Slicing the Pi: Device-Specific IEC 61499 Design

Roopak Sinha[1], Barry Dowdeswell[1], Valeriy Vyatkin[2]
[1]Auckland University of Technology, Auckland, New Zealand
[2]Aalto University, Finland and Luleå Tekniska Universitet, Sweden
roopak.sinha@aut.ac.nz, barry.dowdeswell@aut.ac.nz, vyatkin@ieee.org

***Abstract*— The IEC 61499 Function Block standard describes an architecture to support the development and re-use of software components for distributed and embedded industrial control and automation systems. Often distributed over heterogeneous execution platforms, IEC 61499 applications are highly re-configurable; users can map individual function blocks to run on any available device. However, the standard does not allow differentiating between the capabilities of different devices in a heterogeneous platform. In this paper, we present a framework that facilitates the utilization of device-specific capabilities during the design of function block applications. Device capabilities are wrapped-up in Basic function blocks linking to low-level device drivers, allowing designers to access device features with ease during the design phase. The framework is completely compatible with the IEC 61499 standard, and remains highly flexible. As a case study, we show how function block applications utilizing low-level capabilities of Raspberry Pi devices can be written and deployed using the Holobloc FBDK development environment. This particular setting of using function blocks to program the Raspberry Pi also results in an ideal, low-cost research and teaching platform for distributed computers.**

Keywords*— *IEC 61499, Service interface function blocks, Raspberry Pi, Portable Runtimes, design

I. Introduction

The IEC 61499 Function Block architecture [1] facilitates the design and development of scalable and robust industrial control software. IEC 61499 inherently supports the scheduling and management of event-driven inputs typical of those encountered in real-world environments [2]. Designers can design, model, test individual application components called *function blocks* and then reuse blocks in networks to build larger systems.

A function block application can typically be distributed to execute over networked hardware platforms or *devices* such as programmable logic controllers (PLCs), general purpose embedded controllers, or on personal computers. Slices of the application operating on different devices can still communicate with one another using inter-device network connections, while intra-device communications could use more efficient means such as shared memory. Most design tools can distribute applications over available devices, but they require devices to be *homogeneously* abstracted. More precisely, while devices can physically be completely different, a design tool needs to have a uniform interface to all devices. This helps in deploying any part of an IEC 61499 application to any of the available devices, which is implemented by means of the standardized device management protocol. Design tools insert generic *service-interface function blocks* (SIFBs) between application slices on different devices to facilitate network communications between the slices. The ability to arbitrarily slice IEC 61499 applications and refactor them to execute over different device configurations results in a highly reconfigurable, robust, and flexible architecture ideal for industrial control systems such as baggage handling, fruit sorting, and building automation systems.

The high flexibility provided by IEC 61499 comes at a price; homogeneous abstraction means that device-specific features may be lost. Consider the distributed control system shown in Fig. 1 which uses two Raspberry Pi devices to register user inputs via switches and emit outputs using LEDs respectively. Decision making logic is implemented on a desktop PC.

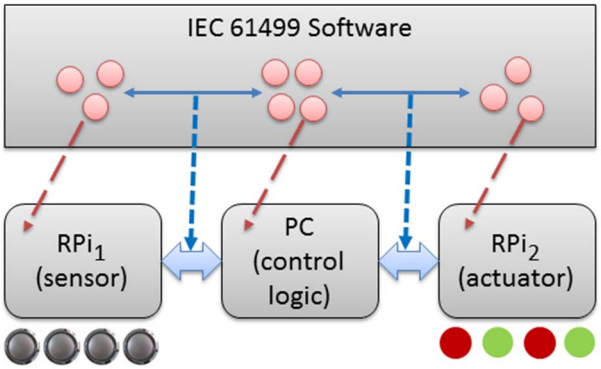

Fig. 1. An application executing on two Raspberry Pi devices.

