Contract-Based Approach to Develop Resilient Cyber-Infrastructure in Industry 4.0

Dilleep Kumar Bhadra,
College of Engineering Bhubaneswar

***Abstract*— The consequences of disruption from internal or external defects are worse as the industrial cyber-infrastructure becomes more crucial to achieving Industry 4.0 goals. As a result, a robust infrastructure is required. In this letter, we suggest a contract-based methodology in which a lightweight resilience manager and contracts are used to link components across levels of the cyber-infrastructure. This enables the system to efficiently identify errors (contract violations tracked by observers) and respond (contract changes made dynamically).**

INTRODUCTION

INDUSTRY 4.0 has the potential to radically improve the productivity of manufacturing systems. The next generation of smart factories can perform more efficiently, collectively, and resiliently [1]–[3]. A resilient system has the ability to maintain and improve services even when challenged by failures and evolutionary changes. It indicates a system to be more flexible, more dynamic and less prescriptive than a traditional fault tolerant system [4].

Resiliency can be obtained using software or hardware-based fault tolerance mechanisms. The latter solution demands physical redundancy such as replication of computational and communicational resources. For software-based solutions, we find techniques such as replication of programs, checkpoints that act as restoration points and monitoring services that rely on timestamps [5] and fault trees, to detect faults. More recently, we see software adaptation techniques that switch between off-line generated code based on ontology [6] or application graphs [7], or dynamically create/modify code at runtime [2], [8].

Our cyber-infrastructure architecture follows the classical layered middle-ware with three different layers [3] (see Fig. 1). The physical layer comprises physical components such as sensors, actuators, controllers, and communication hardware. The platform layer embodies computational and communicational platforms such as operating systems and network managers. Finally, the application layer accommodates the software components which describe the behavior of an application.

Scott *et al.* [5] and Denker *et al.* [9] discussed an outline for a holistic approach for developing a resilient cyber-infrastructure that manages applications, devices, resources, deployment, security, time etc. However, [9] does not present any architecture details. In contrast, RIAPS [5] describes an architecture for developing a distributed resilient cyber-physical system. As an example, a resilient discovery service (DS) was explored; a failure of a publisher/subscriber pair is detected using heartbeat signals and timestamps. When a failure occurs, DS de-registers the pair from the list of registered services. A publisher/subscriber needs to reregister once they become active. The process of de-registering and reregistering is very time consuming and it also needs to be communicated with neighboring nodes. In this scenario, the cause of the failure (e.g., intermittent fault in the physical layer) and the expected recovery time (e.g., 2 s or 2 h) were not available to the DS manager residing in the platform layer. If the recovery time information was available to the DS manager, it can choose not to de-register a publisher/subscriber and avoid the unnecessary time consuming registration process across all neighboring nodes. Thus, there is a need for communication across layers to improve resiliency.

A. Problem Statement

Existing resiliency techniques are inefficient due to two key limitations.

1) They do not consider cross-layer interactions of a cyber infrastructure [2], [5]–[8].
2) The techniques depend on a centralized decision-making component which collects information from other distributed components to detect faults [6], [7], except some which are tightly coupled to an industry standard [2], [8].

Both limitations (*lack of cross-layer communication* and *centralized decision making*) are not acceptable for Industry 4.0, where the cyber-infrastructure is heavily distributed and encourages attributes such as *self-awareness* where even the edge nodes of the network are expected to make decisions [1], [2]. This is unlike most existing pyramid-like infrastructure with predefined hierarchy of connections from sensors/actuators at the lowest levels and decision making software at the highest levels. Hence, there is a need for a new

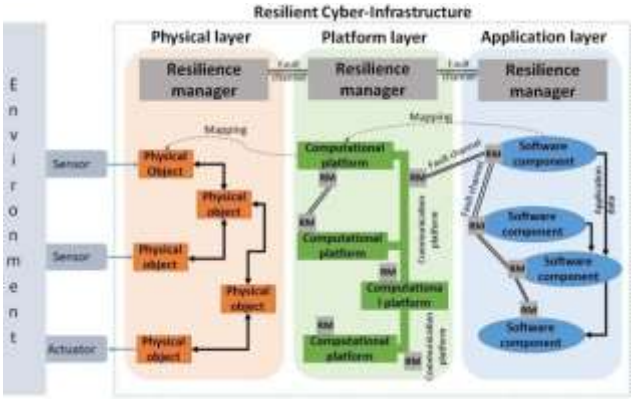

Fig. 1. Proposed contract-based methodology for resiliency. It shows a distributed resiliency management at component-level and across layers.

