Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability

Wenbin Dai, *Member, IEEE*, Valeriy Vyatkin, *Senior Member, IEEE*, James H. Christensen, and Victor N. Dubinin

***Abstract*—In recent years, requirements for interoperability, flexibility, and reconfigurability of complex automation industry applications have increased dramatically. The adoption of service-oriented architectures (SOAs) could be a feasible solution to meet these challenges. The IEC 61499 standard defines a set of management commands, which provides the capability of dynamic reconfiguration without affecting normal operation. In this paper, a formal model is proposed for the application of SOAs in the distributed automation domain in order to achieve flexible automation systems. Practical scenarios of applying SOA in industrial automation are discussed. In order to support the SOA IEC 61499 model, a service-based execution environment architecture is proposed. One main characteristic of flexibility—dynamic reconfiguration—is also demonstrated using a case study example.**

***Index Terms*—Dynamic reconfiguration, flexibility, function blocks, IEC 61499, IEC 61131-3, industrial automation, interoperability, programmable logic controllers (PLCs), service discovery, service-oriented architecture (SOA), simple object access protocol (SOAP), Web services description language (WSDL).**

I. Introduction

THE INDUSTRIAL automation landscape is dominated by the hardware and software paradigm of programmable logic controllers (PLCs). PLCs are widely deployed in almost every branch of industry: manufacturing and assembly lines, building automation, process control, material handling systems, etc. PLC software is commonly developed in accordance with the IEC 61131-3 standard [1], which defines a set of textual and graphical programming languages. However, to the disappointment of system integrators, code portability between various PLC platforms is not fully achievable, due to PLC vendors' own interpretation of the standard and the need for backward compatibility with their legacy systems. An additional drawback is that the design of distributed automation systems is essentially beyond the scope of the IEC 61131-3 standard: the "configuration," considered as the highest level of its software model, is limited to a single PLC device [1], [2]. Therefore, substantial overhead is incurred in a distributed system design using PLCs under the IEC 61131-3 paradigm [3].

A distributed automation system can be designed using the IEC 61499 standard [4] in an abstract, platform-independent way. The initial ambiguities of execution semantics have been resolved in the 2nd edition of IEC 61499, of which Part 1 and Part 2 were published in 2012 and Part 4 was published in 2013. Targeted as solving existing issues of the IEC 61131-3 standard for distributed automation systems, the IEC 61499 standard has its own unique characteristics. First, all algorithms and data must be encapsulated in a software component called a "function block." In the IEC 61131-3 standard, program organization units (POU) including programs, functions, and function blocks are defined for encapsulation. Second, the concept of *global variables* makes IEC 61131-3 POUs extremely difficult to be distributed. In contrast, the absence of global variables in IEC 61499 simplifies reallocation of IEC 61499 function blocks to other devices in distributed automation systems, while the mapping of function block instances (FBIs) to devices is required only at the last stage prior to deployment. Finally, a management model is introduced in the standard to facilitate reconfigurability at runtime [5]. The management model consists of a distribution model, communication interfaces between devices, a set of commands, and suggested protocols.

There are several IEC 61499 implementations, usually including a development environment and runtime execution environment, some of which are developed in research and academia (FBDK/FBRT [6] and 4DIAC-IDE/FORTE [7]), and others being commercial products (nxtStudio/nxtRT61499F [8] and ISaGRAF Workbench/ISaGRAF runtime [9]). Interoperability, portability, and reconfigurability are partially achieved among those IEC 61499 vendors [10]. For example, function block library elements are portable between FBDK, 4DIAC-IDE, and nxtStudio thanks to XML-based representation. As demonstrated in [11], a basic level of interoperability, based on a PUBLISH/SUBSCRIBE communication model implemented via service interface function blocks (SIFBs), can be achieved between all major IEC 61499 platforms, such as FBRT, FORTE, nxtRT61499F, ISaGRAF, and BlokIDE [12]. Since SIFBs are commonly used to access hardware/firmware-provided services, SIFBs developed for one runtime are not usually portable to another platform. However, with the use of proper interface abstractions, SIFB-type definitions may be portable among development environments.

The runtime environments support some reconfiguration actions through *device management interfaces*, which implement, for instance, such commands as creation and deletion of FBIs during execution. Approaches to more intelligent

Manuscript received February 10, 2014; revised February 05, 2015 and April 01, 2015; accepted April 01, 2015. Date of publication April 15, 2015; date of current version June 02, 2015. Paper no. TII-14-1329.

W. Dai is with Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: w.dai@ieee.org).

V. Vytakin is with the Lulea University of Technology, Lulea 97187, Sweden, and also with Aalto University, Espoo 02150, Finland (e-mail: vyatkin@ieee.org).

J. H. Christensen is with Holobloc, Inc., Cleveland, OH 44121 USA (e-mail: james.h.christensen@gmail.com).

V. N. Dubinin is with the University of Penza, Penza 440026, Russia (e-mail: victor_n_dubinin@yahoo.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2015.2423495

actions, such as automatic deployment [13] and automatic fault recovery [5], have been proposed and demonstrated, but are not yet standardized, while other advanced capabilities such as automatic load sharing between controllers are yet to be implemented.

The concept of service-oriented architecture (SOA) has been introduced in the general computing domain to facilitate the creation of distributed networked computer systems. The application of SOA and Semantic Web technologies in industrial automation was proposed by Jammes and Smit [14]. There are several existing approaches to dynamic reconfiguration based on the IEC 61499 standard. A concept of an engineering support environment based on SOA for dynamically reconfiguring IEC 61499 systems is proposed by Thramboulidis *et al.* [15]. However, it seems to be limited only to the design level as no detail on a service-oriented runtime environment is given. The management model is investigated and a reference implementation is provided by Zoitl [16] and an IEC 61499 standard-compliant reconfiguration method is proposed by Strasser *et al.* [17]. One premise of these approaches is that function block-type definitions already exist in the target execution environment, which imposes a serious limitation to runtime reconfigurability of systems. This paper proposes an SOA-based runtime architecture to remove this limitation.

SOA provides a software design pattern, in which software components are only connected via messaging. This design pattern is consistent with primitives of distributed automation systems: *modularity* and *communication* [2], [18]. The loose coupling introduced by SOA ensures interoperability between various platforms regardless of their hardware and software architecture. Finally, each service registered at a service repository could be discovered and invoked from other services by exchanging service contracts. This increases the reusability of programs.

In this paper, a method for the application of SOA to distributed automation systems is proposed. This approach aims at increasing system flexibility and reducing costs of development and integration. This paper is organized as follows. In Section II, recent relevant research works on applying the SOA concept in the industrial automation domain and existing dynamic reconfiguration approaches are reviewed. The potential benefits of SOA principles for distributed automation systems are discussed with applicable scenarios in Section III. A formal model and definitions for mapping SOA principles to IEC 61499 are described in Section IV. Section V illustrates an IEC 61499 execution environment architecture based on SOA. One of the main features of flexibility—dynamic reconfiguration—is demonstrated with a case study in Section VI. Following that, a preliminary analysis and comparison of the proposed SOA-based runtime is provided. Finally, conclusion as well as possible future works is provided in Section VIII.

II. Related Works

Several researchers share the ideas of applying SOA as a kind of middleware to achieve interoperability and communication between distributed nodes in the industrial automation domain.