In this example, specific Raspberry Pi features such as general purpose input/output (GPIO) ports are used to read information using switches and drive LEDs. While it is easy to program these features in low-level Java programming using device driver libraries such as Pi4J [3], these features are not available during IEC 61499 design. The focus of this paper is this disconnect between device specific features and on facilitating device-specific IEC 61499 design.

Existing works have tried to address parts of the issue of using device specific features during software design. In [2], an approach to model devices using semantic-web and service-oriented computing is presented. Embedded devices are modelled using the IEC 61499 compliant device ontology with semantic web-services providing optional configurations for application-specific use of devices. This work however does not focus on the use of individual device-specific features during software design. Most integrated development environments (IDEs) allow designers to use specific system models and configure devices using parameters [2]. However, existing IDEs cannot make device-specific features available to the designer in a seamless, integrated manner. IEC 61499 device models can include manufacturer-provided libraries of *service-interface function blocks* (SIFBs) to model device specific features [4]. Creating such libraries requires expertise in both IEC 61499 SIFB design and detailed knowledge of device details, often making it an infeasible exercise for one-off users or in situations where many previously unsupported devices must be added.

In this paper, we propose a process to allow designers to use device-specific features of devices that do not readily provide an OEM SIFB library. The proposed solution involves encapsulating lower-level device drivers as function blocks that designers can use once they have chosen specific devices to deploy their applications.

The primary benefits of our approach are the speed at which low-level device drivers and I/O protocols can be accessed in the design phase. This can help in creating more optimized, compact, and precise software for heterogeneous hardware platforms. Using our method, users can combine many lower-level device features to create custom templates, allowing faster design times for future projects. Our method also allows users to seamlessly provide several versions of the function blocks that model device-specific feature, where each version is fit for a specific purpose like simulation, testing, and deployment. A limitation of our approach is that the design process becomes more device-specific [4], and hence it is more useful for applications where a decision to deploy on specific devices can be made early in the design phase.

The main contributions of this paper are:

1. A design process to rapidly build libraries for supporting device-specific features during IEC 61499 design.
2. A case study showing the applicability of the proposed design process on a platform containing Raspberry Pi and PC devices.

The rest of this paper is organized as follows. Section II shows how design specific features are encapsulated into function blocks and made available during software design. Section III gives an overall design process enabled by our approach. Finally, concluding remarks appear in Section IV.

II. Encapsulating Design Specific Features into Function Blocks

An IEC 61499 application is typically a nested network of *function blocks*. A function block *type* defines an interface indicating its input and output events and variables, and also whether it is a *basic, composite or service-interface* function block. A basic block type is the elementary building block in IEC 61499, and contains a finite state machine, called an execution control chart (ECC), which transforms data and produces output signals when input events are registered. We can create *instances* of function blocks and connect these instances into networks. A composite function block type encapsulates a network of function block instances. Service-interface function block or SIFBs types provide standard services and communication protocols such as timers, TCP/IP sockets, etc., and can be instantiated and configured to access services provided by the underlying hardware platform.

IEC 61499 applications are *mapped* to a hardware platform which typically contains multiple devices with physical network connections. The application mapping allocates slices (groups of function blocks) to specific devices. The communications between application slices on different devices are tunneled through the physical network connections between the devices. Often, design tools automatically insert SIFBs to transmit events and variable data between application slices.

Fig. 2 presents a mapping of a slice of the distributed control application presented in Fig. 1 to a Raspberry Pi device. The Raspberry Pi [5] is a low-cost an ARM-based microcomputer that provides a highly functional research and development platform for investigating distributed applications. Individual Raspberry PIs can be used to act as sensor and actuator nodes in large distributed control applications and sensor networks. The Raspberry PIs have a number of programmable on-board features such as GPIO, network interfaces and A/V protocols that make them very flexible nodes in such large systems.