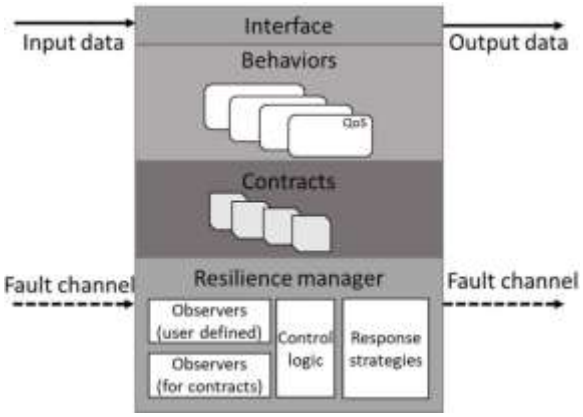

Fig. 2. Overview of the component model with the resilience manager. With respect to Fig. 1, software components of application layer, computational and communication platforms of platform layer are considered as components.

resilient methodology that allows for a more fine-grain distributed resilience management with cross-layer interactions and is aligned with the attributes of Industry 4.0.

B. Proposed Solution

In this letter, we propose contract-based methodology where components across layers of the cyber-infrastructure are associated with *contracts* and a light-weight resilience manager (see Fig. 1). This allows the system to detect faults (contract violation detected using timed or hybrid automata) and react (change contracts dynamically) effectively. When a component-level resilience manager is unable to provide any feasible solution to a fault, a layer-level resilience manager is informed. It has global-level understanding of the system and may find a feasible solution. We now discuss how the proposed methodology aligns with the attributes of Industry 4.0.

1) Self-Reconfigurability: This refers to the ability for the individual node (or a component) to detect disturbances and apply corrective and pre-emptive measures. This attribute aligns with the core contribution of the proposed methodology. We use contracts to clearly define the functionality of a component. Any disturbances that violate the contracts are monitored using observers that run concurrently. As a recovery strategy a component switches between contracts (changing its behavior) to contain faults, a step toward ensuring zero downtime production.

2) Self-Optimization: The availability of the resources in a cyber-infrastructure changes dynamically, detected using a DS [5]. The nodes are expected to self-optimize to improve performance. In order to maximize the performance, component manager tries to select the behavior that locally maximizes the use of available resources to yield the maximum quality of service (QoS). This may lead to a nonoptimal solution, a limitation of this letter.

3) Peer-Awareness: This refers to the ability to communicate with peers to collaboratively diagnose and respond to faults. In the proposed methodology, components communicate via fault communication channels to effectively deal with faults. For example, when a producer that is publishing sensor data needs to be restarted, the consumer is informed that the producer is inactive and knows when it will be reactivated.

I. Proposed Contract-Based Resiliency

In this section, we describe a new methodology for developing distributed resilient architectures, called contract-based resiliency. It is motivated by component-based engineering which has been successfully used for large-scale system designs and the use of contract-based designs for explicitly pinpointing the responsibilities and assumptions of a component [10]. In our approach, we describe an application using a network of components and use contracts to capture the behavior of the components. Finally, we detect failures (contract violation) and react (change contracts dynamically) to the disturbances, providing resiliency.

A. Component Architecture

A component is an open system that receives inputs from the environment, executes a behavior, and generates output to the environment. The environment could be the collection of other components or the physical world. An overview of a component is presented in Fig. 2.

1) *Interface:* It defines the input/output data channels of a component. Data is consumed through input interface, processed by the component, and output data is produced.
2) *Behaviors:* A component can be described using multiple behaviors. Each behavior is associated with a QoS. At runtime, the resilience manager selects the behavior of the component.
3) *Contracts:* A contract specifies assumptions on the behavior of the environment, and guarantees about the behavior of the component [10]. At runtime, the resilience manager can switch between contracts to react to the disturbances in the system.
4) *Resilience Manager:* Detects faults (using observers) and decides (control logic) the best course of action (response strategy). It also responds to the fault information from other components.

B. Example Application

Fig. 3 presents the running example, called conveyor belt block pick-up (CBBP). The application relies on the timing information from sensors S1 and S2 and then activate the robot at the right time to pick-up block from a moving conveyor belt.

