DIAGNOSTIC BELIEF-DESIRE-INTENTION AGENTS FOR DISTRIBUTED IEC 61499 FAULT DIAGNOSIS

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Supervisors

Associate Professor Roopak Sinha,

Professor Stephen G. MacDonell, and Dr Jennifer Gibb

July, 2021

By

Barry Dowdeswell

School of Engineering, Computer and Mathematical Sciences

Intellectual Property Rights

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

Signature of candidate

Acknowledgements

"If one think it worth his while to write my life, I will give you a criterion by which you may judge of its correctness. If he gives me credit for being a plodder, he will describe me justly. Anything beyond this will be too much. I can plod. I can persevere in any definite pursuit. To this, I owe everything."

William Carey, Missionary (1761-1834)

Thank you...

Those two simple, unembellished words summarise everything I want to be able to say to each of you who have supported my own *plodding* on this nine-year journey. It has taken me from being a returning graduate learning again surrounded by undergraduates, through my Master's research, and finally to submitting this Doctoral thesis. Thank you so much for encouraging my plodding, especially when the terrain became hard.

To my precious wife Jane; thank you. Your love and patience sustains me, your encouragement never ceases. I know that you never stopped believing that this could all be brought to completion if we just kept going. Above all, our shared faith in our Father helps all this make sense. We worked side-by-side for so many years and now here we are, each in a stimulating, yet different, career. Thank you for being brave enough to leave it all behind and start a whole new journey together with me. I love you.

To my children Michaela, Philip, and Ben and to their awesome partners Luke, Destiny, and Emily; thank you. Each of you has different gifts that you have graciously shared with me, from words of encouragement, hugs, proof-reading or just being my study-buddies when we were both facing exam time. You are all wonderful and I adore each of you.

To my primary supervisor Associate Professor Roopak Sinha; thank you. You graciously took me under your wing, first as a wet-behind-the-ears research assistant, then as your Master's student, and finally as a PhD candidate. Thank you for your wisdom, constant kindness, encouragement, and guidance. I shall always treasure this time we worked together solving these "wicked problems" that lie at the heart of our shared research interests.

To my second supervisor Professor Stephen MacDonell; thank you. You were always ready with a thoughtful timely word, spoken patiently, that cut to the heart of the research problems we were wrestling with. So often, your wisdom turned the problem right around so we could look at it from a whole new perspective. Thank you so much for all you so graciously shared with me during this time.

To my third supervisor Dr Jennifer Gibb; thank you. Since 2004, we have shared so many coffees together, wrestling through both business and academic life with all of its challenges. You bring your unique perspective to my supervisory team and I treasure our discussions about life and this strange academic world we both inhabit. Thank you for being my constant friend, guiding me though this jungle in your most gracious way, especially whenever I got a bit lost.

Badger (Barry) Dowdeswell, July 2021.

Abstract

Humans demonstrate ingenious ways of solving engineering problems, but their approaches to finding faults are often laborious and time-intensive. Their manual techniques do not scale well, especially when the systems are large, complicated, or widely-distributed geographically. Multi-Agent Systems (MAS) are one way to address these concerns, creating software entities called *agents* that can automate fault-finding tasks. Agents operate co-operatively and semi-autonomously, working in ways that are similar to how teams of humans might tackle a problem.

This research centres on fault identification and diagnosis for complex automation and control systems known as Industrial Cyber-Physical Systems (ICPSs). The IEC 61499 Function Block Standard is a reference software architecture that is well-suited to creating an ICPS. However, while current function block design tools provide limited design-time fault-finding capabilities, they cannot be used to facilitate thorough diagnostic investigations when problems occur later in the life cycle of an ICPS. Larger, safety-critical systems require a level of fault diagnosis that demands more comprehensive approaches; design and development-time testing techniques are not sufficient.

The thesis proposes that it is both feasible and worthwhile to implement fault diagnostics as part of a Model-Driven Development methodology for ICPS that are built from IEC 61499 Function Blocks. Furthermore, it demonstrates how the design and implementation of diagnostic functionality can become a core part of the engineering processes. It demonstrates how to manage the inevitable faults that ICPSs suffer. Diagnostics can and should be a key component of a mature design and development methodology. The research demonstrates how fault identification and management capabilities can be created in-parallel with the components and

subsystems that will ultimately realise the stakeholders' vision for their application.

A Scoping Survey Literature Review was performed to uncover mature fault diagnostic approaches used in the aerospace, automotive and manufacturing domains. These sectors employ ICPSs in safety-critical environments. Hence it is expected that it is in these environments that the most well-refined and mature fault management approaches will have evolved. Following this survey, a Design Science Research plan was created to facilitate multiple, incremental experimental phases. The primary phase produced a Software Architecture Document for a Fault Diagnosis Engine (hereafter referred to as "the engine") by applying an Attribute-Driven Design (ADD) methodology. This process identified which quality attributes would best address the stakeholder's requirements.

This led to the creation of novel software-in-loop telemetry capture devices called *Diagnostic Points*. These operate inside the FORTE IEC 61499-compatible function block application runtime, passing telemetry to and from the GORITE Multi-Agent System framework. By implementing agents who work co-operatively to capture and interpret evidence of system misbehaviour, diagnostic techniques were evaluated to determine which approaches most accurately identify and diagnose faults.

Each subsequent Design Science phase explored and implemented part of the engine proposed in the Software Architecture Document. The scope and evaluation criteria for each Design Science phase are defined by and reference back to the relevant sections of this document. The on-going Architectural Trade-off Analysis Method (ATAM) evaluations of each phase were driven by a set of pre-defined scenarios which were based on typical real-world faults. These were uncovered in both the scoping survey and during the subsequent research. Three journal articles and two conference papers were published during the research to refine the work, informed by peer-reviewed feedback.

The completed engine was evaluated during the design and development of two example function block applications. Following a V-Model methodology, the agents performed diagnostic testing at appropriate development and implementation stages. The results from these projects led to further refinement of both the engine and the Model-Driven Development with Diagnostics

methodology that the thesis proposes.

This research has shown that the engine has reached a level of maturity that is close to the NASA Technology Readiness Level 5 (TRL 5). This level designates technologies that have been demonstrated to work in a relevant or representative physical environment beyond the laboratory. The aspects of the technology that contribute towards meeting the TRL 5 criteria are explored in greater depth in the conclusions chapter.

Publications from this research

Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems.	Dowdeswell B., Sinha R., MacDonell S.G. (2021). Published in the MDPI journal *Future Internet* in their Special Issue *Modern Trends in Multi-Agent Systems* in July 2021. doi: 10.3390/fi13080190
Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems.	Dowdeswell B., Sinha R., MacDonell S.G. (2020). In *IECON 2020 46th International Conference of the IEEE Industrial Electronics Society.* Publication date: October 2020. doi: 10.1109/IECON43393.2020.9254620
Employing Agent Beliefs during Fault Diagnosis for IEC 61499 Industrial Cyber-Physical Systems.	Dowdeswell B., Sinha R., Jarvis D., Jarvis J,. MacDonell S.G. (2020). In *IECON 2020 46th International Conference of the IEEE Industrial Electronics Society*. Publication date: October 2020. doi: 10.1109/IECON43393.2020.9254877
Finding Faults: a scoping study of Fault Diagnostics for Industrial Cyber-Physical Systems.	Dowdeswell B., Sinha R., MacDonell S.G. In Elsevier *Journal of Systems and Software*, Volume 168. Publication date: May 2020. doi: 0.1016/j.jss.2020.110638
TORUS: Scalable Requirements Traceability for Large-Scale Cyber-Physical Systems.	Sinha R., Dowdeswell, B., Zhabelova, G., Vyatkin, V (2018). In ACM *Transactions on Cyber-Physical Systems*, Vol. 3, No. 2, Article 15. Publication date: October 2018. doi: 10.1145/3203208

Contents

List of Tables

List of Figures

Glossary of Terms

Agent	An intelligent agent (IA) refers to an autonomous entity which acts, directing its activity towards achieving goals. Situated Agents operate in a physical environment, basing their activities on decisions made by observing through sensors and acting using actuators. Intelligent agents may also learn or use knowledge to achieve their goals.
API	Application Programming Interfaces define the interactions that are possible between software applications and their hardware.
Architecture	The software architecture of a system defines the structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them (Bass, Clements & Kazman, 2013).
Function Block	The fundamental building block used to create IEC 61499 Function Block Applications.
IEC 61499	The international standard IEC 61499 (IEC, 2013a) defines the function block architecture for industrial process measurement and control systems. It was initially published in 2005. The specification of IEC 61499 defines a generic model for distributed control systems and is based on the IEC 61131 standard.
IED	Intelligent Electronic Device. IEDs are smart electricity grid protection devices whose designation numbers are specified in the IEEE/ANSI standard C37.2 (IEEE, 2008).
SAD	Software Architecture Document. A document template developed by the Carnegie Mellon Software Engineering Institute (SEI) that can be used to document a software architecture.
View	A representation of a whole system from the perspective of a related set of concerns standardized in ISO 42010 (ISO, 2011b). A View conforms to a defining Viewpoint.

View Packet	The smallest unit of architectural documentation that could usefully be given to a stakeholder. The documentation of a View is composed of one or more View Packets.
Viewpoint	A specification of the conventions for constructing and using a View. A pattern or template from which to develop individual views by establishing the purposes and audience for a View, and the techniques for its creation and analysis standardized in ISO 42010 (ISO, 2011b). Identifies the set of concerns to be addressed, and the modelling techniques, evaluation techniques and consistency checking techniques used by any conforming View.
Views and Beyond	The Carnegie Mellon Software Engineering Institute's *Views and Beyond* method for documenting software architectures, as described in Bass and Clements (Bass et al., 2013)

Chapter 1

Introduction

"What I cannot create, I do not understand."
"Know how to solve every problem that has been solved."

The final blackboard comments of Richard Feynman (1988)

Those final thoughts of Richard Feynman were written on his blackboard the day he left Caltech shortly before he died in 1988. The first is an observation, while the second is an imperative. They succinctly capture two of the core research principles that guided his life in science (Gribbin & Gribbin, 1998). Both of those statements have had a profound influence on the research and experimentation that is presented in this thesis.

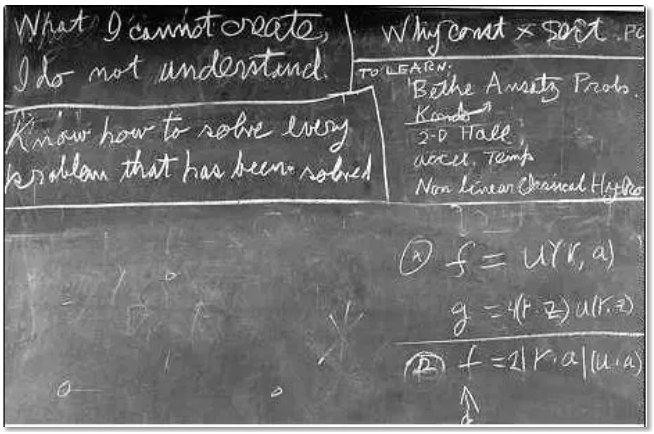

Figure 1.1: Richard Feynman's last blackboard at Caltech (Feynman, 1988)

Embedded control systems, augmented with communications and sophisticated sensors, have led to the development of powerful new hardware and software-driven mechanisms known as *Cyber-Physical Systems* (CPSs). CPSs control their electro-mechanical parts by using on-board computer processors and *transducers*, which are the sensors and mechanical actuators

that enable them to sense and interact with their physical environments. This blending of their "cyber" software aspects and their "physical" mechanical and environmental aspects extends the capabilities of earlier generations of embedded industrial control equipment. It is at this intersection, not the union, between the cyber and the physical that we must seek to understand these entities more deeply (Lee & Seshia, 2016). When we scale a CPS to operate in large, industrial-scale settings, we describe them as Industrial Cyber-Physical Systems (ICPSs) (Karnouskos, Leitao, Ribeiro & Colombo, 2020). There are a number of reference software architectures that can be used to create the cyber computing and control parts of an ICPS. ICPSs created with the formal IEC 61499 Function Block Reference Architecture (IEC, 2003) are the primary focus of this research. . The IEC 61499 architecture has seen a wide uptake in smart electricity grids (NOJA, 2015; Zhabelova, Yang, Vyatkin, Etherden & Christoffersson, 2016) and in manufacturing (García, Irisarri, Pérez, Estévez & Marcos, 2017; Hegny, Hummer, Zoitl, Koppensteiner & Merdan, 2008)

However, like many complex systems, ICPSs are prone to failures. When that happens in safety-critical environments, the results can be catastrophic and sometimes, life-threatening (Feiler, Gluch & Mcgregor, 2016; McGregor, Gluch & Feiler, 2017). Feynman's imperative is a call to learn all that one can about the underlying technology of such entities, verifying the correctness of our designs with experimental evidence before letting them loose on the world. The Scoping Study Literature Review presented in Chapter 2 sought to do that by examining how faults are addressed across three sectors where safety and reliability are of paramount importance. It examines the current gaps and profiles fault identification, diagnosis and management in the aerospace, automotive and industrial manufacturing domains, examining 127 papers written by both academics and industry practitioners. The studies cover the period from 2004 to 2019, spanning the time when ICPSs were first gaining traction in these sectors. The fault identification techniques found ranged from Model-Based Diagnosis through to Data-Driven Artificial Intelligence (AI) with a significant proportion of hybrid methods that blend aspects of both. Each technique was categorised by applying a set of ontology classes designed to classify common characteristics of the techniques (Uschold, Gruninger et al., 1996;

Noy, McGuinness et al., 2001). Standardized Software Quality Attributes were used to further classify the approaches. Finally, the maturity of each approach was evaluated using the NASA Technology Readiness Assessment scale (Mankins, 2009; Héder, 2017). This produced a deeper analysis of the most mature techniques, which are discussed in the conclusions section of the review.

While Test-Driven Development (Hametner, Hegny & Zoitl, 2014; Y. Zhang, 2004) is now a widely-used and mature software engineering discipline, current IEC 61499 Integrated Development Environments (IDEs) provide only limited fault identification capabilities. They do not provide the comprehensive diagnostic resources and functionality needed to allow engineers to exercise both individual and groups of connected function blocks during development. While some of the IDEs allow single function blocks to be exercised, sets of connected blocks cannot be diagnosed together. There is a need for engineers to be able to quickly build semi-automated fault identification capabilities that can be reused throughout the entire life-cycle of the application. This research proposes that fault diagnostics should be an early-stage consideration in the design process so that systems become *Diagnosable-By-Design*. The approach to fault diagnosis described here addresses that need by proposing a Model-Driven Development with Diagnostics methodology to support engineers while they are designing IEC 61499 function blocks to construct their ICPS (Dowdeswell, Sinha & MacDonell, 2020a).

The existing literature uncovered during the Scoping Study also outlined the concept of a *Fault Diagnostic Engine*. Fault diagnostic, management and protection engines are self-contained systems that interact with an ICPS to monitor and intervene in fault situations. While they have a high degree of visibility of the ICPS, they also maintain a degree of separation to ensure that they themselves are not compromised by faults occurring within the ICPS. Benovitz profiled the Fault Protection Engine used on NASA's Mars rover, *Curiosity* (Benowitz, 2014). Airbus employs a similar system that blends a range of Model-Based and Data-Driven AI approaches to monitor the flight control surfaces on their A380 aircraft (Zolghadri et al., 2015). Like the Curiosity fault protection and management engine, the Airbus system relies on monitors distributed throughout the ICPS that capture telemetry from sub-systems. These monitors enable

the fault engines to become aware of problems as they arise. Aerospace systems have also employed software agents to implement the decision logic needed in their fault diagnostic engines to effectively recognise and manage faults. Agents demonstrate ways of implementing autonomous behaviour and decision making that makes them ideal for managing the operations of ICPSs such as these. Studies include Georgeff and Ingrand (1990), who detail the multi-agent fault diagnostic engine that was designed for the NASA Space Shuttle.

The findings from the existing literature helped to shape the Design Science research methodology that is outlined in Chapter 3. Throughout this research journey, there were multiple opportunities to build prototypes and learn new IEC 61499 development systems such as 4diac (Strasser et al., 2008). 4diac is the mainstay of IEC 61499 function block research and application development performed in universities around the world. However, the aim of Design Science must always be to develop theory (Kuechler & Vaishnavi, 2008). It should never become just coding for the sake of coding, a criticism that is sometimes levelled at Design Science research. Rather, our application of the Design Science methodology guided the creation of iterative prototypes that were evaluated at the end of each phase by applying pre-defined metrics. In each iteration, the evaluation asked how well the artefacts had furthered our understanding of the aspects of fault diagnosis that were being investigated. The primary sources for those evaluation metrics were the Software Architecture Document discussed in Chapter 4. This describes the architectural design proposed for the engine, presenting each aspect in a *View* using appropriate diagrams. Each View provides a different perspective to better explain aspects of the application to its stakeholders, addressing their particular concerns and interests. The document demonstrates how SysML diagrams (Delligatti, 2013) provide a succinct, formal description of both the structure and operation of the fault diagnostic engine. Within each View, Quality Attributes for each Functional Requirement help to define empirical performance measures to allow the artefact to be evaluated against its stated goals.

1.1 Framing the research questions

All of the current Integrated Development Environments (IDEs) for function block application development have limited runtime debugging and testing capabilities. Hence, the availability of a fault diagnostic system that could work alongside application engineers would offer significant benefits during many of the design, development, testing and implementation phases. Distributed ICPS applications are too large to test manually, and elusive or intermittent faults can often only be detected during longer post-development burn-in diagnostic trials (Shin, Nejati, Sabetzadeh, Briand & Zimmer, 2018; Zhou, Gou, Huang & Yang, 2018). Therefore, this research can be framed in terms of five primary questions:

RQ1: *What are the current diagnostic approaches from other sectors that are being used or could be used for finding faults in distributed applications built with IEC 61499 function blocks?*

Diagnosis is performed by determining which component or software algorithm of a system is not performing nominally by comparing its behavior to pre-defined expectations. How is that done currently and what approaches could be explored that are not presently being used for IEC 61499 systems?

RQ2: *Which diagnostic needs for IEC 61499 systems cannot be met by current fault-finding approaches?*

Faults in IEC 61499 systems arise from errors in software algorithms as well as timing issues related to the exchange of data between distributed components. The failure of mechanical or electronic peripherals can result in faulty data about the environment being provided to the function block software, causing it to make incorrect control decisions. What aspects of these and other scenarios are not addressed well by current diagnostic approaches?

RQ3: *To what extent can Belief-Desire-Intention Multi-Agent Systems be used as a framework to perform diagnostics for distributed IEC 61499 systems identified in RQ1?*

Multi-Agent Systems have been proposed as a promising approach to implement diagnostics for ICPSs. How would agents address the needs identified in RQ1 and RQ2 better than the approaches profiled in the current literature? In an ideal scenario, fault-finding would be a

co-operative activity between the fault diagnostic system and the system under examination. What characteristics of the IEC 61499 architecture can we exploit to better expose information about components and processes that could be used to make fault-finding and rectification more quantifiable?

RQ4: *How do we architect and develop a Fault Diagnostic Engine for IEC 61499-compliant applications to meet the needs identified in RQ1, RQ2 and RQ3?*

What is the most appropriate way to identify the needs, features, and qualities of the fault diagnostic system? What standards have been adopted by practitioners and which techniques have proven to be effective to document the architecture of the system and evaluate the design choices made?

RQ5: *How can we evaluate the effectiveness of the solution proposed in RQ4 in meeting the diagnostic needs of industrial-scale IEC 61499-compliant systems?*

What scenarios can be created to test and evaluate the architectural design and the implementation of the fault diagnostic system?

1.2 Industrial Cyber-Physical System case studies

Two ICPS exemplar systems were created and evaluated as part of this research. In Chapter 9, the Model-Driven Development with Diagnostics methodology was evaluated by the designing, building, and then diagnosing faults found in an IEC 61499-based Heating, Ventilation and Air-Conditioning (HVAC) room controller. A second case study presented in Chapter 10 evaluated fault diagnostic techniques for Smart Electricity Grid protection devices built using IEC 61499. This chapter also profiles the way teams of agents work in more detail.

The systems developed during this research provided an comprehensive test-bed from which to evaluate fault-finding and diagnostic techniques. HVAC systems are complex control mechanisms that provide climate control for buildings. Their scale varies from small dwelling systems through to installations that manage multiple floors within large complexes. Their design qualifies them as being an ICPS since they rely on telemetry from sensors, electro-mechanical

control of heating and cooling equipment, and communications with external systems.

A separate general-purpose simulation tool called *simbIoTe* (**sim**ulator for **b**uilding **IoT** **e**nvironments) was developed to emulate the physical environment of a building. Figure 1.2 shows the HVAC system while the simbIoTe environment simulation is running. By emulating the physical characteristics of a building, simbIoTe provides temperature readings that the IEC 61499 HVAC room controller can process. It also interacts with an emulated HVAC heating and cooling control plant sub-system to accept amounts of thermal energy to change the state of the simulated environment. The IEC 61499 standard does not provide Human-Machine Interfaces (HMI) capabilities in its architecture, so simbIoTe also emulates the display panel and control switches needed. Co-simulation tools such as simbIoTe enable HMIs to be developed quickly before the real ICPS hardware is available. This allows sensors and actuators to be simulated realistically so that the function blocks can interact with them. The architecture of simbIoTe is discussed further in Appendix B.

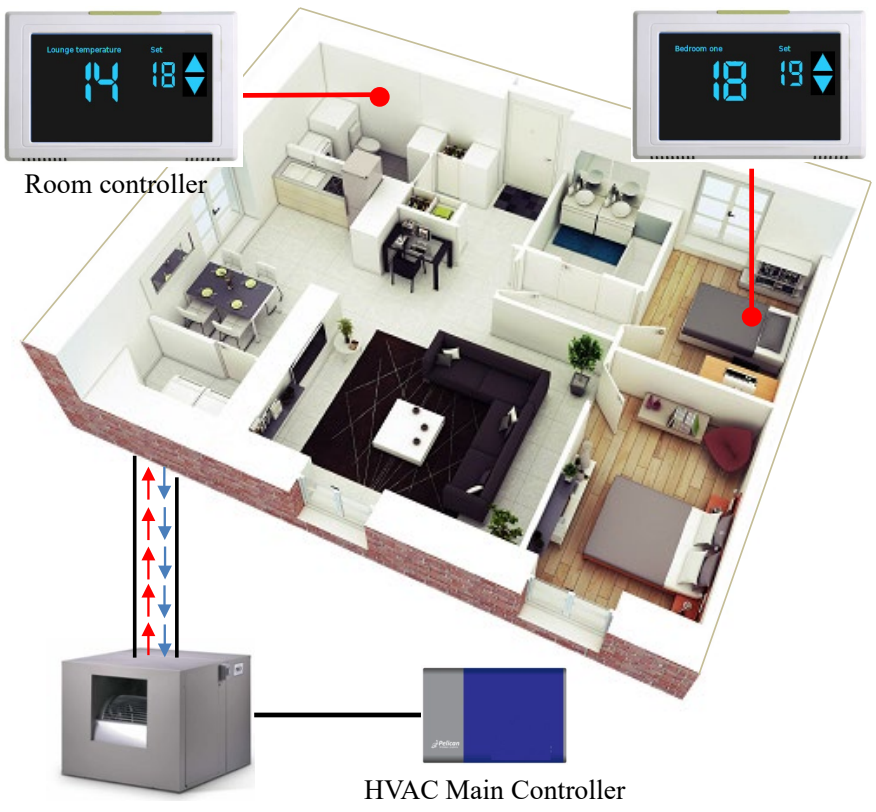

Figure 1.2: HVAC room controllers in a *simbIoTe* emulated environment.

The second case study built upon our earlier research into electric power grid protection. Smart Grid protection devices build upon a rich legacy of electro-mechanical devices that date back to the first Overcurrent Protection devices that came into service in 1901 (Bo, Lin, Wang, Yi & Zhou, 2016). The IEC 61499 function block architecture is gaining traction in this sector since it helps to facilitate the precise timing and control needed to manage faults in safety-critical power reticulation systems (NOJA, 2015). Two different types of protection devices were designed and built. A team of agents was created to investigate how well the agents collaborate and how effectively the fault diagnostic engine could uncover faults in both the design and operation of the devices.

1.3 Contributions from this research

Feynman's final imperative supports his first observation. Novel contributions can only be identified by first knowing what is already known. The fault diagnostic engine is by design a *system-of-systems* (Samad, Parisini & Annaswamy, 2011). It is both composed of and interacts with multiple discrete systems and applications in complex ways. The choice of an agent-based approach was based on the findings of the Scoping Study Literature Review. The agents rely on a novel function block designed during this research to both gather telemetry and interact dynamically with functions blocks operating within the target application. Known as a *Diagnostic Point* or DP, this custom-designed low-latency function block operates natively in the IEC 61499 runtime environment. The evaluations presented later show that the DPs impose a minimal overhead on the function block application itself. This is an important quality attribute since faults related to timing issues could be masked by telemetry-gathering activities that compromise the normal operation of the application. Unlike previous approaches, this technique illustrates how agents can dynamically monitor systems. They do this after reconfiguring the control layer from the agents' execution layer and then deploying light-weight DPs, which are *software-in-loop* points-of-presence. This represents a significant enhancement of the interaction scheme employed in Kalachev et al. (2018), Jarvis et al. (2018), and Christensen (2017). In their

approaches, execution layer agents only initiated function block controlled actions.

The work also seeks to encourage engineers to adopt a Model-Driven Development with Diagnostics approach that leads to *diagnosable-by-design* ICPSs. Engineers interacts with the system in two ways:

- Diagnostic packages are developed in-parallel with the individual function blocks as part of the Model-Driven Development with Diagnostics methodology. Engineers are encouraged to think about faults early-on and design for them, making their components inherently diagnosable-by-design.
- The engineer receives back a report of diagnostic fault information collated by the Fault Diagnostic Engine based on the diagnosis returned by the agents. An example of the report is shown in Figure 10.15.

The GORITE Multi-Agent framework was enhanced by creating a number of novel belief structures for the agents (Jarvis, Jarvis, Rönnquist & Jain, 2013). This provides them with domain-specific knowledge of the IEC 61499 architecture and skills that enable them to interact with the DPs. Agents use this knowledge to dynamically wire each DP into the application before a fault-finding session commences. The research implements *Diagnostic Packages* that allow engineers to specify information needed by the agents to understand the design intent for each function block and how to exercise each block dynamically to uncover faults. A third belief structure enables the agents to capture, organise and consider their findings while hunting for faults. The final diagnosis that the agents present is constructed from these beliefs to report their findings back to the engineers.

The IEC 61499 standard encourages the creation of re-usable function block components. Applying the Diagnosable-by-Design approach proposed here augments those components with long-term, field-proven diagnostic packages that accompany the function blocks through their entire life-cycle. As blocks are re-used and refined alongside these resources, applications become more resilient, robust, and more cost-effective to develop and support.

In the concluding Chapter 11, the work is considered against the ISO-based NASA Technology Readiness Level (TRL) scale (ISO/BSS, 2019). This widely-used ISO standard evaluates

the maturity of a technology and assigns it a TRL scale level number from one to nine. The chapter considers how close the fault diagnostic engine and our methodology come to achieving TRL 5. This level identifies components and technologies that have been validated in a relevant production environment. Technologies that reach this level are half-way up the scale that leads to TRL 9, where technologies are deemed to be flight-proven and can be deployed in mission-critical space environments. Our evaluation uncovered rich opportunities for future work beyond the present research that would contribute to the fields of both ICPS fault diagnosis and Multi-Agent Systems.

Chapter 2

Reviewing the Current Literature

"It ain't so much what you don't know that gets you into trouble, it's what you know for sure that just ain't so."

Attributed to Mark Twain (Samuel Langhorne Clemens) 1835-1910

This Literature Review identifies, categorises, and analyses fault identification strategies for ICPS across the aerospace, automotive, and industrial control domains. Parts of this chapter were adapted for publication in the Journal of Systems and Software article *Finding Faults: A scoping study of fault diagnostics for Industrial Cyber-Physical Systems* (Dowdeswell, Sinha & MacDonell, 2020b).

The aerospace, automotive, and industrial control domains are of particular interest since the ICPS they rely on must operate faultlessly for extended periods of time, often in close proximity to humans (Bolbot, Theotokatos, Bujorianu, Boulougouris & Vassalos, 2018). The ICPS they employ feature high levels of integration between their computational cyber elements and the sensors that provide the information that all operational decisions are made on. For example, ICPS in automobiles now determine the position of highway lane markings accurately, extract information from road signs, and determine the relative positions of adjacent vehicles. Hence, it is reasonable to expect that fault identification and management in these domains will have reached a level of maturity as techniques are refined over multiple generations of ICPS.

The study also examined the similarities and differences in fault diagnostic approaches that have emerged in these three safety-critical sectors over the period surveyed. However, it was decided not to include the medical sector in this study. Medical ICPS have distinctive electronic and biological characteristics, regulatory requirements, and a scale that is worthy of a separate study. Cyber-physical devices in the Consumer Electronics sector were also excluded from the study. They drive a large and expanding share of the market, however they are often less complex than the ICPS in our chosen sectors. The tasks they manage are not usually safety-critical.

The goal was to address RQ2 by presenting a snapshot of fault diagnosis as it is practised today in the three transportation and industrial sectors. By exploring the range of best-practices adopted in these domains, this study seeks to uncover strategies that can be applied to the design of the fault diagnostic engine created in this research.

2.1 Defining the literature review protocol

A Scoping Study was used to map the key approaches that underpin fault diagnosis in these sectors, profiling the sources of both theory and case studies available from practitioners (Cacchione, 2016; Arksey & O'Malley, 2005). By applying a systematic classification to each technique encountered, the study was able to provide an estimate of the relative level of maturity of each approach, highlighting those which are being applied successfully in real-world environments.

Scoping studies are one method of rapidly mapping the key concepts that appear within a research area (Levac, Colquhoun & O'Brien, 2010). Often smaller in scope than full Systematic Literature Reviews or Mapping Studies, Scoping Studies allow the breadth of coverage and the depth of the information extracted to be tailored to an appropriate depth to address research questions appropriately (Wohlin, 2014; Mays, Roberts, Popay et al., 2001). Arksey and O'Malley (2005) and Antman et al. (1992) both explain how Scoping Studies are an appropriate way to quickly capture and present the available information and highlight possible gaps. Our protocol follows the four steps of framing research questions, identifying relevant studies, analysis, and the presentation of the results that were outlined by Arksey and O'Malley (2005).

Cacchione (2016) and Levac (2010) comment that the scope and rigour of scoping studies is evolving. They recommended clarifying and linking the purpose of the research questions while balancing the breadth of the scoping protocol to better fit the questions.

2.1.1 Step One: Framing our research questions

RQ1 from Section 1.1 asks what diagnostic approaches are currently being used in other sectors that may be applicable to IEC 61499 applications. The research questions defined for the scoping study expand RQ1 into three sub-questions:

RQ1.1: *What are the most common and widely-used fault identification and diagnosis techniques employed in ICPS in the aerospace, automotive and industrial control domains?*

Is there a different emphasis and intensity of research in the needs being articulated in each sector? Are these differences due to the scale, complexity, risk profile and safety-critical aspects of the systems they are controlling? What techniques might be applicable to IEC 61499 applications?

RQ1.2: *What relative levels of maturity have the techniques identified in RQ1.1 achieved when assessed using a systematic scale that has been accepted as applicable to these domains?*

What are the empirical scales that measure the maturity of a technique that are accepted or widely-recognised in these sectors? In what ways are they used and has there been a cross-pollination of the successful evaluation techniques into the other sectors? What techniques could be adopted to assess techniques for use in the fault diagnostic engine?

RQ1.3: *What research gaps and challenges in ICPS fault identification and diagnosis are being highlighted in the literature surveyed to answer RQ1.1?*

What are the most challenging and promising areas for further research? Since this study covers multiple industry sectors with potentially different needs, the focus and spread of the challenges is an important perspective. Are there some pressing concerns that are repeatedly

mentioned in the studies? What effects might these issues have on our diagnostic approaches?

Scoping Studies have also been shown to be effective where the researchers do not have a single, tightly-focused research question (Munn et al., 2018). These research sub-questions were therefore designed to identify, highlight and categorize practical fault recognition and diagnosis techniques that have been found to be effective both in laboratory studies and in the field. The study therefore examines multiple yet similar sectors with potentially differing needs. Understanding the focus and spread of the challenges and how they are being addressed is important to practitioners who are designing their own ICPS. It is also relevant to the architects of the fault diagnostic engine and the fault-finding techniques that are the focus of this research.

2.1.2 Step Two: Identification of relevant studies

The initial survey range was the fifteen-year period from 2004 to 2019. This starting point was chosen since it coincides with the emergence of the term *Cyber-Physical System*. The first use of this term can be traced to the National Science Foundation meetings in 2001 that discussed how to network current embedded control systems (Council, 2001; Gill, 2008). Lee (2006) later highlighted the implications of these discussions about connecting discrete embedded systems via networks. Prior to this, Wiener's (1948) earlier pioneering work on cybernetics informed much of the thinking on control systems theory, arguably setting the agenda for later CPS and ICPS research.

Scopus was used to perform a primary search for papers that included the terms "cyber-physical", "aerospace", "aircraft", "automotive", "industrial" and "manufacturing":

TITLE-ABS-KEY(("automot*" OR "aerospace*" OR "avion*" OR "manufac*") AND ("cyber*" OR "embed*") AND ("diagno*" OR "progno*" OR "forensic*" OR "predict*")) AND PUBYEAR > 2004 AND (LIMIT-TO (DOCTYPE,"ar ") OR LIMIT-TO (DOCTYPE,"cp ")) AND (LIMIT-TO (SUBJAREA,"ENGI ") OR LIMIT-TO (SUBJAREA,"COMP "))

Besides high-quality single-topic papers, there was a desire to locate Systematic Mapping Studies (Petersen, Feldt, Mujtaba & Mattsson, 2008) and Systematic Literature Reviews (Kitchenham et al., 2010) that focused deeply on one industry sector. The initial results suggested that differing regulatory and safety certification regimes for automotive and aerospace

systems have led to a range of implementation approaches. There are also significant architectural differences. Automotive control systems are primarily built by integrating third-party supplier subsystems that lead to common product lines. In contrast, aerospace systems are typically more heterogeneous, custom-built solutions. There was also an interest in the degree of collaboration between industry and academia to see how this might have influenced their research focus and outcomes. Industry partnerships often keep academic research grounded and more relevant to the current industry needs (Garousi, Petersen & Ozkan, 2016).

2.1.3 Step Three: Study selection and classification

From an initial pool of 511 candidate papers returned by queries on Scopus, a pilot study was performed on thirty of these papers. By analysing only the titles and briefly reading the abstracts, it became apparent that "*diagnostics*" and "*cyber-physical systems*" were popular buzz-words that many researchers outside of the field like to use to index-tag their papers. This meant that the searches returned a lot of noise, such as irrelevant papers relating to laboratory diagnosis using computers that had found their way into the engineering and computer science fields that Scopus was indexing. Some of the papers that included the index tag for "*CPS*" only discussed the use of CPS indirectly in their main text sections. These papers were discarded in the subsequent filtering stages. Sample papers were chosen for deeper background analysis primarily because they contained well-written explanations of fault identification and diagnosis techniques that provided valuable background information.

Table 2.1 lists an initial set of fault identification or diagnosis approach classifications that identified both broad conceptual differences and a list of specific techniques applicable to those approaches. RQ1.1 asks about the nature of fault identification and diagnostics within the chosen domains. The broadest primary classifications that emerged divided the approaches into three high-level categories that helped to delineate the research activity. Table 2.1 identifies some of the characteristics of the techniques encountered. Approaches can be grouped into the three broad categories of Physics-Based Model-Driven diagnostics, Data-Driven Model-Free Artificial Intelligence (AI) techniques, and Knowledge-Based graph approaches. Hybrid techniques that

Table 2.1: Fault Identification and Diagnosis Classification Categories.

Category	**Example Sub-Categories**
Physics-Based Modeling approaches.	Kalman Filters, Markov Models, Fault Trees, Stoichiometric processes, Model Validation/Invalidation, Monitor-based oracles.
Data-Driven AI Machine Learning approaches.	Artificial Neural Networks, Machine Learning, Fuzzy Logic, k-Nearest Neighbour, Big Data/Data Mining.
Knowledge-based approaches.	Bayesian Decision Theory, Binary Trees, Petri nets, Network Message Analysis, Expert Systems.

blend aspects of these approaches were also uncovered. The similarities and differences between these broad classes are profiled in more detail in Section 2.3.2.

The publication sources included peer-reviewed journal papers, conference papers, and open-access journals. Outside of the academic databases, technical publications and position papers written by industry-based authors with current, practical experience in their field were sought. Examples include automotive-industry papers from SAE International (https://www.sae.org) and aerospace papers written by NASA researchers or their industry partners, such as Lockheed and Boeing. While the papers published by non-academic sources such as SAE were not necessarily peer-reviewed, they often contained detailed results from specific case studies. Industry magazine articles often helped to explain issues or challenges from a specific researcher or industry representative's perspective. Arksey and O'Malley (2005) stress the importance of including this "grey matter" during the second step of their recommended scoping protocol.

The minimum inclusion criterion for the study required it to present and explain the fault identification or diagnosis approach that was being applied. Papers were sought that included case studies demonstrating the effectiveness of techniques. Many papers were excluded because they only mentioned "faults" or "diagnosis" as an aspect of the nature of ICPSs without those terms being core to what they were presenting.

2.1.4 Step Four: Analysing and Presenting the Data

During the first phase of the analysis, the classification categories enabled us to perform a thematic analysis (Cruzes & Dyba, 2011). Each of our categories and sub-categories represents a technique or approach used or proposed by a practitioner as a way of identifying, diagnosing or rectifying a fault (Castleberry & Nolen, 2018). The analysis also examined where diagnostic research is focused in each sector. The findings are presented in Figures 2.9, 2.10 and 2.11.

Further refinements to the search criteria helped to remove some of these extraneous papers automatically. However it was found that this would not identify all irrelevant instances. Hence, the titles were read manually to remove obvious outliers. Abstracts were then read of papers whose titles initially appeared to meet the criteria. The target date range for the main study was then reduced to be the ten-year period from 2008 to 2018 with provision for some snowball papers that extended outside these boundaries. Broad classification categories analysed the papers by source, industry sector and country while more fine-grained analysis was classified using the ontology classes that appear in the published dataset. This data set is available from Mendeley Data at doi 10.17632/wg9shy9rsm.1. The publication sources included peer-reviewed journal articles and conference papers. Technical publications and position papers written by industry-based authors with recent, practical experience in their field were sought that were published outside of the academic databases.

Citation counts were kept for all papers as a broad indicator of relevance. However, such metrics should be treated with caution since newer papers take time to garner citations. Papers were also classified by their country of origin. While this was deemed to be less relevant to our research questions, the information is still preserved in our analysis results. Academic institution affiliations were noted as well as recording in-context the names of the industry collaborators they worked with. Typical collaborators encountered included NASA, ASEA Brown Boveri (ABB) and General Motors (GM).

2.1.5 Paper content classification keywords

The specific diagnostic approaches that a paper presented were classified by assigning a boolean Y or N if they matched a specified criterion. Examples of our categories include tracking whether a manufacturing paper identified their work with issues related to the German Industry 4.0 initiative (J. Lee, Bagheri & Kao, 2015) or if the paper also discussed predictive diagnostics. Discussions about Neural Network-based Artificial Intelligence approaches were noted as well as those that promoted purely Model-based techniques. Hybrid approaches that adopted both Model-based and Model-free techniques were tracked by setting multiple classification indicators either true or false.

2.1.6 Assessing the quality and maturity of techniques

Scoping Studies do not usually attempt to assess the quality of the studies uncovered (Cacchione, 2016). However, two techniques were applied to the data once it had been categorised:

2.1.6.1 Applying Quality Metrics

In addition to identifying features and characteristics of fault diagnosis, the study looked to see if researchers considered the quality dimensions of their approaches. The IEC/ISO 25010 standard (ISO/IEC, 2011a) defines quality metrics that were used to classify the quality aspects of the fault diagnostic techniques encountered. Derived from the ISO Software Engineering Product Quality standard ISO/IEC 9126 (I. IEC, 2001), IEC/ISO 25010 focuses more on software-intensive systems such as the ICPS.

Boehm (1976) and Rubey et al. (1968) defined the original concepts for the quality metrics that these standards evolved from. They proposed that for any given application, individual metrics can provide a qualitative measure of the degree to which the application possesses that associated characteristic. Figure 2.1 illustrates the quality characteristics and sub-characteristics from the Product Quality Model section of the standard that relate to fault diagnostics.

Each sub-category defines a narrower aspect of the degree to which the application possesses that associated parent characteristic. We found that the sub-characteristics of each category were

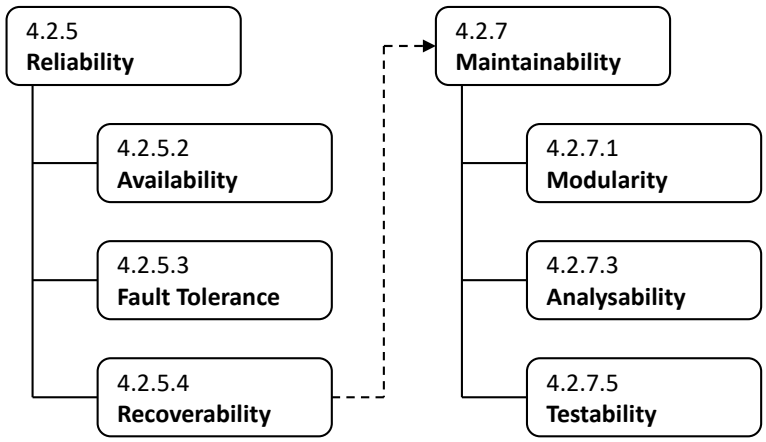

Figure 2.1: IEC/ISO 25010 Qualities related to Fault Diagnostics.

usually easier to quantify since they are more fine-grained. There are also implied dependencies, as shown by the dotted line in Figure 2.1. For example, beneath *Reliability*, IEC/ISO 25010 defines *Recoverability* as the degree to which a device can recover its data in the event of a failure and resume operating correctly. Can this be facilitated by *Modularity*, the device's *Analysability* and its ease of *Testability*? Boehm cautions that these implications need to be considered carefully: software with the qualities of *Device Independence* and *Self-Containedness* is not necessarily *Fault-Tolerant* or *Reliable*.

2.1.6.2 The NASA Technology Readiness Level scale

The NASA Technology Readiness Level (TRL) scale (Straub, 2015; Mankins, 1995) was adapted and used as a qualitative indicator during the classification phase to rank the relative level of maturity of the diagnostic techniques found. The scale is a systematic metric for assessing how mature a particular technology is that. It is now widely used in both aerospace and defence for technology evaluation and selection. The TRL has been progressively refined since the 1980s through its use at NASA, ESA and the US military (Mankins, 2009; Straub, 2015; ESA, 2009). It is now embodied in the standard ISO 16290 (ISO/BSS, 2019). In 2014, the European Association of Research & Technology Organisations (EARTO) identified an increased use of the TRL amongst its members as a planning tool to manage innovation (EARTO, 2014). Table 2.6 in Section 2.4 illustrates the fault diagnostic level characterizations that were adapted

from the NASA categories.

The individual TRLs provide criteria for assigning a classification between TRL 1, representing basic principles being observed or reported, through to TRL 9, characterized by technologies proven in real environments that are ready for widespread adoption. The fault diagnostic TRL descriptions were calibrated using the approach of Terrile et al. (2015). They note that the relative TRL steps are not linear with the steepest steps being in the range TRL 6 to 8. Section 2.4 details the four divisions chosen to classify studies into an appropriate range. The granularity of the resulting TRL categories distinguishes between studies that were purely theoretical and those that were profiling fault diagnostic techniques that are being applied in live environments.

By the end of the classification and analysis phases, fourteen studies had been identified that could be ranked at the highest TRLs between 7 and 9. These report mature, field-proven fault-finding and diagnostic strategies that have been deployed in production environments. In those papers, state-of-the-art exemplars are expected that detail how ICPSs respond to and recover from fault situations they encounter.

2.1.7 Threats to validity

In scoping surveys the primary threats to validity are the choice of which papers to include and the thematic classifications chosen. Surveys are by definition *secondary studies* that report broad, summarized characteristics of *primary studies*, which are the source papers published that present research about an area of interest (Wohlin et al., 2013). Systematic Literature Studies (SLS) provide highly-detailed evaluations of a smaller set of papers (Kitchenham et al., 2010). In contrast, Scoping Studies show where research activity is concentrated and what aspects of a topic are attracting interest. These studies usually examine a larger number of papers in less depth.

Internal validity is concerned with the risks that might lead to an incorrect conclusion. It refers to the degree of confidence that the conclusion is correct and has not been influenced by other variables or factors. In contrast, *External validity* refers to the extent to which the results

of a study and be generalised, applying them to other situations (Hedges, 2012). Internal validity was partially mitigated during the analysis phase by ensuring that each primary paper was initially scanned to determine if it did indeed contain an instance of an ontology class (Haghighatkhah, Banijamali, Pakanen, Oivo & Kuvaja, 2017). For some classes, a list of appropriate synonyms was built iteratively. The inclusion criteria for a paper included a check to see if groups of related terms were present. The classifications defined in Section 2.1.3 such as "*Model-Based*" were expected to show up where models were discussed. However, within the same paper, the classification of "*Model Free*" was expected to be applicable when discussions featured AI, Neural Networks or Data-Driven techniques. Intellectual property restrictions on what can or cannot be published may also be a contributing factor to the amount of detail that can be published about implementations. This was considered when evaluating the relative TRL across sectors.

2.2 Background to the core technologies and sectors

Before examining the results of the survey in detail, the next sections provide a brief technical introduction to the core concepts that define ICPSs, IEC 61499 Function Block architectures, and fault diagnosis. Important domain-specific terms are introduced and general trends observed in the sectors are outlined. The diverse characteristics of these technologies defined many of the constraints that shaped the architectural decisions made during the design of the software architecture, as described in the later chapters of this thesis.

2.2.1 The emergence of Cyber-Physical Systems

By the end of the 1960s, the first general-purpose embedded automation and control systems had been developed for General Motors automotive assembly lines (Laughton & Say, 2013; Parr, 1998). Referred to as Programmable Logic Controllers (PLCs), these specialized control computers often managed individual machines in isolation from other plant machinery. Figure 2.2 shows the first Bedford Model 084 PLC, commissioned in 1969. It demonstrated a 60%

reduction in downtime compared to the electromechanical relay control systems that were in use at the time at General Motors (Lander, 2019).

As manufacturing operations began to demand more complex processing and machinery, they created a demand for real-time data exchange between machinery and the higher-level management systems. Industrial plants slowly became *systems-of-systems* (Samad et al., 2011), with production planning and management systems that relied on up-to-date information from machinery operating on the factory floor. Increasingly sophisticated manufacturing processes required more complex capabilities, leading to the introduction of industrial automation robots.

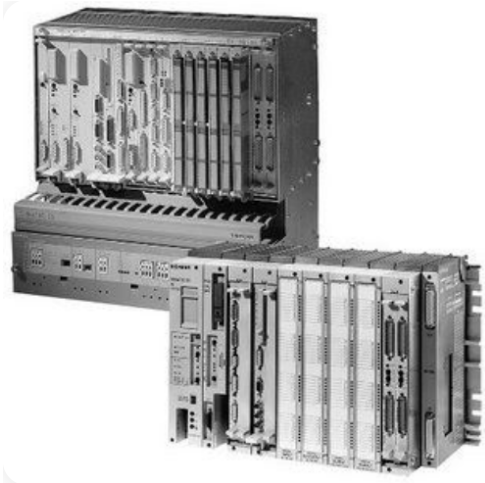

Figure 2.2: Bedford Associates Model 084 PLC, circa 1968.

The term *Cyber-Physical System* (CPS) originated in the work of Bagheti and Gill (2011) in 2006 to describe the new device architectures that were emerging. CPS that operate in industrial or safety-critical environments are referred to as Industrial Cyber-Physical Systems (ICPS). ICPSs have to bridge the divide between their cyber software parts and their physical real-world environments. At this boundary, the discrete behaviour of computational programs has to co-exist with the non-discrete, time-critical behaviours that drive real-world activities. Information passes across these cyber physical boundaries via *transducers*, devices that convert an information-carrying signal from one form to another (Agarwal & Lang, 2005).

