Atmojo, Udayanto Dwi; Salcic, Zoran; Wang, Kevin I-Kai; Vyatkin, Valeriy

A service-oriented programming approach for dynamic distributed manufacturing systems

Published in:
IEEE Transactions on Industrial Informatics

DOI:
10.1109/TII.2019.2919153

Published: 01/01/2020

Document Version
Publisher's PDF, also known as Version of record

A Service-Oriented Programming Approach for Dynamic Distributed Manufacturing Systems

Udayanto Dwi Atmojo, *Member, IEEE*, Zoran Salcic, *Senior Member, IEEE*, Kevin I-Kai Wang, *Member, IEEE*, and Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—Dynamic reconfigurability and adaptability are crucial features of the future manufacturing systems that must be supported by adequate software technologies. Currently, they are typically achieved as add-ons to existing software tools and run-time systems, which are not based on any formal foundation such as formal model of computation (MoC). This paper presents the new programming paradigm of service oriented SystemJ (SOSJ), which targets dynamic distributed software systems suited for future manufacturing applications. SOSJ is built on a merger and the synergies of two programming concepts of service oriented architecture, to support dynamic software system composition, and SystemJ programming language based on a formal MoC, which targets correct by construction design of static distributed software systems. The resulting programming paradigm allows the design and implementation of dynamic distributed software systems.**

***Index Terms*—Dynamic distributed software systems, manufacturing, reconfigurability, service oriented architecture (SOA).**

I. Introduction

INDUSTRY 4.0 promotes a new breed of manufacturing systems comprising automated industrial machines extended with additional sensors and actuators in sensor and/or actuator nodes (SANs). Industrial machines and SANs are governed by software behaviors executed on embedded computers, interconnected via industrial networks or high-level dedicated protocols [1]. These manufacturing systems may require dynamic adaptation and reconfiguration during runtime (e.g., addition or removal of machines/SANs, reconfiguration of existing machines/SANs) to accommodate the continuously changing demand in production or due to failures. Equally important is safe and correct operation of all individual software behaviors as well as their composition that support complex production processes. The correctness and formal verifiability of underlying software design is of the utmost importance and one of key challenges in manufacturing systems.

Currently, there are many programming approaches used to design manufacturing systems. The IEC 61131-3 [2] and IEC 61499 [3] are the most well known. However, both standards lack formal model of computation (MoC). There have been efforts to introduce formal MoC into these standards such as [4], [5], but they lack mechanisms to handle dynamic changes. Attempts to introduce dynamic adaptations into IEC 61499 (such as [6] and [7]) and IEC 61131 (such as [8]) exist, however only partial/limited reconfiguration is enabled. Other programming technologies such as the multiagent systems (MAS) (e.g., JADE [9], JIAC [10]) and the service oriented architecture (SOA) (such as device profile for web services (DPWS) [11], Arrowhead [12]) are used in the context of dynamic systems. While MAS-based approaches are increasingly used for control in industrial automation [13], they need heavy run-time systems and lack formal foundations needed to design correct-by-design software behaviors. Similarly, despite increased interest in using SOA in manufacturing applications, they lack formal underpinnings or MoCs.

The formal MoC-based approach of SystemJ [14] enables the design of safe, correct-by-construction concurrent software behaviors and their use in distributed manufacturing systems. However, it is only suited for static distributed systems, where software behaviors are specified at the design time and do not change during system operation [15]. In this paper, a new programming approach called service oriented SystemJ (SOSJ) is presented in the context of dynamic distributed manufacturing systems. Compared to the state of the art, the novelty of this approach lies in the combination and synergy of the distinctive features of SystemJ and SOA. This paper for the first time presents a complete description of the integral SOSJ framework for dynamic distributed systems design and a comprehensive performance evaluation and comparison between SOSJ and a SOA-only based framework WS4D JMEDS [16] in handling SOA features. The paper focuses on the features that handle dynamicity of correct-by-construction software behaviors.

The rest of this paper is organized as follows. Section II describes a motivating example and identifies some key requirements of dynamic distributed manufacturing systems. Section III briefly introduces the SystemJ language. Section IV describes the SOSJ and how the SOA-based features in SOSJ can be used in dynamic manufacturing systems context. Section V presents performance benchmarks and comparisons

Manuscript received January 29, 2019; accepted May 3, 2019. Date of publication June 27, 2019; date of current version January 4, 2020. Paper no. TII-19-0316. *(Corresponding author: Udayanto Dwi Atmojo.)*

U. D. Atmojo and V. Vyatkin are with the Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland (e-mail: udayanto.atmojo@ieee.org; vyatkin@ieee.org).

Z. Salcic and K. I.-K. Wang are with the Department of Electrical and Computer Engineering, The University of Auckland, Auckland 1010, New Zealand (e-mail: z.salcic@auckland.ac.nz; kevin.wang@auckland.ac.nz).

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.2019.2919153

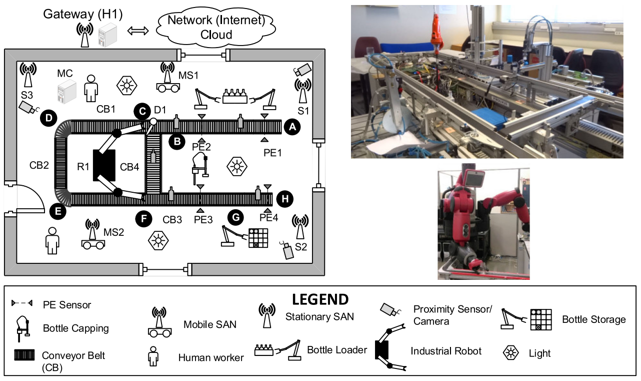

Fig. 1. Physical layout of the motivating example (left), physical setup of the production line A–H (top right), industrial robot R1 (middle right).

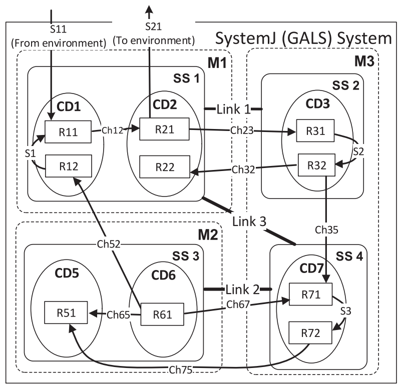

Fig. 2. Graphical illustration of SystemJ system/program.

with a SOA-only WS4D JMEDS. Finally, Section VI concludes this paper.

