This chapter gives a brief introduction of IEC 61499 that is tailored to fit the scope of this book and should be considered a summary of the basic concepts. In the first few sections, we present the concepts regarding structure, such as the different design elements of IEC 61499 and how they fit with each other to build complex control systems. This involves a discussion on various types of function blocks and the hierarchical system model. In the later sections, we discuss the semantics, i.e. the manner in which inputs are processed to implement the behaviour of a control system. In order to illustrate these concepts, this chapter uses a Distribution Station example, with a control system implemented using IEC 61499.

2.1 Distribution Station

The Distribution Station is a mechanical assembly that picks and places work pieces on a network of conveyor belts. Figure 2.1 shows the labelled diagram of a typical Distribution Station, which consists of a Pusher 1 that places work pieces on a pick-up location and an Arm 2 that picks up the placed items and puts them on a conveyor belt. This process, therefore, requires coordination between the independently operating mechanical apparatus that are controlled by individual programmable logic controllers (PLC). IEC 61499 allows programming such systems in an object-oriented manner, whereby all physical and logical components of the system are modelled using IEC 61499 design elements.

A top-down approach of implementation begins by creating a system 3 that contains two devices (labelled as 4 and 5), where each device represents a programmable device (e.g. PLC, PAC, microcontroller). A network segment 8 connects the two devices, thus enabling the communication and coordination between the tasks assigned to each device. Device models in IEC 61499 host device-specific behaviours, such as device drivers to control the external I/O (inputs 13, 15)
and outputs (4, 6), as well as to provide the automation logic as resources (6, 7). This control logic is implemented as a network of interconnected function blocks that model the execution behaviour of the desired task, as well as any related dependencies, such as timers and communication infrastructure (e.g. 7-12). This systematic approach of modelling renders a resource as an independent operational unit of IEC 61499. While, in general, a device may contain more than one resource, in the Distribution Station example, each device hosts only a single resource, i.e. 6 and 7, respectively.

In the following sections, we begin by describing the various types of function blocks and their respective attributes. Subsequently, we revisit the concepts of system, devices and resources and discuss the IEC 61499 implementation of the Distribution Station example.

### 2.2 Basic Function Block

Basic function blocks (BFB) are the atomic units of execution in IEC 61499. A BFB consists of two separations, i.e. a function block interface and an execution control chart (ECC) that operates over a set of events and variables. The execution of a BFB
entails accepting inputs from its interface, processing the inputs using the ECC and emitting outputs. We elaborate on these in the following.

### 2.2.1 A Function Block Interface

A BFB is encapsulated by a function block interface, which exposes the respective inputs and outputs using ports. These input and output ports may be classified as either event or data ports. Figure 2.2 shows the interface of the function block that implements the Arm control logic. This interface exposes input events (PosChange, ItemStatus), output events (ArmCtrl, ArmStatus) as well as input variables (PosReadyToPickup, PosReadyToDropoff, ItemNeedsPickup) and output variables (ArmToPickup, ArmToDropoff, ArmSuck, ArmRelease, ArmClear).

Event ports are specialized to accept or emit events, which are pure signals that represent status only, i.e. they are either absent or present. On the other hand, data ports can accept or emit valued signals that consist of a typed value, such as integer, string or Boolean. Variable ports of a special type Any, can accept data from a range of typed values. In addition, a concept of multiplicity is also applicable to data ports, which allows accepting or emitting arrays of values.

A data port can be associated with one or more event ports, as shown in Fig. 2.2. For example, ItemNeedsPickup is associated with ItemStatus. However, this association can only be defined for ports of the matching flow direction, e.g. input data ports can only be associated with input event ports. This event-data association regulates the data flow in and out of a BFB, i.e. new values are loaded or emitted from the data ports on the interface when an associated event is present. Further discussion on this topic is presented in Chap. 4.

