Achieving resilience for cyber-physical systems with 4DIAC IEC 61499 through parametric contracts

Ng, Daniel Jun Xian

2020

Ng, D. J. X. (2020). Achieving resilience for cyber-physical systems with 4DIAC IEC 61499 through parametric contracts. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/137595>

<https://doi.org/10.32657/10356/137595>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 23:14:52 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

**ACHIEVING RESILIENCE FOR CYBER-PHYSICAL
SYSTEMS WITH 4DIAC IEC 61499 THROUGH
PARAMETRIC CONTRACTS**

NG JUN XIAN DANIEL

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarized materials, and has not been submitted for a higher degree to any other University or Institution.

23/8/2019

.....

Date

.....

NG JUN XIAN DANIEL

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

23/8/2019

.....

Date

.....

A/P Arvind Easwaran

Authorship Attribution Statement

This thesis contains material from 2 papers published in the following peer-reviewed journal and from papers accepted at conferences in which I am listed as an author.

Chapter 4 and part of Chapter 5 are published as M.S. Haque, D.J.X. Ng, A. Easwaran, and K. Thangamariappan, “Contract-based Hierarchical Resilience Management for Cyber-physical Systems”, in Computer, vol. 51, no. 11, pp. 56-65, Nov. 2018. DOI: 10.1109/MC.2018.2876071.

The contributions of the co-authors are as follows:

- A/Prof Arvind provided the initial project direction and edited the manuscript drafts.
- Dr. Mohammad Shihabul Haque and I prepared the manuscript drafts. The manuscript was revised by Karthikeyan Thangamariappan.
- I co-designed the hierarchical resilience framework with Dr Mohammad Shihabul Haque and performed all the experimental work at the Delta-NTU Corporate Laboratory for Cyber Physical Systems, School of Electronic and Electrical Engineering.
- All experiments and the implementation of the case study were conducted by me.

Part of Chapter 5 is published as D.J.X. Ng, A. Easwaran, and S. Andalam, “Contract-based Hierarchical Resilience Framework for Cyber-Physical Systems: Demo Abstract”, in Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS '19), pp. 324-325. DOI: 10.1145/3302509.3313323.

The contributions of the co-authors are as follows:

- I wrote the drafts of the manuscript. The manuscript was revised together with A/Prof Arvind and Dr. Sidharta Andalam.
- I designed and implemented the demonstrator at the Delta-NTU Corporate Laboratory for Cyber Physical Systems, School of Electronic and Electrical Engineering.

23/8/2019

.....

Date

.....

NG JUN XIAN DANIEL

Acknowledgements

I wish to express my most enormous gratitude to my supervisor, Associate Professor Arvind Easwaran, for his patience, support, and guidance during my graduate study. I would also like to thank my family for nurturing and supporting me with my university education. Last but not least, to my wife, Christina, whose unwavering support and love encouraged me to complete this thesis.

Ng Jun Xian Daniel, 23rd August 2019

To my dear family

Abstract

Industry 4.0 has garnered much interest in traditional manufacturing setups to play catch up with the state-of-the-art. This fourth industrial revolution [1] has caused a proliferation of computing devices and sensors onto the factory floor. This proliferation has also caused a paradigm shift in the designing of the plant supervisory management control systems such as Supervisory Control and Data Acquisition, which traditionally controls the automation systems for manufacturing plants and manages the fault recovery mechanisms. With this said, the fourth industrial revolution requires a new framework to improve resiliency in these systems to account for a large number of interconnected devices in a Cyber-Physical System (CPS).

Software-based resilience solutions can provide the necessary flexibility in dealing with failures to reduce downtime and the need for human intervention. We present a contract-based resilience framework for CPS that incorporates Assume-Guarantee contracts to define the user requirements of the CPS. These contracts describe the non-functional requirements which the system is expected to meet and provides a threshold for triggering an alarm (i.e., a fault occurrence). The top-level contract (i.e., root contract) represents the overall requirement of the system, and this necessitates decomposition, which is the process of decomposing the root contract into smaller sub-contracts. The decomposed sub-contracts represent the requirements asked of the different interconnected components in the system. The framework also has observers which serve to check for violations of the sub-contracts and Resilience Managers (RMs) who manage the set of sub-contracts. Together, RMs and observers form a logical hierarchy for decentralized fault monitoring of the entire CPS. A Fischertechnik Sorting Line with Color Detection training model, which represents a factory's assembly line, as well as an industrial Festo Didactic Cyber-Physical Factory, are used to demonstrate the capabilities of the resilience framework. Both the control logic and resilience framework of the assembly line use an open-source platform, 4DIAC, which is a Programmable Logic Controller framework for distributed industrial control based on the International Electrotechnical Commission 61499 standard.

The process described above would require a great deal of manual work if it were to be done for a large-scale CPS. As part of our contribution, we present an automated way of generating the contract hierarchy and deploying it on 4DIAC. This process starts from defining the user requirements, which is in the form of a root contract, and the hardware information of the CPS in an AutomationML (AML) file. Then, the information from the AML file is used to decompose the root contract into a hierarchy of sub-contracts. The entire process completes when we port the decomposed contracts onto the 4DIAC platform by generating the function blocks for resilience management (i.e., RM and observer blocks). The user can then download the function blocks onto its associated hardware for deployment.

Finally, we demonstrate the framework on an industrial testbed to showcase the framework with better interoperability. This master's report presents the translation of a resilience framework into reality.

Contents

Acknowledgements	ix
Abstract	xiii
List of Figures	xvii
List of Tables	xix
List of Abbreviations	xxi
1 Introduction	1
1.1 Manufacturing Systems and Industry 4.0	2
1.1.1 Manufacturing Today	2
1.1.2 Industry 4.0	3
1.2 Problem Statement and Objectives	5
1.3 Outline of the Report	7
2 Background	9
2.1 OPC-Unified Architecture (OPC-UA)	9
2.2 International Electrotechnical Commission (IEC) 61499 and 4DIAC	10
2.3 AutomationML	11
3 Literature Review	13
4 Hierarchical Contract-based Resilience Framework (HCRF)	17
4.1 Overview	17
4.2 Framework Details	18
4.2.1 Hierarchy and Resilience Managers	18
4.2.2 Contracts	19
4.2.3 Observers	21
5 Development and Implementation of the HCRF on a Fischertechnik Model	23
5.1 Model Factory	23

5.2	Resilience Framework	26
5.3	Implementation	28
5.4	Evaluation	31
5.4.1	Fault Scenarios	31
5.4.2	Performance Comparison	33
5.4.3	Advantages and Limitations of the Framework	35
5.5	Experience on Development with IEC 61499	36
6	Automated Toolchain	39
6.1	AutomationML	39
6.1.1	Describing Hardware Capabilities	39
6.1.1.1	Sensors / Actuators	40
6.1.1.2	Computation	40
6.1.1.3	Inter-connections	42
6.1.2	User Requirements	42
6.2	Python Program	43
6.2.1	Decomposition	43
6.2.2	4DIAC Function Blocks	45
7	Industrial Testbed	47
7.1	Festo Didactic Cyber-Physical (C-P) Factory	47
7.1.1	Control	48
7.1.2	Resilience Framework	49
7.1.2.1	Drilling Station	49
7.1.2.2	Camera Station	49
7.1.2.3	ASRS Station	50
8	Conclusion and Future Work	51

List of Figures

1.1	Evolution of industrial manufacturing. Source IoT analytics [2]. . .	2
1.2	Cyber-Physical System based automation. Source IoT analytics [2].	4
4.1	Hierarchical Contract-based Resilience Framework.	18
4.2	Composition of contracts: Contract 1.1 and 1.2 are composed together to form Contract 1.	20
5.1	Fischertechnik Training Model: Sorting line with color detection (EAN-CODE 4048962250404).	24
5.2	Operation flow of the interconnected components in the model factory.	25
5.3	Resilience hierarchy of the components and contracts in the model factory.	26
5.4	Raspberry Pis containing the control and resilience management logic of the various components, as well as the Arduino microcontroller and various electronics.	29
5.5	4DIAC Integrated Development Environment	30
5.6	Function Block Interface: The event and data connections of the color processor application.	30
5.7	Execution Control Chart of a basic function block.	31
5.8	Composite function block network of the color processor application.	32
5.9	Hypothetical designs of a fully centralized and fully decentralized resilience framework.	33
5.10	The number of inter-component communication messages required for the different framework designs.	34
5.11	The time spent on fault recovery for the different framework designs.	34
6.1	AutomationML: Sensor and actuator information embedded within its own internal element (i.e., an object).	40
6.2	AutomationML: Computation resources and the applications are structured and described as such.	41
6.3	AutomationML: InternalLinks illustrated as blue dotted lines connecting the Raspberry Pis to the network switch.	42
6.4	AutomationML: Root contract of the model factory showing its end-to-end requirement.	43
7.1	The Festo Didactic Cyber-Physical Factory.	47

7.2	System configuration of the demonstration on the Festo Didactic Cyber-Physical Factory.	50
-----	--	----

List of Tables

5.1 Input / Output variables for the system. 25

List of Abbreviations

AFTCS active fault-tolerant control system.

AML AutomationML.

ASRS Automated Storage / Retrieval System.

BS bin selector.

C-P Cyber-Physical.

CAEX Computer Aided Engineering Exchange.

CM Control Manager.

COLLADA COLLABorative Design Activity.

CP color processor.

CPS Cyber-Physical System.

DDS Data Distribution Service.

DHT Distributed Hash Table.

DM Discovery Manager.

EC ejector controller.

ERP Enterprise Resource Planning.

FBDK Function Block Development Kit.

FDID fault detection and identification.

FDIS fault detection and isolation.

FTCS fault-tolerant control system.

GPIO General Purpose Input-Output.

HCRF Hierarchical Contract-based Resilience Framework.

HMI Human-Machine Interface.

I/O Input-Output.

IDE Integrated Development Environment.

IEC International Electrotechnical Commission.

IPC Industrial PC.

MC motor controller.

MES Manufacturing Execution System.

MES4 Manufacturing Execution System 4.

NFP non-functional property.

OPC-UA Open Platform Communications Unified Architecture.

OS operating system.

PC pulse counter.

PCB printed circuit board.

PLC Programmable Logic Controller.

RFID Radio-frequency Identification.

RM Resilience Manager.

ROS Robot Operating System.

RPI Raspberry Pi.

SCADA Supervisory Control and Data Acquisition.

SDN Software-Defined Networking.

XML eXtensible Markup Language.

