Preparation of Papers for IEEE Trans on Industrial Electronics (Mar. 2019)

First A. Author1, *Student Membership*, Second B. Author2, *Membership*, and Third C. Author3, *Membership*

***Abstract*—These instructions give you guidelines for preparing papers for IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS. PLEASE NOTE, ALL INFORMATION IN THIS TEMPLATE WRITTEN IN THE RED TEXT SHOULD BE OMITTED FOR PEER REVIEW AS TIE PERFORMS A DOUBLE-BLIND REVIEW. IF YOUR PAPER IS ACCEPTED, WE ASK AUTHORS TO INCLUDE THE RED TEXT INFORMATION IN THE FINAL FILES PDF. Use this document as a template. The electronic file of your paper will be formatted further at IEEE. Paper titles should be written in uppercase and lowercase letters, not all uppercase. Avoid writing long formulas with subscripts in the title; short formulas that identify the elements are fine (e.g., "Nd-Fe-B"). Do not write "(Invited)" in the title. Write full names of authors are required in the author field. Define all symbols used in the abstract. Do not cite references in the abstract. Do not delete the blank line immediately above the abstract; it sets the footnote at the bottom of this column.**

***Index Terms*—Enter key words or phrases in alphabetical order, separated by commas. For a list of suggested keywords, send a blank e-mail to keywords@ieee.org or visit http://www.ieee.org/documents/taxonomy_v101.pdf.**

I. Introduction

INDUSTRIAL Control System(ICS) has got extensive use in the manufacture and mechanical control, especially some critical infrastructures like water conservancy facilities and nuclear power plants. With the growing scale and complexity of ICS, it is of great importance to bring software engineering methods into the development and integration of ICS, in order to meet the evolving industrial requirements and reduce the cost.

Just as the manufacturing industry uses pre-defined standards to produce replaceable components, component-based software engineering (CBSE) improves the flexibility and robustness of the system by treating it as a combination of subcomponents rather than a monolithic entity [1], [2]. Compared with traditional software development methods, CBSE has significant advantages in terms of improving reusability and portability, and shortening time to market. In addition, CBSE has great potential in enhancing the system performance and specification by modular modeling, which helps to control dependencies between components. The cornerstone of CBSE is based on component models that specify what components are, how they can be constructed, deployed, executed, and how the components interact with each other. Component models have been widely adopted in general-purpose software domains to date, including EJB (Enterprise Java Beans), CCM (CORBA Component Model), UML, etc. Even in the field of embedded systems where security and predictability are more demanding, CBSE is considered as a promising technology.

In recent years, concepts such as intelligent robotics, Industrial-Internet-of-Things (IIoT) and Cyber-Physical Systems (CPS) have been prevailed in recent years, which brings rich features along with great challenges into current ICS, particularly when adopting CBSE. As the complexity and functionality keep booming, the software development in industrial domains often requires multi-team and multi-vendor collaboration, asking for great interoperability (cross-platform and cross-company) based on a unifying principle. To achieve complex functions and features, high-level programming languages like python, c++, Java are preferred when implementing components. It is desired that the component model can provide support of multiple programming languages and standardized interfaces for implementation and integration. In addition, on account of the harsh working environment of ICS like long-time high-power operation and electromagnetic disturbance, both the hardware and software are prone to failure, often resulting in frequent replacement or upgrade. However, the downtime cost of updating by rebooting the whole system tends to be intolerable for ICS, especially for the safety-critical ones. Hence, it is desired that software components can be dynamically updated on the fly.

In this paper, we introduce a novel component model, OSAICM (Open Software Architecture for Industrial Component Model), applying to the development of industrial control systems. The model is, on the one hand, rich enough to fit major industrial requirements and on the other hand, general enough to be applicable in most industrial control systems. More narrowly, OSAICM components are well defined in

Manuscript received Month xx, 2xxx; revised Month xx, xxxx; accepted Month x, xxxx. This work was supported in part by the xxx Department of xxx under Grant (sponsor and financial support acknowledgment goes here).

(Authors' names and affiliation) First A. Author1 and Second B. Author2 are with the xxx Department, University of xxx, City, State C.P. Country, on leave from the National Institute for xxx, City, Country (e-mail: author@domain.com).