II. Motivating Example

An example of dynamic manufacturing system (factory) is illustrated in Fig. 1. The figure also shows a few snapshots of the physical setup of some parts of the example in our lab. The factory has a manufacturing shop floor, which utilize stationary SANs, e.g., S1, S2, and S3 to monitor the surrounding environment for security purposes and control the lighting and air conditioning systems to maintain the ambient conditions needed for production. There are multiple mobile SANs, e.g., MS1 and MS2, each equipped with sensors and mounted on an autonomous vehicle, which roam to perform security and safety checks.

The factory has industrial production stations which perform automated bottle capping and storage. Bottles are loaded onto the conveyor belt, which is equipped with photo eye sensors/PE sensors in different positions), by either of the loader stations (at point B or point A). The bottles are transported to get capped by the capping station at point G, and then stored by a storing station at H. If the conveyor CB4 or diverter D1 is not operational, the robot IR1 takes the responsibility to move bottle from C to F. If IR1 is also not operational, the conveyor CB2 can take over the transportation of bottles from C to F via D and E. A master controller MC allows human operator to monitor the production process, while Gateway H1 enables accessibility of all elements through internet. Typical manufacturing facilities have other sections, such as offices, logistics, and warehouses, where SANs are also be deployed for specific purposes.

Based on the motivating example, the programming framework should be capable of satisfying the following requirements:

1) *Concurrency*: The programming framework should be able to handle different types of concurrency in distributed setting. Software behaviors associated with different machines/stations and SANs naturally operate asynchronously with each other, while each of these behaviors may be composed of multiple mutually synchronous concurrent behaviors. This requirement is directly supported by the mechanisms of the SystemJ language.
2) *Reactivity and Determinism*: Typically, software behaviors need to respond to many incoming events from the environment and repeatedly, thus being reactive. Also, reactions on the same sequences of input events should be deterministic, particularly important to safety-critical applications. This requirement is directly supported by the mechanisms of the SystemJ language.
3) *Dynamicity*: In the presented example, dynamic changes in presence of software behaviors is common due to, e.g., addition/removal of machines, failures, and the mobility of SANs. Thus, the underlying programming framework should support handling dynamic changes in terms of presence of software behaviors and enable their composition as the changes in the system happen. This requirement is supported by SOA features incorporated into the framework.
4) *Functional Correctness*: All operations should be performed correctly to support production process. The programming framework ideally should be based on formal foundations to allow the design of correct-by-design and verifiable software behaviors. This requirement can be satisfied by underpinning programming framework by a formal MoC. This requirement is supported by underpinning SOA-based features implemented by formal mechanisms of the SystemJ language.

These requirements have been the main guideline when defining the SOSJ framework.

III. Brief Introduction to SystemJ

SystemJ is a system-level programming language amenable for designing concurrent and distributed systems. It is based on globally asynchronous locally synchronous (GALS) MoC [17], which guarantees determinism and functional correctness within individual formally verifiable software behaviors. Fig. 2 shows a graphical representation of a SystemJ program, where mutually synchronous behaviors called *reactions* are grouped into mutually asynchronous behaviors called *clock domains* (CDs). Reactions can contain child reactions, thus allowing

TABLE I
SystemJ Kernel Statements

Statement	Description
p1;p2	p1 and p2 in sequence
pause	Consumes a logical instant of time (a *tick* boundary)
[input] [output] [type] signal S	Declaring a pure or valued signal
emit S [(exp)]	Emitting a signal with a possible value
while(true) p	Temporal loop
present(S){p1}else{p2}	If signal S is present do p1 else do p2
[weak] abort ([immediate] S) {p}	Pre-empt if S is present
[weak] suspend ([immediate] S) {p}	Suspend for 1 tick if S is present
trap (T) {p}	Software exception
exit T	Throw a software exception
p1 || p2	Run p1 and p2 in lock-step parallel
[input] [output] [type] channel C	Declaring input or output channel
send C[(exp)]	Sending data over the channel
receive C	Receiving data over the channel
#C	Retrieving data (Java object) from a valued signal or channel

hierarchical composition of synchronous concurrent behaviors. CDs in a SystemJ program advance at their own speeds defined by independent logical clock (*ticks*) with detailed descriptions described in [14]. While any CD must run on the same computer/machine, CDs belonging to the same program can be deployed on any sufficient number of machines connected by any type of interconnect (e.g., networked machines, distributed setting).

SystemJ has its own syntax and statements (see Table I) for writing a GALS program and in addition allows the use of object-oriented features of Java language. Interactions between CDs and the program's environment (anything outside SystemJ program, e.g., another program or physical world) and each with the other are facilitated by signal and channel abstract mechanisms, which can carry data encapsulated in any Java objects. Channels and signals are mapped onto various physical interfaces and communication methods, which are handled by the SystemJ runtime system (RTS), refraining the need for programmers to deal with implementations of physical interfaces/communication methods. One or more CDs handled by the same RTS and executed on the same Java virtual machine (JVM) belong to the same SystemJ *subsystem* (SS). A SystemJ program can be partitioned to a number of subsystems residing on different computing machines. CDs residing in different subsystems communicate through the channels where all data is exchanged through physical interfaces/mediums referred to as *links* (can be implemented as, e.g., shared memory, TCP/IP, controller area network, etc.). The example of a program shown in Fig. 2 comprises four subsystems (SS1–SS4), deployed and distributed across three different machines (M1–M3), with links connecting pairs of subsystems.

IV. SOA + SystemJ = SOSJ

While suited to design distributed systems, SystemJ lacks the support to handle dynamic changes in the number of software behaviors of the same program. Meanwhile, SOA paradigm introduces loose coupling, allowing for dynamic management of software behaviors. This section describes a combination of the SOA and SystemJ that creates a new programming framework called SOSJ [18], which supports dynamic changes of the number of functionally correct software behaviors that can act as the providers and consumers of different types of services (physical, logical, or data services).