![DistStnArm: an example of a basic function block](image-url)
2.2.2 Execution Control Chart

The behaviour of a BFB is expressed as a Moore-type state machine, known as an ECC. An ECC reacts to input events and performs actions to generate the appropriate outputs. Figure 2.3 shows the ECC of the Arm controller BFB, which consists of six states, i.e. PickingUp, Waiting, Drop, etc. States in ECCs have provision to execute algorithms and emit output events upon ingress, which are represented as ordered elements in their respective action sets. As an example, the algorithm DropItem is executed, and the ArmCtrl and ArmStatus events are emitted upon entering the Drop state.

The execution of an ECC starts from its initial state (Waiting in Fig. 2.3) and progresses by taking transitions, which are guarded by an input event and an optional Boolean expression over input and/or internal variables. Upon evaluation, a transition is considered to be enabled if the respective guard condition evaluates to true. The ECC will then transition to the next state by taking the enabled egress transition from the source state to the corresponding target state.

Fig. 2.3 Execution control chart of the DistStnArm function block
Fig. 2.4 DropItem
algorithm from the Arm
controller BFB

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>me-&gt;ArmSuck = false;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>me-&gt;ArmRelease = true;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>me-&gt;ReleaseDelay = 5;</td>
<td></td>
</tr>
</tbody>
</table>

2.2.3 Algorithms

An algorithm is a finite set of ordered statements that operate over the ECC variables. Typically, an algorithm consists of loops, branching and update statements, which are used to consume inputs and generate outputs. The IEC 61499 standard allows algorithms to be specified in a variety of implementation-dependent languages. Common languages allowed by various implementations include Structured Text (ST) (e.g. [66]), Java (e.g. [42]) and C (e.g. [1, 148]). The DropItem algorithm from the Arm controller BFB is presented in Fig. 2.4 that uses the C language. Here, the identifier ‘me’ is a pointer reference to the current instance of the function block, which is synonymous to the ‘this’ keyword used in many object-oriented programming languages.

2.3 Composite Function Blocks

Composite function blocks (CFB) facilitate the representation of structural hierarchy. CFBs are similar to BFBs in the sense that they too are encapsulated by function block interfaces. However, unlike a BFB, the behaviour of a CFB is implemented by a network of function blocks.

2.3.1 Type Specification

Basic and composite function blocks specify different type specifications, which are referred to as function block types (FBTypes). A function block network (FBN) may consist of instances of various FBTypes, where any given FBType may be instantiated multiple times. This concept is very similar to the object-oriented programming paradigm, which contains classes (analogous to FBTypes) and their instances, namely, objects (analogous to FB instances). These FB instances connect and communicate with each other using wire connections, and with external signals via the encapsulating function block interface. This facilitates the structural hierarchy, i.e. a given FBN may contain instances of other CFBs that encapsulate sub-FBNs.

Figure 2.5 shows a function block network with two function block instances that communicate with each other using wire connections, e.g. a Boolean output value ItemPresent of the Pusher instance can be read as ItemNeedingPickup by the
Arm instance. Furthermore, some signals directly flow from the interface of the top-level CFB into the encapsulated FBN, e.g. the event InputsChange is read from an external source and made available to the PosChange input event of both the Pusher and Arm instances. However, only compatible signals flow in this manner, meaning that an input event on a CFB interface can only flow into an input event of nested FB interfaces. Similarly, data flow in this manner must also conform to data-type compatibility, e.g. a Boolean input on the CFB interface cannot flow into a string type input of the nested FB interface. One exception to this rule is the Any type, which, as the name suggests, can accept any data type.

This mode of signal flow is thus directly responsible for effecting the interface definition of a CFB, i.e. if a nested FB needs an input from an external source, there must be an input defined on the CFB interface which flows into the said nested FB. This encapsulation of nested FBs from external sources simplifies the reuse of FBTYPES.
2.4 Service Interface Function Blocks

Service interface function blocks (SIFB) can be considered as device drivers that connect the external environment with function block applications. These blocks are used to provide services to a function block application, such as the mapping of I/O pin interactions to event and data ports and the sending of data over a network. Figure 2.1 shows an example of device drivers (see (3)–(6) in Fig. 2.1) that are used to control the programmable device’s I/O for actuation and sensing of the physical environment. There are two categories of SIFBs described in the standard, namely, communication function blocks and management function blocks.