Third C. Author3 is with the National Institute of xxx, City, State C.P. Country (corresponding author to provide phone: xxx-xxx-xxxx; fax: xxx-xxx-xxxx; e-mail: author@ domain.gov).

the whole life-cycle from design time to run-time, supporting for multiple instantiation. In addition, we adopt an XML-like language to describe dependencies between components clearly, including assembly inter components and composition in the components. With the unified interface standard, the model facilitates the collaborative development of manufacturers, improving interoperability and causing easy integration. In the meanwhile, clear and well-defined component interfaces are also critical to software reusability. Besides, as a logic abstraction, the virtual bus provides all of services required for virtual interaction between components. Based on the publish/subscribe model, we use pluggable message middleware to implement multiple communication patterns for its fast communication speed. With the structure of component manager, systems could be less rigid and more reconfigurable.

II. Background and Related Work

A. Component Models

CBSE has been studied since the end of last century, with some ideas proposed on what features a perfect component model should possess. Kung-Kiu Lau and Zheng Wang [3] gave a taxonomy of component models and revealed an idealized component life-cycle, discussing desiderata of CBSE: components are pre-existing reusable software unit that can be produced and used by independent parties; components are able to be composed into composite components and copied and instantiated at deployment. The authors of [4] emphasized three key factors of component models: dynamic reconfigurations, multiple communication styles and non-functional aspects of components (e.g., life-cycle management). In [5], a comparison was made among some component frameworks for real-time embedded systems based on a set of criteria such as supports for distributed applications, resource constraints and real-time attributes(e.g., periodic tasks). There is also a gap between academic and industrial component models where design-oriented features are often difficult to balance with sufficient runtime support [4].

There are a few general-purpose component models existing, to name several major ones, such as JavaBean/EJB, COM/.NET, and CCM. However, models from the general-purpose domain always only provide basic mechanisms for specification and construction of components without considering specific requirements. As the concept of CBSE has received more and more attention in the embedded field, some general component models cannot be applied without significant modifications to address the complex and specific requirements of the embedded environment, resulting in a number of component models designed or evolved for the embedded field. For example, the Rubus component model [6] supports three important activities in resource-constrained and time-critical embedded real-time systems. μ-kevoree [7] puts forward the four-layer concept of the Dynamic Adaptation of component models and makes improvements in its Dynamic Adaptation capability for Cyber Physical System(CPS). Furthermore, some large application domains like automotive, robotics, and aerospace, also see benefits of CBSE and some domain-specific models are proposed.

As an open and standardized software architecture enjoying extensive concern and broad acceptance in the automotive domain, AUTOSAR (AUTomotive Open System ARchitecture) [8] was a large source of inspiration. The main focus of AUTOSAR is the standardization and interoperability of software components, which is exactly one of our pursuits. To provide smooth integration and easy reuse of components from different vendors, AUTOSAR supports multi-type interfaces and multi-instantiation. AUTOSAR defines Virtual Function Bus (VFB) to support interaction between components and hide implementation and communication details. Thus, the system can be integrated early without regard to component physical allocation. AUTOSAR Software Components are compiled and linked into specific executable, losing some benefits of the component-based approach during run-time [9]. In addition, AUTOSAR does not give much thought to component dynamics.

Up to now, numerous component models have been proposed each with their own characteristics and advantages, but few of them had a comprehensive consideration or inclusion of aforementioned features.On the other hand, studies on such component models do not seem to be evolving rapidly these years, which may capture more attention from researchers [1].

B. Publish/Subscribe Communication Paradigm

As an efficient information diffusion paradigm, the publish/subscribe model has recently received increasing attention. The key feature of this paradigm is that the publisher and subscriber are unaware of each other. The publisher sends messages without knowing names, or even existences, of the subscribers. And the subscriber receiving messages is also not aware of the publisher out there. Based on this loose coupling feature, applying pluggable communication middleware based on pub/sub model to the component model makes it relatively easy to add, remove, or rewire components.

The Data-Centric Publish-Subscribe (DCPS) model extending the publish/subscribe model is based on the concept of ”global data space” and facilitates efficient data transport.

