Designing Dynamic and Collaborative Automation and Robotics Software Systems

Zoran Salcic, *Senior Member*, *IEEE*, Udayanto Dwi Atmojo, Heejong Park, Andrew Tzer-Yeu Chen, *Student Member*, *IEEE*, and Kevin I-Kai Wang, *Member*, *IEEE*

***Abstract*—The heterogeneity of execution platforms and operating software in manufacturing machines and robots, as well as various sensors and actuators, creates challenges for integration into larger systems. Existing approaches make use of different types of middleware to mitigate the challenges of designing interoperable systems. However, middleware can significantly impede modular design and composition of software systems that are dynamic in nature. This paper elaborates upon those challenges and proposes using an approach called SOSJ, based on the system-level programming language SystemJ enhanced with service oriented features. This approach allows developers to design dynamic software systems while adopting and incorporating legacy solutions. The approach is demonstrated on the integration of an industrial automation system, incorporating the use of multiple modular mechatronics stations and service robotics systems, represented by ROS-enabled Baxter robots. The proposed approach offers a simple service interface based on abstract objects for integrating robots and automation machines in the SOSJ world, without the need to modify the underlying mechatronics or robotics systems.**

***Index Terms*—Service-oriented Architecture, Dynamic Software Systems, Reconfigurable and Interoperable Systems, Automation Systems, Service Robotics**

I. Introduction

INDUSTRIAL robots have largely been the domain of large-scale manufacturers, with plants costing millions of dollars. However, their costs have been falling, and industrial robots have become accessible for small and medium-sized businesses that offer customers niche products with relatively small and short production runs. There is an increasing focus on intelligent systems [1], [2] that utilize robots and automated specialized machines, as well as their sensors and actuators which have the ability to complement and assist humans in modern manufacturing systems. In particular, there is a need for versatile equipment (e.g. robots) that can meet the needs of multiple roles for creating in a variety of different end products. The desire to do this in a reconfigurable way that also meets safety-critical, fault-tolerance, and real-time requirements presents significant challenges for system designers. Robots such as the Baxter humanoid robot [3] represent a flexible platform for development of software techniques that can fulfil these disparate requirements.

Baxter is designed for safe and reconfigurable use around humans and is capable of performing routine tasks with the support of its own onboard processing. It operates as a standalone robot with two arms for performing mechanical operations, as well as a number of sensors such as ultrasonic, infrared, and camera sensors. Baxter has built-in safety mechanisms for detecting people in its vicinity, as well as compliant actuators that prevent physical harm to humans or other equipment. However, integrating robots like Baxter into manufacturing environments that use typical mechatronic devices can be difficult. In most cases, integration relies on a human operator to "program" the robot, either through writing code or through non-textual methods such as using joysticks, learning-from-demonstration, or block-based programming. To exacerbate the problem, in most cases the robot will then repeat the programmed task indefinitely, with limited flexibility to fulfill new manufacturing needs which initially may not be present in the manufacturing process. These robots also often operate independently from other mechatronics systems which are likely from different vendors, and therefore have limited or no communication or synchronization at the software level. Without direct communication or synchronization, additional means are required to monitor the behavior of the different systems to detect unwanted behavior or faults and make corrective actions. This is contrary to the fast-moving versatility required in small-scale manufacturing environments.

Ideally, robots should be capable of performing different (and possibly new) roles, which could be introduced during runtime, while also collaborating with other robots, machines, and humans in a predictable and analyzable way. The Baxter robot uses the open-source Robot Operating System (ROS) [4], which is not considered among the mainstream software approaches for mechatronic systems. There are opportunities for integration with the types of software systems used in existing intelligent automation systems, with further extensions towards true intelligent manufacturing and man/machine/robot communication. The challenge comes in integrating different concurrent and distributed automation processes that utilize heterogeneous computing platforms with different computational capabilities, connected through communication networks using different protocols, and underpinned by varying run-time support systems and software design frameworks. The dynamic nature of these systems, where individual soft-

Manuscript received 11 July 2017; revised 24 October 2017; accepted 17 December 2017. Date of publication [DATE]

All authors are with the Embedded Systems Research Group, Department of Electrical and Computer Engineering, The University of Auckland, New Zealand (e-mail: z.salcic@auckland.ac.nz).

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

Digital Object Identifier [DOI]

ware behaviors may participate only temporarily in the system, increases the complexity of system design. While robotics software frameworks like ROS exploit concurrency as a major tenet of developing robotic operations, they are not natively designed to meet the requirements of dynamic systems and typically do not match the particular requirements of industrial automation systems. Conversely, software designed using industry standards such as IEC 61131 and 61499 [5] are not specifically designed for handling the types of concurrency found in robotics systems. In order to integrate the two worlds of robotics and industrial automation while supporting dynamic changes, clear pathways for interoperability have to be found.

SystemJ [6] and Service-Oriented SystemJ (SOSJ) [7] have demonstrated the ability to address the interaction and integration of software behaviours, controlling industrial machines and monitoring their external environment with the ability to deal with dynamic scenarios. However, limited attention has been given towards addressing interoperation with and integration of software behaviours developed by other tools and legacy systems, which increasingly appear in the context of the industrial Internet of Things (IoT) and Industrie 4.0 [2]. This paper introduces an approach for a novel use of the SOSJ framework for the integration of heterogeneous and existing software components for achieving dynamic interoperable dynamic software systems, regardless of their original development tools and underlying execution platform(s). The integration is achieved through the use of the SOSJ service interface mechanisms and encapsulation of the services developed in the non-SOSJ world into the SOSJ world. With this solution, the low-level implementation details of the communication and interaction between industrial machines and mechatronic devices, robots, sensors/actuators, and even those associated with human interactions can bwe abstracted away, thus system designers can focus on designing systems without intervening at the low level implementation and configuration of specific software and platforms. In this paper, the approach is demonstrated on a real world case study of integration and interoperation of an industrial automation platform and service robotics system that uses Baxter robots in a new dynamic collaborative system. This example clearly indicates the pathway for (1) achieving integration of almost any type of cyber-physical system (CPS) into a SOSJ-designed software system, and (2) encapsulation of existing systems into SOSJ software behaviours, their integration, and use in new application contexts. The main contributions and novelties presented in this paper are:

1) The approach for integration of the existing ROS robotics framework into the dynamic software system paradigm through properly defined interfaces and integration into SOSJ abstractions.
2) Encapsulation of physical services of the Baxter robot, provided through ROS, into the service paradigm of SOSJ with minimal programming effort. These services are available to use within the SOSJ framework to achieve an almost unlimited number of new application scenarios. The new application scenarios that use Baxter services within SOSJ abstractions are underpinned with and inherit all benefits of a formal GALS model of computation.
3) A clear pathway for the applicability of the approach for the integration and interoperability of existing software systems when creating wider and complex functionality, while also supporting dynamicity at the same time.