While CFBs capture centralized entities, resources are reminiscent of tasks and devices represent PLCs. Hence, both resources and devices need specific entities that facilitate either task-level (inter-resource) or distributed (inter-device) communication. Communication function blocks are SIFBs that provide interfaces that enable communication between IEC 61499 resources. Different types of communication function blocks may be used to describe a variety of communication channels and protocols. Figure 2.1 shows an example of a pair of communication blocks that are used to achieve coordination between the Arm and the Pusher (see (9)–(12) in Fig. 2.1). On the other hand, management function blocks are SIFBs which are used to coordinate/manage application level functionality by providing services, such as starting, stopping, creating and deleting function block instances or declarations. They are somewhat analogous to a task manager in a traditional operating system.

Unlike BFBs, where the behaviour is specified using an ECC, SIFBs are specified using time-sequence diagrams from ISO/IEC 10731 [65]. Here, we present an example of such diagrams depicting the communication between publish-subscribe communication function blocks. The publish-subscribe pair is intended for unidirectional one-to-one or one-to-many communication. Figure 2.6 presents a pair of publish-subscribe communication function blocks, which sends a single data element from the publisher block to the subscriber block. This communication

---

**Fig. 2.6** Publish-subscribe communication function blocks. The publisher function block is configured to publish a single data element, SD_1. The subscriber function block is configured to subscribe a single data element, RD_1.
mechanism is used in the Distribution Station where the two devices coordinate with each other, as shown with labels (9) and (11), which represent a publisher and a subscriber receptively. The meaning of each input and output port on the SIFB interfaces is presented below:

- **INIT**—Event to initialize the SIFB.
- **INITO**—Event to indicate that the SIFB initialization has been completed, which may or may not have been successful.
- **REQ**—Event to request the publisher block to transfer a data element over the network.
- **CNF**—Event to confirm a successful data transfer has been completed by the publisher.
- **RSP**—Event to indicate to the subscriber block that the application has processed the received data element.
- **IND**—Event to indicate that data arrived successfully in the subscriber.
- **QI**—A Boolean to indicate that the SIFB should be initialized when true or to otherwise terminate the SIFB service when false.
- **QO**—A Boolean to indicate successful initialization when true or initialization failure when false.
- **ID**—A communication identification string, such as the IP address and the port number.
- **SD_1**—The data to be sent.
- **RD_1**—The received data.

The publish-subscribe block has three separate phases of execution: connection establishment, data transfer and disconnection. Firstly, Fig. 2.7 shows the sequence diagram depicting the three scenarios for connection establishment, namely, normal establishment, publisher-rejected establishment and subscriber-rejected establishment. For both the publisher and subscriber blocks, the INIT input event is used to establish or disconnect a communication depending on the Boolean value of QI. As normative in the IEC 61499 standard, the suffix ‘+’ is used in conjunction with an input/output event name to indicate that the value of the QI/QO input/output is true at the occurrence of the associated event, while the suffix ‘−’ is used to indicate otherwise. The lines connecting events indicate a cause and effect, where the event on top is emitted prior (cause) to events at the bottom (effect).

Normal establishment occurs when the publisher and subscriber function blocks are successfully initialized. At this point, a connection is established. In the normal establishment scenario, both publisher and subscriber set QI and QO to true, respectively, to indicate successful initialization. Publisher-rejected establishment occurs when the publisher function block tries to initialize, but a connection to the corresponding subscriber block was not established. Subscriber-rejected establishment occurs when the subscriber function block tries to initialize, but a connection to the corresponding publisher block was not established. In either rejection scenario, the QO value will be set to false to indicate failure to initialize (these scenarios are depicted in Fig. 2.7).
Secondly, Fig. 2.8 shows the sequence diagram depicting normal data transfer. During normal data transfer, the publisher block receives the REQ event and sends the data at the SD_1 port to the subscriber block. Once the subscriber block receives the data, it emits the IND event to indicate that the data has been received and sends that data to other function blocks in the application through the RD_1 port. The publisher block also emits the CNF event to indicate a successful transfer. When the data is processed by the function block application, the function block application sends the RSP event to the subscriber block.

