular I/O board dependent information.

Moreover, the I/Os managing policy that has been considered has a user issue: in order to obtain a PLC-style execution (first refresh all the inputs, then execute the logic, then refresh the outputs) each SIFB has to be connected to an E_CYCLE function block, as shown in Figure 4. Otherwise, to read/write some signals as a consequence of the logic, the specific SIFB input event has to be linked to the respective output event of the control function block.
6. Conclusions

In this paper two independently developed execution environments for IEC 61499-based control applications are analyzed. Both environments are based on the same hard real–time linux-based operating system, i.e. the RTAI Linux.

The RTAI-AXE attempts to provide a real-time, reconfigurable and easily maintainable IEC61499-compliant runtime environment, while the CEC approach mainly focuses on real–time performance. The first one exploits object oriented design and implementation to provide a distributed execution environment on a set of interconnected devices running RTAI, while the other one adopts a C language automatically generated code that is ready to be executed on the RTAI linux platform. The use of current software engineering technologies allows the RTAI-AXE to be considered as a framework that supports the deployment and re-deployment of FB-based control applications.

The different ways the same modeling standard has been implemented on the same operating system has provided a lot of suggestions that will be considered for future upgrades. Future collaboration is planned to focus on the comparison of the tools usability and performances working on the same industrial test - beds.

Acknowledgements

The work regarding the CEC approach that is presented in this paper has been partially supported by the European Commission under the sixth Framework Program within the CEC-made-shoe project (IST-NMP 2004-507378).

References