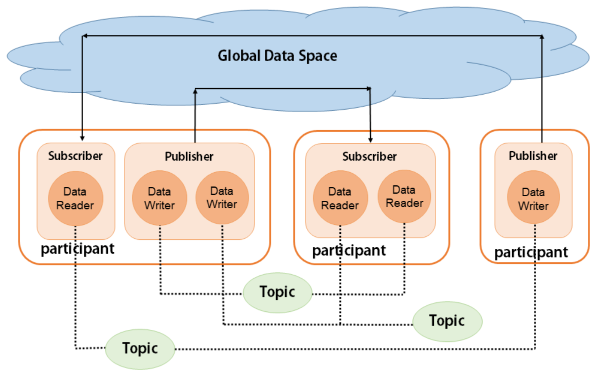

Fig. 1. DCPS Model

As shown in Fig. 1, the DCPS model consists of entities as follow:

1) DomainParticipant. DCPS is constructed of one or more data domains, every domain contains a set of participants.

A domain participant represents the following other entities.

2) Publisher. A publisher sends data to one or more topics of different data types with owning and managing one or several data writers.
3) Subscriber. A subscriber receives and dispatches data for different topics on behalf of one or more data readers
4) Data Writer. A data writer is a typed facade to a publisher, sending data of a given type.
5) Data Reader. A data reader is attached to a subscriber, used by a domain participant to receive and access data.
6) Topic. A topic associates data-object between a data writer and a data reader, with a unique name, a data type, and a set of QoS. The type definition provides enough information for the service to manipulate the data.

The flexible configuration of communication is performed according to corresponding QoS (Quality of Service) policy that the system uses to specify reliability, resource usage, and other required services. In addition, the discovery mechanism of DDS enables the middleware to automatically connect when new participants appear.

There are some traditional middleware technologies, such as Web Services, OPC (OLE for Process Control) and CORBA (Common Object Request Broker Architecture), but most of them are based on the Client/Server model, do not adapt properly to characteristics of dynamic processes. That's why we choose DDS (Data Distribution Service) as message middleware. DDS is an emerging specification based on the pub/sub model which was released by the OMG (Object Management Group). With features of low latency, high throughput, and controllable transmission performance, this technology has been adopted in many fields, like traffic control systems, aerospace, robotics, and IoT. For example, the ROS2 (Robot Operating System), an open-source middleware widely used in robotics domain, adopt the use of DDS compared to ROS1, in order to deal with increasingly complex robot systems [10]. In [11], authors build IEC61499 Service Interface Function Blocks (SIFB) based on DDS.

C. A Case Study

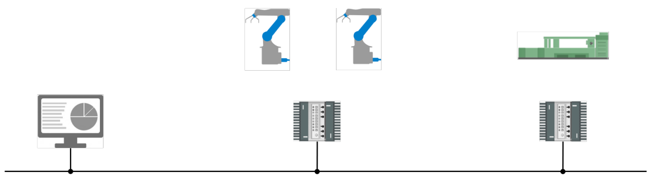

Fig. 2. Case Study

III. OSAI Component Model

In this section, we present the OSAI Component Model and illustrate how the components are composited, instantiated, executed, and how components interact with each other. The OSAI Component Model distinguishes components from their instance and draws on the object-oriented paradigm. By giving the description of life-cycle, component elements, and the Unified Dynamic Bus, we combine the general-purpose software engineering methods with the "OEM-Vendor" industrial cooperating pattern and give a practical component model with both academic and industrial features.

A. Component Construction

A component of the OSAI Component Model is called Application Component (AC). To adopt the object-oriented paradigm, one key feature of the OSAI Component Model is that AC can be differentiated between *AC Type* and *AC Instance*, and an AC Type will be instantiated into one or multiple AC Instances at run-time. In terms of composition, ACs can be composed at type-level, bringing two kinds of AC Types: *Atomic AC Type* and *Composite AC Type*. A Composite AC Type aggregates existing AC Types (atomic ones and composite ones). In fact, the composition in OSAI component model groups existing components and gives a hierarchical view when designing logical software architecture, and then, to reduce the complexity, a Composite AC Type is instantiated and treated as a single component in the components interaction, which was considered as a trend in [].