The paper is organized as follows. Section II presents a motivating scenario that helps contextualize the features of dynamic software systems, and the SOSJ framework is briefly introduced in Section III. Section IV focuses on the integration of the ROS-based Baxter robot and an industrial automation system set-up, and Section V discusses how SOSJ achieves interoperability between the various software platforms at different levels of abstraction, thus serving as a capable software approach that encapsulates existing software products and imports them into the SOSJ world. In Section VI, results of experiments that demonstrate successful integration quantitatively, based on the achieved performance levels of SOSJ in a real-world application, are presented, followed by qualitative comparisons of SOSJ with other related approaches in Section VII, before concluding in Section VIII.

II. Motivating Scenario

The SOSJ programming paradigm is demonstrated through a real-world manufacturing scenario called the Automated Bottling System (ABS), shown in Figure 1A. The primary purpose of this system is to take empty bottles at the input, fill them with a liquid, and affix a screw cap onto the bottle. Additional functionalities can be added, such as filling and mixing different types of liquids, but they are outside the scope of this paper. A central rotary table, shown in Figure 1B, allows for multiple bottles to pass through the various stages in a pipeline fashion, improving throughput while also discretizing the flow of the bottles into separate steps (rather than moving them on a continuous conveyor belt), which simplifies system configuration and reduces timing concerns. There are five active positions in the rotary table as shown in Figure 1A - the bottle enters at position 1, the bottle is filled with the liquid at position 2 (filler), a cap is placed on top of the bottle at position 3, a twisting arm screws the cap on at position 4 (capper), and the bottle is removed at position 5. Photoelectric sensors are used at each position to detect the presence of a bottle. Importantly, the rotary table requires a number of interactions with external systems - loading of bottles at position 1, loading of caps at position 3, and removing the bottle at position 5. Normally, these jobs would be done by human operators, primarily to provide the requisite manual dexterity to manipulate bottles and caps, as well as to recognize faults and adjust behaviors.

In this example the humans are replaced with two Baxter robots, one of which is shown in Figure 1C. Each robot has two arms; each arm has 7 degrees of freedom, with an arm span similar to that of a human worker, along with an embedded camera built into the arm which allows us to use computer vision to verify that tasks have been completed correctly. The robot has its own Intel i7-powered Linux PC running ROS.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2017.2786280, IEEE Transactions on Industrial Informatics

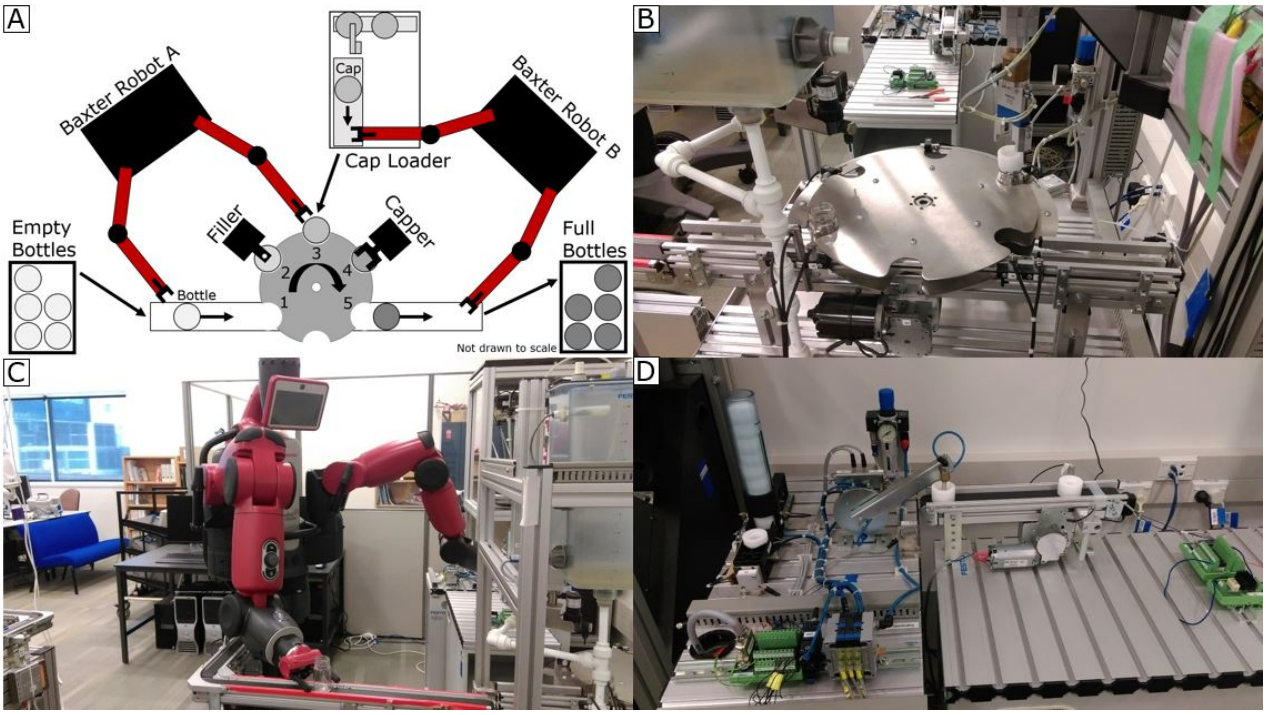


Fig. 1. The overall system diagram for the Automated Bottling Station (ABS) is shown in A), with photos of the components B) the central rotary table, C) the Baxter robot, and D) the cap-loader station.

The robot collects bottles from a box and places them on a conveyor belt that leads to position 1 of the rotary table. The caps are provided in magazines (packs), so a separate cap loading machine is utilized, shown in Figure 1D, which places individual caps on a conveyor belt. This machine uses a Beaglebone Black embedded computer to provide sensing and control capabilities. Baxter picks individual caps up and places them on bottles in position 3 of the rotary table. Lastly, a second Baxter robot picks up the filled bottles and places them in another box. To control the bottling station (including the rotary table, input/output conveyor, filler, and capper), four Altera Cyclone V System-on-Chip devices that have a dual-core ARM Cortex-A9 processor and FPGA fabric, one associated with each mechatronic device, are used to provide computational resources for actuation and synchronization. Therefore, the example is considered heterogeneous with FPGAs, embedded computers, and desktop PCs that need to be integrated together and orchestrated through IP-based network communication. All computing resources are connected to a LAN through a combination of Ethernet and Wi-Fi.

