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

Dynamic Reconfiguration and Adaptation of Manufacturing Systems Using SOSJ Framework

Published in:
IEEE Transactions on Industrial Informatics

DOI:
10.1109/TII.2018.2808270

Published: 01/06/2018

Document Version
Peer reviewed version

Please cite the original version:
Dynamic Reconfiguration and Adaptation of Manufacturing Systems Using SOSJ Framework

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

Abstract—One of the key challenges in modern manufacturing systems is how to dynamically reconfigure software behaviours that govern machines to reflect changes in physical manufacturing process without completely resetting the entire manufacturing operation. The existing software solutions used to describe software behaviours in manufacturing systems are typically not based on formal semantics and model of computation and have limited capabilities in handling dynamic adaptation/reconfiguration. This paper presents the Service Oriented SystemJ (SOSJ) framework that supports a new programming paradigm for designing dynamic distributed manufacturing systems. SOSJ combines the system-level language SystemJ and service oriented architecture (SOA) paradigm to take advantages of both SystemJ’s correct-by-construction formal semantics and SOA’s dynamic features, respectively. The paper describes the concepts and functionalities of SOSJ which enable dynamic reconfiguration of a typical manufacturing system. Performance benchmarks are run to compare the capabilities of SOSJ to a multi-agent system framework JADE.

Index Terms—dynamic software systems; service oriented architecture; industrial manufacturing; reconfiguration

I. INTRODUCTION

INDUSTRIAL manufacturing systems can be considered as cyber-physical systems in which multiple concurrent software behaviours govern industrial sensors and actuators running on embedded controllers. Manufacturing operations are carried out by machines capable of reacting to changes in their environment and adjusting their operations based on intelligence embedded in these software behaviours. In reconfigurable manufacturing systems, these changes often require software behaviours to be re-composed and orchestrated dynamically without system restart and with minimum or no downtime. Thus, the ability to adapt to new configurations of physical machines, dynamic orchestration, creation, removal, and migration of the associated software behaviours become the essential features in modern manufacturing system [1]. In addition, the ability to define a complete system with safe, correct-by-construction functionality is also of utmost importance.

There are many different approaches on how software behaviours are described. The conventional approaches of designing software behaviours for industrial manufacturing process are based on the use of Programmable Logic Controllers (PLCs) as the physical control platforms, and their software behaviours are typically developed using traditional approaches such as the IEC 61131 [2] or IEC 61499 [3]. Despite the extensive use of these approaches, both the IEC 61131 and 61499 lack the formal semantics and Model of Computation (MoC). The introduction of formal semantics has been attempted [4] [5], however, they have no or very limited features that enable dynamic adaptation and reconfiguration.

Also, there are attempts to use other sophisticated software technologies in addressing distributed setting and dynamic behaviours. Two prominent ones are the Multi Agent Systems (MAS) [6] and the Service-Oriented Architecture (SOA) [7]. Some of the examples of the MAS frameworks are JADE [8] and JIAC [9]. While typical MAS frameworks have the ability for handling dynamic reconfiguration, they lack features for designing safe, correct-by-construction software behaviours, particularly formal semantics and underlying MoC. Similarly, SOA approaches like the DPWS [10] and OPC-UA [11] are not based on formal semantics and MoC, despite the increasing popularity of SOA paradigm in the industrial manufacturing context. There are also efforts in providing reconfiguration by utilizing the aforementioned software technologies within the IEC 61131 [12] and the IEC 61499 [13] [14], however only limited/partial reconfiguration is supported. On the other hand, a recent attempt to use a programming language with formal semantics and Globally Asynchronous Locally Synchronous (GALS) MoC, SystemJ [15], demonstrates the ability to easily specify correct-by-construction software behaviours for execution on distributed platforms. However, SystemJ lacks the necessary features to handle dynamic reconfiguration.

In this paper, we present a new programming approach called Service Oriented SystemJ, or SOSJ, introduced partly in [16] [17], is further extended to capture dynamic behaviours such as dynamic creation, removal, and migration of software behaviours. The main contribution in this paper is the new framework with Runtime Support (RTS) for handling dynamic behaviours and reconfiguration in SOSJ context. The paper extends the work in [18] with detailed description of SOSJ’s programming features and RTS which support dynamic adaptation and reconfiguration of manufacturing systems and presents new results and more comprehensive comparison and evaluation of the SOSJ framework.

The remainder of the paper is organized as follows. Section II presents the motivating example and the identified design requirements of dynamic and reconfigurable manufacturing systems. Section III gives a brief introduction of SOSJ. Section IV describes the implemented extension which has been incorporated into SOSJ to handle dynamic behaviours. Section V presents benchmarks and experimental results to evaluate the
performances of SOSJ when dealing with dynamic behaviours. Conclusions are given in Section VI.