An AC Type aggregated within a Composite AC Type is called *AC Type Reference*, which implements the usage of the AC Type and can be duplicated. An Atomic AC Type encapsulates the functionalities and internal behaviors and interacts with the outside world through exposed interfaces. In our opinion, the artifacts implemented inside an individual Atomic AC essentially include four parts as follow:

1) Operation: Under the semantic of a synchronous client/server communication pattern, a component may be equipped with operations that can be requested by other components. An AC can request operation that another AC provides or response to the results of its operation execution by utilizing the *request-reply* interface.
2) Event: Event-driven interaction is also desired for the loosely coupled software components. In the OSAI Component Model, an AC has the capacity to trigger or react to some specific types of events.
3) Data Area: An AC may contain data areas with different types of data. Data can be transferred between different ACs' data area in an asynchronous sender/receiver communication pattern. In addition, the data area can also be mapped with other data areas or IO area of the device for data synchronization.
4) Service: In the run-time environment of OSAI, an AC has no direct access to the underlying operating system. In order to use a mechanism like threading, code fragment of an AC's internal behavior that need to be executed cyclically are wrapped into an artifact called *Service*. Services can be scheduled by run-time infrastructure and mapped with operating system tasks.

The OSAI Component Model also defines 4 types of port to identify the entry points of the ACs for data, event and operation request/reply transfer. As shown in figure x, ports are distinguished between provided ports and required ports.

Fig. 3 gives an example of constructing a Composite AC Type named *COETransport* composing two Atomic AC Type

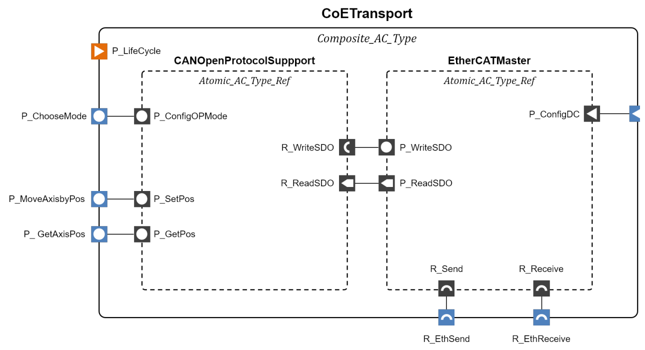

Fig. 3. An example of an AC Type

References: *CANopenProtocolSupport* and *EtherCATMaster*. *COETransport* provides support for communicating with servo drives that control the axises on a robot arm through EtherCAT (quote); *EtherCATMaster* plays the role of a master protocol stack of EtherCAT that interacts with the EtherCAT slaves (servo drives); *CANopenProtocolSupport* provides functionalities to support CANopen (a high-layer communication protocol for embedded systems used in automation) over EtherCAT (CoE). Since the Composition AC Type only exposes the ports on its surface, the *AC Type Reference* will interact with the outside world through the port *delegation* and communicate with another *AC Type Reference*, inside the same Composition AC Type, through port *binding*. Such two concepts are both implemented at type-level. In the scene of robot motion control, instructions for moving an axis or configuring the operation mode will be handled by the *COETransport AC Type* by invoking the request-reply ports which are delegated to ports of *CANopenProtocolSupport AC Type Reference*. These instructions will be transformed and wrapped into SDO (Service Data Object) packages that accord with the object dictionary of CiA 402 (CANopen device profile for motion controllers and drives released by *CAN in Automation*[1]). Then these SDOs, which are written to the *EtherCATMaster AC Type Reference* through the port binding, are transferred to the EtherCAT slaves via EtherCAT along with the process data organized in PDOs (Process Data Objects). Notice that every AC Type, besides the ports exposed from inner artifacts or port delegation, will implement an extra port for receiving events that manage the lifecycle after instantiated at run-time.

B. Lifecycle

In [], the authors proposed an idealized life cycle of software components consisting of three phases: *design*, *deployment* and *run-time*. We bring these concepts into the development workflow of ICS. Figure 4 describes the three phases and several stages when constructing a distributed system based on OSAI.

In the first place, OEM, the integrator, defines the hardware architecture and model the software architecture of the system. By mapping the software architecture with the hardwares,

[1] ***CAN in Automation (CiA)*** is the international users' and manufacturers' organization that develops and supports CAN-based higher-layer protocols.