Certain types of failures in this system can be detected and responded to. For example, there is unused actuation capacity since there are two Baxter robots with two arms each but only three parallel independent tasks to complete with those arms (loading bottles, loading caps, and removing bottles). In our system, the spare arm can be used as a back-up in a fault-tolerant system, specifically for loading the cap onto the bottle, which generally requires good manual dexterity because there can be significant variation in the position and orientation of the caps. If one Baxter arm fails to complete the job successfully, then the spare arm should be used to attempt the task instead. If both arms fail, then human intervention may be required in particularly tricky cases. Other environmental sensors are also added to detect the presence of humans, and if they are too close to the robots or rotary table equipment, the machines should suspend their motion in order to ensure human safety.

In this system, the various machines are abstracted as the software services provided by their software behaviors that offer sensing, actuation, and processing capacity. The SOSJ-based approach allows the integration of these services in the SOSJ world, controlling and orchestrating the system. This has important implications for introducing robots into industrial automation environments because robots (particularly those at the lower cost end of the market) often use different software platforms than those in other industrial machines (such as ROS, in comparison to Programmable Logic Controllers (PLCs) and embedded microcontrollers). Being able to abstract the various data sources and actuation tasks as services minimizes the complexities of integrating multiple disparate computer systems and software while maintaining broad system functionality and control.

III. The SOSJ Framework

SOSJ [7] is a programming framework that leverages both the power of the Service-Oriented Architecture (SOA) paradigm and the system-level language SystemJ [6]. SystemJ is suitable for designing concurrent reactive software systems based on the Globally Asynchronous Locally Synchronous (GALS) Model of Computation (MoC) with a fixed (static) number of software behaviors specified at the design time. It also allows the use of an almost complete set of Java

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 from a value signal or channel

statements by interleaving them with SystemJ statements. A list of SystemJ kernel statements is presented in Table I. Figure 2 shows a graphical representation of an example SystemJ program where mutually synchronous behaviors called reactions (indicated by R) are grouped into mutually asynchronous behaviors called clock domains (CDs), thus making a GALS system.

Interactions between CDs and the environment (i.e. the sensors and actuators of mechatronic stations and robots in our example) are facilitated by abstract objects called signals, while the interaction between CDs are facilitated by abstract objects called channels, without the need to deal with the low-level details of the physical interfaces (e.g. sensor/actuator interfaces and network protocols). The signals and channels are mapped onto various physical interfaces, specified in a configuration file, while the actual implementation of the physical communication is handled by the SystemJ Runtime System (RTS) implemented in Java. One or more CDs managed by the same RTS and Java Virtual Machine (JVM) are considered to be one SystemJ subsystem (SS). One or more subsystems form a SystemJ program (system), which may reside on one computing machine or be deployed and distributed across multiple machines. CDs communicate via channels implemented through different physical interfaces, referred to as links (e.g. implemented in TCP/IP, USB, CAN, shared memory etc.). The example of a SystemJ system shown in Figure 2 comprises three subsystems (SS1-SS3), deployed and distributed across two different machines/computers (M1 and M2), with a link (L1) abstracting away the physical connection between the machines on which subsystems run. For example, L1 could be a TCP/IP connection over Ethernet, while L2 could allow communication between two JVMs using shared memory on machine M2. The actual communication method is not important from a modelling (system specification) perspective because it is abstracted away. A SystemJ system always contains a fixed number of CDs specified at the design time.

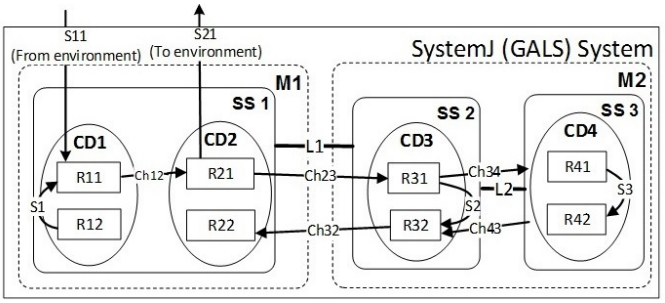


Fig. 2. Graphical illustration of a SystemJ GALS system comprising multiple subsystems.

The lack of support for dynamic systems in SystemJ has been addressed by SOSJ, primarily by the introduction of clock domain states and transitions between those states, as shown in Figure 3. This allows a CD to be dynamically created, made active and inactive, migrated, terminated, and updated, without stopping the operation of the rest of the system. This addition to the concept of clock domains was fundamental for the subsequent extension of SystemJ with the SOA paradigm and associated features to allow full dynamicity.

The SOSJ framework incorporates several SOA run-time processes including Service Discovery, Service Advertisement, and Request for Advertisement, which are based on the existing Simple Service Discovery Protocol (SSDP) [9]. The SOA part of SOSJ is implemented as a run-time library that allows the use of functions through the API calls presented in Table II. In order to handle dynamicity, the SOSJ Run-time System (RTS) extends the original SystemJ RTS. In the SOSJ framework, CDs that encapsulate reactions corresponding to physical sensing and actuation devices are considered as physical service providers. Physical services should advertise themselves when becoming available so that they can be discovered by the rest of the system. CDs that provide processing and computation are referred to as logical service providers in SOSJ. Logical services can interact with physical services and other logical services. Every CD has a standard service interface that ensures interactions follow the SOA paradigm, while still complying with the GALS MoC, which allows all services to be correct by construction and formally verifiable.

Service invocation in SOSJ is based on the SystemJ communication semantics i.e. reactions invoke services provided by other reactions in the same CD via signals, while reactions invoke services provided by other reactions in different CDs via channels. SOSJ also provides built-in mechanisms and programming constructs [7] that allow programmers to deal with dynamic creation, suspension, resumption, termination, and migration of a CD and its associated services. In these cases, channel reconfiguration may be required, where an output channel will be dynamically bound to an input channel.

IV. INTEGRATION OF ROS-BASED SYSTEMS INTO SOSJ

SystemJ and SOSJ are system-level design tools that enable composition of complex systems, either newly specified or

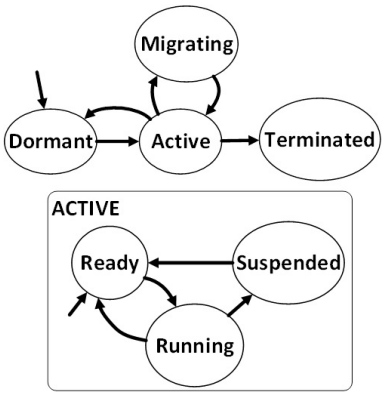


Fig. 3. Valid Clock Domain (CD) states and their transitions.