Lastra *et al.* [19] discussed current trends in industrial automation, especially in factory automation, concluding that immediate benefits could be provided to factory automation by introducing XML, Web services (WSs), and Semantic Web Technologies. Delamer *et al.* [20] proposed a middleware based on the CAMX event and SOA, enabling discovery, message exchanging, and self-configuration in distributed factory systems. A new protocol is invented in order to propagate semantic service descriptions. Lobov *et al.* [21] investigated applying Semantic WSs and ontologies into the manufacturing industries. In their solution, ontology and reasoning are used to define responsibilities for each device (possibly from different vendors), and Semantic Web technologies are used to query operations (services) of industrial processes.

Candido *et al.* [22] investigated a common architecture to support different phases of the device lifecycle by combining evolvable production systems and SOA. As a result, a modular, adaptive, and open infrastructure is formed, in which components can interact and be combined to meet legacy system specifications. The authors also illustrate a dynamic deployment process, which uses a PLCopen device configuration XML file [23] to fill in a predefined service class template and convert to XML deployment file using WS management (WS-management) [24]. This process focuses on providing dynamic deployment to IEC 61131-3 PLCs from a model-driven engineering perspective. No improvement is made on the execution environment, whereas this paper aims for enabling flexibility on the runtime level.

Stoidner *et al.* [25] propose a SOAP4PLC engine to invoke a manufacturing task running on a PLC via WSs. In their approach, which aims at IEC 61131-3 PLCs, adding SOA FBs to process SOAP messages does not seem to improve flexibility and reconfigurability; however, the fundamental concepts of mapping FB to services and using SOAP messages are also applicable for IEC 61499.

Various formal models of SOA have been developed in the IT and Internet-of-Things (IoT) domain [26]–[28]. However, these formal models cannot directly apply to automation systems. For example, in [28], the business process scripts, procedural packages, and all package headers are defined in a system structure which does not exist in the automation domain. Software structures in automation systems are also not covered by any of these models, since there is no function block concept in the general computing domain.

Finally, there are also some SOA-related works in the area of dynamic reconfiguration of industrial automation systems.

Middleware is proposed by Valls *et al.* [29], [30] for reconfiguration of distributed real-time systems based on SOA. A formal system and software model are developed to support the middleware. The dynamic reconfiguration is bounded with time and service composition and is performed at the resource manager during runtime. The entire control is implemented as an automation agent incorporating high-level control and low-level control (LLC). However, dynamic creation and deletion of LLC components' types during operation is not covered in the middleware.

A knowledge-based framework for dynamically adaptive systems is proposed by Thramboulidis *et al.* [15]. The knowledge-based framework is based on ontologies, SOA, and Semantic Web languages. The SOA-based framework is integrated with software agents to achieve negotiation between support systems.

A general concept of autonomous application recovery in distributed intelligent automation and control systems is presented by Strasser *et al.* [5], [17]. The reconfigurability provided by the IEC 61499 standard is used as the basis for dynamic reconfiguration and recovery. IEC 61499 management commands are utilized to create and initialize function blocks at the runtime level. State and data could also be transferred from one IEC 61499 resource to another by management commands. An IEC 61499 standard-compliant reconfiguration method is also proposed [17]. The SOA-based runtime architecture presented in this paper is complementary to the reconfiguration methods proposed by Strasser. The methods could be applied to the proposed SOA-based execution environment without major modifications.

An agent-based approach for dynamic reconfiguration of real-time distributed control systems is proposed by Brennan *et al.* [31]. IEC 61499 function blocks are used to model the system and agents, including coordinator agent, mobile agent, and cohort agent, to achieve dynamic reconfiguration. A similar idea using agents for self-reconfiguration of IEC 61499 systems is presented by Lepuschitz *et al.* [13], [32]. Reconfiguration is managed by an application implemented on the function block level and reconfiguration models are defined using an ontological knowledge base. The reconfiguration model presented by Lepuschitz is based on the model provided by Zoitl in [16] and covers the changing of program sequences; adding, deleting, relocating, and replacing instances of software components; and changing parameters of software components. Dynamically adding, deleting, relocating, and replacing definitions of software components are feasible using these models but not yet fulfilled by the execution environment.

Applying the SOA concept to the implementation of the IEC 61499 standard at the device level is a feasible solution, which provides interoperable automation systems. This paper aims at bridging the gaps in existing reconfiguration models to enhance interoperability and flexibility in distributed automation systems. Not only creating instances but also type definitions [such as function block types (FBTs) and subapplication types] is covered by a new flexible and interoperable execution environment architecture. The flexible and interoperable runtime is also applicable for adapting both legacy- and future-distributed automation systems.

III. Interoperable and Flexible Distributed Automation Systems by Applying SOA Principles

There are several well-known principles of SOA defined in the software engineering domain [37]. The principal aim of SOA is to improve the flexibility, interoperability, and abstraction level of software components. Two key principles of SOA are *loose coupling* and *discoverability*. SOA defines that logic must be encapsulated into *services*, which could only be accessed via *messages*. The loosely coupled software components ensure future expansion to various platforms and future technologies. Complex logic hidden in services provides an abstraction for the system-level overview.

Services and messages are formally defined by *contracts*. A service contract consists of all primary definitions: general information of service, such as name, type, owner, version, and responsibility; functional description including requirements, service operations, and how to invoke the service (message components); and additional information such as quality of service, security, semantics, and description of the service. Service contracts are registered at *service repositories*, so that services could be discovered and invoked by other services.

Existing IEC 61131-3 PLC POUs are tightly coupled. Dynamic reconfiguration of IEC 61131-3 systems is achieved by switching between multiple instances of resource configurations created on PLCs. Online editing is also available in most IEC 61131-3 implementations. However, these implementations are all proprietary and there is no mechanism of managing POUs defined in the IEC 61131-3 standard. Therefore, in this paper, the IEC 61499 architecture will be used as the target platform.

Dynamic reconfiguration is the key to achieve interoperability and flexibility in distributed automation systems. The IEC 61499 standard defines a management model with a set of commands, which have capabilities to create and delete FBIs as well as connections between FBs. FBs can also be started, reset, or stopped using these commands. Benefits and applicable scenarios of dynamic reconfiguration of IEC 61499-compliant systems have been researched in a number of works, e.g., [16], [17], and [31]. These approaches improve flexibility of distributed automation systems. By introducing industrial software agents, intelligent features such as automatic deployment, automatic load sharing, automatic fault detection, and recovery can be achieved [5], [13].

Flexibility requirements are tightly intertwined with the implementation of functional and nonfunctional requirements. For example, even a standalone part of an automation system, such as a safety subsystem that is handling emergency situations, may require the reconfiguration of hardware or software. It is often the case that the safety subsystem needs updates to cover hazards discovered during system operation, which is done by the deployment of new software components to PLCs. Similarly, in airport baggage handling systems (BHSs), baggage screening policies are continuously improving to ensure safety and security. These new functionalities in the baggage screening process must be introduced without stopping systems in airports operating 24 h a day.

The IEC 61499 management model is well suited to address these requirements. Not only FBIs, but also definitions of FBTs, adapter types, and data types can be created or deleted by just sending management commands. In order to achieve that, introducing SOA at the runtime level is an appropriate mechanism. Loosely coupled services ensure that software components can be inserted or removed anytime during execution without any dependency between them. The idea of enabling SOA on the runtime level is already experimented in the automation domain by Candido *et al.* [33] using Inico S1000 remote terminal unit (RTU) [34]. The Inico S1000 RTU provides XML/SOAP [37] external interface, access to I/O modules, and integrated

HMI panels. The integration of WSs into PLCs has also been demonstrated by Mathes *et al.* [35].