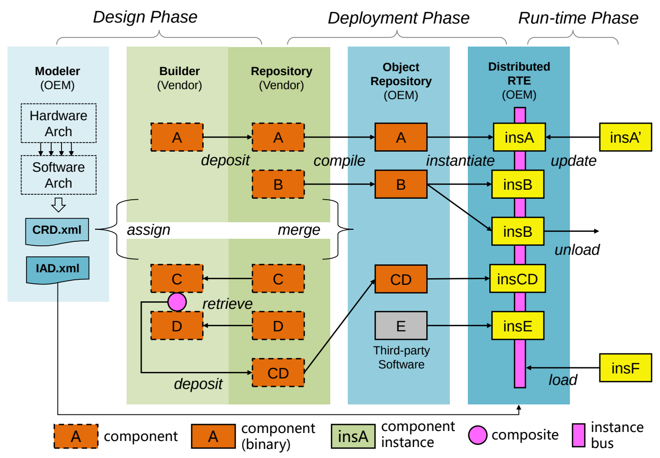

Fig. 4. Deployment workflow of OSAI

OEM can determine how many and what kind of components (ACs) they need and what run-time supports these components require. Consequently, two kinds of XML-based files will be generated from the architecture: Component Type Definition (CTD) and Instance Wiring Description (IWD). CTD describes properties of the AC Type and specification of ports on its surface, remaining the internal contents and behaviors unimplemented; IWD gives an overall perspective about how AC Instances are assembled with the connection among ports. In a collaborative development context, the design phase is often conducted by multiple developers (or vendors). CTD files are assigned to vendors for the development of each AC's internal artifacts. Developers design and implement AC Types independently based on the standardized interface definition (based on XML descriptor) and then store components in respective repositories. ACs can be retrieved from the repository and composed into Composite AC Types, which can also be deposited back for further retrieve and reuse, that conform with the specification in CTD. During the deployment phase, ACs are compiled into object code for the target platform and delivered to the integrator (or OEM). The CTD files are also merged back including a complete XML-based description of internal behaviors and AC Type References of each AC Type. To build the control system, the integrator retrieves components from the object repository and deploy them into the run-time environment (RTE) along with the IWD file which specifies the ports connection and initialization parameters for instantiation. An AC may have one or multiple instances. These instances communicate with each other at run-time via *Unified Dynamic Bus*, which will be detailed later. RTE of OSAI Component Model has the ability to, by means of UDB and *Component Adaption Manager*, have AC Instances loaded, unloaded or updated.

IV. OSAI Run-time Architecture

OSAI proposes a unified architecture for diverse devices, multiple platforms, and distributed features in the industrial application scenarios, to guarantee the deployment and execution of AC instances, and interactions between AC instances are implemented by a united bus. The run-time architecture of