II. MOTIVATING EXAMPLE

In this section, a motivating example used in [19] is revisited. A modular manufacturing system shown in Fig. 1 performs automatic bottle capping and storage. Four processing stations (two loading stations, one capping, and one storing station) are used to perform operations on bottles that are transported using conveyors. Multiple embedded controllers are deployed across the system, acquiring inputs from the sensors and controlling processing stations and conveyors. Station status, sensing information and commands can be exchanged between the embedded controllers via ModBus or CAN (Controller Area Network) Bus. A Master Controller (MC) provides higher level composite services such as creating the bottle routing path using basic services from the sensors and actuators in the system. It also provides an interface service for humans. The possible logical association of software services provided by individual stations with embedded controllers in two system configurations, Configuration 1 (blocks in white background only) and Configuration 2 (blocks with white background and blocks with darker background and italic naming), is also shown in Fig. 1.

![Fig. 1. The physical layout of the manufacturing example (Left) & the logical association of physical stations with embedded controllers (Right).](image)

There are many scenarios likely to occur in the example. For example, the system starts with Configuration 1. During runtime, new functionality is introduced by adding conveyor CB4 and diverter D1, hence creating a new layout referred to as Configuration 2. Such dynamic scenarios need to be handled without restarting the entire system and without affecting other system functionalities. Based on the motivating example, it is important for the underlying programming framework to satisfy the following programming requirements:

1. **Functional correctness**: Manufacturing operation needs to be performed correctly during production. The programming framework based on formal semantics and MoC allows the design of functionally correct software, which control the behaviour of industrial machines and mechatronic devices, in both configurations.

2. **Concurrency and Reactivity**: Manufacturing systems typically consist of multiple computing machines that run asynchronously with each other, with each machine potentially running concurrent software behaviours that need to execute in lockstep (synchronously) with each other. Thus, it is important for the programming approach to support the handling of different types of concurrency. Also, those behaviours would benefit from having effective mechanisms for dealing with reactivity and enabling deterministic software function, especially important for safety-critical scenarios.

3. **Dynamicy**: Dynamic behaviours and reconfiguration of software behaviours (or dynamicy) must be supported by the underlying design and programming framework. Dynamicy is associated with the change of presence of behaviours resulting in the addition of behaviours to the system and removal of behaviours from the system. Addition of behaviours occurs at creation of new behaviours, resumption of suspended behaviours, and at migration of a behaviour from one to another machine for execution. Removal of behaviours occurs at termination of behaviours, suspension of behaviour’s execution, and at migration of behaviours to another location. In the reconfigurable manufacturing context, different physical stations may be installed to or removed from existing product lines or for maintenance reasons. Such situations require the reconfiguration of software behaviours including correct reestablishment of communication between existing behaviours and connections of the behaviours to the environment.

III. BRIEF INTRODUCTION TO SOSJ

SOSJ [16, 17] is a programming framework which leverages both the power of SOA paradigm and system-level language SystemJ [15]. With underlying formal semantics and GALS MoC [20], SystemJ is amenable for designing reactive concurrent distributed systems. SystemJ uses Java as the host language and benefits from its object orientation.

Fig. 2 shows a graphical representation of an example of a SystemJ program where mutually synchronous behaviours, called reactions, are grouped into mutually asynchronous behaviours called clock domains (CDs). Interactions between CDs and with the environment are facilitated by channel and signal abstract objects, without the need to deal with low-level details of physical interfaces (e.g. sensor/actuator interfaces, network protocols). The channels and signals are mapped onto...
various communication interfaces, specified in the CD configuration, and are handled by the SystemJ RTS implemented in Java. One or more CDs which are handled by the same RTS are considered to be in the same SystemJ subsystem (SS). One or more subsystems form a SystemJ program (system), which may reside on one computing machine or distributed across multiple machines.

CDs belonging to different SystemJ subsystems communicate via channels implemented through physical interfaces referred to as links (e.g. TCP/IP, USB, CAN, Java RMI, HTTP, etc.). The example of a SystemJ system shown in Fig.2 comprises four subsystems (SS1-SS4), deployed and distributed across three different machines (M1-M3), with links physically connecting pairs of subsystems. A SystemJ system always contains fixed number of CDs specified at the design time.

The SOA paradigm is added into SystemJ to enable dynamic composition of CDs, thus creating the SOSJ framework. The framework incorporates several SOA features, including the Service Discovery, Service Advertisement, and Request for Advertisement, which are implemented based on the existing Simple Service Discovery Protocol (SSDP) [21]. More details about SOSJ can be found in [16]. However, the mechanisms to handle full dynamicity are not present. These new mechanisms for dynamicity and reconfiguration are introduced in details in Section IV.

IV. EXTENDING SOSJ FOR DYNAMIC RECONFIGURATION

The new features for dynamic reconfiguration extend the static GALS model of SystemJ into dynamic GALS, which allows a program to be described at any point in time with fixed but changeable number of CDs. Dynamic GALS, which is not formally defined in this paper, can be informally considered as a MoC that has state defined by the number of CDs executing at any point in time. As new CDs are introduced or others removed from the program dynamically, the state of the program changes, but within each state the program behaves as a static GALS program. Thus, the semantics of program in any state remains as defined in the original SystemJ [15]. Changes/transition of the states happen outside of the execution time of affected (those that join or exit the program) CDs, in the housekeeping phase when the RTS performs functions of reactive interface and all operations related to dynamic features and reconfiguration in SOSJ. This must be carried out at the end of logical tick of affected CDs as logical tick computations of a CD are considered atomic (tick must complete).