Chapter 1

Introduction

Industry 4.0 [1] has garnered much interest worldwide to create smart factories. Smart factories incorporate complex, large-scale deployment of computational devices and sensors for decentralized decision-making on the factory floors. This decentralized approach may also be performed on older factories to tolerate a mix of traditional manufacturing and industrial practices with newer technologies. Such systems are referred to as Cyber-Physical System (CPS) where cyber components integrate computation with physical processes [3, 4]. A key promise in Industry 4.0 is to reduce factory downtime by having intelligence in the systems to dynamically detect and recover from faults. As more data get generated from the increase in available resources (i.e., sensors and processes), there is better transparency for making the appropriate runtime and fault recovery decisions to keep the assembly line active. This allows for more efficient and productive factories but it also carries risk of higher breakdowns when the systems becomes more complex. This is why, a resilient infrastructure is crucial in achieving Industry 4.0, so that systems can dynamically restore themselves to provide continuity. This continuity shall persist, even when facing changes (e.g., unforeseen faults) [5].

The increased connectivity between computational devices and sensors also presents a challenge for networking and monitoring. It becomes crucial that a robust networking infrastructure is in place for system status monitoring and to ensure the availability and timely arrival of priority packets. As CPS become increasingly involved with distributed infrastructure, it becomes harder for the software engineer to develop and maintain large amounts of application as well as fault handling

code. It is also challenging to ensure that crucial components in the systems adhere to their functional and non-functional requirements.

While manufacturers are keen to adopt newer technologies to enable increased productivity, many are hesitant as they lack courage or capital or necessary knowledge required to coordinate an upgrade from existing systems [6, 7]. It would be more economical if they could incorporate some newer technologies while retaining the capabilities of the older systems they have.

1.1 Manufacturing Systems and Industry 4.0

1.1.1 Manufacturing Today

Current manufacturing practices today stem from the innovations that started after the Second World War (i.e., Industry 3.0). Industry 3.0 was the era in which the advent of information technology (IT) and automation in industrial manufacturing started to replace human labor. Programmable Logic Controllers (PLCs) and industrial robots became more prevalent over the years, which increased the productivity of factory floors [2]. This evolution of IT and industrial automation is shown in Figure 1.1. Better communication technologies within the factory floor such as Industrial Ethernet and across countries through optical fiber technologies led to the current 5-layer architecture commonly used in manufacturing.

FIGURE 1.1: Evolution of industrial manufacturing. Source IoT analytics [2].

The 5-layer architecture shown in Figure 1.1 has the Enterprise Resource Planning (ERP) system right at the top. ERP is a business management tool which integrates a multitude of applications such as inventory and order management,

accounting, and human resources. Information from the ERP, such as production planning and order requirements, are then streamlined to the Manufacturing Execution System (MES). The MES software system manages and monitors sophisticated manufacturing equipment and stores real-time data on the complete production lifecycle of the product. Some operations which the MES does are operation sequencing, resource allocation and status, performance analysis, and maintenance management [8]. This provides manufacturers with real-time workflow visibility and acts as an intermediary between the ERP and process control systems. Supervisory Control and Data Acquisition (SCADA) systems, on the other hand, are industrial process control systems which are a combination of both hardware and software elements [9]. They control industrial processes locally or remotely; monitor, collect, and process real-time data; directly communicate with low-level devices such as sensors, actuators, and Human-Machine Interface (HMI). As part of the SCADA architecture, PLCs serve as computing nodes which are traditionally programmed in ladder logic. These PLCs are directly connected to field devices (i.e., sensors and actuators) through its Input-Output (I/O) signals, and this completes the 5-layer architecture. The SCADA software obtains and processes data from PLCs, and displays them through the HMI to help operators analyze the data and make critical decisions (e.g., rectifying a high error incidence rate on a production line).

1.1.2 Industry 4.0

As discussed earlier, technologies such as Ethernet connectivity, sensors, software solutions provided by MES and SCADA, have all been utilized in the manufacturing industry for years. Cloud-based solutions, such as the ERP systems, have also been used at the enterprise level. So what makes the incoming Fourth Industrial Revolution, referred to as Industry 4.0, different from its predecessor? We can identify four main design principles of Industry 4.0; namely i) Interconnection, ii) Information transparency, iii) Decentralized decisions, and lastly iv) Technical assistance [10].

Interconnection arises from the increased connectivity between machines, devices, sensors, and people through the internet [11]. This pervasiveness of computing and networking are enabled by smaller and cheap hardware, and improved wireless

technologies, respectively. However, existing manufacturing systems from different vendors often run proprietary communication protocols which make intercommunication between them difficult and/or costly. Hence, an emerging standard known as the Open Platform Communications Unified Architecture (OPC-UA), was chosen for driving the Industry 4.0 initiative for open connectivity, interoperability, security, and reliability [12].

The sharing of information becomes ubiquitous with more interconnected devices and people, resulting in information transparency. This allows for a digital twin of the physical factory by linking sensor data (as close to the I/O layer) with digital plant models. The collection of data also supports developments in complex algorithms to enable applications such as machine learning, improved predictive maintenance, reconfigurability, and more.

As more embedded computers get onto the factory floor, decentralized decisions with the availability of data allow for better decision-making and increase overall productivity [10]. Human operators will no longer be bothered with trivial decision-making, and their role in factories will change to complement this. When a machine-unsolvable problem occurs, the HMI interface needs to aggregate and visualize information comprehensively for the human operators to make informed decisions quickly and on short notice [13].

FIGURE 1.2: Cyber-Physical System based automation. Source IoT analytics [2].

In short, Industry 4.0 is envisioned to create smart factories consisting of flexible, reconfigurable CPSs where the 5-layers may no longer exist as distinct layers, see Figure 1.2. Boundaries between individual factories will also cease to exist, with communication going both ways instead of just being streamlined downwards. A

standard, such as the OPC-UA, will enable enterprise systems with customer orders to directly interface with the production line to create small batches with a just-in-time inventory. More importantly, machines will gradually be able to manage themselves and the production process, reducing human resources.

1.2 Problem Statement and Objectives

Cyber-infrastructure disruptions can have severe and costly consequences. Therefore, there is a need for a scalable and resilient CPS infrastructure. Traditional hardware-based redundancy techniques are expensive and would not scale well in a large-scale CPS. Conversely, software-based techniques are cheaper and flexible, and the hardware infrastructure required can be adapted easily to provide for this.

Resilient infrastructure should be able to detect faults efficiently and be able to recovery from these faults automatically by dynamically reconfiguring itself. Ideally, a recovered system should allow for minimal disruption to normal operations. Even if a full system recovery is not possible, recovery mechanisms should be resilient enough to initiate partial functionality such that there is continuity in the system until engineers can be called in to rectify and restore full system functionality.

Therefore, it is crucial for a software resilience framework to have the following attributes to align with the envisioned Industry 4.0:

- **Light-weight:** The addition of the resilience framework should be simple and easy to implement for large-scale CPS.
- **Dynamicity:** The ability to respond to changing requirements or faults during runtime, and apply corrective and preemptive measures.
- **Fault detection:** A myriad of fault detection techniques can be employed to detect faults such as heartbeats, time-stamping, finite state machines, and hybrid automata.
- **Scalability:** As future factories become more extensive, the framework must be easy to scale to account for the numerous devices, machine, and people being interconnected.

- **Availability:** Resilience management needs to be available 24/7 to keep the production lines running.
- **Code separation:** Resilience management codes and application codes need to be separated to keep the CPS easy to develop and maintain.

Keeping the above attributes in mind, the Hierarchical Contract-based Resilience Framework (HCRF) [14] was proposed in our earlier work. It uses a formal parametric contract-based methodology to detect faults dynamically. The framework also encompasses a hierarchical approach to manage the numerous components foreseeable in a large-scale CPS efficiently. Observers report contract failures (i.e., the occurrence of a fault) to Resilience Manager (RM) who supervise the fault recovery mechanism. RM are organized in a hierarchy to enable faster fault detection and recovery within their sub-hierarchy groupings, and to make them more manageable. Generating this hierarchy would require decomposition techniques on the high-level contracts which represent overall user requirements on the system. An RM's recovery reaction to faults depends on the mix and magnitude of contract violations surfacing.

Following the design of the HCRF, we came up with a automated software toolchain in order to provide an automated methodology for deploying the HCRF onto real world systems. Information on the hardware components and user requirements of the system are first captured onto a human-readable format and stored as an AutomationML (AML) data file. This AML file gets parsed into our software which then decomposes the contracts defined, based on the user requirements and the hardware component information. Next, the resulting sub-contracts formed after decomposition gets ported onto the International Electrotechnical Commission (IEC) 61499 4DIAC platform [15].

Others have tried to incorporate resiliency into the design of the system. For example, the simplest way is to have redundancy built into the system but this comes at a high cost and is not scalable given the large scale of CPS. Another method is having active fault-tolerant systems but this would require domain expertise and needs to be customized for individual systems. A third method is through software-based approaches, in which we try to address their shortcomings with an implementation of our own.

This report focuses on the design and implementation of the HCRF, which provides resiliency, code separation between application and fault recovery, and is easily scalable to keep the production floor moving. We also developed a software toolchain to aid in deploying the HCRF. An implementation of the framework was demonstrated on both a Fischertechnik model factory testbed [16] and the Festo Didactic Cyber-Physical (C-P) Factory [17].

1.3 Outline of the Report

Chapter 2 presents the necessary background related to the terminologies and concepts used in the report.

Chapter 3 reviews existing literature on fault-tolerant control systems (FTCSs) and resilience frameworks.

Chapter 4 presents the key features, concepts, and details of the HCRF.

Chapter 5 shows how the HCRF can be developed and implemented onto a model factory testbed.

Chapter 6 presents details of the developed software toolchain that aids in the deployment of the HCRF.

Chapter 7 describes the implementation of our resilience framework on the Festo C-P Factory.

Chapter 8 proposes ideas for future work and concludes this report.

Chapter 2

Background

In this chapter, concepts, terminologies and technologies related to the development of our proposed HCRF are discussed.

2.1 OPC-Unified Architecture (OPC-UA)

OPC-UA is a global communication standard [12, 18] that can fulfill the complex requirements of Industry 4.0. Firstly, there is a need for "Machine-to-Machine" communication, which defines the communication between two machines or the data transfer between a more or less intelligent device and a central computer. Secondly, there is a need for remote device access, as machines and field devices no longer just send basic sensor information. They are able to process and combine data from other surrounding devices, creating extra value for users. With machines being networked to form "smart" objects that are assembled into "smart factories", this networking set-up creates an internet of things infrastructure which needs to be able to communicate with one another seamlessly. Thus a global communication standard that fulfills these requirements would be ideal for Industry 4.0.