SOSJ MoC is a "dynamic" GALS MoC. Although not fully formally defined, we present an informal introduction into this MoC. The SystemJ GALS MoC is for static software systems which have fixed number of software behaviors (CDs) defined at compile time. SOSJ MoC is an extension of SystemJ's MoC, where new CDs can enter and/or exit a software system at any feasible point in time (at the boundary of the logical ticks of CDs in any subsystem). Whenever a number of CDs in the system changes, the system enters its new "global state," which is represented with the list and number of all CDs and interconnecting channels in the system. However, between this state and the next state (represented with different set of CDs in the system), the system is considered static, thus theory related to static GALS systems is applicable to the system until it stays in the same state.

A. Software Behaviors: SOA and SystemJ Perspective

The SOA paradigm refers to software behaviors as service entities, while in SystemJ, CD is a software behavior composed from one or more synchronous reactions. Thus, a reaction may take different roles, of service consumer, or provider, or both. Since the synchronous reaction [19] guarantees determinism of execution and is driven by logical clock of the CD it belongs to, it cannot be considered an exact equivalent of service entity of the SOA paradigm. However, the reaction as a service provider and/or consumer inherits all benefits of synchronous MoC it is based upon.

Being based on SystemJ, service interfaces in SOSJ adopt the communication mechanisms of SystemJ. Service invocation and provision between reactions in the same CD is done through signals, while it is performed via channels for reactions belonging in different CDs. Although how reactions are allocated to the CDs is a design decision, this decision determines that SystemJ communication mechanisms should be used to perform service invocations and receive the service responses.

Based on the role of reactions that are composed in a CD, we illustrate service invocations by scenarios, as shown in Fig. 3, as follows.

1) CD is composed of one or more reactions which all have one role only (service consumer or service provider). An example of this scenario is shown in Fig. 3(a), where reactions R11 and R12 have service consumer role, while R21 and R31 have service provider role. Possible service invocation scenarios include, e.g., R11 can invoke (request) services offered by R21 via channel Ch1121 and receive results (responses) via channel Ch2111.
2) CD is composed of reactions with a mix of service provider and service consumer roles. An example of this scenario is shown in Fig. 3(b), where reactions R41 and R43 have service provider role and R42 and R44 have service consumer role. In this example, R51 can invoke the services offered by R41 via Ch5141 and receive

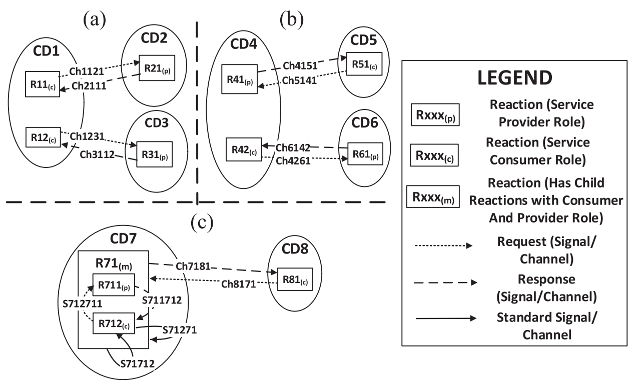
(a)
(b)
(c)
CD1
CD2
CD3
CD4
CD5
CD6
CD7
CD8
LEGEND
Reaction (Service Provider Role)
Reaction (Service Consumer Role)
Reaction (Has Child Reactions with Consumer And Provider Role)
Request (Signal/Channel)
Response (Signal/Channel)
Standard Signal/Channel

Fig. 3. Illustrations showing examples of scenario 1), 2), and composite services.

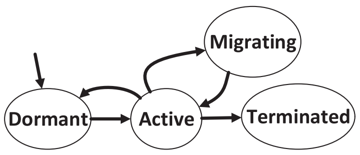

Fig. 4. CD macrostates and their transitions.

results via Ch4151, while R43 receives request from R44 through signal S4443 and replies with response through signal S4344.

Also, it is possible that reactions in a CD are connected with other reactions in the same CD (via signals) or other CDs (via channels) to produce new services created by composition of their services. In scenario 2), individual reactions in a CD can have mixed service producer and consumer roles, since they may comprise child reactions. An example of composite service is shown in Fig. 3(c), where reaction R71 acts as an orchestrator that forms a composite service through the composition of itself with services from R712 and R711 via signals and process data acquired from R712 through signal (S71271). As a service consumer, R81 can invoke the composite service via Ch8171 and receives results via Ch7181.

With the adoption of the SOA paradigm, the pairings of channels used by reactions to invoke and receive service in different CDs must be reconfigurable to allow dynamic interactions/aggregations of services from reactions in different CDs and in case of change of presence of software behaviors. It should be noted that despite the aforementioned changes in the nature of CDs, a SystemJ subsystem is still considered as a "container" that encapsulates CDs handled by the same RTS and JVM and semantics of signal is unchanged.

In addition, the features that allow the creation, termination, and mobility of service entity during runtime in such dynamic, service-oriented systems are crucial. For this purpose, CDs are enhanced with a set of "higher-level" states, referred to as macrostates, as shown in Fig. 4, with semantics as described in the following.

1) *Dormant*: The CD code is present in the machine, however it is not yet schedulable. The CD moves to the active state once the CD descriptor, signals, channels, and service descriptions are created.

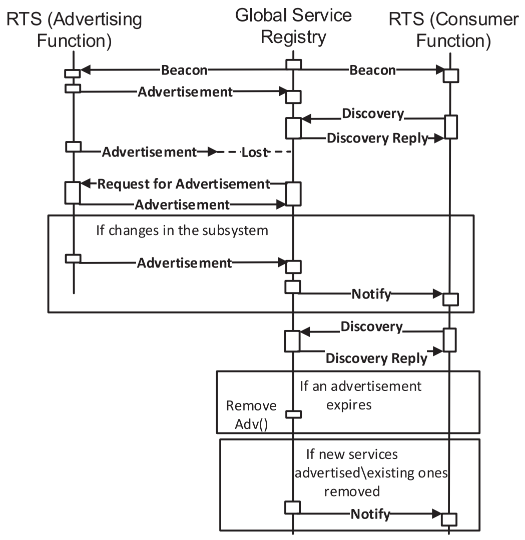

Fig. 5. Interaction between the global service registry and SOSJ RTS.