TABLE II
SUBSET OF THE SOSJ API

Function Call	Description
SOSJ.ConfigureInvocChannel()	Make a request for channel reconfiguration to the RTS for service invocation via channel
SOSJ.GetInvocChannelReconfigStat()	Obtains the status of the channel reconfiguration process (successful or not) from the RTS
SOSJ.CreateChanInvReqMsg()	Generate service invocation message for service invocation via channel
SOSJ.CreateSigInvReqMsg()	Generate service invocation message for service invocation via signal
SOSJ.CreateSigInvRespMsg()	Generate a response to service invocation message via signal
SOSJ.StoreService()	Stores service description into the internal service registry in the RTS
SOSJ.CreateChanInvRespMsg()	Generate 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

developed using other tools and languages that are subsequently wrapped up into SystemJ abstractions. The design and system composition approach is based on unifying the underlying physical and behavioral (software) worlds through the concepts of clock domains and the services they offer, while abstracting communication using signals and channels. In this section, we will explain the approach for wrapping and integrating Baxters physical services as parts of the ABS example described in Section II.

Robot Operating System (ROS) [4] is an open-source middleware framework that allows for collaborative robotics software development, saving time for developers with a set of ready-made libraries and tools. In the context of industrial automation, one of the challenges of ROS is its loosely coupled nature; ROS nodes broadcast over communication channels with no guarantee that there are any nodes listening, or that any nodes will confirm that messages have been received. This also makes timing guarantees very difficult to enforce. In addition, ROS is primarily available for use with C++, Python, and Lisp only; it is claimed that ROS has language independence, but based on the authors observation, only experimental libraries with limited capabilities are maintained in Java.

TABLE III
SUBSET OF THE SOCKET SERVER API

API Function	Description
analog_state(object_name)	Get the current value of an analog sensor or actuator
analog_setoutput(object_name, value, timeout (optional))	Sets the value of an analog actuator (e.g. the PC fan)
digital_state(object_name)	Get the current value of a digital sensor or actuator
digital_setoutput(object_name, value, timeout (optional))	Sets the value of a digital actuator (e.g. LED lights)
limb_moveto(object_name, px, py, pz, pr, pp, pya)	Move an arm to a specified endpoint in terms of {x,y,z,roll,pitch,yaw}. Calls a built-in script that performs the inverse kinematics calculations
limb_getpose(object_name)	Get the current endpoint position of an arm as {x,y,z,roll,pitch,yaw}
sonar_getdata(object_name)	Get the current sonar array values

The proposed approach to bridging the gap between SOSJ and ROS is to utilize the SOSJ link interface that uses TCP/IP to expose an API, translating between SOSJ service invocations and ROS commands. This abstracts away the operation of ROS into a set of APIs and allows communication from any language as long as that language has support for a network protocol stack and sockets. This removes any need for the system designer to be familiar with ROS. Importantly, it also means that any computer that tries to communicate with the ROS-supported device, e.g. the Baxter robot, does not need to have ROS installed on it either, which is an advantage since ROS (in its installed form) can be quite bulky due to a number of additional libraries/tools that accompany it. As such, the approach can be extended to a much wider variety of devices and systems, with their functionality made available through service interfaces accessible via network sockets. In the case of Baxter and ROS, a simple Python-based socket server was developed and deployed on the Baxter PC itself, which automatically registers with the ROS master as a node, and includes a number of simple scripts that also abstract away complicated tasks like access to the sonar data and inverse kinematics for arm movement. Table III shows some of the commands available in our API.

The socket server catches errors and returns error codes when necessary, or simply returns the data as a string. Integrating the socket server with SOSJ framework can be easily done by mapping the available API through abstracted SOSJ signals. The code for the socket server is available online at https://github.com/AndrewChenUoA/baxter_socket_server.

In this approach, ROS is considered to be part of the physical environment to SOSJ. ROS-based sensors and actuators are encapsulated as SOSJ services (i.e. CDs) and exchange data via signals. As such, our approach offers a unified method to integrate physical and logical services implemented in any language, network, or platform. Since our framework encapsulates ROS functions, the developer does not need to understand the underlying communication mechanisms of ROS, reducing the development and system setup time. The socket server

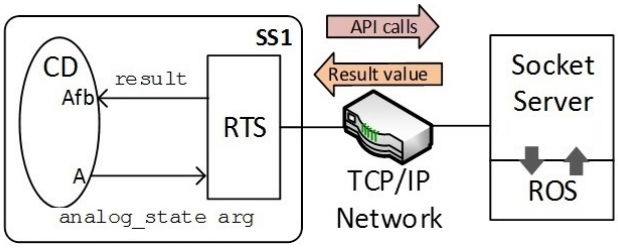


Fig. 4. SOSJ-ROS communication via the socket server.

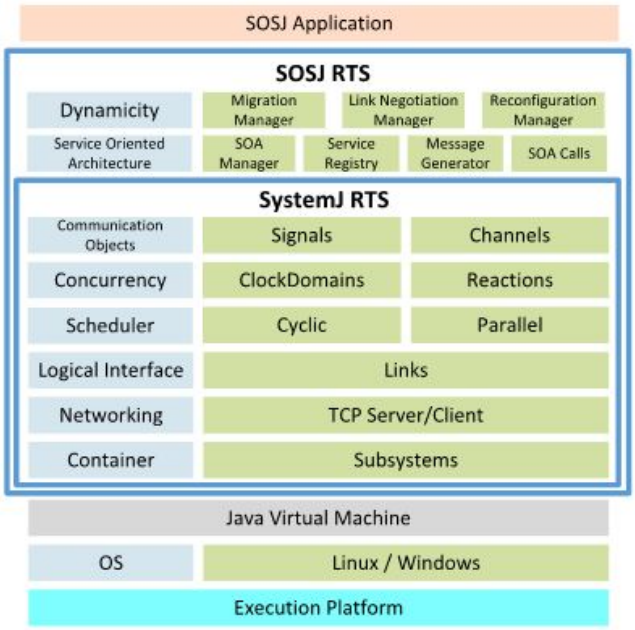


Fig. 5. The SOSJ RTS software stack.

that runs on the Baxter PC expects a string command in the format API_Function_Name followed by a list of arguments delimited by a whitespace character. In the case where a CD emits a signal with a command, it is transmitted to the socket server via a TCP/IP socket. An example process of how signals are exchanged between a SOSJ CD and the socket server is demonstrated in Figure 4.