At the very core of OPC-UA, the OPC foundation is a vendor-independent non-profit organization. There is no requirement of being a member to be able to utilize OPC-UA technology or for developing OPC-UA products. Also, OPC-UA runs on all operating systems (OSs) and even runs on embedded systems without an OS.

These features make it easy for all parties to adopt the OPC-UA standard. OPC-UA is also highly scalable. It scales from a 15kB footprint to single- and multi-core hardware systems which run on various CPU architectures such as Intel, ARM, and PowerPC. It has also been successfully implemented on embedded field devices such as Radio-frequency Identification (RFID) readers, SCADA/HMI products as well as MES/ERP systems. Users are also able to secure their communication channels through user and application authentication, signing of messages and the encryption of the transmitted data itself.

Lastly, OPC-UA has been certified as an IEC standard (IEC 62541), with tools and test laboratories providing for testing and the certification of conformity.

2.2 International Electrotechnical Commission (IEC) 61499 and 4DIAC

In order to meet the needs of the computational sophistication of industrial automation for Industry 4.0, a new software design is required. Traditionally, control systems were designed based on PLCs. HMIs are also provided by a wide variety of different panels, lights and switches. Advanced HMIs also provide color displays as well as touch sensitive screens for operator interactions. Typically, a large PLC system will have a number of PLCs communicating via proprietary high-speed networks. The PLCs will be connected to a large number of I/O signals for handling sensors and actuators. These systems tend to be developed by writing large monolithic software packages, which will be hard to reuse for new applications and difficult to integrate with one another. Data and functionality of an application cannot be shared with another even when using similar machines. This creates significant system development time when the designer is concerned with mapping signals between devices and providing the drivers required to allow different types of instruments and controllers to communicate. Following that, some vendors have started implementing PLC logic on PC hardware such as SoftPLC, creating another class of devices termed Industrial PC (IPC) which are widely adopted today. However, the problems with creating individualized software for the system of PLC or IPC remain.

Several ways of programming PLCs exist under the IEC 61131 standard. There are three graphical and two textual programming languages defined under this standard, namely Ladder diagrams, Function Block diagram, Sequential Function Chart, Structure Text and Instruction List. In order to achieve high levels of integration from top level systems such as the MES to field level devices, and yet enable flexible systems that can be re-engineered rapidly, the IEC 61499 standard was developed [19]. IEC 61499 defines a domain specific modeling language for developing distributed industrial control solutions. The standard builds upon the function block concepts defined in IEC 61131-3 and defines how function blocks can be used in distributed industrial process, measurement and control systems. Function blocks have been used as an established concept for robust, reusable software components. It can provide a software solution to a small problem, such as valve control, or control a huge portion of a plant such as an entire production plant. Algorithms are allowed to be encapsulated in function blocks which can be understood by those who are not technically inclined. Each function block has a set of defined inputs, which are read by the internal algorithm when it runs. The algorithm's outputs are then written to the function block's outputs. Consequently, applications can be built by networks of function blocks formed by the interconnection between the function blocks' inputs and outputs. Apart from the function blocks, the standard also defines the system model which defines available control devices and the communication relationships among them, forming a network of communicating devices. Communication links can also be of different types and may be connected to different communication segments.

Eclipse 4DIAC [15] is one available open source infrastructure for distributed industrial process measurement and control systems based on the IEC 61499 standard. It includes an Integrated Development Environment (IDE), FORTE - a runtime environment, a function block library based on HOLOBLOC libraries [20], and example projects which have been implemented on 4DIAC. The software implementation of the HCRF, and the testbeds' applications were designed on 4DIAC.

2.3 AutomationML

Reference Architectural Model Industrie 4.0 [12] provides a reference document for Industry 4.0 so that all stakeholders share a common perspective and develop

a common understanding for its most important aspects. Apart from OPC-UA, AML [21] was also referred to in the document as part of an approach towards achieving end-to-end engineering. AML started in 2006 as an initiative from nine companies and research institutes to reduce engineering efforts [21]; namely Daimler, ABB, KUKA, Rockwell Automation, Siemens, NetAllied, Zühlke, and the universities of Karlsruhe and Magdeburg. Today, many engineers still struggle with a heterogeneous tool landscape and engineering data are stored in proprietary formats which could only be opened by a select number of tools. With that in mind, the consortium started the development and standardization of AML as an open, neutral, eXtensible Markup Language (XML)-based, and free engineering data format. This means that the AML file can be exported and imported by engineering tools correctly and without the risk of data loss while doing so. Some possible plant engineering specific data that can be stored within the AML format are: plant structure, geometry and kinematics, logic descriptions, relations between objects, and network-related data. To achieve this, AML leverages on existing data formats by adapting and combining them to standardize under the IEC 62714 standard. For example, Computer Aided Engineering Exchange (CAEX) allows for defining the hierarchical structure of a plant or a series of components [22]; the COLLABorative Design Activity (COLLADA) format provides for the geometry and kinematic descriptions; and the PLCopen XML format describes all of the logic definitions [23]. Therefore, as industry players are adopting this standard as part of their engineering tools and workflow, we do not want to reinvent the wheel, and would like to incorporate this into our toolchain.

Chapter 3

Literature Review

A fault happens when a component in the CPS malfunctions. There can be several types of faults. When actual and sensed measurements in the CPS differ, it is a sensor fault [24]. Similarly, when the intended input to the actuator differs from the actual output, an actuator fault occurs [24]. Cyber faults are faults that occur within the cyber layer, which could be unexpected execution cycles, missing communication packets, etc. Since we are dealing with a CPS, cyber faults that occur in the computing devices would affect the physical process as well, and may lead to catastrophic results.

The most traditional and intuitive way to achieve fault resiliency is to have redundancy in the system [25]. However, redundancy comes at a high cost; involving extra spatial, computational, and energy strain, and only handle faults of components for which a replica exists. Given that CPSs are to scale massively, having hardware redundancy would introduce further communication and synchronization overheads. Therefore, redundancy alone would not be sufficient nor feasible for Industry 4.0. There have also been works on fault-tolerant control systems (FTCSs) which are able to tolerate component malfunctions while maintaining desirable stability and performance attributes [26]. FTCSs also comprise of fault detection and isolation (FDIS) or fault detection and identification (FDID) systems. Fault identification is important as it is the first step in maintaining the desired performance. One such example is the active fault-tolerant control system (AFTCS) on nonlinear chemical process systems [27] that focuses on Lyapunov stability. However, using AFTCSs are very component specific and require in-depth knowledge

to apply them for each field of application (e.g., aircraft, automotive or nuclear power plants [26]). Thus, such methods cannot be easily applied and used in general for the manufacturing domain. Moreover, some of the FDIS tools developed were focused on being just a diagnostic or monitoring tool, rather than being part of the FTCS [26]. Since this would not provide autonomy for fault recovery, it does not align with the objectives of Industry 4.0.

As opposed to redundancy and fault tolerant control systems, several software-based resilience approaches have been proposed. In [28], their software-based ontology approach focuses on data availability and the continuity of this data. A centralized runtime manager detects a failure of the data publishing node p through heartbeats. Then, based on ontology, a new node p' which can provide the same information is identified as an alternative and is dynamically created to provide this service. However, with its core resilience functionality centered on one centralized runtime manager, it runs the risk of a single-point failure. Their current implementation which runs on the Robot Operating System (ROS), provides the functionalities needed to accomplish their approach. However, the creation of new ROS nodes to provide for the missing data currently incurs significant start-up time, and is not yet suited for real-time applications. The authors also assume that the ontology (a crucial part of their methodology for their method to work) is already available.

In RIAPS [29], a distributed, resilient CPS framework was proposed. Similarly to [28], it focuses on the resilience of information publishers and subscribers. Their framework consists of a resilient Discovery Manager (DM) service, which allows the applications in the system to discover each other and work collaboratively. While the previous approach relied on a centralized manager, the DM runs on OpenDHT, a Distributed Hash Table (DHT) implementation. This implementation, however, does not provide full data replication on all nodes but provides some redundancy. The DM checks and detects the failure of publisher-subscriber application services through periodic heartbeat signals and timestamps. It maintains a list of live services and de-registers them when a failure occurs. Application services are also required to re-register themselves should they come back online. While this is a distributed approach, registration and de-registration of the application services are time-consuming, and the exact cause of these lost services remain unknown. However, we should note that RIAPS not just provides for resilience functionalities

but comprises other components working together to build a decentralized software platform.

iLand [30] presents an approach for building a real-time reconfigurable service-oriented distributed system. Applications in this system are described as a graph where each vertex is a service (self-contained functionality) provided by the system's component in a distributed manner. Applications are built by connecting services in the form of a graph and the edges represent messages exchanged among them. Based on faults that occur during runtime, the Control Manager (CM) would select an alternative service. This knowledge needs to be brought in during the initialization phase, to make sure that all timing properties are satisfied. The CM also stores a default configuration as a backup to keep basic functionality of the system running. Once again, the CM that decides on the reconfiguration of application services is unaware of the reasons that cause the fault in the first place. Moreover, to compartmentalize information within the system, every time the CM performs a reconfiguration, it has to consult other managers to obtain information about services, service implementations, and the application itself before reconfiguration.

Increasingly complex functional and safety requirements of CPSs contribute to complicated and hard to understand control applications. Typical manufacturing applications have 17% of control code for normal operations, and the remaining 83% accounts for fault handling code on average [31, 32]. With more components, there are large amounts of code which can be difficult to understand while maintaining the original codebase. This problem is exacerbated when application codes are directly linked with fault handling codes [32]. Thus, there is a need for an approach which decouples fault handling techniques from application code.

Therefore, a resilience management framework which is scalable, quick to detect and fast to recover from faults, and separates application code from fault handling code is immensely beneficial.

Chapter 4

Hierarchical Contract-based Resilience Framework (HCRF)

4.1 Overview

In order to overcome the challenges a future smart factory would face, we propose our HCRF [14]. The HCRF is a light-weight resilience management framework which manages system components in the CPS. Components within the CPS can be sensors, actuators, controllers, and communication hardware. RMs are associated with components to manage the recovery response in the event of a fault, while observers are used to monitor for faults. Assume-guarantee contracts [33] are used to capture the guarantees provided by system components (i.e., requirements) which are monitored by observers during runtime. Deviations from these guarantees (i.e., contract failure) trigger a fault by the observers, and this is reported to the RM associated to it. RMs manage a set of contracts and decide on the recovery response. The RMs and contracts are also structured in a hierarchy to allow for scalability and to reduce communication overheads among the RMs. Depending on the combination and extent of contract violations, an RM may either respond by changing contract parameters (i.e., modify and hence potentially degrade component performance) or propagating the fault to a higher level RM as a response. We can decompose contracts into sub-contracts which allow for independent lower-level decision-making by the RMs, thus creating a hierarchy of resilience management.