2) *Active*: The CD is created and schedulable. If migration is invoked, it moves to *migrating*. The CD goes to *terminated* after the command to terminate is issued and the CD is erased from the memory, or moves to *dormant* when the command to "update" (e.g., modify with new code) is issued.

3) *Migrating*: The CD can move from one subsystem to another, and is considered to be in the migrating state until the migration process is completed, which then it moves to the *active* state in the destination subsystem.

4) *Terminated*: The CD is considered in the terminated state if the CD is erased from the memory.

More in-depth discussion about CD macrostates and their transitions is not within the scope of this paper and readers are referred to [20].

B. SOSJ Framework

The SOSJ framework is developed by extending the original SystemJ with CD macrostates and with SOA features of loose coupling and dynamic composition. The global service registry (GSR) in SOSJ is maintained as a stand-alone application outside of the SOSJ program. Besides storing the service description of advertised services, GSR also handles other SOA functionalities. service description is written in XML and is stored in JavaScript object notation format by the SOSJ framework.

The interaction between the GSR and the SOSJ RTS through the SOA functionalities is shown in Fig. 5, with SOA functionalities as follows.

1) *Beacon*: Transmitted by the GSR application and refreshed periodically to inform all parties of its presence.

2) *Advertisement*: Sent periodically, before it expires; contains a list of services of all CDs with service provider role residing in a particular subsystem.

3) *Notify*: Transmitted after the GSR receives an advertisement and indicates that creation, termination, and

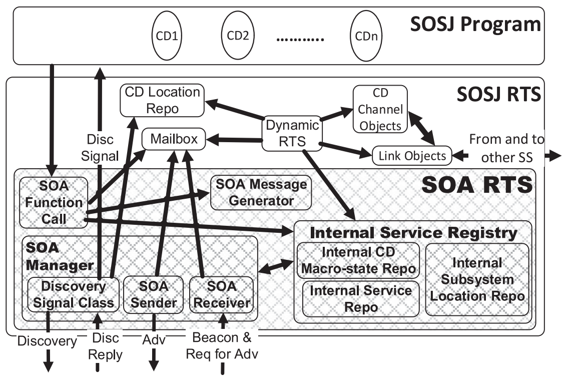

Fig. 6. SOA RTS within SOSJ RTS.

mobility occurs in a particular subsystem, or after new services are advertised to the GSR or existing services are removed.

4) *Request for Advertisement*: Transmitted by GSR when an advertisement is approaching its expiry. If an advertisement is not refreshed within a certain time or "expired," the subsystem associated with the advertisement is considered unavailable, and the service descriptions and the information regarding macrostate of CDs associated with the advertisement are removed from the GSR.
5) *Discovery*: Transmitted to the GSR to obtain the list of advertised services.
6) *Discovery Reply*: Transmitted by the GSR as a reply to Discovery.

The part of the SOSJ RTS that provides SOA support (diamond-patterned) is shown in Fig. 6. The SOA manager comprises threads that receive beacon, advertisement, discovery, discovery reply, request for advertisement, and notify and transmit all of them except Beacon. The SOA message generator provides the function to generate SOA messages. The internal service registry represents the data structures in the RTS which store service descriptions, CD macrostates, and the physical location of subsystems. The SOA function Calls/application programming interfaces (APIs), which can be used in addition to SystemJ statements and Java as integral part of SystemJ are shown in Table II. Table I includes also APIs that allow channel reconfiguration, which is handled by the SOSJ RTS when a particular output channel needs to bind dynamically with another input channel.

In addition, SOSJ provides built-in mechanisms and APIs that allow programmers to deal with dynamic creation, suspension, resumption, termination, and migration of CDs and associated services. More details are available in [21].

As a guideline, a CD with service consumer role needs to use two dedicated SOSJ signals, called SOSJDisc and SOSJDiscReply, for transmitting discovery and receiving discovery reply, respectively. To invoke service offered by different CD via channel, the CD needs to utilize a pair of channels, one input and one output channel. The output channel is used to send service invocation message, while the input channel is used to receive reply from the provider CD. Similarly, a CD with service provider role also needs to utilize a pair of channels, to receive

TABLE II
SOSJ SOA PROGRAMMING CONSTRUCTS

Function Call	Description
SOSJ.ConfigureInvocChannel()	Makes a request for channel reconfiguration to the RTS for service invocation via channel
SOSJ.GetInvocChannelReconfigStat()	Obtains the status of channel reconfiguration process (successful or not) from the RTS
SOSJ.CreateChanInvReqMsg()	Generates service invocation message for service invocation via channel
SOSJ.CreateSigInvReqMsg()	Generates service invocation message for service invocation via signal
SOSJ.CreateSigInvRespMsg()	Generates a response to service invocation message via signal
SOSJ.StoreService()	Stores service description into the internal service registry in the RTS.
SOSJ.CreateChanInvRespMsg()	Generates a response to service invocation message via channel
SOSJ.GetAction()	Gets the 'action' name included in service invocation message
SOSJ.GetData()	Gets any data/value included in service invocation message
SOSJ.SetCDServVisib()	To include/exclude service description of services of a CD for Advertisement

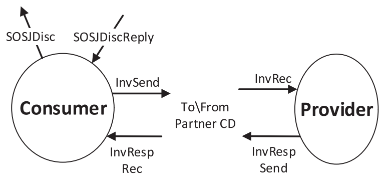

Fig. 7. Graphical illustration of the "template" of consumer and provider CD.

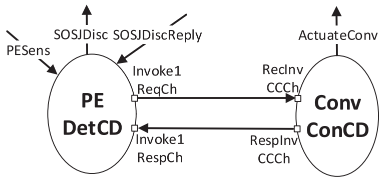

Fig. 8. Graphical illustration of the CDs described in listing 1.

invocation request and transmit the reply. An example ("template") is shown in Fig. 7, with the consumer CD ("consumer") and the provider CD ("provider").

C. Code Example—Service Invocation Using SOSJ

To show how the programming constructs of SOSJ can be utilized, listing 1 presents a code snippet of parts of the example shown in Fig. 1. PEDetCD (line 1–35) is a CD that obtains the PE (photo eye) sensor reading, e.g., PE1, to detect workpiece on the conveyor and invoke the conveyor service. ConvConCD (line 36-54) is a CD which governs the conveyor service, e.g., CB1. The graphical illustration of both is shown in Fig. 8.