One of the benefits of bridging SOA with IEC 61499 is that plug and play software components at runtime level (PnP) [36] are enabled by the service discovery mechanism. PnP is widely adopted in the computing world, e.g., in the USB interface. In industrial automation, plug and play relies on service *discoverability*, i.e., the ability of a new controller to be recognized automatically by other controllers on the same network when it is plugged into industrial fieldbuses. By exchanging service contracts between the new controller and existing controllers, the new controller could identify existing system configurations.

A second benefit of SOA is achieving interoperability between PLCs regardless of the details of internal implementations. In the IEC 61499 standard, SIFBs are usually platform-dependent; to access those blocks from another platform, communication blocks must be inserted. Considering each function block as a running service, function blocks running in one PLC could be invoked by other function blocks in other distributed PLCs without involving any SIFB within the same framework. Consequently, a PLC program could be built based on invoking external service libraries if external communication latencies are minimal compared to execution time of function blocks. Those FB libraries could be located at and invoked from PLCs in existing systems implemented by another system integrator even if their original source code is hidden.

Finally, the SOA-based approach brings benefits in terms of cost savings. The existing redundancy approaches for PLCs are mostly hardware-oriented: a duplicated (backup) controller is operating in parallel with the primary controller—reading inputs but not emitting outputs. A dedicated link between the primary and the backup PLC is continuously monitoring health status of the primary PLC. When a failure occurs in the primary PLC, the dedicated link will activate the backup PLC. By adopting SOA-based PLCs, complete sets of redundant hardware are not necessary. The redundancy could be achieved in a software-oriented approach using available computing power on other PLCs and allocating tasks automatically to available PLCs. Potentially, physical PLCs could be moved to a local cloud. Virtual PLCs could be setup on the local cloud to provide cheaper but more powerful redundancy solutions. When a soft PLC is faulted, the local cloud could immediately create a new instance of the soft PLC and switch tasks over. The hardware cost could be reduced significantly by eliminating redundant hardware.

IV. Formal Mapping Between IEC 61499 Function Blocks and SOA

In order to investigate what SOA features can be achieved in IEC 61499, a formal mapping is defined in this section. Some SOA characteristics may not be necessary in the IEC 61499 standard. The proposed formal mapping will provide only the relevant parts. The basic principles of SOA are *loose coupling* and *discovery,* whose implementation implies the existence of a service requester, a service provider, and a service repository. In the general definitions of SOA, the logic is encapsulated into service providers and registered at service repositories. When a program intends to invoke a particular logic from a service provider, the requested service will be located by the service repository for the service requester. Consequently, the service requester can access the service provider via sending messages.

SOA is defined in [27] as a four tuple

$$\mathrm{SOA} = (S, \mathrm{Spro}, \mathrm{Sreq}, \mathrm{Srepo})$$

where S is a set of services; Spro is a set of service providers; Sreq is a set of service requesters; and Srepo is a service repository.

Considering the IEC 61499 architecture, all services can be divided into function block services (S_F), resource services (S_R), and device services (S_D)

$$S = S_F \cup S_R \cup S_D, \quad S_F \cap S_R = \oslash,$$
$$S_R \cap S_D = \oslash, \quad S_F \cap S_D = \oslash.$$

Only function block services are considered in this section.

Definition 1: Each service $s \in S_F$ is mapped to an IEC 61499 FBT

$$\mathrm{s.t.}: S_F \rightarrow \mathrm{FBType}$$

where FBType is a set of IEC 61499 FBTs. One FBT can be mapped to more than one service from S_F because of multiple instantiation of FBT.

A service could be either atomic (self-contained without invoking other services) or composite (consist of other services) [37]

$$S = S_c \cup S_a$$

where S_a is a set of atomic services and S_c is a set of composite services. A composite service $s_c \in S_c$ consists a set of interconnected other (internal) services

$$s_c = \mathrm{fc}(s_1, s_2, \ldots, s_n), \quad s_i \in S, \ \ i \in [1, n]$$

where fc is a composition function.

There are three FBTs defined in the IEC 61499 standard: basic function block (BFB), composite function block (CFB), and SIFB

$$\mathrm{FBType} \ = \ \mathrm{BFBType} \ \cup \ \mathrm{CFBType} \ \cup \ \mathrm{SIFBType}$$
$$\mathrm{BFBType} \ \cap \ \mathrm{CFBType} \ = \ \oslash.$$

The functionality of basic FB is defined by a state machine [execution control chart (ECC)] and algorithms activated during state transitions. Service interface FB is used as a "black box," which is mainly responsible for external communications and platform-dependent functions. Based on the SOA approach, both *BFB* and *SIFB* cannot encapsulate other function blocks (implementation of an *SIFBType* as a *BFBType* or a *CFBType* is not limited in the IEC 61499 standard, but in principle, the details of an *SIFBType*'s internal implementation may be hidden from the user).

Definition 2: An atomic service is used to represent every IEC 61499 basic and SIFB

$$\forall \mathrm{s} \in S_F[\mathrm{st}(s) \in (\mathrm{BFBType} \cup \mathrm{SIFBType}) \rightarrow s \in S_a].$$

In the standard IEC 61499 implementation, to create intercommunication between two BFBs, SIFBs must be inserted in both resources where the BFBs reside. In the SOA-based approach, BFBs are able to send and receive messages from any other FBs under the same service framework.

Composition of other services (encapsulation of BFBs and SIFBs) is achievable using the composite FBT in the IEC 61499 standard. The CFB encapsulates BFBs, SIFBs, or even CFBs, which form a function block network.

Definition 3: A composite service type is used to represent every IEC 61499 CFB

$$\forall s \in S_F\ [\mathrm{st}(s) \in \mathrm{CFBType} \rightarrow s \in S_c].$$

The top-level entity for IEC 61499 is the *system configuration*. Each system configuration consists one or several devices. Each device may contain one or more resources. FBIs are created in a function block network (application) initially, then allocated to various resources and devices. One of the key features of the IEC 61499 standard is event-triggered execution—an FB instance is only activated when any input event connected from another FB instance is triggered. A FBI is defined as a five tuple

$$\mathrm{FBI} = (\mathrm{EIS},\ \mathrm{EOS},\ \mathrm{DIS},\ \mathrm{DOS},\ \mathrm{IVS})$$

where EIS is a set of event inputs; EOS is a set of event outputs; DIS is a set of data inputs (DI); DOS is a set of data outputs (DO); and IVS is a set of internal variables.

In terms of SOA, when an FB instance is triggered by another FB instance, this FB instance is considered as a service provider.

Definition 4: An FB instance FBI performs as a service provider when an event input is triggered

$$(\mathrm{ei} \in \mathrm{EIS}\ [\mathrm{fei(ei)} = \mathrm{true}]) \rightarrow \mathrm{FBI} \in \mathrm{Spro}$$

where ei is an event input of the FB instance; fei is an event input trigger function whose result is true when any input event is detected.

Vice versa, if the FB instance emits an output event, it is a service requester.