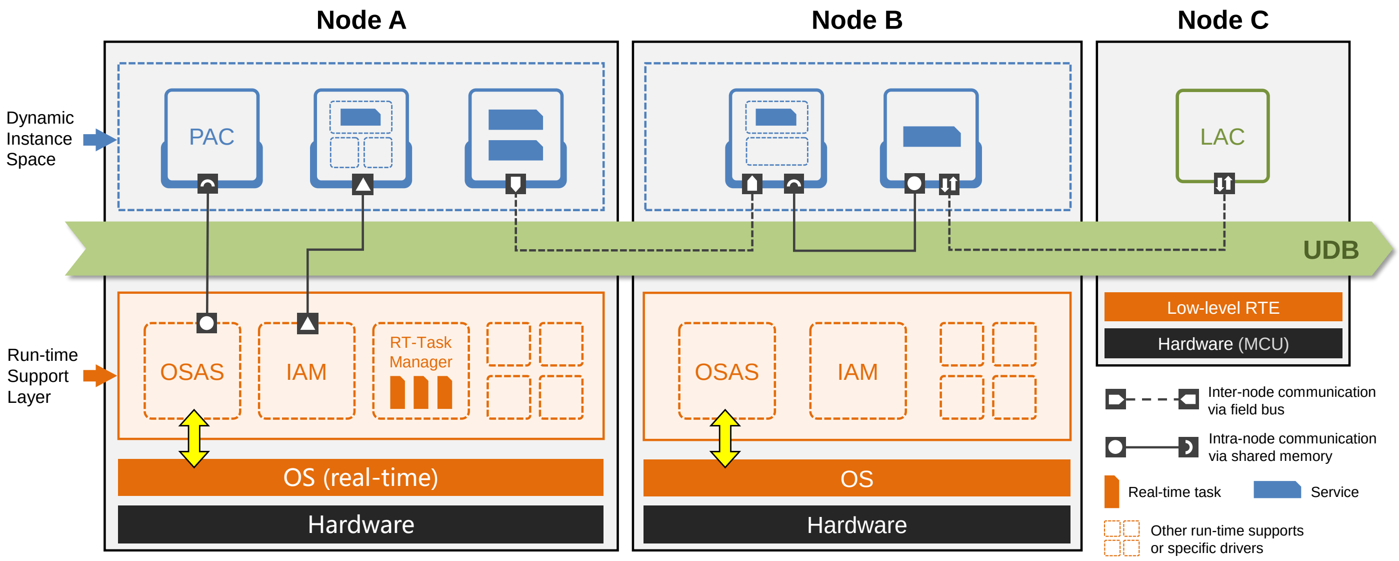

Fig. 5. Architecture

OSAI aims at a variety of devices with different performance configurations (the heterogeneous characteristic) in ICS, including both high-profile devices that can deploy OS(real-time OS included) and small-scale devices based on MCU. ACs deployed on these two types of devices are divided into PAC and LAC accordingly. In the distributed architecture of OSAI, each embedded device is called a *Node*. LACs are deployed on small MCU devices, typically there is a single LAC in such a Node, along with a specific run-time environment for this LAC at the low level, such as the IEC 61131-3 runtime. As for devices with powerful capability, OSAI defines a hierarchical architecture including Dynamic Instance Layer and Runtime Infrastructure Layer.

1) Dynamic Instance Layer
This layer is where all of ACs are deployed and executed, like IoC container in Spring, ACs are loaded into this layer. Based on the IWD file, an AC Instance is instantiated by an AC Type, which then connections between each other are established. A PAC instance is wrapped by a docker-like container to perform the isolation between components, and communications among containers are implemented via OSAI Connection. Components in this layer are under the management of CAM (Component Adapter Manager) in Run-time Infrastructure Layer, a particular component responsible for reading IWD files, instantiating components, linking connections of ports, bounding interface events, and triggering events that control component run-cycle.
2) Run-time Infrastructure Layer
Run-time Infrastructure Layer runs above OS and contains modules for run-time support. It is a basic layer that provides a configurable RTE for the deployment and execution of PAC including some device-specific IO drivers. Modules in the Run-time Infrastructure Layer are scalable for different Nodes, and these modules interact with the PACs in the Dynamic Instance Layer via the UDB. Specifically, the data passing or function calls between a PAC and module in the Run-time Infrastructure Layer are through the similar ports connection as the communication between PACs.

A. Unified Dynamic Bus

B. Dynamics

In order to better perform the component lifecycle management in runtime and support dynamic properties of the component model (i.e., loading, unloading and updating components on the fly), each component provides a state machine to describe state transitions of component instances during service life.

The state machine defines possible states that an instance of a component can experience as INIT, LOADED, READY, ACTIVE, DEACTIVE, DELETED, and BLOCKED, UPDATED needed for dynamic updating, as shown in Fig. 6. The dynamic behavior of a component instance at runtime is as follows: Each component has a lifecycle interface of event-triggered type, which enables migrations between component states by calling operations on that interface, such as *loaded2ready()*, *active2deactive()*, and so on. (1)-(6) state transitions are the processes of a component from start to end in general, and (7)-(9) are additional parts that components go through when they need to be updated dynamically at runtime.

1) In the INIT state, the instantiation takes place, parameters are initialized, and the lifecycle interface is created. At the same time the component registers and loads with the help of CAM, providing interface information, and resources are allocated by the system.
2) By calling *loaded2ready()*, the corresponding DDS entities with QoS policies are created according to the component interfaces. Established connections with other components, the component waits to run.
3) The role of *ready2active()* is to make the component instance start working by binding services to runtime