The new extension of the SOSJ RTS is developed based on the proposed macro-states of the CD described in [19]. In the SOSJ context, reconfiguration constitutes dynamic creation, suspension, resumption, termination and strong and weak migration of CDs (strong migration implies the preservation of data, communication, and execution state, while none of the above applies in weak migration). The reconfiguration is restricted to the CD level to retain macro-state consistency and determinism within each CD. The macro-state of a CD is represented as a Java object called CD Descriptor. The CD macro-states and their transitions were introduced in [19].

The extensions to the SOSJ RTS proposed in this paper introduce the required functionality to achieve full dynamicity. The complete SOSJ RTS is illustrated in Fig. 3. It consists of the SOA RTS and Dynamic RTS. The new extension is denoted as Dynamic RTS (blocks with downward diagonal background), while the SOA RTS that supports SOA is depicted in Fig. 3 (block with grid background) and has been elaborated in detail in [16]. Assuming the use of round-robin CD scheduling, the current implementation of Dynamic RTS executes its functions during the housekeeping phase, which occurs at the end of tick of a CD (or if there are multiple CDs in a subsystem after the execution cycle of all CDs is completed as defined by the CD scheduling within the subsystem). This is implemented within the RTS, which guarantees consistency of channels for communication between CDs. This is done to preserve semantics of design entities and execution of SOSJ (and also SystemJ) program.

![Fig. 3. The complete SOSJ RTS.](image-url)

### TABLE I. AVAILABLE PROGRAMMING CONSTRUCTS TO SUPPORT DYNAMIC BEHAVIOIRS AND RECONFIGURATION.

<table>
<thead>
<tr>
<th>Programming Constructs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJ.CreateCD(cdN, cdM, cdSD)</td>
<td>Dynamic creation of CD with input parameters of CD name (cdN), file name of the CD configuration (cdM), and file name of the CD service description (cdSD)</td>
</tr>
<tr>
<td>SJ.SuspendCD(cdN)</td>
<td>Dynamic suspension of CD with the cdN parameter.</td>
</tr>
<tr>
<td>SJ.WakeUpCD(cdN)</td>
<td>Dynamic resumption of CD with the cdN parameter.</td>
</tr>
<tr>
<td>SJ.KillCD(cdN)</td>
<td>Dynamic termination of CD with the cdN parameter.</td>
</tr>
<tr>
<td>SJ.MigrateCD(cdN, cdM, cdSD, migD, migT)</td>
<td>Migration of CD with input parameters including cdN, cdM, cdSD, the name/identifier of the destination subsystem (migD), and the migration type (strong or weak) (migT).</td>
</tr>
</tbody>
</table>

The **Physical Signal Classes** represents the class files of SystemJ signal physical interface implementation used to interact with the environment. The **CD Channel Objects** implement the SystemJ channels. The **CD Scheduler** is responsible for scheduling of CDs in the subsystems. The **Mailbox** is a data structure which buffers interactions between different functions in the SOSJ RTS. The **Link Objects** denotes all physical communication interfaces (links) used for inter-subsystem communications.
The **Dynamic RTS** consists of a number of threads and functionalities that facilitate dynamic behaviours. Components in the **Dynamic RTS** interact with the SOA RTS part through the **Mailbox**, however, the **Reconfiguration Manager** in the **Dynamic RTS** accesses the internal service registry (which is a part of the **SOA RTS** ) directly without going through the **Mailbox**. The **Reconfig Req Receiver** is a thread acting as a server which receives incoming request from external source (e.g. human operator) to enable ‘manually triggered reconfiguration’ (which will be elaborated in Section IV.E). The **Dynamic Function Call** is a class which implements the **SOSJ** programming constructs (thoroughly described and explained in [19]) available to programmers to support dynamic behaviours and reconfiguration. These programming constructs are summarized in Table I. In the following subsections, new components in the **Dynamic RTS** shown in Fig.3 are described.

The **SOSJ Config** comprises three different data structures, the **CD Config Repository**, the **CD Descriptor Storage**, and the **Link Config Repository**. The **CD Descriptor Storage** stores CD descriptors, which are Java objects that contain data, execution state of CDs in the subsystem and their signal and channel communication status. This data structure is particularly important as it enables suspension and resumption of CD execution while retaining all data and execution state, and also strong migration of a CD. The **Link Config Repository** is a data structure which stores the configuration of links used by a subsystem.