Definition 5: An FB instance FBI performs as a service requester (or notification) when an output event is emitted

$$(\mathrm{eo} \in \mathrm{EOS}\ [\mathrm{feo(eo)} = \mathrm{true}]) \rightarrow \mathrm{FBI} \in \mathrm{Sreq}$$

where eo is an event output of the FB instance; feo is an output event emit function whose result is true when any output event is fired.

The last element in the SOA definition is the service repository. Service repository is a storage, which maintains a collection of service definitions—service contracts. A service contract defines a communication agreement, which is independent from implementation. The service repository is defined as

$$\mathrm{Srepo} = \{\mathrm{scnt}(s) | s \in S\}$$

where scnt is a service description function, and $\mathrm{scnt}(s)$ is a particular service description.

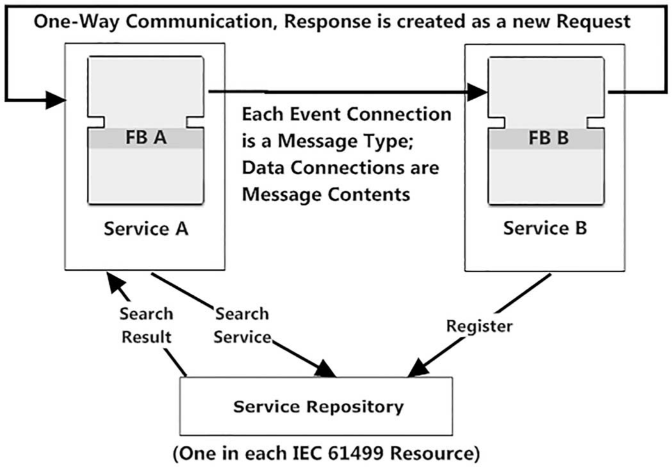

One-Way Communication, Response is created as a new Request
FB A
FB B
Each Event Connection is a Message Type; Data Connections are Message Contents
Service A
Service B
Search Result
Search Service
Register
Service Repository
(One in each IEC 61499 Resource)

Fig. 1. SOA basic structure in IEC 61499.

As shown in Fig. 1, function blocks communicate with each other via event and data connections. In the SOA view, one event connection with associated data connections between two function blocks is considered as the message pattern. Each event connection is registered as a message type in the service contract. In the message content, all the variables associated with the event by a standard WITH declaration in the FB-type definition are encapsulated.

In the general SOA implementation, each message is defined as a two-way communication where response data could be sent back to service requesters. This is also not necessary in the IEC 61499 mapping, as event and data connections are one-way communication only. Response messages will be considered as new request messages from service providers back to service requesters.

Message types are defined in each service contract. The *IS-PART-OF* relation between services and messages types can be defined by *sm* function

$$\mathrm{sm} : \mathrm{MSGType} \rightarrow S$$

where *MSGType* is a set of message types used in the SOA.

Every message type $\mathrm{MSG} \in \mathrm{MSGType}$ consists of one request message type (MSGreq) and any number $i >= 0$ (0, 1 or more) of response message types (MSGres)

$$\mathrm{MSG} = (\mathrm{MSGreq}, \mathrm{MSGres})$$
$$i > 0 \rightarrow \mathrm{MSGres} = \{\mathrm{MSGres}_1, \ldots, \mathrm{MSGres}_i\}$$
$$i = 0 \rightarrow \mathrm{MSGres} = \oslash.$$

Definition 6: A request message type (MSGreq) comprises a message name (MSGname) mapped to event input ei and message parameters (MSGparam) mapped as DI associated with this event input in a FB interface

$$\mathrm{MSGreq} = (\mathrm{MSGname}, \mathrm{MSGparam})$$
$$\mathrm{MSGname} \leftrightarrow \mathrm{ei}$$
$$\mathrm{MSGparam} \leftrightarrow \mathrm{DI}.$$

Definition 7: A response message type (MSGres) comprises a message name (MSGname) mapped to event output eo and

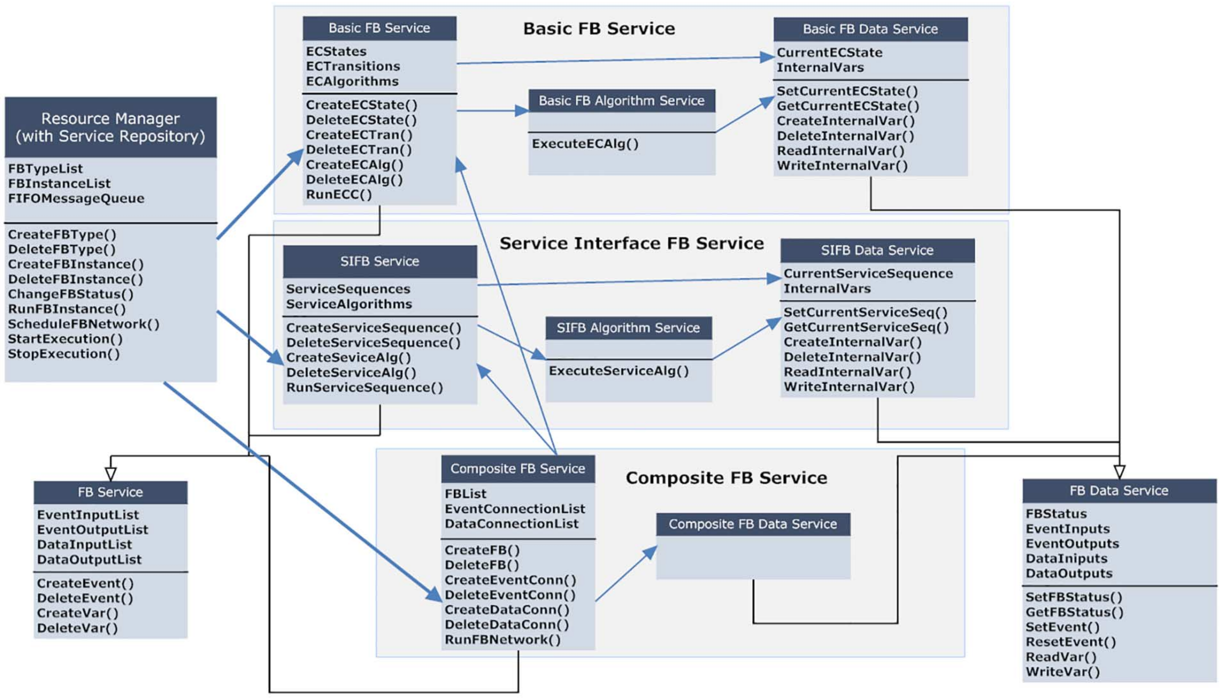


Fig. 2. SOA-based IEC 61499 runtime class diagram.

message parameters (MSGparam) mapped as DO associated with this event output in a FB interface

$$\text{MSGres} = (\text{MSGname},\ \text{MSGparam})$$
$$\text{MSGname} \leftrightarrow \text{eo}$$
$$\text{MSGparam} \leftrightarrow \text{DO}.$$

Another option is to use *adapter connections* as defined in the IEC 61499 standard for representing bidirectional communication.

The proposed formal mapping between SOA and IEC 61499 provides a guideline on how to implement the IEC 61499 service-based execution environment architecture and inter-FB communications.