This hierarchy also enforces a strict coordination protocol among the RMs when recovery solutions cannot be found at lower levels.

4.2 Framework Details

4.2.1 Hierarchy and Resilience Managers

FIGURE 4.1: Hierarchical Contract-based Resilience Framework.

Figure 4.1 shows how the RMs, components, observers, and contracts are structured together along with their interactions. A component can have a local RM and a contract tied to it. We use parametric contracts to enable efficient runtime updates to the hierarchy so that system degradation can be a possible recovery solution (e.g, reducing the speed of a conveyor when machines are failing). An observer is used to check for contract violations, enabling quick fault detection. It is also possible that an RM is not associated with any component and manages a series of lower-level RMs when contracts on the lower levels could affect each other. This assignment of duties among managers creates a hierarchy which allows decomposition of the resilience management functions, aiding with scalability. This also allows for local fault recovery for scenarios that can be handled locally, reducing the need to propagate the problem upwards.

In our framework, resilience management is the collective duty of a group of RMs. Managers are assigned with contracts which are used by the observers to monitor the system for any faults. It makes decisions based on information from its contracts as well as from other managers. We enable efficient communication between the RMs by only having them to communicate fault information when a fault occurs, and provide parametric updates for any changes required. Due to this design, a virtual hierarchy of RMs and their contracts are established. The RM determines if there are any local recovery solutions available under its discretion. If there is a local solution, the observer is informed of the parameter update to prevent recurring fault reporting. However, when no solution exists, the RM propagates fault information to the higher-level RM, see Figure 4.1. The higher-level RM uses the information it has, which may also come from other lower-level RMs, to perform the fault recovery analysis. This chain of interactions can be inferred from Figure 4.1.

4.2.2 Contracts

A contract consists of the following:

- **Inputs:** Input variables to the component.
- **Outputs:** Output variables of the component.
- **Parameters:** Variables which allow parameterized specifications [34] on the assumptions and guarantees.
- **Assumptions:** Assumptions on the inputs and on the environment in which the component operates in.
- **Guarantees:** Guarantees on the outputs that the component is expected to fulfill.

Contract parameters are inferred from adjustable variables in the component's capabilities. For example, a standard piece of equipment in manufacturing plants is the conveyor belt used for transporting unfinished and finished goods. The plant's throughput can be modified through the speed of such conveyor belts and thus be used as a contract parameter. Functions based on these parameters are

used in assumptions and guarantees of contracts. Although contract assumptions and guarantees could be defined using any desired logic, we restrict our focus to Boolean logic for implementing efficient observers.

FIGURE 4.2: Composition of contracts: Contract 1.1 and 1.2 are composed together to form Contract 1.

When contracts are composed together, care must be taken to ensure that the resulting hierarchy satisfies desirable properties for contract composition and refinement (defined in [33] and reproduced below). For example, Figure 4.2 shows three contracts (i.e., 1, 1.1 and 1.2) where contract 1 is the composition of contract 1.1 and contract 1.2. Components 1 and 2 each operate on the inputs A and B, respectively, and the resulting output C is to be guaranteed to be a positive number. Contracts 1.1 and 1.2 individually enforce this, which can also be enforced similarly by contract 1. In particular, the composition of a set of contracts belonging to some lower-level components needs to be a refinement of the contract in the higher-level parent component. It is also essential to ensure that the root level contract satisfies (is a refinement of) the user provided end-to-end requirements.

Refinement of contracts: A contract C' is a refinement of contract C when the following conditions are satisfied:

- Assumptions of C' are the weaker set of assumptions of C
- Guarantees of C are the weaker set of guarantees of C'

Composition of contracts: Contracts C_1 and C_2 can be composed as $C_1 \otimes C_2$ when the following conditions are satisfied:

- If the guarantees of one component ($C_{1/2}$) are independent of the assumptions of the other, then the assumptions of $C_1 \otimes C_2$ are the stronger of the assumptions of C_1 and C_2 .
- If they are not independent, then the assumptions of $C_1 \otimes C_2$ are the weakest assumptions such that when they are conjuncted with the guarantees of C_1 (likewise C_2), the assumptions of C_2 (likewise C_1) are implied.
- Guarantees of $C_1 \otimes C_2$ are the conjunction of the guarantees of C_1 and C_2 .

Note that the outputs of one component that are inputs of the other are disregarded from $C_1 \otimes C_2$. Composition of contracts of this kind is useful when composing lower-level component contracts that bring a cause-effect chain into a higher-level subsystem contract.

Generation of the contracts is derived from the user's end-to-end requirements on the CPS. This can be done iteratively from the bottom up by accessing the capabilities of low-level components that make up the entire system. Each component is possibly assigned different contracts based on their functionalities to fulfill the given requirements. The whole resilience hierarchy is composed of the contracted components to form the system.

4.2.3 Observers

For every contract, observers check whether the contract violations occur based on the contract's expected behavior (i.e., guarantees) during runtime. It is possible for observers to be designed using heartbeats, time-stamping, finite state machines [35], timed automata [36], and hybrid automata [37] to enforce contractual obligations. When a failure of a contract occurs, a fault happens, and this is reported to the RM.

Chapter 5

Development and Implementation of the HCRF on a Fischertechnik Model

5.1 Model Factory

To demonstrate the potential benefits of the hierarchical approach described in the previous chapter, the case study presented in this chapter is based on a Fischertechnik training model to replicate an industrial CPS. This model factory, as shown in Figure 5.1 is a sorting line which sorts tokens based on their color into storage bins.

The parts of the model factory, including its actuators and sensors, are described below:

- **Light sensors:** Two light sensors for the detection of a token on the conveyor belt.
- **Color sensor:** This sensor provides an analog signal for color determination of a token.
- **Ejector:** One of three ejectors is used to push the color sorted token into the storage bins.

- **Storage bins:** There is a total of three storage bins, each with a light sensor.
- **Direct current motor:** The motor powers the rotation of the conveyor belt.
- **Pulse counter:** An encoder to track the movement of the conveyor belt through step counts.
- **Conveyor belt:** This physical belt transports the token to its bin.
- **Tokens:** There are one white, one red, and one blue colored token. However, only the white token is used in this case study.

FIGURE 5.1: Fischertechnik Training Model: Sorting line with color detection (EAN-CODE 4048962250404).

A token first enters the conveyor belt from the left and is detected by the first light sensor (LS_1). It moves along the conveyor belt and reaches the color sensor which then identifies the color of the token (i.e., white). As it moves along the conveyor belt, it would be detected by a second light sensor (LS_2). Once the token goes past this sensor, it reaches the ejectors which can eject the token into one of the three bins (i.e., Bin 1, Bin 2 or Bin 3). The white token is designated to be ejected into Bin 1. The movement of a token is tracked through the accounting of the number of steps traversed on the conveyor belt by the pulse counter.

Five components were designed to achieve the sorting process described above. A motor controller (MC) regulates the belt's rotation, and a pulse counter (PC) tracks the belt's steps. Tokens which are placed on the conveyor belt at LS_1 goes through the color sensor, which is triggered by the color processor (CP). A decision-making component, which is a bin selector (BS) in this case, determines the color of the token and sends that information to the ejector controller (EC). The EC then determines when to eject the token into its designated bin. This inter-component dependency creates an end-to-end latency requirement from the beginning where LS_1 is located, to the end where the bin resides. The operation flow of the model factory and its end-to-end latency requirement are illustrated in Figure 5.2 with the variables used listed in Table 5.1.

FIGURE 5.2: Operation flow of the interconnected components in the model factory.

TABLE 5.1: Input / Output variables for the system.

Variable	Definition
M_S	Motor Speed
$LS_{1/2}$	Light Sensor 1/2 Output
CV_{CP}	Annotated Color Value
SC	Current Step Count
SC_{CP}	Token Step Count at CP
SC_{BS}	Token Step Count at BS
E_{BS}	Bin ejection information for EC
T_{EC}	Trigger Ejector

5.2 Resilience Framework

The objective of the model factory is to sort tokens into their respective bins. In this case, the white token is to be sorted into Bin 1. As seen from the sorting process described in Section 5.1, multiple components are involved in making this happen. A fault could lead to a longer response time of a component, violating its latency contract. As a result, the end-to-end latency requirement may no longer be satisfied. Figure 5.3 shows the resilience hierarchy composed for this case study. At the lower levels, components CP and BS each have a latency contract (C_{CP} and C_{BS}) to guarantee their typical response times, C_L .

FIGURE 5.3: Resilience hierarchy of the components and contracts in the model factory.

Contract: C_{CP}

- **Inputs:** LS_1
- **Outputs:** SC_{CP} ; CV_{CP}
- **Parameters:** M_S
- **Assumptions:** $(M_S = S_1) \vee (M_S = S_2) \vee (M_S = S_3)$
- **Guarantees:** $LS_1 \implies (SC_{CP} \neq 0) \wedge (CV_{CP} \neq null)$ within $f_{CP}(M_S)$

Contract: C_{BS}

- **Inputs:** $SC_{CP}; CV_{CP}$
- **Outputs:** $SC_{BS}; E_{BS}$
- **Parameters:** M_S
- **Assumptions:** $(M_S = S_1) \vee (M_S = S_2) \vee (M_S = S_3)$
- **Guarantees:** $(SC_{CP} \neq 0) \wedge (CV_{CP} \neq null) \implies (SC_{BS} \neq 0) \wedge (E_{BS} \neq null)$ within $f_{BS}(M_S)$

Contract C_{LM} manages the two lower-level contracts allowing for a time duration that is minimally the response times of CP and BS but no more than the time taken for the token to reach LS_2 . Hence, the RM at the L_1 level has some flexibility for allowing either CP or BS to overrun their executions when faults occur. If even longer computation times are required, the RM at L_1 reports a fault to L_2 . This contract checks if both contracts C_{CP} and C_{BS} are satisfied and are generated using the contract composition technique described in Section 4.2.2.