The **CD Config Repository** is a data structure which stores the configuration of all CDs in a subsystem, which includes the mapping of their signals and channels. This data structure is crucial for CD creation, channel reconfiguration, as well as the instantiation and termination of Java threads that handle the physical interface of **SOSJ** signals with the CD configuration shown in Listing 1. The listing depicts an example of the configuration of CDs in SS1 illustrated in Fig.2. Two CDs in the example are **CD1** and **CD2**. Each CD has the following attributes: **CDClassName** (line 2, line 23), **Channel** (line 3, line 24), and **Signal** (line 10, line 31). The **CDClassName** defines the file name of the CD’s code file. The **Channel** attribute defines the CD’s channel configuration, while the **Signal** attribute defines the CD’s signal configuration. The **Channel** attribute has two sub-attributes, **input** (line 25) and **output** (line 4), which contains the configuration of input and output channel ports, respectively.

If any input or output channel is used, the input and output sub-attributes will contain the configuration of each output and input channel ports present in the CD (line 5-7, line 26-28). For input channel port, the configuration has the ‘From’ attribute, which defines the channel’s partner (output channel port) and the partner’s CD (line 27). For output channel port, the configuration has the ‘To’ attribute, which defines the channel’s partner (input channel port) and the partner’s CD (line 6). For example, channel **ChA** in Fig.2 is formed through the pairing of output channel port named **Ch12** in **CD1** and input channel port named **Ch21** in **CD2**. During runtime, the pairing of channel ports may be changed (e.g. one output channel port to be coupled/paired with another input channel port or vice versa), and new channel mapping/pairing configuration is saved after channel reconfiguration is performed for future channel reconfiguration.

Like the **Channel** attribute, the **Signal** attribute also has input (line 32) and output (line 11) sub-attributes, which comprise the configuration of individual input and output signals. Each input and output signal configuration has the **Class** attribute (line 13, line 34) which specifies the class file of signal physical interface implementation. The configuration may also have other implementation-specific attributes. For example, the output signal **S11** uses the **TCPReceiver** class which implements transmission of data to the environment via TCP/IP (line 13). The class needs two parameters, the IP address (‘**IPAddress**’) of the destination where the data is sent to (line 14) and the port number (‘**Port**’) which the data is sent through (line 15).

**A. Reconfiguration Manager**

The **Reconfiguration Manager** is the main component that ensures reconfiguration is achieved properly. It is responsible for including CD for or excluding CD from execution based on its macro-states, storing and fetching CD descriptors to and from the **CD Descriptor Storage**, altering the service description (when necessary) stored in the service registry inside the **SOA RTS**, and configuring and establishing signals, channels and links when necessary following the configuration stored in the **CD Config Repository**. Fig.4 shows the flowchart which governs the operation of the Reconfiguration Manager. Referring to Fig.4, the process flow of Reconfiguration Manager differs based on the dynamic behaviour it handles. In case of CD suspension, the Reconfiguration Manager excludes the CD from the scheduler. As described in [19], the physical interface of **SystemJ** input signal is typically handled by a separate Java thread. If a CD that has input signals is suspended, the threads which handle the physical interface of the input signals need to be terminated. It is the responsibility of the Reconfiguration Manager to trigger the termination of these threads and also to update the modified CD Descriptor.

In case of CD resumption, the Reconfiguration Manager fetches the stored CD descriptor from the **CD Descriptor Storage**. Since the threads handling the physical interface of **SystemJ** input signals are terminated during CD suspension

```
1   "CD1": { "CDClassName": "CDClass1", 2   "Channel": { 3     "output": { 4         "Ch12": { 5             "From": "CD2.Ch21" 6             "To": "CD1.Ch12" 7         } 8     } 9   } 10   "Signal": { 11     "output": { 12         "S11": { 13             "Class": "TCPReceiver", 14             "IPAddress": "192.168.1.5", 15             "Port": "10005" 16         } 17     } 18   } 19   } 20 21 22   "CD2": { 23     "CDClassName": "CDClass2", 24     "Channel": { 25         "input": { 26             "Ch21": { 27                 "From": "CD1.Ch12" 28         } 29     } 30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 ...
```
process, the Reconfiguration Manager has to reinstantiate these threads. Finally, the Reconfiguration Manager includes the CD into the CD Scheduler for execution, hence the CD resumes its execution from the point and state prior to its suspension.

In case of CD termination, the Reconfiguration Manager excludes the CD from the CD Scheduler, terminates any threads handling the physical interface of input signals used by the CD, and removes the CD Descriptor. If the CD configuration is stored in the CD Config Repository, the CD configuration is stored in the service registry and its code. In case of CD creation, the Reconfiguration Manager removes the existing CD descriptor (if a new CD is created to replace the existing CD), initializes the CD configuration and service description, and creates the CD descriptor. Next, the CD’s signals and channels are configured based on the CD configuration, and the link creation process is started (if a new link is required). Once the link is created, its configuration is stored in the Link Config Repository. Note that the link creation process is handled by functionality explained in Section IV.D. Finally, the CD Configuration and Service Description are saved before the CD is included to the CD Scheduler for execution.