V. Interoperable and Flexible IEC 61499 Execution Environment Architecture Based on SOA

In order to implement creating and deleting FB-type definitions dynamically, an SOA-based IEC 61499 runtime is proposed in this section based on the formal mapping presented in the previous section. The class diagram of the SOA-based IEC 61499 runtime architecture design is shown as Fig. 3. Each class in the diagram is implemented as a software service according to the *Definition 1*.

The key service is the resource manager. The resource manager is responsible for receiving and interpreting management commands from IEC 61499 IDEs, and in principle from other function blocks as well, and composing a response. As defined in the IEC 61499 standard, key words for management commands are *CREATE, DELETE, START, STOP, KILL, QUERY, READ, WRITE,* and *RESET*. The target element could be one of the following: *FB, Connection, FBType, AdapterType, DataType,* and *Parameter*. The resource manager is also implemented as the service repository, which keeps a list of all FB types and instances.

The basic pattern for the function block service execution model is given in Fig. 2. A function block service contains three service components: 1) predefined service (static); 2) compiled service (dynamic); and 3) data service. The core part is static service whose service contract is predefined. As shown in Fig. 2, the static service refers to the FB service, which contains interface definitions compulsory for every FB type. When a request to create a new FB type is received at the resource manager, a new FB service instance is instantiated and the service endpoint of this instance is registered in the repository list (Fig. 1) by invoking the *CreateFBType* function. When service types are no longer needed, these FB service instances and their endpoints can be deleted by calling the *DeleteFBType* function via management commands. Multiple FB instances of the same type will share the same FB service interface definition and logic implementation. Dynamic services are unique for each function block definition, e.g., logic in a SIFB or EC state algorithm in a BFB. Again, only one dynamic service interface definition and logic implementation is needed for one FB type. The last part of the FB service definition is data services. Data services are responsible for storing all FBI data such as values of all input, internal, and output variables. For every FB instance, an individual FB data service instance is instantiated when the *CreateFBInstance* function is invoked. Service endpoints of both FB types and instances will be stored in the resource manager once created. To remove an FB instance and its registration from the repository, the *DeleteFBInstance* function will be activated. Separation between logic and data ensures no wasted program memory due to duplicated logic.

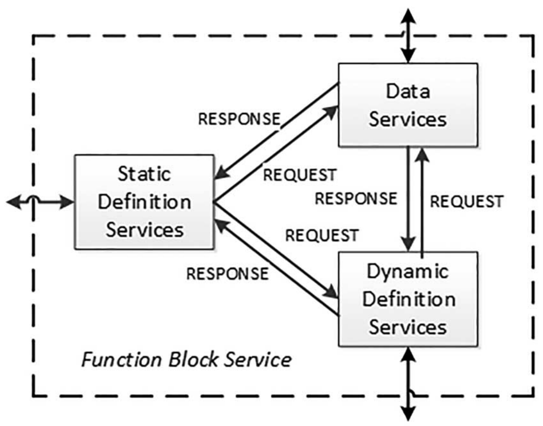


Fig. 3. Design pattern for SOA-based IEC 61499 FBs.

Following the basic pattern, a BFB service is defined as illustrated in the top row of Fig. 3. Basic FB service inherits from the base FB service, which contains the interface list. In addition, EC state declarations including state transition declarations and state actions are defined in basic FB services. EC state actions include service endpoints, which indicate the address of the assigned EC algorithm in the dynamic services of this FBT. Service endpoints of EC actions are retrieved from the resource manager as well (refer to Fig. 1).

Dynamic services of basic FBTs refer to EC algorithms and EC transition conditions. EC algorithms are normally written in one of the IEC 61131-3 programming languages. Most commonly, they are coded in the structured text (ST) and ladder diagram (LD) languages. For efficiency purposes, each EC algorithm in ST, LD, or any other language can be compiled into a function in the dynamic service instead of interpreting those languages at runtime level. When an EC transition condition is activated, related EC algorithm(s) will be invoked via the predefined service endpoint from the static service. The same approach applies to EC transition conditions as well. In order to avoid complicated interpretation of Boolean algebra, each EC transition condition is also compiled into a function. A Boolean value is returned to indicate whether the Boolean expression is true.

The base FB data service class is mainly responsible for storing FB status (IDLE, RUNNING, KILLED, or STOPPED) and variable values of FB instances. The basic FB data service extends the base FB data service with internal variables and EC states. Current EC state could be fetched from or updated to data services. Internal variable values could be accessed from, written to, and be overridden in data services.

Secondly, for service interface FB types, as seen from the second row in Fig. 2, service sequence definitions and algorithms associated with SIFB service definitions are extended. Service sequence in SIFB is a set of ordered transactions, which describes a particular functionality or process. In the dynamic definitions, all algorithms defined in service sequences are compiled and service points are stored in the repository located inside the resource manager (refer to Fig. 1). From the data services part, the only difference is that the current service sequence status is in place instead of current EC state. In some existing IEC 61499 platforms, any change required for SIFB will lead to recompiling the entire runtime. By adopting the proposed SOA-based design pattern, SIFBs could also be dynamically created, modified, or deleted. SIFBs are typically applied for communicating with *I/O* modules and external devices. Fieldbus protocols are implemented using SIFBs, which provide access to *I/O* modules. Multiple FBs are able to read input values from a single-input module SIFB. Each output can only be written by one FB due to limitation of IEC 61499 data connection: only one data connection is allowed to each DI of any function block. This ensures that *I/O* can only be accessed by PLC internally and one-to-one output mapping ensures data integrity.

Finally, for a composite FB service, there is no dynamic definition and no extension to data services required as shown in the last row of Fig. 3. In the static definitions, there is a new FB network definition, which contains all nested FBs and connections between those FBs. For every FB encapsulated in a composite FB, a new data service is created for this FB instance.

The execution of an FB network is implemented based on the event-queuing system concept from discrete-event systems theory [38]. A first-in first-out (FIFO) message queue is introduced for each IEC 61499 resource manager. The resource manager will keep monitoring this message queue after the *StartExecution()* function is invoked. When an output event is triggered by any FB, this FB acts as service requester (refer to *Definition 5*). A request message that includes this output event with associated DO is queued (refer to *Definition 7*). While the event queue is not empty, the resource manager will fetch the next request message from the FIFO (refer to *Definition 6*) and trigger the corresponding event input from the target FB as service provider by invoking *RunFBInstance()* function (refer to *Definition 4*). Composite FBs shall only be invoked by the resource manager (refer to *Definition 3*). Basic FBs (run ECC) and service interface FBs (run service sequence) could be called from either the resource manager or other composite FBs (refer to *Definition 2*). The FIFO message queue ensures no message lost between services and only one message is delivered to one FB service at the same time. An FB network is executed in sequential order according to the service sequence generated by *ScheduleFBNetwork()* function.

In the current implementation of IEC 61499 service-based runtime, WSs description language (WSDL) [37] is used for defining service contracts. Request and response messages between function block services are encoded using simple object access protocol (SOAP) [37]. Function block service types are discoverable by the resource management service using the WS-discovery protocol [37]. Management commands are interpreted and the resulting actions are handled by the resource management service as the body of SOAP messages.