In this example, the CD can run on anywhere, e.g. on any networked machine or on the Baxter PC. Referring to Figure 4, the CD first emits an output signal A with a value analog_state arg to request for the current value of an analog sensor (as described in Table III). The SOSJ Runtime System (RTS) will then transform the signal into a TCP stream which gets transmitted over a network to the socket server. After receiving this message, the socket server invokes the requested function in ROS, which will return a sensor value as a result. Then, the value is transmitted back to the RTS via the network and forwarded to the CD on input signal Afb. Once the signals can be exchanged between ROS and SOSJ, it becomes possible to develop higher level behaviors that are using those signals and offer them as services in SOSJ. This way, the complex behaviors (controllers) of Baxter can be developed using either SOSJ or SystemJ and integrated into larger applications.

It should be noted that it becomes the responsibility of the

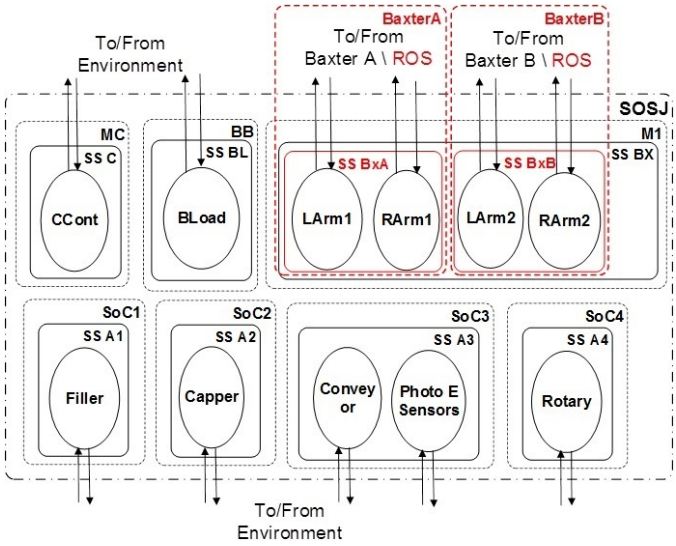


Fig. 6. Graphical illustration of the controller CDs in the ABS example.

CD to check if an operation initiated by signal emission has been completed. For example, a signal can be emitted to tell the socket server to issue a command to ROS to move the Baxter left arm to a particular position. This will take some amount of time to complete the physical movement. Once the command is completed a notification will be returned via an input signal into the CD. This approach allows encapsulation of complex physical services offered by Baxter into SOSJ.

The SOSJ RTS software stack is shown in Figure 5. SOSJ programs can be executed on any JVM-enabled platforms with or without an operating system. Since SOSJ is an extension of SystemJ, it is built on top of the original SystemJ RTS. The SystemJ RTS implements all of the features supporting SystemJ program execution; for example, the *TCP Server/-Client* block handles the TCP/IP communication links. The RTS allows the designers to opt how CDs are scheduled, whether in round-robin (indicated by the *Cyclic* block) or in parallel (indicated by the *Parallel* block) schedulers for the CDs. The *Signals* and *Channels* blocks indicate the part of the RTS which provide and handle the implementation of signal and channel communication respectively. The *Sub-systems* block provides the subsystem encapsulation which permits the execution of groups of CDs on different JVM instances. The *Links* block provides the implementation of the physical communication interfaces which form SystemJ/SOSJ links. The SOSJ SOA RTS enables design of GALS systems using SOA based approaches by introducing the common SOA facilities such as service discovery/advertisement, the notion of service consumers and producers, and CD lifecycles. Furthermore, the dynamic reconfiguration features of the SOSJ RTS allow creation, suspension, resumption, termination, and migration of CDs and channel reconfigurations during runtime.

V. The Automated Bottling Station in SOSJ

In this section, the SOSJ approach to designing the ABS from Section II is presented. Figure 6 shows the graphical illustration of the CDs which correspond to the mechatronic

devices and Baxter robots. There are six CDs which correspond to mechatronic devices. The BLoad CD controls the cap loader station and runs on a Beaglebone Black platform. The Filler, Capper, and Rotary CDs govern the filler, capper, and rotary table and run on the SoC1, SoC2, and SoC4 Altera Cyclone V platforms respectively. The SoC3 Altera Cyclone V platform executes the Conveyor and Photo E Sensors CD, which correspond to the conveyor and photoelectric sensor devices. Four additional CDs act as the wrappers that expose Baxter arm physical services to other SOSJ CDs. These CDs, namely LArm1, RArm1, LArm2, and RArm2, correspond to each Baxter arm of Baxter A (1) and Baxter B (2) respectively. The Baxter CDs can be executed on any networked computing machine that executes SOSJ. For example, as illustrated in Figure 6, the Baxter CDs can run on a separate machine M1, or even run on separate Baxter onboard PCs (indicated in red colour). Finally, the CCont CD runs on the machine MC and acts as an orchestrator which controls all mechatronic device and Baxter services. The ABS model in SOSJ can be deployed on any Java-enabled platform, centralized or distributed, without any changes of the source code specification since all physical I/O and communication links are abstracted away in SOSJ. Using SOSJ, it is possible to establish channel communication between CDs during execution time to support dynamic reconfiguration. For example, the ABS can be formed by using a central coordinator that dynamically invokes individual machines and Baxter services. A code example that demonstrates the invocation of the Baxter arm service through channel reconfiguration is presented in Listing 1. The invoked arm services are dynamically selected according to their priority and availability.