Contract: C_{LM}

- **Inputs:** LS_1
- **Outputs:** $SC_{BS}; E_{BS}$
- **Parameters:** M_S
- **Assumptions:** $(M_S = S_1) \vee (M_S = S_2) \vee (M_S = S_3)$
- **Guarantees:** $LS_1 \implies (SC_{BS} \neq 0) \wedge (E_{BS} \neq null)$ within $f_{LM}(M_S)$

The contract in EC, C_{EC} monitors for the expected arrival of the token at LS_2 where the current step count SC needs to coincide with $(SC_{CP} + Offset)$, where $Offset$ is the number of steps between CP and LS_2 .

Contract: C_{EC}

- **Inputs:** $SC; SC_{CP}; LS_2$
- **Outputs:** None
- **Parameters:** None
- **Assumptions:** True
- **Guarantees:** $LS_2 \iff (SC = SC_{CP} + Offset)$

Finally, the root level contract C_{MC} is used by the L_2 RM of MC. C_{MC} is the composition of contracts C_{LM} and C_{EC} . This contract guarantees that all tokens

seen at LS_1 have a bin allocation before the token reaches LS_2 and that the token reaches LS_2 at the correct step count as it should.

Contract: C_{MC}

- **Inputs:** $LS_1; LS_2; SC; SC_{CP}$
- **Outputs:** $SC_{BS}; E_{BS}$
- **Parameters:** M_S
- **Assumptions:** $(M_S = S_1) \vee (M_S = S_2) \vee (M_S = S_3)$
- **Guarantees:** $[LS_1 \implies (SC_{BS} \neq 0) \wedge (E_{BS} \neq null) \text{ within } f_{LM}(M_S)] \wedge [LS_2 \iff (SC = SC_{CP} + Offset)]$

When the RM at L_1 reports a fault to L_2 , the resilience framework can rectify this problem by adjusting either contracts' latency parameters at runtime. The parameter used in this example is the motor speed, M_S . By adjusting the parameter, the RM ensures that the end-to-end requirement is once again satisfied. In this scenario, the higher level L_2 RM may choose to reduce the conveyor belt's speed (M_S) to satisfy the end-to-end timing requirement, whenever the underlying fault is significant. The two levels of resilience show the flexibility offered by the contract hierarchy, as it can compensate for a timing fault in one component using slack from another, thus avoiding this degradation in some cases or if necessary, degrade the throughput of the system but still maintain operations.

5.3 Implementation

As seen in Figure 5.4, instead of traditional PLCs commonly found in the industry, four Raspberry Pi (RPI) 3s are used to hold the control applications of the PC, CP, BS, EC, and MC. An RPI comes with a 1.2 GHz Quad-Core Processor, 1 GB RAM, and multiple General Purpose Input-Output (GPIO) pins. Each RPI runs the Jessie Raspbian GNU/Linux 8.0 operating system (kernel version 4.9.35-v7). An Arduino Pro Mini microcontroller is used as an analog to digital converter to process the analog color sensor output for the RPI. Additionally, due to the voltage differences between the RPIs and the model factory, voltage converters are used to interface them together. All the RPIs are interconnected over Ethernet through a network switch.

FIGURE 5.4: Raspberry Pis containing the control and resilience management logic of the various components, as well as the Arduino microcontroller and various electronics.

The software implementation of the resilience framework, as well as the sorting line application, is done on 4DIAC [15], an open source framework for event-driven industrial automation and control that follows the IEC 61499 standard [19]. It provides a development environment shown in Figure 5.5, which shows the function blocks for the CP component. 4DIAC also provides a runtime environment, *FORTE*, which runs on the RPI.

In Figure 5.5, the lower three pink function blocks depicted belong to the local CP RM, the top left block represents the application logic of CP, and the right-most block shows the observer for the contract. Each contract is associated with a corresponding observer to monitor for its violations. This arrangement allows for segregation between application code and fault handling code. Communication between function blocks is handled through the use of an in-built Publisher/Subscriber mechanism. Two main types of function blocks are used, the basic and the composite. Figure 5.6 represents how a standard function block interface looks. The top half interface has event I/O connections presented by the red dots, and the lower half has the data inputs and outputs represented by the blue dots.

FIGURE 5.5: 4DIAC Integrated Development Environment

FIGURE 5.6: Function Block Interface: The event and data connections of the color processor application.

A basic function block has its functionality described by an Execution Control Chart, which is a state diagram, as shown in Figure 5.7. Each state can have multiple actions. Each action has either one or zero algorithms and one or zero events. The algorithms in 4DIAC are written in Structured Text or C++.

FIGURE 5.7: Execution Control Chart of a basic function block.

The composite function block has its functionality defined by a function block network, as seen in Figure 5.8. The function block network can consist of any of the two types of function blocks (i.e., basic or composite function blocks).

5.4 Evaluation

5.4.1 Fault Scenarios

The following six fault scenario types can happen if faults were manually injected into the model factory through the RPIs and the 4DIAC function blocks:

1. **CP violation:** C_{CP} is violated (i.e., $C_L > f_{CP}(M_S)$) but $C_L + f_{BS} \leq f_{LM}$.
2. **BS violation:** Same as above.
3. **CP and BS violation:** L_1 RM still has sufficient slack.
4. **L_1 violation:** L_1 RM reports a fault to L_2 RM.
5. **EC violation:** EC RM reports a fault to L_2 RM.
6. **EC and L_1 violation:** Both the RMs of EC and L_1 reports a fault to L_2 RM.

FIGURE 5.8: Composite function block network of the color processor application.

5.4.2 Performance Comparison

We aim to compare the time and amount of communication required for fault-recovery in our resilience framework compared to representative (hypothetical) designs of fully centralized and fully decentralized resilience frameworks. These hypothetical designs do not have the concept of a management hierarchy and are illustrated in the following Figure 5.9.

FIGURE 5.9: Hypothetical designs of a fully centralized and fully decentralized resilience framework.

As with any fully centralized design, faults that occur in any component will be sent to a centralized manager. For the case study described in the earlier section, there are three components (CP, BS, and EC) and one centralized L_1 RM on the MC. This would require one message for fault reporting and three messages for a response (one to each component) for every fault that occurs.

In a fully decentralized design, the four components (CP, BS, EC, and MC) and their RMs will communicate with one another. Any fault occurrence would require each RM to reach consensus for fault-recovery. Assuming that the best design to reach consensus requires three sets of messages: i) fault reporting, ii) response with a possible solution and iii) the chosen solution. This requires nine messages in total for each fault that happens. In both of the centralized and decentralized designs, components are assumed to have fault detection capabilities to ensure that the comparison is fair.

Our resilience framework and the theoretical frameworks were evaluated with the fault scenarios mentioned in Section 5.4.1. Each evaluation run would have scenario types 1, 2, and 3 occurring twice and types 4, 5, and 6 occurring once. This results in a total of 12 faults during each run because scenario types 3 and 6 generate two faults each.

FIGURE 5.10: The number of inter-component communication messages required for the different framework designs.

The total number of messages generated by each design is shown in Figure 5.10. Our framework had 21 messages generated while the fully centralized and decentralized frameworks each had $4 \times 12 = 48$ and $9 \times 12 = 108$ messages, respectively. This translates to communication savings of 56% and 81% when compared to the two designs.

FIGURE 5.11: The time spent on fault recovery for the different framework designs.

As for the amount of time spent for fault-recovery, measurements from our model factory showed message latency of 1ms and time taken for decision-making to be 0.5ms on average. Our framework would require one decision-making step at either L_1 or L_2 for fault scenarios 1, 2, 3 and 5, while types 4 and 6 require two (both at L_1 and L_2). Thus, the time required for each framework design is shown in Figure 5.11. A video explanation of our framework implementation can be viewed at [38].

5.4.3 Advantages and Limitations of the Framework

When a fault arises in the system, our communication protocol design dictates that only fault messages and contract parameter values are exchanged between the RMs. Original root contract and sub-contract specifications do not change at run-time. As soon as a solution is found within the hierarchy, the recovery process stops, allowing us to reduce the communication overheads as compared to existing fully centralized and decentralized architectures. Our hierarchical approach is also robust against single-point failures, to a certain extent. The possible recovery solutions that our hierarchical approach offers can be identical to those provided by centralized or decentralized architectures. These solutions are only constrained by how the designers had planned for the diverse number of fault scenarios that can occur. Let us assume that in a centralized manager, generating a recovery solution requires information to be sourced from different remote components. This is equivalent to our framework when a local RM fails to find a suitable recovery method and passes along the fault information upwards to its parent RM. This gradual flow of knowledge would eventually result in a higher-level RM receiving all the fault messages before deciding on the appropriate course of recovery. The same logic would hold when comparing with a decentralized architecture, as the information flow between components in a decentralized design requires for a significantly more complicated protocol. This results in greater efforts to make sure a solution that can be achieved in a centralized manager can be obtained by a decentralized architecture.

The types of faults that the framework currently handles are restricted to faults that are computational in nature and those which can be detected by a software approach. We currently focus on the non-functional aspects of the system, which has an impact on the overall operation of the system. In our model factory, we handled computational faults (CP and BS violations) with regards to its execution latency as well as an implicit physical fault (EC violation). Both the contracts assigned to CP and BS are related to their execution latencies in providing an output within a time limit based on the physical properties of the model factory. A failure in either contracts meant that the computational aspect of the components had failed to conform to the original user requirements, and this computation failure would not be able to successfully eject the token into its respective bin. However, as long as there is still an output coming from the failed component, we assumed

that the device was still working (i.e., had a higher CPU load at that point of time) and could facilitate a recovery method by slowing down the motor speed of the conveyor belt. This indirectly provided more time for the computation to occur, and would result in a successful token ejection albeit with a lower throughput. Similarly, if the component had failed to meet its contractual requirements, it is possible that the component failed and was no longer responsive. For this kind of failure, contracts which check for heartbeats coming from the different components can be implemented as well. This type of failure can cover a wide range of devices from sensor nodes to computational or communication systems, and to actuation hardware. Apart from computational fault on execution latency, we can monitor for other non-functional properties such as power utilization, and current or forecasted throughput of the system. Monitoring for power utilization can provide useful insights on the machinery running. A machine that is drawing excess power may be in need of maintenance or could be faulty and requires an overhaul. Throughput monitoring of the system would tell us how well the production line is running, which is part of measuring for overall equipment effectiveness, a common performance indicator used in manufacturing. As long as the fault can be captured by our framework, it would be possible to devise methods for fault recovery.