First, PEDetCD obtains the sensor reading from the PE sensor via signal PESens (line 10). If the sensor indicates that a workpiece is detected (line 12), the CD triggers service discovery by emitting SOSJDisc signal (line 13), the dedicated SOSJ signal used for transmitting discovery, and then waits to

```
1   PEDetCD(
2         output String signal SOSJDisc;
3         input String signal SOSJDiscReply;
4         output String channel Invoke1ReqCh;
5         input String channel Invoke1RespCh;
6         input String signal PESens;
7   )->{
8         {
9   //...further variable instantiation and behaviour description…//
10              await (PESens);
11              String peval=(String)#PESens;
12              if(peval.equals("high")){
13                   emit SOSJDisc;
14                   await (SOSJDiscReply);
15                   String serv = (String)#SOSJDiscReply;
16                   Hashtable mtRes = DoMatching(serv, Loc);
17       String actNameConv = mtRes.get("actionName").toString();
18       String servCDName = mtRes.get("servCDName").toString();
19    String servChanName = mtRes.get("servChanName").toString();
20    //...further service matching implementation...//
21   boolean reconfstat1 = SOSJ.ConfigureInvocChannel ("PEDetCD",
"Invoke1ReqCh", servCDName, servChanName);
22     if(reconfstat1){
23       pause;
24       boolean reconfstat2 = SOSJ.GetInvocChannelReconfigStat
("PEDetCD", "Invoke1ReqCh");
25      if(reconfstat2){
26       String ChReqMsg = SOSJ.CreateChanInvReqMsg("PEDetCD",
"Invoke1RespCh", actNameConv);
27            send Invoke1ReqCh(ChReqMsg);
28            receive Invoke1RespCh;
29            //...further behaviour description...//
30            }
31         }
32     //...further behaviour description...//
33       pause;
34     }
35  }
36  ConvConCD(
37          output String signal ActuateConv;
38          input String channel RecInvCCCh;
39          output String channel RespInvCCCh;
40    )->{
41         {
42        while(true){
43          receive RecInvCCCh;
44         String recMsg = (String)#RecInvCCCh;
45         //...further behaviour description...//
46          emit ActuateConv(params);
47    SOSJ.ConfigureInvocChannel("ConvConCD","RespInvCCCh",
recMsg);
48          pause;
49       String rMsg = SOSJ.CreateChanInvRespMsg(recMsg,"ACK");
50          send RespInvCCCh(rMsg);
51          pause;
52       }
53     }
54   }
```

receive discovery reply (which will be sent by the global service registry) via signal SOSJDiscReply, the dedicated SOSJ signal for receiving discovery reply (line 14). Once the discovery reply is received, the discovery reply message is extracted (line 15) and the CD performs service matching to find the conveyor service (line 16). In the example, the CD uses a Java method named "DoMatching" for service matching, however its implementation is application-specific, e.g., the method looks for the conveyor to actuate based on the location of the detected workpiece, i.e., position of PE1 sensor (represented as "Loc" in line 16). Once the service is found, the CD makes a request to perform channel reconfiguration to bind the Invoke1ReqCh with the input channel used by the ConvConCD to receive request message (i.e., RecInvCCCh) using the SOSJ programming construct (line 21).

Once the channel reconfiguration request is passed to the SOSJ RTS (lines 21 and 22), and if the status of the channel

TABLE III
COMPUTATIONAL CAPABILITY OF EMBEDDED CONTROLLERS

Platform	Computational Capability
BB	1 GHz single core ARM Cortex A8, 512 MB RAM
RPI2B	900 MHz quad core ARM Cortex A7, 1 GB RAM
Altera DE1	800 MHz dual core ARM Cortex A9, 1 GB RAM

reconfiguration indicates that the reconfiguration is successful (line 24), the CD proceeds to generate service invocation request message (line 26) and then transmits it to the ConvConCD via channel Invoke1ReqCh (line 27). Then, the CD waits for a response message sent from the ConvConCD via channel Invoke1RespCh (line 28) which indicates the completion of the conveyor actuation.

V. PERFORMANCE EVALUATION AND COMPARISONS

A. Handling Channel Reconfiguration for Service Invocation

Two scenarios are considered. Both involve one CD with a reaction (consumer role) attempting to invoke the service offered by a reaction (provider role) in another CD, in this case, the consumer CD invoking the photo eye sensor (e.g., PE1 in Fig. 1) service offered by another CD. The Beaglebone black (BB) platform, raspberry Pi 2B (RPI2B) and ARM processor in altera/intel cyclone V device on Terasic DE1 board (DE1) are chosen as the execution platforms, as given in Table III, to compare the performance of channel reconfiguration. The first scenario considers both CDs belonging to the same subsystem and running on the same machine, while in the second scenario, each CDs belongs to two different subsystems deployed on separate machines. The second scenario utilizes a 100 MB/s IPv4-based Ethernet network with one network router.

Prior to starting the measurement, 20 500 executions of the processes (in this case, channel reconfigurations) are performed to allow for the "warmup" of the JVM, i.e., for the just-in-time compiler to compile the byte-code of the executed functionalities. After this, the measurement is performed on 1000 and 2000 channel reconfigurations for each scenario. The average time to perform reconfiguration over 10 runs is calculated. The time needed to perform channel reconfiguration is measured using Java timer ([21] describes the reconfiguration implementation). The results for the two scenarios are shown in Figs. 9 and 10, respectively.

It can be noticed that the overall average time required to perform channel reconfiguration within the same subsystem is significantly lower than in subsystems distributed in the network. The time for channel reconfiguration is spent on three different operations: to send a request to perform channel reconfiguration to the RTS to query whether the partner input channel (which will receive the service invocation request message) is available to be bound to the corresponding output channel used to send the service invocation request message; to obtain the acknowledgment to proceed with the channel reconfiguration; and allow RTS of the involved subsystem to modify the corresponding channel pairing and its mapping accordingly.