Listing 1. Code snippet showing the invocation of Baxter arm service (SOSJ)

```
1  CCont(
2     output signal SOSJDisc;
3     input String signal SOSJDiscReply;
4     output String channel InvReq1;
5     input String channel InvResp1;
6  )->{
7    {
8    //...further code...//
9    emit SOSJDisc;
10   await (SOSJDiscReply);
11   String serv = (String)#SOSJDiscReply;
12   Hashtable mRes = DoMatching(serv,Pos,Pos2);
13   //...further service matching code...//
14   boolean reconfstat1 = SOSJ.ConfigureInvocChannel ("
         CCont", "InvReq1", InvdCD, InvdChan);
15   if(reconfstat1){
16     pause;
17     boolean reconfstat2 = SOSJ.
           GetInvocChannelReconfigStat ("CCont", "InvReq1
           ");
18     if(reconfstat2){
19       String mSend = SOSJ.CreateChanInvReqMsg ("SS1","
             CCont", "InvResp1", actionName, ArmPos);
20       send InvReq1(mSend);
21       receive InvResp1;
22       //...further code...//
23       }
24     }
25   }
26     //...further code...//
27   }
28 }
29 LArm1(
30   input String channel RecInv1;
31   output String channel RespInv1;
32   output String signal ToLArm1;
33   input String signal ToLArm1fb;
34 )->{
35   {
36   while(true){
37     receive RecInv1;
38     String msg = (String)#RecInv1;
39     String ActName = SOSJ.GetAction(msg);
40     String data = SOSJ.GetData(msg);
41     if(ActName.equals("MoveLArm")){
42       emit ToLArm1("limb_moveto left_limb "+data));
43       await (ToLArm1fb);
44     }
45     //...further code...//
46     send RespInv1(ack);
47     pause;
48     }
49   }
50 }
51 RArm2(
52   input String channel RecInv2;
53   output String channel RespInv2;
54   output String channel ActRArm2;
55   input String signal ActRArm2fb;
56 )->{
57    //..code...//
58 }
```

The snippet presents 3 CDs. The CCont CD (line 1-28) is the orchestrator which invokes the service offered by other CDs, such as the Baxters arm services (i.e. LArm1 and RArm1). The LArm1 CD (line 29-50) encapsulates the left arm service of Baxter A through signal interfaces with the socket server. Similarly, the RArm2 CD (line 51-58) is a CD which encapsulates the right arm service of Baxter B. Initially, the CCont CD performs service discovery by transmitting a discovery message via signal SOSJDisc, a dedicated SOSJ signal for transmitting the discovery message (line 9), and then waiting to receive the discovery reply containing the service description of available services via signal SOSJDiscReply, a dedicated SOSJ signal for receiving the discovery reply (line 10). Then, CCont performs service matching to find the Baxter arm service to invoke, by executing the service matching method (line 12, represented as a Java method called DoMatching() that takes into account the Baxter robot physical location in the ABS). The service matching may return the Baxter A left arm service. If the Baxter A left arm is unavailable due to failure, the service matching returns the Baxter B right arm service, which is also capable of performing a similar task (to move a cap from the cap loader on to the bottle in a rotary table). Based on the service interface information obtained from discovery and service matching, CCont can trigger channel reconfiguration for service invocation (line 14) to reconfigure channel InvReq1 to couple with the receiving channel. If the channel reconfiguration is performed successfully (line 15-18), the CCont generates and sends the service invocation message via channel InvReq1 (line 19-20), and then waits to receive a reply via channel InvResp1 (line 21).

If CCont invokes the LArm1 service, LArm1 will receive the service invocation message from CCont via channel RecInv1 (line 37). Once the service invocation message is obtained (line 38) from the input channel, the CD can invoke the service that the CCont wishes to access (line 39) (in this example, MoveLArm, as shown in line 41, which corresponds with the actuation of the Baxter left arm). If such is the case, the CD emits a signal (ToLArm1) which will be translated into a command and sent to the socket server to call the API (limb_moveto left_limb) that actuates the Baxter left arm

(line 42), and then waits for a response via signal ToLArm1fb (line 43). Finally, the CD notifies the CCont CD via channel RespInv1 indicating the completion of the operation (line 46). Note that before reaching the send statement in line 46, in this scenario channel RespInv1 is reconfigured in the same manner with the approach shown in line 14-17 to bind the RespInv1 with InvResp1, hence these details are omitted in the listing (line 45).

VI. Experimental Results and Discussions

A set of benchmarks were run to demonstrate the performance of SOSJ communicating with the developed socket server. The performance of SOSJ in communicating with the socket server is compared against the Multi-Agent System (MAS) framework JADE [10]. A brief qualitative comparison between SOSJ and JADE in terms of major features is summarised in Table IV. For the quantitative comparison the benchmarks consider two scenarios: the first is to run the Baxter arm CD (in SOSJ) or Baxter arm agent (in JADE) on a separate machine, Raspberry Pi 2 B (900 MHz ARM processor, 1 GB RAM), and the second is to run the CD or the agent on the Baxter PC (Intel i7-3370 3.4 GHz, 4 GB RAM), with both execution platforms running Linux. The benchmarks measure the round trip communication time to invoke two APIs, limb_getpose (to obtain the arm endpoint position) and limb_moveto (to actuate the arm to a particular endpoint position), through the socket server. This essentially distinguishes between a sensor service and an actuation service (where the time to complete the actuation is included). In the experiments, the total round trip time of 500 and 1000 limb_getpose and of 50 and 100 limb_moveto API invocations is measured using the Java system timer and averaged. These results are presented in Figure 7.

The results show that SOSJ has comparable performance in communicating with the socket server to perform (Baxter) physical service invocation in comparison to JADE. These results also show that the communication time (a few milliseconds) is essentially negligible in comparison to the actuation time for actuation services (in the order of seconds). The difference in average round trip time between SOSJ and JADE in invoking limb_moveto is negligible. The difference in average round trip time between SOSJ and JADE in communicating with the socket server for invoking limb_getpose is slightly in favour of JADE, which can be attributed to the SOSJ RTS handling SOSJ signals and transforming them to and from physical TCP/IP signals in order to interact with the socket server. However, this small tradeoff in performance is justified, considering the fact that unlike JADE, SOSJ completely abstracts away the communication with the environment, in this case the Baxter robot arms. A code snippet presented in Listing 2 shows how a JADE agent interacts with the environment. After the agent receives a message (from the environment) via UDP/IP by invoking RecvViaUDP() method (line 1), it sets a physical output pin, e.g. pin number 15 of a Raspberry Pi 2 B platform, to high (line 4) by invoking the SetGPIO() method, and transmits data via TCP/IP by executing the SendViaTCP() method to communicate with the

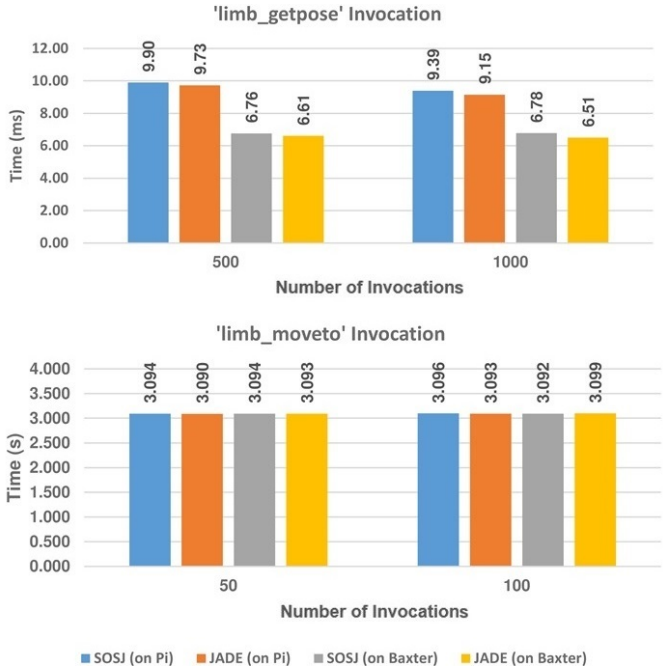


Fig. 7. Average round trip (service provision) time for limb_getpose and limb_moveto