Communication infrastructure disruptions were not covered in this work but have been explored in a similar fashion [39]. The authors proposed the idea of having a contract-based framework to manage the communication delays of network flows in industrial setups. The framework was combined with Software-Defined Networking (SDN) where the network components are associated with delay contracts and managed by a resilience manager. The SDN is required for management of the network flows in the networking infrastructure. In the event of a delay or failure, the RM would decide on the best response strategy through a delay-aware path finding algorithm to reroute the network flows to provide resilience.

5.5 Experience on Development with IEC 61499

BlokIDE provides a immersive design environment for Model-Driven Engineering of programmable electronics (i.e., Programmable Integrated Circuits to PLCs) [40]. It provides auto-generation of ISO-C code that can run on a variety of platforms

as long as a C compiler was available for it. We first started our development experimenting with the BlokIDE platform. The auto-generation of ISO-C code was extremely appealing for embedded devices that we had targeted for deployment. While the generated ISO-C files are marketed as human readable, the computer generated variables do get confusing as the project grows larger. This problem was exacerbated when we needed to include communication between the devices that were designed with BlokIDE as there was no built-in protocols for this. It was tedious to manually modify the C-code generated to include a communication protocol. As we needed a communication protocol that was versatile for our needs, we chose Data Distribution Service (DDS) by Real-Time Innovations. DDS is a publish/subscribe messaging service and provides open interfaces which allows for portability, interoperability and is very feature rich. However, the feature richness of DDS also proved to be a disadvantage as it needed to be configured extensively. This added to the complexity of implementing DDS along with the C-code generated by BlokIDE. BlokIDE is available as an extension for Visual Studio versions 2010 and 2013, which has since ceased development. While the two versions can still be downloaded from the Microsoft website as of today, we do not know when this support will end. Lastly, since this is a research developed tool, help documentations on the tool were inadequate.

HOLOBLOC was the first prototypical implementation of the IEC 61499 and was originally developed by Rockwell Automation, led by Dr. Jim Christensen. It is a software that enables users to build and test data types, function block types, adapter types, functions, resources types, device types, network segment types and system configurations according to the IEC 61499 standard [20]. It is now managed by HOLOBLOC INC., currently led by Dr. Jim Christensen, a for-profit organization which provides consultation and customized training for the IEC 61499 Standard and its associated (FBDK) software. It uses the standard XML format to describe function blocks and is written in Java. The runtime environment, FBRT, is also a Java-based implementation, which runs over a Java Virtual Machine. Although standard IEC 61499 function blocks are provided by the Function Block Development Kit (FBDK), device specific features need to be written by the designers themselves. One example would be to access the GPIO pins on the RPI, and this separates the design process from the FBDK. Once again, this would add onto the development efforts of the user. Moreover, the

user interface for FBDK was not intuitive and help documents and tutorials were lacking.

Ultimately, we chose 4DIAC because it provided a comprehensive infrastructure for developing with IEC 61499. Firstly, it is the only open source tool that is up to date, apart from HOLOBLOC. It comes with an IDE, a runtime environment, and makes use of the function block library provided by HOLOBLOC. Moreover, it has an extensive array of help available online. For instance, it has a dedicated forum, where users and developers provide their insights, and a well-structured help documentation which has step-by-step tutorials. Finally, it has been tested with several types of hardware devices and have the corresponding device specific function blocks which users can easily access and can focus solely on developing their own applications. It is noteworthy that other open source tools are available, such as OOONEIDA-FBench [41], ICARU_FB [42], GASR-FBE [43]. However, these tools have not been updated for quite some time. For users that require support and a stable platform, they can refer to ISaGRAF [44], the first commercial tool by Rockwell Automation, nxtSTUDIO [45] by NXT Control or to the Function Block Service Runtime [46] by Yue Yi Automation.

Chapter 6

Automated Toolchain

Currently, contracts are manually written based on the hardware and capabilities of the components in the CPS and its user requirements. This method is tedious, error-prone, and would not scale. A method which can automate this process is ideal and beneficial to large-scale CPS. Firstly, we want to be able to store user requirements and hardware capabilities of the CPS in a human-readable format, the AML file. Next, this AML file gets parsed into the toolchain, which then decomposes the contracts defined within. The decomposition is based on the user requirements and the component information available. Lastly, the resulting sub-contracts after decomposition can be transferred to the IEC 61499 4DIAC platform by creating the corresponding observer function blocks for each sub-contract automatically. To this end, we developed a software toolchain that fulfills the entire process and will be describing it in the following sections.

6.1 AutomationML

6.1.1 Describing Hardware Capabilities

The model factory used in the case study from Section 5.1 will serve as our reference for building the AML file to describe its hardware capabilities. We use the AML Editor v5.12 provided by AutomationML e.V. to create, edit, and visualize the AML file. Here, we first define how the components of the model factory translate onto the AML structure.

6.1.1.1 Sensors / Actuators

FIGURE 6.1: AutomationML: Sensor and actuator information embedded within its own internal element (i.e., an object).

Figure 6.1 shows part of the AML editor with the information on the sensors and actuators under the SystemUnitClassLib. As AML is object-oriented, the system unit class library describes concrete types of objects reused within engineering [22]. The highlighted sensor, Light Sensor 1 has a “Sensor” role attached to it. Role classes are used to attach generic semantics to an AML object instance and to describe requirements of this object instance [47]. Light Sensor 1 also has an attribute attached, which refers to a property belonging to this AML object [22]. This particular attribute defines the sensor’s output information (LS1), that is of the Boolean data type. Actuators are given the role of the “Actuator” accordingly, but each actuator has input as an attribute instead. For example, the compressor would have an input to turn it on or off.

6.1.1.2 Computation

Similarly, to represent the computational hardware components used in the model factory, the four Raspberry Pis with their functionalities are shown in Figure 6.2. As an example, we focus on RPI 1, which classifies as an “EmbeddedDevice”.

FIGURE 6.2: AutomationML: Computation resources and the applications are structured and described as such.

Under the RPI 1 object instance, it defines the RPI's interfaces which has one Ethernet port given the class of "SignalInterface". The "SignalInterface" describes a single connection point of an AML object and provides the ability for the object to be linked with another interface through CAEX InternalLinks [22], which we show later. RPI 1 also houses two separate "Resource", the Pulse Counter and the Color Processor, each nesting the "Process", Step_Generation and Color_Identification, respectively. The two resources here reflect the PC and CP components we have discussed earlier in Chapter 5 and shown in Figure 5.2. (Note how the inputs and outputs of the CP component are identically mapped.) This allows our decomposition algorithm (explained later on) to be able to map out the components required to satisfy the root contract requirements. Component information on their execution latencies, such as its mean and standard deviation, are also stored here.

Some other information on the model factory are also present. The different colored work tokens classify under "Product"; "Structure" defines the conveyor belt and bins present; "Communication" has a network switch described which is used in the setup.

6.1.1.3 Inter-connections

In Section 6.1.1.2, we mentioned the use of CAEX InternalLinks, which are for connecting objects to one another. While the Interface class gives us the ability to store information on the interface used, InternalLinks [22] show us how the objects connect to one another. This is illustrated in Figure 6.3 where each Raspberry Pi's Ethernet port links to a corresponding Ethernet port on the network switch.

FIGURE 6.3: AutomationML: InternalLinks illustrated as blue dotted lines connecting the Raspberry Pis to the network switch.

6.1.2 User Requirements

User requirements of the model factory needs to be translated into the form of contracts, which can then be stored in the AML file. There is a non-functional end-to-end requirement on the system to make sure that the total response time, from the first input LS_1 to the trigger of the ejector T_{EC} , is satisfied. Figure 6.4 shows the root contract for this requirement that needs to be satisfied within 1720 ms. Similarly to the contract attributes mentioned earlier, we define the inputs, outputs, parameters, assumptions, guarantees, and non-functional properties associated with the root contract. This information is then used when decomposing

the root contract into sub-contracts subsequently. Listing 6.1 shows a snippet of the AML file that is generated for the model factory.

FIGURE 6.4: AutomationML: Root contract of the model factory showing its end-to-end requirement.

6.2 Python Program

Now that we have the AML file generated, an AML interface needs to be developed to extract and make sense of the information available. To do this, we developed a Python program that achieves this. The underlying structure of the AML file is XML-based, and that information is stored in an InstanceHierarchy of InternalElements [22]. Since the XML data is organized in such a hierarchy and we know the generic names of the elements in this structure, information of the CPS from the AML file can be extracted accordingly.

6.2.1 Decomposition

Since a root contract C_r is defined within the AML file, we first obtain its i) inputs I_r , ii) outputs O_r , iii) assumptions A_r , iv) guarantees G_r , and lastly v)

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <CAEXFile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.dke.de/CAEX" SchemaVersion="3.0" FileName="
  RedFactory_Extended.aml" xsi:schemaLocation="http://www.dke.de/
  CAEX CAEX_ClassModel_V.3.0.xsd">
3   <AdditionalInformation AutomationMLVersion="2.0" />
4   <SuperiorStandardVersion>AutomationML 2.10</
    SuperiorStandardVersion>
5   <SourceDocumentInformation OriginName="AutomationML Editor"
    OriginID="916578CA-FE0D-474E-A4FC-9E1719892369" OriginVersion="
    5.1.1.0" LastWritingDateTime="2018-11-15T13:20:42.5055484+08:00
    " OriginVendor="AutomationML e.V." OriginVendorURL="www.
    AutomationML.org" OriginRelease="5.1.1.0" OriginProjectTitle="
    unspecified" OriginProjectID="unspecified" />
6   <InstanceHierarchy Name="RedFactory">
7     <Version>0</Version>
8     <InternalElement Name="Sorting Line 1" ID="463399c4-a3e5-41fd
    -9bac-e31d679ef97c" RefBaseSystemUnitPath="Fischertechnik
    Training Models/ Sorting Line with Color Detection">
9       <InternalElement Name="Sensors" ID="651b1036-e83e-47b1-be7d-
    aca0599964b4">
10        <InternalElement Name="Light Sensor 1" ID="3dde7190-438d
    -4579-81b0-f2f6066afdd3">
11          <Attribute Name="Output" AttributeDataType="xs:boolean">
12            <Attribute Name="LS1" AttributeDataType="xs:boolean">
13              <DefaultValue>False</DefaultValue>
14            </Attribute>

```

LISTING 6.1: Sample XML of the AML file generated for the model factory.

non-functional property (NFP) of interest x_r . Likewise, we gather all available components information: i) inputs I_i , ii) outputs O_i , and iii) estimated value x_i of the NFP. Only after the complete extraction of component information and knowledge of the root contracts stored within the AML file, we can then begin the decomposition process. Algorithm 1 is simple and intuitive. From Figure 5.2, we know that there is a dependency of the various components in the system based on their input and output relationship. We decompose the root contract by identifying a chain of dependencies among the components such that the outputs O_i of a preceding component leads to the inputs I_j of the next component. The search continues until a set of chained components matches the original set of inputs and outputs of the root contract. For every component involved in the chain, we formulate a sub-contract. The algorithm defines the guarantees of the sub-contracts based on the component's inputs and outputs, while assumptions are mirrored from the root contract as each hierarchy of contracts holds the same assumptions. Inputs and outputs for the sub-contracts remain as they were from

Algorithm 1 Contract Decomposition and Generation