VI. Dynamic Reconfiguration Example Using SOA

Many research results have already proved that IEC 61499 is capable for dynamically reconfigurable distributed automation systems at the runtime level [5], [13], [16]. In this work, dynamic reconfiguration will be only demonstrated in some enhanced features based on the service-oriented runtime—in

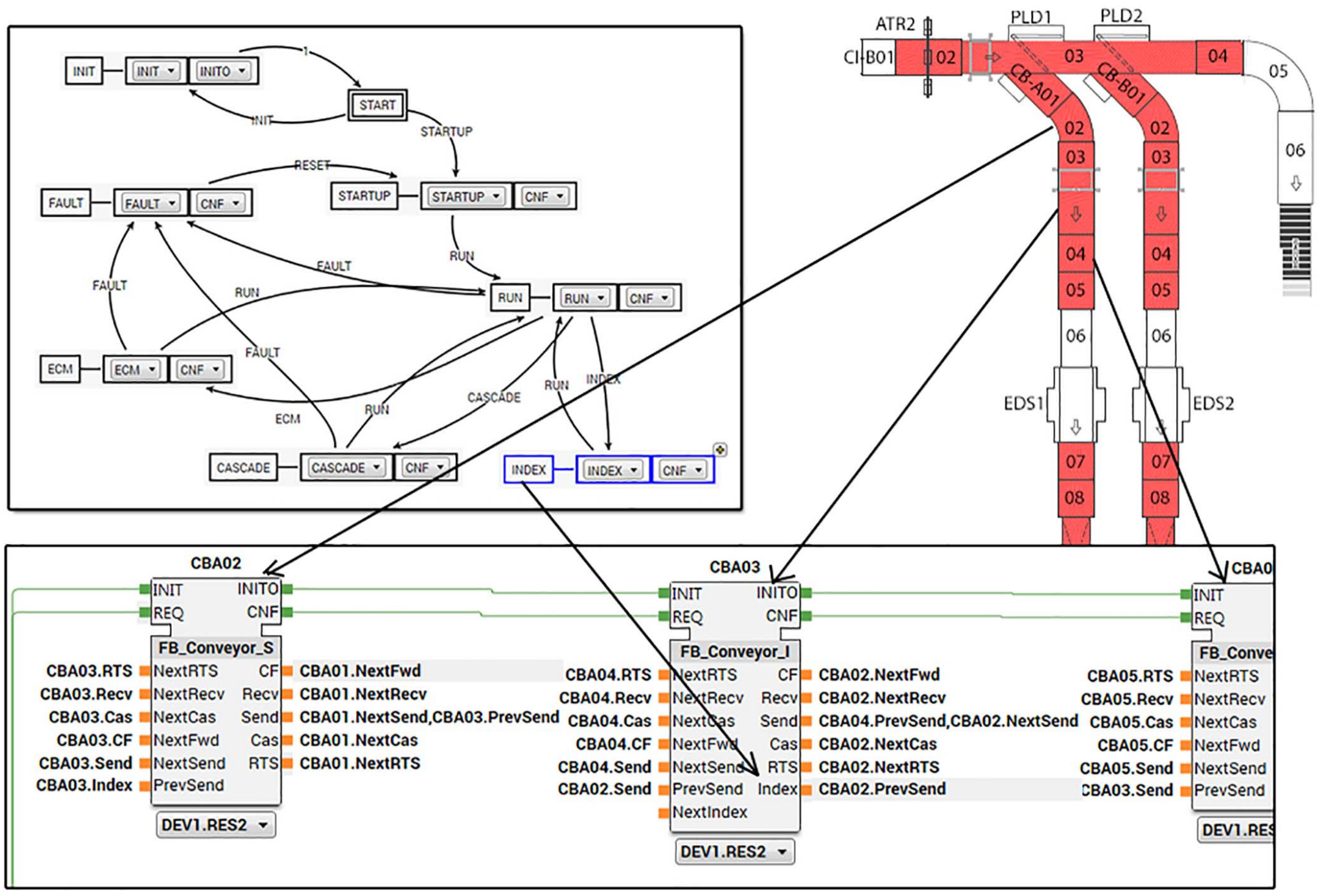


Fig. 4. BHS screening subsystem layout and FB dynamic reconfiguration implementation.

this case, delete existing FB type and create a new FB type. A screening subsystem of an airport BHS is used as the case study example here.

As shown in the device layout diagram (Fig. 4), baggage units are inducted from the in-feed conveyor lines CI-B01 to 08 into the two screening lines (CB-A and CB-B). Bags are equally distributed to two lines by plough diverters (PLD1 and PLD2). An exposure detection system (EDS) X-ray machine (EDS1 and EDS2) is the core part of each screening line. Bags cleared from the security check by EDS machines will be sent to sortation subsystems. When none of the screening lines is operable (fault) or die-back occurs due to exceeding the maximum system throughput, bags will be sent to CIB-06 for manual inspection.

During normal operation, a large volume of bags is sent to manual inspection during peak hours. In order to reduce workload for operators, "indexing" (queuing) functionality needs to be added to conveyor CB-A03 and CB-B03. The indexing functionality is enabled to accumulate bags when the downstream conveyors are stopped. A conveyor leaves indexing mode automatically when the downstream line returns to full flow.

As seen in Fig. 4, a new EC state "INDEX" is inserted into the conveyor control BFB-type *FB_Conveyor_I*. A new event input "INDEX" is added to indicate when a downstream conveyor is stopped. When this event is triggered, the EC transition condition from state "RUN" to "INDEX" is satisfied. The conveyor will switch back to RUN mode when downstream conveyors are available. Table I lists the steps of the reconfiguration sequence that adds the queuing functionality without stopping normal operation.

VII. Preliminary Analysis of SOA-Based Runtime

The case study example is tested with the BHS emulator provided by Glidepath Group [39] and the function block service runtime (FBSRT) on a Beaglebone Black board [40] with AM335x 1 Ghz CPU, 512M DDR3 RAM and 4 GB ROM. The FORTE runtime [7] will be used as the comparison reference as similar implementation (both using C++). The message throughput test is performed by repeating 10 000, 100 000, and 1 million times according to the reconfiguration sequence demonstrated in Table I.

As seen from the result of the message throughput test in Fig. 5, the average time for a SOAP message is approximately 0.4 ms on a persistent connection between services. The average time is five times more (2.4 ms for each message) if connections are closed after messages are sent. From tests for 10 000, 100 000, and 1 million messages, the measured connection establishment time is 2 ms. In the FORTE approach, function blocks are invoked by method calls between classes. As a result, there is no connection overhead. In the FBSRT approach, SOAP messages are passed over local TCP/IP stack within same device and over Ethernet between devices. Persistent connection between services is a feasible solution. However,

TABLE I
RECONFIGURATION SEQUENCE FOR ADDING INDEX FUNCTIONALITY

Steps	Management Commands
1: STOP CBA03 FB (Type:FB_Conveyor_S)	<Request ID="1" Action="STOP"> <FB Name="CBA03" Type="FB_Conveyor_S" /> </Request>
2: DELETE all Event and Data connections from/to CBA03 FB	<Request ID="2" Action="DELETE"> <Connection Source="…" Destination="…" /> </Request>
3: Delete CBA03 FB	<Request ID="3" Action="DELETE"> <FB Name="CBA03" Type="FB_Conveyor_S" /> </Request>
4: Create FB Type: FB_Conveyor_I	<Request ID="4" Action="CREATE"> <FBType Name="FB_Conveyor_I" > … // FB Type Definition for FB_Conveyor_I </FBType> </Request>
5: Create FB CBA03 (Type:FB_Conveyor_I)	<Request ID="5" Action="CREATE"> <FB Name="CBA03" Type="FB_Conveyor_I" /> </Request>
6: Create all Event and Data connections from/to CBA03 FB	<Request ID="6" Action="CREATE"> <Connection Source="…" Destination="…" /> </Request>
7: Create new data connection for Index	<Request ID="7" Action="CREATE"> <Connection Source="CBA03.Index" Destination="CBA02.PrevSend" /> </Request>
8: Start FB CBA03	<Request ID="8" Action="START"> <FB Name="CBA03" Type="FB_Conveyor_I" /> </Request>