Feedback from the environment that is used to make control decisions is captured by a class of transducers called *sensors*. Sensors convert physical characteristics such as orientation, velocity, temperature and proximity into electrical signals that can be processed by the embedded control computer (Gilchrist, 2016; Jazdi, 2014). ICPS also perform physical tasks using *actuators*, mechanical devices that can receive an electrical signal from the control computer and cause a change within their environment (Lee & Seshia, 2016). Motors are special classes

of actuators that create translational or rotational movement. This ability to work autonomously, process data from complex sensors, perform mechanical tasks, and operate collaboratively within networks of heterogeneous ICPSs distinguishes these devices from the earlier, more limited PLCs. Figure 2.3 shows a typical entity in an ICPS, a Franka Emika Panda collaborative robot (Workers, 2020). Collaborative robots or "cobots" often share assembly operations with humans.

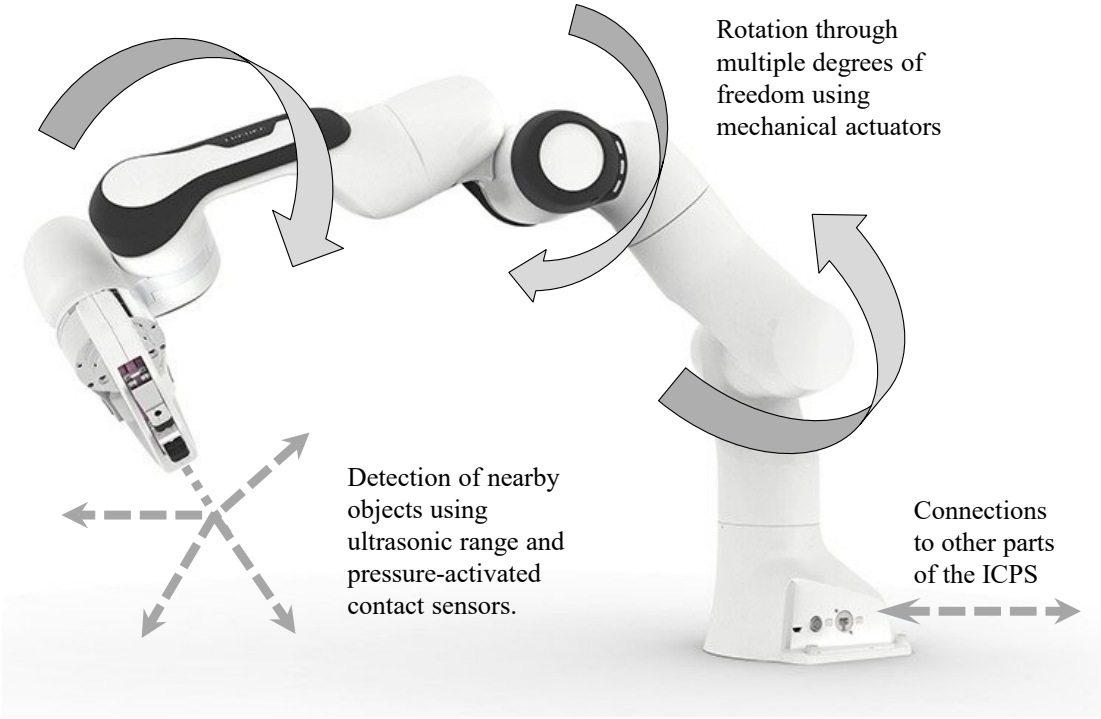

Figure 2.3: A Franka Emika Panda Collaborative Robot operating as part of an ICPS.

Collaborative robots perform a range of pre-programmed tasks, but as an ICPS they can autonomously modify their behaviour in real-time by sensing the presence of nearby objects in their work area. They also work in conjunction with other robots close by, exchanging telemetry over secure networks. At all times, the robot has to honour hazard prevention protocols when it is working in close collaboration with humans. Cremona et al. (2019) comment that safety-critical environments invariably involve time-critical constraints. Hence, ICPSs often have to make operational decisions that are time-critical, based on their perception of the environment they are operating in (Bajracharya, Maimone & Helmick, 2008). This ability to operate safely in

potentially hazardous environments is a core requirement demanded of many ICPS.

2.2.2 The IEC 61499 Function Block reference architecture

The capabilities of PLCs grew rapidly as industry began to find innovative ways of automating manufacturing operations. That demanded more robust software architectures to run the PLC application software on. The standard that preceded IEC 61499, IEC 61131, was released in 1993 (IEC, 2003). It became the predominant reference software architecture adopted by PLC vendors. IEC 61131 drew together the features being implemented independently by PLC vendors into a common standard. The standard incorporated the vendors' preferred programming languages while supporting the development of libraries of device interfaces. However, IEC 61131 relies on a polled execution model which does not scale well as PLC applications become more complex. Its programming languages also lack support for object-oriented code and distributed communications (Thramboulidis, Frey et al., 2011). Hehenberger (2016) comments that one of the advantages of an ICPS is the cost saving that modularity, adaptability, and reusability offer when implementing larger, distributed systems.

The IEC 61499 standard introduced in 2005 addresses these limitations by providing an object-oriented and event-driven architecture that scales well (IEC, 2013a, 2013b, 2006). Figure 2.4 shows a custom Function Block (FB) that has been developed from a standard Basic Function Block (BFB) type. This function block processes a temperature reading in degrees Fahrenheit that is received from another function block that interfaces directly with a temperature sensor.

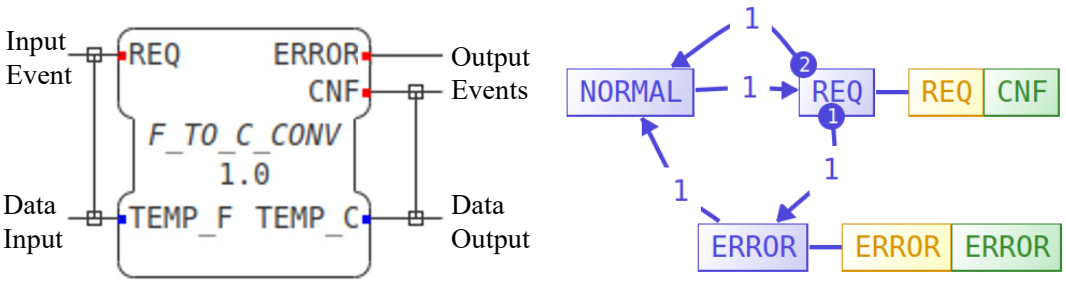

Figure 2.4: IEC 61499 Function Block that converts temperatures with its ECC.

The IEC 61499 standard defines a convention that *Data Inputs* such as the `TEMP_F` port are

drawn on the left side of the function block symbol. When new data has been made available from another connected function block to the input `TEMP_F`, the *Input Event* `REQ` is triggered. Each function block implements its own Moore-type finite state machine (Moore et al., 1956) where *Output Events* and *Data Outputs* shown on the right of the function block symbol depend only on the current state and the values of the Data Inputs it has received. The IEC 61499 *Execution Control Chart* (ECC) (Lindgren et al., 2015) shown in Figure 2.4 details the states and state transitions possible for this function block. For each state transition, one or more software algorithms are triggered to process the data and make control decisions. After the algorithms have executed, the converted temperature is output via `TEMP_C` and the output event `CNF` is triggered. If the temperature cannot be converted, the ECC transitions to the error state and the output event `ERROR` is triggered.

This event-driven behaviour and the ability to encapsulate both state management and algorithms in a function block facilitates loose-coupling and scalability. This modularity also encourages component re-use by enabling developers to build new applications from proven libraries of function blocks.

2.2.3 Fault identification and diagnosis in ICPS

One of the inevitable consequences of the increasing reach and complexity of ICPSs is that they present unique diagnostic challenges when they fail. A *fault* in an ICPS is defined as any change in the operation of an ICPS that leads to unacceptable behaviour or degraded performance (Thombare & Dole, 2014). Before a fault can be recognised, the fault detection system must possess sufficient knowledge of what constitutes both normal and abnormal operation within that ICPS (Milis, Eliades, Panayiotou & Polycarpou, 2016; Harirchi & Ozay, 2018). The subsequent sections examine faults and their diagnosis across three domains where fault identification and management are critical. The papers show that there is a wide range of alternative fault diagnostic approaches whose adoption has been driven primarily by the unique needs of those sectors.

Smart sensors are devices that contain in-built electronics to process readings and transmit

them to the ICPSs they are connected to (Eifert, Eisen, Maiwald & Herwig, 2020). Faults in sensors can include failures in electronic components, both intermittent and permanent, as well as calibration errors that report false readings. Actuators such as those that move aircraft flight control surfaces rely on feedback from multiple position sensors to determine if they have moved to the correct position. Since smart sensors and actuators have their own embedded data processors, software errors can lead to faulty telemetry or operations. The recent failure of the airspeed and angle of attack sensors on the Boeing 737 MAX 8 led to the incorrect operation of the on-board MCAS flight stability software, resulting in catastrophic behavior of the aircraft (Ragheb, 2019).

Identification and diagnostic strategies have to take into account the design characteristics of the systems they are diagnosing. By enabling agents to utilise domain-specific knowledge of IEC 61499 architectures, sets of probable faults can be identified in-advance. The technical characteristics of sensors and actuators can also be made available to agents so they can recognise unexpected behaviors. For example, when the `TEMP_F` input of this function block is triggered by the `REQ` input event shown in Figure 2.4, it should result in either a converted temperature being output or a range error being reported within a pre-determined time interval. Similarly, a request to a temperature sensor to deliver a new reading should result in a response in a predefined timeframe. The AM2302/DHT22 temperature sensor used in later examples is designed to only return a reading once every three seconds; it cannot be sampled any faster than that (AOSONG, 2020). Hence, the diagnostic parameters for testing this sensor must factor that limitation in.

Fault Detection and Identification (FDI) systems therefore adopt a range of approaches. A fault *residual* is defined as the difference between observed system responses and expected fault-free predictions. If these residuals are sufficiently large, then the system must respond appropriately, perhaps by triggering an alarm or performing a deeper investigation. Palmer et al. (2018) describe the primary characteristics of an ideal fault management system: detection methods should be accurate, decisive, and they must act quickly to stop faults propagating further down into other subsystems.

When designing an ICPS, engineers are invariably faced with two choices if they want their systems to operate reliably for the maximum amount of time:

1. *Make them 100% correct and reliable*. They should fail less often, preferably never. This option is often either too expensive or impossible to achieve in many situations.

   or:

2. *Make them more self-aware.* Designers could give their ICPSs the ability to rapidly identify, diagnose and perhaps self-heal as quickly as possible. They may fail, but they will rapidly become operational again. This scenario is typical of the systems NASA designs for its planetary rovers; they are too far away for a human to go and service when something fails. The time delay to control them from Earth also means that they have to be able to work autonomously when required to. Hence, they must be able to respond to problems autonomously, deciding on a course of action from pre-agreed alternatives. They should also be able to self-repair if possible.

These descriptions of fault diagnosis blur the boundaries between design, testing, real-time operational monitoring and self-healing. Hence, a fault diagnosis engine should be able to work collaboratively with an engineer during early-stage design as well as operate semi-autonomously to run longer-term field burn-in trials by itself.

2.2.4 Background to the Automotive Sector

The automotive industry has become increasingly dependent on cyber-physical control systems since Volkswagen introduced their first electronic fuel injection system in 1968 (Osswald, Matz & Lienkamp, 2014). Before this, the forerunners of modern CPS were used in General Motors factories during the late 1960's on their assembly lines.

By 2007, typical high-end vehicles such as the BMW Series 7 relied on 67 distinct micro-controllers called *Embedded Control Units* (ECUs) (Pretschner, Broy, Kruger & Stauner, 2007). The 2011 Chevrolet Volt hybrid required ten million lines of code, which was four million more than was required by the F-35 jet fighter of the same year (Pesce, 2011). In 2009, Charette

believed that by 2016 vehicles would contain up to 100 separate ECUs requiring 100 million lines of source code (Charette, 2009). By 2016, the Ford P-150 utility contained 150 million lines of code operating on 150 ECUs (Saracco, 2016). Saracco attributes this complexity partially to the increased outsourcing of components and complete subsystems to specialised third-party manufacturers. As a result, by 2018, the primary role of automotive manufacturers had become that of systems integrators. They build their vehicles from a collection of complex third-party sub-assemblies, each of which comes with its own ECUs and software. This contrasts starkly with the aerospace industry which maintains much tighter control over its componentry manufacture due to more stringent regulatory and safety-certification requirements (Youn, Hong, Oh & Ahn, 2015; Ferrell & Ferrell, 2017). Charette (2009) provides an additional perspective on this complexity, stating that 30% of the software in automobiles was devoted to diagnostics that assist car mechanics to isolate faults that occur in the field. However, with the current state of the art, it is still difficult for engineers to identify the exact causes of problems.

As the cost of hardware reduces while its capabilities increase, new, advanced features are primarily being implemented via software only. The automotive control systems interact with their users and each other in a myriad of complex ways, often requiring the exchange of data between engine management, electronic stability controls, security and environmental systems. McKinsey reported that by 2014, the ECUs in typical vehicles were capable of exchanging over 25GB of data per hour (McKinsey, 2014). This explosion of capabilities and the resultant complexity is driven in part by the economic value that these systems return. By 2014, the annual value of the software created by automakers had reached 133 billion EUR (Braun et al., 2014). Extrapolating beyond this suggested that by 2020, this would rise to 179 billion EUR (Arpinen, Hämäläinen & Hännikäinen, 2011) for total vehicle sales of 1 trillion EUR. The total global expenditure on research in the automotive sector was expected to rise to US$100 billion annually by 2020 (Hill, Menk & Swiecki, 2016).

Figure 2.5 was adapted from Braun et al. (2014), and Broy and Kruger (2007) with additional data from Hill and Menk (2016). It shows that in 2000, the percentage value of software and computing hardware contributing value to the entire automobile was 22%, of which just 20%

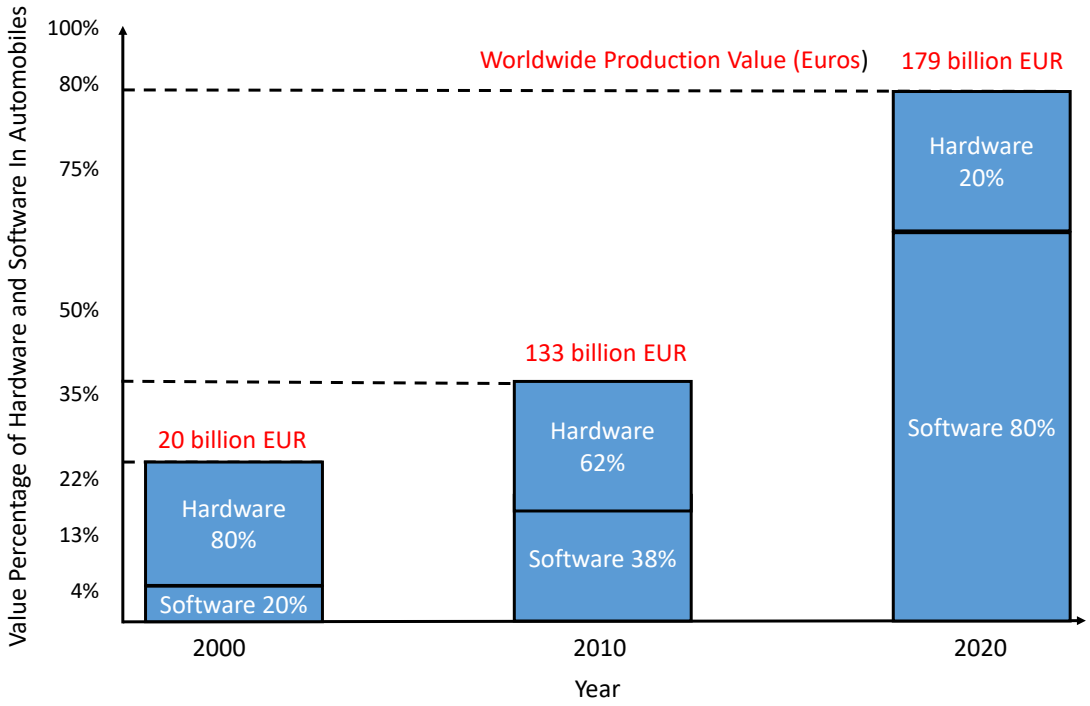

Figure 2.5: The importance of software in automobiles (from Braun et al. 2014).

was software. By 2020, it was expected that software will constitute 80% of the value of the vehicle while the proportional value of the hardware is predicted to have dropped to 20%. The percentages in the boxes show the proportion of the software value to the hardware value that make up each solution. This is expected, since as the hardware becomes increasingly more cost-effective and powerful, the expenditure on implementing features in software alone will continue to become more cost-effective. By 2030, the ratio of electronic to mechanical components in an automobile is expected to rise to 50% (Steinkamp, 2018). While these predictions may have been affected by the 2020 COVID-19 pandemic, there are indications that there are more pressing concerns about technology being expressed by consumers. Figure 2.6 reports a Deloitte 2020 study that surveyed the effect that media reports of accidents with autonomous vehicles had on consumer confidence. The graph charts how cautious consumers felt about adopting new automotive autonomous vehicle technologies (Deloitte, 2020).

Such complexity drives the need for more sophisticated fault diagnostics that can operate in the field. The Stout Risius Ross Automotive Recall Report (2018) observes that in the years

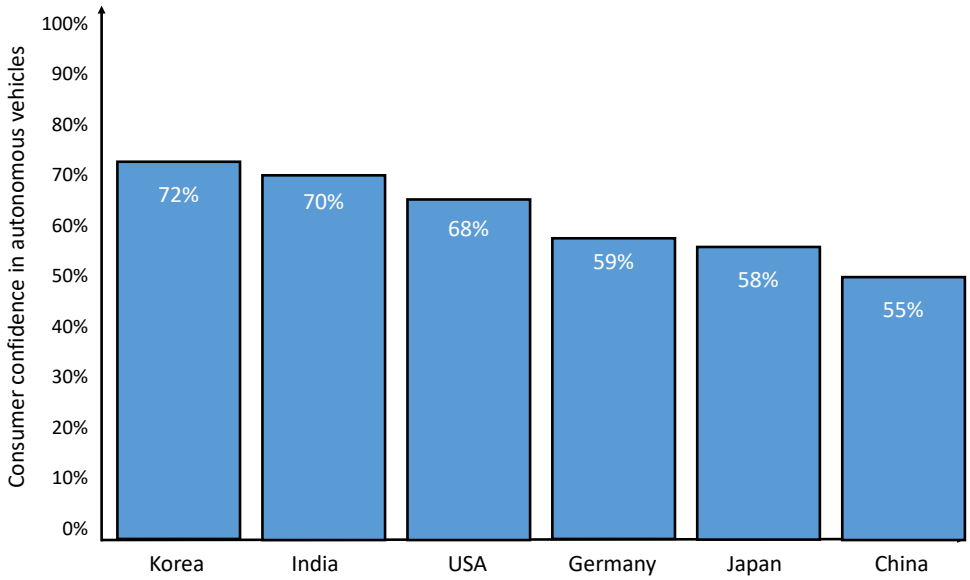

Figure 2.6: Consumer confidence in autonomous vehicle technologies in 2020.

from 2011 to 2015, software-related automotive recalls rose from less than five percent to fifteen percent. They suggest that remote diagnostics and over-the-air updates could lead to US$35 billion in savings in maintenance costs per year in this sector (Mansour, Farag & ElHelw, 2012)

2.2.5 Background to the Aerospace Sector

At the same time that General Motors were pioneering industrial control, NASA was developing the first embedded avionics systems with Integrated Circuits (ICs) for use in their Apollo Moon landing program (Hall, 1972).

Unlike the automotive sector, the aerospace industry is characterized by software and hardware development life cycles for ICPSs that can exceed 35 years (Vassev & Hinchey, 2014). Vassev and Hinchley also note that while Boeing has outsourced more than 70% of the design and production of the 787 Dreamliner, they still maintain very tight control through high levels of supplier interaction. Similarly in the military and NASA, safety is of paramount importance. Current automotive systems are usually defined as *fail-safe*; they ultimately shut-down when faults occur that they cannot manage (Ghadhab, Kuntz, Kuvaiskii & Fetzer, 2015). That is a

luxury not afforded to ICPSs in the avionic and aerospace sector where terminating their current operations in-flight is not an option (Bradatsch, Ungerer, Zalman & Lajtkep, 2011). These two characteristics drive most design and methodology decisions in this sector (Youn et al., 2015; ESA, 2009). Hence, the Spiral and V-Model development methodologies with their emphasis on continuous verification and validation predominate in both the aerospace and automotive sectors (Pecheur, Simmons & Engrand, 2006). The V-Model is discussed in more detail later in Section 9.3.

Aircraft ICPSs can also be viewed from the perspective of the scale of the environment they operate in. In 2013, the US Air Traffic Control system managed over 7,000 aircraft in-transit simultaneously (Sampigethaya & Poovendran, 2013). However, the design of the youngest ground infrastructure components of this system is 47 years old. While time has shown it to be highly reliable, the air traffic control system is expensive to maintain in its current form. Current estimates for support alone are $US40 billion annually (Oster & Strong, 2017). In the midst of this, aircraft manufacturers are also driven by the need to reduce fuel consumption and pollution levels by introducing more dynamic engine management. Given that implementing changes in ground-based systems is difficult and costly, the main opportunities for innovation continue to lie within the individual aircraft themselves. As in the automotive industry, opportunities for innovation lie predominately in the evolving software, not in the hardware.

Modern aircraft are, by design, inherently unstable in flight (Sharp et al., 2010). Dynamic Flight Stability is managed not by mechanical linkages to control surfaces and engines but via networked ICPSs embedded in the mechanical components themselves, hence the use of the term “fly-by-wire”. Modern aircraft fly because their ICPS make rapid, continuous flight adjustments to maintain stability. By distributing control over multiple nodes with redundancy and fault tolerance, current aircraft designs better mitigate the risks of single-point failures. However, single-point failures still occur. The recent failure of the airspeed and angle of attach sensors on the Boeing 737 MAX 8 illustrates this. The fault led to the incorrect operation of the on-board MCAS flight stability software, resulting in the catastrophic behaviour of the aircraft (Ragheb, 2019).

Sampigethaya (2013) notes that improvements in the design of ICPSs have led to the development of Integrated Modular Avionics (IMA) systems that employ multi-core, multi-processor architectures. Such systems are more general-purpose, allowing similar discrete modules to distribute processing across redundant nodes. This has reduced the need for highly-specialized avionics ICPS modules as the hardware the applications run on becomes more homogeneous. This leads to the need for innovative distributed diagnostic approaches with the ability to self-heal via redundant nodes. Boeing's In Vehicle Health Monitoring (IVHM) prognostic systems are evidence of initiatives to improve safety by proactively managing potential failures (Stephenson, 2006). However, the aerospace industry does not employ specialized development methodologies for the prognostic and autonomous features of their systems. Rather, these activities are accommodated within their existing Spiral and V-Model methodologies. Vassev and Hinchey (2014) comment that requirements engineering in these areas remains a promising research area where only a limited number of approaches have been explored.

2.2.6 Background to Manufacturing

Embedded systems were replaced by modern ICPSs as factory operations became more complex and integrated. Ramos et al. (2011) report that the maintenance of factory production systems can be up to 60% of the manufacturing cost. Janasak and Beshears (2007) argue that this is because devices must operate reliably with maximum up-time since downtime increases the overall cost of ownership. Their conclusion is that early diagnostics and fault detection leads to far more rapid problem resolution, ultimately reducing costs.

Manufacturing is often characterized by industry or country-wide initiatives, including the German-based Industry 4.0 (Kang et al., 2016; Wu et al., 2017) and standards institutes, such as the U.S. National Institute of Standards and Technologies (NIST). These organizations seek to foster co-operation as well as help formulate standards. Within these groups, ICPS fault diagnostic initiatives feature predominantly. These are cited as the facilitators of Smart Manufacturing (Nuñez & Borsato, 2017) and sustainability (Monostori, 2014).

Fault diagnostic initiatives in manufacturing therefore embrace two, distinct fronts. Significant research is dedicated to improving the operation of the factory machinery itself (J. Lee et al., 2015; Fleischmann, Kohl & Franke, 2016). At the same time, the devices being manufactured have to be able to participate in fault-management programs when they are deployed to customers. Approaches include Built-In Self Test (BIST) (Smith & Lowenstein, 2009a) and embedded fault reasoning (Grimm, Watzke, Hubauer & Cescolini, 2012) while Lapira and Bagheri (2013) and Chen and Yang (2012) discuss predictive maintenance using data-driven approaches.

2.3 Findings from the study

This section summarises the findings from the studies and the implications for the fault diagnostic techniques chosen for use by the agents. The 127 papers chosen for inclusion present a wide range of diagnostic techniques, published across diverse venues. Additional resources from outside the target domains provided valuable background information. They are listed in the final Miscellaneous section of the paper list available at doi 10.17632/wg9shy9rsm.1.

2.3.1 Industry and academic collaboration

The distribution of papers that discussed industry and academic collaborations is shown in Table 2.2. Such collaborations were detailed in 39% of the papers, with the lowest proportion reported in the manufacturing sector. Nearly 50% of all automotive and aerospace papers reported research performed with an industry partner. Janasak and Beshears (2007) discuss their work with Raytheon's partners, profiling case studies with General Motors, NASA and General Electric. Similar studies from Schoeller (2007) with the US Airforce as well as Efkemann (2008) and Dwyer (2010) with NASA demonstrate research into practical, industry-focused problems.

Baheti and Gill (2011) propose that the diverse architectural, computational and control characteristics of ICPS demand a multi-disciplinary approach that makes academic-industry partnerships highly-effective. Boules et al. (2016) looked deeply at the changing landscape of academic computing research in their US-based study. They suggest that the timeline from

Table 2.2: Industry collaboration by sector.

Collaboration Partner	Aerospace	Automotive	Manufacturing	All
NASA	8			8
General Motors		3		3
United States military	2			2
BMW		2		2
Fraunhoefer Institute			2	2
ASEA Brown Boveri			2	2
Boeing	1			1
German Aerospace	1			1
Raytheon	1			1
Mercedes		1		1
AUDI		1		1
Xerox			1	1
Siemens			1	1
Others	6		6	12
Totals	19	7	12	

research to commercialization is shortening significantly since industry sectors are seeking disruptive innovations and that these are predominantly computing-led. Boules also notes the rise in the number of publications that include a lead author with a corporate involvement, reporting that by 2015, 40% of deep-learning papers published globally emerged from authors with a corporate affiliation.

2.3.2 Diagnostic techniques encountered

Fault identification and diagnostic methods encountered were described in the papers as having evolved along three primary pathways: Physics-Based Modelling and analysis frameworks, Data-driven or Model-free AI techniques, and Knowledge-Based graphical approaches. The study also identified hybrid approaches which blend aspects of these methods.

Physics-Based, Model-Driven Diagnostics

Modeling is used by designers to gain a deeper understanding of a system. By creating models that imitate the physical characteristics of the ICPS components, they can explore the interaction

the sensors and other physical devices have with the cyber parts of the ICPS (Lee & Seshia, 2016). Physics-Based Modelling techniques for diagnostics rely on consistency checks against these models. These detect the differences between the telemetry captured from the live ICPS and the values predicted by the model. Table 2.3 summarises Physics-Based Modelling diagnostic techniques encountered across our survey domains.

Table 2.3: Physics-based Modelling techniques across all sectors.

Technique	**Aerospace**	**Automotive**	**Manufacturing**	**All**
Kalman Filters	23%	0%	14%	13%
Markov models	17%	8%	9%	11%
Fault Trees	17%	19%	5%	14%
Model invalidation	43%	41%	64%	48%
Monitor-based Oracles	10%	11%	9%	10%

Consistency checks use data captured by observers who filter the individual readings to distinguish between noise caused by telemetry errors and values that indicate faulty behaviour (Koitz, Lüftenegger & Wotawa, 2017). These differences will often be small but seldom non-zero when the ICPS is performing within acceptable tolerances (Sankavaram, Kodali & Pattipati, 2013). Techniques for determining when an aspect of a model is invalidated were discussed in 48% of papers, especially in the industrial control domain. Both Kalman Filters and Markov Models were discussed as ways of recognizing model invalidation. These techniques implement observers that can process sequential measurements that vary over time. Kalman Filters are more applicable when the range of possible readings is highly-linear. They apply recursive algorithms where weighted-averages are used to estimate the next value. They work well in noisy environments that produce sequences of unreliable readings. Zolghadri et al. (2015) describe an implementation of a Kalman filter to detect jamming of a flight control surface by filtering the error signal before it is processed by the on-board avionics. The authors explain how the number of sensors providing input to the model affects both the design and worst-case performance. Tuning the model parameters requires trade-offs against the real-time capacities of the diagnostic systems that rely on the model. Shraim et al. (2018) discuss fault management for

quadrotor unmanned vehicles to improve rotor positioning accuracy. Unmanned Aerial Vehicles (UAV) require real-time fault tolerance since they now rely on autonomous, sensor-driven stability control that is no longer managed entirely by the pilot. The models used have to take into account the complex aerodynamic characteristics of the UAV. Dearden et al. discuss similar aspects of autonomous operation, describing fault diagnostics for Mars Rovers where Kalman filtering provides situational awareness to indicate fault conditions (Dearden et al., 2004). They contrast the number of sensors required to manage rover operations with the low computational power available to perform fault identification using multiple subsystem models.

In contrast, Markov Models are used to model non-linear, randomly changing systems with discrete states. A dynamic model is Markov or has the Markov Property if the future state of a system depends only on a limited number of previous states. Markov Chain and Markov Decision processes rely on observing the full set of values or *states* for the aspect of the ICPS that is being diagnosed. In contrast Hidden Markov Models operate where the sequential state of a system is not fully observable. Kunst et al. profile damage propagation through an ICPS using Hidden Markov models (Kunst, Judkins, Lynn & Goodman, 2009). Similarly, Windmann and Niggeman (2015) and Ribero et al. (2011) both apply Markov Models to monitor industrial processes and identify faults as they propagate.

Fault Trees are a way of modelling all reasonably-probable fault scenarios (Schulte, 2018). They are tree structures that facilitate a top-down, systematic approach to identify chains of possible faults. Logical operators can be applied to nodes to identify likely fault pathways. Fault trees are usually considered to be knowledge-based approaches but they were most often encountered in studies that employed hybrid approaches. Mohre et al. demonstrate correlations between fault tree nodes and compositional safety analysis models (Mohrle, Zeller, Hofig, Rothfelder & Liggesmeyer, 2015). Kassmeyer et al. (2016) apply fault trees to track fault scenarios across multiple automotive feature variants.

Across all sectors, a wide range of specialized Model-Invalidation approaches were encountered, both theoretical and in-use. Provan (2014) discusses how acceptable inputs can be modeled, an important pre-requisite to detecting misbehaviour. Monitors are code within

a fault identification system that is responsible for detecting anomalous situations or behavior (Benowitz, 2014). Similarly, Monitor-Based Oracles provide ways of both capturing and evaluating possible fault occurrence (Yen, Zhang, Bastani & Zhang, 2017; Abel, Adir, Blochwitz, Greenberg & Salman, 2013; Schoeller et al., 2007).

Formal modelling languages including the Architecture Analysis & Design Language (AADL) (P. H. Feiler, Lewis & Vestal, 2006) and Modelling and Analysis of Real-time and Embedded Systems (MARTE) (OMG, 2020). These are mainly employed to create models of an ICPS during their design phases. AADL originated in the aerospace sector to model embedded systems and has now found wide use in the automotive domain. MARTE extends the UML to provide similar capabilities. Huang et al. (2014) describe a simulation platform modelled in AADL that allows transient faults to be evaluated. Khlif and Shawky (2008) demonstrate how to use AADL to design co-simulations that are easier to diagnose later. Shulte proposes a state machine architecture for fault detection based on SysML (Schulte, 2018). However, no papers in the survey discussed production ICPS implementations that employed either AADL or MARTE models from the design phases directly. Procter and Feiler address this partially by presenting an introduction to the AADL EMV2 Error Library where they discuss the use of an error ontology during modelling (Procter & Feiler, 2020). The literature was searched for examples of the use of EMV2 in production fault diagnostic systems beyond the design phase, but no examples applicable to IEC 61499 were found. Lu et al. discuss redundancy approaches using AADL and EMV2 however their work does not demonstrate how to apply their fault trees in a production, real-world example (Lu, Dong, Wei & Xiao, 2018). Similarly, Zhang et al. discuss the design of fault tolerant systems using EMV2, but it is applicable only to early-stage modelling (W. Zhang, Shen, Huang, Yang & Xue, 2017).

Creating and maintaining models is labour-intensive, requiring careful design to ensure the model captures all the characteristics needed to represent the system. Many of the fault techniques rely on detecting situations where a model is invalidated. However, Milis et al. (2016) highlight the amount of effort needed to calibrate models. Provan (2014) also discusses two practical impediments to effective model-based diagnosis: the failure to integrate diagnostic

modelling early enough in the requirements process and ambiguities in the models themselves at run-time.

Data-Driven fault diagnostics

Data-Driven diagnostic techniques employ training and learning to forge a representation of the system's behaviour (Sankavaram et al., 2013). Unlike Physics-Based models, Data-Driven fault detection does not rely on the existence of pre-built models. This approach is preferred when the ICPS can provide telemetry that contains enough information to distinguish between either normal or degraded operations. AI fault diagnosers make sense of that information by using discriminating logic that copes with the changes seen in the ICPSs as they occur. This ability to make intelligent decisions distinguishes AI from machine learning, which involves the ICPS learning without being explicitly programmed. Milis (2016) discusses cognitive agents that apply expert reasoning to mimic the behaviour of human experts.

Table 2.4: Model-Free Artificial Intelligence techniques across all sectors.

Technique	**Aerospace**	**Automotive**	**Manufacturing**	**All**
Artificial Neural Networks	54%	50%	47%	50%
Machine Learning	38%	17%	18%	24%
Fuzzy Logic	8%	25%	18%	17%
Big Data	8%	0%	18%	10%
Condition Monitoring	8%	8%	24%	14%

Table 2.4 presents Model-Free Artificial Intelligence techniques across the three domains of interest. Artificial Neural Networks (ANN) (Langer, Eilers & Knorr, 2009; Sargolzaei, Crane, Abbaspour & Noei, 2016; Lapira et al., 2013) and Pattern-recognition algorithms (Fang, Shi, Dong, Fan & Ren, 2017) are also Machine Learning techniques. However, the analysis showed that these terms were used interchangeably in many publications. For this reason, they were categorised separately in this study. Since they do not rely on static, pre-built models as reference points, they remove the need to keep the model up-to-date as the system evolves. Data-Driven diagnostic systems learn behaviours through training. Detection logic allows them to compare

current values with previously learnt values (Ramos et al., 2011). Hence, these Model-Free methods do not have to build a complete understanding of the underlying architecture of the system being examined (Iverson et al., 2012).

Data-Driven approaches often scale better than Model-Based techniques (Wu et al., 2017; Yen et al., 2017). As long as sufficient computational resources are available, Data-Driven techniques work as effectively with a large number of sensors as they do with a few (Iverson et al., 2012). Since they construct knowledge representations dynamically, they are often easier to update than formal models (Azam, Pattipati, Allanach, Poll & Patterson-Hine, 2005).

Unlike Model-Based methods, Data-Driven approaches do not assume the probabilistic distributions of sampled values that Markov processes rely on (Wu et al., 2017). Similarly, AI methods, including machine learning, do not rely on processes being stochastic or random. The trade-off is that while Physics-Based models are labor-intensive to create, model-free techniques require comprehensive and often large example data sets to train the observers (Marzat, Piet-Lahanier, Damongeot & Walter, 2009; Schwabacher & Goebel, 2007; Lapira et al., 2013). Iverson et al. (2012) explain that for avionic ICPSs, large volumes of archival sampled values are collected during routine operations that can be used for training neural networks.

Fuzzy Logic employs truth values that are real numbers between zero and one rather than being boolean (Kim, Lee & Lee, 2011; Y. Chen, 2011; Khoukhi & Khalid, 2015). This allows decisions to be made about non-numerical or imprecise data from ICPSs, stored in structures called fuzzy sets. These sets represent partial truths and decisions are made by arriving at a consensus. Fuzzy Logic algorithms are able to re-evaluate thresholds for situations where values are expected to change dynamically as the system is being observed. Song (2010) discusses recognizing faults using threshold predictions. Each sampled value is checked to see if it falls within a range defined by the previous value read.

Condition monitoring allows Data-Driven fault observers to obtain real-time data about the ICPS they are monitoring. These data points replace the reference values that pre-built Model-Based solutions rely on since AI and machine-learning approaches are do not rely on models (Wu et al., 2017). Lee et al. (2015) and Fleischmann et al. (2016) describe these

techniques in terms of system health monitoring. Where deviations from the norm are observed, the result is similar to the model-invalidation discussed earlier. Wang et al. discuss this in the context of cloud computing and predictive maintenance (Wang, Zhang, Duan & Gao, 2017).

Wang et al. (2018) caution against over-reliance on AI approaches. They suggest that, given the complexity of some fault scenarios, the conclusions drawn by data-driven systems may not be sufficiently robust to be free of false positives and negatives. However, Iverson et al. profile fault finding for the International Space Station (ISS), reporting that when a large amount of nominal data is available, Data-Driven systems can become highly effective at detecting anomalies (Iverson et al., 2012).

Knowledge-Based approaches

Knowledge-Based approaches are applicable where large amounts of historical data are available. The underlying dependencies that define the system are derived from these sources using a range of techniques.

Table 2.5: Knowledge-Based techniques across all sectors.

Technique	Aerospace	Automotive	Industrial	All
Bayesian Networks	0%	17%	43%	31%
Binary Decision Trees	0%	33%	0%	15%
Petri nets	0%	0%	43%	23%
Message Analysis	0%	16%	29%	62%

All fault diagnosis systems need to observe real-time data, basing their evaluations on either qualitative or quantitative aspects of the telemetry. However, only Knowledge-Based approaches utilise significant amounts of historical data to inform their classifiers (J. Lee et al., 2015). Unlike Data-Driven AI approaches, Knowledge-Based methods do not require pre-classified training sets. Rather, they mine the historical data using statistical methods. Chen et al. (2011) explain the value of historical information gathered from experts in building knowledge bases to inform current fault diagnoses. However, it was not clear from the papers surveyed why there was no definitive evidence of Knowledge-Based approaches to fault-detection in Aerospace. A subsequent SCOPUS survey of papers revealed eight papers that discussed Knowledge-Based

approaches in this sector, however all of these were at an earlier time outside the data range of the survey.

The resultant dynamic models Knowledge-Based approaches construct are represented using dependency graphs. Petri nets are directed bipartite graphs where nodes represent discrete fault events that may occur. The graph arcs define possible transitions between states (X. Yang & Chen, 2009; Cabasino, Seatzu, Mahulea & Silva, 2010).

Bayesian Belief Networks are Knowledge-Based directed graphs that model probabilities (Y. Chen, 2011; Kurz, Kaspar & Pilz, 2011). Each node represents a step in a cause and effect chain with a conditional probability. While observing, the fault system updates the probability at a node when new information is available. Hence, Bayesian networks can provide both diagnostic and predictive evaluations.

Binary Decision Diagrams are directed acyclic graphs. Waszecki et al. (2015) encode observation patterns extracted from messages exchanged by automotive ECUs to capture fault scenarios that can be evaluated during diagnosis. Network message analysis also complements other Knowledge-Based approaches, either as a carrier of fault messages or as an indicator of misbehaviour (Song et al., 2010). Schweppe et al. (2009) discuss the Automotive Keyword Protocol ISO 14230:2000 (ISO/IEC, 2000), a widely-accepted standard for analysing faults via network messages exchanged over a vehicle's CAN bus. Pons et al. (2015) outline a similar approach using Causal Graphs rather than Binary Decision Diagrams.

Hybrid fault diagnostic approaches

Hybrid approaches that blend techniques from any of the three broad approaches were encountered in 14% of the papers but featured in 19% of all industrial control studies. Hybrid techniques skewed the overall ratios of our three primary categories since practitioners can adopt any combination of methods to create their fault identification and diagnostic methodologies. Figure 2.7 illustrates the spread of Hybrid approaches across our three domains. Lee et al. (2017) employs Model Invalidation from the Physics-Based Modelling category with Condition Monitoring from the Data-Driven AI category in an intelligent manufacturing scenario. This allows

their system to analyse and predict faults from patterns shared via a cloud-based system. The system is implemented using intelligent agents. Chen et al. (2011) combine Bayesian networks with Fuzzy Logic to diagnose faults in automotive braking systems while Banerjee et al. (2013) profiles a system with an amalgam of Fuzzy Logic Data-Driven predictors and Model-Based statistical data.

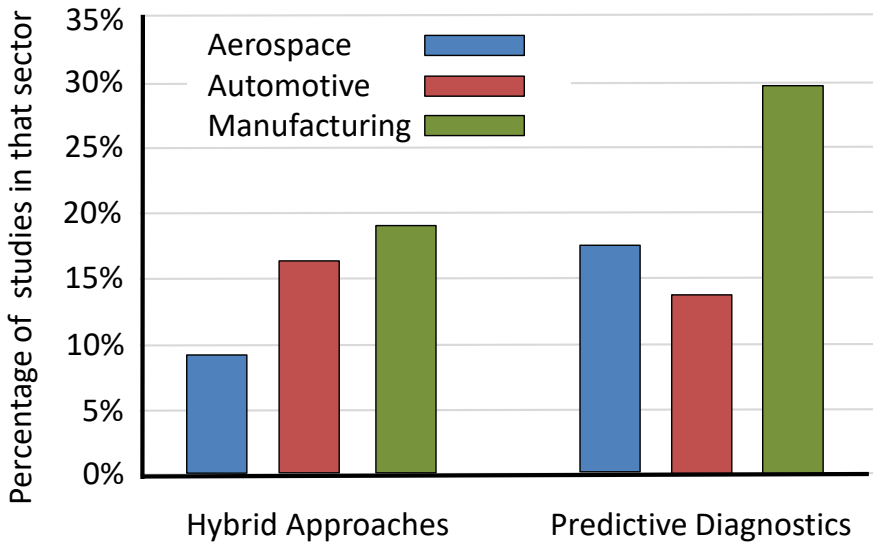

Figure 2.7: Hybrid and Predictive Approaches across all sectors.

Using multiple approaches in this way enables practitioners to apply the most appropriate technique to different aspects of an ICPS. Rizzoni et al. (2009) discuss how both model-based and neural network techniques facilitated the development of on-board diagnostics and fault monitoring to measure vehicle emissions in automobiles. They trace the motivation for continuously assessing emission compliance in each vehicle back to the California Air Resource Board (CARB) requirements that came into force in 1991. Each vehicle is required to monitor its own emissions to ensure compliance. That required neural network approaches to facilitate the tasks of data capture and sensor filtering followed by model invalidation to test compliance.

Predictive Diagnostic techniques

Predictive Diagnostics or Prognostics is the ability to detect the signs of an impending fault before a failure occurs and to estimate when it might happen (Schwabacher & Waterman, 2008).

Figure 2.7 suggests that the ability to predict ICPS faults in advance is of interest in all three domains. Predictive Diagnostics becomes feasible when it is possible to both capture and process large amounts of high-fidelity data about the operation of an ICPS and recognize the fault symptoms in-advance. Janasak and Beshears (2007) state that one aim of European air carriers is that by 2050, all flights should arrive within one minute of their scheduled time. Current delays and disruptions can be up to fifteen minutes due to undiagnosed faults, an issue that better predictive capabilities might alleviate.

2.3.3 Overall trends in the data

In each sector, there is an emphasis on the development of smart sensors and the conditioning of the sensor data using a range of techniques such as Kalman Filters or Markov models. Coupled with that, the representation of ideal values or behaviour was described using either models or dynamically using AI data-mining techniques. Once a definition of what is normal can be determined, deviations from expected values or behaviours can be detected. Artificial Neural Networks and Machine Learning were evidenced as alternatives to Model Invalidation in the Data-Driven AI category. However, the widespread use of hybrid techniques in different parts of the ICPS reflects the complexity of the systems being profiled: no single technique for fault recognition and analysis predominates or is sufficient for all needs. The predominance of Data-Driven techniques in aerospace is in contrast to the lack of evidence for the use of Knowledge-Based approaches in that sector, while Network Message Analysis was a technique profiled in 29% of the manufacturing studies that employed Knowledge-Based approaches. Those contrasts are explored more deeply in Section 2.4 where the most mature techniques are examined in more detail.

2.4 Investigating mature fault diagnostic techniques

RQ1.2 asked what levels of maturity the diagnostic techniques adopted in each sector have achieved. Table 2.6 details the Technology Readiness Level fault classifications adapted for

this study in-parallel with the corresponding NASA descriptions. Mankins (1995) explains that each level in the TRL scale represents a different maturation of the technology or methodology. Heder (2017) notes that the TRL has drawn criticism for its use outside of the environment it was originally designed for, explaining that in the European Union the approach has not always been tailored properly for specific disciplines. However within NASA, the concept of "flight-readiness" was already deeply ingrained in their culture (Feldman, 2000).

Assessing the maturity of a technological approach requires a careful evaluation of the context that it is being trialled or applied in. The TRL categories are divided into four distinct groups. Studies classified as TRL 7 to 9 represent the most mature implementations. They provide a fascinating glimpse of techniques which are either close to or fully operational in live production environments. Adapting this concept to machinery to establish what stage of technological readiness it has reached was a natural step within their context. This was considered when designing the study, carefully crafting the adaptations of the individual NASA level descriptions to ensure the proposed fault level descriptions stayed true to the intent of the TRL.

Studies from TRL 5 and 6 provide evaluations from trials performed in highly-realistic environments beyond the laboratory. They use case studies to illustrate how the diagnostics will work in particular situations. In contrast, studies at TRL 3 and 4 present functioning prototypes that are being evaluated in either a laboratory or simulated environment. TRL 1 and 2 categorize fault identification and diagnosis techniques that are purely theoretical or are presented with a formal mathematical treatment. Papers at this level do not report concrete outcomes from case studies or field trials.

Figure 2.8 highlights the TRL maturity levels observed across all the domains of interest. Amongst the survey papers are studies of fault identification and diagnostic techniques that have moved beyond the laboratory and are being applied in real-world environments. In these papers, it is expected that there would be state-of-the-art exemplars that detail how ICPSs respond to and recover from fault situations they encounter. The papers classified at TRL 7 and above present evaluations of how well these techniques detect and analyse faults and why these approaches

Table 2.6: Adapting the NASA Technology Readiness Levels for Fault Diagnosis.

TRL	NASA categorisation	Proposed Fault categorisation
9	Actual system "flight proven" through successful mission operations.	Actual fault diagnostic system proven through successful identification and classification of real faults in a production environment.
8	Actual system completed and "flight qualified" through test and demonstration (ground or space).	Actual fault diagnostic system qualified through test and demonstration in a production environment.
7	System prototype demonstration in a space environment.	Functioning prototype demonstrated in a production environment.
6	System/subsystem model or prototype demonstration in a relevant environment (ground or space).	Functioning prototype demonstrated finding and/or diagnosing faults in a relevant environment beyond the laboratory.
5	Component and/or breadboard validation in a relevant environment.	Creation of a breadboard and/or software validation that can search for and/or identify faults in a relevant environment.
4	Component and/or breadboard validation in a laboratory environment.	Creation of a breadboard and/or software validation that can search for and/or identify faults in a laboratory environment.
3	Analytical and experimental critical function and/or characteristic proof-of-concept.	Proof-of-concept experiment with an appropriate simulation of the fault environment.
2	Technology concept and/or application formulated.	Concept and technology to perform detection and/or diagnosis proposed, including a mathematical formulation.
1	Basic principles observed and reported.	Basic fault detection or diagnosis principles observed and reported.

were adopted by practitioners.

2.4.1 Studies of mature technologies from aerospace and avionics

Figure 2.9 highlights where the diagnostic research is focused in the aerospace sector. The research concentrates mainly on flight control, high-dependability and predictive fault management. This correlates with the observations from the studies at the highest TRL discussed in this section. Benowitz (2014) profiles the Fault Protection Engine currently used by the NASA Mars Curiosity rover. Since the Rover is too far away to rely on external systems for assistance, the fault protection engine has to proactively and autonomously manage faults within a large number of interrelated subsystems.