```

procedure DECOMPOSE
  Given  $C_r = (I_r, O_r, A_r, G_r(x_r))$ 
  for each component  $j, j < n$  in AvailableComponents do
    if ( $I_r == I_j$ ) then
      Store component  $j$  in DependencyChain
    end if
    for each component  $k, k < n$  components do
      while ( $O_r \neq O_k$ ) do
        if ( $O_j == I_k$ ) then
          Store component  $k$  in DependencyChain
        end if
      end while
    end for
  end for
end procedure
procedure FORMULATECONTRACTS(DependencyChain)
  for each component  $i$  in DependencyChain do
    Assign  $I_i$  and  $O_i$ 
    Replicate  $A_i$  from  $C_r$ 
    Define  $G_i(x_i)$  from its  $I_i$  and  $O_i$  variables
  end for
end procedure

```

the components. x_i values for the contracts are averaged from repeated component execution runs which are also stored in the AML file.

6.2.2 4DIAC Function Blocks

Lastly, what is left to be done is the deployment of the generated sub-contracts onto the 4DIAC platform. For each of the sub-contracts, a corresponding observer function block related to the monitoring of the NFP is created. For the sub-contracts in our case, latency observers are created for each of the components CP, BS, and EC. An FBT file based on XML, describes a function block. A partial sample of the latency observer FBT file is shown in Listing 6.2. Recall, there is a need to separate fault-handling code from application code. Here, the function blocks generated are used solely for resilience management. Application developers can focus on writing application code separately before combining them with the resilience function blocks at a later stage.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/
  LibraryElement.dtd">
3 <FBType Comment="Modified for realtime clock reference in linux (
  CLOCK_REALTIME)" Name="LatencyObserverV3">
4   <Identification Standard="61499-2"/>
5   <VersionInfo Author="Daniel Ng" Date="2019-06-20" Organization="
  NTU" Version="0.0"/>
6   <InterfaceList>
7     <EventInputs>
8       <Event Comment="Initialization Request" Name="INIT" Type="
  Event">
9         <With Var="Latency"/>
10      </Event>
11      <Event Comment="Start Execution Request" Name="StartTrigger"
  Type="Event"/>
12      <Event Comment="End Execution " Name="EndTrigger" Type="
  Event"/>
13      <Event Comment="Update from Resilience Manager" Name="
  LatencyUpdate" Type="Event">
14        <With Var="Latency"/>
15      </Event>
16      <Event Comment="Expiry from timer " Name="LatExpired" Type="
  Event"/>
17    </EventInputs>
18    <EventOutputs>
19      <Event Comment="Initialization Confirm" Name="INITO" Type="
  Event">
20        <With Var="ReqLatency"/>
21        <With Var="ElapsedTime"/>
22      </Event>
23      <Event Comment="Latency exceeded observation" Name="
  LatFailure" Type="Event"/>
24      <Event Comment="Epsilon update if it exceeds given latency"
  Name="LatRequest" Type="Event">
25        <With Var="ReqLatency"/>
26      </Event>
27      <Event Comment="Elapsed time update" Name="ElapsedTimeE"
  Type="Event">
28        <With Var="ElapsedTime"/>
29      </Event>
30    </EventOutputs>
31    <InputVars>
32      <VarDeclaration Comment="Parameter - Latency for execution"
  Name="Latency" Type="ULINT"/>
33    </InputVars>
34    <OutputVars>
35      <VarDeclaration Comment="Exceeded execution time" Name="
  ReqLatency" Type="ULINT"/>
36      <VarDeclaration Comment="Actual execution time" Name="
  ElapsedTime" Type="ULINT"/>
37    </OutputVars>
38  </InterfaceList>

```

LISTING 6.2: Sample FBT file that describes the latency observer generated for the sub-contracts to be imported into the 4DIAC Integrated Development Environment

Chapter 7

Industrial Testbed

We demonstrate our framework on an industrial testbed, the Festo Didactic C-P Factory. In order to showcase the framework with better interoperability, we incorporated OPC-UA functionality into 4DIAC as part of the demonstration. The OPC-UA standard is currently undergoing development and revisions, but numerous industrial vendors have started to include them into their product line [48]. Going forward, we will only see wider adoption of this standard as we head towards Industry 4.0.

7.1 Festo Didactic Cyber-Physical (C-P) Factory

FIGURE 7.1: The Festo Didactic Cyber-Physical Factory.

Figure 7.1 shows the entirety of the assembly line. This demonstrator was designed to mimic the assembly of a mobile phone. The process starts in Station A (Automated Storage / Retrieval System (ASRS)) where the front covers are stored. These front covers are placed on workpiece carriers. When a work order comes along with a pallet, the Manufacturing Execution System 4 (MES4) software provides a workpiece location, which the 2-axis robot arm then goes to pick it up and place it onto the pallet that is on the conveyor belt. Station B is where drilling is simulated for the holes on the front cover. The pallet then moves to Station E (Robot Assembly) where the raw materials (printed circuit board (PCB) and fuses) are stored and comes with a camera for optical inspection. The robot uses a pneumatic gripper to place the workpiece front cover from the bypass conveyor in the station onto its working position. It then changes to another pneumatic gripper that obtains the PCB from a storage box and places it within the front cover. A third pneumatic gripper is then employed to place a fuse onto the PCB. The robot then changes back to its first gripper to return the assembled workpiece onto the pallet at the bypass conveyor. The pallet travels to Station F (Magazine) where the back cover is placed, and the following Station G presses it into place. The process ends back at the ASRS station, which stores the completed mobile assembly together with the remaining unfinished mobile front covers. At every station, the assembly line employs RFID technology to identify and track the state of each pallet, and this information is conveyed back to MES4 and stores the data in a centralized database. It is also crucial to note that the assembly line works in a centralized manner. Whenever a pallet arrives at the stations, the PLC sends the RFID information (carrier ID) and requests for instructions.

7.1.1 Control

Out of the entire assembly line, we made use of only the Drilling, Camera, and ASRS stations. Instead of relying on the original MES4 to control and actuate the line, we replaced it with our software. The control logic and resilience framework were programmed with 4DIAC and executed on RPI 3s. At the heart of every station lies a Siemens Simatic ET 200SP coupled with a 1512SP CPU. This PLC remains to provide an OPC-UA server and serves as the interface for all the onboard I/O, and we created OPC-UA clients on 4DIAC to communicate with the PLCs.

All communication between the PLCs and RPIs were accomplished through OPC-UA.

7.1.2 Resilience Framework

For demonstrating the resilience framework on the line, we came up with the following four fault scenarios:

1. Machine failure (inductive sensor) that is used for the detection of a pallet's arrival.
2. Machine failure (RFID sensor) that is needed to read the carrier number on a pallet.
3. Scenarios 1 and 2 happening concurrently.
4. MES failure.

Since the original MES was no longer being used, similar MES-like functionality was created to mimic its job. The system configuration of 4DIAC is as shown in Figure 7.2. The workflow of each station is described in the following sections.

7.1.2.1 Drilling Station

1. Upon pallet arrival, the stopper observer checks if both inductive sensors were triggered within 2 seconds of the first sensor.
2. Station reads the RFID, checks with the MES on which drilling action needs to be done based on carrier ID, and stores action associated with the pallet in a historical buffer for MES failure. The station also sends the carrier ID to the following station (Camera).

7.1.2.2 Camera Station

1. Upon pallet arrival, the stopper observer checks if both inductive sensors were triggered within 2 seconds of the first sensor.

FIGURE 7.2: System configuration of the demonstration on the Festo Didactic Cyber-Physical Factory.

2. Station reads the RFID, and sends the carrier ID to the following station (ASRS).

7.1.2.3 ASRS Station

1. Upon pallet arrival, the stopper observer checks if both inductive sensors were triggered within 2 seconds of the first sensor.
2. Station reads the RFID, and checks with MES on whether to load, store or let the pallet through, based on its carrier ID. The station then sends the carrier ID to the following station (Drilling).
3. RFID observer checks if the RFID sensor throws an error, which indicates failure. If so, the station retrieves the carrier ID from stored RFID buffer sent from the previous station (Camera).

When the MES fails during its operation, historical data stored at the individual stations are used to compensate for the required information so that the assembly line continues production. A video clip on the demonstration is available at [49].

Chapter 8

Conclusion and Future Work

This work aims to develop a resilient cyber-infrastructure for Cyber-Physical Systems (CPSs). A resilience framework based on contracts was developed and implemented on a model testbed, which represented a few practical scenarios. The International Electrotechnical Commission 61499 standard for distributed systems was also explored to achieve a clear separation of application code and fault handling code. It also made the applications modular which can be distributed among several host controllers such as the Raspberry Pi. Experimental results showed in Chapter 5 revealed promising results for our framework in terms of message savings of 56% and 81% when compared to fully centralized and decentralized designs. Our framework also had shorter fault-recovery timings.

We then made the framework more comprehensive by providing a methodology for specifying user requirements and hardware information of the CPS in a human-readable format through AutomationML (AML). The AML file parses through our software toolchain to automatically generate contracts for deployment. The process completes with the contracts being deployed as observers in 4DIAC.

Finally, to complete the thesis, we deployed the resilience framework on the Festo Didactic Cyber-Physical Factory testbed to illustrate some potential benefits of having our framework. Machine-to-machine communication was realized through the Open Platform Communications Unified Architecture to present how seamless it was to interface the different hardware together if and when vendors are willing to follow a standardized protocol.