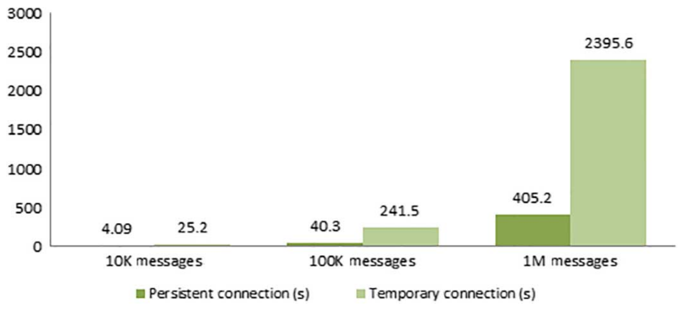


Fig. 5. SOAP message throughput test on FBSRT.

the number of persistent connections must be limited due to the large memory requirement imposed by numerous active service endpoints.

Second, the memory requirement test is performed. Two compiled function block file sizes are compared between FBSRT and FORTE as shown in Fig. 6: basic FB—*FB_Conveyor_I* and composite FB—*FB_EStopZone*. The basic FB file size in FBSRT is twice as large as the FORTE version. SOAP message send and receive functions must be embedded into every FBT in FBSRT. FBTs are separated in individual files, which can be created or deleted. FORTE compiles all files into one single executable file, which is more efficient on memory consumption; however, this reduces flexibility. Nested FBs in a CFB are flattened to individual software services, so no compiled file is required for CFB in FBSRT. The CFB type only contains its service call sequence and interface data. Overall, the FORTE version for two files is still 19.3K smaller compared to the FBSRT version (48.7K vs. 68K).

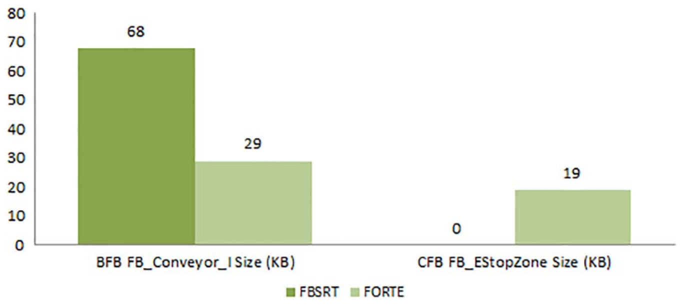


Fig. 6. Size for compiled FB files on FBSRT and FORTE.

To conclude, FORTE is designed for small devices with limited memory and computing power and the tightly coupled and minimal communication overhead architecture fits well for its scope. FBSRT demonstrates a qualitatively new level of flexibility with slightly higher hardware performance requirements than FORTE. The SOA-based structure ensures flexibility and interoperability for future-proof automation systems.

VIII. CONCLUSION AND FUTURE WORK

With the goal of improving reconfigurability, flexibility, and interoperability of distributed automation systems, the feasibility of applying the SOA into the industrial automation domain has been investigated. Based on formal definitions for implementing SOA with the IEC 61499 standard presented, any element from all FB types as well as instances could be dynamically created or deleted without interrupting normal execution in the SOA-based IEC 61499 runtime. Original source code could also be retrieved from the runtime in case the original code is missing or out-of-date. Enhanced interoperability enables platform-independent external services (from other automation systems or even Internet of Things) to communicate with FBSRT using standard WSs protocols.

Continuing from this work, the plug-and-play feature using service discovery protocol will be further investigated. Autonomic service management will be introduced in order to achieve self-manageable and adaptive systems. Rule-based configurable execution behaviors will be defined in order to support various existing IEC 61499 execution semantics and performance comparison. Finally, the optimization of SOA at the runtime level will be investigated to reduce memory consumption and communication overhead.

REFERENCES

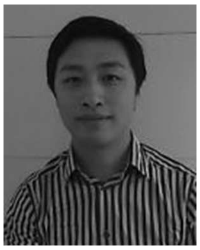
[1] *IEC 61131-3:2013, Programmable Controllers–Part 3: Programming Languages*. Geneva, Switzerland: International Electrotechnical Commission, 2013.

[2] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1234–1249, Aug. 2013.

[3] W. Dai, V. Vyatkin, and J. Christensen, "Applying IEC 61499 design paradigms: Object-oriented programming, component-based design, and service-oriented architecture," in *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*. Boca Raton, FL, USA: CRC Press, 2015.

[4] *Function Blocks: International Electrotechnical Commission*, Geneva, Switzerland, International Standard IEC 61499-1:2012 et seq.