Earlier rover designs implemented discrete fault management within each subsystem. On Curiosity, the architecture implements *monitors*, code within each module whose responsibility is to recognise anomalous behavior. Each module has specific knowledge of the subsystem they are operating within that informs their judgments while filtering sensor readings. Monitors signal problems by raising an error flag. As well as detecting faults, they maintain a count of the occurrences that is later used by the fault protection engine to ascertain how persistent or

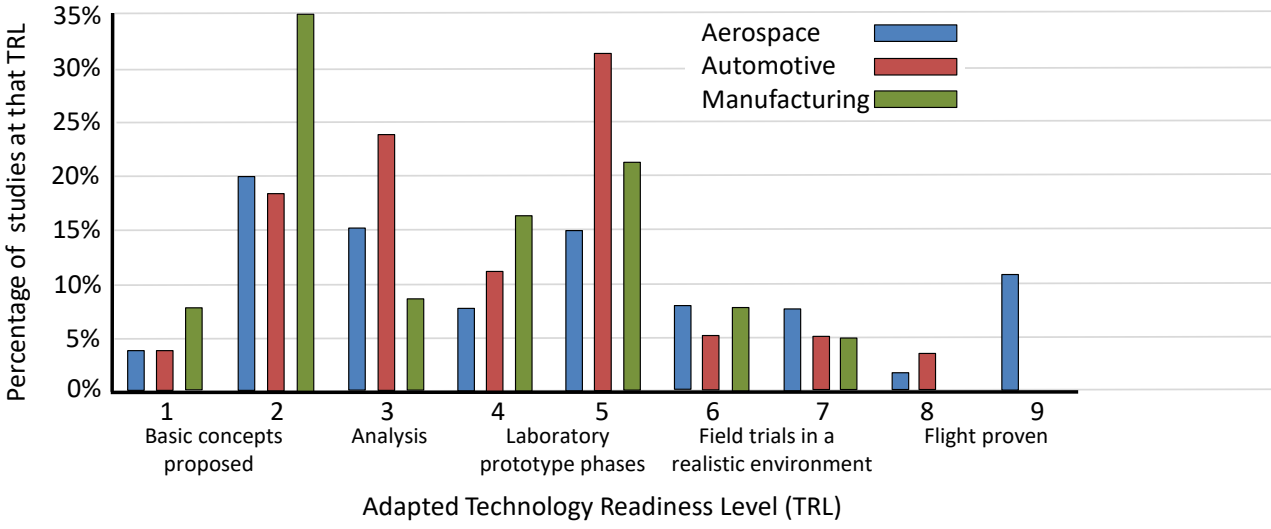

Figure 2.8: Technology Readiness Level by Sector

rover by polling in its own time, making decisions without being flooded by messages from subsystems. The fault engine maintains a model that contains a response that is appropriate to each situation the monitors are signalling. Curiosity has over 1,000 monitors operating at any one time. Since Curiosity may be performing a number of different tasks at any time, ranging from landing to exploration, fault management has to be contextual.

Curiosity's Model-Based approach is in contrast to the Hybrid Model-Based and Data-Driven approach employed by Zolghadri et al. (2015). They profile the flight surface control systems they developed for the Airbus A380. Like Curiosity, their fault management is situation-aware. They note that fault signatures are often difficult to detect when an aircraft is parked or taxiing, or when the data rates from sensors are low. Their approach calculates *residuals*, the result obtained by comparing the current servo positions with the estimated position predicted by the model. They tune the sensitivity of Kalman Filters to establish a trade-off between reliably detecting signals and robustness with respect to normal environmental variations. Azam et al. (2005) take a similar approach using neural networks to dynamically model and monitor

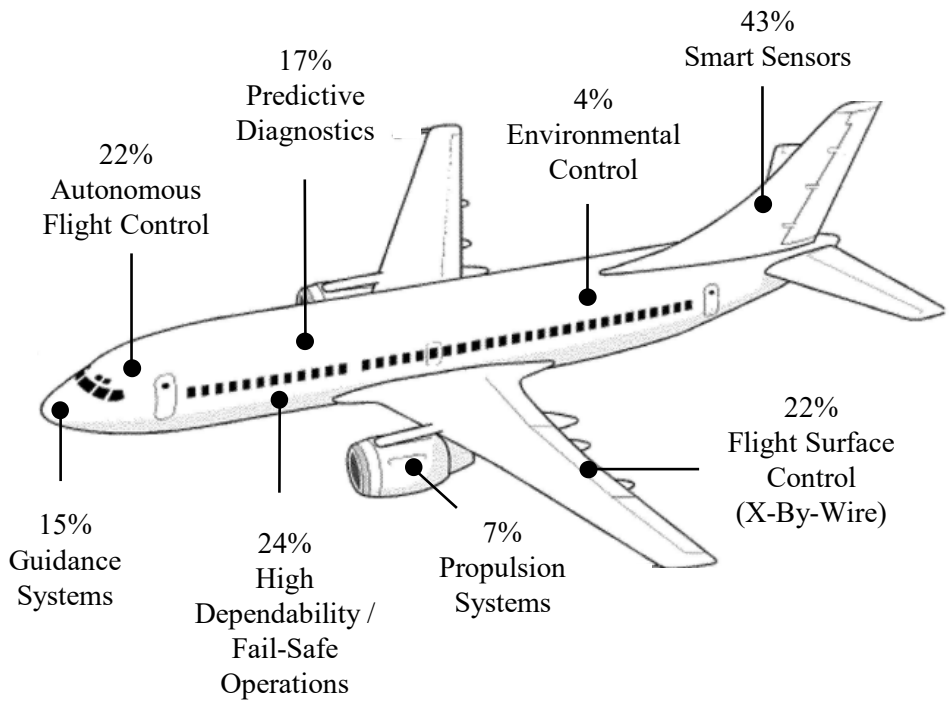

Figure 2.9: Where diagnostic research is focused in the aerospace sector.

fifty flight parameters. They discuss the difficulty of using model-based approaches that cannot manage the complexity of accommodating all reasonable parameters in all flight modes. Their Data-Driven approach also provides estimates of fault severity.

Iverson et al. (2012) provide a highly detailed treatment of the hybrid fault monitoring system certified by NASA for International Space Station (ISS) operations. Across a set of three related and highly detailed studies, Schwabacher et al. (2010) discuss their fault management for the Ares I-X launch pre-diagnostics systems (Schwabacher & Goebel, 2007; Schwabacher & Waterman, 2008; Schwabacher, Aguilar & Figueroa, 2009). Their NASA Inductive Monitoring System (IMS) is a ground-based ICPSs that processes telemetry from the ISS in near real-time. It relies on rule-based, Model-Based and Data Driven algorithms in three distinct subsystems of the IMS. They employ a clustering approach from a fixed number of training points, an approach that allows them to rapidly tailor IMS for new situations. Schwabacher et al. note that there is a need for mission-critical systems such as these to be flight-certified since ground controllers rely on them to make go/no go decisions about launches. They note that some Space Shuttle launches were delayed due to unreliable fault diagnoses. When launch faults can be evaluated more rapidly, redundant or hot-swappable modules can be deployed to reactivate launch sequences to meet critical time windows. Studies such as these help to explain the proliferation of hybrid techniques encountered. In aerospace, 54% used Artificial Neural Networks and 38% employed Machine Learning, coupled with a range of Model Invalidation methods that were discussed in 43% of all aerospace studies.

2.4.2 Studies of mature technologies from the automotive sector

The automotive ecosystem is built up of millions of discrete, complex and mostly unconnected ICPSs. Each vehicle operates as a self-contained network of co-operating subsystems. Stout's Automotive Defect and Recall Report shows that in 2018, nearly eight million vehicles were recalled in the US to address software-based defects (Steinkamp, Levine & Roth, 2019). That total is higher than all the recalls for software issues in the previous five years. Figure 2.10 highlights where diagnostic research is focused in the automotive sector.

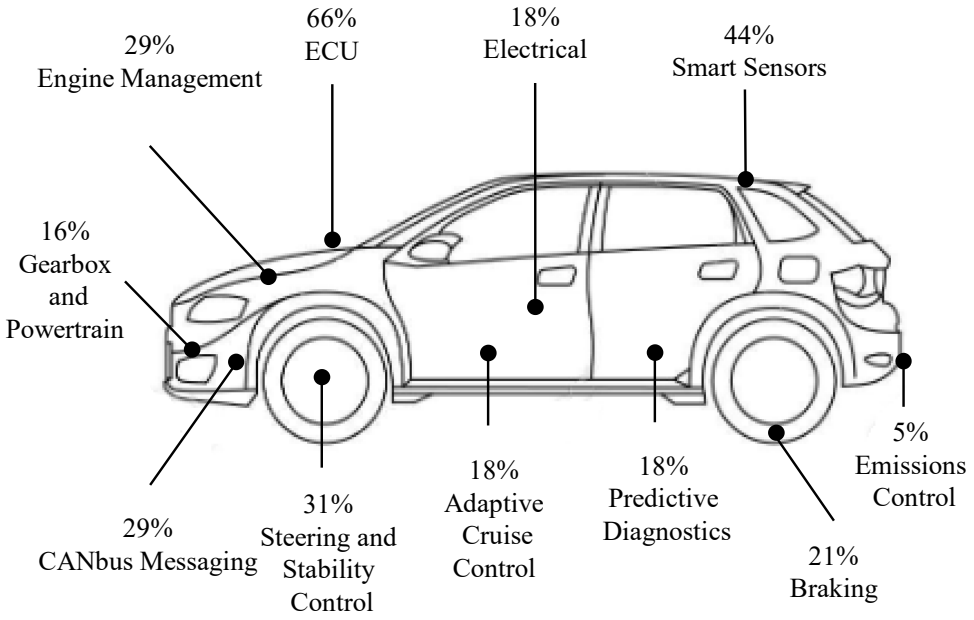

Figure 2.10: Where diagnostic research is focused in the automotive sector.

Modern vehicles feature up to 120 embedded ECUs, connected by five or more system buses (Ebert & Favaro, 2017; Hegde, Mishra & Gurumurthy, 2011). Sarecco highlights how large and complex the software is currently in vehicles, reporting that the 2017 Ford 150 pickup requires 150 million lines of code (Saracco, 2016). Charette (2009) contrasts this with the F-35 Joint Strike Fighter that required only 5.7 million lines of code while the Boeing 787 Dreamliner uses only 6.5 million lines. This complexity is reflected in the automotive survey papers at the highest TRL. Nasri et al (2019) explain that the increasing sophistication of in-car electronics, including Adaptive Cruise Control, Lane Detection and Light Detection And Ranging (LIDAR) technologies, leads to more intricate fault scenarios. They detail the implementation of diagnostics that analyse messages flowing between subsystems on the vehicle's Controller Area Network (CAN). The current diagnostic tools rely on proprietary software from vendors that are not easy to integrate into system-wide diagnostic frameworks. They detail their implementation of a hierarchical chain of localised diagnosers that are monitored by a single global fault analyser. A Directed Graph approach is used to identify faults, capturing CAN messages via hardware-in-loop connections.

The scope of what is deemed a "safety-critical" component in the automotive sector is also

changing. In May 2018, back-up cameras became mandatory on US vehicles, transforming an optional luxury item into something that required much more rigorous quality control and deeper vehicle integration (Steinkamp et al., 2019). Over-the-Air (OTA) access to diagnostic data from automobiles is profiled as one route to addressing the difficulty of fault-finding in disconnected automotive ICPSs. The global remote diagnostics market is forecasted to grow at 17% annually over the next five years, driven primarily by the potential operational cost savings to auto makers (Technavio, 2018). Steinkamp et al. (2019). describe General Motors new OTA system which is capable of handling 4.5 TB of data per hour from vehicles

However, Dragojevic et al. (2018) identify remote access to diagnostic data from a vehicle as a significant technical challenge. Traditional automotive architectures featured highly-specialized ECUs that were optimized for minimal functionality to balance safety concerns. Full operating systems for vehicles emerged though middleware such as Adaptive AUTOSAR (Fürst & Bechter, 2016), leading to greater opportunities to aggregate diagnostic data that could be shared with remote fault analysis systems. Without functionality such as OTA, remote vehicle diagnostics cannot be performed in an IIoT ecosystem. Dragojevic et al.(2018) profile their work on an OTA bridge solution that connects with the on-board vehicle network. However, they note that Adaptive AUTOSAR needs to encompass safety aspects to certifiable levels before it can be widely deployed.

Kane, Fuhrman and Koopman detail the use of runtime monitor-based oracles that mine the data used by OTA systems for fault finding (Kane, Fuhrman & Koopman, 2014). Runtime monitors analyze system traces to see if they conform to acceptable behavior patterns. They tune their oracles using large amounts of previously captured telemetry and describe methods used during live vehicle trials. Since monitors operate as hardware-in-loop devices and often interact with safety-critical components, they have to be designed as high-integrity devices. They address this by creating isolated monitors with well-defined interfaces.

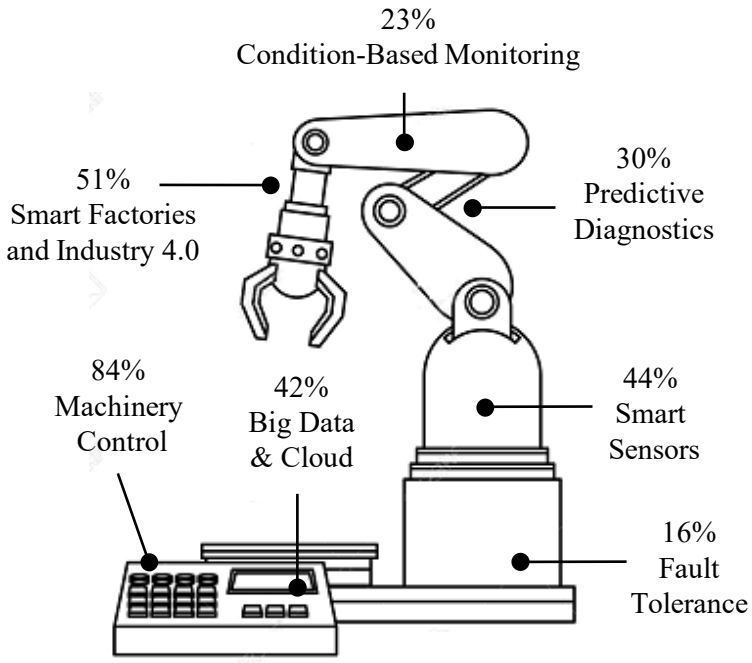

Figure 2.11: Where diagnostic research is focused in the industrial control sector.

2.4.3 Studies of mature technologies from manufacturing

Unlike the automotive and aerospace sector, most industrial systems are stationary in one location and are therefore easier to connect into factory-wide monitoring systems. Industrial production machinery therefore offers numerous opportunities to perform local or remote diagnostics. Ramos et al. cite maintenance costs of up to 60% of the production costs as a key driver for factory diagnostics and prognostics (Ramos et al., 2011). Figure 2.11 highlights where diagnostic research is focused in the manufacturing and industrial control sector. International, industry-wide initiatives foster standardization across this sector. Chen et al. (2012) discuss trials of sensors for gearboxes in the context of manufacturing initiatives such as the Machinery Information Management Open Systems Alliance (MIMOSA) (MIMOSA, 2019). Lee, Jin and Bagheri (2015) discuss Industry 4.0 and Big Data as a similar drivers of standardization. Their approach demonstrates end-to-end factory machinery feeding sensor data into multiple analytical systems for near real-time fault identification and prediction. They employ deep-learning for Data-Driven prognostics.

Ramos et al. (2011) profile Service Orientated Architectures to expose fault-finding services

at multiple factory levels. Their case study focuses on self-recovering machinery that is supported by the factory infrastructure using hardware-in-loop techniques. Manufacturing is typically managed by multi-layer IT infrastructures that connect higher-level Enterprise Resource Planning (ERP) through layers down to factory automation systems such as SCADA. Ramos et al. also profile their eSonia system which manages assets on multiple levels. Many production operations require assembly lines to be able to be re-configured dynamically to suit changes in demand. This requires a degree of self-awareness from plant equipment, which must be able to signal if it is available when changes are requested.

2.5 Conclusions and planning for the next phase

This Scoping Study was written with a view to providing an overview of mature fault identification and diagnosis techniques prior to the architectural design phase of the fault diagnostic engine commencing. It presents a view of the current practice of practitioners who are creating ICPSs in sectors that are complex and often safety-critical. Note how the wide use of Model-Based (62%) and alternative Data-Driven AI (33%) techniques across the aerospace, automotive, and industrial control domains reflects the complexity of the current ICPS application space.

As the number of interconnected ICPSs increases along with the intricacy of the tasks they manage, the use of Model-Based approaches alone was often profiled as becoming intractable. For example, Milis (2016) discussed the difficulties of calibrating models to align them with real systems. The scalability of models was also discussed in this context, but only Yen et al. (2017) discussed partial models, a technique for segmenting models into sub-models. No studies were encountered that profiled Digital Twins as a solution, highlighting that research in this area may be at an early stage of development. Model-Based diagnosis remains a viable strategy, yet how practitioners create complete-enough or partial models quickly and reliably remains a challenge. The AADL EV2 Error Annex has potential to be used beyond the early modelling stages, however no evidence of its use in the field was found.

Model-Free AI approaches were evidenced as a viable way of addressing this challenge,

Table 2.7: Recurring themes across all sectors.

Theme	Aerospace	Automotive	Industrial Control
Existing degree of connectivity?	Low.	Becoming more connected.	Already highly. connected.
Difficulty of becoming more connected?	Hard due to distance and low bandwidth.	Hard due to large number of discrete vehicle instances.	Already highly-connected due to high degree of localization.
Amount of diagnostic data available?	High.	Becoming very high.	Already very high.
Need for autonomous operation?	Very high in remote planetary rovers and drones	Very high due to cost of local fault repair.	Already established via predictive diagnostics and self-management.

demonstrating the increasing sophistication of current machine learning systems. However, there was no discussion of explainable AI, where the decisions made by algorithms must be justified empirically.

Predictive Diagnostics is a promising area that was often discussed in-context with the ability to mine sensor data with enough granularity to allow faults to be predicted. Predictive techniques were prevalent in 30% of all industrial control studies, driven by the availability of large amounts of local data. Further research to develop remote connectivity in the aerospace and automotive sectors should lead to more powerful predictive capabilities. However, the potential volume of the data available from these ICPSs also presents challenges of scale.

Statistical aspects of Knowledge-Based diagnostic approaches were not represented in recent studies across the aerospace sector. Most applications of the technique in the automotive and industrial control sectors discussed Bayesian approaches and various Petri net derivatives. This may be due to the increasing presence of Hybrid approaches which employ Knowledge-Based methods in the midst of other techniques. There was little evidence of traditional Expert Systems.

Connectivity is a key characteristic of ICPS, yet it has deeper implications in our sectors

of interest. Table 2.7 illustrates how connectivity for facilitating diagnostics is made more challenging because of the different environments ICPSs operate in. Brief discussions in the papers of emerging cloud technologies pointed towards ways of establishing connectivity in more achievable ways.

While the TRL analysis provided a way of identifying and profiling the most mature approaches, those results cannot always be extrapolated across all three sectors. Almost all the avionic and aerospace studies profiled originated from organizations who were partnering with agencies such as NASA and ESA. These do not face the same intellectual property restrictions that restrict what we might expect to find published in the automotive and industrial control sectors.

2.6 Implications from revisiting the research questions

RQ1.1 was designed to address the original research question RQ1. While this research ultimately adopted a multi-agent approach, research questions RQ1 and RQ2 were designed to be technique-agnostic. RQ1 sought to identify and understand general fault-finding techniques for serious, industrial-scale CPS and to consider if any of those techniques might be applicable to IEC 61499. Then, RQ2 sought to identify gaps in those general fault-finding strategies that could not address the particular needs of function blocks. While multi-agent approaches were identified in the literature searches, but they were not the prime focus of the survey.

The proliferation of hybrid fault systems that blend different aspects and techniques reached 19% in the industrial control sector, indicating the importance of further research into multiple-method solutions, where models are tuned by real-time data. Design-for-Certification was highlighted as a significant driver to ensure products could be deployed beyond the laboratory (Dragojević et al., 2018; Schwabacher & Goebel, 2007; Schwabacher et al., 2010).

However, creating and maintaining models is clearly labour-intensive. Once created, Model-Based fault identification techniques rely on detecting situations where some aspect of a model is invalidated. Detecting anomalies highlights where part of the ICPS is not performing nominally.

However, Milis et al. (Milis et al., 2016) highlighted the effort needed to calibrate models. Provan (Provan, 2014) provides further support for one of the goals of our research, the focus on Model-Driven Development with Diagnostics based design. Provan cites the failure to integrate diagnostic modelling early enough in the requirements process and the problem of ambiguities in the models themselves at run-time. RQ1.2 asked what levels of maturity the techniques have achieved. In Section 2.4.1 and in Table 2.3, model-based approaches and model-invalidation are shown to predominate in all three sectors.

RQ1.3 sought to identify key gaps and challenges. The IEC 61499 architecture provides design artefacts which may be of use to address these concerns if a Model-Based approach is adopted following the evaluation of the architectural design. Each function block application provides an application definition file that details all of the function blocks and their connections. An IEC 61499 function block is a model with well-defined characteristics. Along with the full function block application definition file, these resources may be sufficient for the agents to automatically elucidate their own model of the application. Coupled with the function block type definitions, the agents can be supplied with sufficient information about the behavioural characteristics of each function block. This should lead to models which could compare real-time data to facilitate model invalidation during diagnostic exercises.

This suggests that augmenting an ICPS with a fault-diagnostic engine facilitates collaborative self-diagnosis and long-term predictive diagnostics with a potentially life-cycle wide scope. This should lead to ICPSs in which it is easier to find faults. More than that, making the design of diagnostics a key part of Model-Driven Development should lead to the creation of inherently more reliable ICPSs, ones that are demonstrably more correct-by-construction (Hamilton & Zeldin, 1980).

RQ2 asks which diagnostic needs of IEC 61499 cannot be met by these techniques. No paper directly addressed how to interact with an ICPS from inside the application apart from those that detailed Built-In Test (BIT) approaches (Hale & Bollas, 2018) or variations on Unit Testing (Hametner et al., 2014). This points to a need for novel approaches that are able to dynamically create and configure points-of-presence that can capture telemetry directly from

function blocks.

Chapter 3

Crafting a Research Methodology

"In general, we look for a new law by the following process: First we guess it. Then we compute the consequences of the guess to see what would be implied if this law that we guessed is right. Then we compare the result of the computation to nature, with experiment or experience, compare it directly with observation, to see if it works. If it disagrees with experiment it is wrong. In that simple statement is the key to science. It does not make any difference how beautiful your guess is. It does not make any difference how smart you are, who made the guess, or what his name is - if it disagrees with experiment it is wrong. That is all there is to it."

"The Character of the Physical Law", Richard Feynman (1964)

Chapter 1 detailed the research questions and proposed ways to explore fault identification, diagnosis, and management techniques for ICPSs. The primary goal is to provide a way of diagnosing faults across the entire life-cycle of an ICPS built with IEC 61499 function blocks. Allied to that, the secondary goal seeks to find ways of encouraging engineers to address fault management early in their design cycles. Chapter 9 later describes the Model-Driven Development with Diagnostics methodology that resulted from this thinking to demonstrate the value of this approach.

This chapter details the Design Science Methodology that was chosen to guide this research. Alternative methodologies were considered before deciding to perform a series of design and experimental phases with appropriate evaluation criteria. The first phase undertaken was an architectural design of the engine. This provided a detailed description of the subsystems needed

to construct the engine. For each subsystem, appropriate quality and performance criteria were proposed to provide a way of evaluating the outcome of that phase. Design Science is iterative: the outcome and learnings from each phase drive the refinement of the phases that follow. Hence this chapter only provides a broad outline of the direction forward. Each of the chapters that follow have a similar structure, documenting the goals of the phase, providing additional background information, and then considering how well the part of the engine that was developed performs under appropriate conditions.

3.1 Design Science in context

Hevner, March and Park (2004) describe Design Science as a paradigm that seeks to extend the boundaries of a discipline by creating artefacts iteratively. They explain how the step from one planned phase to the next includes an evaluation of the effectiveness of what was built to guide the refinement of the goals and direction for the next phase. However, Wohlin and Aurum (2015) caution that choosing an appropriate research methodology is hard. They explain that this is often because some of the underlying assumptions of alternative methods are not well-understood by practitioners before they chose their methodology.

Hindsight was valuable during this doctoral research. The Design Science methodology applied during the prior Master's research led to the development of a novel, robust methodology for fault traceability as well as a prototype of a software application to demonstrate it (B. Dowdeswell, Sinha & Haemmerle, 2016). Niels Bohr described an expert as "... *one who has found out by his own painful experience all the mistakes that one can make in a very narrow field*" (Coughlan, 1954, page 62). Executing a methodology, making mistakes, and learning from them proved to be highly valuable during both the prior and the current research.

Easterbrooke (2008) explains that feedback-driven cycles address one of the most common concerns in research projects that involve building artefacts. Researchers sometimes fall into the trap of continuously coding an application or building a device simply for the pleasure of doing it. While serendipity often arises during such "tinkering", more often this behaviour is

symptomatic of a loss of focus. That invariably stalls progress. Losing sight of their goals often hinders the recognition of novel, applicable and creative outcomes encountered along the way (Rodríguez, 2017). Gregor and Hevner (2013) also discuss the role of artefacts, citing G.H.Harvey's important question about research that should be asked before it is undertaken, and especially before it is published: *"Is is true? Is it new? Is it interesting?"* (Wilson, 2002, page 338).

Focus groups are one approach to help steer the refinement and evaluation process (Tremblay, Hevner & Berndt, 2010; O'Raghallaigh, Sammon & Murphy, 2012). During all stages of this research, the three members of the supervisory team participated in regular, semi-formal evaluation sessions. These were complemented by numerous informal discussions. That interaction proved invaluable, helping the lead researcher to stay focused on key deliverables that illustrated how the core goals were being met. This approach also helped to identify potential publications applicable to the current work. In each of the research chapters that follow, journal and conference paper publications that arose out of the work are noted.

This research could have been performed in an industry company setting, considering real-world scenarios as exemplars to test the engine by following an Action Research methodology (Avison, Lau, Myers & Nielsen, 1999). Santos et al. (2011) suggest that situating a research program in the midst of a production environment provides valuable expert input from practitioners. Often, the artefacts produced are more effective in addressing real-world problems than purely laboratory-grown solutions. However, no local organizations or companies could be identified who use IEC 61499 for ICPS development in New Zealand. Alternatively, providing a number of theoretical Case Studies based on overseas implementations such as the NOJA Power Smart Grid closer components could have been valuable (NOJA, 2015). However, their designs are proprietary, so limited technical implementation information is available to allow experiments to be performed.

Runeson and Höst caution on relying on what they call "toy" case studies that lack sufficient depth (Runeson & Höst, 2009). This concern was addressed by analysing several generations of IEC 61499 Smart Grid applications made available by Dr Gulnara Zhabelova from the

Luleå Institute of Technology in Sweden. Her work with these components is discussed in her articles (Zhabelova et al., 2016) and in a subsequent journal article she co-authored with us (Sinha, Dowdeswell, Zhabelova & Vyatkin, 2018). These devices proved invaluable. They were re-developed and are discussed in-depth later in Chapter 10.

3.2 A framework for Design Science research

Hevner and March (2004) proposed a framework for conducting design science research that includes seven key aspects. These are summarized in Table 3.1. Each aspect proposes a protocol and appropriate processes to drive each design phase forward.

Table 3.1: Design Science Research Guidelines

Guideline 1: Design as an Artefact.	Design Science research must produce a viable artefact in the form of a construct, a model, a method or an instantiation.
Guideline 2: Problem relevance.	The objective of Design Science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design evaluation.	The utility, quality, and efficacy of a design artefact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research contributions.	Effective Design Science research must provide clear and verifiable contributions in the areas of the design artefact, design foundations, and/or design methodologies.
Guideline 5: Research rigor.	Design Science research relies upon the application of rigorous methods in both the construction and evaluation of the design artefact.
Guideline 6: Design as a search process.	The search for an effective artefact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of research.	Design Science research must be presented effectively to both technology-oriented and management-oriented audiences.

3.2.1 Guideline 1: Design as an Artefact

Hevner and and March's first guideline states that Design Science research must produce a viable artefact. Norman (2013) asserts that design is manifested not just in the quality of the resulting artefact, but also in the quality of the theory and processes that produced it. The primary artefact produced by this research is a fault diagnostic engine that addresses RQ3 from Section 1.1. That artefact demonstrates diagnostic fault-finding techniques and collaboration by agents. It also demonstrates how the use of the engine addresses limitations in the current IEC 61499 development IDEs by enabling fault finding during development and commissioning phases. The goals of IEC 61499 include re-usability of function block components (IEC, 2013b). Hence the engine should encourage the development of diagnostic resources to make those library components more robust and diagnosable-by-design (Dowdeswell, Sinha & MacDonell, 2020a).

3.2.2 Guideline 2: Problem relevance

The engine contributes nothing unless it addresses real-world issues and problems faced by the IEC 61499 development community (Hevner et al., 2004). However, this research should also attempt to reach beyond the existing gaps to propose alternative approaches to address Guideline 4. Applegate and King (1999) caution that there is a balance between the need for rigour, discussed later in Guideline 5, and the desire to only include experiments that are deemed relevant. They argue that there is a need for both.

3.2.3 Guideline 3: Design evaluation

The phase that creates and documents the software architecture of the engine also provides the framework for all subsequent evaluations of the engine. The requirements and the architecture focus the engine's functionality on addressing the immediate needs expressed by the stakeholders. That requires a balanced approach to ensure that the engine also seeks to advance the field, as discussed in Guideline 2 above. The evaluation criteria created in the Attribute-Driven Design

process presented in Chapter 4 details quality attributes that each subsystem of the engine should possess. Each chapter that documents the implementation of one or more aspects of the engine therefore includes an evaluation based on Architectural Tradeoff Analysis Method (ATAM) principles (Kazman, Klein & Clements, 2000). The results of each evaluation help to refine the scope and activities of the subsequent Design Science phases.

3.2.4 Guideline 4: Research contributions

Section 1.3 briefly outlined the expected contributions from this research. The Model-Driven Design with Diagnostics methodology attempts to build on earlier contributions in the field from other practitioners including Bregon et al. (2014), Basha et al. (2012) and Bezivin (2004).

The engine is not a simple creation; finding faults in a complex, misbehaving ICPS is hard. Looking outside the field of software engineering, Greenberg (1991) reinforces Goethe injunction to "*Dream no small dreams for they have no power to move the hearts of men*". Improved functionality alone is not sufficient to drive innovation, since innovation only occurs when technologies garner adoption by a community of users. Excellent designers incorporate both style and aesthetics in their artefacts. Gelernter describes this as *"machine beauty"*, the marriage between efficacy and simplicity, driving innovation in artefacts such as the engine as much as it does in science and technology (Gelernter, 1999, page 3). Expert photographers will assert that when considering cameras, there is a world of difference between core functionality and functionality with style.

3.2.5 Guideline 5: Research rigour

Research rigour in the context of the design and development of the engine is ensured by performing the evaluations discussed in Guideline 3. However, there is a fine balance between adhering to plans, entertaining alternative approaches, and finding serendipitous solutions to seemingly intractable problems. Buchanan (1992) and Brooks (1987) both argue that while some technological advances are the result of creative and innovative design science processes, some solutions are arrived at via capricious or sometimes arbitrary routes. They describe these

as the *"Ah Ha!"* moments that make research in these realms so rewarding.

3.2.6 Guideline 6: Design as a search process

Design is an iterative search to find an effective solution for a problem. Simon (1996) describes this as a cyclic process of generating design alternatives followed by testing each iteration against known requirements and constraints. However, when the range of possible solutions or routes that could be followed is large, we cross into the space described by Buchanan as *"wicked problems"* (Buchanan, 1992, page 15). These are problems whose descriptions are extremely difficult to formulate and where the data that accompanies them is confusing or indeterminate. While Design Science processes through iterative phases, it is sometimes hard to predict the number of iterations that would be needed to produce a solution. In such circumstances, the solution may emerge unexpectedly.

Hevner and March (2004) suggest that one way of keeping such scenarios under control is to continually attempt to determine how close the artefact developed in an iteration is to being an optimal solution. This assessment can be facilitated by the application of Guideline 3 that addresses the methods for evaluating designs.

3.2.7 Guideline 7: Communication of research

The Diagnosable-by-Design methodology and the engine were never intended to be something that only existed within the confines of a research environment. Solutions mature when they are continually presented with real-world scenarios that describe contemporary problems. Likewise, when research is published, there is a juxtaposition of fresh ideas from other researchers that can be used to approach problems in new ways. Hence, throughout this work, collaboration with other researchers and examining both alternative methodologies and similar engines in the wild were key contributors to maturing the diagnostic approaches. Chapter 11 proposes what the next post-doctoral phase of research should involve, including further publications from this work.

3.3 Considering the Design Science Research phases

The scope and expected goals of each Design Science phase are detailed in Table 3.2. This broad scope was created early in the PGR9 Confirmation of Candidature process that commenced in August 2017.

It was expected that these phases would probably overlap, might be performed in a different order, and might contain discontiguous sub-phases. In hindsight, the list closely reflects the overall research and development stages that were executed. Progress was tracked over the 215 weeks of the doctoral program by writing brief weekly research discussion and evaluation reports. A total of 145 reports were submitted and reviewed by the primary supervisor during the program. The subsequent sections outline the activities and milestones from each of those proposed phases in more detail.

Table 3.2: Design Science Phases for the creation of the engine.

Phase	**Description of the phase and goals**
Architecting the engine	Propose the software architecture of the fault diagnostic engine. Document the architecture in an appropriate software document format. Define evaluation criteria for each subsystem.
Creating agents	Create a single agent and then multiple agents. Define and evaluate goals for fault diagnosis. Demonstrate agent interaction, independent operation, resource sharing and state management. Evaluate the agents against the Software Architecture Document evaluation criteria.
Diagnostic Point design	Creation of the customised Diagnostic Point function block type and evaluation of its performance. Demonstrate the qualities of being light-weight and low-latency. Show how a single agent can marshal multiple diagnostic points, capture telemetry and inject test data into function block inputs. Evaluate the diagnostic points against the Software Architecture Document evaluation criteria.
What agents believe	Design and create the core belief structures needed by the agents to allow them to manage the fault evidence they capture. Evaluate the agents belief structures and capabilities against the Software Architecture Document evaluation criteria.
How do agents act?	Design and develop the diagnostic language that allows a situated agent to use its skills and abilities. Show how this language enables the agent to form and communicate beliefs after it has performed a diagnostic investigation. Evaluate the performance of the agent against the Software Architecture Document evaluation criteria.
Explore representative systems	Demonstrate the fundamentals of Model-Driven Development with Diagnostics by creating and evaluating a set of representative IEC 61499 function block applications. Demonstrate interactions with agents to capture telemetry from all defined Diagnostic Points. Identify candidate faults from a set of pre-defined fault scenarios. How well does the completed engine and the fault diagnostic methodology perform against the evaluation criteria defined for it?

3.3.1 Architecting the engine and creating agents

Time spent in December 2017 at the Luleå Institute of Technology in Sweden shaped much of the early thinking on fault diagnosis for IEC 61499 architectures. It also helped to identify gaps in both the existing literature and the current development tools. This led to a clearer understanding of the potential of multi-agent systems to address those needs.

The Scoping Survey Literature review and the subsequent publication (Dowdeswell, Sinha & MacDonell, 2020b) was created in-parallel with studying agents and learning the 4diac development system (Strasser et al., 2008). The GORITE engine was studied during three-week course that was led by Associate Professor Dennis Jarvis at Luleå. Discussions with Dennis resulted in the adoption of his GORITE multi-agent framework and the creation of a prototype team of agents. This work led to the agent architecture discussed further in Chapter 5.

The Software Architecture Document outlined in Chapter 4 was developed in-parallel with the journal article on the architecture, the implementation of the engine in GORITE, and the prototyping of the first fault diagnostic agent. This work occupied two years of the research program.

3.3.2 Creating diagnostic points

It became clear while learning 4diac and understanding the architecture of IEC 61499 that traditional Built-In Test Points (BITs) used in typical embedded systems fault-finding were inadequate for our purposes (Smith & Lowenstein, 2009b; Hale & Bollas, 2018). 4diac supports dynamic re-configuration of function block connections to support redundancy and resilience (Zoitl, 2009). The ability to allow the agents to dynamically wire-in points where they could capture telemetry led to the design of the diagnostic point function blocks discussed in Chapter 6. The mechanism by which they rely on decision making from the agents ensures that these points-of-presence deep inside the function block application remain light-weight. This ensures that they impose minimal impact on the application while they are operating.

3.3.3 Agent beliefs and how they act

Much of the research in 2019 and 2020 focused on how agents could assimilate and interpret fault information. The agents proposed in this research needed to possess sufficient domain-specific knowledge of IEC 61499 to allow them interact in meaningful ways with the architecture. The thinking that drove the formulation of the belief structures proposed in Chapter 7 was refined by the first IECON paper published in 2020 (Dowdeswell, Sinha, Jarvis, Jarvis & MacDonell, 2020). How the agents would then use their skills, domain knowledge, and how they would manage their beliefs resulted in the work profiled in Chapter 8 and the second IECON 2020 conference paper (Dowdeswell, Sinha & MacDonell, 2020a).

3.3.4 The development of exemplar systems

The two exemplar systems presented in Chapters 9 and 10 were used to evaluate and refine the engine and the agents. This work explored both the diagnosable-by-design characteristics of the function block applications and how specific faults could be recognised using the diagnostic points and the actions of the agents.

The conclusions and future work discussed in Chapter 11 highlight a symmetry which emerged during this research. In traditional software development that employs Test-Driven Development, the tests created are primarily used only during the design and development stages (Y. Zhang, 2004; Jamro, 2014). In contrast, the diagnostic resources and techniques explored proved to be applicable across the entire life-cycle of an ICPS.

Chapter 4

Architecting the Engine

"It follows, therefore, that architects who have aimed at acquiring manual skill without scholarship have never been able to reach a position of authority to correspond to their pains, while those who relied only upon theories and scholarship were obviously hunting the shadow, not the substance. But those who have a thorough knowledge of both, like men armed at all points, have the sooner attained their object and carried authority with them."

"The Ten Books on Architecture", Vitruvius, circa 20 BC.

This chapter profiles the creation of the software architecture of the multi-agent based fault diagnostic engine that was referred to earlier as "*the engine*". Parts of this chapter were adapted for the MDPI journal Future Internet as the article *Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems*. It was published in the July 2021 Special Issue *Modern Trends in Multi-Agent Systems* (B. Dowdeswell, Sinha & MacDonell, 2021).

The purpose of this phase of the research is to propose and document a software architecture that will contain a point-of-reference to each of the software elements used to construct the engine. RQ4 posed in Section 1.1 asks how we should architect and develop an engine that is able to observe and interact with an IEC 61499 function block application. Coupled with that, RQ5 asks how the resultant architecture can be evaluated to determine how effective it is in meeting the diagnostic needs of its users. A second key deliverable from this research is a refinement to the practice of Model-Driven Development to incorporate the creation of early-stage design-time fault diagnostics (Dowdeswell, Sinha & MacDonell, 2020a).

Each IEC 61499 application may have differing diagnostic needs. These can include timing (Lindgren et al., 2015), interfacing with sensors and actuators (Black & Vyatkin, 2010), or complex algorithms that must cater for a wide range of functionality required of the ICPS (Zhabelova et al., 2016). Hence the specific fault-finding resources that are created should complement the individual needs and characteristics of each function block. IEC 61499 encourages encapsulation and reusability (Vyatkin, 2011). It is therefore sensible to package diagnostic resources for each block so that they are also encapsulated, fitting naturally into the libraries of mature components that development teams build up over time.

Software architectures are documented to justify and explain the features required, showing how each part will satisfy the goals and business needs of an organization. Bass et al. (2013) define the software architecture of a system as the bridge between these business goals and the application that fulfils them. While business goals are often abstract, the resultant architectural documentation must capture and propose a clear, unambiguous design upon which to build the application to satisfy those goals. The elements of the architecture maintain relations with other elements which mediate the interactions between the subsystems to enable the engine functionality. The architecture of a software-intensive computing system is therefore the collection of structures which describe how the system is, or will be, assembled (Bass et al., 2013). These constructs help designers reason about the system. A software architecture is also a set of decisions that are the result of technical, business and social influences (Herbsleb & Roberts, 2006).

Many designs flounder because they do not capture the full extent of what the users' originally wanted or needed. It is also vital to understand how the requirements constrain the scope implicitly. Failing to understand clearly what is *not needed* leads to the creation of features that may have seemed like a good idea at the time, but that did not originate from justifiable needs. While the early-stage requirements for the engine were drafted in the initial stages of this research, they were progressively refined by the findings that emerged from the literature review presented in Chapter 2. After exploring what creating a software architecture involves in more detail, this chapter then presents the high-level context view of the software

architecture proposed for the engine. Each of the sub-systems are then explored and evaluated in the subsequent chapters.

Modelling a system well and creating an excellent architectural description is hard. Van Heesch et al. (2012) comment that determining how well stakeholders understand the architecture that is presented to them is difficult to measure. Rigour is needed in the early design phases and in every subsequent phase of a product's life cycle. Rozanski and Woods suggest that an architectural model should be "*good-enough to achieve its purpose*" since architects often lack enough time to complete their models (Rozanski & Woods, 2019, page 159). The language and terminology employed, coupled with the way concepts are represented diagrammatically, should strive to make sure stakeholders clearly understand what is being proposed and why. In software architecture, as in civil construction, each subsequent story has to rest on solidly-cast, verifiable foundations (Richardson & Wolf, 1996).

Vitruvius's observation highlights the need for software architects to also be practitioners. The architects of the engine need a sufficient understanding of how faults are found in industrial control systems, elementary control theory, and how ICPS transducers work in practice. The literature review presented in Chapter 2 provided much of the background that was needed to understand these concepts and alternative ways of managing faults. That leads to a deeper understanding of how to realise both the core functionality of the engine and the qualities, such as resilience, performance, and scalability, that are implied in the early-stage requirements. These qualities are refered to as *Quality Attributes*. They provide a practical way to quantify the non-functional aspects of the system. Once these are understood, they will have a profound influence on the final architectural decisions made by the designers. A resilient software architecture should rest on well-proven patterns, recognised standards and well-defined Quality Attributes (ISO/IEC, 2011b).

4.1 Attribute-Driven Design and Requirements

Software-Intensive Systems are those complex solutions where software exerts significant influences over the design, construction, deployment and evolution of the system as a whole (Braun et al., 2014; Guessi, Cavalcante & Oliveira, 2015). Attribute-Driven Design (ADD) is a systematic methodology for designing the architecture of software-intensive systems (Wojcik et al., 2006). ADD relies on having sufficient, well-defined functional requirements before the architectural design phase can proceed. IEEE Standard 610 defines a requirement as a "*condition or capability needed by a user to achieve an objective*" (IEEE, 1991, page 65). The standard qualifies this definition further, asserting that a requirement should be a "*documented representation*" of a need. However, requirements for software-intensive systems such as the engine quickly become highly-detailed. This can lead to ambiguity when multiple interpretations of the same need are possible (C. Ribeiro & Berry, 2020; Sabriye & Zainon, 2018; Segal, 2017).

ADD is both a cyclic and an iterative process (Nord, McHale & Bachmann, 2010). It begins by ensuring that there are sufficient functional requirements, that the design and business constraints are well-understood, and that candidate quality attributes have been identified. Each element is then designed, documented and evaluated until all the features have been considered (Van Heesch et al., 2012). Figure 4.1 illustrates the steps in the ADD process that were followed to create the architecture for the engine. An additional prototyping phase, Step 6A, was added to the list of steps recommended by Wojcik et al. (2006). This was used to better understand aspects of a feature that needed empirical validation to clarify it sufficiently. This extra step proved invaluable. The findings it led to helped to better identify both functional and quality requirements, particularly those related to performance, ease-of-use, and scalability. Examples of these changes and refinements are explored in-context in the subsequent development chapters. Prototyping is also a key part of the Model-Driven Development with Diagnostics methodology employed to develop the ICPSs in Chapters 9 and 10.

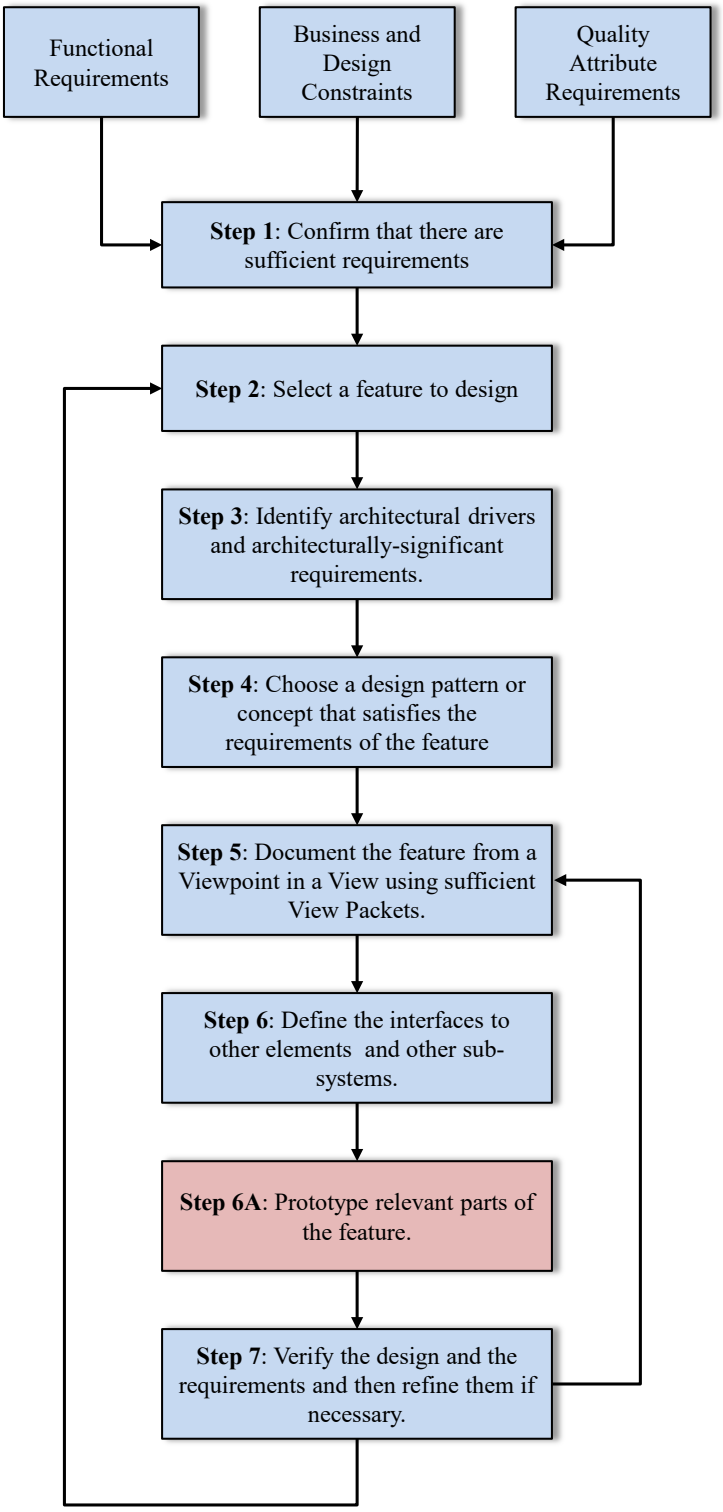

Figure 4.1: Attribute-Driven Design steps from Wojcik (2006) with additional Step 6A.

During the creation of the architecture, the desired quality attributes are often elucidated by working alongside stakeholders in structured Quality Attribute Workshops (QAWS) (Barbacci, Ellison, Lattanze, Stafford & Weinstock, 2003). The proposed architecture can then be evaluated through scenarios that were created using the Architectural Tradeoff Analysis Method (ATAM) (Kazman et al., 2000). The ATAM provides a step-by-step empirical validation of the architecture, created from both our experimental results and the Quality Attributes defined in the architecture document. The ATAM for the engine included the creation of a number of prototypes of architecturally significant aspects when the ATAM highlighted issues that needed to be explored more deeply. During the subsequent development of the engine, similar ATAM evaluations were performed and included in the chapters to determine how closely the component or subsystem satisfied the architectural requirements.

4.2 Choices for the architecture of agents

Agent-based approaches were uncovered during the literature review presented in Chapter 2. However, adopting an agent-based approach that employed the Belief-Desire-Intention (BDI) agent paradigm (Bratman, 1987) was an alternative that became more attractive, realistic, and practical after working with Associate Professor Dennis Jarvis in Sweden on his GORITE framework. This work coincided with the time spent performing the literature survey. GORITE proved to be a very productive environment to work in and it was possible to create a primitive, running agent quickly. Being Open-Source, GORITE provided a great learning environment. Further explanations of the BDI paradigm and agents are provided later in Chapter 5. The architecture presented in the rest of the current chapter demonstrates how it was possible to extend GORITE to build the Fault Diagnostic Engine around it. That consideration resulted in a refinement to RQ3 focus it on BDI rather than just general agent-based approaches. Hence, the results of RQ1 and RQ2 crystalised the choice of a model-based approach using an agent paradigm, and the work with Associate Professor Jarvis helped to show that BDI was a promising choice. That partially explains why the link between RQ2 and RQ3 is not immediately obvious.