Lastly, Fig. 2.9 shows the sequence diagram depicting the disconnection phase. The disconnection can be initiated by either the publisher or the subscriber. The disconnection is initiated when either the publisher or subscriber block receives an INIT event and a false value on the QI port. If the disconnection is initiated by the publisher block, a signal is sent to the subscriber block to disconnect the connection. Once termination is successful, the publisher and subscriber blocks emit their respective INITO events and set the QO ports to false. If the disconnection is initiated by the subscriber block, the connection is disconnected without notifying the publisher block because of the unidirectional nature of the publisher-subscriber pair.
2.5 System, Devices and Resources

Device and resource models are defined in IEC 61499 to reduce the gap between the physical components of the systems (e.g. microcontrollers, PLCs, sensors and actuators) and the logical components of the automation logic, i.e. the various types of function blocks. This method of modelling automation systems bears resemblance to the object-oriented paradigm, where the system model sits at the highest level of object definition.

2.5.1 Device Model

IEC 61499 defines a device as ‘an independent physical entity capable of performing one or more specified functions in a particular context and delimited by its interfaces’. A device model, therefore, is the functional definition of a physical component in a larger distributed system. Each device may contain some inherent behaviour owing to its physical subcomponents, such as timers and reset interrupts, as well as some mapped behaviours, e.g. an automation task modelled using a function block network. In order to manage the complexity of devices, the concept of resource models is used. A device may contain zero or more resources encapsulating independent function or tasks. Figure 2.1 shows two devices (4, 5), with one resource each (6, 7, respectively).

2.5.2 Resource Model

IEC 61499 defines a resource as ‘a functional unit having independent control of its operation, and which provides various services to applications including scheduling and execution of algorithms’. A resource model, therefore, is the functional definition of an independent task executing on a device. Such tasks are
2.5 System, Devices and Resources

segregated from each other in such a way that a particular system resource (e.g. a sensor or an actuator) may only be accessed and operated upon by a single resource. Due to the absence of shared variables, resources and devices communicate using communication function blocks in order to perform the coordination between tasks.

2.5.3 System Model

The system model is used to represent an overall automation system and is defined as ‘a collection of devices interconnected and communicating with each other by means of a communication network consisting of segments and links’. Each device is capable of performing a set of independent tasks that coordinate by means of a communication network and, thus, constitute a distributed system. Figure 2.10 shows the system model for the distributed system configured with two devices containing their respective resources.

A system consists of two separations, namely, an application model and a device and resource configuration. The former describes the actual automation logic, while the latter implements its execution. The application model is primarily an FBN that consists of instances of various types of function blocks, as shown in Fig. 2.11. It is the topmost level of hierarchy of FBNs and implements the automation logic of the overall system.

This holistic view of an application model provides an unobstructed view of the overall system’s behaviour but must be partitioned in order to be implemented in a distributed fashion. For the said purpose, a subset of the application can be mapped on to a device containing zero or more resources in the configuration to implement localized subsystem/task, e.g. the function block DistStnArm is mapped to the armDevice in the system implementation. Similarly, the function block DistStnPusher is mapped to the pusherDevice for the purpose of implementing the Distribution Station as a distributed system. This partitioning of an application can raise communication dependencies, e.g. cross-device or cross-resource wire connections must be routed through a communication network. However, such dependencies can be detected automatically and resolved in a supervised manner.