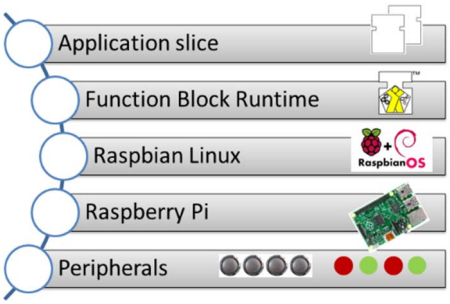

Fig 2. Mapping an application slice to a Raspberry Pi

The control application, written in the function block development kit (FBDK) [7] IDE is compiled into three slices that execute using the FB runtime or FBRT [7] on the three target devices shown in Fig. 1. The execution model used by the FBDK runtime FBRT is the Non-Preemptive

Multi-Threaded Resource (NPMTR). This model implements a depth-first scheduling algorithm. When an output event is emitted, it immediately causes the input event on the receiving block to be invoked, even if there is already a queue of other output events awaiting processing.

The FBRT executes over a Java Virtual Machine running on the Linux Operating System on each of the two devices. Communications between the two slices, which are simply connections between the function block instances on either side at the application level, are resolved into device-specific network interfaces (such as UDP or TCP/IP communications) automatically by the use of SIFBs. Communication SIFBs are commonly used and can be configured at design level to use specific ports, IP addresses, etc. by the designer.

Unfortunately, application slices must use the GPIO interfaces on the Raspberry Pi devices but no standard OEM SIFB library allowing access to the GPIO and other on-board features is available for the Raspberry Pi. While these features can easily be programmed in Java using the Pi4J device-driver library [1], they cannot be directly used during the design of the IEC 61499 control application.

In order to make device-specific features available to the designer, we propose an elegant encapsulation of low-level features into basic blocks. While our approach is currently specific to FBDK and Raspberry Pi devices, it can be easily extended to any IDE and device combination with minimal effort.

The encapsulation of device-level features into basic blocks follows the following steps:

1. Constructing a basic block interface: A designer may customize device-specific features for application-specific use. The interface of the new basic block must provide sufficient customization ability in addition to allow the application to request services using events. Consider Fig. 3, which shows the interface of a newly constructed basic block RPI_GPIO to allow the use of the Raspberry Pi GPIO during application design. The interface contains the input event REQ associated with the input variables CMD and DATA_IN. The block samples the input variables when the event REQ is read. The output even RSP is emitted by the block and is associated with the output variable DATA_OUT. This simple interface can be used by designers within applications to drive the GPIO.

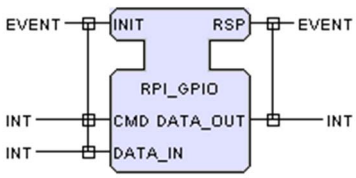

Fig. 3. Interface for the RPI_GPIO basic block

2. Linking device-driver libraries with external libraries: The newly created basic block must somehow be able to interface to low-level device driver libraries. In the case of Raspberry Pi, the Pi4J library provides the device drivers to access on-board features including the GPIO. The IEC 61499 standard specifies a standard open Extensible Markup Language (XML) format for describing basic blocks [9]. The XML format contains a *compiler* tag, allowing external code like Pi4J to be linked with the basic block as follows.

```
<FBType Name="RPI_GPIO" Comment="Basic Function Block Type" >
...
<CompilerInfo
    header="package fb.rt.rpi;
    import com.pi4j.io.gpio.GpioController;
    import com.pi4j.io.gpio.GpioFactory;
    import com.pi4j.io.gpio.GpioPin;" >
</CompilerInfo>
...
</FBType>
```

Pi4J libraries

However, instead of linking the basic block directly to Pi4J, we use an external *mediator class* written in Java, as shown in Fig. 4.

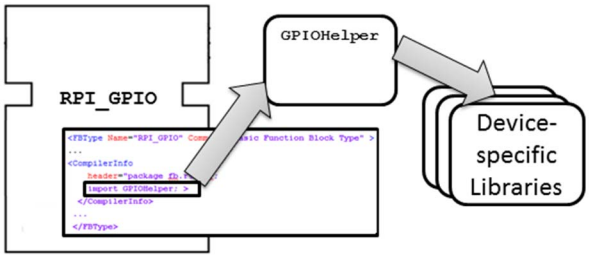

Fig. 4. Using a helper class to decouple basic blocks and device-specific libraries.