In case of CD migration, the flowchart explains the process flows occurring on the CD source and destination location. There are three different flows, two happen at the source and one happens at the destination. The first flow is taken if the CD is to migrate (indicated as ‘Outgoing, before transfer process’ branch). In this flow, the Reconfiguration Manager of the source location excludes the CD from the scheduler (if the CD is included for scheduling), then it triggers the migration data transfer process. The transfer process itself is handled by different functionality in the Migration Manager explained in Section IV.C.

The third flow is taken if the migration process is completed at the destination location (indicated as ‘Incoming, transfer completed’). In this flow, two branches indicated as ‘Strong Migration’ or ‘Weak Migration’ are possible.

In case of strong migration, the Reconfiguration Manager gets the CD configuration and service description. Because strong migration retains data and execution state, the CD descriptor of the migrating CD is also transferred to the destination and will be used for resumption of the execution on the destination, hence, creating a new CD descriptor is not necessary. Next, the Reconfiguration Manager reconfigures the signals and channels of the CD based on the CD configuration, and then link creation process is started if a new link is needed. Finally, the CD configuration and service description are stored before the CD is included to the scheduler for execution. In case of weak migration, once the Reconfiguration Manager obtains the CD configuration and service description, it creates the CD descriptor, configures the signal and channel of the CD, and if a new link is needed, initiates link creation process. Finally, the CD configuration and service description are stored before the CD is included in the scheduler for execution.

B. Migration Manager

The Migration Manager consists of two functionalities which handle the migration between the source and destination subsystems, the Migration Receiver and the Migration Sender. The sequence diagram of the following migration process is illustrated in Fig.5. When a migration process is started, the Migration Sender thread is initiated by the Reconfiguration Manager and it sends a request to initiate migration to the destination subsystem (Mig Req Msg). Once it receives a reply indicating that the destination is ready to initiate the transfer process (Mig Resp Msg), the Migration Sender begins the data transfer (Mig Data). The Link & Migration Request Handler receives any request messages to initiate migration (Mig Req Msg). It transmits a reply message (Mig Resp Msg) as a response to the request for migration process, and a reply message as a response when the handshake for link creation between two subsystems is initiated by the Link Requester (Link Resp Msg) (see Fig.6). After the Link & Migration Request Handler sends a reply message to inform that it’s ready to initiate migration process, it will start the Migration Receiver thread to receive the data transferred from the source location (Mig Data).

C. Link Negotiation Manager

The Link Negotiation Manager consists of three functionalities. The Link Requester initiates the handshake for link creation between two subsystems by transmitting a request message (Link Req Msg), which will be received by the Link & Migration Request Handler of the partner subsystem, and receives reply from it (Link Resp Msg). When a reply is
received, the **Link Requester** starts the **Link Negotiator Sender** thread, while the partner subsystem, after sending the reply to the requesting side, starts the **Link Negotiator Receiver** thread. Then, both sides perform the handshake process to create a mutual physical interface for a link to be used by both subsystems. More detailed illustration of the handshake process is shown in Fig. 6.

![Link negotiation sequence diagram](image)

**Fig. 6.** Link negotiation sequence diagram

The Link Negotiator Sender requests (Req Intf Alloc) for information on the physical communication port currently used for existing links of the partner subsystem, which is responded by the Link Negotiator Receiver of the partner subsystem (Dest Intf Alloc). Then, the Link Negotiator Sender finds a physical port available to be used by both sides for the new link (GetIntfAlloc()), and transmits the configuration of the mutual physical port (for the new link) to the partner subsystem (Intf Alloc To Use). Finally, the link is established by both sides and the configuration of the new link is stored in the **Link Config Repository**.

**D. Manually Triggered Reconfiguration**

Apart from the aforementioned new functionalities, the SOSJ framework also provides a mechanism that allows manual triggering of the functionalities in the Dynamic RTS from a GUI. A client application provides the means for user to manually generate events (e.g. from a remote computer) which will be sent to the SOSJ RTS to trigger dynamic behaviours & reconfiguration. Fig. 7 shows the GUI of the developed client application.

![The GUI of the SOSJ client application](image)

**Fig. 7.** The GUI of the SOSJ client application

The client application allows the user to perform orchestration and reconfiguration by sending commands which trigger dynamic operations from the drop down menu labelled as **Command**. Available commands are CreateCD, SuspendCD, WakeUpCD, KillCD, and MigrateCD, which have identical purposes as the SOSJ constructs described in Table I. Each command requires user provided input parameters. Table II describes each field in the SOSJ client application. To demonstrate how the ‘manual’ triggering of dynamic reconfiguration can be performed using the client application, the conveyor CD creation scenario from the motivating example is considered. Fig.7 shows how inputs for the triggering CD creation via the client application should be provided through the GUI based on the CD creation scenario. In this example, the conveyor CD named ‘CB4CD’, with CD configuration and service description given in the ‘cb4cdconfig.xml’ and ‘cb4cdsd.xml’ files, respectively, will be created in the subsystem named ‘ControllerSS’ with physical location of 192.168.1.5 (IPv4 address).