![IEC 61499 system containing two devices with nested resources](image-url)
2.5.4 Implementation of the Distribution Station

The Distribution Station is implemented using the standard IEC 61499 constructs and is shown in Fig. 2.12. All function blocks in this implementation can be traced back to the high-level diagram shown in Fig. 2.1. This implementation is performed by instantiating a device model twice in a system model. Each device, in turn, contains an instance of a resource model, thereby creating a total of two nested instances of the said resource model. On the other hand, a device-independent implementation of the automation logic is created as two BFBs, namely, DistStnPusher and DistStnArm, which control the Arm and Pusher, respectively. The automation task is then assigned to the resource instances by means of a mapping process.

The mapping process enables the device-specific behaviours to be decoupled from the functional behaviours, i.e. automation tasks are not made part of the resource definition. This approach isolates the automation logic from the physical model and allows reuse of device and resource models in a system, as well as easy reconfigurability. Figure 2.12 highlights the use of this concept, where:

- Numbered labels match corresponding blocks in Fig. 2.1;
- Function blocks with a thick-solid border depict the automation logic mapped to a resource;
- Function blocks with a thick-dashed border depict the device drivers that are defined in a device FBType;
- The remaining function blocks are part of the resource FBType.

An aggregated FBN is constructed to allow these function blocks to interact seamlessly and to execute in the scope of a device.

Compilation of this implementation creates two binary executables, one for each device. The device- specific I/O blocks (13-16 in Fig. 2.12) allow interaction with external devices, e.g. Arm and Pusher for the purpose of actuation and sensing. The
Fig. 2.12 A distributed IEC 61499 implementation of Distribution Station
communication blocks (⑤-⑩ in Fig. 2.12) allow the device to coordinate using the Ethernet network, thus making the system ready to perform the desired distributed automation task.

### 2.6 Adapter Interfaces

Adapters are similar to *inheritance* in objected-oriented programming. It is used so that similar function blocks are able to share a common interface. Adapter function blocks can be configured into *acceptors, providers* or both. Acceptors are much like sockets, while providers are like plugs in an electrical system. Providers provide a service to the acceptor function blocks. For example, a function block which implements a low-pass filter to clean noisy signals can be a *provider*. Function blocks which sense physical quantities, such as acceleration and temperature, may be *acceptors* of the low-pass filter function block.

### 2.7 Execution Models for Function Blocks

We have so far covered the syntactic aspects of the standard. The semantic aspects deal with the mechanisms by which a given FB-based design (such as the one shown in Fig. 2.12) can be expected to provide the desired outcome. This section provides an overview of the semantic concepts, which will be further elaborated in Chap. 4.

In order to interpret IEC 61499 models as behaviours, the structural definitions must be paired with semantics, i.e. rules defining how to execute ECCs and their interconnections. The execution semantics can be realized via a run-time environment (which is analogous to a scheduler in an OS kernel) that takes on the responsibility of scheduling events, function blocks and the data transfer between them. An alternate approach embeds these semantics within the generated code, thus making the execution independent of any run-time environment. Benefits of the latter approach is a higher performance due to lower demand for computation power and a smaller memory footprint.

Scheduling of function blocks, i.e. when to execute a function block, can be performed in two different ways. The *event-triggered* scheduling executes a function block when a corresponding input event occurs. The subsequent execution may generate other events, which may, in turn, trigger the execution of other function blocks. In the presence of multiple events, usually a queuing mechanism is used to service events one at a time. Thus, the behaviour of the overall system depends on the event queue and its management. IEC 61499 run-time environments that adopt this execution approach are FBRT [42], FORTE [1] and FUBER [29].

The alternate approach for scheduling function blocks is the cyclic execution model, which resembles the PLC scan cycle. In this approach, each function block in the given network executes once per cycle, sampling its inputs and producing
outputs. Any event generated can be processed in the same cycle by other function blocks that are further down in the per-cycle order of execution. To achieve a robust execution order, a topological sort can be performed to schedule an event producer before an event consumer. Consequently, any cycle that is discovered needs to be resolved by using unit-length buffers to delay their processing by one cycle. \( ISaGRAF \) [66] and the synchronous approach presented in this book use the cyclic-scan approach. While \( ISaGRAF \) relies on a run-time environment, the approach expounded in this book relies on static scheduling as elaborated further in Chap. 4. In the following subsections, a brief discussion about existing execution approaches is presented.