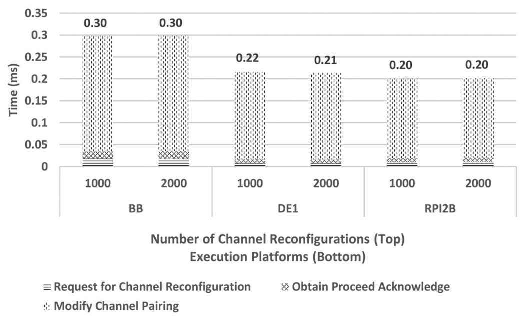

Fig. 9. Average time required to perform channel reconfigurations in case of the single subsystem.

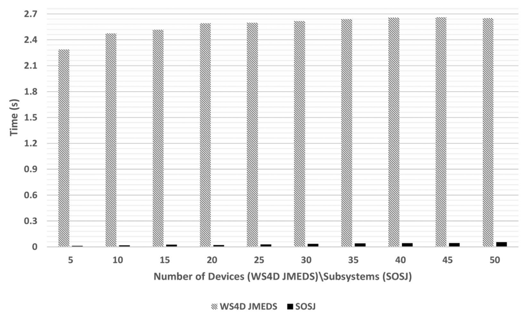

Fig. 11. Comparison of average RT time of WS4D probe—probe match and SOSJ discovery—discovery reply.

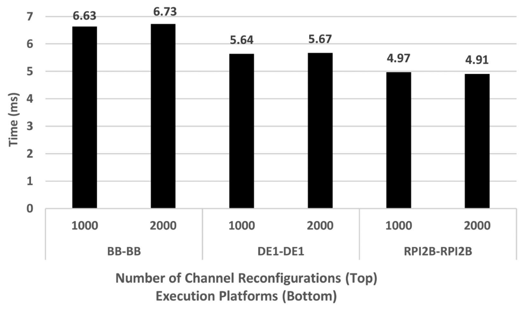

Fig. 10. Average time required to perform channel reconfigurations in case of distributed subsystems.

For single subsystem, as seen in Fig. 9, most of the time to perform channel reconfiguration is spent on the "modify channel pairing," which occurs during the time when the RTS executes the SystemJ reactive interface functions at the end of current tick of a CD, also called the house-keeping time.

For distributed subsystems, the results are presented as the sum of the time to perform three operations because the time spent on the first and second operation is significantly much smaller than the time spent on the third operation. In distributed system case, significantly longer time is required because the RTSs of the corresponding subsystems need to exchange data on parameters needed for channel reconfiguration over the network. Figs. 9 and 10 also indicate the dependence of the overall performance on the performance of three execution platforms.

B. Comparisons With WS4D JMEDS

The performance of SOSJ is compared with a SOA-only based approach of WS4D JMEDS (which adopts the device profile for web service/DPWS standard [22]) in handling SOA functionalities and service invocation. The WS4D JMEDS is chosen because it is a popular SOA-based approach used in other related research [23] and also based on Java, so the performance is similarly affected the use of the JVM. The same procedure to "warmup" the JVM as explained in Section V-A for both SOSJ and WS4D JMEDS is applied.

1) Service Discovery Handling: In service discovery, the DPWS standard has the equivalent of SOSJ's discovery and discovery reply, namely probe and probe match, respectively. A probe message is transmitted (multicast) by a consumer to find available DPWS *devices* (entities that provide services), and upon receiving the Probe message, DPWS devices respond with probe match message (unicast) back to the consumer. The benchmarks consider a scenario of 5-50 SOSJ subsystems (with three conveyors, one diverter, and one photo eye CDs, in total five CDs in each subsystem) with 25 s advertisement expiry time in SOSJ, and 5–50 DPWS devices, in which each set of five DPWS devices represents three conveyors, one diverter, and one photo eye sensor. The 5–50 subsystems and DPWS devices are deployed on 1–5 BB platforms, with each BB running a maximum of ten subsystems (in SOSJ) or ten DPWS devices (in DPWS), while the SOSJ global service registry application runs on another BB. Discovery/Probe message is sent from and discovery reply/probe match is received by a PC, which logs the timestamps of when the discovery message is sent and the discovery reply message is received (in SOSJ) and when the probe message is sent and probe matches, sent from all DPWS devices, are received (in WS4D JMEDS), using Wireshark packet sniffer application. It should be noted that SOSJ subsystems typically contain multiple CDs, where a single CD typically represents a single manufacturing device (e.g., conveyor device, photo eye device, etc.), while in DPWS, a DPWS device represents a single manufacturing device. Thus in the scenario, there are 25–250 CDs since each subsystems contains 5 CDs (i.e., 25–250 manufacturing devices), in comparison to only 5–50 DPWS devices (i.e., 5–50 manufacturing devices) in WS4D JMEDS.

The average round trip time (avg RT Time) of 100 discovery–discovery reply (in SOSJ) and probe–probe Match (in WS4D JMEDS) is calculated. Note that in contrast to SOSJ that has a separate service registry application that receives discovery from clients and transmits discovery reply to clients (centralized discovery), in WS4D JMEDS Probe message is transmitted via multicast (one to many), with potentially multiple probe match replies coming from multiple devices (decentralized discovery). The results are shown in Fig. 11. For the considered scenario, SOSJ has lower avg RT Time compared to WS4D JMEDS, which is attributed to the WS4D JMEDS runtime that waits for all DPWS devices to send probe match, while in SOSJ, the client only waits for the GSR to send discovery reply. While having "discovery proxy" is possible in WS4D JMEDS to reduce message traffic, only an example implementation of discovery proxy is provided in the framework and programmers is expected

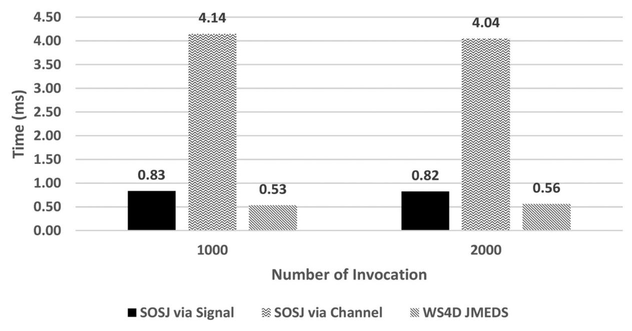

Fig. 12. Average round trip time of invocation (single machine setting) in SOSJ and WS4D JMEDS.