The helper class `GPIOHelper` simply contains methods for reacting to the requests sent to the basic blocks and may link to either Pi4J when the code is executed on Raspberry Pi or to an appropriate simulation code when simulation is required. Decoupling the basic block from the actual library containing the device drivers provides flexibility in running the code differently for simulation, testing, and hardware execution. Fig. 5 shows simplified pseudo-code for the helper class which contains a single `request` method for initializing, reading from, and writing to the complete GPIO port. The code can be made more complex or be replaced by code to simulate Raspberry Pi GPIO if needed.

```
//import Raspberry pi libraries
public class GPIOHelper {
    public static int request(int command, int data) {
        switch (command) {
            case 0: //initialize
            case 1: //read
            case 2: //write
        }
    }
}
```

Fig. 5. Java code inside a helper class.

3. ***Creating an ECC to use device-specific features***: The final step is to create an ECC within the `RPI_GPIO`

basic block to enable the user to configure and access low-level features. For the Raspberry Pi GPIO, this includes calling the helper class method `request` (Fig. 5.) with the values of the function block input variables `CMD` and `DATA_IN` (Fig. 3) when the input event `INIT` is read by the block. A simplified ECC is shown in Fig. 6.

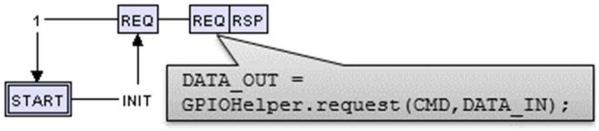

Fig. 6. ECC for the `RPI_GPIO` block

A library of basic blocks for configuring and accessing device-specific features can be quickly built by an IEC 61499 designer with some basic understanding of underlying libraries. Users can then configure and use these features by adding instances of the basic blocks in the library, without having to know the low-level implementation details. Mediator helper classes can automatically link to code appropriate for the user's run configuration; real device drivers can be used when the code is executed onto the real devices whereas simple text or visual simulation code can be used during simulation.

Creating a library of basic blocks instead of using service interface blocks involves tradeoffs. Service interface blocks are programmed using sequence diagrams, often not used by a standard IEC 61499 designer. Also, SIFBs tend to be specific for a device [10] and cannot be easily replaced by different versions for simulation purposes. Also, basic blocks are self-documenting as designers can view their ECCs whereas SIFBs typically hide their implementation details. On the other hand, SIFBs are usually optimized for a device as well as specific runtimes, providing better performance. Moreover, SIFBs can generate events within a function block library when an event happens in the connected device specific feature. E.g., when a packet is available to read, a *subscribe* SIFB can automatically generate an event to flag data availability [10]. By using basic blocks to encapsulate device-specific features, our approach only allows polled reads and therefore works in cases where polling frequency in software is higher than the frequency at which devices can update. Overall, we chose basic blocks due to the ease of use and integration of low-level device-specific features in IEC 61499 applications.

III. DESIGN PROCESS

A. Overall Design Process

The overall design process enabled by our approach is shown in Fig. 7. Designers must first create or reuse basic function block libraries for each device using the process outlined in Section II. IEC 61499 IDEs like FBDK can then be used to design the control logic. At this stage, when the