[5] T. Strasser and R. Foschauer, "Autonomous application recovery in distributed intelligent automation and control systems," *IEEE Trans. Syst. Man Cybern. Part C, Appl. Rev.*, vol. 42, no. 6, pp. 1054–1070, Nov. 2012.
[6] FBDK. (2005). *Function Block Development Kit/FBRT–Function Block Runtime* [Online]. Available: http://www.holobloc.com/
[7] (2011). *4DIAC-IDE/FORTE: An Open Source IEC 61499 IDE and Runtime* [Online]. Available: http://www.fordiac.org
[8] (2009, Jun.). *nxtControl GmbH, nxtStudio and nxtRT61499F—Next Generation Software for Next Generation Customers* [Online]. Available: http://www.nxtcontrol.com/
[9] (2005). *ISaGRAF Workbench and Runtime* [Online]. Available: http://www.isagraf.com
[10] J. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard software tools and runtime platforms," presented at the ISA Automation Week: Asset Performance, 2012.
[11] J. Yan and V. Vyatkin, "Distributed software architecture enabling peer-to-peer communicating controllers," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2200–2209, Nov. 2013.
[12] BlokIDE, *Model-Driven Engineering Design Environment* [Online]. Available: http://www.timeme.io
[13] W. Lepuschitz, A. Zoitl, and M. Vallee, "Toward self-reconfiguration of manufacturing systems using automation agents," *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 52–69, Jan. 2011.
[14] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Trans. Ind. Informat.*, vol. 1, no. 1, pp. 62–70, Feb. 2005.
[15] G. Koumoutsos and K. Thramboulidis, "A knowledge-based framework for complex, proactive and service-oriented e-negotiation systems," *Electron Commerce Res.*, vol. 9, pp. 317–349, 2009.
[16] A. Zoitl, *Real-Time Execution for IEC 61499*. Research Triangle Park, NC, USA: ISA, ISBN: 978193439-4274, 276 pp., 2009.
[17] T. Strasser, M. Rooker, G. Ebenhofer, and A. Zoitl, "Standardized dynamic reconfiguration of control applications in industrial systems," *Int. J. Appl. Ind. Eng.*, vol. 2, no. 1, pp. 57–73, 2014. doi: 10.4018/ijaie.2014010104.
[18] Q. Zhu, Y. Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli, "Optimizing the software architecture for extensibility in hard real-time distributed systems," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 621–636, Nov. 2010.
[19] J. Lastra and I. Delamer, "Semantic web services in factory automation: Fundamental insights and research roadmap," *IEEE Trans. Ind. Informat.*, vol. 2, no. 1, pp. 1–11, Feb. 2006.
[20] I. Delamer and J. Lastra, "Service-oriented architecture for distributed publish/subscribe middleware in electronics production," *IEEE Trans. Ind. Informat.*, vol. 2, no. 4, pp. 281–294, Nov. 2006.
[21] J. Puttonen, A. Lobov, and J. Lastra, "Semantics-based composition of factory automation processes encapsulated by web services," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2349–2359, Nov. 2013.
[22] G. Candido, A. Colombo, J. Barata, and F. Jammes, "Service-oriented infrastructure to support the deployment of evolvable production systems," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 759–767, Nov. 2011.
[23] PLCopen, "PLCopen technical committee 6: XML formats for IEC 61131-3," Tech. Rep. v2.01, 80 pp., 2009.
[24] *Web Services for Management (WS-Management) Specifications*, DMTF Standard DSP 0226, 2010.
[25] C. Stoidner and B. Freislenben, "Invoking web services from programmable logic controllers," in *Proc. IEEE Int. Conf. Emerg. Technol. Factory Autom.*, 2010, pp. 1–5.
[26] R. Kyusakov, J. Eliasson, J. Delsing, and J. van Deventer, "Integration of wireless sensor and actuator nodes with IT infrastructure using service-oriented architecutre," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 34–51, Feb. 2013.
[27] M. Clavreul, S. Mosser, M. Blay-Fornarino, and R. France, "Service-oriented architecture modeling: bridging the gap between structure and behavior," *Model Driven Eng. Lang. Syst.*, vol. 6981, pp. 289–303, 2011.
[28] S. Cherif, R. Ben Djemaa, and I. Amous, "ReMoSSA: Reference model for specification of self-adaptive service-oriented architecture," in *New Trends in Databases and Information Systems*, New York, NY, USA: Springer, 2014, pp. 121–128.
[29] M. Valls, I. Lopez, and L. Villar, "iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 228–236, Feb. 2013.
[30] M. Vallee, M. Merdan, W. Lepuschitz, and G. Koppensteiner, "Decentralized reconfiguration of a flexible transportation system," *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 505–516, Aug. 2011.
[31] R. Brennan, M. Fletcher, and D. Norrie, "An agent-based approach to reconfiguration of real-time distributed control systems," *IEEE Trans. Rob. Autom.*, vol. 18, no. 4, pp. 444–451, Aug. 2002.
[32] M. Merdan, M. Vallee, W. Lepuschitz, and A. Zoitl, "Monitoring and diagnostics of industrial systems using automation agents," *Int. J. Prod. Res.*, vol. 49, no. 5, pp 1497–1509, 2011.
[33] G. Candido, C. Sousa, G. Di Orio, J. Barata, and A. Colombo, "Enhancing device exchange agility in Service-oriented industrial automation," in *Proc. IEEE Int. Symp. Ind. Electron.*, 2013, pp. 1–6.
[34] (2010). *Inico S1000 User Manual* [Online]. Available: http://www.inicotech.com/doc/S1000%20User%20Manual.pdf
[35] A. Girbea, C. Sucio, S. Nechifor, and F. Sisak, "Design and implementation of a service-oriented architecture for the optimization of industrial applications," *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 185–196, Feb. 2014.
[36] M. Sorouri, V. Vyatkin, S. Xie, and Z. Salcic, "Plug-and-play design and distributed logic control of medical devices using IEC 61499 function blocks," *Int. J. Biomechatron. Biomed. Rob.*, vol. 2, no. 2, pp. 102–110, 2013.
[37] T. Erl, *Service-Oriented Architecture: Concepts, Technology and Design*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2005, 760pp.
[38] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed., Berlin, Germany: Springer-Verlag, 2008, vol. XXIV, 776 pp.
[39] Glidepath Group. (2005). *Airport Baggage Handling System Integrator* [Online]. Available: http://www.glidepathgroup.com
[40] BeagleBoard.org Foundation. (2015). *BeagleBone Black* [Online]. Available: http://beagleboard.org/black



Wenbin Dai (GM'09–M'13) received the Bachelor of Engineering (with Hons.) degree in computer systems engineering, and the Ph.D. degree in electrical and electronic engineering from the University of Auckland, Auckland, New Zealand, in 2006 and 2012, respectively .

He is an Assistant Professor with Shanghai Jiao Tong University, Shanghai, China. He was a Postdoc Fellow with Lulea University of Technology, Lulea, Sweden, from 2013 to 2014. He was also a Software Engineer with Glidepath Limited Ð, a New Zealand-based airport baggage handling system provider, from 2007 to 2013. His research interests include IEC 61131-3 programmable logic controllers, IEC 61499 function blocks, industrial cyber-physical systems, cloud-based simulation, and software architecture in industrial automation.

Valeriy Vyatkin (M'03–SM'04) received the Ph.D. degree in computer science from the State University of Radio Engineering, Taganrog, Russia, in 1992.

He is on joint appointment as Chaired Professor of Dependable Computation and Communication Systems, Lulea University of Technology, Lulea, Sweden, and a Professor of Information and Computer Engineering in Automation, Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar at Cambridge University, Cambridge, U.K., and had permanent academic appointments with the University of Auckland, Auckland, New Zealand; the Martin Luther University of Halle-Wittenberg, Halle, Germany; as well as in Japan and Russia. His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; and distributed architectures and multi-agent systems applied in various industry sectors, including smart grid, material handling, building management systems, and reconfigurable manufacturing.

Dr. Vyatkin was the recipient of the Andrew P. Sage Award for the Best IEEE Transactions paper in 2012.

James H. Christensen received the Ph.D. degree in chemical engineering and computer science from the University of Wisconsin at Madison, Madison, WI, USA, in 1967.

He is currently with Holobloc Inc., Cleveland Heights, OH, USA. He is an internationally recognized expert in the standardization and application of advanced software technologies to the automation and control of manufacturing processes.

Dr. Christensen was the recipient of the Rockwell International Engineer of the Year and Lynde Bradley Innovation Awards in 1991 for his achievements in pioneering applications of object-oriented programming in Smalltalk, and in 2007, he received the IEC 1906 Award and Process Automation Hall of Fame membership for recognition of his accomplishments in the international standardization of programming languages and architectures for industrial automation.

Victor N. Dubinin received the Diploma degree in computer science and the Ph.D. degree in computer science from the University of Penza, Penza, Russia, in 1981 and 1989, respectively.

From 1981 to 1989, he was a Researcher, and from 1989 to 1995, he was a Senior Lecturer with the University of Penza. Since 1995, he has been an Associate Professor with the Department of Computer Science, University of Penza. In 2003, 2006, and 2010, he was awarded the German Academic Exchange Service (DAAD)-grants to work as a Guest Scientist at Martin-Luther-University, Halle-Wittenberg, Germany. He was a Visiting Researcher at the University of Auckland, Auckland, New Zealand, in 2011, and at the Lulea University of Technology, Lulea, Sweden, in 2013 and 2014. His research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.