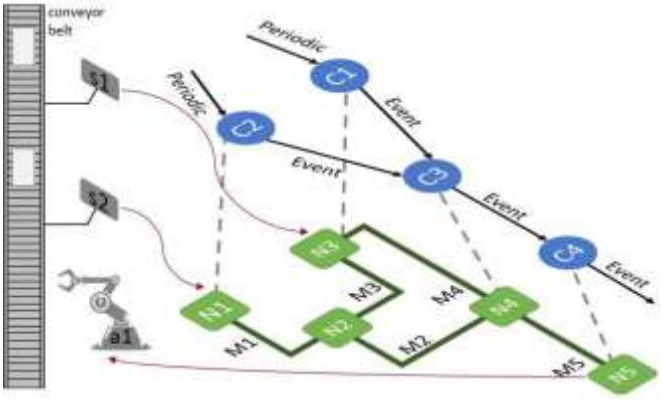

Fig. 3. Overview of the mapping between software components ($c_1, \ldots, c_4$) of the application layer and the computational components ($n_1, \ldots, n_5$) of the platform layer. Physical layer is not shown for brevity.

1) Application Graph: The CBBP application is described as a set of connected components in the form of a graph, called *application graph*. The nodes of the graph are components and the edges represent communication between components. See the graph in Fig. 3 with components c_1, c_2, c_3, and c_4. Component c_1 samples sensor ($S1$) data *periodically* (T_{samp}). It applies different signal processing techniques [described using different component behaviors (see Fig. 2)] to extract the time (t_1) when the center of a block is aligned with sensor $S1$. When the block is detected, the timing information (t_1) is sent to component c_3. Similarly, component c_2 periodically samples sensor $S2$ to compute and send the timing information (t_2) to component c_3. Based on the received timing information, c_3 computes the time (t_3) when the robot needs to be activated and then sends it to component c_4. Finally, the robot is activated to pick-up a block from the assembly line.

2) Mapping Between Application and Platform Layers: The platform layer comprises a set of computational platforms ($n_1, \ldots, n_5$) and connecting communication platforms ($m_1, \ldots, m_5$). The mapping of the components in an application graph to computational platforms is shown in Fig. 3. For example, component c_1 is mapped to computational platform n_3. Assuming that c_1 has four possible behaviors ($\text{BEH} \in \{\text{beh}_1^{c1}, \text{beh}_2^{c1}, \text{beh}_3^{c1}, \text{beh}_4^{c1}\}$), based on the mapping information we assume an execution time cost function $\text{EC} : \text{BEH} \rightarrow \mathbb{R}$ is given. Furthermore, we assume the communication between components is handled by a network infrastructure. We assume a communication time cost function $\text{CC} : \text{BEH} \rightarrow \mathbb{R}$ is given.

C. Generating Contracts From Timing Deadlines (Off-Line)

Using the speed of the conveyor belt we can compute the time taken for a block to travel from sensor (S1) to the actuator (robot), denoted as $\Delta_{s_1 a_1}$. We assume the decomposition of end-to-end timing constraints (e.g., 10 s from S1 to robot) into deadlines for each component is given. For the CBBP example, we assume the end-to-end latency is decomposed as described by (1), where all terms are constants

$$T_{s_1 \rightarrow c_1} + T_{c_1 \rightarrow c_3} + T_{c_3 \rightarrow c_4} + T_{c_4 \rightarrow a_1} \leq \Delta_{s_1 \rightarrow a_1}. \quad (1)$$

1) Contracts to Satisfy the Timing Constraint $\mathbf{T_{s_1 \rightarrow c_1}}$*:* Constant $T_{s_1 \rightarrow c_1}$ refers to the time taken for n_3 to read sensor s_1 and send the data to component c_1. Based on existing notation [10], the first contract on n_3, denoted by symbol $C_{n_3}^1$ is as follows:

$$C_{n_3}^1 : \begin{cases} \text{inputs:} & s_1\_\text{data} \in \mathbb{R} \\ \text{outputs:} & c_1\_\text{data} \in \mathbb{R} \\ \text{assumptions:} & \top \\ \text{guarantees:} & c_1\_\text{data} = s_1\_\text{data} \textbf{ within } T_{s_1 \rightarrow c_1}. \end{cases}$$

Furthermore, we know that the sensors must be sampled periodically every T_{samp}. This requirement can be treated as a separate contract on n_3, $C_{n_3}^{\text{sampling}}$. In this case, both contacts can be composed[1]

$$C_{n_3}^{1_r} : \begin{cases} \text{inputs:} & s_1\_\text{data} \in \mathbb{R} \\ \text{outputs:} & c_1\_\text{data} \in \mathbb{R} \\ \text{assumptions:} & \top \\ \text{guarantees:} & c_1\_\text{data} = s_1\_\text{data} \textbf{ within } T_{s_1 \rightarrow c_1} \\ & \textbf{every } T_{\text{samp}}. \end{cases}$$

2) Contracts to Satisfy the Timing Constraint $\mathbf{T_{c_1 \rightarrow c_3}}$ *:* Constant $T_{c_1 \rightarrow c_3}$ refers to the time taken for n_3 to send the output data of c_1 to n_4 over a network and n_4 to send the data to c_3.