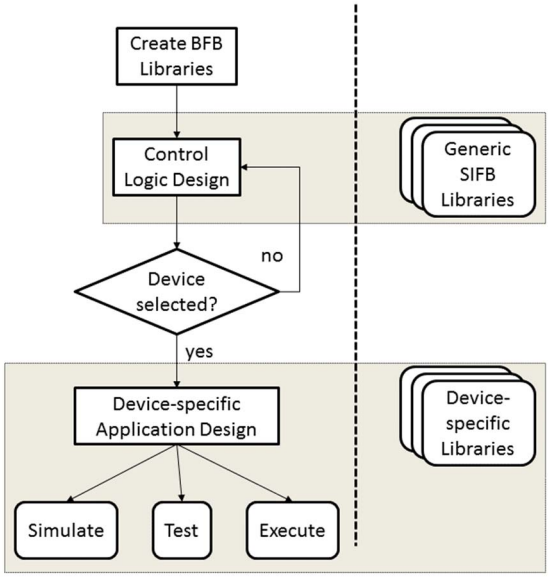

Fig. 7. Overall design process

choice of devices to deploy the application has not been made, designers can still make use of generic SIFBs like publish/subscribe blocks that are inherently supported by all devices. Once the user selects a device, corresponding device-specific features are then made available to the user to create more device-specific applications.

The device-specific libraries can then be used to simulate, test, or execute the application. Depending on the use, the mediator helper classes linked to the basic blocks contained in the device-specific libraries can relate to simulation, test or execution specific code.

B. Designing an application for the Raspberry Pi and PC

We now present details of how the control application shown in Fig. 1 was developed in a device-specific fashion for a platform containing two Raspberry Pi and one PC. Following the overall design process provided in Section III.A, we first created device-specific libraries for accessing features of the Raspberry Pi such as the GPIO. Next, platform-independent control code, such as the logic of driving the outputs based on the inputs received, was written using FBDK.

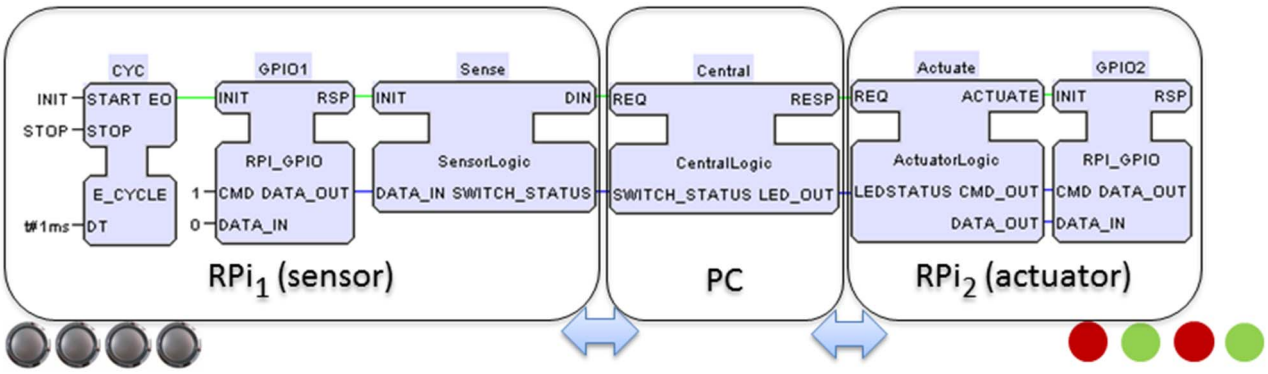

Fig. 8. Deploying the control application onto two Raspberry Pi devise and a PC.

The next step was to map the application to two Raspberry PIs acting as sensor and actuator nodes. When these devices were added, the device-specific libraries developed earlier become available to the user. This was achieved without having to modify FBDK by inserting all basic blocks in the device-specific library into the Raspberry Pi device type in FBDK. When the user selects a device, all basic blocks in the library appear within the device and can then be used in the application.

Fig. 8 provides the top level view of the control application described previously in Fig. 1. The application is distributed onto three devices. Two Raspberry Pi devices RPi_1 and RPi_2 act as the sensor and actuator respectively. A PC runs the control logic that reads the status of the switches connected to RPi_1 and then actuates the LEDs connected to RPi_2. The basic block `RPI_GPIO` which encapsulates the device-specific GPIO feature (described in Section II) is used in two ways. In RPi_1, the instance `GPIO1` provides the necessary functionality to poll the GPIO and read the status of the switches. In RPi_2, the instance `GPIO2` actuates the LEDs based on information received from the control logic.