While we have presented an automated toolchain that extracts information, decomposes root contracts, and deploys the sub-contracts onto the 4DIAC platform, more can be done to improve the software toolchain. The AML file has the potential to extend our work further. For example, engineers could provide application code and we can link it to the associated hardware. Another missing element would be to have a list of recovery alternatives available for the end-users of the CPS. This list can also be part of the AML file. With the additional information, our tool could not only generate the observer function blocks but also generate the function blocks for application and resilience management. Besides all these, the current decomposition algorithm is quite elementary. Further research needs to be done to provide a more formal and generic way for decomposition. We also need a way of refining the sub-contracts generated while still fulfilling the original root contract.

Bibliography

- [1] B. Marr. Why Everyone Must Get Ready For The 4th Industrial Revolution, 2016. URL <https://www.forbes.com/sites/bernardmarr/2016/04/05/why-everyone-must-get-ready-for-4th-industrial-revolution/#683740cc3f90>. [xiii](#), [1](#)
- [2] K. L. Lueth. Will the Industrial Internet Disrupt the Smart Factory of the Future?, Mar 2015. URL <https://iot-analytics.com/industrial-internet-disrupt-smart-factory/>. [xvii](#), [2](#), [4](#)
- [3] J. Lee, B. Bagheri, and H. Kao. A Cyber-Physical Systems Architecture for Industry 4.0-based Manufacturing Systems. *Manufacturing Letters*, 3: 18 – 23, 2015. ISSN 2213-8463. doi: <https://doi.org/10.1016/j.mfglet.2014.12.001>. URL <http://www.sciencedirect.com/science/article/pii/S221384631400025X>. [1](#)
- [4] 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 Transactions on Industrial Informatics*, 13(2):725–736, April 2017. ISSN 1551-3203. doi: [10.1109/TII.2016.2595401](https://doi.org/10.1109/TII.2016.2595401). [1](#)
- [5] J. C. Laprie. From Dependability to Resilience. *DSN, Anchorage, AK, USA*, 8, 01 2008. [1](#)
- [6] McKinsey Digital. Industry 4.0 after the Initial Hype, 2016. URL https://www.mckinsey.com/~media/mckinsey/business%20functions/mckinsey%20digital/our%20insights/getting%20the%20most%20out%20of%20industry%204%200/mckinsey_industry_40_2016.ashx. [2](#)

-
- [7] J. Hurley. Why the UK Must Invest in Smart Factories, March 2018. URL <https://www.raconteur.net/technology/uk-must-invest-smart-factories>. 2
 - [8] ANSI ISA. ISA-95.00. 03-2005, Enterprise Control System Integration, Part 3: Activity Models of Manufacturing Operations Management. *System, and Automation Society*, 2005. 3
 - [9] S. A. Boyer. *SCADA: Supervisory Control And Data Acquisition*. International Society of Automation, USA, 4th edition, 2009. ISBN 1936007096, 9781936007097. 3
 - [10] M. Hermann, T. Pentek, and B. Otto. Design Principles for Industrie 4.0 Scenarios. In *2016 49th Hawaii international conference on system sciences (HICSS)*, pages 3928–3937. IEEE, 2016. 3, 4
 - [11] F. J. N. de Santos and S. G. Villalonga. Exploiting Local Clouds in the Internet of Everything Environment. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 296–300. IEEE, 2015. 3
 - [12] M. Hankel and B. Rexroth. The Reference Architectural Model Industrie 4.0 (RAMI 4.0). *ZVEI*, 2:2, 2015. 4, 9, 11
 - [13] D. Gorecky, M. Schmitt, M. Loskyll, and D. Zühlke. Human-Machine-Interaction in the Industry 4.0 Era. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 289–294, July 2014. doi: 10.1109/INDIN.2014.6945523. 4
 - [14] M. S. Haque, D. J. X. Ng, A. Easwaran, and K. Thangamariappan. Contract-Based Hierarchical Resilience Management for Cyber-Physical Systems. *Computer*, 51(11):56–65, Nov. 2018. ISSN 0018-9162. doi: 10.1109/MC.2018.2876071. URL doi.ieeecomputersociety.org/10.1109/MC.2018.2876071. 6, 17
 - [15] A. Zoitl, T. Strasser, and G. Ebenhofer. Developing Modular Reusable IEC 61499 Control Applications with 4DIAC. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 358–363, July 2013. doi: 10.1109/INDIN.2013.6622910. 6, 11, 29

- [16] Sorting Line With Color Detection 24V - Education. URL <https://www.fischertechnik.de/en/products/teaching/training-models/536633-edu-sorting-line-with-color-detection-24v-education>. 7
- [17] CP Factory – The Cyber-Physical Factory. URL <https://www.festo-didactic.com/int-en/learning-systems/learning-factories,cim-fms-systems/cp-factory/cp-factory-the-cyber-physical-factory.htm?fbid=aW50LmVuLjU1Ny4xNy4xOC4xMjkzLjc2NDM>. 7
- [18] Unified Architecture. URL <https://opcfoundation.org/about/opc-technologies/opc-ua/>. 9
- [19] A. Zoitl. *Real-Time Execution for IEC 61499*. ISA, 2008. ISBN 1934394270, 9781934394274. 11, 29
- [20] HOLOBLOC INC. Resources for the New Generation of Automation and Control Software. URL <https://www.holobloc.com/>. 11, 37
- [21] A. Lüder N. Schmidt. AutomationML in a Nutshell. Technical report, 2015. 12
- [22] AutomationML Consortium. Whitepaper AutomationML Part 1 - Architecture and general requirements. Technical report, 2016. 12, 40, 41, 42, 43
- [23] AutomationML Consortium. Whitepaper AutomationML Part 4 - AutomationML Logic. Technical report, 2017. 12
- [24] J. Gertler. *Fault Detection and Diagnosis in Engineering Systems*. New York : Marcel Dekker, c1998., 1998. ISBN 0824794273. 13
- [25] A. Laszka, W. Abbas, Y. Vorobeychik, and X. Koutsoukos. Synergistic Security for the Industrial Internet of Things: Integrating Redundancy, Diversity, and Hardening. In *2018 IEEE International Conference on Industrial Internet (ICII)*, pages 153–158, Oct 2018. doi: 10.1109/ICII.2018.00025. 13
- [26] Y. Zhang and J. Jiang. Bibliographical Review on Reconfigurable Fault-Tolerant Control Systems. *Annual reviews in control*, 32(2):229–252, 2008. 13, 14

- [27] P. Mhaskar, A. Gani, N. H. El-Farra, C. McFall, P. D. Christofides, and J. F. Davis. Integrated Fault-Detection and Fault-Tolerant Control of Process Systems. *AIChE Journal*, 52(6):2129–2148, 2006. [13](#)
- [28] D. Ratasich, O. Höftberger, H. Isakovic, M. Shafique, and R. Grosu. A Self-Healing Framework for Building Resilient Cyber-Physical Systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 133–140, May 2017. doi: 10.1109/ISORC.2017.7. [14](#)
- [29] S. Eisele, I. Mardari, A. Dubey, and G. Karsai. RIAPS: Resilient Information Architecture Platform for Decentralized Smart Systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 125–132, May 2017. doi: 10.1109/ISORC.2017.22. [14](#)
- [30] M. Garcia Valls, I. R. Lopez, and L. F. Villar. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 9(1):228–236, Feb 2013. ISSN 1551-3203. doi: 10.1109/TII.2012.2198662. [15](#)
- [31] K. Güttel. *Konzept zur Generierung von Steuerungscode für Fertigungsanlagen unter Verwendung wissensbasierter Methoden*. Fortschritt-Berichte VDI / 20. VDI-Verlag, 2013. ISBN 9783183444205. URL <https://books.google.com.sg/books?id=iVw5mwEACAAJ>. [15](#)
- [32] M. Steinegger, A. Zoitl, M. Fein, and G. Schitter. Design Patterns for Separating Fault Handling from Control Code in Discrete Manufacturing Systems. In *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pages 4368–4373, Nov 2013. doi: 10.1109/IECON.2013.6699838. [15](#)
- [33] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J. Raclet, P. Reinke-meier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen. Contracts for System Design. Research Report RR-8147, INRIA, November 2012. URL <https://hal.inria.fr/hal-00757488>. [17](#), [20](#)
- [34] E. S. Kim, M. Arcak, and S. A. Seshia. A Small Gain Theorem for Parametric Assume-Guarantee Contracts. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, pages 207–216. ACM, 2017. [19](#)

- [35] Z. E. Bhatti, R. Sinha, and P. S. Roop. Observer Based Verification of IEC 61499 Function Blocks. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 609–614, July 2011. doi: 10.1109/INDIN.2011.6034948. 21
- [36] L. Mhamdi, B. Maaref, H. Dhouibi, H. Messaoud, and Z. S. Abazi. Diagnosis of Hybrid Systems through Observers and Timed Automata. In *2016 International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 164–169, April 2016. doi: 10.1109/CoDIT.2016.7593554. 21
- [37] T. A. Henzinger. The Theory of Hybrid Automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, July 1996. doi: 10.1109/LICS.1996.561342. 21
- [38] D. J. X. Ng. Contract-based Hierarchical Resilience Management for Cyber-Physical Systems, 2018. URL <https://youtu.be/bmqxD0Jgaz4>. 34
- [39] R. H. Jhaveri, R. Tan, A. Easwaran, and S. V. Ramani. Managing Industrial Communication Delays with Software-Defined Networking. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, Aug 2019. doi: 10.1109/RTCSA.2019.8864557. 36
- [40] The University of Auckland Pretzel. BlokIDE. URL <https://pretzel.ece.auckland.ac.nz/#!research?project=iec61499>. 36
- [41] OOONEIDA-FBench. URL <https://sourceforge.net/projects/oooneida-fbench/>. 38
- [42] ICARU_FB. URL <https://sourceforge.net/projects/icarufb/>. 38
- [43] GASR-FBE. URL <https://sourceforge.net/projects/gasrfbe/>. 38
- [44] ISaGRAF Technology. URL https://www.rockwellautomation.com/en_US/detail.page?docid=209076c017d6dd586c895e9e3a4856e4. 38
- [45] nxtSTUDIO. URL <https://www.nxtcontrol.com/en/engineering/>. 38
- [46] Function Block Service Runtime. URL <http://www.iec61499.cn/>. 38
- [47] AutomationML Consortium. Whitepaper AutomationML Part 2 - Role class libraries. Technical report, 2014. 40

- [48] C. Masson. Why the OPC UA Standard – and What’s Next?, Apr 2018. URL <https://blogs.microsoft.com/iot/2018/04/11/why-the-opc-ua-standard-and-whats-next/>. 47
- [49] D. J. X. Ng. IMPACT Line Scenarios, 2018. URL <https://youtu.be/zQjWrg3-9RM>. 50