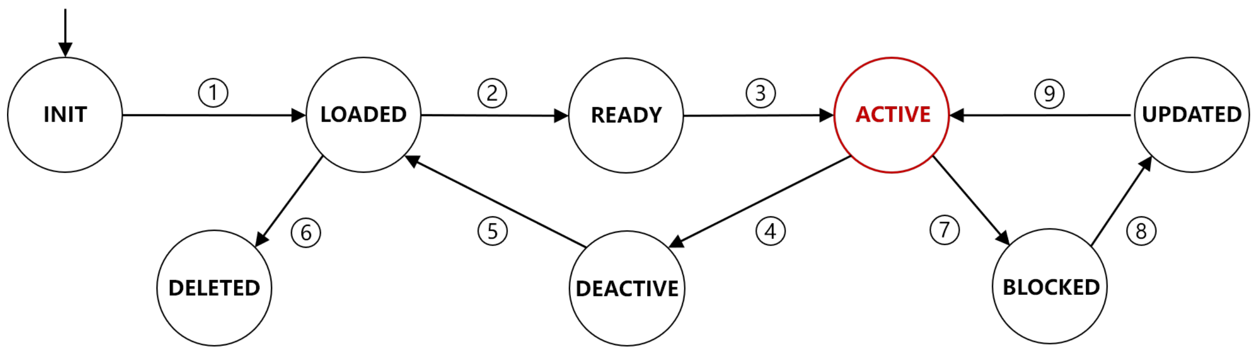

Fig. 6. State machine

tasks. In the ACTIVE state, the component is fully activated and can communicate with other components.

4) When the component execution is completed without updating requirement, the *active2deactive()*operation executes, CAM refuses new invocations for the component and does not deactivate the component until the current invocation is processed.
5) After the component completes the execution of the current invocation, the DDS entities are deleted, components are disconnected, services and tasks are unbound, and resources are released.
6) Calling the operation *loaded2deleted()*, the component unloads with logout of interfaces.
7) Especially, if the component needs to be updated, CAM blocks new invocations and communications for this component, while the old component continuing to process the current invocation.
8) Once the component that needs to be replaced completes the current execution, with the operation *blocked2updated()*, CAM unties services with tasks, unlink connections, replaces the new component instance, redirects the reference, resets connections, and dynamic updating completes.
9) Now wake up invocations that previously blocked, the new component return to the ACTIVE state continues to run after binding services to tasks.

As a component model applies to ICS, in order to better deal with system failures or upgrade requirements when the system is on running, we provide enough support to component dynamic updating, which is a great help in bug fixing, adding functionalities, runtime optimization, and avoiding the huge cost of downtime. In fact, we benefit a lot from choosing the DCPS model as message middleware to implement the dynamic update. Flexible configuration of QoS policies and the discovery mechanism make it easy to reconnect components, with minimal or no impact on other components and the system. In the DCPS model, each entity has a set of QoS policies to describe data transmission. The corresponding QoS policies can be configured to properly constrain the communication cycle, reliability, resource allocation, etc. There are some QoS policies that provide effective support for dynamic update of components, such as:

1) durability -specifies whether or not the context will store and deliver previously published data to new entities.
2) liveness -specifies and configures the mechanism that allows the DataReader to detect when the DataWriter becomes disconnected or ”dead.
3) resource limits -controls the amount of physical memory allocated for entities and how they occur if the dynamic allocation is allowed.

To be specific, QoS policies can be used to block data when the component receiving messages needs to be updated dynamically. After the update operation starts, the old component exists, i.e., the original DataReader offline. DDS uses the durability QoS policy to preserve the unreceived data on permanent storage. The update finishes and the new DataReader broadcasts the subscription, DDS sends the staging data. It is similar if there is a component gets a need for dynamic updating when it sends messages. After the old component is removed, DataReader detects the disconnection of the original DataWriter via the liveness QoS, so it begins to receive the data sent by the new component.

V. Evaluation & Implementation

A. Comparison

In order to reflect the characteristics and evaluate the properties of the OSAI component model more systematically, we selected some component-based approaches for comparison. We chose both domain-specific and general-purpose models to make this comparison more meaningful. Following the dimensions in basic requirements and advanced features, the information of these component models is collected in table I. From the contract, we can summarize these models roughly as follows (in alphabetic order) :

1) AUTOSAR (AUTomotive Open System ARchitecture) [] is a standardized software architecture in the automotive field. The standard supports both client-server and sender-receiver communication modes. AUTOSAR provides hierarchical architecture at design-time and supports multiple instantiation at runtime.
2) CCM (CORBA Component Model) [] is a general-purpose flat component model based on the CORBA middleware. Thus components are not tied to any particular language or platform as long as the CORBA middleware is used. Component interfaces are described