4.3 The role of standards in software architecture

The influence standards have over architectural designs is profound. Metcalf and Miles (1994) describe standards as the exemplars that guide technical changes in industries. International standards draw together and codify best practices that then evolve and mature. For each architectural decision made during the design of the engine, support was sought from an appropriate international standard, in much in same the way a software designer draws on Software Design Patterns to build on solid foundations.

Three primary international standards were used to document the elements of the software architecture. The Unified Modelling Language (UML) (ISO, 2019c) and its subset SysML (ISO, 2019b) standardise the way architectural diagrams are drawn. To define the element ontology, software terms were used from the IEEE 610-1991 Standard Glossary of Software Engineering Terminology (IEEE, 1991).

ISO and IEEE have developed and codified a set of related architectural standards. IEEE 42010-2011 is a recommended practice for the activities of architectural design (ISO, 2011b). It is a conceptual framework that places many of the key concepts in-context with their roles in the design of software intensive systems. Terms such as *Views* and *Viewpoints* that are explained in later sections and *stakeholder concerns* are defined. The hierarchy it presents helps to the explain the relationships that link these entities. IEEE 42010-2011 is written in terms of "*shall*", "*should*", and "*may*". The standard does not define any conformance of systems, projects, organisations, processes, methods, or tools. Rather, its purpose is to provide a way of evaluating an architectural design for conformance to the recommended practices.

While functional requirements are goals that define what a feature must do, *Quality Attributes* are *quality* requirements. These are goals that specify characteristics such as how suitable, how rapid, how modifiable or how reliable that feature must be. ISO/IEC 25010:2011 (ISO/IEC, 2011a) defines and standardises categories of quality attributes so that they can be understood by stakeholders and used consistently by architects. Figure 4.2 illustrates the quality attribute categories provided by ISO 25010.

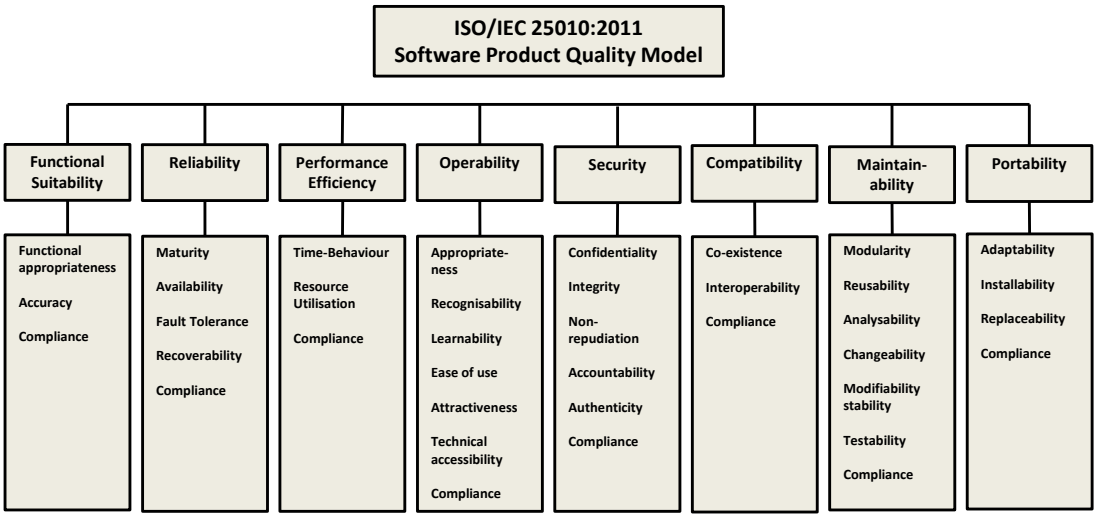

Figure 4.2: The ISO/IEC 25010:2011 Software Product Quality Model.

The IEC 61499 Function Block reference architecture is defined in three related standards. IEC 61499-1-2013 Part 1 defines the function block architecture in terms of four distinct models (IEC, 2013a). The IEC 61499 Architecture System Model defines a function block application as a collection of interconnected devices that communicate via a network. Individual function blocks are *devices*; each block encapsulates functionality and presents an interface that connects it to other blocks. The architecture also defines *resources*, the operational units which control the activities within each function block. A resource is an object instance of a function block type. Function Blocks are assembled according to an Application model that allows individual blocks to communicate via their interfaces, exchanging information by triggering events that cause the function block instance to process input data and deliver output data.

Two related standards support IEC 61499 Part 1. IEC 61499-2-2013 Part 2 specifies the compliance requirements for a function block development tool (IEC, 2013b). Examples of compliant IDEs discussed later in this research include 4diac (Strasser et al., 2008) and nxtCONTROL (nxtControl GmbH, 2016). The standard defines guidelines for the symbology used to draw function blocks and their interfaces. The ancillary Function Block Standard IEC 61499-4:2013 Part 4 specifies rules for compliance profiles (IEC, 2006). This part of the standard defines how compliant development tools can provide the ability to share their library

elements. The XML file formats used to store application definitions and individual function blocks are defined in this standard as Data Type Definitions (DTDs).

4.4 Documenting the architecture

Robust, well-documented architectures remain one of the most significant factors driving the delivery of successful projects (Brown & McDermid, 2007). However, deciding on an appropriate format for documenting a software architecture is not a trivial task. Clements et al. (2003) caution that all the effort put in by the architects is wasted if the documentation they produce cannot be understood later by the developers who have to build it. The Carnegie-Mellon *Views and Beyond* document template (Mellon, 2018) was used to capture the design of the engine in each of the steps of the ADD . The resulting software architecture document, referred to hereafter as "*the document*", is included in Appendix A. The document faithfully re-creates each of the sections of the Carnegie-Mellon template to illustrate the maturity and scope of the Views and Beyond approach (Bass et al., 2013).

Supporting this approach, the ADD process, coupled with the sound Carnegie-Mellon documentation framework, provides a way for each element of the design to be documented systematically. By describing the structure of each architecturally-significant feature, architectures provide the framework needed to reason about the elements that make up a system, how they interact and how well they meet the objectives demanded of them (Bass et al., 2013). Hence the role of a software architect is to propose an appropriate solution by blending different, well-proven architectural styles. Bass (2013) comments that an architecture that is not documented, and is therefore not available to others, is a significant risk in any commercial project. Young reinforces this: "*A design without specifications cannot be right or wrong, it can only be surprising!*" (Young, Boebert & Kain, 1988, page 18).

The first sections of the document in Appendix A provide both a background understanding and a context for the engine. The Document Roadmap on page 3 explains the structure of the Carnegie Mellon document, explaining the role of each section in the hierarchy. Section 1.2

outlines the purpose of the architectural document and begins to outline the context of the engine. The system stakeholders are identified followed by an explanation of each of the Viewpoints and what they are used for.

4.4.1 Structures, Elements and Relations

Structures are sets of *elements* that are held together by one or more *relations*. Doberkat (2003) explains that a relation links an input to an element to a corresponding output or a state. Relations can imply hierarchies and dependencies that define which modules provide support or inherit their capabilities from other elements. The elements of a structure interact with each other via *interfaces*. The interfaces of an element partition the structures into both private and public parts. Architecture is concerned with the public side of interfaces. However, a structure is only architectural if it supports reasoning about the system. One of the most common uses of a structure is to partition systems into implementation units, often referred to as *modules*. The relations between the elements help drive the reasoning about how to partition the system logically into modules. The details of the element's implementation reside in the private side of the interface and these are, by definition, not architectural. Private parts are invisible beyond the elements interface boundaries and are not relevant when describing the architecture. This concept of relations is particularly applicable to IEC 61499 function blocks since they are implemented as black-boxes with public interfaces that the agents and the engine interact with to capture telemetry. This leads to a practical separation of concerns which simplifies the creation of the architectural documentation: the design and implementation of the private parts of each element will be the responsibility of the developers later. The interface is therefore an important aspect of the architecture since it captures a set of expectations about *what* an element does without saying *how* it will do it.

4.4.2 Views, Viewpoints and View Packets

Applications such as the engine are too complex to be captured in any single diagram. Architects address this complexity by delivering a set of *Views*, structured descriptions of subsets of related

architectural features. Each View is presented from the perspective or *Viewpoint* of a particular stakeholder. Since stakeholders have different concerns and interests, a View illustrates how that part of the architecture addresses their needs. The rational behind each feature is detailed, providing diagrams that present all the elements described in that View (Van Heesch et al., 2012).

A View is represented by a collection of models, described in both formal and informal ways (ISO, 2011b). An *Architectural View* is therefore a representation of a coherent set of architectural elements, written for or by the system's stakeholders. It presents how the system will be or is constructed as a way of helping them to understand how the system will operate. A particular View is only produced if, and only if, it addresses the concerns of one or more important stakeholders (Bass et al., 2013). The core of any architectural document is therefore a cohesive set of Views; no single View or set of models can capture all the intricacies of a complex system. Figure 4.3 illustrates how in *Views and Beyond*, Bass et al. (2013) extend the 4+1 Views Model of Architecture proposed by Kruchten (Kruchten, 1995) to incorporate Rozanski and Woods (2011) seven distinct architectural Viewpoints. The stakeholders' Viewpoints are considered in the light of the Architecturally-Significant Requirements. Multiple *View Packets* can be used to further divide a View into separate sections, presenting aspects of the View in more depth. May (2005) describes a View Packet as the smallest piece of information a stakeholder requires, presented from a particular viewpoint.

Kruchen (1995) proposed the 4+1 view of software architecture that included four perspectives: Logical, Process, Physical and Development. Kruchten augments these with a fifth view that links these four perspectives together using Use Cases or *scenarios*. IEEE 42010-2011 extended this further, introducing the concept of a *Viewpoint* (ISO, 2011b). Viewpoints are analogous to software design patterns. The standard prescribes the recommended practices for documenting a software architecture. It defines a Viewpoint as a means of constructing a View that is independent of a particular system. Viewpoints must be specified by stakeholders since they detail the architectural aspects of a system that they are concerned with. Viewpoints are templates that encapsulate the recommended models, languages and techniques for creating

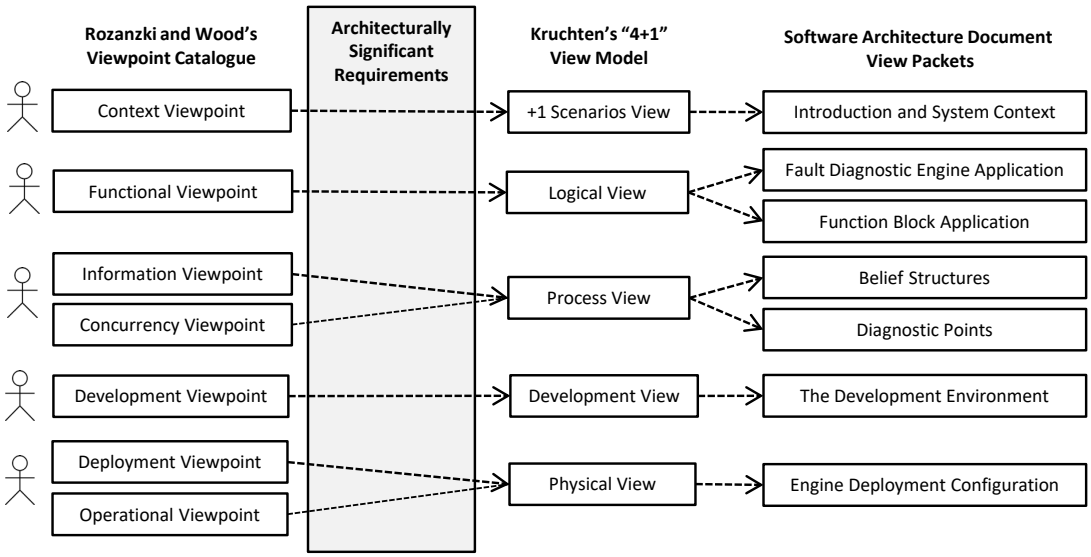

Figure 4.3: Viewpoints, ASRs, and Views used in the Software Architecture Document.

a particular View. Rozanski and Wood (2011) define a catalogue of six core viewpoints to complement Kruchten's Views. The Viewpoint Catalogue of Rozanski and Woods is explored in more detail in Section 1.5 of the document. Figure 4.3 shows the Views that were chosen to document the individual sections of the document detailed on the right of the figure. These include:

Context Viewpoints describe the relationships, the dependencies and interactions between the system and the environment that includes other systems, people and external entities. This viewpoint is different from the other viewpoints since it concentrates on external aspects of the system, not its internal structure and functionality. This viewpoint is useful for capturing aspects and dependencies that are important but which reside outside of the core system focus.

Functional Viewpoints describe how the system functions, detailing the elements that make up the solution and how they relate to each other. Functional viewpoints capture the shape of the system and are often the first views that are created. Functional diagrams address strategic issues about how the system will be put together, information flow, runtime performance and ultimately how it will be deployed. Functional Viewpoints are often the best place to document quality attributes.

Information Viewpoints describe how information is captured and flows between structures

and elements. They also detail static data storage concerns.

Concurrency Viewpoints are similar to Functional Viewpoints but focus solely on deeper explanations of how the solution will manage concurrency aspects. How will functional elements synchronise simultaneous activities and which operations must honor task precedence? This viewpoint uses diagrams that show process and thread structures as well as models of interprocess communication.

Development Viewpoints present aspects of the architecture of concern to developers, implementation specialists and long-term support teams. Development Viewpoints augment the other architectural Viewpoints, providing more detail about the internal constructs that the developers will go on to create. Localising these details within this Viewpoint ensures that unnecessary detail that may be of no interest to other stakeholders does not confuse their own Viewpoints.

Deployment Viewpoints describe the environment that the system will be deployed into. Like the Development Viewpoints, this perspective also hides unnecessary detail from other Viewpoints. Information about runtime environments, upgrade and roll-out dependencies fit well into this Viewpoint.

Operational Viewpoints describe the environment that the system will be deployed into. Like the Development Viewpoints, this perspective also hides unnecessary detail from other Viewpoints.

SysML diagrams were adopted in the document as a way of presenting the elements of an architectural feature in a consistent way. SysML is a modeling language defined in the ISO 19514 standard (ISO, 2019b). SysML is an extension of the Unified Modeling Language (UML) defined in the ISO 19501 standard (ISO, 2019c). SysML uses the same syntax as UML and extends the range of diagrams to add Activity, Requirement and Block diagram types which are ideal for describing systems-of-systems architectures. UML and SysML both encourage systematic naming conventions for elements and the diagrams which contain them. This makes it easier to build the Views and Beyond Element Catalogs that describe each View. For every View in the Kruchten 4+1 model, there are UML and SysML diagrams that are appropriate to document the elements in that View.

4.4.3 Identifying Stakeholders

Stakeholders are the people who either have a degree of ownership of the system or who will later be affected by its design, implementation and operation (Rozanski & Woods, 2011).The stakeholders have an interest or "stake" in the successful creation of the engine. Documenting a software architecture is the process of creating a depiction of the proposed system that will help the stakeholders understand it enough to answer their concerns about how it will behave. They have differing concerns that are addressed during the architectural design phase by presenting appropriate documentation, showing how the proposed architectural features will address their individual needs. The primary stakeholders identified in Section 1.4 of the Software Architecture Document include:

Company and Investor representatives who are the ultimate owners of the intellectual property of solutions such as the engine. They provide financial and management oversight for the creation of the engine. In their context, the engine is regarded as a solution to one of their business problems. While they are highly-relevant to the long-term viability of the engine, they are not discussed further in this research.

Software Engineers, who will specify diagnostic tests for the function block application they are designing. In the proposed Model-Driven Development with Diagnostics methodology, diagnostic resources are created early by software engineers and live in the hierarchy of the primary requirements. Like other artifacts, they are created at appropriate points during the design and development phases.

Test Engineers, who provide support for the application during early-stage testing and later during go-live commissioning. The set of diagnostic tests they employ may be focussed on differing needs. Examples include longer duration burn-in trials that would not necessarily be performed by Software Engineers.

Maintenance Engineers, who provide long-term support for the system. Their immediate concern is to quickly identify what has gone wrong with a production ICPS and restore full functionality. Their investigations may also influence later maintenance or design changes as the product line evolves. At any time during the products lifetime, especially if it fails, the

whole system or part of it can be switched to diagnostic mode. The engine uses pre-prepared diagnostic resources to identy faults more quickly by approaching the problem systematically. **Developers** who will implement and refine the software architecture while developing the engine.

Apart from the first group of Company and Investor representatives, all of these stakeholders are people who would use the engine as a tool as part of their development, testing, or maintenance activities.

4.4.4 Identifying Architecturally-Significant Requirements

Step 3 in Figure 4.1 is a critical step in an ADD. Architects seek to understand the needs of their stakeholders by first identifying those functional requirements that are *architecturally-significant* (Clements et al., 2003). Architecturally-Significant Requirements (ASRs) describe the functionality that will have a significant influence on the architectural design choices made. They influence or constrain the decisions an architect can make (Van Heesch et al., 2012). Chen et al. (2012) outline an approach for identifying ASRs among the requirements. Architecturally significant requirements express quality-based needs. Chen gives the example of a functional requirement for an application to display temperatures in both Fahrenheit and Celsius. In contrast, the quality requirement for the uptime of the system to be "99.999 percent" embodies the ISO 25010 Quality Attribute of Availability (ISO/IEC, 2011b). Fulfilling the requirement for displaying a reading in degrees Fahrenheit and Celsius requires additional software functions to be written. In contrast, achieving the quality characteristic of "High Application Availability" requires architectural decisions regarding fail-over to alternative hardware or the choice of different sensors with higher reliability.

ASRs often affect multiple parts of the system rather than just the functionality they are describing. Bass et al. (2013) comment that ASRs are often architecturally significant because of the high cost of changing that feature later. Requirements that are *Strict* limit the architectural choices available while *Trade-offs* are the compromises that the designer must make to balance one desired property or quality against another. Trade-off points occur where no single

architectural approach satisfies all the requirements of that feature. Table 4.1 details two of the ASRs for the engine categorized using Chen's criteria. Neither of the two examples break assumptions made during the original requirements elicitation. However, the requirement to be able to interact with multiple, distributed IEC 61499 applications has a wide impact on how information is exchanged and collated from different parts of the application. It also requires trade-offs that are related to the design of the diagnostic points and how they gather telemetry. This requirement is also deemed to be difficult to achieve, leading to the decision that this requirement is architecturally-significant. In contrast, the performance of the diagnostic points is critical but does not have a wide-impact; the diagnostic points are self-contained and only communicate through the *NIOserver* component. However the development of the diagnostic points is complex, leading to them also being classified as architecturally-significant.

4.4.5 The Context of the Engine

After the introductory sections of the document have provided an outline of the architecture, the individual Views are presented starting with the System Context View in Section 3.1. This corresponds to the Kruchten +1 Scenario View that helps to place all of the other Views in their proper context. The concept of this View can be thought of as walking around the engine and looking at it from a different angles, understanding each feature as it is presented from differing perspectives. The System Context View is usually one of the first views created by an architect. It is a high-level view, designed to present the purpose and business drivers that underpin the

Table 4.1: Chen et al.(2013) criteria for being Architecturally Significant.

ID	Requirement	Wide impact?	Requires Trade-offs?	Strict?	Breaks Assumptions?	Difficult to achieve?	Architecturally Significant?
1	The agents must be able to interact with multiple, distributed parts of the system that is under diagnosis.	Yes	Yes	No	No	Yes	Yes
2	The operation of the Diagnostic Points shall not degrade the performance of the system under diagnosis by more than 5%.	No	Yes	Yes	No	Yes	Yes

architectural decisions made about the engine. It is also the View which often undergoes the most refinement while the other views are being documented later. Woods and Rozanski (2009) explain that the Context View helps to define external dependencies of the architecture. The view complements the other views which focus more on documenting consistent and complete internal interfaces between subsystems.

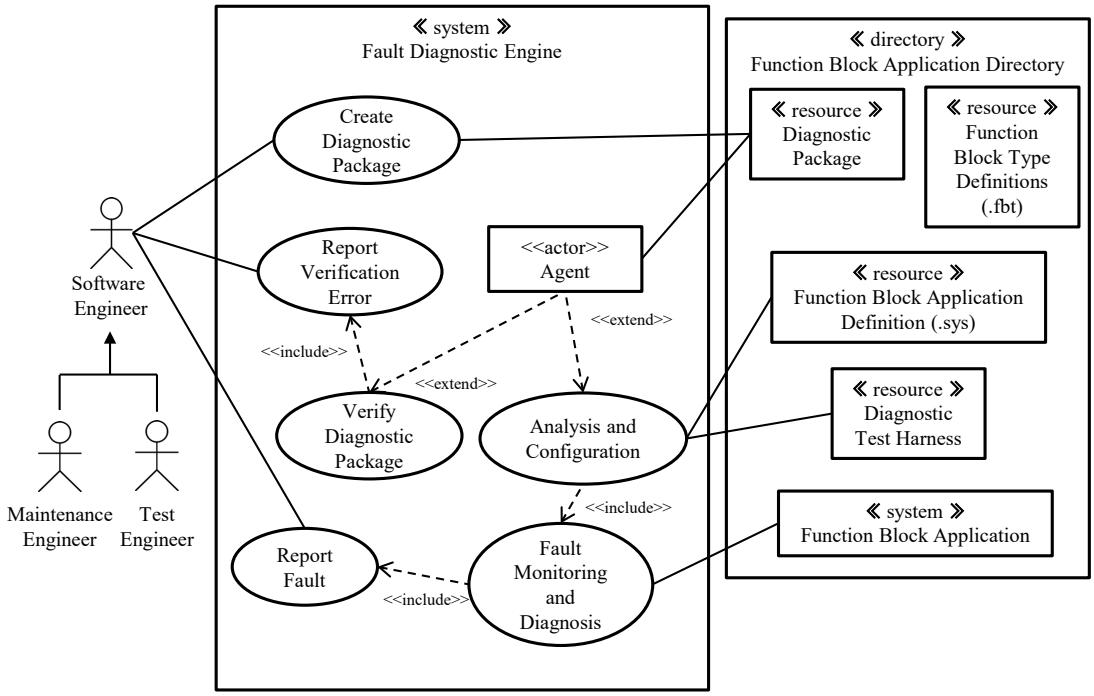

Figure 4.4: The System Context View of the Fault Diagnostic Engine.

The Context View addresses how the architecture will address the needs of the stakeholders who either use or will develop the engine. As explained in Section 4.4.3, stakeholders have differing concerns so alternative view packets are needed to address their individual needs in more detail. For example, Software Engineers are concerned with operational features of the engine and the agents while they use the system to help them develop function block applications. Their needs include interactive fault-finding sessions during early-stage development. In contrast, Maintenance Engineers use the engine, configured with different agent goals, for longer-term fault monitoring and management. Figure 4.4 illustrates the activities the agents can perform and the subsystems the engine is constructed from in this context.

The engine is a stand-alone system that interacts with the separate function block application,

detailed on the right of the diagram. Design resources created in the IDE used during the creation of the function block application are available to the agents. They use this structural information to configure the telemetry connections they need to interact with the function blocks during fault-finding. Each of the subsequent views details the subsystems shown in this diagram in more detail from differing viewpoint perspectives. Constraints that drove architectural decisions are discussed within each context. In this way, Views and Beyond creates a framework in which each view or viewpacket can provide a description of each architectural element the subsystem relies on, the interfaces they provide, and the relations they maintain.

4.4.6 The Logical View of the Engine

Figure 4.5 presents the Logical View of the engine from Rozanski and Woods Functional Viewpoint. A layered architectural style was chosen since the self-contained GORITE Team Manager subsystem operates in a hierarchy within the GORITE framework. Below that is the the primary operational interface to the agents operating in a lower layer. There they hunt for and manage faults by reaching down into the layer where the function block application runs. This layered approach also reflects the separation of concerns regarding where the levels of responsibility reside in the engine. Unlike previous approaches, the agents dynamically monitor systems. They do this after reconfiguring the control layer from the agents' execution layer, interacting with the light-weight diagnostic points. As noted earlier, this represents a significant enhancement of the interaction scheme employed in Kalachev et al. (2018), Jarvis et al. (2018), and Christensen (2017). In their approaches, execution layer agents were responsible for the function block controlled actions. This approach was chosen to satisfy the constraints that the presence of diagnostic test points should not adversely impact the normal operation of the individual function blocks.

The refinement of the agent subsystem is discussed further in Chapter 5. This was driven by later evaluations that identified the need to multi-thread agents to better manage their scheduling and resources. The engines communication subsystem operates in the agent layer, providing the interface to the separate function block application layer. Sub-applications of the function block

application can run on their own hardware platforms as distributed device instances in this lower layer. GORITE operates in the first layer, coordinating the team activity by providing agents with sets of *goals* to follow semi-autonomously. GORITE provides a number of team goal model options, some of which indicate the logical order that these goals should be executed in. The first prototype detailed in Chapter 5 implemented a Sequential Goal Model where each goal follows on from the next logically. The agents operate in their own layer, monitored while they operate and signalling back to the Team Manager Agent layer when they have either completed or failed their current goal. Goals are then rescheduled or re-assigned as needed. This delegation allows for semi-autonomous behavior by the agents as they evaluate fault evidence and make decisions about what the status of their goal is currently.

However, GORITE does not provide ready-made agents; agents have to be custom-designed for their target domain while remaining compliant with the framework architecture. The architecture of the agent has to equip them with the skills needed to perform those goals which, for them, are domain-specific fault finding techniques applicable to IEC 61499. Skills and belief structures are discussed later in Chapter 7. Note that the primary purpose of the document is to detail the custom architecture designed for the agents. Hence, details of the GORITE architecture are discussed only when that helps to clarify the design decisions made about the

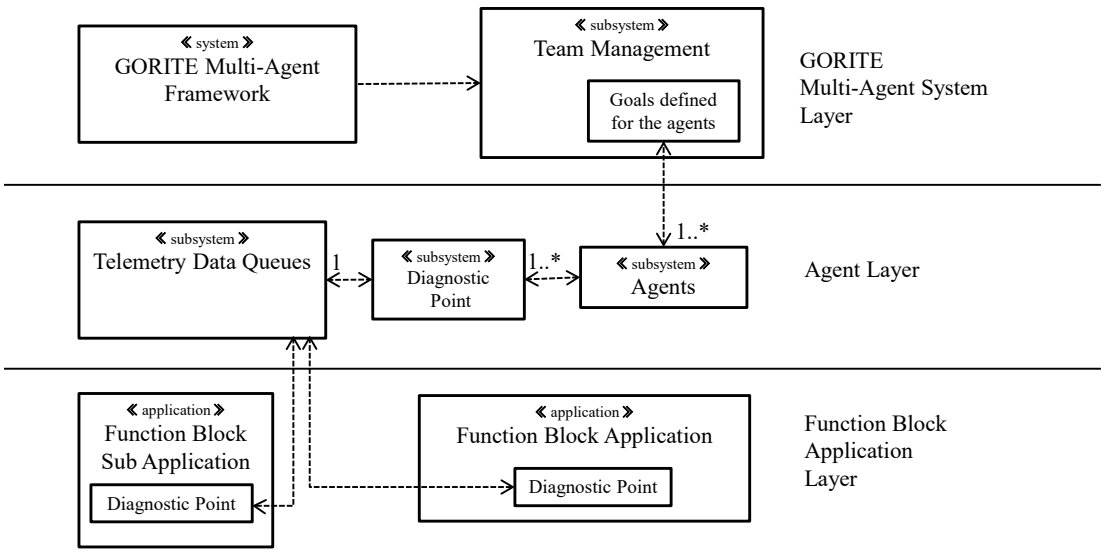

Figure 4.5: The layered architectural style of the engine.

agent's architecture.

Agents are responsible for reaching into function block applications to capture telemetry. Since the engine operates on its own platform separate from the target function block application hardware, the agents are operating asynchronously. All data is exchanged via bi-directional TCP network queues where there is one discrete First-In-First-Out (FIFO) packet queue for each Diagnostic Point. The architecture and operation of Diagnostic Points is discussed in-depth in Chapter 6.

4.5 Evaluating software architectures

Software Engineering, and Software Architecture in particular, are reflective disciplines (Hazzan, 2002). After creating a well-documented architecture, the design decisions made should then be reviewed and evaluated carefully to ensure they will meet the quality criteria identified for them. Conclusions and recommendations from the evaluation are used to refine the architecture before the application is constructed. The Architectural Tradeoff Analysis Method (ATAM) (Kazman et al., 2000) frames these evaluations in a set of scenarios that examine significant features of the architecture. The goal of the ATAM is to understand the consequences of architectural decisions. Kazman et al. define *risks* as architectural decisions which are potentially problematic. In contrast, they describe *non-risks* as good design decisions. A *sensitivity point* is an aspect of the architecture where one or more quality attributes are highly-correlated with the architectural choices made. Changes to that part of the architecture might compromise other qualities with undesirable consequences. *Trade-offs* are decisions that balance both the risks and sensitivities inherent in an element. They often occur in features that have multiple sensitivity points. ATAM *scenarios* propose ways that an aspect of the architecture can be evaluated in the light of the qualities it embodies, the known risks, the sensitivities, and the trade-offs made. By following a structured process, the results are consistent and repeatable: architectural evaluation is an iterative process.

Barcelos and Travassos comment that a software architecture document visualizes the

application with a high degree of abstraction (Barcelos & Travassos, 2006). They suggest that the ATAM process is highly subjective and that there is a need to identify more concrete implications of the architectural decisions made. This helps to prevent defects propagating from the design down into the application. Bass et al. note that rectifying defects early is invariably less costly than remediation work later (Bass et al., 2013). However, Reijonen et al. were concerned that there is a steep learning curve between understanding the theory of the ATAM and then applying it to obtain meaningful outcomes (Reijonen, Koskinen & Haikala, 2010). While scenarios are widely used in business, its application in software development is less evident. Scenarios propose situations that would exercise parts of the architecture and are driven by questions that probe the implications of an architectural decision. Information from the scenarios also influences the scope of the diagnostic tests. Reijonen et al. also comment that the original ATAM approach does not include preparation or post-work. This was addressed when evaluating the engine after the creation of the first draft of the document by prototyping relevant parts of the architecture to investigate issues uncovered during the ATAM.

4.5.1 Constructing the ATAM Utility Tree

Utility Trees are used during the ATAM to present quality attributes in a hierarchical tree structure that highlights the importance and risk of each attribute. Figure 4.6 shows part of the Utility Tree constructed during our ATAM showing quality attributes derived from ISO 25010. The main branch Functional Suitability leads to the sub-qualities identified for the engine in Section 2.1.3 of the document. Each of the sections give examples that evaluate the Views against these qualities, identifying the risks, sensitivities, and trade-offs, and proposing scenarios to evaluate suggested changes.

4.5.2 ISO 4.2.2 Performance Efficiency

Efficiency for the engine is defined in terms of how well it uses the resources of the platform it is deployed on. QAS 2 in the document requires that the agents must be able to operate faster

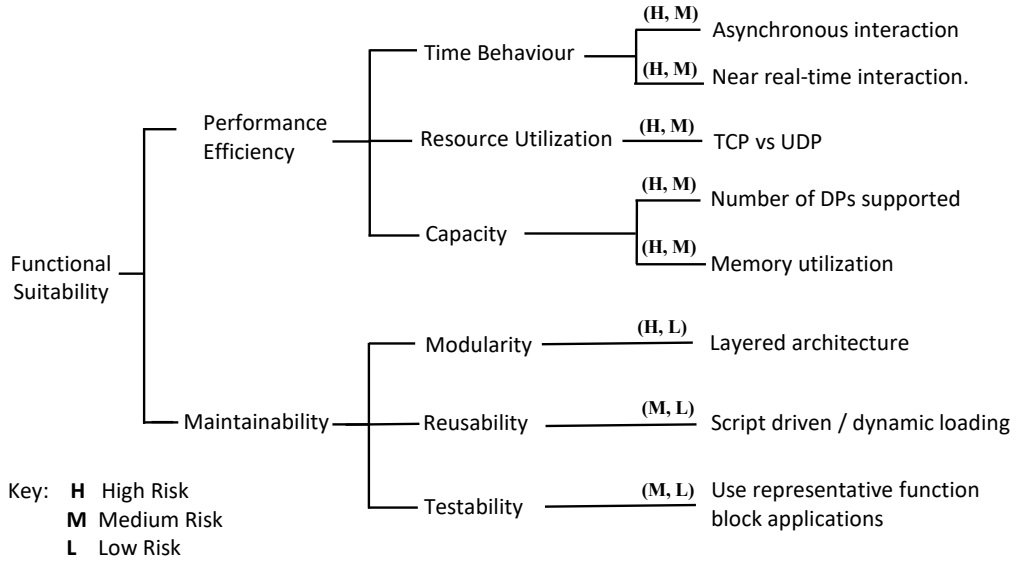

Figure 4.6: The ATAM Utility Tree for the Engine

than the function block application. While the diagnostic point only has to mediate the flow of data to and from the engine, the agent also has to analyze the data streams from multiple DPs and execute diagnostic strategies preemptively. QAS 5 requires the engine to be able to be run natively on the same hardware as the function block application. However that is unlikely to be a viable option in most cases. Separate and redundant fault diagnostic engines that operate reliably while other systems are failing are the norm in avionic and aerospace control systems (Benowitz, 2014; Goupil et al., 2014). Where a large ICPS are being diagnosed, the engine must provide sufficient resources to ensure that the agents can perform all tasks within the required time window.

4.5.3 ISO 4.2.8 Portability

The engine uses Java and C++ as development languages to address the Portability sub-qualities of Adaptability and Installability. The GORITE engine and the agents are implemented in Java to ensure they can operate on desktop, server and embedded system platforms. The diagnostic points are custom Service Interface Function Blocks implemented in C++ for compatibility with the FORTE function block runtime (4diac, 2019). 4diac diagnostic points could later be

converted to C# to run with the nxtCONTROL runtime (nxtControl GmbH, 2016). The engine is considered low risk for these qualities.

4.5.4 ISO 4.2.5 Reliability

Reliability is also addressed by running the engine on its own hardware. QAS 4 requires the engine to operate reliably even when the function block application is failing or has failed. The sub-category of Recoverability is addressed by the diagnostic points ability to re-connect automatically if they lose their connection to the engine. Data is exchanged asynchronously between the agent and the diagnostic points it is capturing data from or sending test data to. A First-In, First-Out (FIFO) queue structure is used by the engine. Transmission Control Protocol (TCP) (Agency, 1981) is a connection-oriented protocol that provides guaranteed packet delivery and error correction. In contrast, the User Datagram Protocol (UDP) (DARPA, 1980) is a raw, connectionless protocol that provides no error checking or guarantee of delivery. A Non-Blocking TCP socket protocol client was implemented rather than use one that employed UDP. The quality attribute trade-offs of reliability and performance were considered between the extra overhead of TCP versus the simplicity of packet management that guaranteed reliable delivery provided. TCP does not require additional packet buffering for the FIFO queue manager since only complete packets are received. This also simplified the client C++ code for the DP function blocks. Further scenario testing is recommended with a large number of diagnostic points to evaluate the latency of high-traffic environments.

4.5.5 ISO 4.2.6 Security

Security is partially addressed by running the engine on its own platform. However in the current design, the data exchanged between the diagnostic points and the agents is not encrypted. This presents risks when the diagnostic harness is deployed on a function block application that is operating in a production environment rather than in design or testing. diagnostic points are TCP clients and the data exchange packet structure is available publicly. This poses problems if the engine is impersonated by a rogue application. In that scenario, it would be possible

to inject false data into the function block application and route it through to other function blocks that are not gated at the time. Tanveer et al. (Tanveer, Sinha & MacDonell, 2018) propose a secure-by-design approach that implements encrypted data streams using custom function blocks and security keys that is applicable to this architecture. Future work on the engine should consider the integration of the approach used by Tanver et al. directly into the diagnostic points since as edge devices that create a pathway directly into the application, they present security risks that should be addressed before the engine is deployed in an industrial environment.

4.6 Conclusions and planning the next phase

This chapter has explored the requirements of the engine as well as the principles and techniques that were used to create a well-documented software architecture, one that can be shared amongst stakeholders.

In his book that illustrates the use of SysML diagrams, Delligatti cautions both architects and stakeholders not to succumb to what he calls "*The Myth of Model-Based Software Engineering*" (Delligatti, 2013, page 9). The architectural description of a system, expressed in the models and documented in a software architecture document, exists separately from the system itself. A well-implemented system should be a faithful representation of a well-designed software architecture. However, the architecture alone cannot make any claims that the system as delivered will meet the stakeholders' needs perfectly. Deligatti explains that stakeholders often harbour the concept that their system is somehow auto-generated from the documentation. They make an a priori assumption that time spent creating high-quality models and excellent documentation is in Delligatti's words:

"... the Easy Button: You push it and good things pop out. To put this more concretely, they believe, incorrectly, that Model-Based Systems Engineering makes every engineering task easier and reduces cost at every point in the life cycle." (Delligatti, 2013, page 10)

Modelling a system well and creating an excellent architectural description is hard. Rigour is needed in the early design phases and in every subsequent phase of a product's life cycle.

A sound architecture addresses risks and proposes the use of tried-and-proven architectural patterns, supported by evidence from international standards. A well-documented architecture supports its recommendations by showing how each decision is underpinned by appropriate Quality Attributes. Finally, the language and terminology employed, coupled with the way concepts are represented diagrammatically, should strive to make sure stakeholders understand clearly what is being proposed and why. In software architecture, as in civil construction, each subsequent story has to rest on solidly-cast foundations.

Chapter 5

Crafting Agents

"To you, a robot is a robot ... but you haven't worked with them, you don't know them. They're a cleaner, better breed than we are."

Dr Susan Calvin speaking in the introduction to "*I, Robot*", Isaac Asimov (1950)

By the end of the literature review that was presented in Chapter 2, it was evident that agents were considered to be a promising way to implement the adaptive, procedural tasks needed to investigate faults. This chapter now addresses aspects of RQ2, which asks which needs of IEC 61499 can be met by agent-based approaches. It also explores RQ1, examining ways in which an agent can apply the most promising Model-based fault diagnostic techniques uncovered in the literature. It also presents the Belief-Desire-Intention agent paradigm in more detail to explore aspects of RQ3, which seeks to understand what aspects of this approach to agent interaction is applicable to fault-finding tasks (Bratman, 1987; Wooldridge, 2009).

The papers showed how agents had been employed for collecting data and communicating the results (Sankavaram et al., 2013; Milis et al., 2016). Other studies suggested ways to consider Model-Based approaches alongside AI techniques or a hybrid blending that combined appropriate aspects of both (Mohrle et al., 2015). The Design Science plan detailed in Chapter 3 provided the scope to prototype an agent and then experiment with alternative fault-finding methods. During these experiments, the model-invalidation techniques of Kassmeyer et al. (2016), Provan (2014), and Lee et al. (2017) suggested ways to establish beliefs that the agents can use

to manage fault information.

This chapter begins by outlining the Belief-Desire-Intention paradigm that underlies the GORITE Multi-Agent Framework chosen to implement the engine (Jarvis et al., 2013). The development of a prototype agent based on the architecture proposed in Chapter 4 is then presented and evaluated. This first experimental phase created the foundation on which to build the beliefs and interaction skills infrastructure needed for fault-finding. These are the core structures that enable an agent to support engineers who are employing a model-driven development with diagnostics methodology. The evaluation of the architecture led to refinements to address quality attributes that the first prototype did not satisfy well enough. This iterative refinement approach, informed by quality requirements and empirical measures from the software architecture, led to a more robust agent subsystem for the engine.

5.1 Situated, intentional agents

Agents were originally conceived as a way of computerizing tasks and processes in the way that humans typically might address them. Like humans, agents are able to adapt their strategies autonomously and work collaboratively to cope with changes in the environment they perceive. Agents have emerged as a promising approach to creating large distributed and self-adaptive systems including ICPS (Calvaresi, Marinoni, Sturm, Schumacher & Buttazzo, 2017; Braberman, D'Ippolito, Kramer, Sykes & Uchitel, 2015).

The first concepts of agent-like behaviour and cognition were discussed in the context of the A.I. research being performed in the mid 1950s. Wiener's work on cybernetics addressed many of the control aspects of computer intelligence while pointing to how reasoning might be implemented (Wiener, 1948). McCarthy's pioneering work on artificial intelligence (1955) led to the first Procedural Reasoning Systems (PRSs) (M. P. Georgeff & Lansky, 1987; M. Georgeff & Ingrand, 1989). These were the earliest approaches to creating real-time, expert-like systems that could demonstrate reasoning (Hewitt, 1971). However, it was clear that many aspects of A.I. led to intractable computational problems. Hewitt revisited this early research in 2008,

examining how Church's early work on logic did not lead to the creation of practical A.I. entities that could perform useful work in a reasonable time-frame (Hewitt, 2008).

In the context of intelligent reasoning, it is useful to draw analogies about how humans and computers differ in the way they might perform the same tasks. Bratman (1988) and Wooldridge (2009) provided an early background to intentional agents who, unlike many business computer systems, are situated in an environment they can interact with by themselves when they determine that they need to. Bratman introduced the concept of Beliefs, Desires and Intentions (BDI), describing how agents form beliefs about the environment they are operating in and perceiving. Reasoning about those beliefs leads to the formation of desires to achieve goals that are fulfilled by pursuing intentional tasks. While ICPS strive to be deterministic, the world they operate in is not. Agents are one approach to working with devices that must bridge that temporal divide. On the cyber, computational side of an ICPS, time does not matter. A computer will execute an algorithm as fast as it can, but it does not embody any implicit sense of urgency. However, on the physical side in their environment, time matters for an ICPS. Tasks such as stopping when a human comes dangerously close to a robot is time-critical and cannot be ignored.

Agency is an abstract principle that asserts that agents are entities that are capable of acting by themselves (Griss, 2001). Acting on behalf of another entity is also a fundamental aspect of agency (Green, Hurst & Nangle, 1997). Garcia et al. define a practical implementation of agency in terms of the agent's state, its intrinsic properties, and the roles it is capable of performing. They also comment that the emergence of powerful networking revitalized the investigation of agents as a way to engineer complex, self-adaptive multi-entity systems, leading to the creation of Multi-Agent Systems (Garcia, Silva, Chavez & Lucena, 2002). Wooldridge and Jennings (1995) further clarify this concept of agency, defining agents as software-based entities with hardware components that exhibit one or more of the following characteristics:

- **Perception and reactivity.** ICPS and agents both require the ability to perceive their environment accurately (Janasak & Beshears, 2007; Klar & Huhn, 2012). Based on their understanding of what is going on around them, agents determine how they should

react or respond. Agents are designed to be able to decide on a course of action by choosing from a number of pre-defined alternatives. The diagnostic approaches in the literature review papers profiled agents that were concerned primarily with observing behaviors and ascertaining the current state of the system. Examples were given of agents implementing both Model-Based (Provan, 2014; Milis et al., 2016) and Model-Free Artificial Intelligence diagnostics (Wu et al., 2017; Sankavaram et al., 2013). Where machine learning informs their understanding, they may adopt approaches that were not manually prescribed for them.

- **Autonomy.** Agents are able to perform tasks working independently, often without direct intervention or supervision by humans. They manage their own internal state and have control over their decisions and resulting actions.
- **Social ability and interactivity.** Agents can work co-operatively, sharing goals and tasks to achieve outcomes that a single agent might not be able to accomplish alone. They are capable of inter-agent communication as well as exchanging information with humans (Poslad, 2007). Garcia et al. describe agents as "*complex objects with an attitude*" (Garcia et al., 2002, page 3). In industrial contexts, agents can be considered to be "*...autonomous, problem-solving, and goal-driven computational entities with social abilities*" (Leitão & Karnouskos, 2015, page 3).
- **Knowledge management**. While general agents exist, many implementations are highly-specialised, addressing tasks that require domain-specific knowledge and skills. Modest and Thielecke (2016) profile extensions to the V-Model to show how to integrate agent-based diagnostic functionality. Their approach encapsulates avionics domain knowledge to create standard libraries of resources and techniques. These are captured in failure-indicator matrices. They envisage the use of agent techniques to create a set of standard functions and interfaces that can be deployed as rule-based expert systems.

While agents are often created using object-oriented design patterns, there is a distinct difference in the way agents interact with other components in systems-of-systems. Objects implement methods and functions and cannot refuse to execute when called by a component

that holds a reference to an instance of that object. In contrast, an agent operates in a contract or agreement with other subsystems. When requested to perform a task or provide a service, an agent will evaluate its current priorities and only comply if it can (Shehory, 1998; Wooldridge, 2009).

5.1.1 Beliefs-Desire-Intention execution models

Bratman (1987) was originally concerned with how agents deliberate. Georgeff et al. (1989) extended the BDI concepts, replacing the implementation of desires with goals that are mutually-consistent. In Georgeff and Ingrams's model, when an agent has a desire that it commits to, it realizes its intention via a *goal*. Goals specify a course of action. Algorithm 1 illustrates the traditional BDI execution (Jarvis et al., 2013):

while *true* **do**
    wait for next goal event;
    based on the current beliefs, select a plan to achieve the goal;
    execute the selected plan;
    update beliefs;
end

Algorithm 1: Traditional Belief Desire Intention Execution Model.

However, goals are transitory, leading to conflicts with the traditional BDI paradigm. Implementations of BDI models are often computationally intractable (Schut, Wooldridge & Parsons, 2002). Rao and Georgeff (1995) proposed ways to get around this difficulty including limiting belief models to only encompass the current state of the environment and representing intentions only in plans. Georgeff, Pell, and Pollack (1998) later commented that BDI remains the best-known and most studied model of practical reasoning. However, while BDI is concerned with how an agent manages its beliefs and actions, it gives no consideration to how multiple agents should interact or collaborate. This aspect was addressed later in Section 5.2 since the architecture of the engine is designed to support multiple agents who take responsibility for different aspects of fault investigations.

The strength of the BDI paradigm is that it more closely models human reasoning in the

real world where rational beliefs should influence actions that enable an entity to complete an intentional task. Georgeff proposes that the focus of agent research should be on environments that are chaotic, where the observable state of the region is dynamic and uncertain (Georgeff et al., 1998). Systems that try to operate in such environments are only able to maintain a limited local view of their region; a model of their entire world would be computationally intractable. All practical systems are resource-bounded; they have finite computational capabilities and resources. Hence an agent that has to operate successfully under such conditions requires an operating paradigm such as BDI that addresses these concerns in practical ways. Later work in Chapters 7 and 8 demonstrate how three distinct belief structures were created. One of these became the primary unit of information storage about faults both maintained by and exchanged between the agents.

By 2019, Gartner were still proposing that agents were one of the top ten important technologies driving autonomous systems development (Gartner, 2019). Marik et al. discuss how Service-Oriented Architectures today build upon many of the original agent principles (Marik, Gorodetsky & Skobelev, 2020). While BDI is still the predominant agent paradigm, a number of alternative agent frameworks have emerged since Bratman and Wooldridge's original work. Jones and Wray (2006) provide a comparison of two significant multi-agent frameworks, GOMS (Card, Newell & Moran, 1983) and Soar (Laird, Newell & Rosenbloom, 1987; Wray & Jones, 2005). Both of these architectures approach agent interactions from a computational intelligence perspective. By 2005, more than eighty different agent platforms existed (Ganzha & Jain, 2012).

5.1.2 The GORITE Multi-Agent Framework

Later work by Rönnquist (2007) and Jarvis (2013) led to the development of the GORITE (Goal ORIented TEams) Multi-Agent Framework. GORITE is written in Java and is available to researchers via an Open Source license. GORITE implements intentions as explicit activities, making no distinction between goals and plans. This approach simplifies the specification of goals that can be assigned to an agent who then executes them independently. Jarvis et al. argue

that the main focus of multi-agent research is to extend existing BDI frameworks to address limitations in their ability to reason about intentions and actions (Jarvis et al., 2013, page 4). However, while BDI is concerned with how an agent manages its beliefs and actions, it gives no consideration to how multiple agents should interact or collaborate. This aspect was addressed later in Section 5.2 since the architecture of the engine is designed to support multiple agents who take responsibility for different aspects of fault investigations.

During a research posting during 2017 at the Luleå Institute of Technology in Sweden, there was an opportunity to work alongside one of the co-creators of GORITE, Associate Professor Dennis Jarvis, exploring the GORITE framework under his guidance. During that time, a prototype agent that could execute simple fault-finding goals was created and evaluated. An example physical environment was modelled that simulated the concrete blocks in the wall of a hydro-electric storage dam. Under the control of a randomised algorithm, the blocks in the dam were stressed, which sometimes caused blocks to spring a leak. The application implemented a single agent that could pursue a set of goals to first configure the simulation, monitor the dam wall for problems, and then respond when the dam sprung a leak.