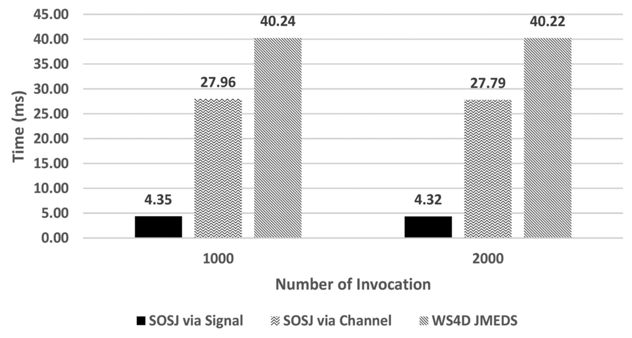

Fig. 13. Average round trip time of invocation (distributed machine setting) in SOSJ and WS4D JMEDS.

TABLE IV
SUMMARY OF COMPARISONS

Features	SOSJ	WS4D JMEDS
Functional Correctness	Supported by underlying formal GALS MoC	No underlying formal semantics or MoCs
Reactivity	Supported	Limited
Deterministic Services	Supported	Not Supported
Concurrency	Asynchronous and synchronous concurrency	Asynchronous concurrency only
Abstracted Communication with Environment	Supported	Limited
Dynamic scenario handling	Dynamic creation, suspension, resumption, termination, and migration	Dynamic creation and termination only
Service invocation	Via channels or signals (shared memory or network). Channel rendezvous supports reliability	Always performed over the network. No inherent handshake mechanism
Memory footprint	787 kB	1014 kB

to implement their own discovery proxy depending on their applications.

2) Service Invocation Handling: The following two benchmarks are run to compare the performance of SOSJ and WS4D JMEDS in handling service invocation. The benchmarks consider the example of consumer CD invoking the PE sensor service offered by a provider CD: (i) both consumer and provider reside on a single BB; and (ii) the consumer and provider reside on two different networked BBs. The benchmark considers 1000 and 2000 consecutive service invocations. In SOSJ, service invocation in benchmark (i) is achieved via channel and via signal for invocation within one CD, while in benchmark (ii), it is performed via channel involving CDs in different SOSJ subsystems and via signal involving CDs in different SOSJ programs. In WS4D JMEDS, in benchmark (i), both provider and consumer are handled by the same WS4D JMEDS runtime environment instance, while in benchmark (ii), the provider and consumer are handled by their respective runtime environment instances in each platform. The total round trip time to perform each run of 1000 and 2000 service invocations is measured using Java timer and the average is calculated over ten runs. Then, the average round trip time for single service invocation (of 1000 and 2000 invocations) is calculated and the results are shown in Figs. 12 and 13.

As shown in Fig. 12, in the first benchmark, WS4D JMEDS achieves lower average time compared to SOSJ in service invocation in case of single machine scenario. In SOSJ, service invocation via signal has lower performance overhead compared to invocation via channel as rendezvous/handshake mechanism is not used, however data delivery is not guaranteed. Higher performance overhead is observed in invocation via channel, introduced by the handshake/rendezvous mechanism and channel reconfiguration on both the consumer and provider's side. In SOSJ, when both the consumer and provider CDs belong to the same subsystem, signal and channel communication are achieved through shared memory. On the other hand, WS4D JMEDS service invocation always goes through the DPWS communication stack that utilizes TCP/IP.

With regard to distributed setting (results shown in Fig. 13), SOSJ achieves better performance in service invocation compared to WS4D JMEDS. Performed over UDP/IP, SOSJ invocation via signal puts ahead performance over reliability in service invocation (i.e., no guarantee in data delivery). Meanwhile, invocation via channel is slower as it guarantees data delivery. In this benchmark, invocation messages sent through channel are exchanged via TCP/IP. Despite having the added feature of guaranteed data exchange and the use of TCP/IP, SOSJ invocation via channel has better performance compared to WS4D JMEDS, which relies only on its communication stack to support the reliability in service invocation.

3) Qualitative Comparisons: SOSJ Versus WS4D JMEDS: This section presents some additional comparisons between SOSJ and WS4D JMEDS, which are given in Table IV.

SOSJ utilizes built-in reactive programming constructs which support software behaviors that react to events from the environment through signals, while WS4D JMEDS has limited features for supporting reactivity. While WS4D JMEDS supports asynchronous concurrency, achieving true synchronous concurrency is virtually impossible without underlying formal MoC. In contrast, being based on GALS MoC, SOSJ supports both asynchronous and synchronous concurrency as part of the language. For handling dynamic scenarios, SOSJ provides built-in features to handle wider range of application scenarios, as demonstrated in [21]. With regard to service invocation, in WS4D JMEDS it goes over the network regardless of the location of the service consumer and provider. Also, successful invocation is guaranteed in SOSJ through the use of channels.

VI. Conclusion

Based on the synergy of SOA concepts and system-level language SystemJ, SOSJ offers new support for designing dynamic distributed software systems typical for manufacturing applications. In this paper, code examples were presented to show how the SOA features of the SOSJ framework are used, and benchmark evaluations were conducted in handling SOA functionality over competing SOA-only approach. Our future works are on the extending SOSJ with new standard mechanisms for programmers to support reconfigurability through simple abstractions and interface, as well as achieving interoperability with other software tools used in manufacturing context, e.g., IEC 61499, through the use of service-oriented approach demonstrated in this paper.

References

[1] S.-H. Leitner and W. Mahnke, "OPC UA–Service-oriented architecture for industrial applications," *ABB Corporate Res. Center*, vol. 48, pp. 61–66, 2006.

[2] K. H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. New York, NY, USA: Springer, 2010.

[3] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Trans. Ind. Inform.*, vol. 7, no. 4, pp. 768–781, Nov. 2011.

[4] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1599–1614, Dec. 2009.

[5] R. Sinha, P. S. Roop, G. Shaw, Z. Salcic, and M. M. Y. Kuo, "Hierarchical and concurrent ECCs for IEC 61499 function blocks," *IEEE Trans. Ind. Inform.*, vol. 12, no. 1, pp. 59–68, Feb. 2016.

[6] J. Yan and V. Vyatkin, "Extension of reconfigurability provisions in IEC 61499," in *Proc. IEEE 18th Conf. Emerg. Technol. Factory Autom.*, 2013, pp. 1–7.