Component c_1 has four behaviors with different execution times on n_3, as described earlier. For each of the behaviors, we generate contracts, e.g., for behavior beh_1^{c1} the contract $C_{n3}^{c_1 \text{beh}_1}$ assumes nothing (represented using the symbol $\top$) and guarantees execution of beh_1^{c1} **within** $\text{EC}(\text{beh}_1^{c1})$.

We do not assume the communication delay to be a constant. At runtime, communication interface on n_3 and the network controller engage to create a contract C_{n3}^{c1n4} that provides a time bound on the delivery of c_1's data to n_4.

Based on an expected time ($T_{n_4 \rightarrow c_3}$) for processing the received data on n_4 and sending it to c_3, a single contract $C_{n_4}^{c1c3}$ is created where it assumes nothing and guarantees the delivery within $T_{n_4 \rightarrow c_3}$.

Finally, to monitor the timing constraint $\mathbf{T_{c_1 \rightarrow c_3}}$, observer O^{n_4} executes on n_4 to monitor the timestamps on the data from c_1 to c_3. Component n_4 was chosen since c_3 is mapped to n_4. Note that if the observer was to execute on any other platform component such as n_5, then timing information from n_4 need to be sent to n_5. This unnecessarily increases the fault detection time of the architecture.

D. Runtime Reconfiguration

1) Application-Level Reconfiguration: The state of the application is described by an ordered set of contracts currently selected by the components to execute their behavior. For the running example, the ordered set of components is $[c_1, c_2, c_3, c_4]$. The initial state of the application can be $[C_{n_3}^{c_1 \text{beh}_1}, C_{n_1}^{c_2 \text{beh}_1}, C_{n_4}^{c_3 \text{beh}_2}, C_{n_5}^{c_4 \text{beh}_1}]$. Any *application-level reconfiguration* is a transition from one application state to another. This state transition occurs when a fault is detected (violation of a contract) and a new contract is adopted by a software component.

2) Platform-Level Reconfiguration: As described earlier, the platform layer consists of computational and communicational platforms. Once again, the state of the platform can be described as an ordered set of contracts, e.g., $[C_{n_3}^{c1n3}, C_{n_4}^{c1c3}, \ldots]$. Any *platform-level reconfiguration* is a

[1]Rules of composition is a part of the future work. They can be based on existing work [10].

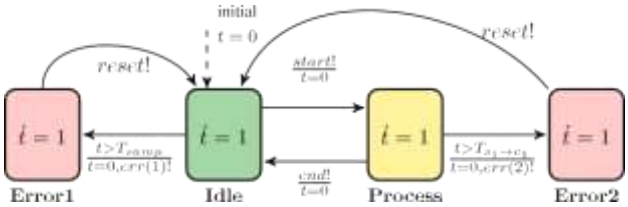

Fig. 4. Observer implemented using timed automata. It detects if the process deadline ($T_{s_1 \rightarrow c_1}$) or sampling rate (T_{samp}) are not satisfied.

transition from one platform state to another, e.g., due to high network traffic the contract C^{c1n3} needed to be rene- C_{n3} , representing a reconfiguration at platform level. Note, this may or may not lead to an application level reconfiguration.

II. Fault-Detection and Response

In the proposed resilience architecture, fault detection is tightly coupled with the contracts. A fault detection (violation of contracts monitored using observers) and response (change in contract by the resilience manger) will be managed efficiently. In the following, we discuss some of the faults experienced in distributed systems [11].

A. Hanging Processes

In this case, execution time of a process is longer than normal. For our example, component c_1 may take longer than expected. The time constraints are captured using the contract $C_{n_3}^{1^r}$ in Section II-C1. It states that the process must be completed within the duration of $T_{s_1 \rightarrow c_1}$ and the process must be executed every T_{samp}.

1) Fault Detection Technique: The contract $C_{n_3}^{1^r}$ is associated with an observer. The observer $O_{n3}^{1^r}$ is implemented using timed automaton [12] (see Fig. 4).