5.1.3 Formal definitions of Agents and Multi-Agent Systems

An *active agent* is defined as one that encompasses its own thread of control (Wooldridge, 2009). Unlike object instances which can only undergo a change of state when one of their methods is called, active agents can initiate a state change based on their own deliberations. Adapting the notation of Kidney and Denzinger (2006), an agent ag_i can be defined as:

Definition 5.1.1 (Agent) $ag_i = \langle Sit, Act, Dat, f_{sit} \rangle$ *is a tuple where:*

1. $ag_i \in Ag$, *the set of all agents that exist in the engine,*
2. Sit *is the set of situations that an agent can be in. The characteristics of the agents situation* $sit_j \in Sit$ *is defined by the nature of the goal it is pursuing or an individual task within that goal it is performing.*
3. Act *is the set of actions that an agent can perform in that environment,*

4. *Dat is the set of internal data the agent maintains about its state and the environment it is situated in, and*

5. *f_{sit} is a function defined for the current situation over the data values that allows the agent to determine its next actions such that $f_{sit} : Sit \times Dat \rightarrow Act$*

Agents are situated within *environments*. For an ICPS, an environment is defined as a bounded physical area that the ICPS operates in and has some measure of control or influence over (Weyns, Omicini & Odell, 2007). The space outside the boundary of the environment is beyond the direct control of the ICPS, however it may receive or transmit information across that boundary. An environment Env that an ICPS inhabits, can interact with, and shares information about with the agent is defined as:

Definition 5.1.2 (Environment) *$Env = \langle S, s_0, Act, f_{Env}, O \rangle$ is a tuple where:*

1. *S is the set of all the possible states the environment can be in,*

2. *$s_0 \in S$ is the initial state of the environment,*

3. *Act is the set of all possible actions that can be performed by the agent within the environment,*

4. *f_{Env} is a transition function that specifies how each of the actions $act_n \in Act$ causes a change in the state of the environment such that $f_{Env} : s_q \rightarrow s_n$, and*

5. *O is the set of observations that the agent can make about the environment. An observation $o_i \in O$ is a perception that an agent can make as a result of watching or influencing the activities of the ICPS operating within the environment. Observations usually result in the agent gathering more information about the state of the ICPS.*

The Env is often a lot richer than the agent's own data, captured in its own Dat, which is always a partial representation of the environment as currently perceived by the agent (Roos, ten Teije, Bos & Witteveen, 2002; Calvaresi et al., 2017). Not all actions that occur in the Env are observable. The ICPS environment that the agents can perceive is constrained in two ways:

1. Agents do not attempt to diagnose faults that are outside the environment of the ICPS that they have been assigned to observe. By implication, an agent is situated in the environment of only one ICPS.

2. Agents do not usually interact directly with the environment of the ICPS they are diagnosing. Rather, all information about the environment is captured by observing or interacting with the ICPS and reasoning about the ICPS activities that result. In some situations, the agents could use information such as the temperature of a room that they have sampled from a reference sensor that is not under the control of the ICPS being diagnosed. However, for the purposes of this study, that capability is not considered. An exception to this was made to allow the agent to specify a test temperature to be emitted from the simbIoTe simulated environment during an experiment described in Chapter 9. This opened up a way to exercise simulated temperature sensors that were otherwise unobservable.

By way of example, an agent ag_i that has been assigned to watch for and investigate faults is situated or given access to observe and interact with the ICPS in the environment Env of an ICPS. Within the bounds of that environment, a set of situations Sit exist that the agent can find itself in or transition to. Each of these situations corresponds to an operational mode of the agent where particular types of activities can be performed. The agent has moved into that situation because it has been given a goal to pursue. While investigating from the perspective of a particular situation, the agent may take one of several alternative actions $actp$ where $actp \in Act$ to gain more information about what is happening or intervene to mitigate the fault. The agent begins to take an action by executing a function f_{ag} that is applicable to that situation. For example, in the case of a function block that does not appear to have processed an input correctly, the agent may choose an action that exercises the full range of the inputs and outputs, comparing the results against expected values. The action of exercising the function block will result in the internal state of the function block changing, which may effect the environment.

It follows from this that a Multi-Agent System MAS can be defined as:

Definition 5.1.3 (Multi-Agent System) $MAS = \{Ag, Env\}$ *where:*

1. Ag *is a set of agents, and*
2. Env *is the environment the agents currently observe and interact with via the ICPS.*

5.1.4 Agent goals, actions, and behaviours

Within GORITE, teams of agents are assembled and allocated goals. A goal that an agent attempts to achieve is defined in terms of the set of actions Act that the agent can perform to work towards a desired outcome. Jarvis et al (2013) make no distinction between the traditional BDI concept of a plan and a goal; a plan in GORITE is just an explicit goal. The pursuit of a goal is evidenced as a set of *behaviors*. The behavior Beh of an individual agent ag_i is described by the set of actions that it takes in its environment while it is attempting to achieve a particular goal:

Definition 5.1.4 (Behaviour) $Beh_{task} = \{act_1,\ act_2,\ ...,\ act_t\}$ *where*

1. $act_n \in Act$*, are the actions the agent* ag_i *is capable of performing, and*
2. act_t *is the action that causes the agent to terminate its operations when that task is completed. Since agents have the ability to self-determine what the appropriate course of action might be in a given environment, it is possible that a number of different behaviors could be exhibited that still achieve the same outcome.*

Agents need to possess and know how to use skills that enable them to perform the actions that constitute a behaviour Beh_{task} applicable to the goal. Skills are realised as part of the Interaction belief structure detailed in Chapter 7.1.1. Then, the overall behavior exhibited by a Multi-Agent System is the set of all the behaviours being exhibited by all the individual agents in the system:

$$Beh_{MAS} = Beh_{ag_1} \bigcap Beh_{ag_2} \bigcap ... \bigcap Beh_{ag_j}$$

Agents only begin executing activities when they are assigned a goal to pursue. The GORITE Java *execute()* function provides a method body in which to implement the code to allow an agent to perform tasks towards fulfilling the goal. Goals are designed to execute until the agent either completes, suspends or fails the goal. A BDI Goal in GORITE is defined as:

Definition 5.1.5 (Goal) *$Goal = \langle N, Ga, gs_w \rangle$ is a tuple where:*

1. *N is a unique identifier that names the goal,*
2. *Ga is the subset of actions the agent can perform that are applicable to fulfil this goal where $Ga \subset Act$, and*
3. *gs_w is the current state of the goal where $gs_w \in Gs$, the set of defined GORITE goal states.*

Figure 5.1 illustrates the structure of a GORITE goal named CONFIGURE_DIAGNOSTICS. The agent is using a domain-specific skill called *loadfbapp()* to analyse the function block application it will examine. Using that knowledge, it then creates a specific diagnostic test harness using the skill *createHarness()* that allows it to interact with the application during fault-finding. The agent that has been assigned this goal determines the status of the goal to be returned to the GORITE agent scheduler once the agent decides that the goal has either been completed or has been interrupted. The interaction skills are discussed in more detail in Chapter 7.

```
Goal configureDiagnostics() {
    return new Goal (CONFIGURE_DIAGNOSTICS) {
        //
        // execute()
        // =========
        public Goal.States execute(Data data) {
            int loadStatus = skills.loadfbapp(applicationPath, applicationName);
            if (loadStatus == XMLErrorCodes.LOADED) {
                // The function block application was loaded and
                // analysed successully. Now create the diagnostic
                // harness.
                if (skills.createHarness(fbapp, dps, applicationPath, server)) {
                    return Goal.States.PASSED;
                } else {
                    return Goal.States.FAILED;
                }
            } else {
                return Goal.States.FAILED;
            }
        }
    }
};
```

Figure 5.1: Implementation of the agents CONFIGURE_DIAGNOSTICS goal.

GORITE defines the following set of Goal states that can be returned from an *execute()* method:

- PASSED: The current goal has been completed successfully by the *DiagnosticAgent*. This usually results in the *TeamManagerAgent* assigning a new goal if there is a subsequent task that follows on from the goal that has just been completed.
- STOPPED: The agent cannot complete the current goal at the present time. This usually signifies that there has been some sort of obstruction in the environment that is stopping the agent working on the goal. The agent is recommending to the *TeamManagerAgent* that the goal should be re-scheduled to be attempted again at a later time.
- FAILED: The agent cannot complete the current goal and has determined that further attempts to re-start and complete the goal would be futile. The *TeamManagerAgent* will not attempt to re-schedule this goal again during the current fault diagnostic session. The agent has the ability to count its own goal repeats since it manages its own state. Hence, it can decide after a number of attempts that the goal has failed. In this case, it will return a belief and the Goal.State of FAILED. GORITE will not re-schedule a failed goal.

5.2 Prototyping and evaluating the agent architecture

Figure 5.2 was adapted from the original design in Section 3.2 of the Software Architecture Document. It illustrates the hierarchy of the agents within the GORITE framework and the way the components specified for the engine in the Software Architecture Document were implemented. The *DiagnosticTeam* component creates a team of agents and assign goals to them. However, the Open Source version of GORITE does not provide a way of implementing multiple agent instances that can be multi-threaded. Since an *execute()* function is designed to operate until the status of the goal changes, that implies that different agents cannot perform their goals in parallel. The first prototype single agent performed goals that included the configuration of the function block application, watching a simple fault, and then responding by notifying

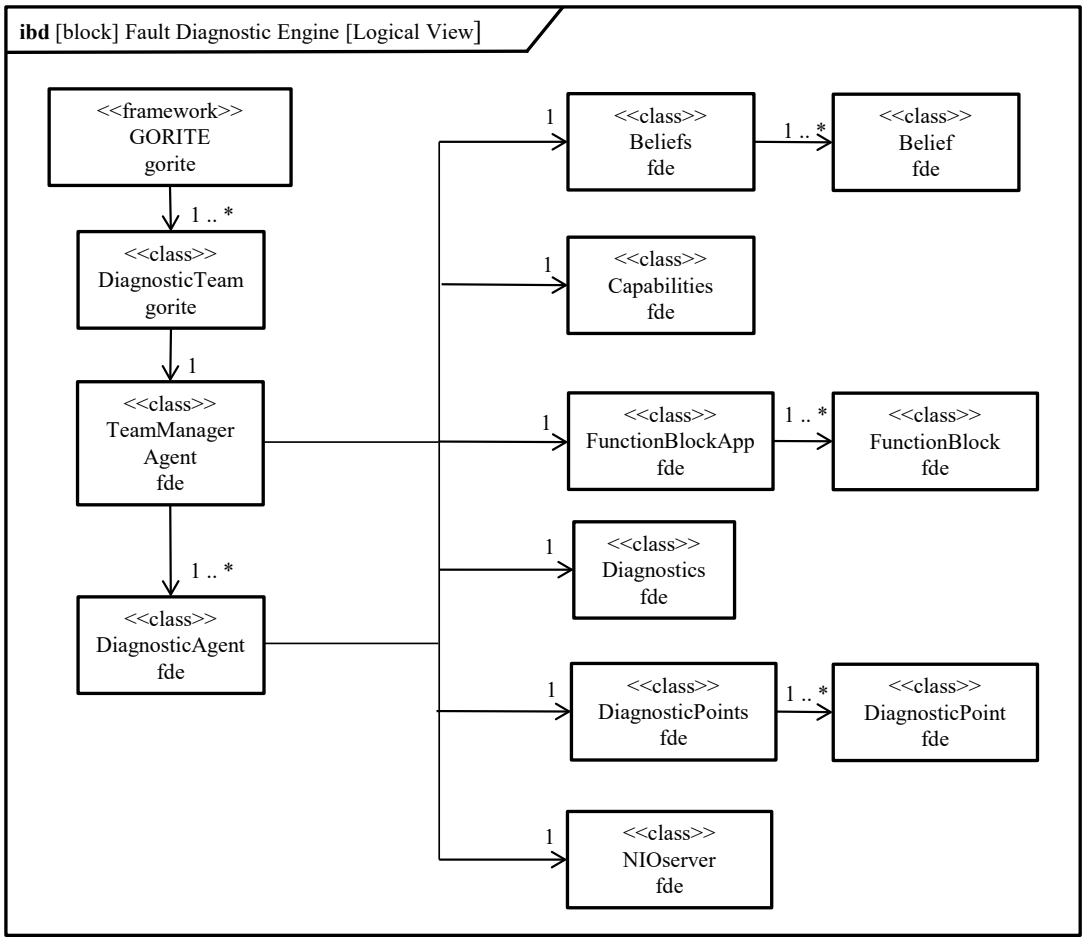

Figure 5.2: Logical View of the team of agents within GORITE and the engine.

GORITE. As each goal $Goal$ was completed, the *DiagnosticTeam* scheduled the next goal for the agent or re-scheduled a goal that could not be completed during this iteration.

This scenario was evaluated using the ATAM detailed in Table 5.1. It identifies the quality attributes including Concurrency and Scalability that are not satisfied by trying to execute agents on a single execution thread. The agents also need to be able to persist their own private copies of resource objects that have their own state management requirements.

Table 5.1: ATAM for the agent architecture.

Quality Attribute	**Concern**	**Trade-offs**	**Solution**
Concurrency, Scalability	Agents need to be able to operate asynchronously.	Memory and processor overheads	Operate agents on independent threads.
Resource allocation	Agents need to instantiate their own resource instances as independent objects	Memory and processor overheads	Private objects owned by each agent. Use synchronised method calls. Share some common instances as Singletons.

The role of the new *TeamManagerAgent* is illustrated in Figure 5.3. This component is an agent that executes its goals in a similiar way to the traditional GORITE goal execution method illustrated previously in Figure 5.1. It operates as the only agent within the GORITE main thread, co-ordinating its activities closely with the other GORITE functionality that it relies on for framework support. An example of the *execute()* for one of these goals is shown

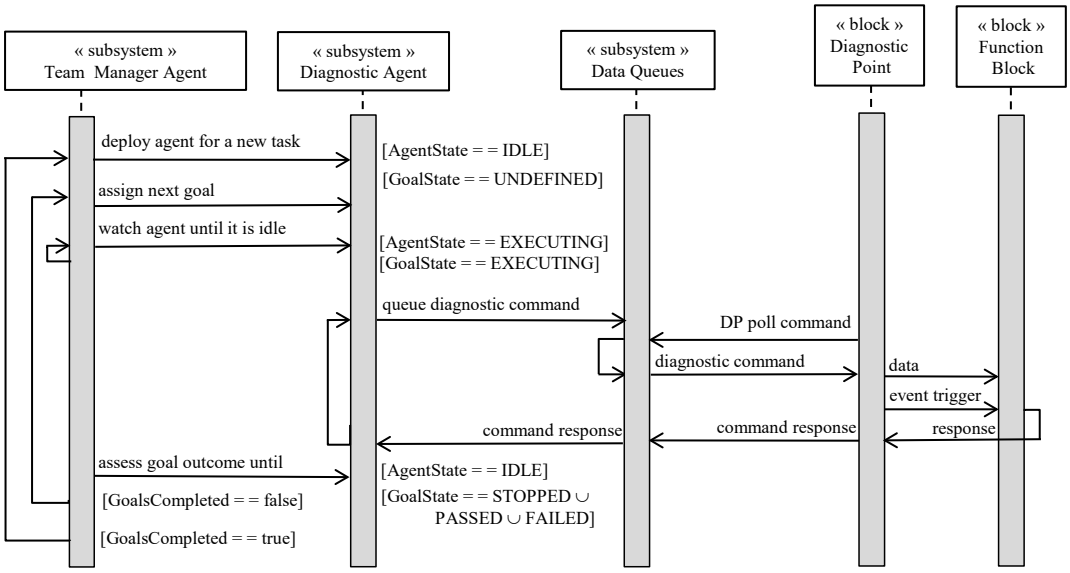

Figure 5.3: Team, agent, and diagnostic point interactions across the execution layers.

in Figure 5.4. The *TeamManagerAgent* executes a set of GORITE Sequence Goals named `MANAGE_AGENT_n` where its role is to assign goals to idle agents and monitor their progress. Setting up a separate management goal for each agent allows individual GORITE *Data* pools to be maintained for each agent instance. The *TeamManagerAgent* is therefore able to delegate tasks to multiple agents and watch them each time it checks on their activities by executing its set of goals sequentially. This is a generic structure that is repeated for the collection of agents. It can process a variable list of goals that are specific for the diagnostic task or the particular role

```
//
// manageAgent1()
// ==============
Goal manageAgent1() {
    return new Goal (MANAGE_AGENT_1) {
        //
        // execute()
        // =========
        public Goal.States execute(Data data) {
            Goal.States goalState = Goal.States.PASSED;
            String currentGoal = data.getValue("currentGoal").toString();

            if (agent[1].AgentState() == AgentStates.EXECUTING) {
                // Leave the agent to get on with the task; it is busy.
                // Ask GORITE to come back to this goal later.
                sleep(500);
                goalState = Goal.States.PASSED;

            } else if (agent[1].AgentState() == AgentStates.IDLE) {
                if (agent[1].GoalName() != UNDEFINED_GOAL) {
                    // The agent had a goal which has now been either completed or stopped.
                    // so decide what to do next. The goalState returned will also have
                    // been determined by the evaluateGoal() method.
                    evaluateGoal(agent[1]);
                    String nextGoal = data.getValue("nextGoal").toString();
                    agent[1].GoalName(nextGoal);
                    goalState = agent[0].GoalState();

                } else {
                    // The agent is awaiting an inital goal to execute.
                    String firstGoal = data.getValue("firstGoal").toString();
                    agent[1].GoalName(firstGoal);
                    agent[1].AgentState(AgentStates.EXECUTING);
                    goalState = Goal.States.PASSED;
                }
            }
            return goalState;
        }
    };
}
```

Figure 5.4: Implementation of the *TeamManagerAgent* MANAGE_AGENT_n goal.

chosen of an agent. GORITE sequentially executes the set of individual MANAGE_AGENT goals for the Team Manager Agent. Once setup, the engineer only needs to specify a set of goals and the number of agents. GORITE handles the rest of the scheduling and re-scheduling.

By applying this refinement suggested by the ATAM, the redesigned *DiagnosticAgent* class allows multiple agents to be instantiated, each on their own execution thread. Each named agent instance provides a set of Java thread synchronised methods that ensure the setting and reading of the `AgentState()` and `GoalState()` agent properties does not cause collisions and is interleaved appropriately. Agents execute their own goals such as `CONFIGURE`, `MONITOR`, `DIAGNOSE_FAULTS`, and `REPORT_DIAGNOSIS`, transitioning from first configuring the IEC 61499 application they are assigned to watch, watching the telemetry streams, and intervening when fault evidence is observed. Each transition to a new goal is initiated by reporting that the current goal is complete whereupon the *TeamManagerAgent* evaluates the goal status and assigns a new goal. Five agents were created and evaluated to address the ATAM findings. No collisions or conflicts were observed in the scenarios.

Both types of agents have full access to all the other resources available from the engine. Objects such as the non-blocking server *NIOserver* were developed in subsequent phases to manage the function block fault telemetry queues. A single instance of this communications subsystem can be shared amongst multiple agents. The architecture of this component is discussed in more depth in Chapter 6. Instantiating agents on their own threads required a different approach to goal execution. This resulted in the creation of the new *TeamManagerAgent* class which is similar in structure to the *DiagnosticAgent* class. Subsequent discussion in Chapter 6 details the architecture of the Diagnostic Points which capture the function block telemetry. Chapter 7 then explores the *Beliefs* structures in more detail, including the sysem-under-diagnosis beliefs contained in the *FunctionBlockApp* belief structure.

5.3 Conclusions and planning the next phase

The Software Architecture Document proposed the design of the GORITE agents that were prototyped and evaluated in this Design Science phase. Establishing an appropriate configuration of the multi-agent framework subsystems and evaluating them via iterative prototypes led to revisions such as multi-threading some of the subsystems. The next phase implements the Diagnostic Points that the agents can use to interact with the test function block applications that were created to evaluate these components.

Chapter 6

Creating Diagnostic Points

"You see, but you do not observe. The world is full of obvious things which nobody by any chance ever observes."

Holmes speaking to Watson in "*A Scandal in Bohemia*". Arthur Conan Doyle (1891)

ICPS are dynamic, adaptive systems. Their cyber parts rely on data captured from their physical environments. How they respond to what they observe causes changes to occur both to their internal disposition and to things within their environment. In an ideal situation, most of those interactions should accomplish useful work. However, during the normal course of their activities, faults will inevitably occur. This chapter describes a way to provide the agents with the ability to interact with an IEC 61499 function block application so that they can observe its behaviour directly. It describes a novel approach called Diagnostic Points, hereafter referred to as "DPs". Sections of this chapter were adapted for publication in the IEEE paper *Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems* (Dowdeswell, Sinha & MacDonell, 2020a).

DPs build on a rich legacy of Built-In-Test (BIT) approaches to fault-finding (Hale & Bollas, 2018; Kochte & Wunderlich, 2017). The chapter illustrates how to enable an agent to gather real-time telemetry from a running IEC 61499 application by observing, isolating, and then interactively exercising function blocks of interest. Whereas BIT interfaces for traditional embedded systems are hard-wired in at design time, DPs are dynamic software-in-loop data

capture and injection points created by the agents in real-time. DPs extend the BIT paradigm by being both more versatile and feature-rich.

The agents created in the previous Design Science phase implemented the framework in which to model and investigate the interactivity needed to respond to faults. Simpson and Shepherd (1991) describe the general problem of fault diagnosis as being extremely difficult. They assert that the process of optimizing a sequence of diagnostic tests that could be applied to any given system as being NP-complete. The literature examined in Chapter 2 corroborates their assertion, demonstrating just how wide the choice of diagnostic approaches is that the agents could apply. However, being NP-complete also implies that it should be possible for the agents to choose an optimal diagnostic approach from the resources available to them in a reasonable time. Providing them with domain knowledge of IEC 61499 and specific skills constrains their choices, addressing the generality of the fault diagnosis problem that concerned Simpson and Shepherd. This research addresses these and other agent challenges by providing the agents with interactive components that they can deploy non-intrusively deep inside an IEC 61499 application. Keiner defines this approach to integrating diagnostics as the "*structured process that maximizes the effectiveness of diagnostics by integrating individual diagnostic elements of testability, automatic testing, manual testing, training, maintenance and technical information*" (Keiner, 1990, page 1).

The architecture for the DPs was proposed in Section 3.3.5.2 of the Software Architecture Document to gather the telemetry required to enable the agents to perform diagnostic tasks and address RQ3 and RQ4. The design was realised by developing and evaluating two iterations of DP prototypes. As in earlier Design Science phases, each design relies on the quality attributes defined earlier for this engine. An ATAM is presented which evaluates how well each prototype DP addresses the needs of the engine. This phase of the work contributes the latest version of the DP, which is now a light-weight Composite Function Block (CFB) that incorporates a customised Service Interface Function Block (SIFB). Agents now deploy and wire-in as many DPs as they require to tap into the diverse data streams of the function block application.

6.1 Capturing telemetry and prior approaches

Telemetry is defined as the in situ collection of measurements or other types of data at remote points and the transmission of that data back to receiving equipment that can process it further (Carden, Jedlicka & Henry, 2002). The internal operation of a function block is not directly observable (Preuße, Missal, Gerber, Hirsch & Hanisch, 2010). This forces agents to treat function blocks as black-boxes whose behaviour or mis-behaviour can be inferred from the data they exchange with each other.

The agents operate within the environment of the fault diagnostic engine, deployed on its own hardware. They rely on telemetry they can capture since the IEC 61499 application is most often executing on its own, separate hardware platform. The rational for creating stand-alone fault management engines is discussed in-depth in Benowitz's (2014) description of the NASA Curiosity Mars Rover and in the Airbus fault engine discussed by Goupil et al. (2014). A fault diagnostic engine has to keep functioning reliably even when the application it is analysing is failing. Hence the DPs need to provide remote data capture and interaction points that can operate as software-in-the-loop entities within the IEC 61499 application running on its own hardware. The architecture chosen for the DPs facilitates communications with agents via TCP network connections. Section 2.2.2 of the Software Architecture Document and the ATAM discussions outlined the reasons for choosing TCP over the Universal Datagram Protocol (UDP).

Several prior works have investigated the creation of software-in-loop entities to test or find faults in function block applications. Hametner, Hegny, and Zoitl (2017) proposed a Unit-Test framework for IEC 61499 that was capable of injecting test values. They identify the smallest testable unit in IEC 61499 as being the fundamental Basic Function Block since it is the smallest observable entity. Hence, in line with our proposed approach, they also treat function blocks as black-boxes. They describe a test runner, similar in concept to that used by the JUnit Unit-Test framework, that they implement as a custom function block. This is wired into the application being tested. However, they identify limitations in the current runtimes that will need to be addressed to test complex function block applications, especially those with timing constraints.

Glanz et al. (2016) discuss a similar approach to capturing behaviours for IEC 61499

applications. They outline a mechanism to determine if a specific input to a function block produces the correct output by injecting representative faults. The range of issues they seek to detect include outputs that do not transition, random outputs, delays, and unstable outputs that do not latch properly. They note the need to syncronise with the 4diac/FORTE runtime event chain, a requirement our DP approach addresses. Glanz et al. describe how their test data inputs must be accompanied by appropriate event triggers, otherwise the function block will ignore the new data. At the time of publication, they did not describe a practical implementation or a way of inserting their additional test-point function blocks into the application. This requirement to enable our agents to deploy DPs autonomously is addressed in our approach.

6.2 How Diagnostic Points operate

To illustrate how the version one DPs operate, Figure 6.1 shows an example Fahrenheit to Celsius temperature converter function block `F_TO_C_CONV` inside an IEC 61499 application.

The `F_TO_C_CONV` function block is part of a larger Heating Ventilation and Air-Conditioning

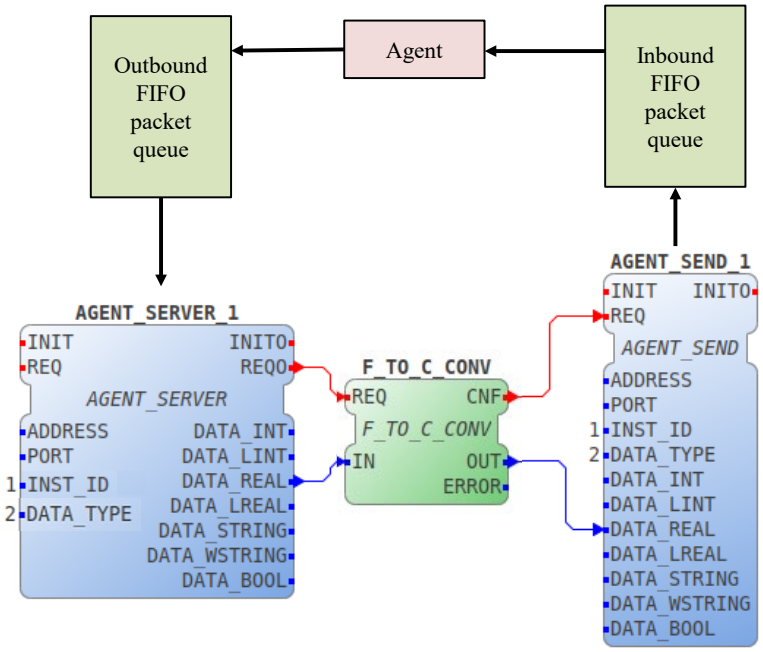

Figure 6.1: Agent and DPs interacting with the F_TO_C_CONV function block.

(HVAC) function block application that is profiled in more detail later in Chapter 9. A complementary pair of DPs has been wired into the input and output data and the event ports of this function block. The non-blocking TCP server `NIOserver` within the engine was detailed earlier in Figure 5.2. This subsystem manages multiple First-In-First-Out (FIFO) input and output network queues of data packets. Each queue is responsible for the traffic to and from a single DP instance. The `AGENT_SERVER_1` function block implements its own TCP server that the agent can connect to as a TCP client. The agent can then inject test Fahrenheit temperatures into `AGENT_SERVER_1`. This DP then outputs a Real number to the `IN` data port and triggers the event `REQ` that causes `F_TO_C_CONV` to convert the Fahrenheit value to Celsius.

The temperature readings output from the `F_TO_C_CONV` block are captured by the second function block `AGENT_SEND_1` whenever the `CNF` event fires. This DP instantiates its own TCP client that can connect to the inbound FIFO packet queue managed by the engine. This asynchronous connection is established by the DP during startup. The agent retrieves data packets from the FIFO queue that were sent by `AGENT_SEND_1` from the output port `OUT` that carries converted temperatures out of `F_TO_C_CONV` as Real numbers. The asynchronous nature of these connections ensures that the agent can manage multiple diagnostic points without needing to meet unrealistic timing constraints. Data is queued reliably by `NIOserver` in the correct order and is retrieved in-sync with both the event chain of the function block application and the processing cycle of each agent.

The `AGENT_SERVER_1` and `AGENT_SEND_1` diagnostic blocks provide a port for each defined IEC 61499 data type. These range from Boolean types through to long and short Integers, Real numbers, and Strings. While the IEC 61499 standard (IEC, 2013a) describes an ANY data type, it is not implemented within 4diac/FORTE. An efficient type conversion to a String format in C++ was implemented within the DPs to ensure that data was processed rapidly and packed efficiently into the telemetry data packets.

Figure 6.2 details the packet structure implemented to exchange data between the DPs and the engine FIFO data queues. The start-of and end-of packet characters are used to ensure that the packet has not been malformed and to allow multiple variable-length packets to be unpacked

from the input buffer reliably in a single read from a TCP port.

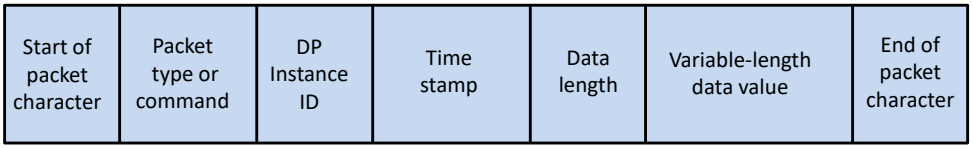

Figure 6.2: Definition of the Data Packet structure used to exchange telemetry.

The data type can be inferred by the agent since it was established during the set up of the DP. The input `DATA_TYPE` is used by the block itself to determine which port will carry the data and the format conversion needed into and out of the string format that is carried in the data packet. TCP addresses and the Host listener port are also specified via input parameters. Table 6.1 lists the packet commands implemented for these version one DPs:

Table 6.1: Packet Type Commands supported on version one Diagnostic Points.

Packet type enumerator	**Description**
TRIGGER_DATA_VALUE	Packet contains a data value to inject into the function block. Used by `AGENT_SERVER` type blocks.
SAMPLED_DATA_VALUE	Typed data value sent to the agent from `AGENT_SEND`.

6.3 Evaluating the version one DP architecture

An ATAM scenario was created where five pairs of DPs were operating on five instances of `F_TO_C_CONV` blocks. All DPs were under the control of a single agent executing a fault-finding goal. Table 6.2 details the ATAM concerns identified. Injecting an incrementing sequence of test temperatures that varied by one degree Fahrenheit allowed the set of captured outputs to be matched to the inputs sent. While triggering the `F_TO_C_CONV` blocks at 100ms intervals in repeated trials, no missing sequences of values were observed. Scalability and Performance were evidenced by the processor load on the Linux system that was running the engine instance. It showed acceptable memory loads and processor peaks of no more than 70%, suggesting that the operations did not compromise the performance of the engine. Separate tests running the function block application alone suggested that the majority of the processor usage

Table 6.2: ATAM for the version one Diagnostic Point architecture.

Quality Attribute	**Concern**	**Trade-offs**	**Evaluation and solution**
Scalability, Performance	Multiple DPs need to be marshaled by agents and operate without losing telemetry.	Synchronisation. Memory and processor overheads.	Multiple DPs were demonstrated operating reliably.
Reliability, Functional Suitability	TCP versus UDP for network communications.	Extra network overhead and versus performance versus reliability, error -correction and ease-of-implementation.	The choice of TCP over UDP was justified by the reduction in errors and guaranteed delivery it provides. The performance and network overhead was determined to be minimal.
Functional Appropriateness, Easy-of-use	Configuration of the version one DPs is cumbersome and difficult to automate. It also requires each test to be configured individually, making iterative walks through the function block application difficult.	None except for the added complexity of the version two DPs.	Testing each function block requires DPs to be re-wired differently for normal monitor operation versus fault-finding.

was overhead from the FORTE function block runtime that executed the function blocks (4diac, 2019). Evaluating the version one DPs showed that exchanging data between the agents and the function block application via labeled data packets was a viable approach. Traps were placed in the engines queue processing code to identify packets that could not be unpacked reliably. A run of over four thousand packets exchanged during a set of repeated test, no issues were identified.

However, configuring the DPs for fault monitoring versus interactive fault injection required the application to be stopped, rewired to insert the appropriate DPs, and then re-started. This proved to be cumbersome since it did not allow the agents to follow multiple fault paths without the need to stop and re-configure. A more versatile approach would deploy all DPs initially as listeners who could be commanded by the agent to switch to test value injection mode on-command. This would allow a single test harness to be created that still allowed the function block application to operate normally when deployed for long-term observation. On-command, the DPs would dynamically switch their operating mode to allow test values to be injected.

6.4 Creating version two of the DP architecture

Figure 6.3 shows the construction of the version two diagnostic point function block DP. This is a Composite Function Block that contains an instance of the new AGENT_GATE Service Interface Function Block. The AGENT_GATE block implements a pass-through gateway so that a DP can be used to break and re-connect the data and event lines that connect function blocks together.

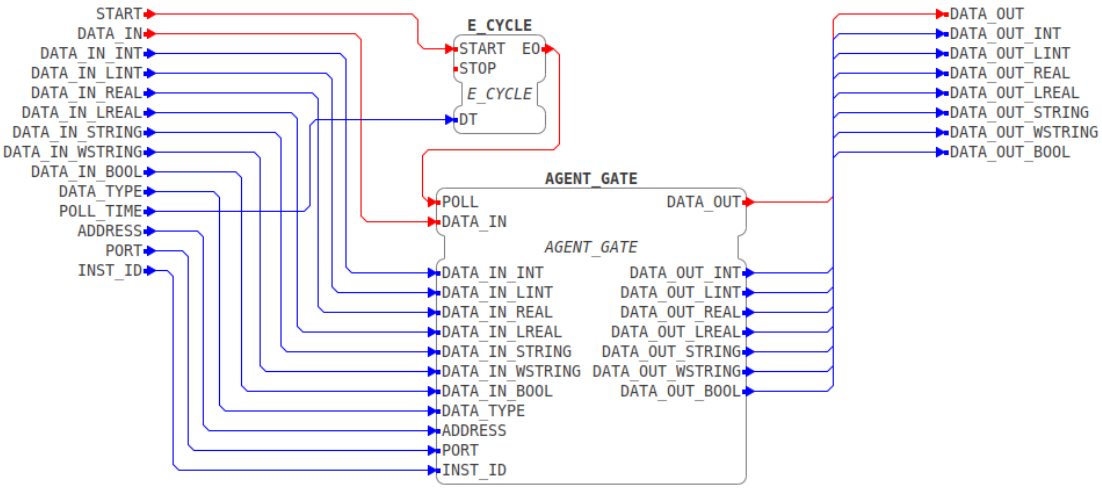

Figure 6.3: Diagnostic Point Composite Function Block - version two.

Composite Function Blocks expose a single input and output interface that encapsulates a more complex internal configuration. The configuration parameters from the version one DP are extended to include a new POLL_TIME parameter that allows the performance of the DPs to be fine-tuned by the agent during the configuration of the diagnostic test harness.

Figure 6.4 shows the original F_TO_C_CONV function block after it has been surrounded by three version two DP instances. In this configuration, the DPs initially pass all data and events through transparently after the agent has deployed them to operate in their default PASSTHROUGH_ENABLED mode. Each data value is also sent back to the agent after it has been passed through the DP. The AGENT_GATE block implements a TCP client connection to a single named NIOserver gateway hosted on the engine. A TCP socket handover then assigns a unique socket to maintain the connection between the agent and the DP. The agent communicates with the DP wired into the function block application via an in-memory object

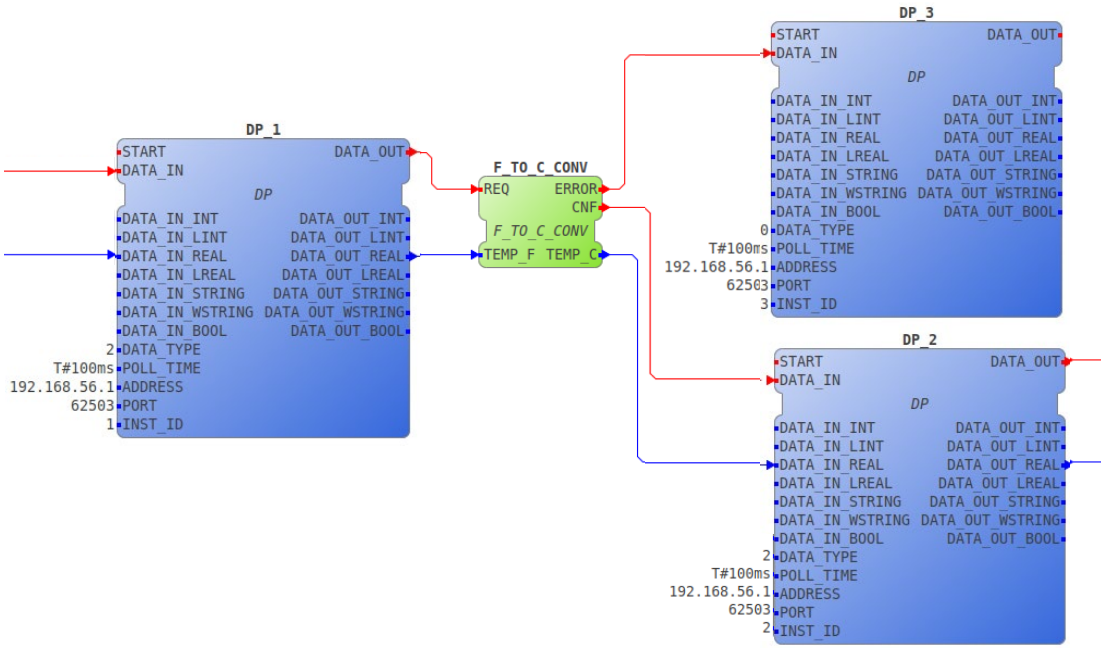

Figure 6.4: DP interacting with the F_TO_C_CONV function block - version 2.

interface, also referred to as a *DiagnosticPoint*, established within the engine. Figure 5.2 from Chapter 5 illustrates the collection of DiagnosticPoints maintained by the agent.

The Reliability quality attribute was addressed by making the DP function blocks more fault-tolerant to network issues by ensuring that they manage their TCP client connections themselves. In the event of a loss of connection, the DP re-establishes a new connection by itself. Since the data packets carry a unique DP instance identifier, the matching DiagnosticPoint instance in the engine automatically re-routes the packets to and from the correct FIFO packet queue via the new socket. This ensures that all network interruptions are handled asynchronously by the `NIOserver` and the DiagnosticPoint subsystems. In this sense, the DiagnosticPoint acts somewhat like a Digital Twin of the DP within the function block application (Kritzinger, Karner, Traar, Henjes & Sihn, 2018). Later, this was shown to make the creation of the diagnostic scripts the agents use to execute their goals simpler to construct. The was because the instance references to the collection of DPs marshalled by the agent now have a much simpler format.

Table 6.3: Packet Type Commands for version two of the Diagnostic Points.

Packet type enumerator	Description
SAMPLED_DATA_VALUE	A typed data value sent to the agent that has been either captured from an input or an output port on the function block.
PASSTHROUGH_ENABLED	Command received from the agent to switch the DP to its transparent pass-through mode.
POLL_AGENT	The DP is polling the agent, signaling that it is ready to receive a new data value to inject into the function block. The value is returned in a post-back from the NIOserver.
TRIGGER_ENABLED	Command received from the agent to switch from its transparent pass-through mode and begin requesting and injecting test data values. Used in conjunction with the GATE_OPEN and GATE_CLOSE commands.
GATE_OPEN	Instructs the DP to open the gate to traffic from other function blocks after switching back to its PASSTHROUGH_ENABLED mode.
GATE_CLOSE	Instructs the DP to close the gate, blocking traffic to and from other function blocks after switching to its TRIGGER_ENABLED mode.
TRIGGER_DATA_VALUE	A typed data value received from the agent to inject into the function block.

Table 6.3 details the enhanced packet commands supported by the version two diagnostic point and the agents. The gate functionality allows the agent to dynamically ring-fence either a single function block or a set of related blocks to exercise them. Data and events are blocked from passing into or out of that ring-fenced area to allow the blocks to be tested in isolation, unaffected by other parts of the application. The agent can then restore the pass-through mode of those blocks and move on to investigate other blocks of interest. In this way, the agent can walk through an entire application, investigating, probing, and then move on without having to re-wire any additional diagnostic points. This improvement to the version one approach has resulted in a more resilient approach that allows the agents to change the type of interventions they choose to make during both short-term design testing and longer-term burn-in monitoring and fault management.

6.5 Evaluating the version two DP architecture

Experiments with other example function blocks suggested ways to extend the functionality of the new DP block. The original version one block could not capture or trigger events if there was no corresponding data input or data output port to work with. This functionality was added and used extensively later in the examples described in Chapter 9. Many of the signals the

HVAC Room Controller used only required blocks to exchange events. Examples include the switches and the request to sample the temperature sensor.

The work in Chapter 10 also required the DPs to capture precise timing telemetry. This was accomplished by using the Linux epoch timing values. The epoch value is the number of milliseconds that have elapsed since 1st January 1970. The DPs capture the time when an event is triggered, not when they received the trigger instruction from the agent. Immediately after the triggering the event, they return the timestamp value in a data packet with the type `TIMESTAMP`. Timestamps captured from other related DPs provide the time a subsequent event of interest occurred. Subtracting these values provides millisecond-level event time measurements. Since the timing is captured inside the function block application itself rather than being an observation by the external agent, the timing is inherently more accurate. This later allowed the timing requirements of the Smart Grid components to be verified empirically.

6.6 Conclusions and planning for the next phase

The Software Architecture Document proposed the initial design of the diagnostic points and their evaluation criteria. These were prototyped, evaluated, and refined in this Design Science phase. By addressing the quality attributes of Functional Suitability, Ease-of-Use, and Reliability, a more versatile diagnostic point methodology and implementation was created.

The Fault Diagnostic Engine equips equips a generic set of multi-purpose agents with domain-specific skills and knowledge of the IEC 61499 architecture. These were encapsulated as individual skills that enhanced the capabilities of each agent. Once an engineer specifies possible test points for each function block during the design phase, the agent can use this information autonomously to dynamically create the diagnostic points and use them. This has significant advantages over the Built-In Test (BIT) points traditionally used in embedded system design. 4diac/FORTE allows dynamic re-wiring of function block application which makes this approach possible.

The creation of the agent framework and the diagnostic points established the pre-requisites

for performing more extensive fault-finding in subsequent phases. Subsequent chapters discuss how the agents actually deploy the diagnostic points they need after first analysing the function block application they have been assigned to investigate. This work includes the formation of the *belief structures* employed by the agents that drive both their reasoning and their final diagnosis.

Chapter 7

What an Agent Believes

"Beliefs are choices. First you choose your beliefs. Then your beliefs affect your choices."

"*The Light in the Heart*", Roy T. Bennett (2016)

Wooldridge (2009) describes two characteristics of agents that are important when we apply them to perform fault diagnosis. Firstly, agents situated in an environment are capable of semi-autonomous behaviour. This allows them to follow fault evidence trails by themselves through a malfunctioning ICPS. By observing and evaluating fault symptoms without assistance from humans, they attempt to reconcile the way the ICPS is operating against known profiles of acceptable performance. Secondly, by coordinating their actions with other agents operating within the same environment, they can co-operatively share both the analysis tasks and evidence gathering to ultimately present a diagnosis of the faults.

Central to these capabilities is the way that an agent is able to form and manage *beliefs* about what it is observing. Beliefs are opinions held by an agent, formed during a methodical examination and testing of individual parts of the ICPS they are investigating. Beliefs are dynamic: previously-held beliefs may be reinforced or discarded, based on new evidence that an agent has captured (Friedman, 1997). This progressive refinement adds both integrity and weight to beliefs, allowing the relative probability of alternate fault hypotheses to be considered in the light of the opinions the agent now holds. Once the agents can both form and examine

their beliefs, the foundations exist for them to begin to reason about what they have seen, ultimately leading to them presenting a fault diagnosis. The work in this phase addresses RQ3, demonstrating how the central tenet of the Belief-Desire-Intention paradigm is realised by creating ways for agents to manage their beliefs. The belief structures built for the engine were described initially in Section 3.2.5.1 of the Software Architecture Document. This chapter now profiles and evaluates the three belief subsystems in more detail. Sections of this chapter were also adapted for publication in the IEEE paper *Employing Agent Beliefs during Fault Diagnosis for IEC 61499 Industrial Cyber-Physical Systems* (Dowdeswell, Sinha, Jarvis et al., 2020).

7.1 Defining beliefs

Beliefs provide a model of the domain the agent operates in (Ganzha, Jain, Jarvis, Jarvis & Rönnquist, 2013). An *atomic belief* is an opinion about one characteristic of the ICPS or its environment formed from evidence that the agent holds at a given time. Hence the ability to revise a belief is one of their key skills that allows them to maintain an up-to-date model when situations change or when the weight of evidence changes the agent's perceptions.

Dennett (2009) defines a First-Order Intentional system as one that has beliefs and desires, but no beliefs and desires about its beliefs and desires. This corresponds to the BDI concept proposed by Bratman (1987). Dennett then defines a Second-Order Intentional system as one that has formed beliefs, desires and intentional states about the beliefs and desires of other systems as well as its own. The primary beliefs formed by each agent in the engine are First-Order. The beliefs that the agent forms about the function block application it is diagnosing share many of the characteristics of Second-Order Intentional Systems. Definition 7.1.1 is the primary definition of a belief that all the subsequent beliefs used by the agents are built upon:

Definition 7.1.1 (Beliefs) *Every agent ag contains a set of beliefs $B = \{b_1, \ldots, b_n\}$ such that each belief $b \in B$ is a tuple $\langle \triangle, v \rangle$ where:*

- $\triangle$ *is a skill that the agent can use and*

- *v is the veracity of the belief held by the agent about the skill. This may be $true$, $false$ or $undetermined$.*

Veracity is defined as the degree to which a concept or unit of information conforms to the truth or known facts (Merriam-Webster, 2021). The quantities assigned to v can range from true or false supported by numeric values or percentages that indicate how much a behavior is deviating from expected norms. A fault diagnosis is understood to be a summary and evaluation of the set of beliefs held by an agent. When the application is performing nominally, this diagnosis could be a No Fault Found (NFF) belief on the part of the agent. Within the engine, agents can maintain three types of beliefs: *interaction* beliefs, *system-under-diagnosis (SUD)* beliefs, and *dynamic diagnostics* beliefs.

7.1.1 Beliefs about their abilities to perform their goals

Agents are imbued with knowledge or beliefs about the skills and tools they can wield to perform goals. The human analogue of this type of belief includes the characteristic of being confident that they know how to use a particular skill to perform a task. Interaction beliefs are the agent's knowledge of the abilities it possesses that allow it to perform domain-specific tasks:

Definition 7.1.2 (Interaction Belief) *A belief $b = \langle \triangle, v \rangle$ is an interaction belief when $\triangle$ describes a pair $(ag, \mathcal{S})$ where ag represents an agent and the $\mathcal{S}$ is the signature of a method that can be used by that agent to interact with the function block application and other agents.*

An example of an Interaction Belief b_i realised as a skill is the ability of an agent ag_i `Jane.say(`*`text`*`)` where `Jane` is a reference to another agent in the agent's team. Agents share diagnostic data on asynchronous back channels where all communication is buffered in personal queues. Another agent skill, `Jane.hear()`, retrieves the last message sent from that agent. This belief structure models IEC 61499-specific domain knowledge needed to interact with the function block application being diagnosed.