TABLE I
COMPARISON OF SOME COMPONENT-BASED APPROACHS

Component models	domain-specific	hierarchical architecture	interface definition	platform independent	language	dynamics	communication styles	multiple instantiation	support for periodic tasks
OSAI	yes	design-time	xml-based	yes	language independent	yes	multiple communication styles	yes	yes
AUTOSAR	yes	design-time	c header files	yes	C	no	request-response,sender-receiver	yes	yes
CCM	no	no	CORBA IDL	yes	language independent	yes	request-response,triggering	no	no
IEC61499	yes	no	N/A	yes	language independent	no	triggering,pipe&filter	no	yes
RUBUS	yes	design-time	c header files	no	C	no	pipe&filter	no	no
SOFA 2.0	no	fully	java	yes	java	yes	multiple communication styles	no	no
TwinCAT	yes	no	c++ header files	no	C/C++	yes	request-response,dataarea mapping	yes	yes

in the CORBA IDL and they provide synchronous or asynchronous communication.

3) IEC 61499 []provides a component solution for automation and distributed control systems extended by IEC 61131. The execution of components called Function Blocks(FBs) in this standard is event-driven and data transmission is via pipefilter way. The implementation platform and language with IEC 61499 are also not limited.
4) Rubus [6] is a domain-specific component model focusing on real-time properties and resource-constraint problems of embedded systems. It runs on the base of the Rubus OS. Implementation is done by C and communication style is pipe&filter.
5) SOFA 2.0 [4] is an extension of the SOFA(Software Appliances) component model with hierarchical architecture and features like dynamic reconfiguration, control parts, and multiple communication patterns. SOFA 2.0 has to be developed by Java language, so it is only compatible with platforms supporting JVM(Java Virtual Machine).
6) TwinCAT(The Windows Control and Automation Technology)[2] is a PC-based automation software developed by Beckhoff. TwinCAT offers TcCOM (TwinCAT Component Object Model) derived from the COM (Component Object Model) to define characteristics and behaviors of application modules. Modules in TwinCAT support multiple instantiation and periodic tasks. However, the run-time environment of TwinCAT relies on specific hardware devices.

From the table, we can observe that the OSAI component model not only possesses characteristics like hierarchical design and the independency of language and platform, but also provides advanced features including multiple communication styles based on the DCPS model, multiple instantiation, and run-time dynamics. Equipped with comparatively comprehensive features above, the OSAI component model, to some extent, can be applied to embedded distributed systems in more domains, not merely for ICS.

B. Analysis

C. Case Implementation

VI. FUTURE WORK

VII. CONCLUSION

REFERENCES

[1] L. Neto and G. M. Gonçalves, “Component models for embedded systems in industrial cyber-physical systems,” in *INTELLI 2018: The Seventh International Conference on Intelligent Systems and Applications*, 2018.
[2] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, “Towards component-based robotics,” in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 163–168. IEEE, 2005.
[3] K.-K. Lau and Z. Wang, “A taxonomy of software component models,” in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 88–95. IEEE, 2005.
[4] T. Bures, P. Hnetynka, and F. Plasil, “Sofa 2.0: Balancing advanced features in a hierarchical component model,” in *Fourth International Conference on Software Engineering Research, Management and Applications (SERA’06)*, pp. 40–48. IEEE, 2006.
[5] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava, “Comparison of component frameworks for real-time embedded systems,” in *International Symposium on Component-Based Software Engineering*, pp. 21–36. Springer, 2010.
[6] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback, “The rubus component model for resource constrained real-time systems,” in *2008 International Symposium on Industrial Embedded Systems*, pp. 177–183. IEEE, 2008.
[7] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, “A dynamic component model for cyber physical systems,” in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pp. 135–144. ACM, 2012.
[8] “Autosar,” https://www.autosar.org/.
[9] J. Feljan, L. Lednicki, J. Maras, A. Petričić, and I. Crnković, “Classification and survey of component models,” 2009.
[10] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ros2,” in *Proceedings of the 13th International Conference on Embedded Software*, p. 5. ACM, 2016.
[11] I. Calvo, F. Pérez, I. Etxeberria, and G. Morán, “Control communications with dds using iec61499 service interface function blocks,” in *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pp. 1–4. IEEE, 2010.

[2]http://www.beckhoff.com.cn/cn/default.htm?twincat/