Standard publish and subscribe SIFBs are used to channel inter-device communications between the Raspberry Pi devices and the PC. These generic SIFBs are available as library elements within FBDK and can be inserted automatically by the software when the application slices are mapped to different devices. Generic SIFBs can execute automatically in the FBRT environment, and do not require any user effort. On the other hand, instances of `RPI_GPIO` require that the associated helper classes and any further dependencies such as the Pi4J library are available when the code is compiled.

The application shown in Fig. 8 may be simulated within FBDK. We customize the helper classes associated with the device-specific blocks so that they can be used to simulate, test or run the application. During simulation on a PC, the helper class `GPIOHelper` provides command-line outputs to monitor the status of device-specific features. The code is not linked with the Pi4J library, and hence no unresolved dependencies remain during compilation. When the code is run on the Raspberry Pi, a different version of the helper class which links to the Pi4J is used, and ensures that the physical GPIO is used. While it is ideal to support this feature automatically via IDE-support, we did not have the opportunity to modify the internals of FBDK and instead implemented this facility by compiling the code with the appropriate helper classes depending on the intended usage.

Note that the control application in Fig. 8 contains an `E_CYCLE` block instance. `E_CYCLE` is a generic SIFB that acts as a timer and generates an output `EO` at regular time periods. The `E_CYCLE` block in Fig. 8 runs on RPi_1 and generates the output `EO` every 1 millisecond. This input is then read by the `GPIO1` instance to poll the status of the switches. The `E_CYCLE` block is required in our setting as device-specific blocks like `RPI_GPIO` only support polling and cannot automatically generate events when data changes.

It is important to design helper classes so that they can cater to all potential scenarios in which users can use the associated device-specific basic blocks. In Fig. 9 we deploy the control application shown in Fig. 9 entirely onto a single Raspberry Pi. In this case, the instances `GPIO1` and GPIO2 of RPI_GPIO in fact relate to the same GPIO port on the actual device. Having two instances of the block running on the same device can cause problems like resource contentions and race conditions. However, we cannot prevent users from creating multiple instances of these blocks as the IEC 61499 standard has no such restriction. . Further, development systems such as FBDK do not allow developers access to the Constructor section of the classes they create. A better way is to tweak the helper class `GPIOHelper` such that it interacts with a static GPIO reference linked to the device's GPIO. In case multiple instances are used in the same device, all read and write requests are channeled through the same static GPIO reference. The application can now be deployed across multiple devices where each `RPI_GPIO` instance will connect to the GPIO of the device it is mapped to, or we can map the application to a single device and all instances of `RPI_GPIO` will use the same GPIO in a synchronized manner. This makes control application developed using our approach to be more flexible and completely independent of the number of instances of device-specific basic blocks and