**TABLE II. FIELD NAMES AND THEIR DESCRIPTION**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS Name</td>
<td>Subsystem identifier containing the CD to be reconfigured</td>
</tr>
<tr>
<td>SS Address</td>
<td>The physical location (e.g. IP address, URI, etc.) of the subsystem</td>
</tr>
<tr>
<td>CD Name</td>
<td>The CD identifier</td>
</tr>
<tr>
<td>CD Config</td>
<td>The CD configuration file</td>
</tr>
<tr>
<td>CD Service Description</td>
<td>The CD service description file</td>
</tr>
<tr>
<td>Migration Type</td>
<td>The type of the CD migration (strong or weak)</td>
</tr>
<tr>
<td>Migration SS Destination</td>
<td>The identifier of the destination subsystem of the migrating CD</td>
</tr>
<tr>
<td>Channel Name</td>
<td>The channel name to be reconfigured</td>
</tr>
<tr>
<td>Channel Direction</td>
<td>The direction of the channel defined in the Channel Name field</td>
</tr>
<tr>
<td>Partner Channel Name</td>
<td>The name of the CD where the partner channel is used</td>
</tr>
<tr>
<td>Partner Channel Name</td>
<td>The name of the partner channel defined in the Channel Name field</td>
</tr>
</tbody>
</table>

As shown in Fig.7, for CD creation, the client application requires 6 parameters as the inputs which the user needs to provide, the CreateCD should be chosen on the Command field, the CD Name field filled with ‘CB4CD’, the CD Config field with ‘cb4cdconfig.xml’, the CD Service Description with ‘cb4cdsd.xml’, the SS Name with ‘ControllerSS’ (subsystem identifier), and the physical location of the subsystem in the SS Address field with ‘192.168.1.5’. Once the above inputs are provided in the respective fields, The **Transmit Command** button needs to be pressed to trigger the transmission of the reconfiguration command from the client application. The command corresponds to an event accompanied with the reconfiguration parameters generated and sent by the client application to the RTS of the subsystem ‘ControllerSS’ deployed on a platform with physical location (IPv4 address) of ‘192.168.1.5’. Once sent, the event is received by the **Reconfig Req Receiver** of the RTS of the ControllerSS subsystem, which will pass the reconfiguration parameters to the **Reconfiguration Manager** to perform the CD creation as described in Fig. 4 in the ‘CD Creation’ flow.

All of the aforementioned changes are included to enable SOSJ to target a wider range of dynamic behaviours and scenarios, which is essential for dynamic distributed systems like reconfigurable manufacturing systems.

**E. Handling Dynamic Behaviour in SOSJ - Code Example**

This section showcases how the programming constructs in Table I can be used by programmers to perform dynamic adaptation/reconfiguration on parts of the example of the manufacturing system. Listing 2 presents the code snippet of a
CD which uses the programming constructs and CD which governs the behaviour of conveyor.

As shown in Listing 2, the conveyor CD is described in lines 11-33. The CD receives requests to actuate the conveyor via channel ConvReqInCh (line 17). Based on the received request, the conveyor CD can actuate the conveyor to move product (line 20-23) or to stop (line 24-27). This CD also sends a response message to the requester via channel ConvRespOCh (line 30). Another CD named TrigCD is described in lines 1-9. This CD invokes the SOSJ programming constructs to trigger dynamic behaviours. In this example, the CD waits for a trigger by human operator via signal OpTrig (line 5). Once the trigger is present, the CD invokes the SJ.CreateCD() to trigger the dynamic creation of conveyor CD (line 6). The same mechanism in line 6 can also be used to ‘overwrite’ an existing CD and replace it with a ‘newer version’ of the CD (e.g. with updated code, data computation, etc.).

Listing 2. Code example showing how to use SOSJ constructs.

```
1 TrigCD()
2   input signal OpTrig;
3   ->{  
4     if (...further CD code...)
5       await(OpTrig);
6       SJ.CreateCD("CB4CD","CBCDMap.xml","CBCDSO.xml");
7       pause;
8     if (...further CD code...)
9   }
10 }
11 ConvCD()
12 output String signal ConvEn,ConvF,ConvB;
13 input String channel ConvReqInCh;
14 output String channel ConvRespOCh;
15 ->{  
16   while (true)
17     receive ConvReqInCh;
18     String str=(String)#ConvReqInCh;
19     String dataAction = SOSJ.GetAction(str);
20     if (dataAction.equals("Forward")){
21       emit ConvEn("high");
22       emit ConvF("high");
23       emit ConvB("high");
24     } else if (dataAction.equals("Stop")){
25       emit ConvEn("low");
26       emit ConvF("low");
27       emit ConvB("low");
28     }
29     if (...further CD code...)
30     send ConvRespOCh(RespMsg);
31     pause;
32   }
33 }
```

V. PERFORMANCE EVALUATION AND COMPARISONS

This section describes experimental results and the performance comparison of SOSJ and JADE.