Figure 7.1 shows a set of Diagnostic Points that an agent has wired into the application between a `Z_TEMPERATURE` sensor function block and the `F_TO_C_CONV` block. When the

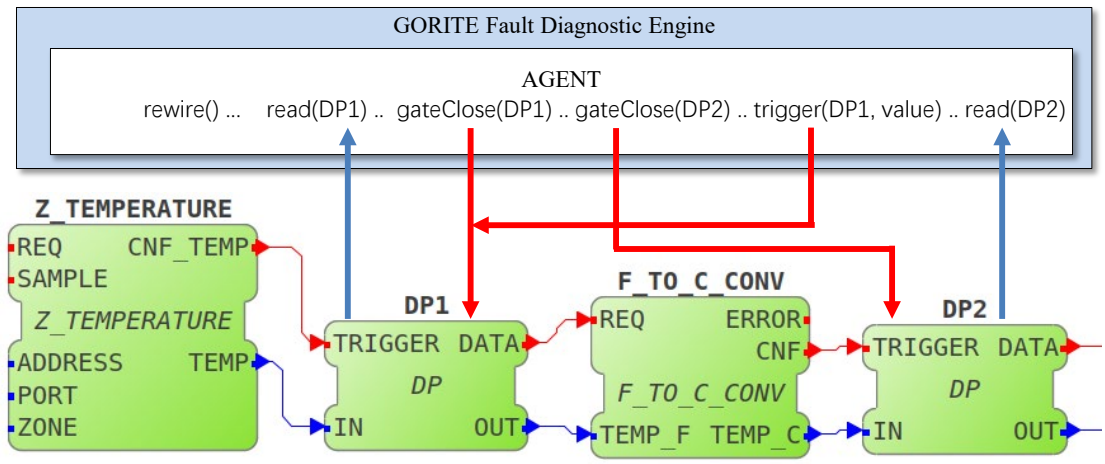

Figure 7.1: Capturing and triggering Diagnostic Points.

agent performs a `rewire()` skill or ability, the `DP1` and `DP2` diagnostic point function block instances are inserted into the event and data paths. Later, the `gateClose()` command isolates this incoming information path from `Z_TEMPERATURE` so that a `trigger()` command can inject test values which are then captured further down the path by `DP2` using the `read()` command. Note that if a diagnostic point is capturing the input side of a function block, the `gate()` function must close only the inbound side. Conversely, when the diagnostic point is capturing an output, only the outbound data and events should be blocked from being sent onto the next connected function block.

While other belief sets are dynamic, the capabilities captured in this first set are static, intrinsic skills since they are core domain-specific abilities of the agent. Hence, the veracity of interaction beliefs is always $true$ since an agent must demonstrate "confidence" in its own abilities.

7.1.2 Beliefs about the function block application

Before the diagnostic goals are assigned to individual agents, the Team Manager agent constructs a belief structure that provides the agents with knowledge about the structure of the function block application to be monitored and diagnosed. Note that all agents possess the same skill sets, so this task could be delegated to any agent in the team. This ensures that a distributed

function block application can be managed by a team of independent agents who co-ordinate the monitoring and diagnosis of the entire system.

Definition 7.1.3 (System-Under-Diagnosis belief) *A belief $b = \langle \triangle, v \rangle$ is called a System-Under-Diagnosis belief when $\triangle$ is described by the triple $\langle FB1, trg, FB2 \rangle$. $FB1$ and $FB2$ are function block instances in the system being diagnosed, and trg represents the conditions (events and variable values) under which a transition can be triggered by the agent from $FB1$ to $FB2$.*

IEC 61499 Application Definition files are optimized to work with development IDEs such as 4diac (Strasser et al., 2008) and nxtCONTROL (nxtControl GmbH, 2016). However, the XML structure of these files is hard for an agent to navigate dynamically since the parameters for each function block are stored in different parts of the XML-format file. Figure 7.2 shows the agents' belief structure about a typical function block application that has been restructured as a Directed Graph. The five function block instances used in the application are modelled as nodes and connections as directed edges. The name of an edge corresponds to the output event or data output on the function block the node describes. The internal data structure that holds this information is much easier for the agents to navigate when rewiring the function block application to insert DPs. Figure 7.3 provides more detail for the FunctionBlockApp structure shown previously in the Software Architecture Document.

Figure 5.2 shown previously illustrated the logical structure that maintains a list of function blocks where each entry is a complex FunctionBlock node. A separate list in the Connections class contains a Connection instance for each data or event connection between two function blocks. Agents are able to access this data more efficiently during their fault-finding by referencing a individual function block and all its details as a single instance inside this belief data structure.

The FunctionBlockApp class provides an `update()` method so that if changes are made to the copy of the function block or the connection, the entry in the list of blocks can easily be updated. This architectural approach using an indexed list of objects means that new methods and data can be added to the structure easily without complicated changes to the interface.

Figure 6.3 showed each of the data types provided by each diagnostic point and the static parameters the agent must configure for each instance.

Each node of this belief structure provides the agent with detailed knowledge to allow it to reliably interact with each function block. Agents are responsible for determining by themselves how to interact with a particular function block, verifying input and output data types so they can provide type-safe test values and interpret telemetry correctly. The Directed Graph structure also allows agents to navigate fault paths autonomously, making decisions about the next possible fault location while they investigate. This belief structure is dynamic. It provides the agents with a way to remember the result of diagnosing a function block as they traverse the application and update their beliefs about what they have observed.

The `rewire()` interaction skill relies on the application belief structure for determining the data types and connections that are being broken and re-wired. Figure 7.4 shows an example function block application before and after the diagnostic points have been inserted into the data and event connections.

During the design phase, a package of information was created that identifies the diagnostic points that are available for each function block. These diagnostic packages contain sets of

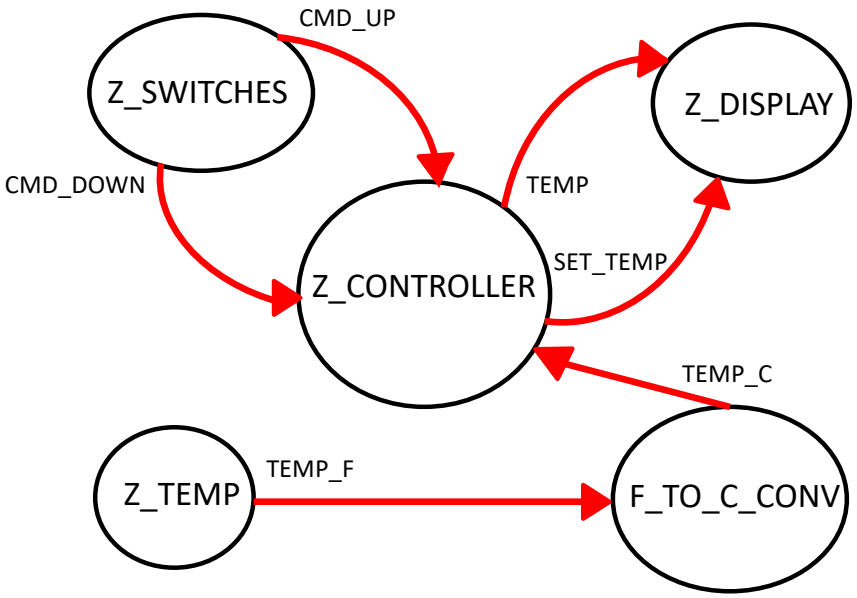

Figure 7.2: The Function Block Application as a Directed Graph.

test values associated with data pathways through the function block that can be used during diagnosis to determine if the function block is performing correctly. The agent interprets these diagnostic packages while iterating each function block as it builds its application belief structure. A *diagnostic harness* is then created by inserting all the numbered `DP` instances into the function block using `rewire()` commands. A belief is also established for each function block that has diagnostic points that can be monitored.

The literature review discussed models of ICPS and the techniques of fault-finding that involved invalidating one or more aspects of the model. Creating models is labor-intensive (Milis et al., 2016; Mugan, 2014). Using interaction belief skills and domain knowledge of the IEC 61499 function block architecture, the agents construct their own model of the application they are examining. For fault diagnosis, first-order beliefs are sufficient to form this model that

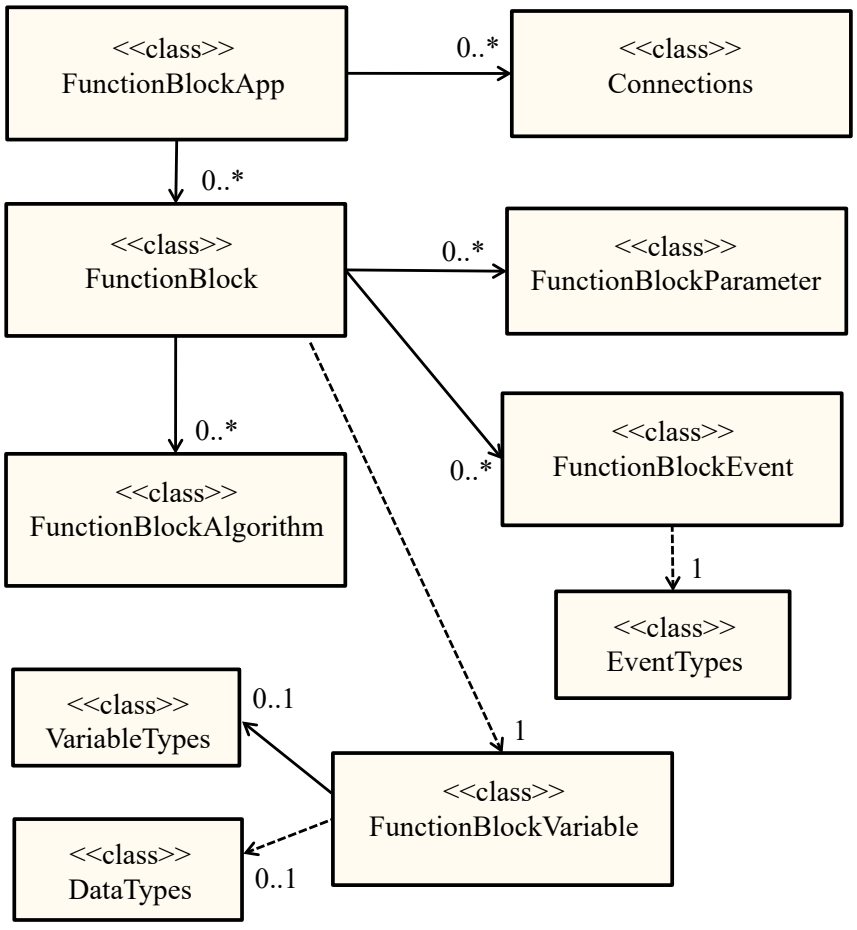

Figure 7.3: FunctionBlockApp Class Diagram for the graph and its nodes.

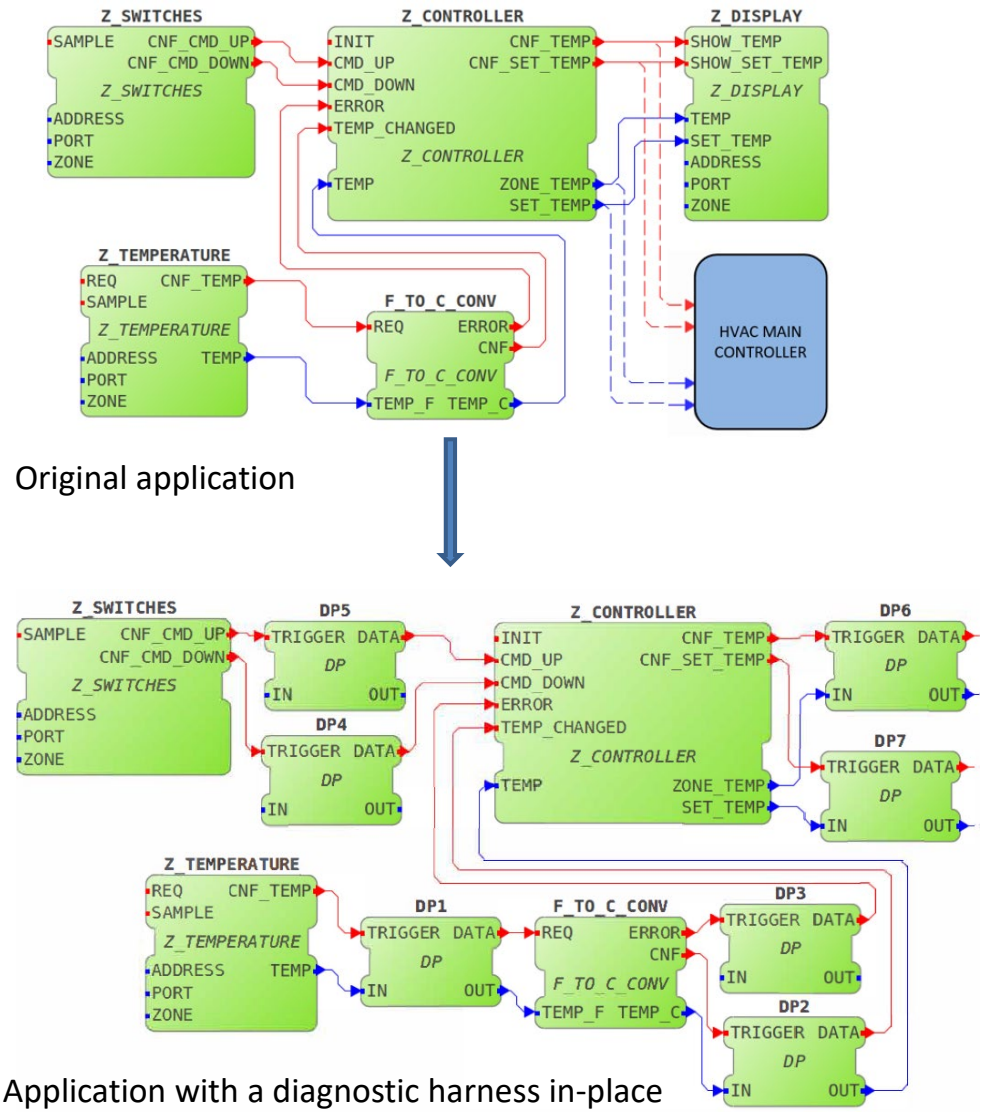

Figure 7.4: Before and after example of the `rewire()` interaction belief.

encompasses both the design of the ICPS that the agents are examining as well as sufficient knowledge of the physical environment the ICPS is interacting with. The agents gather all their knowledge of the physical environment via the ICPS itself.

7.1.3 Beliefs about what is happening

The agent forms Dynamic Diagnostic Beliefs as it observes that is happening and exercises function blocks of interest. During fault-finding tasks, the agent forms new beliefs as well as

modifying its existing beliefs. Dynamic Beliefs are fuzzy logic opinions about a particular function block, sensor, actuator or algorithm in the function block application.

Definition 7.1.4 (Dynamic Diagnostic Belief) *A belief $b = \langle \triangle, v \rangle$ is a dynamic diagnostics belief if $\triangle$ is represented as a pair (FBi, f_q) where:*

- *FBi is a function block instance of the system under diagnosis and*
- *f_q is a valid fault code for FBi, obtained from a set of fault codes F and.*
- *v is the veracity of the belief held by the agent based on what it has observed. This may be $true$, $false$ or $undetermined$.*

Genesereth et al. (1994) discuss a deeper aspect of veracity that is an intrinsic assumption that can be made about an agent. An agent must not only know what the truth is, it must also always tell the truth. That means our definition of reasoning does not have to include ideas of ethics and honesty; the agent is not sentient or motivated by its own self-interest above that of its goals. While the veracity that the agent holds at any time may be $undetermined$ or verified as $true$ or $false$, that belief can always be assumed to be an accurate judgment. Its v is only limited by the abilities of the agent to accurately capture and process the telemetry available and the limits of its System-Under-Diagnosis belief set.

The set $B = \{b_0, b_1, ..b_n\}$ captures the agent's beliefs about the function block application where b_i represents either a single Basic Function Block or a network of blocks connected as a discrete Composite Function Block. A Basic Function Block represents the smallest unit of functionality an agent can test and hence beliefs are atomic at this level.

Before the engine instructs the agent to begin operating, the agent is provided with a set of goals that include monitoring for fault signatures and executing diagnostic plans. The agent is also provided with a definition of what constitutes normal behaviour for this function block application. One example of normal behavior for the room controller shown in Figure 7.4 is the appearance of temperature readings at 500ms intervals from `Z_TEMPERATURE TEMP`. These propagate through the controller to appear at `Z_CONTROLLER ZONE_TEMP`. Even when a

temperature change has been requested, signaled by either a `CMD_UP` or `CMD_DOWN` event, the HVAC Industry ASHRAE Standard 55 specifies the optimal occupant thermal comfort rate of change should be a $\Delta t \pm 0.3^{o}C$/min (ASHRAE, 2004).

The agent pursues its GORITE `Monitor` goal, launching the function block application to run with its diagnostic harness enabled. All DP instances begin passing events and data through transparently to other function blocks as well as back to the agent. The agent establishes an initial belief for the `Monitor` goal such that:

$$b_0 = \langle FBZ\_\text{TEMPERATURE},\ v_{\text{undetermined}},\ f_0 \rangle$$

The agent continues pursuing this goal until one or more of its primary beliefs about normal operation are invalidated. If the result of the agent using the primary monitoring plan to look for normal behaviour determines that the function block application is operating within tolerance then $v_{\text{undetermined}} \rightarrow v_{\text{true}}$ as the agent reinforces this primary belief. Other primary beliefs monitor the response from other parts of the function block application. For an agent, the pursuit of its `Monitor` goal is the repeated re-evaluation of each of its beliefs by running the prescribed tests at defined intervals.

Invalidating any one of these beliefs in B causes the agent to signal its GORITE Team Manager that it has abandoned its primary `Monitoring` goal and that it is awaiting a new goal. If it is instructed to being a fault investigation goal such as `Diagnose`, the agent then adopts a divide-and-conquer strategy for exploring in the temperature sensor subsystem of the room controller. The agent first issues `gateClose()` commands to some of the `DP` instances within the temperature sub-system. One diagnostic test triggers `F_TO_C_CONV` to exercise its events, inputs and outputs. A range of nominal Fahrenheit temperatures are injected via `DP1` and captured as Celsius values at `DP2`. Out-of-range values such as absolute zero ($-459.67^{o}F$) should trigger the `ERROR` event, captured by `DP3`. If all test values are converted and captured correctly, the agent updates the v of the $b_{F\_TO\_C\_CONV}$ to `true`. The agent continues down the data path checking each subsequent function block and updating its beliefs until all the function blocks have been tested. This process caters for the possibility of multiple fault candidates. In

subsequent `Analyse` and `Report` goals, the agent proposes a diagnosis after iterating each belief to examine its veracity.

7.2 Conclusions and planning for the next phase

The Software Architecture Document proposed the design of belief structures which have been explored in more depth in this Design Science phase. Belief creation and management provide a mechanism that allows an agent to remember what has happened during fault-finding. As beliefs are built up, they can be re-considered to decide which steps to take next. The next phase implements the language that the agents use to perform their fault-finding tasks and refine their beliefs as they gather evidence. It includes an ATAM that evaluates the belief structures and the diagnostic language constructs that they use to interact with function block applications.

Chapter 8

How Agents Act

"When I finally find that one willing agent, I'll have found my prize in the Cracker Jack box."

Richelle E. Goodrich alluding to Jim Steinman while speaking about her own writing (2015).

The previous chapters discussed the creation of the agents, what they believe, and the diagnostic points they use to interact with function blocks. This chapter now continues to address RQ3 and extends the earlier discussion of agents and their beliefs from Chapter 7. It profiles how the agents use those skills and resources to gather information about faults they uncover. It presents the language that was created to enable the agents to interact with the application they are examining, and how diagnostic resources are created while designing functions blocks.

Coordinating the actions of the agents and synchronising their interactions with function block applications presented some significant challenges. The Introduction chapter noted Lee and Seshia's observation that it is at the intersection, not the union, between the cyber and the physical that we must understand these entities more deeply (Lee & Seshia, 2016). The architecture of the diagnostic points proved to be robust, but matching a test input to a resultant output consistently and reliably required careful consideration and many long, contemplative walks on cold nights.

In a way similar to the earlier chapters, an ATAM was used to evaluate this approach

to enabling the agents, their capabilities, the diagnosis language, and to address the issues encountered. This laid the foundation to allow subsequent chapters to then present the design and analysis of two example systems in more detail. These later chapters demonstrate the complete Model-Driven Development with Diagnostics methodology in practice and illustrate the scope of the engine's capabilities.

8.1 Creating Diagnostic Resource Packages

Understanding the diagnostic needs of a function block begins by considering the way the block will process the data it receives. The IEC 61499 Function Block reference architecture provides a formal definition of what a data input, a data output, and an event represents (IEC, 2013a, 2013b). Events, data encapsulation, strict data typing, and the use of a finite-state ECC help to encourage good design practices (Dubinin & Vyatkin, 2008; Preuße et al., 2010; Moore et al., 1956). These characteristics of IEC 61499 reinforce the formal aspects that define the architecture and ensure it remains robust.

This research introduced the concept of a Diagnostic Package in earlier chapters. At design time, the function block is visualised in one of the IDEs such as 4diac (Strasser et al., 2008) or nxtCONTROL (nxtControl GmbH, 2016). The ports that accept data into the function block and the output ports that deliver data after processing define possible telemetry data capture and injection points. When these are identified by the designer, they are captured in a file that augments the information in the function block type definition files, which have a file extension of `.fbt`. Diagnostic packages are XML-format files whose file name matches that of the function block type name. They have the extension `.dpg`.

Figure 8.1 shows two of the function blocks that feature in later versions of the HVAC Room Controller. The `TEMPERATURE` block is a SIFB that is able to capture a simulated temperature reading from the simbIoTe simulator. The second block `F_TO_C_CONV` accepts a Fahrenheit temperature reading and converts it to Celsius. The blue squares identify potential telemetry capture points, matching the data input to the event that triggers it or the output port to the event

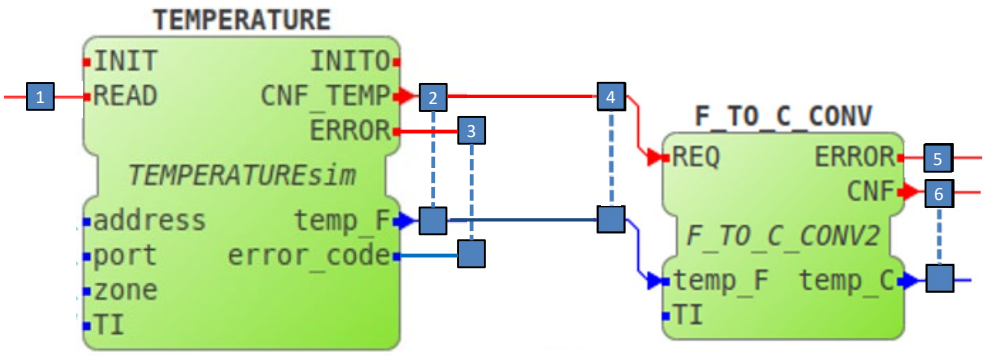

Figure 8.1: Potential Diagnostic Points for the function blocks.

that confirms it.

8.1.1 How agents manage their tasks

Chapter 5 explained how the Team Manager agent assigns goals to its team of independent agents, rescheduling their work as goals are either completed or changed. The types of goals that can be assigned to an agent fall into four broad categories. They include configuring a function block application, monitoring for faults, intervening to investigate a problem, and reporting a diagnosis. Once a goal is assigned to an agent, they are responsible for selecting the actions that will achieve their current goal successfully (Jarvis et al., 2013, 2018).

Humans make decisions about what they should do by first understanding what the task involves and what is expected as a "deliverable" from completing it. They then consider the options available to them and choose one approach so they can start working. Consider the task of measuring the frequency at which a light is flashing on a control panel. A human might decide to go and observe the light, counting the flashes for a period of time. They would return with a belief about what they had observed. Their belief could have a veracity v of $true$, signifying that the light was indeed flashing. They could reinforce that belief b_n with evidence that included the number of flashes per second observed. Alternatively, the worker could return with a belief that the light had failed, leading to a v of $false$. In both cases, their goal would be deemed to be complete. However, in the worst case scenario, the worker may not be able to locate the light or perhaps the control panel in the plant. In that case, the v would remain $undetermined$ and

the worker's goal $Goal_n$ would have a Goal State gs_n of $failed$ or $suspended$ since they could not complete the task.

This approach to delegated, autonomous goal management has been replicated in a similar way for the agents using GORITE and the extensions detailed in Chapter 5. In this research, Dynamic Diagnostics Beliefs have become the fundamental unit of fault information that the agents manage and exchange with each other. When an agent is assigned the goal of monitoring for faults, it begins by retrieving diagnostic instructions from its System-under-Diagnosis belief structure. This provides information about which function blocks have diagnostic instructions available that it can access. Each set of instructions is written as a method that was made available in the diagnostic packages created by the function block application engineers. The agents watch for faults by executing each set of instructions cyclically until a fault is discovered. When that happens, the agent terminates its monitoring goal and returns a Dynamic Diagnostics Belief that details the fault back to the Team. It then awaits re-assignment to investigate the fault further.

The agent could return a veracity v of false signalling that no fault had been found in that particular function block. If the agent is traversing through multiple function blocks, it would continue searching for other faults until it reaches the end. At that point, it would report back a single belief of that either no fault had been found or provide consolidated fault information. The paradigm implemented for belief exchange always returns a single fault to Team Manager when a goal either completes or fails.

In the earlier example, the human worker did not necessarily have to know what makes the light flash to complete their task successfully. Their stated goal was to observe, which involves rote actions rather than deep reasoning. In a similar way, the agent only has to execute the steps in the instructions and return a belief about what it has observed. While this mechanical behaviour allows the agents to perform efficiently, they operate more like an Expert System than an AI (Bertolini, Mezzogori, Neroni & Zammori, 2021; Calegari, Ciatto, Mascardi & Omicini, 2021). No pragmatic reasoning is being performed, but the beliefs they return form a diagnosis that is useful to the designers and test engineers.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBDiag SYSTEM "DiagnosticElement.dtd">
<FBDiag Name="TEMPERATUREsim" Comment="Diagnostic information for TEMPERATUREsim ver 1">
    <DPS>
        <DP Event="READ"/>
        <DP Event="CNF_TEMP" Port="temp_F"/>
        <DP Event="ERROR" Port="error_code"/>
    </DPS>
    <Diag>
        <DiagTest>
            <Name>Monitor</Name>
            <Code>Belief monitor() {</Code>
            <Code>if (TEMPERATURE_READ.readEvent()) {</Code>
            <Code>  if (TEMPERATURE_ERROR.readEvent(TEMPERATURE.READ.timestamp())) {</Code>
            <Code>      // The function block is reporting an error.</Code>
            <Code>      belief.Veracity(VeracityTypes.FALSE);</Code>
            <Code>      belief.add("Error code " + TEMPERATURE.ERROR.value());</Code>
            <Code>  } else {
            <Code>      if (TEMPERATURE.CNF_TEMP.readEvent(TEMPERATURE.READ.timestamp())) {</Code>
            <Code>          // The function block is returning temperature readings.</Code>
            <Code>          belief.Veracity(VeracityTypes.TRUE);</Code>
            <Code>          belief.add("Temperature " + TEMPERATURE.CNF_TEMP.value()</Code>
            <Code>                      + " in " + TEMPERATURE.CNF_TEMP.elapsedTime() +"ms");</Code>
            <Code>      }</Code>
            <Code>  }</Code>
            <Code>  return belief;</Code>
            <Code>}</Code>
        </DiagTest>
    </Diag>
</FBDiag>
```

Figure 8.2: Diagnostic Package definition sections in the TEMPERATUREsim.dpg file.

8.1.2 The structures within Diagnostic Packages

Figure 8.2 shows the XML-format sections of the `TEMPERATUREsim` diagnostic package file. The <DPS> segment group contains multiple <DP> definition segments that specify each interaction point for the `TEMPERATUREsim` function block. Attributes identify the event and the data input or output port associated with it that can provide telemetry. Specifying both the event and the port removes any ambiguity when an event is associated with multiple ports or a port is attached to multiple events. Note how in the the first example, the event `READ` is not linked to a port. The event is a trigger to cause the block to query the temperature sensor, which does not require an associated data input. Figure 8.3 shows the three DPs created by the agents when they process the TEMPERATUREsim.dpg file.

A single diagnostic package definition may result in multiple discrete diagnostic points being instantiated and wired into the application when the test harness is created by the agents.

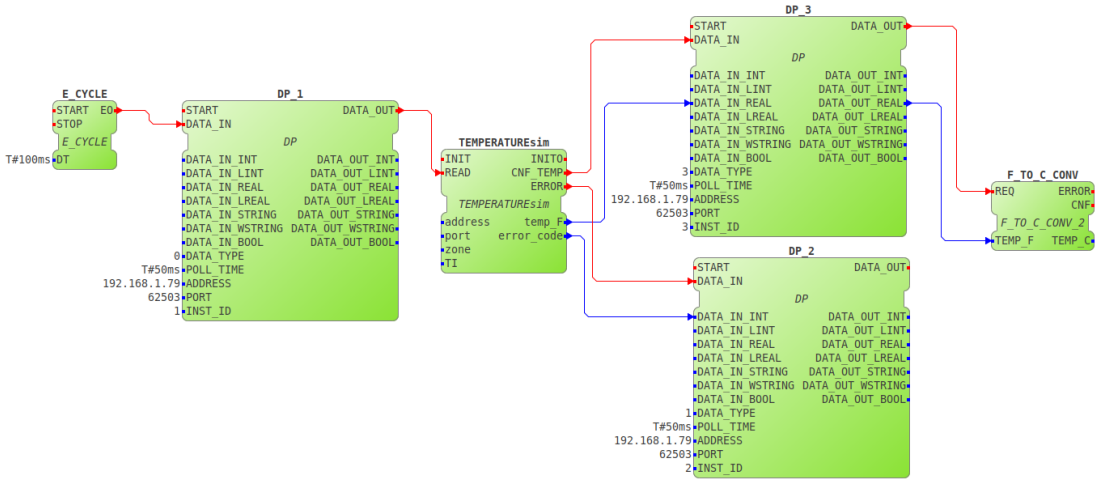

Figure 8.3: Diagnostic Points created by the agents from TEMPERATUREsim.dpg

The connections between the function blocks that `TEMPERATUREsim` was originally wired to have been broken and re-connected to insert the DP instances.

8.1.3 The diagnostic task language

The actions that an agent can take to monitor or interact with a function block via a diagnostic point are expressed in a Java-based language developed during this research. It is object-oriented and allows each diagnostic point to be referenced as a named object. Figure 8.4 shows an example of the source code for monitoring a `TEMPERATUREsim` type function block.

A diagnostic point is referenced using its instance name and the name of the event that it is observing. The read() method is overloaded for each IEC 61499 data type to ensure that it casts the data correctly to the objects *value()* property. The timestamp for the event is also captured and made available via the *elapsedTime()* property. The elapsed time value is calculated with reference to the time of the READ event which caused the function block to emit the corresponding output `CNF_TEMP` or `ERROR` event.

The source code format shown in Figure 8.4 is not able to be executed directly by the agents yet. A code pre-processor is currently under-development to allow the engine to dynamically compile and load each function retrieved from a diagnostic package. The object code that

```
Belief monitor() {
    if (TEMPERATURE.READ.read()) {
        if (TEMPERATURE.ERROR.read(TEMPERATURE.READ.timestamp())) {
            // The function block is reporting an error.
            belief.Veracity(VeracityTypes.FALSE);
            belief.add("Error code " + TEMPERATURE.ERROR.value());
        } else {
            if (TEMPERATURE.CNF_TEMP.read(TEMPERATURE.READ.timestamp())) {
                // The function block is returning temperature readings.
                belief.Veracity(VeracityTypes.TRUE);
                belief.add("Temperature " + TEMPERATURE.CNF_TEMP.value()
                        + " in " + TEMPERATURE.CNF_TEMP.elapsedTime() +"ms");
            }
        }
    }
    return belief;
}
```

Figure 8.4: Source code for monitoring a `TEMPERATUREsim` function block.

is produced is Java 11 compliant and will be compiled into byte-code before loading it with the Java Reflection() methods. At this stage of the research, the source code is manually converted into the object code required so that is can be run by the agents on-demand. The post-doctoral development work planned for the compiler is discussed further in Chapter 11. The function returns a Belief object for the Dynamic Diagnostic Belief b_q that carries the v as well as additional free-text information. Each invocation of the *belief.add()* method creates an additional line of information separated by a new line character.

8.2 Gimbaling

The second function block `F_TO_C_CONV` has a diagnostic package with a different type of script in it. *Gimbaling* is a term adopted for this research from the aerospace industry (Boeing, 1998). It is used to describe the test of an actuator on an aircraft or spacecraft that exercises its motion between its extremes. Figure 8.5 shows a Rocketdyne RX-25 Space Shuttle engine gimbaling to port during a test firing (NASA, 1995). During

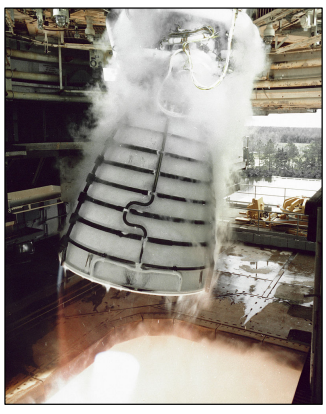

Figure 8.5: Space Shuttle RX-25 engine gimbaling.

pre-flight checks of flight surfaces, a pilot *gimbals* the rudder and ailerons of the aircraft to ensure that they respond to commands and move smoothly to both of their range limits. In a similar way, a function block can be gimbaled by supplying a comprehensive set of test values that force it to process the values through their entire range. This exercises the algorithms and the ECC, exposing both boundary, state management, and coding issues. Figure 8.6 shows the DPs inserted by the agent that surround `F_TO_C_CONV` to capture its operation.

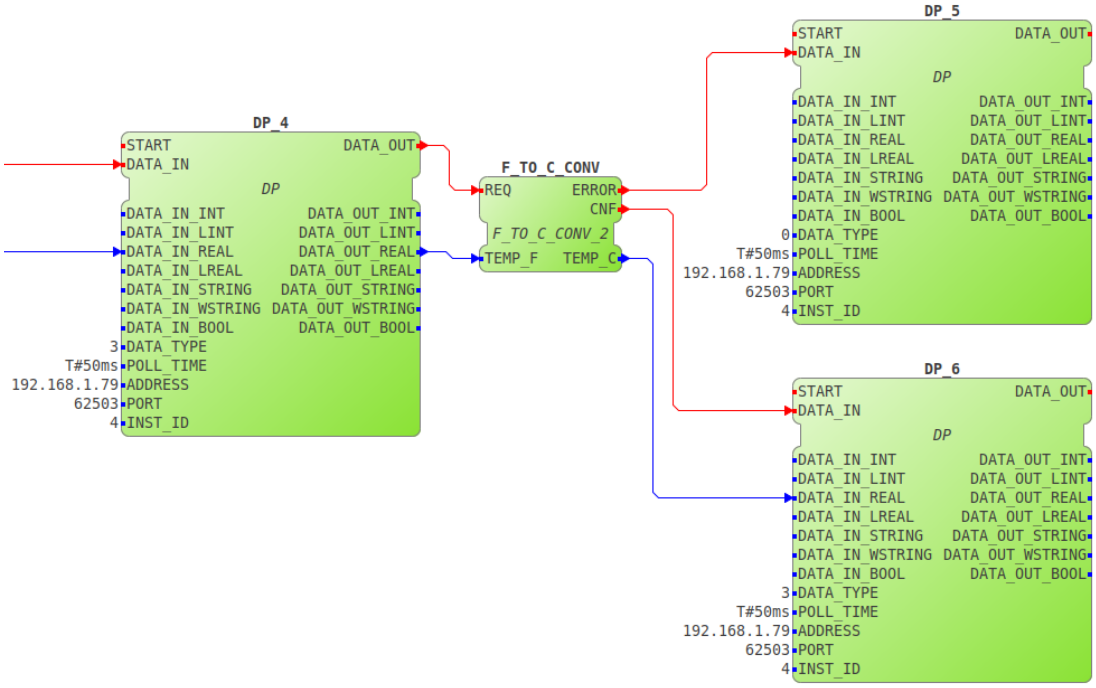

Figure 8.6: Diagnostic Points created by the agents from F_TO_C_CONV.dpg

When an agent is assigned a potential fault to investigate, the gimbal scripts allow it to walk through a fault path, isolating and exercising each function block to determine if it is operating correctly. Figure 8.7 shows the code that gimbals the `F_TO_C_CONV` function block. The F_TO_C_CONV.gateClose() command temporarily blocks `TEMPERATURE` from generating temperature readings for `F_TO_C_CONV` to process. This switches DP_4 from its normal transparent passthrough mode to now accepting test values to trigger the block.

The *gateClose()* command determines if the DP is managing an input to the function block or an output from it. This determines if it will block signals coming into the diagnostic point from other function blocks or if it needs to block signals going out of the diagnostic point. This ensures that testing `F_TO_C_CONV` with a wide range of values does not adversely affect other

```
Belief gimbal() {

    const absZero = -459.67; // Absolute zero Fahrenheit.
    double temperatureF = absZero;
    double expectedTemperatureC = 0;
    double temperatureC = 0;
    int errorCount = 0;

    F_TO_C_CONV.REQ.gateClose();

    while (temperatureF < 100.0) {
        expectedTemperatureC = (temperatureF - (double) 32.0) * (double) 0.555;

        F_TO_C_CONV.REQ.trigger(temperatureF);
        // Allow the event to process
        delay(100);

        if (F_TO_C_CONV.CNF.read(F_TO_C_CONV.REQ.timestamp())) {
            // Retrieve the Celsius temperature from the temp_C data port.
            temperatureC = F_TO_C_CONV.CNF.value();
            if (temperatureC != expectedTemperatureC) {
                errorCount++;
                belief.add("Error : temp_C should be " +
                           expectedTemperatureC + " instead of " +
                           temperatureC);
            }
        }

        if (F_TO_C_CONV.ERROR.read(F_TO_C_CONV.REQ.timestamp())) {
            if (temperatureF > absZero) {
                errorCount++;
                belief.add("Error returned after converting " + temperatureF);
            }
        }