[7] W. Dai, V. Vyatkin, J. H. Christensen, and V. N. Dubinin, "Bridging service-oriented architecture and IEC 61499 for flexibility and interoperability," *IEEE Trans. Ind. Inform.*, vol. 11, no. 3, pp. 771–781, Jun. 2015.

[8] A. J. Hoffman and A. H. Basson, "IEC 61131-3-based Holonic control of a reconfigurable manufacturing subsystem," *Int. J. Comput. Integr. Manuf.*, vol. 19, pp. 1–15, 2015.

[9] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, "JADE—A java agent development framework," *Multi-Agent Program.*, vol. 15, pp. 125–147, 2005.

[10] M. Lützenberger, T. Konnerth, T. Küster, J. Tonn, N. Masuch, and S. Albayrak, "Engineering JIAC multi-agent systems," presented at the Int. Conf. Auton. Agents Multi-Agent Syst., Paris, France, 2014.

[11] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski, "WS4D: toolkits for networked embedded systems based on the devices profile for web services," in *Proc. 39th Int. Conf. Parallel Workshops*, 2010, pp. 1–8.

[12] P. Varga *et al.*, "Making system of systems interoperable – The core components of the arrowhead framework," *J. Netw. Comput. Appl.*, vol. 81, pp. 85–95, 2017.

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

[14] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Comput. Lang., Syst., Struct.*, vol. 36, pp. 317–344, 2010.

[15] A. Malik, Z. Salcic, C. Chong, and S. Javed, "System-level approach to the design of a smart distributed surveillance system using system," *ACM Trans. Embedded Comput. Syst.*, vol. 11, pp. 1–24, 2013.

[16] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski, "WS4D: toolkits for networked embedded systems based on the devices profile for web services," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, 2010, pp. 1–8.

[17] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Depart. Comput. Sci., Stanford University, Stanford, CA, USA, 1984.

[18] U. D. Atmojo, Z. Salcic, and K. I. K. Wang, "SOSJ: A new programming paradigm for adaptive distributed systems," in *Proc. IEEE 10th Conf. Ind. Electron. Appl.*, 2015, pp. 978–983.

[19] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.

[20] U. D. Atmojo, Z. Salcic, and K. I. K. Wang, "Extending SOSJ framework for reliable dynamic service-oriented systems (short paper)," in *Proc. IEEE 8th Int. Conf. Service-Oriented Comput. Appl.*, 2015, pp. 213–218.

[21] U. D. Atmojo, Z. Salcic, and K. I. K. Wang, "Dynamic reconfiguration and adaptation of manufacturing systems using SOSJ framework," *IEEE Trans. Ind. Inform.*, vol. 14, no. 6, pp. 2353–2363, Jun. 2018.

[22] OASIS. (27 July). OASIS Devices Profile for Web Services (DPWS). 2017. [Online]. Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01

[23] WS4D. Projects | Web Services for Devices, Jan. 19, 2017, [Online]. Available: http://www.ws4d.org/projects/

Udayanto Dwi Atmojo (S'10–M'18) received the Sarjana Teknik [equivalent of B.E.(Hons.)] degree in electrical engineering from Universitas Gadjah Mada, Yogyakarta, Indonesia, in 2010, and the Ph.D. degree in electrical and electronics engineering from the University of Auckland, Auckland, New Zealand, in 2017.

He is currently a Postdoctoral Research Fellow with the Information Technology in Automation Research Group, Department of Electrical Engineering and Automation, Aalto University, Finland. His research interests include software engineering and distributed architecture for industrial manufacturing and automation systems, industrial informatics, and embedded systems.

Zoran Salcic (SM'98) received the B.E., M.E., and Ph.D. degrees in electrical and computer engineering from Sarajevo University, Sarajevo, U.K., in 1972, 1974, and 1976, respectively.

He is a Professor and the Chair in Computer Systems Engineering with the University of Auckland, Auckland, New Zealand. He has authored or coauthored more than 300 peer-reviewed journals and conference papers and several books. His main research interests are related to the various aspects of cyber-physical systems that include complex digital systems design, custom computing machines, design automation tools, hardware-software codesign, formal models of computation, languages for concurrent and distributed systems and their applications such as industrial automation, intelligent buildings, and environments, collaborative systems with service robotics and many more.

Dr. Salcic is a Fellow of the Royal Society New Zealand and recipient of the Alexander von Humboldt Research Award in 2010.

Kevin I-Kai Wang (M'06) received the B.E.(Hons) degree in computer systems engineering and the Ph.D. degree in electrical and electronics engineering from the University of Auckland, Auckland, New Zealand, in 2004 and 2009, respectively.

From 2009 to 2010, he was a Design Engineer in the home automation and traffic sensing industries. He was a Postdoctoral Research Fellow with the University of Auckland in 2011. He is currently a Senior Lecturer with the Department of Electrical and Computer Engineering, University of Auckland. His current research interests include distributed computational intelligence, pervasive healthcare systems, industrial monitoring and automation systems, and biocybernetic systems.

Valeriy Vyatkin (M'03–SM'04) received the Ph.D. and Dr.Sc. degrees in applied computer science from Taganrog Radio Engineering Institute, Taganrog, Russia, in 1992 and 1999, respectively, the Dr.Eng. degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999, and the Habilitation degree from the Ministry of Science and Technology of Sachsen-Anhalt, in 2002.

He is on joint appointment as the Chair of Dependable Computations and Communications, Luleå University of Technology, Luleå, Sweden, and Professor of Information Technology in Automation, Aalto University, Finland. He is also the Codirector of the international research laboratory Computer Technologies, ITMO University, Saint-Petersburg, Russia. Previously, he was a Visiting Scholar with Cambridge University, Cambridge, U.K., and had permanent appointments with the University of Auckland, New Zealand; Martin Luther University, Germany, as well as in Japan and Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures and multiagent systems in various industries: smart grid, material handling, building management systems, data centers, and reconfigurable manufacturing.

Dr. Vyatkin was the recipient of the Andrew P. Sage Award for the Best IEEE Transactions Paper in 2012. He is the Chair of IEEE Industrial Electronics Society (IES) Technical Committee on Industrial Informatics.