TABLE IV
Comparisons Between SOSJ and JADE

Feature	SOSJ	JADE
Functional correctness	Supported by formal semantics and GALS MoC	No underlying formal semantics and MoC
Communication interface	Abstracted via signals and channels, handled by SOSJ RTS	Manually implemented or handled by extra middleware
Framework size	787 kB	2712 kB
Dynamic behaviour handling	Creation, suspension, resumption, termination, migration (strong and weak)	Same, except weak migration is not supported.

socket server to access Baxter physical services (line 6). Since they are not natively provided by JADE, these methods need to be implemented manually by the programmers or call other functions provided by a third party library or middleware which deals with the implementation details of the physical digital interfaces, TCP/IP, and UDP/IP communication links, which would be abstracted away in SOSJ. Additionally, SOSJ guarantees the compliance of the designed system with the GALS MoC, and satisfies a wider range of features and system requirements than JADE, as shown in Table IV.

Listing 2. Code snippet showing the invocation of Baxter arm service (JADE)

```
1       String mesg = RecvViaUDP(8766);
2       if(mesg!=null){
3             //...process message..//
4             SetGPIO("Raspberry Pi 2 B","output",15,"
                  high");
5             //...further computation..//
6             SendViaTCP("192.168.1.7",8456, data1);
7       }
```

As an alternative to the proposed approach, in order to achieve the integration of manufacturing devices and robots, system designers may opt to utilize the ROS framework only. This requires deployment of ROS behaviors to control manufacturing devices and robot (Baxter), and instantiation of

ROS slave nodes that allow interaction between Baxter and manufacturing devices. In that case, there is no need to use the socket server or other solutions which enable interoperability between different software approaches. However, this approach would require system designers to understand and manipulate ROS low-level functions, especially for the other manufacturing devices. Also, unlike SOSJ and JADE, ROS offers only limited support for handling dynamicity and that would impede the design of dynamic/reconfigurable manufacturing and robot systems. Furthermore, compared to SOSJ and JADE, ROS is the largest in terms of framework size, requiring at least 407 MB (up to 572 MB if complementary ROS libraries are added).

VII. Related Work

The use of service robots in industrial applications is often very challenging because of various practical issues such as precision and reconfiguration. Interestingly, most existing research focuses on intuitive and human-friendly methods for programming robot actions [11], while integrating service robots with other automation machines is barely investigated [12]. One fundamental reason is due to the fact that most existing industrial robot control software is vendor specific. However, the trend is moving towards using open-source robot platforms such as ROS [4]. As such, researchers have started developing common communication interfaces to bridge ROS and other peripherals. For example, ROS-Industrial [13] is an extension of ROS which introduces standardised interfaces for common external sensors, actuators, and industrial devices to communicate through ROS messages in a hardware-agnostic way, including code generation and automatic testing of functionality. Robot Raconteur [14] is a communication framework which enables software developed in different languages to communicate through a TCP/IP network. ROSBridge [18] is a middleware that provides an additional abstraction layer on top of ROS which allows non-ROS software to utilize or interact with ROS-based programs. These approaches offer similar features as the socket server solution implemented in our approach, but typically have a larger memory footprint and may still require the programmers to have in-depth knowledge of how ROS works in order to be used effectively. Additionally, they lack the ability to support higher level system composition which is critical for industrial automation applications [12].

The work presented in [15] uses a service interface function block (SIFB) of IEC 61499, which is customised to allow interoperability between IEC 61499 function blocks (that control industrial devices) and ROS (that handles industrial robots). The work presented in [16] introduces an approach and library which allows communication between IEC 61131 and ROS. The work presented in [17] proposes a communication layer to integrate ROS processes (or ROS Nodes) into IEC 61499 applications. The first two approaches offer a way to interface with ROS without looking into the dynamic nature of robotics and system reconfiguration. The last approach, while enabling systems to be dynamic by incorporating the SOA paradigm, is not based on a formal semantics or MoC. This leads to challenges in ensuring safe and functionally correct controls and interactions with robots, which become more difficult in the case of dealing with complex and distributed robot services.

Other software technologies amenable for designing systems like the ABS with industrial robots and manufacturing machines include Multi-Agent Systems (MAS) and Service Oriented Architectures (SOA). MAS frameworks such as JIAC [19] and JADE [10] possess the capabilities to handle dynamic behaviors. SOA-based technologies like WS4D [20] allow dynamic composition of software services and handling of dynamic behaviors to a limited extent. However, these tools are typically not based on any formal semantics or MoC and generally require heavy run-time environments to be in place, which is not always possible for embedded sensor and actuator peripherals. The heavy runtime environment can be observed in Table IV where the JADE memory footprint is almost four times the SOSJ memory footprint. Being based on a formal MoC with formal semantics, SOSJ allows design properties to be checked before they are deployed. The formal nature of the approach is also used at compilation time to produce code that preserves the properties of the designed system in the object code.

VIII. Conclusion

The SOSJ approach presented in this paper demonstrates the ability to integrate existing software products into the SOSJ world through services that can be used in a dynamic context. This allows system designers to create new dynamic software systems even if their constituent behaviors are not dynamic in nature. The services are produced and consumed by concurrent behaviors called clock domains. Wrapper clock domains can be easily written to encapsulate functionalities of other software products, like software developed for Baxter in the ROS world, and use them in new application scenarios. The main advantage of this is that the new behaviours are underpinned by a formal GALS model of computation. The same methodology can be applied to other software products that govern the operation of different types of sensors and actuators, typically used in CPS. The proposed approach allows dynamic composition of software behaviors, without rebooting the system, and can be utilized in the applications and scenarios that require dynamic reconfiguration or fault-tolerance. We have demonstrated this on an example of integration with ROS-based service robots. In future work, we plan to investigate how the approach can be utilized to achieve interoperability with other types of existing approaches and legacy systems, and how higher level behaviors like fault-tolerance can be more comprehensively supported. We also plan to extend our comparison of SOSJ against other frameworks like JADE and ROS qualitatively and quantitatively, in terms of setup and configuration time, level of automatic code generation, and operational overheads. Finally, we plan to investigate how the approach can be used to design dynamic interoperable systems that include parts with hard real-time requirements.