        temperatureF = temperatureF + (double) 7.5;
    }

    F_TO_C_CONV.REQ.gateOpen();
    if (errorCount == 0) {
        belief.Veracity(VeracityTypes.TRUE);
        belief.add("Nominal.");
    } else {
        belief.Veracity(VeracityTypes.FALSE);
    }
    return belief;
}
```

Figure 8.7: Source code for gimbaling the `F_TO_C_CONV` function block.

parts of the application. The loop then exercises the block by incrementing the temperature from absolute zero to 100°F. The trigger() command supplies each test value to the `temp_F` input via the DP. The converted Celsius temperature is captured from the data output `temp_C`

when the event CNF fires. Testing in this way ensures that algorithms such as the Fahrenheit to Celsius conversion, which are implemented inside the block in C++, correctly cast their data types accurately for all cases.

Errors using floating-point numbers and other coding errors are a potential source of faults. These can go undetected during development if the boundary and incremental diagnostic testing is not rigorous enough. Absolute zero Fahrenheit should be the only temperature that triggers the ERROR event. The script keeps count of the number and types of errors, which is used to determine the v of the belief that is returned after the gimbling completes. The gateOpen() command restores the transparent passthrough mode since other function blocks that will be tested may rely on inputs from other downstream activities.

8.3 Dynamic and static considerations of agent actions

Belief management has always been a concern in traditional BDI systems, particularly because the modification of a belief could be complex and impose overheads that impact the performance of the agent. In BDI implementations such as JACK (Howden, Rönnquist, Hodgson & Lucas, 2001) and Soar (Georgeff et al., 1998), the beliefs of an agent are predominantly static since they represent information about aspects of their environment which do not change rapidly. However, of the three agent belief types discussed in this research, only the Interaction Belief type discussed in Section 7.1.2 is static. This belief type is used to encapsulate IEC 61499 domain-specific skills. Once learnt, those skills are not modified by any activities the agent performs or the interactions that the agent has with the function block application.

Dynamic beliefs are a core feature of the Fault Diagnostic Engine that drive agent interactions with the function block application. Agents use dynamic beliefs which they can modify based on diagnostic evidence they gather during their fault-finding activities. Dynamic beliefs created during this research are of two types:

System-Under-Diagnosis beliefs, discussed in Section 7.1.2, encapsulate beliefs that the agent has formed about an individual function block instance that is part of the function block application under diagnosis. These beliefs carry information about the structure and characteristics of a particular function block. Each function block can be considered to be a node in a directed graph, where each edge of that node represents its connections to other function blocks.

Dynamic Diagnostics Beliefs, discussed in Section 7.1.3, capture the results of an agent's fault-finding while examining a particular function block. These have a veracity or degree of truthfulness v that is dynamic; the agent can revise such beliefs when new fault evidence is captured and evaluated.

The evaluations reported later in Chapter 9 and Chapter 10 showed that rather than impacting the performance of the agents, this approach to belief management actually enhanced their performance. The agents were custom designed around these belief structures to make the management of beliefs simple and highly efficient. GORITE does not provide templates for agents, so this provided ample opportunity to carefully design and tune the agents from their inception to be high-performing and have efficient belief management. Also, since the agents are not running natively on the function block application platform, they do not significantly impact the latency of the function block application under diagnosis. As noted in the Introduction, this is an improvement on the approaches used earlier by Kalachev et al. (2018), Jarvis et al. (2018), and Christensen (2017).

8.4 Evaluating the agent action architecture

Monitoring and gimbling both require the agent to be able to correctly match an input event to a corresponding output event. The event of interest can either be on the same function block or, in more sophisticated scripts, a different one further down the data path. That matching initially proved to be problematic. The issue relates to the way that a DP function block returns telemetry back to the agent via the FIFO network queues. Figure 8.8 shows the contents of the FIFO network packet queues during a typical gimbaling diagnostic session.

The diagram shows the agent using the F_TO_C_CONV.REQ.trigger(temperatureF) command to inject a value of 80°F via DP_4. It expected that the converted temperature returned via DP_5 should be 26.67°C. However the next entry in the FIFO packet queue for DP_5 is a reading of 51.25°C. That packet was generated before the gate on DP_4 was closed to block data from the normal operation of the function block application. The correct matching result was generated during the test, but it is on the bottom of the FIFO queue. Attempts to flush the FIFO queues via program commands from the agent proved to be ineffectual.

The solution was to use the timestamp that was developed during the creation of the diagnostic points in Chapter 6. This was originally designed to allow the agents to verify the timing requirements of function blocks. Note that the correct result has a timestamp that is five milliseconds after the timestamp of the trigger packet. All other entries in the queue occurred before that time. The read() command was redesigned so that it discards all irrelevant packets and stops when it has found the first packet with a timestamp greater than or equal to the trigger timestamp. This approach also works if the block has not generated the result immediately. That situation would occur if the algorithm that processes the `REQ` event takes a significant time to

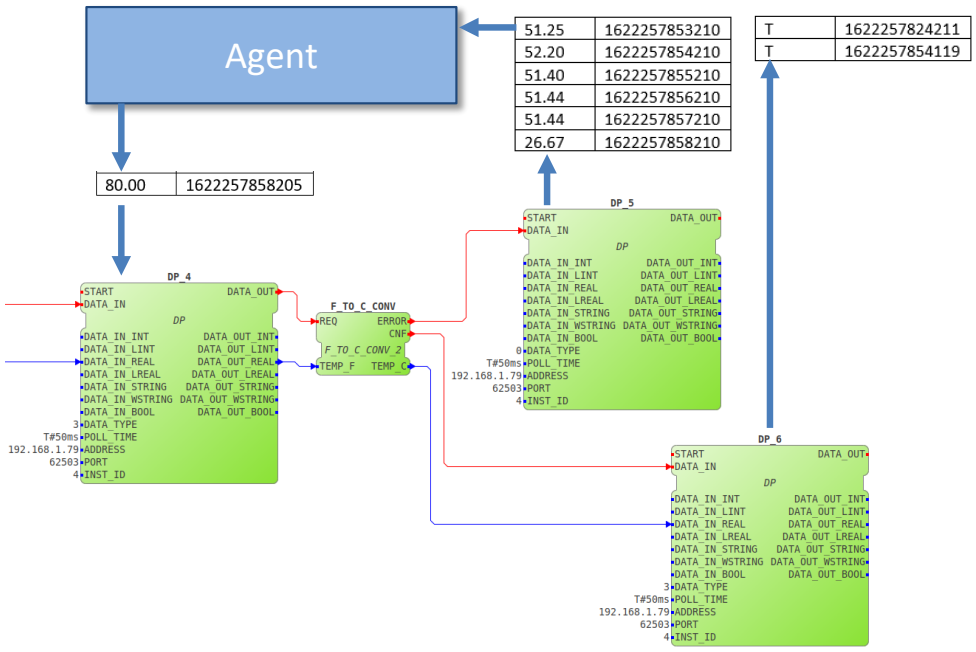

Figure 8.8: FIFO network packet queues during a gimbal diagnostic test.

Table 8.1: ATAM for evaluating the agent action architecture.

Quality Attribute	Concern	Trade-offs	Evaluation and solution
Performance, Timing, Accuracy.	Errors in synchronisation between inputs and outputs due to excess packets in the FIFO queues.	None.	Use the timestamps to detect response packets to trigger() commands which are older than than the trigger timestamp.
Functional Appropriateness, Ease-of-Use, Functional Suitability.	Ensuring that the diagnostic script language is as simple and expressive as possible.	A sophisticated language pre-compiler will be required. -correction and ease-of-implementation.	A lot of the information that the pre-compiler would need to check the integrity of the is available in the system-under-diagnosis belief structures already.
Ease-of-use.	The code editor used by the engineers to create diagnostic scripts needs to be highly-interactive.	IDE development adds complexity, but this does not impact runtime performance of the engine.	A lot of the information that the editor and pre-compiler is available in the system-under-diagnosis belief structures already.

execute. In that case, the read() command empties the FIFO queue without finding a packet match. It then initiates a retry wait automatically and returns after either finding a valid packet or timing-out. Table 8.1 summarises the other parts of the ATAM evaluation that examined the needs and performance, ease-of-use, and other aspects of the agent's action language.

Fritzson et al. (1998) note that simulation monitors such as the approach that was adopted in the engine are well-established for verification. The rich information available in the system-under-diagnosis belief structures assists in this process. They discuss industry-standard languages, such as Property-Specification-Language (PSL) and System Verilog Assertions (SVA), as examples that rely on domain-specific information. The system-under-diagnosis belief structure relies on similar formalisms to provide unambiguous information regarding the existence of, the data types, and nature of the external interface that each function block type provides.

8.5 Conclusions and planning for the next phase

The three prior chapters and this current chapter have detailed the implementation of all the significant architectural features of the engine. They have also explained how the agents

use these subsystems to perform their tasks. The remaining chapters profile Model-Driven Development in more detail by examining how well it supported the design and implementation of two example IEC 61499 function block applications.

Chapter 9

Model-Driven Development with Diagnostics in Practice

"In a word, the computer scientist is a toolsmith - no more, but no less. It is an honourable calling. If we perceive our role aright, we then see more clearly the proper criterion for success: a toolmaker succeeds as, and only as, the users of his tool succeed with his aid. However shining the blade, however jewelled the hilt, however perfect the heft, a sword is tested only by cutting. That swordsmith is successful whose clients die of old age."

Frederick P. Brooks, Jr. "The Computer Scientist as Toolsmith II" (1996, page 2).

As Brooks wisely asserts, tools such as the engine are only of value if they support their users' well. The design and creation of an ICPS is a pragmatic discipline, demanding robust methodologies and sound evaluation practices to ensure the delivery of a dependable solution. Excellent engineers are risk adverse: the concepts of Model-Driven Development with Diagnostics would have to make sense to them before they will ever consider adopting them as accepted practices. This chapter profiles the methodology in detail, developing an example system to profile each step of its creation. Portions of this chapter were adapted for publication in the IECON 2020 conference paper *Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems* (Dowdeswell, Sinha & MacDonell, 2020a). The traceability techniques for function blocks used in this chapter were published previously in a journal article for the ACM Transactions on Cyber-Physical Systems entitled *TORUS: Scalable*

Requirements Traceability for Large-Scale Cyber-Physical Systems. (Sinha et al., 2018).

Brooks reflected in his earlier writing on the "*plasticity of code*", describing the delight of working in such a "*tractable medium*" (Brooks, 1975, page 2). This ability to refactor code, both easily and often, encourages iterative design practices. However, refactoring an ICPS is not hacking or thrashing at code trying to make it work with recalcitrant hardware. Fowler (2018) asserts that the refactoring of artefacts like these has to rest on sound foundations where requirements are clearly defined and understood. Since the cyber parts of an ICPS interact deeply with their environment, the physical characteristics of transducers need to be understood in-tandem with the software requirements.

However, the earlier scoping literature survey demonstrated that software, not hardware, is now the predominant driver of innovation in ICPS (Braun et al., 2014). Hill and Menk (2016) expected that by 2020, software would constitute 80% of the value of a motor vehicle, which is now arguably an ICPS. The Stout 2020 Automotive Warranty and Recall Report illustrates the implications of this shift (Steinkamp, 2020). Steinkamps analysis shows that for the last two years, software, not hardware, defects led to the largest number of single-issue vehicle recalls. The report states that the number of recalls caused by software issues rose from five million in 2009 to fifteen million in 2019. Faults such as these cannot always be diagnosed or rectified easily in the field.

Throughout this thesis, the role of the engine and its agents has been proposed as a way to blend appropriate modeling with fault diagnostics for the design, creation, and long-term support of ICPS. The focus has been on learning how to construct and diagnose ICPSs that use the formal IEC 61499 reference architecture. An example IEC 61499 function block application was developed from its earliest conceptual stages after first defining clear requirements and quality attributes. As each function block was then crafted, its characteristics and diagnostic needs were evaluated. At the same time, matching diagnostic resources were created. When each block was ready, it was then wired into the evolving function block application. The agents then exercised the connected function blocks, providing the degree of quantitative feedback expected in test-driven development (Y. Zhang, 2004). Traceability was established from each

requirement to one or more of the function blocks that satisfy the requirement. The development phase included both simulation and development of the components on the final hardware platform. The room controller hardware and software were then verified against the original design specifications. Finally, a field trial of the completed controller was performed. Agents monitored the room controller, evaluating and logging the activities they observed as it interacted with the room occupants and the environment changed around it.

This approach encouraged the creation of function blocks that are *diagnosable-by-design*, echoing Hamilton and Zeldin's *correct-by-construction* paradigm (Hamilton & Zeldin, 1980). In parallel with the creation of this example system, the same reflective consideration was applied to the engine and the agents themselves. The initial design of the agents and the way they operated was refined as the interaction they had with the function blocks was understood more clearly. This provided another level of validation of the architecture while identifying practical changes needed to parts of the engine.

9.1 The HVAC room controller subsystem

The ICPS used to illustrate the methodology presented in this chapter is a room controller for a Heating, Ventilation and Air-Conditioning (HVAC) system. HVAC room controllers are self-contained devices that interact with the complex plant machinery that provides climate control for entire buildings. The Mitsubishi Corporation PAR-33MAA-J room controller shown in Figure 9.1 is typical of the room controllers available today (Mitsubishi, 2021).

Figure 9.1: PAR-33MAA-J

In HVAC systems, room controllers are *edge devices* that monitor the environmental conditions in a designated zone, interacting with its occupants. Edge devices operate out on the periphery of an ICPS network, performing computational and control tasks independently in

their local environment (Shi & Dustdar, 2016). Their design qualifies them as being an ICPS since they rely on telemetry from sensors, electro-mechanical control of heating and cooling plant equipment, and communications with external systems. Push-button or touch-screen actuators are used to request changes to the room temperature by the occupants. This ability to interact with their physical world, distribute capabilities, and share tasks allows devices such as these to scale well and be highly-responsive to their environments.

9.2 Model-Driven Development stages in-context

Most design and development tasks for ICPS are complex optimization problems for real-time systems (Zave, 1982). The operating characteristics of sensors and actuators add another level of complexity to both the functional and quality requirements (Vogel-Heuser, Schütz, Frank & Legat, 2014; Cabrera et al., 2010). This is because an ICPS often operates under strict timing constraints where function blocks have to interact with real-world events that are both asynchronous and exhibit parallelism (Vyatkin, 2011). The actions they perform in their environment therefore demand rapid, timely responses and often, high-precision. Hence, designers must remain cognizant of not only software constraints, but also the physical and electrical characteristics of the devices they are interfacing with and relying on. Modelling addresses these needs by providing ways of creating diagrams and entities that explore the design space, including a consideration of possible faults. These perspectives then allow alternative options to be considered in constructive ways. Models also help other, often less technical, stakeholders reason about the design decisions that have to be made.

The scoping literature survey highlighted Provan's (2014) concerns about the failure to integrate diagnostic modelling early enough in the requirements process. Provan also discussed the problem of ambiguities in the models themselves at run-time. Model-Driven Development with Diagnostics seeks to address this by encouraging the parallel development of diagnostic resources alongside the design of the function blocks themselves. The iterative stages of the methodology have been refined during this research by building function block applications

such as the HVAC room controller to trial the process steps.

Developing an ICPS is arguably as much an art as it is a science, where form and function matter in differing degrees (Townsend, Montoya & Calantone, 2011). In each of the stages that are detailed in the following sections, design decisions are influenced by more than just engineering concerns. How an ICPS presents data to a user has aesthetic aspects that must influence the design of the interface. Some of those "users" are the agents who are observing the room controller operating and our premise in this research is that an ICPS should be diagnosable-by-design to make their tasks easier and more effective. A good design should co-operate as much as possible with its designers and users to make validation and verification more robust. To achieve this, those foundations have to be laid early in the design phases and matured through each of the subsequent stages that realise the completed ICPS. Instinct and experience on the part of the engineers play a vital part in this process. Isaac Newton observed that "*Truth is ever to be found in the simplicity, and not in the multiplicity and confusion of things*" (Newton, 1687, page 120). A design should be realised as the simplest artefact that satisfies the requirements well.

9.3 Applying the V-Model to ICPS development

The scoping literature survey found examples of software and hardware development projects for ICPS that adopted variations to the traditional V-Model development methodology (Modest & Thielecke, 2016; Abel et al., 2013; Neto et al., 2018). This software engineering methodology originated in Germany as *Das V-Modell* in the 1990s (Bröhl & Dröschel, 1995). The approach augments traditional Waterfall methods by coupling iterative steps with complementary evaluations. It has seen wide adoption in the ICPS community, where Agile approaches to ICPS development are still in their infancy (Ahmad, 2019). Lee (2006) suggests that this is partly because an ICPS requires us to re-think the way we model hardware and software development processes, which are often separate and disconnected in Agile paradigms. Hence the V-Model predominates, especially in the automotive and aerospace domains.

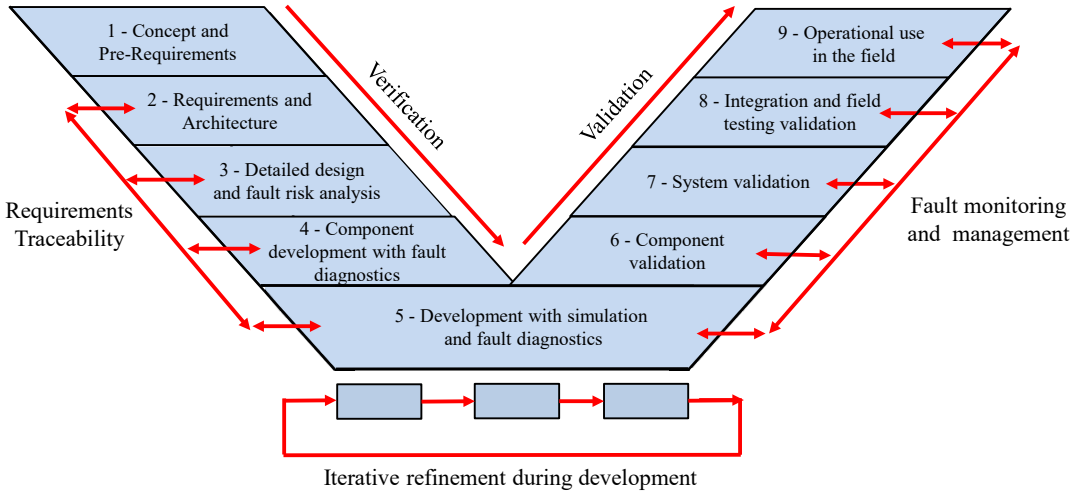

Figure 9.2: Blending traceability and diagnostics into the V-Model for ICPS.

The German government initiative VDI 2206:2204 *Guideline for V-Model Processes* outlines the recommended stages in an enhanced V-Model implementation (Ingenieure, 2004). Figure 9.2 shows the process flow of the V-Model that was adapted for this research, blending traceability with the design of fault identification and fault diagnosis into the left-hand side stages. In each stage, verification of the artefacts was performed before moving on to the next stage. During the horizontal Development phase, the agents performed diagnostic testing and evaluation while each function block was being created. The proposed flow is highly-adaptable, providing a process framework to support Agile sprints of development if desired. Graessler et al. (2020) propose practical ways of introducing Agile techniques into the V-Model development stage. They suggest that the way Agile encourages the on-going involvement with stakeholders, provides iterative sprints, and establishing a consensus on changes helps to address concerns that the V-Model is just another alternative to Waterfall.

Requirements traceability is demanded in safety-critical avionics and aerospace ICPS development. Standards such as DO-178C/ED-12C (ISO, 2011a) specify the audit and long-term management processes required that form a key criteria for air-worthiness certification (Youn et al., 2015; Brosgol, 2011). In the version of V-Model employed here, the scope of the traceability linkages is from the requirements phase through to the individual function blocks that satisfy those requirements. The initial requirements created in Section 9.5 are shown in Table 9.2. They

were refined during the design and development phases as each function block was created. The completed requirements list with the traceability references is shown later in Table 9.4.

The left and right arms of the V-Model are symmetrical. For each stage on the left-hand side, there is a complementary stage on the right-hand side that ensures that the deliverables from the stages are valid. For example, the focus in the Stage 4 Component Development stage is on the creation of individual function blocks. The complementary Stage 6 Component Validation right-hand phase focuses on ensuring that all those blocks actually work together now that they have been wired into the application during the Stage 5 development.

To ensure that the evaluations performed are sufficiently robust, the left-hand arm of the V-Model focuses on *verification*, determining if the system is being constructed correctly (Murphy, Wakefield & Friedman, 2008; Easterbrook, 1999). This includes ensuring that the requirements meet the stakeholders' needs and that both the architectural and engineering practices are sound. Being able to trace a requirement through to the components that fulfil it is a key part of this verification process. The corresponding right-hand arm of the V-Model then focuses on *validation*, determining if the system has been built correctly according to what the architects and designers verified with the stakeholders. During the Stage 4 Component Development and Stage 5 Development phases, the diagnostic agents support the designers and developers. Diagnostic testing helps them verify each function block component as it is created. Faults uncovered by the agents are remedied iteratively during these phases.

In the second half of the V-model, the fault diagnostic engine and the agents provide support during the component validation, the validation of the completed system, and the final integration and implementation operations. Later, the same diagnostic resources are available for long-term operational maintenance and fault-finding. Fault diagnostics and requirements traceability provide additional levels of support during these crucial, often complex, validation steps. Section 9.6.1 details the practical aspects of implementing and maintaining traceability within the V-Model. Each of the V-model stages used during the creation of the HVAC room controller is profiled in more detail in the sections that follow.

9.4 Stage 1: Concepts and Pre-Requirements

Pre-Requirements are statements, concepts, needs, and wants elicited from stakeholders. They are usually written in free-form natural language, are often incomplete, and are sometimes ambiguous (Vogel-Heuser et al., 2014; Zowghi & Gervasi, 2002). They form the starting point from which designers then begin to understand and explore their options.

The early ideas for the room controller included the concept sketch shown in Figure 9.3. Free-form sketches such as these are often used to complement the textual requirement statements (Wüest, Seyff & Glinz, 2012). They describe what is needed without considering how the ICPS might meet those needs (C.-W. Yang, Dubinin & Vyatkin, 2019). The first ideas about quality attributes often emerge during this phase. The artefacts created include informal diagrams and lists of issues as well as expectations of performance, usability, and quality factors.

The sketch illustrates how the control panel could display the current room temperature while the set temperature display will show what it should be. The panel accepts commands from occupants who can request changes to the temperature in their environment. Note how the sketch captures a quality requirement of sub-second response times for button presses and display updates. This was driven by the users' expectation that the room controller and its display should be "lively" and respond in near real-time to their commands. There is also an open question about the type of buttons that should be used. Physical switches add cost, but provide a different tactile response to users.

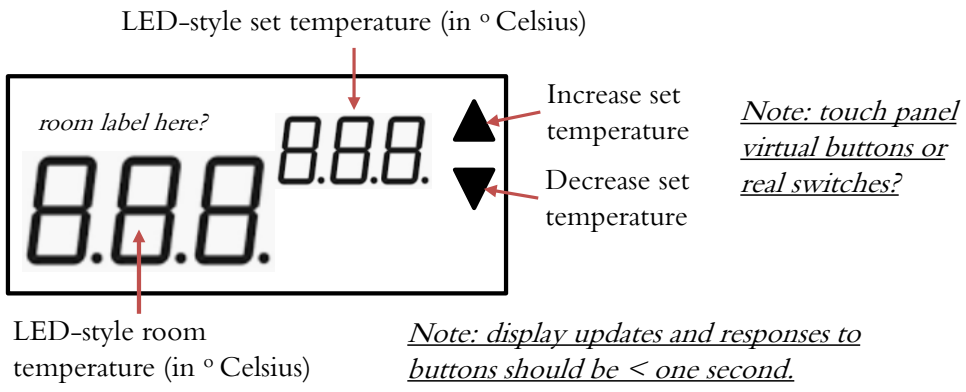

Figure 9.3: Early-stage concept sketch of the room controller.

9.4.1 Transitioning between phases

The V-Model and the traditional Waterfall model share many similarities, including the need to be able to transition smoothly and reliably between stages. Well-defined boundaries help to delineate progress while checklists ensure that a stage correctly addresses the scope defined for it. However, during this research, the practical activities in the Stage 3 Design and Risk Analysis stage and the Stage 4 Component Development stage often overlapped. A prototype of a component, even if this was only the creation of the function block type in the IDE, usually helped to clarify design questions. For more complex issues, coding a more detailed prototype was found to be valuable. Anitha et al. (2013) note that varying degrees of prototyping is one approach to managing the volatility of changing requirements for ICPS that the V-Model accommodates well.

Table 9.1 shows the transition criteria proposed for the HVAC room controller stages.The criteria for transitioning between phases was drafted during Stages 1 and 2. Attributes such as *completeness*, *consistency*, and *correctness* provide evidence that the tasks performed in the stage are complete-enough (Zowghi & Gervasi, 2002). The criteria chosen may also have to take into account aspects specific to the product. An example is the integration of the temperature sensor during the development stage; does the sensor chosen work correctly with the function block written to manage it? Paz and El Boussaidi (2017) provide a detailed example of the phase transition criteria for the V-Model they used while designing an aircraft landing gear control system. Since their environment is safety-critical, the phase transition criteria, along with the traceability framework, are required artefacts for DO-178C certification. Completeness and correctness are discussed further in Section 9.10.

9.5 Stage 2: Requirements and architecture

Pre-Requirements are then converted into formal requirements by creating a stakeholder-wide glossary of terms that make sense to the stakeholders. Table 9.2 illustrates the refined requirements for the room controller. These requirements rely on a glossary or *ontology* that names the

Table 9.1: Transition criteria for the V-Model stages

Stage	Transitions to	Criteria for transition to the next stage
1 - Concepts and Pre-Requirements	2 - Requirements and Architecture	• Agreement with the stakeholders that all their requests have been captured and noted.
2 - Requirements and Architecture	3 - Detailed design and risk analysis	• All functional requirements and quality attributes have been documented using agreed terminology. • The application architecture has been proposed showing how the function block application sub-systems meet the requirements.
3 - Detailed design and risk analysis	4 - Component development with diagnostics	• Function blocks for each sub-system have been designed. • Traceability links have been established between requirements and function blocks. • The risks that could lead to a fault in a sub-system or component.
4 - Component development with fault diagnostics.	5 - Development with simulation and fault diagnostics.	• All components are complete, ready to begin wiring the function blocks into the application. • Diagnostic testing has verified that individual components meet their requirements. • All simulation parameters have been loaded into simbIoTe.
5 - Development with simulation and diagnostics	6 - Component validation	• The function block application is shown to be operating nominally, communicating with the simulator and processing all telemetry generated by the simulator. • Real sensors and actuators have been evaluated and deployed to replace the simulated components.
6 - Component validation	7 -System validation	• Diagnostic routines verify that all production components are operating nominally.
7 - System validation	8 - Integration and field testing	• The completed application is deployed for trials in a representative environment. • Monitored by diagnostics to establish that the performance is nominal and that it meets both the functional and quality attributes defined for it.
8 - Integration and field testing	9 - Operational use in the field.	• The completed application is deployed for field trials. • Monitored by diagnostics to ensure that it continues to perform nominally for the maximum possible up-time.

proposed components and their attributes (Uschold et al., 1996; Noy et al., 2001). Ambiguous terms from the Pre-Requirements are replaced by well-defined terms such `HVAC_Controller` to identify elements, attributes such as `Temperature` and `Poll_time`, and operations such as `decreases` and `notify`. Derived from the ISO Software Engineering Product Quality standard ISO/IEC 9126 (2001), ISO Standard 25010 focuses on software-intensive systems such as ICPS. It defines relevant quality attributes such as Performance and Timeliness (ISO/IEC, 2011a). Table 9.2 therefore highlights the relationship between the functional requirements and the corresponding quality attributes that the design must satisfy.

There is a clear distinction between the way the Pre-Requirements explain to the designers

Table 9.2: Functional and quality requirements for the Room Controller.

Requirement and Quality IDs	Description	ISO 25010 Quality Attributes
RQ1	when Temperature.increases or Temperature.decreases then	
QA 1.1	HVAC_Controller.notify occurs in < 500 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
QA 1.2	DisplayedTemperature.update occurs in < 250 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
RQ2	Temperatures must be displayed in Celsius.	
QA 2.1	Temperature.unit = Celsius.	4.2.1 Functional Suitability 4.2.1.2 Functional correctness
RQ3	Set temperatures must be displayed in Celsius.	
QA 3.1	SetTemperature.unit = Celsius.	4.2.1 Functional Suitability 4.2.1.2 Functional correctness
RQ4	when SetTemperatureUp.pressed then	
QA 4.1	HVAC_Controller.notify occurs in < 500 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
QA 4.2	SetTemperature.update occurs in < 250 ms	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
RQ5	when SetTemperatureDown.pressed then	
QA 5.1	HVAC_Controller.notify occurs in < 500 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
QA 5.2	SetTemperature.update occurs in < 250 ms	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
RQ6	when Temperature > 176 Fahrenheit OR Temperature < -40 Fahrenheit then	
QA 6.1	DisplayedTemperature.update occurs in < 250 ms showing [ERROR] and	4.2.2 Performance Efficiency 4.2.2.1 Time behavior 4.2.1.2 Functional correctness
QA 6.2	HVAC_Controller.notify occurs in < 500 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior
RQ7	The temperature sensor can only be read at three-second intervals.	
QA 7.1	TemperatureSensor.Poll_time >= 3 secs.	4.2.1 Functional Suitability 4.2.2 Performance Efficiency 4.2.1.2 Time behavior

what is needed and the detailed designs that emerge while the requirements are being explored more deeply. Experienced IEC 61499 architects often assign familiar or well-understood terms in their ontologies that are likely to become the names of input and output parameters or events for the function blocks created later. This iterative refinement of requirements is normal as the concepts are socialised amongst the stakeholders and design team. An example of this is the addition of RQ7. One of the temperature sensors considered later can only be read reliably every three seconds. That requirement identifies a possible constraint and quality attributes were assigned to capture its characteristics.

9.6 Stage 3: Detailed design and fault risk analysis

After creating the requirement statements and reaching agreement with the stakeholders, formal diagrams are crafted to provide views of the design from differing perspectives. During this part of the design and architecture phase, the set of function blocks that will be needed is proposed and modelled. While the Pre-requirements sketches and statements help to present the Context View, the SysML Block Diagram shown in Figure 9.4 provides a view from the Functional Viewpoint. Each block on this SysML Block Diagram will be realized by one or more function blocks. SysML conveys a shared understanding of the proposed structure by employing standard, well-known conventions and nomenclature. However, SysML is not necessarily the most appropriate diagramming approach for all aspects of IEC 61499 design. Figure 9.5 shows part of the same function block application modelled in the 4diac IDE.

IEC 61499 IDEs such as 4diac and nxtCONTROL create model elements that are more dynamic since they can be refined easily. These flexible design artefacts are then complemented with other SysML or UML diagrams created with external tools such as Sparx Enterprise Architect (Sparx, 2020). SysML Class diagrams often detail the public and private methods as well as the property signatures provided by the class. In the IEC 61499 reference architecture, function blocks are analogous to classes (IEC, 2013a). The function block diagram presents the class type definition interface information using the convention of ports to represent input data

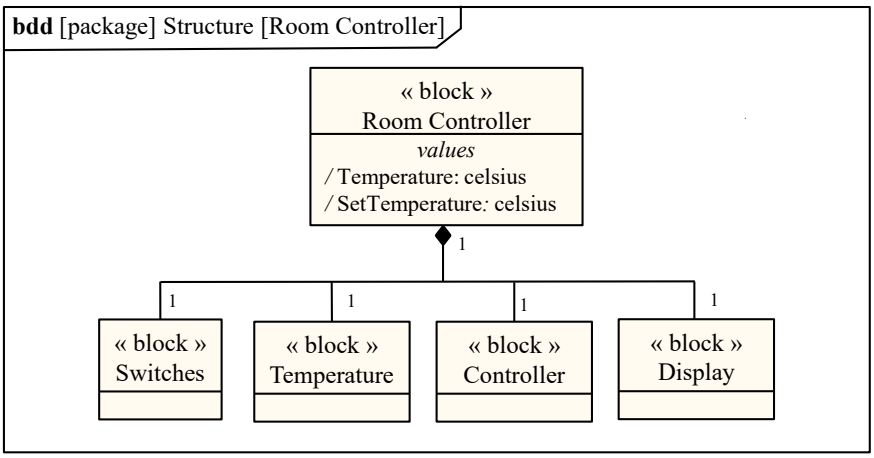

Figure 9.4: SysML Block Diagram for the room controller function block application.

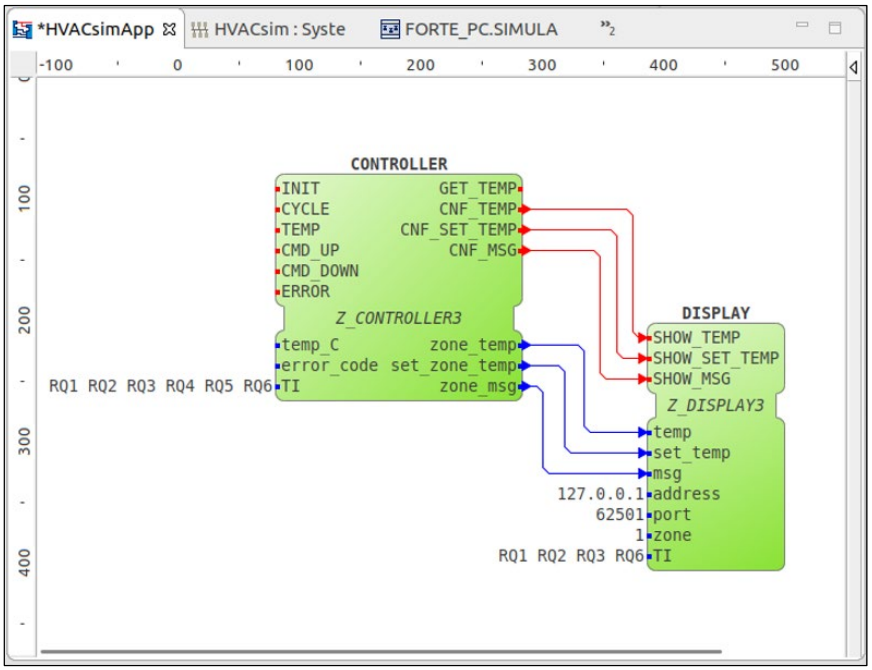

Figure 9.5: Function blocks modelled as class diagrams on the 4diac design surface.

and events on the left hand side. The corresponding output data and events are drawn as ports on the right hand side. A convention used at the Luleå Institute of Technology was adopted while working with their team. Their systems show events at the top of the function block symbol written in capital letters. Data ports are named in lower-case below them. The exception to this is the `TI` static parameter introduced in this research which captures traceability information. It is capitalised since it is an abbreviation for Traceability ID.

Architectural design tools such as Sparx Enterprise Architect augment the traditional UML and SysML static diagram artefacts with metadata that captures attributes including requirements, versioning and traceability (Sparx, 2020). IBM Rational Rhapsody extends these modelling capabilities further with the ability to generate code from design artefacts (IBM, 2016). In a similar way, IEC 61499 IDEs such as 4diac provide the capabilities to model, evaluate, and then generate executable function blocks as compilable C++ classes. In the Model-Driven Development practiced while designing the room controller function blocks, the 4diac function block diagrams have now replaced most SysML class diagrams during on-going development. However, the SysML Block Definition Diagram is still invaluable during early design stages

since it provides a blueprint to guide the selection and design of blocks that will later be created in the IDE.

In a similar way, Figure 9.6 presents a SysML Sequence Diagram that illustrates the expected order of and the flow of data and events while the room controller is operating. IEC 61499 IDEs do not currently generate system-wide interaction diagrams such as these. It introduces the `HVAC_Controller` subsystem that manages the building's heating and air circulation plant equipment. Each of the room controllers pass temperature readings and set temperature requests to this central control equipment. At this stage of the modelling, the design and communication protocol with the `HVAC_Controller` is not detailed further except to note that there is a `notify` interaction.

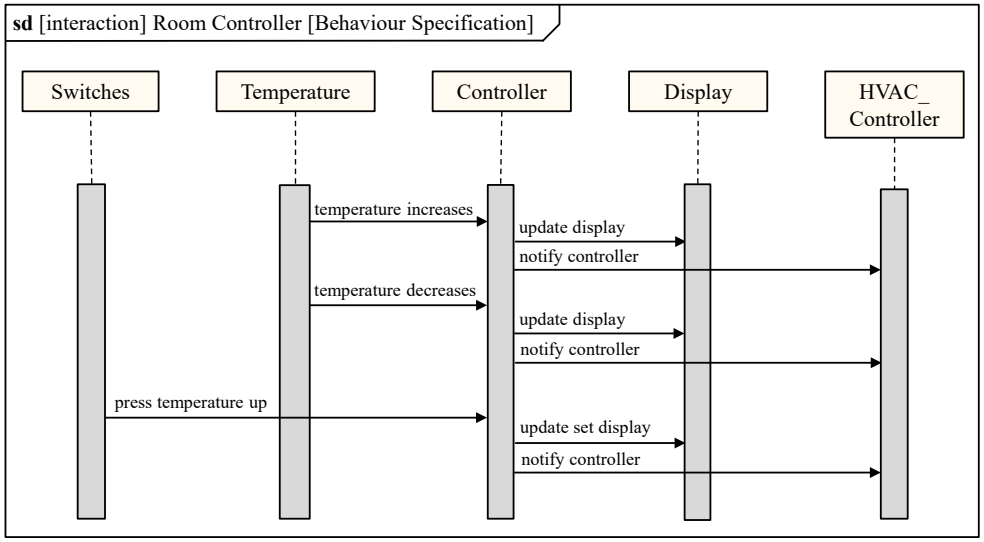

Figure 9.6: Room Controller SysML Sequence Diagrams.

IEC 61499 encourages component re-use, so suitable function blocks may already exist in the library, accompanied by their documentation and pre-built diagnostic resources. When new blocks are required, they are can be created from a wide range of standard 4diac templates. Additional data inputs, outputs, and events are specified with appropriate data types. In this way, function blocks can be created rapidly in the IDE. At this stage, it it not necessary to write the matching algorithms for each function block's activities. This allows the proposed function blocks and their connections to be modelled on the design surface and validated against the

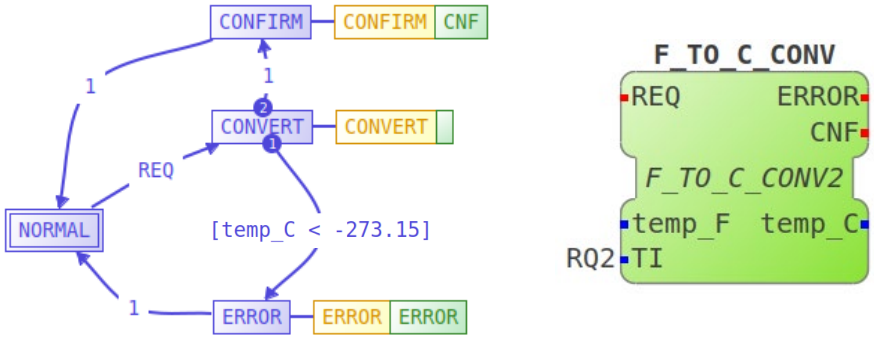

Figure 9.7: F_TO_C_CONV with its Execution Control Chart (ECC).

requirements.

Figure 9.7 shows the new `F_TO_C_CONV2` function block created for the room controller as it appears on the 4diac design surface. The Execution Control Chart details the transitions, the states, and the algorithms that are triggered by the Moore-type State Machine that drives this function block (Moore et al., 1956). Note that an ECC only documents a single function block and that only Basic Function Blocks can have an ECC (IEC, 2013a).

The function block performs a standard conversion between Fahrenheit and Celsius. An event is received by the input event port `REQ` notifying that a new reading is available on the input data port `temp_F`. If the conversion results in a temperature that is below absolute zero, then the ECC transitions to its `ERROR` state instead of outputting the new Celsius reading via its `CONFIRM` state.

9.6.1 Establishing requirements traceability

An earlier phase of this research examined ways of tagging function blocks with requirement identifier codes so they could be traced back to their parent requirements (Sinha et al., 2018). The approach used in the TORUS (Traceability of Requirements Using Splices) methodology augmented the nxtCONTROL function blocks with additional XML tags. These were embedded in the function block type definition comment lines. Requirement ID tags were also created in the algorithm code. However, this method had limitations since only parent function block types could be tagged rather than individual function block instances. The nxtCONTROL IDE

also prohibited the creation of additional, non-compliant XML tags in function block instance definitions that might have been used to store Requirement IDs at that level.

The current fault diagnostics research employs a different approach. A static input parameter named `TI` with a String data type was added to each function block when it was created. Figure 9.8 shows the function block `TEMPERATURE` on the 4diac IDE design surface. This block is an instance created from the `TEMPERATUREsim` function block type. The `TI` parameter shows that this function block helps to satisfy RQ6 and RQ7. Using the String data type provides the flexibility needed to accommodate different requirement naming conventions that are compliant with an organisations project compliance guidelines. It also allows the designer to assign more than one Traceability ID to each function block instance when necessary.

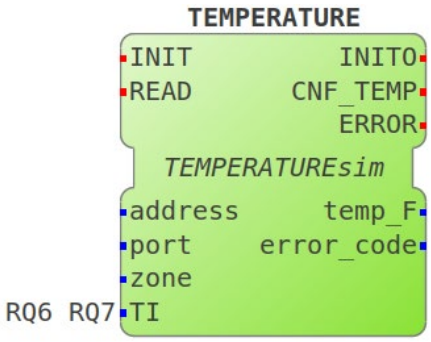

Figure 9.8: Requirement Traceability IDs on a function block instance.

Note that unlike the TORUS approach, this method of specifying a Traceability ID is not tied to the function block type. Instead it is persisted at the function block instance level for each instance created. Since is it is an IEC 61499-compliant parameter definition, this ensures on-going compatibility with 4diac, nxtCONTROL and future IEC 61499 IDEs. Further, since the parameter does not generate algorithm code in the function block, it imposes a minimal overhead on the function block application. This method also improves on the former TORUS approach by enabling the Traceability ID to always be visible on the IDE design surface. This ensures that it can be edited and kept up-to-date easily by developers. This is important since safety-critical compliance with standards such DO-178C mandate that traceability strategies must identify orphan code artefacts that cannot be traced back to at least one parent requirement (ISO, 2011a).

9.6.2 Analysing transducer and environmental characteristics

ICPS rely on their interactions with a wide range of sensors and actuators. It is important that the design encompasses an understanding of the physical and electrical characteristics of the transducers that have been chosen. A preliminary analysis demonstrated that the AM2302/DHT22 temperature sensor satisfied the cost/benefit criteria of the room controller. However, it only provides readings in degrees Fahrenheit (AOSONG, 2020).

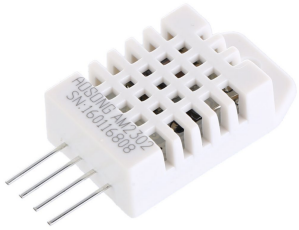

Figure 9.9: Temperature sensor AM2302/DHT22.

This constraint was addressed by the addition of the converter function block `F_TO_C_CONV` to satisfy RQ2. The manufacturer's data sheet for the sensor was also added to the set of design artefacts. The first two function blocks created in 4diac for the room controller are shown in Figure 9.10 as they appear on the 4diac IDE design surface.

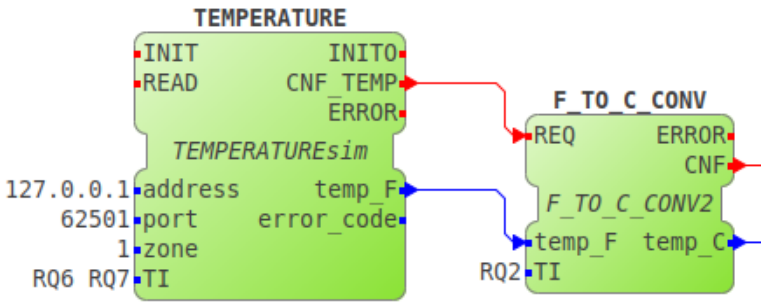

Figure 9.10: Temperature function blocks in the IDE.

During this design stage, it is important to factor in the device driver requirements. The AM2302/DHT22 employs a three-wire serial data link connection and a proprietary data exchange protocol to return its readings. Each reading must be initiated by pulsing the combined input-output data port for 18 ms in accordance with the timing constraints specified in the protocol. The manufacturer's data sheet shows that requesting and returning a sampled value requires a data exchange lasting at least 80 ms (AOSONG, 2020, page 7). Within that time, each data bit is carried in a pulse lasting 80 μs. These are challenging timing constraints, since calling a C++ library routine from within a function blocks internal `executeEvent()` method must not interfere with the event cycle that controls the rest of the function block application. Timing errors are therefore a potential fault risk, coupled with a checksum verification that indicates that the data was corrupted by interference. The data sheet also mandates that the

interval between repeated readings must be at least three seconds.

Constraints such as these must be made known to the agents by factoring them into the design of the diagnostic routines. They also influence the decision as to whether a function block should be designed to *push* or *pull* IEC 61499 events. The decision was made to have the `TEMPERATUREsim` function block pull, responding to events that arrive on its `READ` event input port. This ensures that `CONTROLLER` is able to orchestrate all the activities of the room controller. Regardless of how fast it is queried, `TEMPERATUREsim` manages the three-second timing constraint internally by remembering the time of the last reading and the last temperature value read. If another reading is requested less than three seconds after the previous one, the block returns the previous reading rather than querying the AM2302/DHT22 sensor again. This is a reasonable constraint: the ASHRAE Environmental Standard 55 for HVAC systems proposes that a suitable rate of change of the temperature for ensuring the comfort of room occupants should be no more than 0.3°C/min (ASHRAE, 2004). Hence sampling the temperature at three-second intervals is unlikely to affect the accuracy of the readings given the likely rate-of-change.

Table 9.3: Risks and Diagnostic Needs Identified

Risk ID	Function Blocks involved	Description and notes
1	TEMPERATURE	Checksum error indicates invalid data during a reading.
2	TEMPERATURE	Sensor failure stops the device returning a temperature reading.
3	TEMPERATURE	Reading is out-of-range indicating a sensor failure.
4	TEMPERATURE	Failure to read sensor value due to a device driver failure.
5	TEMPERATURE	Inability to meet timing constraints during the sensor read.
6	CONTROLLER	Logic error in the ECC state transitions. Design-time, field upgrade, or maintenance issue.
7	CONTROLLER	Failure to communicate with the main HVAC_Controller system
8	SET_TEMP_UP, SET_TEMP_DOWN	Failure to read the switch value due to a device driver failure or switch mechanical failure.

This process of considering the requirements in the light of the needs of the function block software as well as the capabilities and nature of the transducers leads to the creation of a risk-list. Table 9.3 illustrates the information captured at this stage of the design cycle. These risk descriptions help to influence the choice of which faults should be catered for by creating

appropriate diagnostic resources. Some diagnostic tests will address one function block or need while others will describe potential faults that affect multiple, connected blocks.

9.7 Stage 4: Component development with diagnostics

Stage 4 provides a separate phase in which to perform more detailed prototyping and development of the individual function block components. Isa et al. (2015) and Angeleva (2016) comment that the real value of a prototype is the way in which it stimulates engineers to think about the implications of their design choices and to consider alternatives. The aim of this V-Model phase is to code and verify the complete set of function blocks which will be assembled into the function block application during the Stage 5 development phase that follows. IEC 61499 application development is conceptually similar to creating a building with LEGO, except that the engineers often have to design their own bricks, or in this case, function blocks. The list of candidate function blocks identified in Stage 3 was divided into logical groups to schedule their development. The characteristics of the system will indicate which blocks are the most likely candidates to create first.

Since Unit Testing is not practical for function blocks, the agents support the developer by exercising the blocks while they are being created, ensuring that all inputs and outputs work correctly. A complete function block application operates like a production line, where data inputs feed logically into one section, the data is processed, and decisions are made before a data output feeds the information into the next stage of the pipeline.

9.7.1 The role of simulation in design and development

Neither the 4diac or nxtCONTROL IDEs provide ways to connect and use the output of a sensor or control an actuator at design time. That would require additional hardware and software drivers that are often only available when the function blocks are running on hardware that provides I/O ports for connecting electronic devices. To address this need, Model-Driven Development with Diagnostics uses simulation tools that work alongside the engine to provide

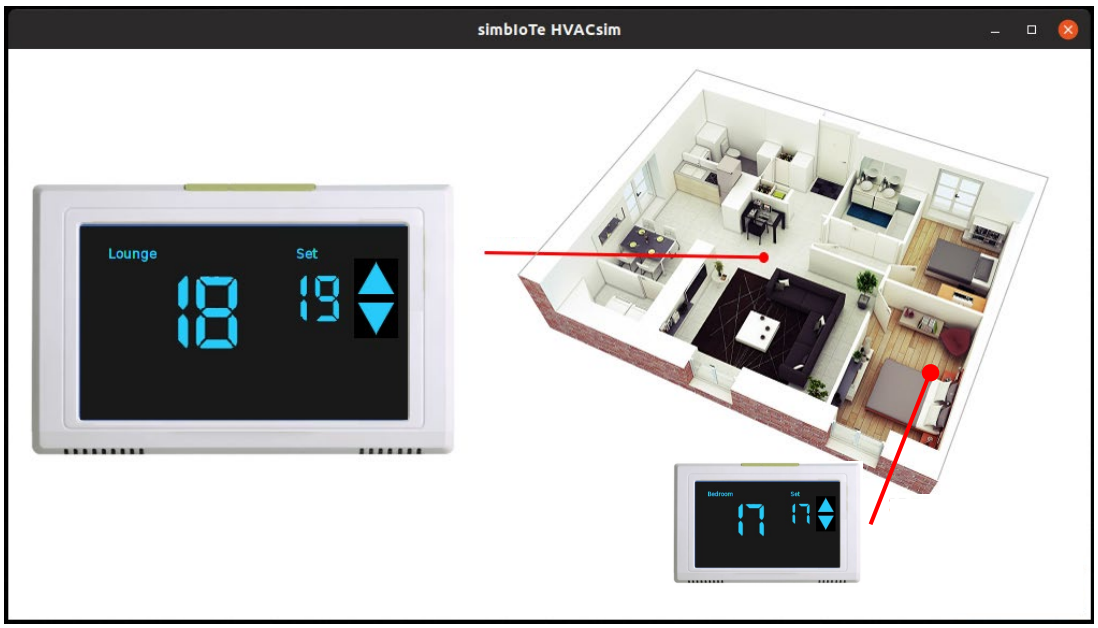

Figure 9.11: The HVACsim room simulation modeled in *simbIoTe*

the necessary telemetry.

The simbIoTe (**simu**lator for **b**uilding **IoT e**nvironments) application was developed in-parallel with the fault diagnostic engine to provide a flexible, customisable environment. Simulation provides a way to quickly start the development of function block applications and build diagnostics before the production hardware is ready. While it is being developed, a function block can be stressed and diagnosed for faults early in its design cycle. Simulation techniques such as these are widely used in both the aerospace and automotive sectors, where the lead time for complex hardware is often measured in years (McGovern, Cohen, Truong & Fairley, 2007). The architecture of simbIoTe is discussed separately in more detail in Appendix B.

During the Stage 5 Development phase, simbIoTe provided a realistic simulation named HVACsim to model the environment of a building. Figure 9.11 shows simbIoTe emulating temperatures and HMI panel displays for the rooms in a building. The ambient outside temperature as well as the temperature drift can be configured, which are then managed by algorithms that simulate realistic temperature changes. When a function block requests a reading, simbIoTe returns the temperature for a specific room via an API. Similarly, when the function block application has data to display, the emulated room panel updates the UI part of the simulation.

Data is returned and sent in packets that have a similar command and data structure to that developed for the diagnostic points.

As more and more blocks are completed and parts of the hardware become available, fewer aspects of the environment need to be simulated. The hardware and the function block application begin to interact with the real physical environment around them while those parts of the application that are still under development continue to rely on data from simbIoTe. This iterative refinement of the application using fault diagnostic agents, supported by differing levels of simulation or "co-simulation" (Cremona et al., 2019), is a key aspect of the methodology proposed in this research.

9.7.2 Designing individual function blocks

Designing a new function block begins by considering the subset of the requirements that can be fulfilled by one or more function blocks as a logical sub-unit. An example of a logical sub-unit for the room controller is the set of function blocks shown on the bottom left of Figure 9.12. These blocks capture the current temperature, convert it to the correct temperature scale, and

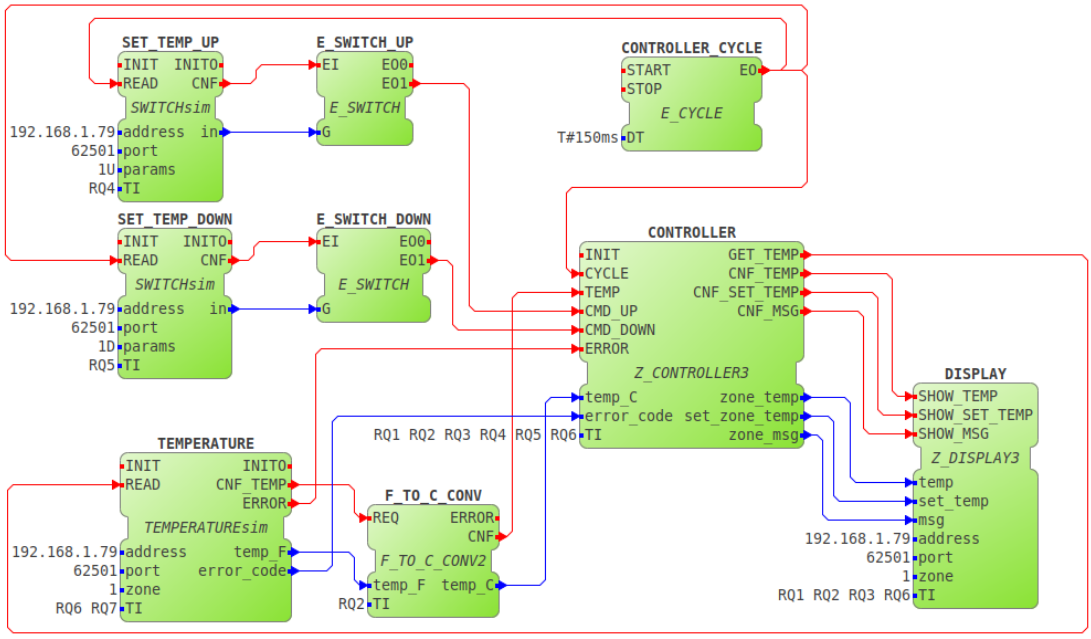

Figure 9.12: Function Blocks proposed for the room controller.

then deliver it to the main controller. As discussed previously, a design decision had to be made to determine if the temperature subsystem pushes new temperatures to the controller when they change, or if the controller polls the sub-unit at regular intervals to pull data. This is no single right answer for that; such decisions are often driven by the experience of the designer or developer. The final choice will also be influenced by an understanding of both the requirements and the characteristics of the sensor that is to be used to read the temperature.

Individual function blocks are not usually designed in isolation. Inputs provide data that is evaluated and transformed inside the function block so it can fulfill its purpose within the application. Depending on the output from its internal algorithms, the state of the block may change and data coupled with events may be output. While designing applications in this way, there is a subtle, but natural, inclination to drive the flow of information in the diagrams from left to right. This is consistent with the IEC 61499 standard conventions. The temperature capture and conversion sub-section felt natural to place on the left. Similarly, it was seemed logical to position the `CONTROLLER` that co-ordinates the information from all sensors and actuators near the center of the model. While the display and on-screen switches may later be realized as a single integrated unit, their functions were separated in the model with the switches expressed as inputs on the left and the display as outputs on the right. Creating such uniformity of expression is a key goal of model creation (Cicchetti, Ciccozzi & Pierantonio, 2019; Shah, Kerzhner, Schaefer & Paredis, 2010). The event connections are shown as red lines and the data connections are shown are blue lines in Figure 9.12. Note how they are routed tidily so the data flow can be understood more clearly. While designing the blocks in this stage, events and data ports were repositioned up and down the sides to create a more logical routing between blocks. This led to an information flow that was much easier to trace later. Well-understood diagramming conventions such as these not only speed up the design process, they remove ambiguity, foster better communication between stakeholders, and lead to specifications that can be refined iteratively (Vyatkin, 2011). Later experiments demonstrated how this convention also makes it easier to identify potential fault pathways.

9.7.3 Establishing traceability before the next stage commences

Once all the components have been proposed and their interfaces have been defined, the traceability matrix shown in Figure 9.4 can be completed. This verifies the traceability of all the requirements before the next stage commences.

Traceability should be verified carefully and regularly through all subsequent stages, updating the `TI` parameter on each function block within 4diac and on this table whenever they

Table 9.4: Functional and quality requirements with their traceability links.

Requirement and Quality IDs	**Description**	**ISO 25010 Quality Attributes**	**Traces to these Function Blocks**
RQ1	when Temperature.increases or Temperature.decreases then		CONTROLLER DISPLAY
QA 1.1	HVAC_Controller.notify occurs in < 500 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	
QA 1.2	DisplayedTemperature.update occurs in < 250 ms	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	
RQ2	Temperatures must be displayed in Celsius.		CONTROLLER
QA 2.1	Temperature.unit = Celsius	4.2.1 Functional Suitability 4.2.1.2 Functional correctness	DISPLAY F_TO_C_CONV
RQ3	Set temperatures must be displayed in Celsius.		DISPLAY
QA 3.1	SetTemperature.unit = Celsius	4.2.1 Functional Suitability 4.2.1.2 Functional correctness	CONTROLLER
RQ4	when SetTemperatureUp.pressed then		SET_TEMP_UP
QA 4.1	HVAC_Controller.notify occurs in < 500 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	CONTROLLER
QA 4.2	SetTemperature.update occurs in < 250 ms	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	
RQ5	when SetTemperatureDown.pressed then		SET_TEMP_DOWN
QA 5.1	HVAC_Controller.notify occurs in < 500 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	CONTROLLER
QA 5.2	SetTemperature.update occurs in < 250 ms	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	
RQ6	when Temperature > 176 Fahrenheit OR Temperature < -40 Fahrenheit then		CONTROLLER TEMPERATURE
QA 6.1	DisplayedTemperature.update occurs in < 250 ms showing [ERROR]	4.2.2 Performance Efficiency 4.2.2.1 Time behavior 4.2.1.2 Functional correctness	
QA 6.2	HVAC_Controller.notify occurs in < 500 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behavior	
RQ7	The temperature sensor can only be read at three-second intervals.		TEMPERATURE
QA 7.1	TemperatureSensor.Poll_time >= 3 secs	4.2.1 Functional Suitability 4.2.2 Performance Efficiency 4.2.1.2 Time behavior	

are refined or replaced (Mäder & Egyed, 2015; Mader, Gotel & Philippow, 2008). Once all the blocks had been proposed, their functionality was checked against the requirements to confirm the design decisions that had been made. Very often, comments and questions were entered into the algorithm body as text for the developers to consider in Stage 4. The correctness, completeness and consistency criteria are considered continually during this stage, guided by the traceability linkages. All function blocks should carry `TI` information. Those that do not are often standard 4diac library blocks such as `E_CYCLE` that have not been modified during this research.

Creating a function block in the IDE provides a class instance that contains the input and output events for each block. The data type for each of the data ports must be specified when they are created. When blocks are connected, their input and output data types at each end of the link must match: the IDE stops designers connecting incompatible ports together. This constraint helps to finalise the nature of the data flow, especially in cases where numeric data is being exchanged. The precision required for each value must be considered to avoid incorrect data being propagated through the system that might generate faults.

Earlier, it was explained that at this stage of the design, the algorithms do not need to be coded. During the creation of the Execution Control Chart transitions, it was found that time spent coding them at this stage of the design was counter-productive. The type and purpose of some algorithms was extremely fluid. The delegation of error management and processing often moved between different blocks as the data flow was elucidated. However, modeling the application in this stage clarified most issues.

9.8 Stage 5: Development with diagnostics

The Stage 5 Development phase was divided into two major parts. The first completed the coding and diagnostic testing of the individual components and then wired them together to create the first iteration of the application. During this time, the diagnostic packages provide iterative fault-finding for each aspect of the design. Simulation provided a way to follow data

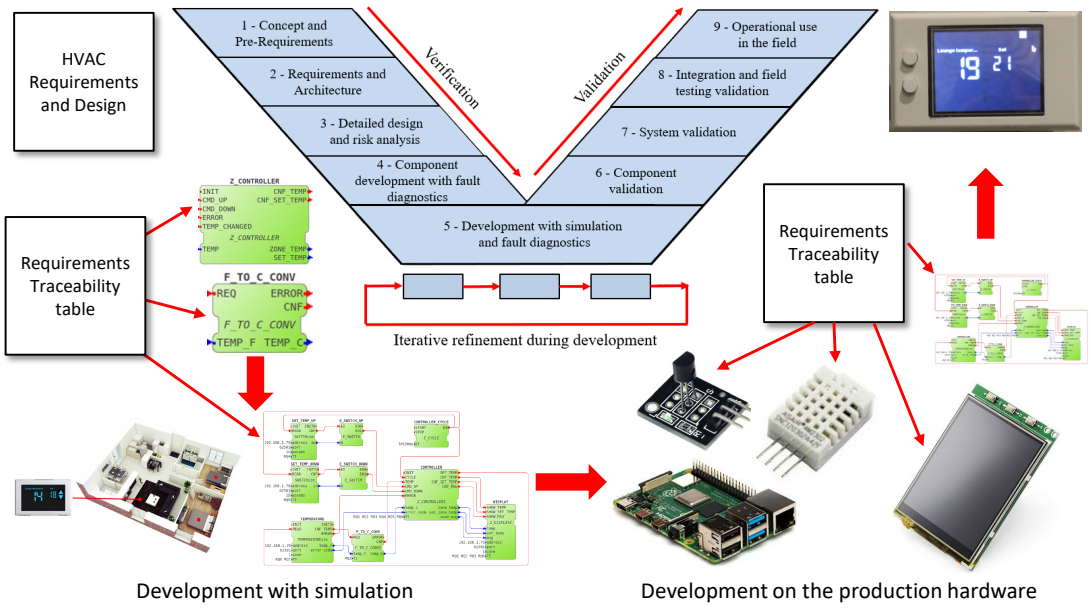

Figure 9.13: The V-Model Stage 5 Development phases in perspective.

paths and verify the correct operation of each block. The second part of this phase commenced when the hardware platform was available and integration with the sensors and the actuators could begin. As noted earlier, the availability of the production hardware often impacts how quickly this part of the development can begin. Delays typically occur in domains where the hardware is complex, novel, or the lead-time of the components is long. Figure 9.13 shows the two development stages in perspective with the other stages of the V-Model.

The function blocks that implement the HVAC room controller are representative of the primary types of blocks available in IEC 61499. `F_TO_C_CONV2` is a Basic Function Block (BFB) that uses its own ECC and algorithms to convert a temperature to a different scale. In contrast, `TEMPERATUREsim` was built upon the IEC 61499 template for a Service Interface Function Block (SIFB). These are specialised function block types that are ideal for exchanging data between the function block application and physical transducers. A number of different conventions guide the design of SIFBs so they operate reliably within the IEC 61499 event-driven architecture. One convention is that SIFB's do not implement standard IEC 61499 ECCs. This gives them the flexibility to specify their own state management and interaction cycles with real-world sensors and actuators. Figure 9.14 illustrates the internal structure of

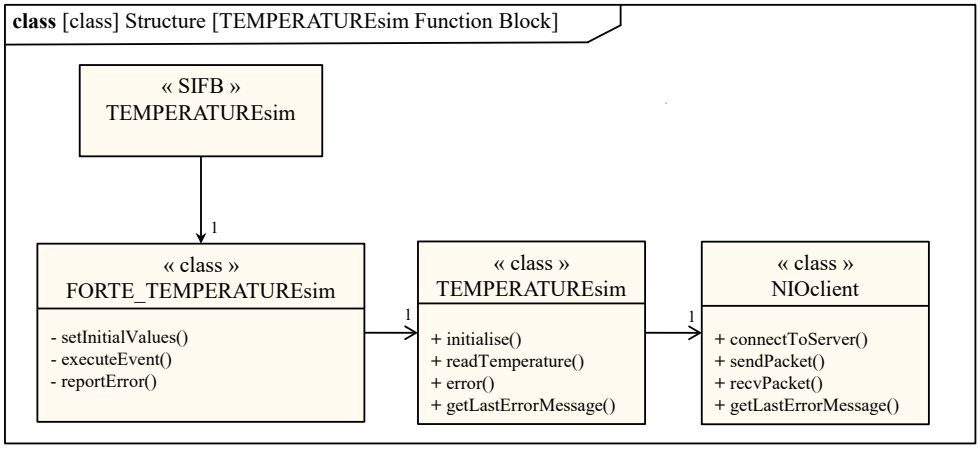

Figure 9.14: TEMPERATUREsim Service Interface Function Block internal structure.

the `TEMPERATUREsim` SIFB and the primary methods it uses to interface with simbIoTe's emulated temperature sensor and the function block application.

When the function block application is operating, `TEMPERATUREsim` receives an event via its `READ` input event. Since there is no AM2302/DHT22 temperature sensor hardware available at this stage of the development, the C++ sensor driver class is replaced by a simulator library class called TEMPERATUREsim. This class is capable of establishing a TCP client connection to the internal server that is the gateway into and out of simbIoTe. The `NIOclient` C++ class is the same one used by the diagnostic points to communicate with the agents. This generic communications library was created during this research and built into the 4diac FORTE runtime so that it can be shared by other components. The packet structure used to exchange data was shown previously in Figure 6.2. The TCP connection and the room parameters are provided on the `address`, `port`, and `zone` function block parameter inputs during design. When a connected block requests `TEMPERATURE` to provide a reading, a packet with the command `GZ` is sent to simbIoTe. The packet identifies which temperature zone it wants to read. The temperature is then returned synchronously via a postback reply. The data packet that returns from the simulator is a formatted string that emulates the AM2302/DHT22 serial interface sensor data packet (AOSONG, 2020). This ensures that the simulator mimics the intended hardware as closely as possible so that the `TEMPERATUREsim` class can be replaced directly by the `DHT22` class when it is available to read from a real electronic sensor. This ensures

that the step from simulation to real hardware requires minimal changes to the TEMPERATURE function block code since the interfaces are as similar as possible.

The SWITCHsim function block operates in a similar way. Within simbIoTe, switches and buttons are rendered on the emulated HMI screen and internal counters record each time they are pressed. The function block establishes a connection to the simulation in the same way that TEMPERATUREsim does. In the production application, the 4diac IX function block is able to read the status of an I/O port such as the General Purpose Input Output (GPIO) pins on a Raspberry Pi or WAGO computer. By emulating the operation of the standard 4diac interface function blocks, it becomes easier to insert the production function block into the application later to replace the simulated component. The CONTROLLER is designed to expect an event only when a switch has been pressed. This requires the addition of a buffer function block E_SWITCH since each time the switch status is read, SWITCHsim emits its CNF event after setting in true or false depending on the logic level of the pin. This is an example of a design decision that was made to cause the actuator to push an event itself rather than wait to be being polled or pushed by another block.

9.8.1 Gimbaling the temperature subsystem with simbIoTe

Sensors and actuators are components that operate on the boundaries of the cyber and physical parts of an ICPS (Dillon, Chang, Singh & Hussain, 2012). The events they receive from within the function block application request them to deliver a reading. They are also able to trigger events themselves during the IEC 61499 operating cycles. However, the sensing that they perform on the physical side is usually outside the control of the IEC 61499 event chain. This introduces difficulties when diagnostic testing is to be performed on function blocks such as TEMPERATUREsim, SET_TEMP_UP, or SET_TEMP_DOWN. There is no event or data input that a DP could use to inject representative test values into.

A novel way was found to address this with simbIoTe. Figure 9.15 details the gimbaling script developed for TEMPERATUREsim that takes control of the simbIoTe Emulation subsystem to allow the agents to supply test values directly to the HVACsim simulation.