A. Qualitative comparison

The capabilities of SOSJ and JADE are compared and summarised in Table III.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>SOSJ</th>
<th>JADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Correctness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reactivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deterministic Execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractions Communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with the Environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strong</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weak</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As a typical MAS-based approach, JADE agents are not based on formal semantics or MoC. To determine whether agent implementation behaves according to the specification, JADE programmers are limited to the use of debugging. However, debugging, as a means of software testing, ‘can show the presence of errors but never their absence’ [22]. Even with rigorous debugging effort (which will be more challenging and difficult with the increase in system scale and complexity), there is no assurance of functional correctness of JADE agents due to the lack of underlying formal semantics and MoC. Besides the support for correct by specification design, the software behaviours in SOSJ are amenable for formal verification, providing programmers additional means for guaranteeing system functionality without exhaustive testing.

JADE agents run asynchronously with each other. While individual JADE agents may consist of hierarchical concurrent software behaviours referred to as JADE behaviours, JADE behaviours are also asynchronous with each other. JADE programmers may emulate synchronous-like behaviour by resorting to low-level constructs, e.g., mutexes for synchronization, but they do not comply with a formal MoC. Also, the use of such constructs is considered as one of the main causes of software failures and deadlocks [23], which may not be completely found despite rigorous debugging effort. In contrast, while CDs are asynchronous concurrent each to the other, the underlying GALS MoC in SOSJ enables programmers to describe synchronous concurrent software behaviours (reactions) that execute in lockstep in individual CDs. This also removes the need for manual synchronization using low-level constructs like mutexes, avoiding potential software bugs/failures caused by deadlocks.

As a MAS-based approach, JADE enables programmers to describe software agents which can respond/react to events from the environment. SOSJ also allow CDs to react to events from the environment through signal. However, SOSJ signal mechanism completely abstracts the implementation details of physical interfacing with the environment, allowing programmers to focus on system-level functions. Such feature is not provided by JADE. Also, SOSJ’s built-in constructs provide the design of deterministic (software function-wise) software behaviours and enables safe preemption with explicit scope. These features are important to allow the design of software behaviours for safety-critical scenarios. Meanwhile, such features are not available in JADE.

To satisfy the dynamicity requirement, JADE provides the features to handle dynamic creation, suspension, resumption, termination, and migration of agents. However with regards to migration, JADE only allows for strong migration. Compared to JADE, SOSJ allows not only dynamic creation, suspension, resumption, termination, and strong migration, but also weak migration. In terms of memory footprint, SOSJ requires less memory (787 kB) compared to JADE (2712 kB). The original implementation of JADE framework lacks an important feature to enable agents to migrate between execution platforms/computing machines. Thus, in a scenario when agent migration between execution platforms is needed, programmers are responsible to provide the mechanisms themselves, or alternatively, uses an extra library, e.g. the IPMS (Inter Platform Mobility Service) [24]. The library introduces mechanisms that enable JADE agents to migrate between execution platforms, however the inclusion of the library will incur an additional memory footprint of 237 kB. Meanwhile, SOSJ framework includes built-in mechanisms which allow CD to migrate...
between subsystems on the same or different execution platforms.

B. Experimental Results

For the experiments, the following four scenarios with the allocation of software behaviours to the embedded controllers shown in Fig. 1 are considered. Scenario 1 considers a change from Configuration 1 to Configuration 2 during runtime after CB4 and D1 are introduced and their corresponding services are removed permanently without stopping the software controllers on Controller 6. Scenario 2 considers a change from Configuration 2 to Configuration 1 after CB4 and D1 and their corresponding services are removed permanently without stopping other software behaviours on Controller 6. Scenario 3 considers the temporary halt of CB4 and D1 and their corresponding services for maintenance purpose without stopping other software behaviours on Controller 6. Once the maintenance operation is finished, the software controllers of CB4 and D1 are restarted. Scenario 4 considers the migration of the software behaviours that govern CB4 and D1 from Controller 6 to a new hardware controller named Controller 7 (to distribute computational loads) without stopping the other software behaviours on Controller 6.

Experimental Results

For the experiments, the following four scenarios with the allocation of software behaviours to the embedded controllers shown in Fig. 1 are considered. Scenario 1 considers a change from Configuration 1 to Configuration 2 during runtime after CB4 and D1 are introduced and their corresponding services are removed permanently without stopping the software controllers on Controller 6. Scenario 2 considers a change from Configuration 2 to Configuration 1 after CB4 and D1 and their corresponding services are removed permanently without stopping other software behaviours on Controller 6. Scenario 3 considers the temporary halt of CB4 and D1 and their corresponding services for maintenance purpose without stopping other software behaviours on Controller 6. Once the maintenance operation is finished, the software controllers of CB4 and D1 are restarted. Scenario 4 considers the migration of the software behaviours that govern CB4 and D1 from Controller 6 to a new hardware controller named Controller 7 (to distribute computational loads) without stopping the other software behaviours on Controller 6.