2) Response to the Fault: The observer failure is detected by the control logic of the resilience manager. As a response strategy, the component c_1 is restarted and the resilience manager of component c_3 is notified about the type of fault and the expected recovery time via the fault channel (see Fig. 2).

Based on the recovery time of a producer (e.g., c_1), the control logic of the consumer (e.g., c_3) may select different response strategies. For example, c_3 can reduce its execution time by selecting a different behavior. In this case, based on the recovery time beh_3^{c1} can be selected instead of beh_1^{c2}. Of course, this reduces the QoS. This response is achieved by switching the contracts at runtime from $C_{n3}^{c1\text{beh1}}$ to $C_{n3}^{c_1\text{beh}_2}$, resulting in an application-level reconfiguration.

B. Network Outages

In this case, communication link has failed. For our example, consider the connection between n_3 and n_4.

1) Fault Detection Technique: Using heartbeat signals component-level resilience managers of n_3 and n_4 periodically detect the status of the connection.

2) Response to the Fault: Observers fail when the heartbeat signals are not received by the resilience managers of n_3 and n_4. This triggers the control logic which can selects a response strategy such as switching from a wired to a wireless connection. Furthermore, if the component-level resilience managers were unable to re-establish the link, they would inform a layer-level resilience manager such as an SDN controller to find an alternative route.

IV. Conclusion

We have presented a contract-based methodology for enabling resilient cyber-infrastructure for Industry 4.0. Applications are described as a set of modular components that are distributed over a network. Contracts are used for describing the component's interaction with other components (and

itored using observers. We detect failures (contract violation) and react (change contracts dynamically) to the disturbances, for providing resiliency.

We intend to create a multidimensional resilience metric in the future to assess resilience in relation to other performance metrics, including recovery time, throughput, security, and safety. In addition, we want to enable parameterized contracts so that they can react to errors in various components automatically. For instance, the robot's motor speed can be decreased (update contract with regard to conveyor speed) in reaction to failing to successfully pick up blocks from the moving conveyor by the deadline. This concept aligns with our goal of supporting plug-and-produce, which calls for dynamic reconfiguration.

References

[1] L. Jay, B. Behrad, and H.-A. Kao, "A cyber-physical systems architecture for Industry 4.0-based manufacturing systems," *Manuf. Lett.*, vol. 3, pp. 18–23, Jan. 2015.

[2] W. Dai, V. N. Dubinin, J. H. Christensen, V. Vyatkin, and X. Guan, "Toward self-manageable and adaptive industrial cyber-physical systems with knowledge-driven autonomic service management," *IEEE Trans. Ind. Informat.*, vol. 13, no. 2, pp. 725–736, Apr. 2017.

[3] J. Sztipanovits *et al.*, "Toward a science of cyber—Physical system integration," *Proc. IEEE*, vol. 100, no. 1, pp. 29–44, Jan. 2012.

[4] L. Strigini, "Fault tolerance and resilience: Meanings, measures and assessment," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter *et al.*, Eds. Heidelberg, Germany: Springer, 2012, pp. 3–24.

[5] E. Scott, I. Madari, A. Dubey, and G. Karsai, "RIAPS: Resilient information architecture platform for decentralized smart systems," in *Proc. IEEE Int. Symp. Real Time Comput.*, Toronto, ON, Canada, May 2017, pp. 125–132.

[6] D. Ratasich, O. Höftberger, H. Isakovic, M. Shafique, and R. Grosu, "A self-healing framework for building resilient cyber-physical systems," in *Proc. IEEE Int. Symp. Real Time Distrib. Comput.*, Toronto, ON, Canada, 2017, pp. 133–140.

[7] M. G. Valls, I. R. Lopez, and L. F. Villar, "iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 228–236, Feb. 2013.

[8] W. Lepuschitz, A. Zoitl, M. Vallée, and M. Merdan, "Toward self-reconfiguration of manufacturing systems using automation agents," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 41, no. 1, pp. 52–69, Jan. 2011.

[9] G. Denker *et al.*, "Resilient dependable cyber-physical systems: A middleware perspective," *J. Internet Services Appl.*, vol. 3, no. 1, pp. 41–49, May 2012.

[10] A. Benveniste *et al.*, "Contracts for systems design: Theory," Inria Rennes Bretagne Atlantique, Rennes, France, Rep. RR-8759, 2015.

[11] G. Kola, T. Kosar, and M. Livny, "Faults in large distributed systems and what we can do about them," in *Proc. Int. Euro Par Conf. Parallel Process.*, Lisbon, Portugal, 2005, pp. 442–453.

[12] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994.