REFERENCES

[1] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos, Software-Defined Industrial Internet of Things in the Context of Industry 4.0, *IEEE Sensors*, **16**(20), pp. 7373-7380, 2016.

[2] H. Kagermann, W. D. Lukas, and W. Wahlster, Recommendations for implementing the strategic initiative INDUSTRIE 4.0, *Forschungsunion*, 2013.

[3] E. Guizzo and E. Ackermann, The rise of the robot worker, *IEEE Spectrum*, **49**(10), pp. 34-41, 2012.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, ROS: an open-source Robot Operating System, *International Conference on Robotics and Automation Workshop on Open Source Software*, 2009.

[5] C. Gerber, H-M. Hanisch, and S. Ebbinghaus, From IEC 61131 to IEC 61499 for distributed systems: a case study, *EURASIP Journal on Embedded Systems*, 2008.

[6] A. Malik, Z. Salcic, P. Roop, and A. Girault, SystemJ: A GALS Language for System Level Design, *Computer Languages, Systems & Structures*, **36**(4), pp. 317-344, 2010.

[7] U. D. Atmojo, Z. Salcic, and K. I-K. Wang, "SOSJ: A new programming paradigm for adaptive distributed systems", *IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 978-983, 2015.

[8] Z. Salcic, U. D. Atmojo, H. J. Park, A. T-Y. Chen, and K. I-K. Wang, "A Framework for Designing Dynamic and Interoperable Automation and Robotics Systems", *IEEE 15th International Conference of Industrial Informatics (INDIN)*, 2017.

[9] T. Cai, P. Leach, Y. Wu, Y. Goland, and S. Albright, "Simple service discovery protocol/1.0", *Internet Engineering Task Force*, http://tools.ietf.org/html/draft-cai-ssdp-v1-01, 1999.

[10] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, "JADE a java agent development framework" in *Multi-Agent Programming*, Springer, pp. 125-147, 2005.

[11] Z. Pan, J. Polden, N. Larkin, S. Van Duin, and J. Norrish, Recent Progress on Programming Methods for Industrial Robots, *Robotics and Computer-Integrated Manufacturing*, **28**(2) pp. 87-94, 2012.

[12] M. Hagele, K. Nilsson, N. Pires, and R. Bischoff, Industrial Robotics, *Springer Handbook of Robotics*, Springer, pp. 1385-1422, 2016.

[13] S. Edwards and C. Lewis, "Ros-industrial: applying the robot operating system (ROS) to industrial applications", *IEEE Int. Conference on Robotics and Automation, ECHORD Workshop*, 2012.

[14] J. D. Wason and J. T. Wen, "Robot Raconteur: A communication architecture and library for robotic and automation systems", *IEEE International Conference on Automation Science and Engineering*, 2011, pp. 761-766.

[15] N. Iannacci, M. Guissani, F. Vincentini, and L. M. Tosatti, "Robotic cell work-flow management through an IEC 61499-ROS architecture", *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1-7, 2016.

[16] M. de Sousa and H. Sobreira, "On adding IEC61131-3 support to ROS based robots", *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1-4, 2013.

[17] G. Ebenhofer, H. Bauer, M. Plasch, S. Zambal, S. C. Akkaladevi, and A. Pichler, "A system integration approach for service-oriented robotics", *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1-8, 2013.

[18] C. Crick, G. Jay, S. Osentoski, and O. C. Jenkins, "ROS and Rosbridge: Roboticists out of the loop", *ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 493-494, 2012.

[19] M. Ltzenberger, T. Konnerth, T. Kster, J. Tonn, N. Masuch, and S. Albayrak, "Engineering JIAC multi-agent systems", *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp. 1647-1648, 2014.

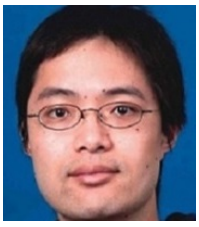
[20] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski, "WS4D: Toolkits for Networked Embedded Systems Based on the Devices Profile for Web Services", *39th International Conference on Parallel Processing Workshops*, pp. 1-8, 2010.

Zoran Salcic is a professor and holds a chair in computer systems engineering at the University of Auckland, New Zealand. He has the BE (72), ME (74) and PhD (76) degrees in electrical and computer engineering from Sarajevo University. 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 co-design, 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. He has published more than 300 peer-reviewed journal and conference papers and several books. He is a Fellow of the Royal Society New Zealand and recipient of the Alexander von Humboldt Research Award in 2010.

Udayanto Dwi Atmojo received the Sarjana Teknik (equivalent of BE(Hons)) degree in electrical engineering from Universitas Gadjah Mada, Yogyakarta, Indonesia in 2010, and the PhD degree in electrical and electronic engineering from the University of Auckland, New Zealand in 2017. He is currently a postdoctoral researcher at Aalto University, Finland. His research interests include software architecture and paradigms for cyber physical systems, embedded systems, and industrial automation.

Heejong Park received his B.E. degree in Computer Systems Engineering and Ph.D. degree in Electrical and Electronics Engineering in 2010 and 2016, respectively, both from the University of Auckland, Auckland, New Zealand. He is currently working as a research fellow in the Department of Electrical and Computer Engineering, the University of Auckland. His main research interests include compilation and verification techniques for safety-critical systems, formal semantics of programming languages, and real-time computing.

Andrew T-Y. Chen received the BE(Hons) degree in Computer Systems Engineering with First Class Honours and the BCom degree in Marketing, Innovation and Entrepreneurship from the University of Auckland, New Zealand, in 2015. He is currently pursuing a Ph.D. degree in Computer Systems Engineering, at The University of Auckland, New Zealand. His research interests include embedded systems, computer vision, hardware/software co-design, robotics, data analytics, and engineering education.



Kevin I-Kai Wang received the BE(Hons) degree in Computer Systems Engineering and the Ph.D. degree in Electrical and Electronics Engineering from the University of Auckland, New Zealand, in 2004 and 2009 respectively. He worked as a design engineer in the home automation and traffic sensing industries from 2009 to 2010. He rejoined the University of Auckland as a Post-Doctoral Research Fellow in 2011. He is currently a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland. His current research interests include distributed computational intelligence, pervasive healthcare systems, industrial monitoring and automation systems, and bio-cybernetic systems.