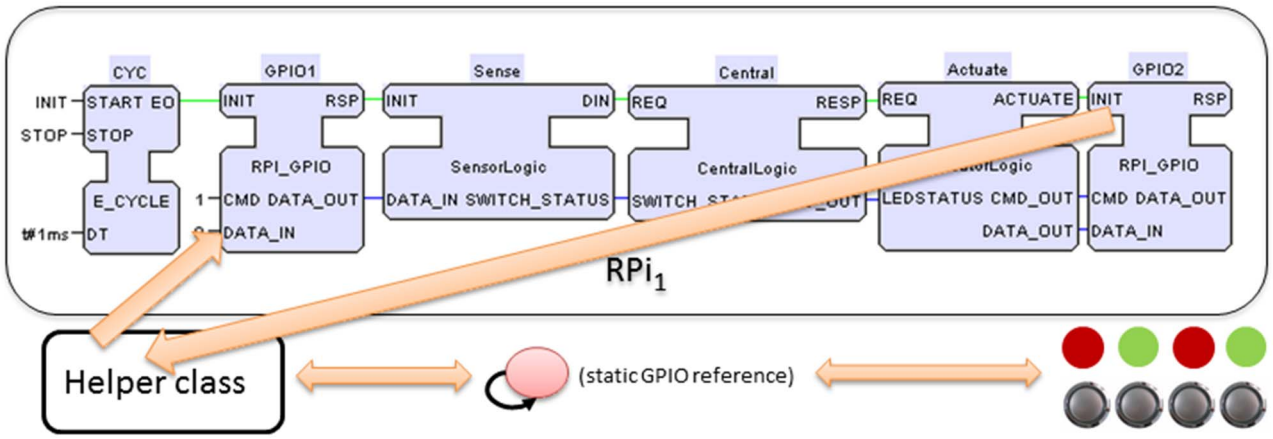

Fig. 9. Deploying the control application onto a single Raspberry Pi.

the number of supporting devices present. Of course, device-specific basic blocks can only be used when at least one Raspberry Pi device is available.

IV. Conclusions

We present an elegant method to encapsulate device-specific features into basic function blocks. Our method enables the use of device-specific features early during the design of IEC 61499 software. Using device-specific features early in the design phase involves a trade-off between flexibility and device specificity. While the design process enabled by our method means that applications are inherently specific to chosen devices, it potentially reduces the time to customize applications when devices are chosen late. In contrast to device-specific SIFBs that are made available by manufacturers and IP vendors, we allow users to rapidly build a customized device-specific basic function blocks library for unsupported devices. Helper classes acting as mediators between basic blocks and low-level device driver libraries help provide flexibility in how code is executed during simulation, testing or deployment scenarios.

Future works include embedding tool support for our method by extending open development environment like 4DIAC [11]. Further effort is required in automating the compilation and execution of device-specific code for simulation or direct execution. Finally, the issue of automatically refactoring code after a specific device is de-selected resulting in the inability to use a device-specific feature must also be investigated.

V. Acknowledgements

The authors would like to acknowledge Manju Parthinathan for her help in developing a proof of concept application for this research work. This research was supported by funding from the Auckland University of Technology's Researcher Development Fund, 2014-15.

References

[1] International Electrotechnical Commission. "IEC 61499-1: Function Blocks - Part 1: Architecture." International Standard, Second Edition, Geneva 1 (2012).

[2] K. C. Thamboulidis, G. S. Doukas, and G. V. Koumoutsos, "Device Modeling for a Flexible Embedded Systems Development Process," in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, 2007, pp. 337-343.

[3] The Pi4J Project. "Import Pi4J Packages". Retrieved from http://pi4j.com/usage.html, 2015

[4] A. Zoitl and V. Vyatkin, "IEC 61499 Architecture for Distributed Automation: the 'Glass Half Full'View," *IEEE Industrial Electronics Magazine,* vol. 3, pp. 7-23, 2009.

[5] Raspberry Pi. "What is a Raspberry PI?" [Online]. Available at https://www.raspberrypi.org/help/what-is-a-raspberry-pi/

[6] K. Thramboulidis, "Different perspectives [Face to Face;" IEC 61499 function block model: Facts and fallacies"]," *Industrial Electronics Magazine, IEEE,* vol. 3, pp. 7-26, 2009.

[7] J. H. Christensen, "Function block development kit." [Online]. Available: http://www.holobloc.com.

[8] V. Vyatkin, H.-M. Hanisch, S. Karras, and X. Cai, "IEC61499 as an Architectural Framework for Integration of Formal Models and Methods in Practical Control Engineering," 2002.

[9] C. Sunder, A. Zoitl, J. H. Christensen, V. Vyatkin, R. W. Brennan, A. Valentini, *et al.*, "Usability and Interoperability of IEC 61499 based distributed automation systems," in *Industrial Informatics, 2006 IEEE International Conference on*, 2006, pp. 31-37.

[10] V. Vyatkin, *IEC 61499 function blocks for embedded and distributed control systems design*: ISA-Instrumentation, Systems, and Automation Society, 2007.

[11] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, *et al.*, "Framework for distributed industrial automation and control (4DIAC)," in *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, 2008, pp. 283-288.