### 2.7.1 FBRT

The \textit{function block run time} (FBRT) [42] is a Java-based run-time environment. The \textit{Function Block Development Kit} (FBDK) generates Java code for FBRT. The chosen execution scheme is the event-driven model, where occurrence of an event in the system is mapped to a direct function call on the function block instance. This results in a \textit{depth-first} model for event propagation. The advantage of this execution model is the simplicity of the generated code. However, it has several disadvantages. The generated code requires a Java virtual machine to run on the target device, which may not be suitable for resource-constrained implementations. Secondly, the depth-first event propagation may potentially require very deep memory stacks on the target device, i.e. if an invocation results in a long chain of cascading events [45]. This is especially so in cases where event loop-backs are present in the function block network.

### 2.7.2 FORTE

\textsc{ForTE} is the companion run-time environment for the 4DIAC-IDE [1] function block editor and code generator. Unlike the approach used in FBRT [42], \textsc{ForTE} adopts a \textit{breadth-first} event propagation scheme. All external and generated events are queued in a FIFO event buffer and are consumed by the respective function blocks in a sequential manner. This significantly reduces the depth of call stack for long event chains. A reported disadvantage [143] of this technique is the slow and bulky generated code due to multithreading, which may not be suitable for resource-constrained embedded systems.
2.7.3 **FUBER**

FUBER [29] is an interpreter for IEC 61499 designs that adopts a *breadth-first* event propagation approach, similar to FORTE [1]. However, unlike the global event buffer of FORTE, FUBER has chosen to create a local FIFO event buffer for each function block instance. When an event has to be notified to a function block instance, a new event is queued in its event buffer, and the function block instance is queued in a scheduler queue. The scheduler then executes the queued instances in a FIFO manner [143] to consume the events. This approach has similar advantages to the approach of FORTE.

2.7.4 **ISaGRAF**

ISaGRAF [66] adopts the PLC scan cycle execution model [131], where each function block is mapped to a separate PLC program. During a scan cycle, a function block is executed if at least one associated event is present. In this manner, function blocks are executed in a round-robin fashion with a specific order. The events produced during execution are immediately available to consumer function blocks. If the consumer block is scheduled after the producer block, the event can be consumed in the same scan cycle; otherwise, the event will be consumed in the next scan cycle. In this model of execution, the behaviour of the system is dependent on the order in which the function blocks have been scheduled.

2.7.5 **Synchronous Execution**

In this book, we propose a synchronous approach for the execution of function blocks. This approach does not require a run-time environment on the target device and has a higher performance than other function block execution models [148]. An additional advantage of this execution model is that the generated code is deterministic and deadlock-free [147]. Therefore, it not only suits various application domains but also supports a wide range of devices with varying computation power and memory capacity. Chapter 4 elaborates on this execution model.

2.8 **Discussion**

This chapter presented the basic concepts about structure and semantics of IEC 61499. We started with an overview of how distributed systems are designed and how different types of function blocks fit in this design. The concept of system,
devices and resources facilitates an object-oriented approach for designing the overall system. Basic and composite function blocks are primarily used to model the behaviour of the system, whereas service interface function blocks are used to implement low-level functions, such as device drivers or communication interfaces.

We further discussed how these models are interpreted and executed using run-time environments and their respective mechanisms. We also alluded to a synchronous approach for executing function blocks and its key benefits, which will be presented in detail in the subsequent chapters.

In the next chapter, we provide a background on synchronous programming using the well-known Esterel language. This background is foundational for the understanding of subsequent material in Chaps. 4 and 5.
Model-Driven Design Using IEC 61499
A Synchronous Approach for Embedded and Automation Systems
Yoong, L.H.; Roop, P.S.; Bhatti, Z.E.; Kuo, M.M.Y.
2015, XVIII, 194 p. 135 illus., 66 illus. in color., Hardcover
ISBN: 978-3-319-10520-8