The experiments aim to find the average reconfiguration time calculated over 10 executions of dynamic behaviours that achieve the above scenarios on 50 and 100 identical software behaviours of each CB4 and D1 (CDs in SOSJ, agents in JADE), each implementing their respective manufacturing functions. The Raspberry Pi 2 B running Linux OS is chosen as the execution platform. Scenario 1 and 2 are achieved through the dynamic creation and termination of software behaviours that govern CB4 and D1 during runtime, respectively. Scenario 3 is realized through dynamic suspension and then resumption of software behaviours governing CB4 and D1. Scenario 4 is achieved through the (strong) migration of software behaviours governing CB4 and D1 (JADE doesn’t support weak migration). The results of dynamic behaviours handling to achieve Scenario 1 and 2, Scenario 3, and Scenario 4 using SOSJ and JADE are shown in Fig.8, Fig.9, and Fig.10, respectively. The charts in these figures include range bars that show the average time and its variation for the value of the standard deviation.

From the overall results, it appears that the number of agents affects JADE performance. This can be seen from the ratio of increase of the average time in achieving all dynamic behaviours and also the variations of the time, as the number of agents increases. This can be attributed to the fact that JADE applies pre-emptive scheduling of agents, which affects the overall scalability of JADE and the time for JADE to complete the execution of dynamic behaviour handling functionalities. This doesn’t apply to SOSJ, which has the dynamic behaviour handling functionalities executed outside the execution time of CDs, i.e., the housekeeping phase.

With regards to dynamic creation, pre-emptive scheduling of agents handled by the same runtime environment significantly affects the time to complete creation & initialization of new agents (denoted by ‘Creation of Agents Objects & Initialize Agents’), and the time to register them to JADE Agent Management System (AMS) & starts their execution (denoted by ‘Start Agents Execution’). Meanwhile, SOSJ RTS is much less affected by the number of CDs than JADE. The dynamic creation in SOSJ requires less average time compared to JADE, with high proportion spent for initializing CD configuration and service description of new CDs. Meanwhile, SOSJ shows comparable performance compared to JADE in achieving dynamic termination. Note that in Fig.8, TerminateCD() represents the operation to remove the CD descriptor from the scheduler, terminate the threads handling the physical interface of input signals, and preempt any occurring channel communications in the CD, while AddCD() represents the operation to instantiate signals and channel of the CD and include the CD to the scheduler for execution.

With regards to both dynamic suspension and resumption handling, JADE runtime requires more time (compared to SOSJ RTS) to exclude agents from execution and store their data and execution state (during suspension) and to retrieve agents data and execution state and then include the agents for execution (during resumption).

With regards to migration, the benchmark performs behaviour migration in JADE using two different approaches to observe any differences in the framework’s performance for both cases. The first is to trigger behaviour migration for all agents to migrate after all agents have been created, indicated as “JADE (A)”, and the second is to trigger behaviour migration for the agent to migrate after it has been created, indicated as “JADE (B)”. Meanwhile, the benchmark performs behaviour migration in SOSJ by initiating a single migration and migrates all the CDs to the destination. As shown in Fig.10, SOSJ requires shorter average time to complete migration than JADE. SOSJ uses TCP/IP stream-based communication to transfer data
and codes to keep overheads at minimum. Meanwhile, JADE uses Agent Communication Language (ACL)-based message (of the FIPA standard [25]) to transport agents and their codes, which contributes to its high performance overhead.

This paper presents new additions, particularly all new runtime support, to the SOSJ framework that enables full dynamicity of software behaviours (dynamic creation, suspension, resumption, termination, and migration). Based on a reconfigurable automatic bottle capping and storage system as a motivating example from manufacturing domain, the essential design and operation requirements to support a wider range of dynamic options are identified and introduced. Example benchmarks show the capability of SOSJ framework to handle different dynamic scenarios using the new extensions with no need for entire system restart compared with a MAS-based framework JADE. Future works will include introducing built-in security features and support for hard real-time behaviours within the framework. We also plan to investigate the performance of the framework compared to other approaches, and how to map (translate) a standard IEC 61499 framework which has reconfiguration provisions such as those in [13] and enhanced with features proposed in this paper on SOSJ framework.

VI. CONCLUSIONS

A rather interesting result is the difference in performance between the “JADE (A)” and “JADE (B)” case. The results show that performance of “JADE (A)” is significantly lower than “JADE (B)”, which can be caused by either (or both) of the following reasons: (1) the overall performance of JADE decreases when a high number of agents are handled by the same runtime environment and (2) the performance of the ACL message processing in JADE decreases when higher number of agents are queued for migration. Another factor which makes the migration in SOSJ advantageous compared to JADE is that SOSJ migration handles the transfer of any dependencies which are used by the migrating CDs, while in JADE, this feature is not intrinsic in the framework. In JADE, programmers are responsible in ensuring any dependencies required by the migrating agents are available at the destination to use (by the migrating agents) once the migration process is finished.

REFERENCES


**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 Ph.D degree in Electrical and Electronics Engineering from the University of Auckland, New Zealand in 2017. He is currently a Postdoctoral Fellow at Aalto University, Finland. His research interests include software architectures and paradigms for cyber physical systems, embedded systems, and industrial automation.

**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 Ph.D (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.

**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.