```
Belief gimbal() {
    double temperatureF = 0;
    double tempertureC = 0;
    double expectedTemperatureC = 0;
    int errorCount = 0;

    // Create a connection to the simulator.
    simbIoTe sim = new simbIoTe();
    if (!sim.connect("192.168.1.79", 62501)) {
        belief.Veracity(VeracityTypes.FALSE);
        belief.add("Cannot connect to the simulator. " +
                   sim.lastErrorCode);
        return belief;
    }
    // Stop the simulator generating environmental changes
    // and close the gate into the temperature subsystem.
    sim.send("HALT");
    TEMPERATURE.READ.gateClose();

    // Gimbal the temperature subsystem.
    temperatureF = (double) -10;
    for (cnt = 1; cnt < 50; cnt++) {
        expectedTemperatureC = (temperatureF - (double) 32.0) * (double) 0.555;
        // Set the test temperature in the HVACsim Environment.
        sim.send("SZ1", temperatureF);
        // Trigger the sensor event to cause it to read the temperature.
        TEMPERATURE.READ.trigger();
        if (F_TO_C_CONV.CNF.read(TEMPERATURE.READ.timestamp())) {
            temperatureC = F_TO_C_CONV.CNF.value();
            if (temperatureC != expectedTemperatureC) {
                errorCount++;
                belief.add("Error : temp_C should be " +
                           expectedTemperatureC + " instead of " +
                           temperatureC);
                }
            }
            if (TEMPERATURE.ERROR.read(TEMPERATURE.READ.timestamp())) {
                errorCount++;
                belief.add("Error returned from read " + TEMPERATURE.ERROR.value());
            }
        }
        temperatureF = temperatureF + (double) 9.9;
    }
    TEMPERATURE.READ.gateOpen();
    sim.send("RESTART");

    if (errorCount == 0) {
        belief.Veracity(VeracityTypes.TRUE);
        belief.add("Nominal.");
    } else {
        belief.Veracity(VeracityTypes.FALSE);
    }
    return belief;
}
```

Figure 9.15: Gimbaling TEMPERATUREsim using simbIoTe.

The script instantiates a `sim` object from within the engine which has methods that allow a packet command and a data value to be sent by an agent directly to simbIoTe.

The commands HALT and RESTART were added to the simbIoTe Environment interface to allow the agent to request it to stop or restart managing its own temperature changes. The script then generates a range of test Fahrenheit temperatures and sends one using the SZ1 command during each test. This sets the temperature within that zone that the Environment retains but does not modify while it is in its HALT mode. The TEMPERATURE.READ.trigger() method then causes the function block to retrieve a new reading from the simulator.

Later tests introduced the command SZ1ERR which causes simbIoTe to return a data packet with a faulty checksum. This is used to test the error capabilities of the sensor function block. Note that the script references the `F_TO_C_CONV` diagnostic point that was not defined in TEMPERATUREsim.dpg. The set of diagnostic packages is used to create a set of diagnostic points whose scope is application-wide. Hence, any script can establish a reference to any diagnostic point. The pre-processor code generated during the script compile has access to the global definition table for the DP objects. This capability is discussed further in later stages where it is used to create system-wide monitoring diagnostic scripts. Note that for distributed IEC 61499 applications, the scope of the diagnostic points is restricted to the sub-application rather than the whole application. A similar script was developed to exercise the switches.

Two interesting errors were detected during the diagnostic trials. In one case, the agents detected an issue where the digits in the constant in the Fahrenheit to Celsius formula had been accidentally reversed inside `F_TO_C_CONV`. While exercising the switch subsystem, the events connecting `E_SWITCH_UP` and `E_SWITCH_DOWN` were inadvertently crossed over in the 4diac IDE while connecting them to the `CONTROLLER` event inputs. The agents reported that no matching events were generated for either of the switches.

9.8.2 Implementation on the real hardware

The simulation phase of the development allowed all the requirements to be verified, including the expected timings for RQ4 and RQ5. The error reporting for RQ6 was also verified during

the gimbaling. The simulation phase also verified all ECC transitions and the correct processing of sensor readings.

However, implementing the AM2303/DHT22 sensor on the Raspberry Pi 4 platform was problematic. A C++ driver had been developed previously to evaluate the AM2303/DHT22 as it ran natively on the Pi outside of the FORTE function block runtime. This used the WiringPi C++ driver library to control the General Purpose Input-Output (GPIO) pins on the Pi. While this was a useful exercise, WiringPi has now been deprecated and the preferred I/O method for 4diac is the `IX` block for reading GPIO input pins. The complementary `QX` function block is recommended for sending data out via a GPIO port. These 4diac blocks uses the Linux *sysfs* interface. Since Linux version 4.8, the GPIO sysfs has also been deprecated (Kernel, 2018). The driver protocol for reading the sensor uses sysfs to read a single serial I/O interface pin. After polling the sensor to request a reading, the port must transition from being an input to an output. The timing constraint for this change proved to be too rapid for the `IX` and `QX` blocks to keep up with. Wiring these blocks to the same GPIO pin is not allowed. Attempts were made to generate a hybrid block based on their design, but further experiments showed that the 4diac sysfs interface also had issues with switching that fast.

Subsequent research uncovered an alternative sensor that worked successfully. Newer sensors use drivers that operate as extensions to the Linux kernel. These are loaded dynamically using the Linux *modprobe* command. The DS18B20 temperature sensor shown in Figure 9.16 interfaces with the kernel virtual file system to communicate with applications. The sensor uses a single wire serial interface with a protocol similar to the AM2303/DHT22. However, the packet protocol does not need to be developed in 4diac. The code uses much simpler Linux file I/O commands. Opening the devices unique symbolic link system file in read-only mode causes the kernel driver to refresh the sensor reading before it allows the file to be opened. A replacement function block for `TEMPERATUREsim` called `DS18B20` was developed that appears on the bottom left of the completed HVAC_Pi application

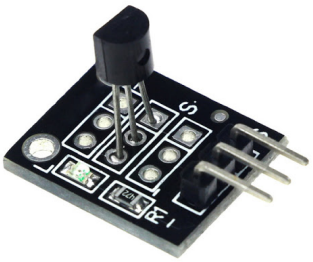

Figure 9.16: Temperature sensor DS18B20.

shown in Figure 9.17. This naming convention was adopted to enable the new DS18B20 function block and its diagnostic resources to be found easily in the library later. The sensor was shown to satisfy all the timing constraints in the requirements. The instance name of the function block remains `TEMPERATURE`, so no changes to the Traceability table were required.

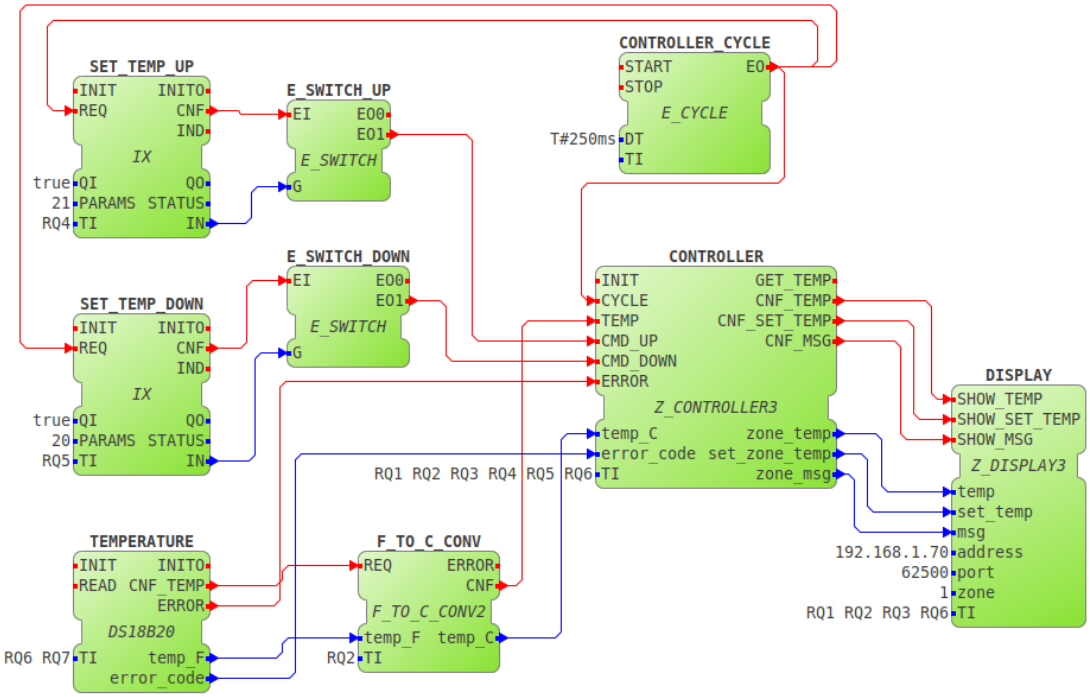

Figure 9.17: The HVAC_Pi function block application with sensors and display drivers.

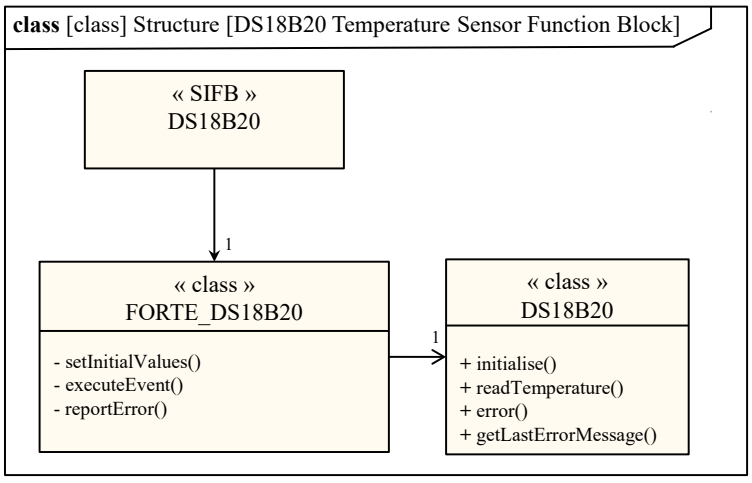

Figure 9.18: The DS18B20 Temperature Sensor Service Interface Function Block.

Issues such as these are commonly encountered while developing sensor and actuator interfaces for ICPSs. They point to the need for more robust early-stage prototyping to uncover issues that could later delay the development phases unnecessarily (Exner et al., 2015; Angeslevä et al., 2016).

Code from simbIoTe that had been used to create the simulated HMI display panel was also adapted to create a lightweight HMI panel that could drive a TFT display panel. The completed devices deployed on a Raspberry Pi 4 are shown in Figure 9.19.

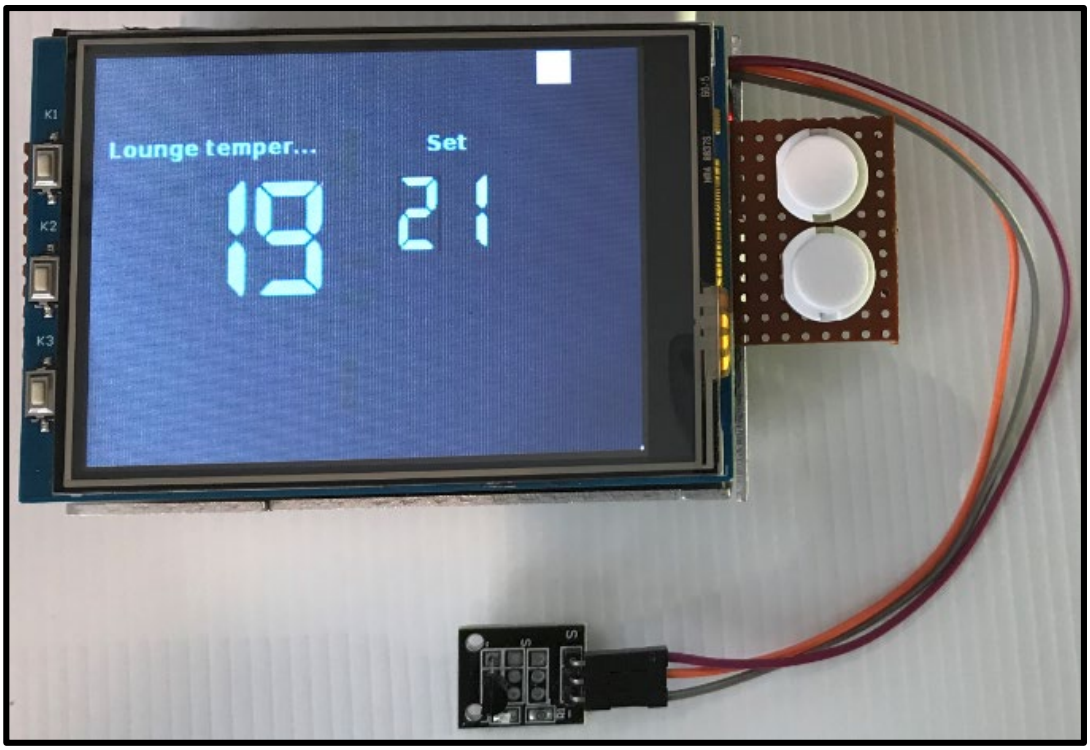

Figure 9.19: The HVAC room controller hardware with its display panel and sensor.

9.9 Stage 6: Component validation

The Stage 6 Component validation phase complements the earlier Stage 4 Component development with diagnostics phase. The aim of the stage is to validate the production components individually and within their subsystems now the first production version of the application is believed to be complete. This stage is usually a laboratory or test-bed evaluation, performed by

a different team to the developers who created the application in Stage 5.

The same gimbaling scripts that were used during the development stages are now run across all the function blocks running on the proper hardware. The gimbaling scripts isolate sections of the application using the gate() command so that extreme test values do not cause other parts of the application to react unnecessarily. No simulation is employed at this time. It is important that the same scripts are used, since they should have been designed to uncover issues that occur in both the simulated and real environments. The purpose is to validate the internal consistency, completeness, and correctness of the whole application against original requirements. This is similar to the approach to requirements validation of Zowghi and Gervasi (2002) discussed earlier.

9.10 Stage 7: System validation

Hamilton and Zeldin (1980) discuss the relationship between design and verification in the context of early embedded systems design for Project Apollo. They emphasise that there must be a fundamental assumption made during the design process for it to be effective. The design tasks must include techniques that facilitate the verification of the target design and the validation of the completed product that are applicable to the Stage 7 System Validation performed here. These tasks include:

The identification of redundancies: Redundant components are those that duplicate functionality needlessly. This is distinct from the those components that provide fault-redundancy. Where a requirement traces to multiple components, the system validation should ensure that the components are used to support different aspects of the requirements rather than duplicate capabilities.

Logical incompleteness: This is evidenced by the presence of unfulfilled requirements. By definition, a design or implementation that has not satisfied all its requirements is incomplete. Again, this is determined by validating the operation against the requirements and their traceability links specified in Table 9.4.

Inconsistencies in the system definition: This includes behaviours that are inconsistent with the activities specified in SysML diagrams such as Figure 9.6 which detailed the sequence of the data exchange between the HVAC components. Other sources of inconsistency include expected execution times and descriptions of behaviours in response to error conditions.

This was addressed in Stage 7 by using the same diagnostic resources that were used to perform the validation in Stage 6. However, the emphasis is on longer-stage burn-in verification where the agents operate primarily in their Monitor mode. A diagnostic package named system.dpg configures the same diagnostic points that were used in earlier verification trials. This package contains monitor scripts that capture output events such as those available on `TEMPERATURE` and match them to the output events that should occur on `CONTROLLER`. If the application fails to satisfy a timing constraint or a fault is detected, the agent will step into its diagnostic mode where it uses multiple gimbaling scripts to walk logical data paths implied by the individual function block .dpg files and in the system.dpg package.

Figure 9.20: The HVAC room controller during a Stage 8 field trial.

A limited burn-in trial was performed during this stage. During the course of one day, the room controller monitored a room while connected to the engine. The agents noted changes to the room temperature requested by a user at random intervals. A wire was removed from the

DS18B20 module to simulate a failure by the sensor. This generated the correct belief records when the agents detected the issue and responded.

9.11 Stage 8: Integration and field testing validation

Once a limited-time burn-in trial has completed successfully, longer-stage verification can be performed by situating the HVAC room controller in a production environment. The same monitoring can be performed by the agents to complete a longer duration trial. No matter how thorough the verification and validation is, field trials often reveal behaviors by ICPSs that were not expected. This echoes Brooks comment that "*a sword is only tested by cutting...*": the device has to be validated on the battlefield (Brooks Jr, 1996, page 2).

9.12 Stage 9: Operational use in the field

The final stage of the V-Model brings the project to completion, mirroring Stage 1 where the Concepts and Pre-Requirements were first drawn up for the project. The field operation of the device relies on the same diagnostic support which ensured that all components could complete their Stage 8 Integration and field testing validation.

9.13 Conclusions and planning the next phase

This chapter profiled the development of a prototypical function block application, demonstrating how the diagnosable-by-design methodology works in practice. The iterative V-Model design and development phases with their parallel verification steps included the early-stage creation of diagnostics by engineers. Later, on the parallel arm of the V-Model, the deliverables were validated against the original requirements and design specifications. These validation phases were augmented with diagnostic monitoring and fault-finding during the system integration steps. Graessler et al. explain that the core of the V-Model is the decomposition of the ICPS elements designed in the stages of the left-hand arm (Graessler & Hentze, 2020). These are followed by

the gradual integration and validation of each subsystem through to the final acceptance of the product. The enhanced V-Model employed in this research showed how fault diagnosis added a further level of verification and validation to these processes. Table 9.5 details the benchmarks that this chapter demonstrated for the methodology, the engine, and the diagnostics during the creation of HVAC Room Controller.

Table 9.5: Benchmarking the engine and the diagnostics.

Benchmark	**Requirement**	**Size and complexity**	**Description**
B1	Diagnosable-by design methodology.	Low not complex.	Demonstrate the integration of fault diagnosis to all relevant stages of the V-Model. Illustrates the ease of use of requirement traceability directly within the 4diac IDE. Demonstrate the methods of risk analysis for identifying potential faults that must be addressed during diagnostic design.
B2	Simulation using simbIoTe.	Large, complex.	Supporting the Stage 4 component development stage to exercise the individual components before the production sensors and actuators are available. Creation of the function block library resources to interact with simbIoTe.
B3	Direct agent interactions with simbIoTe.	Moderate, not complex.	Demonstrates ways the agents can provide fault test data to simbIoTe to exercise simulated sensors and actuators for edge interactions that cannot be gimbaled by the agents alone.
B4	Monitor and gimbal script functionality.	Large, complex.	Demonstrates the ease of scripting the fault diagnostic commands and skills that the agents can employ while monitoring or diagnosing a fault.
B5	Event trigger interactions.	Low, complex.	Illustrates how the diagnostic points can use timestamps to synchronise trigger and event capture reliably to capture telemetry.

Diagnostic resources developed in this way continue to provide value throughout the life-cycle of the ICPS since they are crafted while the knowledge of the system is still fresh in the minds of the architects and developers. This tailors them to better meet the specific fault-finding needs of each system. Iterative development and refinement, where each issue uncovered by the fault diagnostic agents is addressed early, builds a level of resilience into applications. This approach reinforces the value of diagnostic-driven design for function block applications. The alternative approach of creating after-the-fact diagnostics when the ICPS is complete is both time-consuming and error-prone. It requires teams to revisit the specifications again from the beginning, often long after they were originally crafted.

The next phase applies these same techniques to demonstrate how multiple agents can be employed to share fault-finding tasks while developing protection components for a Smart Grid using IEC 61499. In the process, it examines how agents co-operate and how they are able to capture precise timing measurements while validating the components of the application.

Chapter 10

Smart Grids and Agents

"I want my research to be in the places where there are gaps and my solutions to be innovative. I want my findings to be sound and verifiable by my peers: It's as simple and as difficult as that."

paraphrase of a quotation by children's author Kevin Henks (2020).

The previous chapter demonstrated how Model-Driven Development with Diagnostics works in practice by profiling the design and development of a function block application in depth. This chapter now examines a set of Smart Electricity Grid fault protection devices, exploring how well the engine performs in a regulated, safety-critical environment. The IEC 61499 Smart Grid protection devices created during the present research are based on designs that Dr Gulnara Zhabelova and the author worked on at the Luleå Institute of Technology during 2017. That research focused primarily on requirements traceability for the smart grid devices Gulnara and her colleagues had created, but also examined architectural features of the IEC 61499 devices. This work was published in a journal article for the ACM Transactions on Cyber-Physical Systems entitled *TORUS: Scalable Requirements Traceability for Large-Scale Cyber-Physical Systems.* (Sinha et al., 2018).

Techniques for evaluating telemetry, the capture of accurate timing, reliability, and fault-tolerance are crucial aspects of smart grid protection. The same V-Model processes and stages that were detailed in Chapter 9 were used to develop the devices discussed here. However, this chapter focuses primarily on the diagnostics that were developed. It evaluates how well they

helped to validate the design by identifying and responding to faults. It demonstrates how a team of agents can work collaboratively, each one taking responsibility to monitor or investigate a separate protection device. When a fault is detected, it details the way an agent intervenes to diagnose and report the problem by exchanging Dynamic Diagnostic Beliefs. During this research, the belief structures developed in earlier phases have proven to be an ideal way to capture fault information in structures that can be both exchanged between agents and analysed empirically.

Hamilton and Zeldin caution that "*...all the intricacies of a complex system are by nature beyond the grasp of any human being.*" (Hamilton & Zeldin, 1980, page 31). The Model-Driven Development methodology, this time demonstrating the use of multiple agents to support the engineers, is one approach to address the aspects of complexity that concerned them.

10.1 Smart Grids, substations, and protection devices

In electrical equipment, it is not the voltage, but the current which causes the most damage: components fail when their electrical ratings are exceeded. Excessive currents cause overheating, which will destroy both passive and active components (Awad & Bollen, 2003).

In the past, electricity generation fault protection equipment was purely electromechanical. The first Overcurrent Protection units came into service in 1901 (Bo et al., 2016). They were designed to be highly risk-adverse, tripping their circuit breakers whenever they detected faults on the high-voltage distribution lines they were monitoring. They operated in a hierarchical structure under centralised control, often isolated from other equipment on the grid. Since individual devices could not communicate with each other, they were unable to determine the extent of downstream or upstream faults and disturbances. While these units proved to be robust and reliable, they often reacted to disturbances by triggering unnecessary shutdowns whenever they acted over-cautiously while trying to protect valuable local assets. Fault-finding was performed by engineers, often working in close proximity to high-tension lines and live plant equipment.

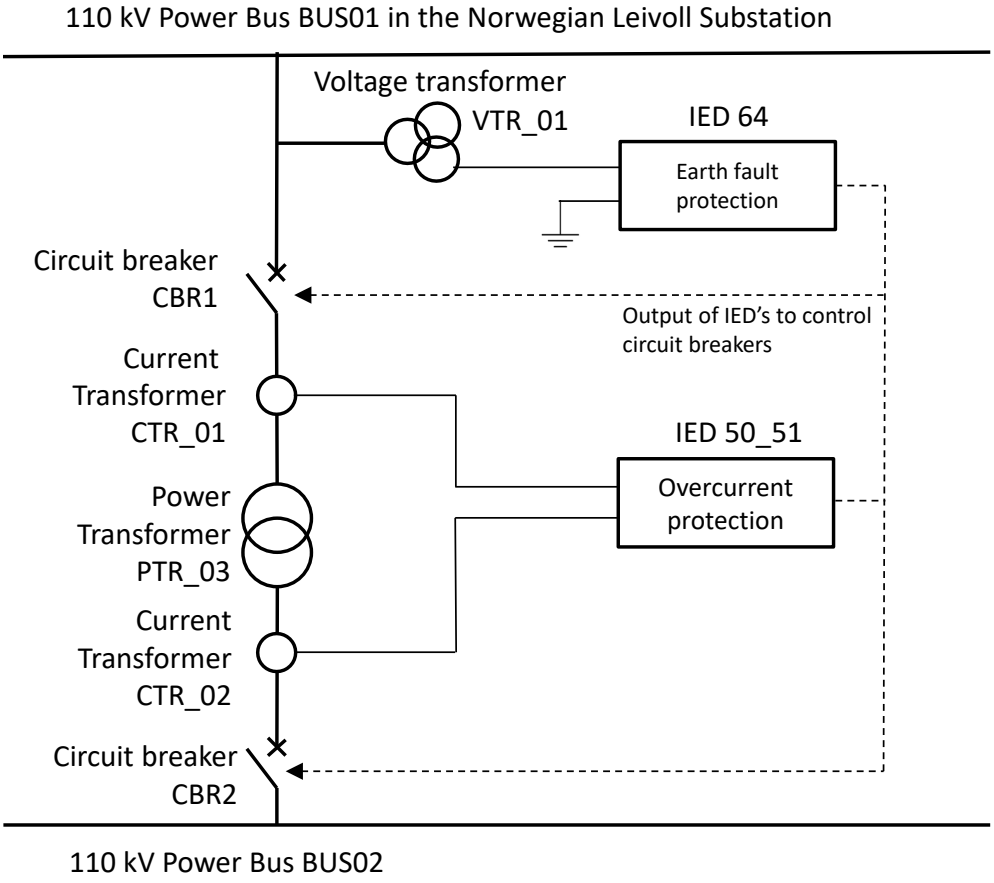

Figure 10.1: Leivoll substation represented as an IEC 61850 Single-Line Diagram.

Modern Smart Grids address this by operating intelligently within their local protection zones. They can also share information with other devices that they use to make more informed control decisions. However, the main driver for modernisation is the need to improve efficiency and drive-down maintenance costs (Ketter, Collins, Saar-Tsechansky & Marom, 2018; Carvallo & Cooper, 2015). Electromechanical equipment still controls most of the European substation automation systems, but many of the components used are now more than 30 years old (Zhabelova et al., 2016).

Figure 10.1 details the equipment used in a typical section or *bay* of a modern smart grid-enabled substation. The installation relies on both PLCs and specialised protection equipment referred to as Intelligent Electronic Devices (IEDs) (IEEE, 2008). IEDs are programmable and IEC 61499 has seen a significant uptake in the Smart Grid sector, where many IEDs now

implement their control systems using this architecture (NOJA, 2015). Substation designs are documented using Single-Line Diagrams. These employ conventional symbols for hardware that are standardised in the IEC 61850 standard (ISO, 2021). Single-Line Diagrams emphasise functional capabilities rather than the physical layout of the substation (Brand, Brunner & Wimmer, 2011). This standard also details the software interfaces used to communicate with the hardware.

This Single-Line Diagram shows the design of the Leivoll substation which featured in the work of Zhabelova et al. (2016) and Sinha et al. (2018) that provides power to the Norwegian Railway Network. The 110kV Alternating-Circuit (AC) electricity buses feed the power transformer PTR_03. This transformer is the primary equipment that needs to be protected in this section of the substation. It converts high-tension AC power that enters the substation from the Norwegian National Grid into the 15 kV needed by the trains (Giordano et al., 2011).

Two of the primary types of IEDs are shown in Figure 10.1. These are profiled and then developed in the sections that follow. Their IED device designation number is an IEEE/ANSI function number that specifies the type of fault that the IED is designed to mitigate (IEEE, 2008). In Section 10.4, the two devices are combined into a single multi-functional device that is representative of the type of equipment currently available from smart grid equipment vendors.

10.2 IED 64 Earth Fault Protection

An Earth or Ground Fault is a failure caused by the unintentional connection of a high-voltage electricity conductor to the earth. This can cause electric currents to flow between parts of the equipment that should be insulated. The transformer housings can become energised, exposing both the equipment and plant engineers to dangerous voltages (G. Zhabelova & Vyatkin, 2012). The pathway to earth generates oscillating eddy currents, often leading to secondary currents that cause further damage when the insulation that normally protects the equipment breaks down. Figure 10.2 shows these current pathways in a typical scenario. Hayes et al. (2019) demonstrated this type of fault, describing their experiment that studied a tree that fell across

a 12.5 kV distribution line. The tree was able to conduct current to ground for eight minutes before bursting into flames.

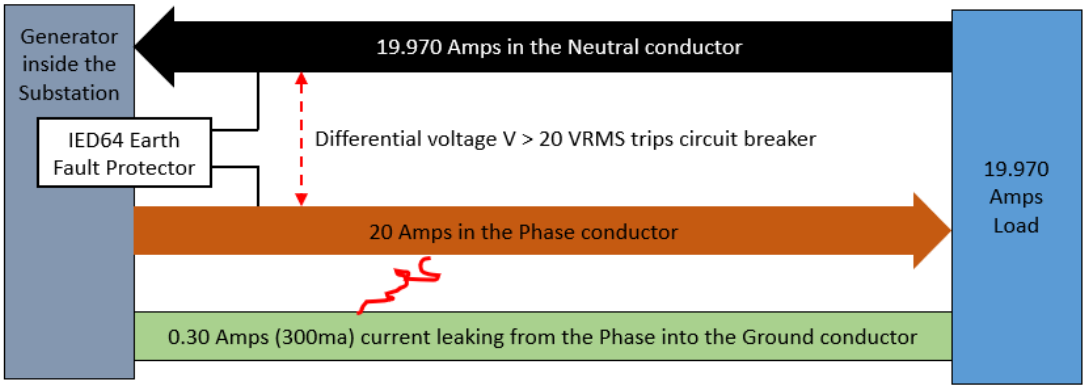

Figure 10.2: Earth Fault currents between conductors in a substation bus line.

The ISO 60255-181 standard specifies the functional requirements for ground fault protection devices (ISO, 2019a). If the oscillating current causes an imbalance between the conductors, a differential voltage is generated. The functional requirements and quality attributes for an IED 64 device to detect and respond to this type of fault are specified in Table 10.1.

Table 10.1: Functional and quality requirements for the IED 64.

Requirement and Quality IDs	**Description**	**ISO 25010 Quality Attributes**	**Traces to Function Blocks**
RQ1	whenever IED61.voltage > IED64.pickup_voltage for time > IED64.trip_delay_time then		IED_64 earth_fault
QA 1.1	CBR1.trip occurs within 10 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	E_DELAY
QA 1.2	CBR2.trip occurs within 10 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	

10.2.1 Designing an IED 64 with IEC 61499 function blocks

Figure 10.3 shows the `IED64` Composite Function Block that encapsulates the component `earth_fault` and `E_DELAY` blocks. Earth leakages are detected by measuring the voltage difference between the BUS01 and BUS02 conductors and the ground. The Voltage transformer VTR_01 provides a connection point for an Analogue-to-Digital Converter (ADC) that transforms the analogue RMS voltage into a digital value. This can then be read by the function block

`earth_fault`. The ground fault detection and protection methods were based on technical specifications from Kuphaldt (2019).

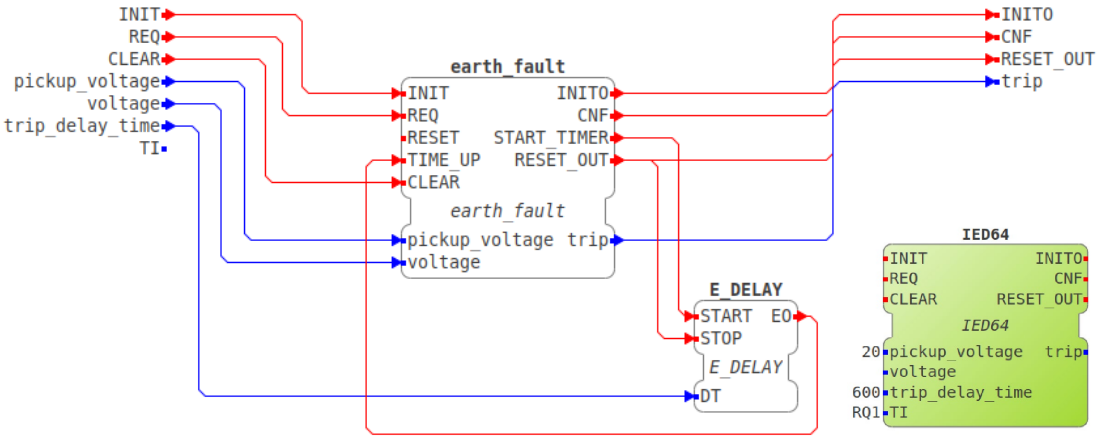

Figure 10.3: IED64 Earth Fault protection Composite Function Block.

Figure 10.4 shows the Execution Control Chart that drives the `earth_fault` function block. The input parameter `pickup_voltage` specifies the Root-Mean Square (RMS) voltage that is the acceptable limit of the AC voltage generated by the ground fault. If this limit is exceeded, `earth_fault` starts the `E_DELAY` timer with an interval specified in milliseconds on the `trip_delay_time` parameter. If the RMS value of the ground fault voltage drops below the pickup_voltage before the timer interval expires, the `CNF` event is not fired and the timer

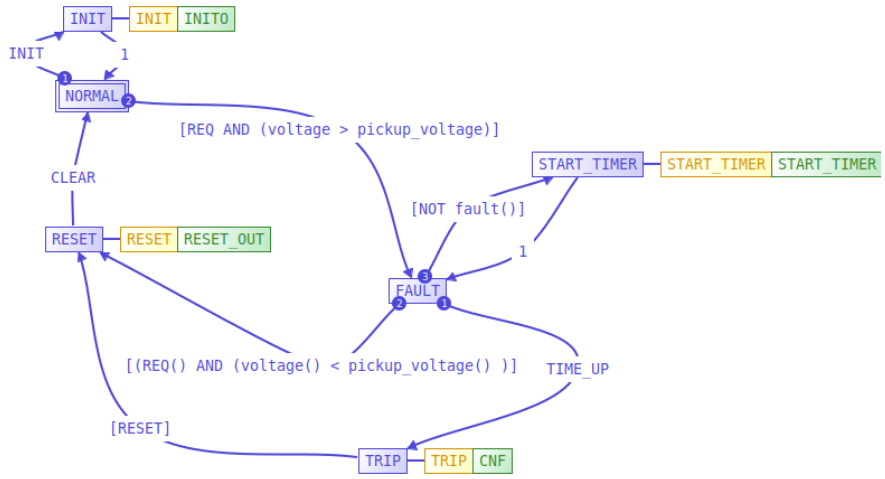

Figure 10.4: Execution Control Chart (ECC) for the `earth_fault` Function Block.

is stopped by the RESET_OUT event. If the time is exceeded, then the CNF event is triggered. This event is used to trigger the other function blocks that open the circuit breakers CBR1 and CBR2. The circuit breakers can be reset to their closed state to restore power when the input event CLEAR is received from another function block.

10.2.2 Diagnosing faults during design and operational stages

Applying the Model-Driven Development with Diagnostics protocol, the diagnostic package IED64.dpg was created while designing the IED function blocks. It contains two gimbal diagnostic scripts that the agent executes sequentially. The first script shown in Figure 10.5 demonstrates how the agent verifies the operation of the voltage trip detection. Earth Fault

```
Belief gimbal_voltage() {
    double pickup_voltage = 19;
    double voltage = pickup_voltage - 5;
    int errorCount = 0;

    IED64.REQ.gateClose();

    for (int test = 1; test <= 20; test++) {
        IED64.REQ.trigger(voltage)

        if (IED64.CNF.read()) {
            // The IED has tripped.
            if (voltage <= pickup_voltage) {
                // The IED has tripped incorrectly.
                errorCount++
                belief.add("Error : IED64 tripped incorrectly on voltage "
                        + voltage + " for pickup_voltage " + pickup_voltage);
            }
            // Reset the IED
            IED64.CLEAR.trigger();
        }
        voltage = voltage + 1;
    }

    IED64.REQ.gateOpen();

    if (errorCount == 0) {
        belief.Veracity(VeracityTypes.TRUE);
        belief.add("Nominal.");
    } else {
        belief.Veracity(VeracityTypes.FALSE);
    }
    return belief;
}
```

Figure 10.5: Diagnostic script for verifying the voltage trip behaviour of the IED 64.

devices are usually configured with a static trip delay time chosen to suit the characteristics of the equipment that is being protected (Kuphaldt, 2019; ABB, 2019). The script exercises the function block, ensuring that it does not trip when the earth fault voltage is below the `pickup_voltage` trigger level. Note how in this situation, simbIoTe is not needed since the diagnostic point can trigger an event on `IED64` that allows it to supply test voltages. Figure 10.6

```
Belief gimbal_voltage_and_time() {
    double pickup_voltage = 19; // volts RMS
    double voltage = pickup_voltage - 5;
    long trip_delay_time = 600;  // milliseconds.
    long diff = 0;
    int errorCount = 0;

    IED64.REQ.gateClose();

    for (int test = 1; test <= 20; test++) {
        IED64.REQ.trigger(voltage)

        if (IED64.CNF.read(IED64.REQ.trigger())) {
            // The IED has tripped.
            if (voltage <= pickup_voltage) {
                // The IED has tripped incorrectly.
                errorCount++
                belief.add("Error : IED64 tripped incorrectly on voltage "
                        + voltage + " for pickup_voltage " + pickup_voltage);
            } else {
                diff = IED64.CNF.elapsedTime() - trip.delay.time;
                if (diff > 0 ) {
                    errorCount++;
                    belief.add("IED64 tripped on voltage " + voltage +
                            " " + diff + "ms late.");
                } else if (diff < 0) {
                    errorCount++;
                    belief.add("IED64 tripped on voltage " + voltage +
                            " " + Math.abs(diff) + "ms early.");
                }
            }

            // Reset the IED
            IED64.CLEAR.trigger();
        }
        voltage = voltage + 1;
    }

    IED64.REQ.gateOpen();

    if (errorCount == 0) {
        belief.Veracity(VeracityTypes.TRUE);
        belief.add("Nominal.");
    } else {
        belief.Veracity(VeracityTypes.FALSE);
    }
    return belief;
}
```

Figure 10.6: Diagnostic script to verify timing and trip behaviour of the IED 64.

shows the second gimbal script. This script is used to verify the timing of the trip event on the IED. It verifies that the elapsed time from the start of the over-voltage to the time that the device trips complies with RQ1.

10.3 IED 50 and 51 Overcurrent Protection devices

While the IED64 device detects ground leakage current, that is not the only type of current fault that can occur. There are multiple overcurrent protection mechanisms, each of which is designed to protect against faults particular to the different types of transformers, electric motors, or generator fault scenarios. Overcurrent protection IEDs are classified into two primary types:

Instantaneous Overcurrent protection is provided by Type 50 IEDs (ISO, 2019a). These protection relays trip their circuit breakers instantaneously whenever the current flowing through the plant equipment they are monitoring exceeds the pre-programmed *pickup current* value specified by the engineers. Instantaneous overcurrent protection devices are typically used to protect against serious faults that are unlikely to be transient (UIDAHO, 2018).

Time Overcurrent protection is provided by Type 51 IEDs (ISO, 2019a). These trip their protection relays based on a combination of the magnitude of the current and the duration of the overcurrent in milliseconds. The formulae that are used to calculate the time that the IED should trip are described by a set of inverse exponential curves that are defined by two international standards, IEEE-242-2001 (Std, 2001) and the IEC/ANSI standard 60255-151 (ISO, 2009). These are different, but not competing standards. The purpose of having multiple time-curve specifications for time overcurrent protection is related to a concept called *coordination* (Kuphaldt, 2019). Typical substations have multiple transformers and types of generation equipment, all connected to the grid via high-tension bus lines. In a well-designed smart grid, only the IED closest to the fault will trip, isolating the smallest section of the system without causing disruption to other generating equipment. Each IED does this by being tuned to perform optimally with the plant equipment it is protecting, ensuring that it does not trip unnecessarily. The configuration employs one of the formulae from the IEC/ANSI or IEEE

standards to allow the IED to calculate the trip time dynamically. Short-time curves provide intermediate protection for situations that begin with an overload but may lead to a serious fault. Longer time curves provide protection from overloads which the system can handle for a longer durations without damage. (Bizjak & Baruti, 2017; UIDAHO, 2018).

The functional requirements and quality attributes for the combined IED 50 and IED 51 device to enable it to identify and respond to this type of fault are specified in Table 10.2. RQ2 is for Type 50 Instantaneous Overcurrent and RQ3 is for Type 51 Time Overcurrent protection.

Table 10.2: Functional and quality requirements for the IED 50 and 51 devices.

Requirement & Quality IDs	**Description**	**ISO 25010 Quality Attributes**	**Traces to Function Blocks**

RQ2	for IED50_51.curve_type == 0 then
	whenever IED50_51.Icurrent > IED50_51.Ipickup_current
	for time > IED50_51.min_trip_time then
			IED50_51
			overcurrent
			E_DELAY

QA 2.1	CBR1.trip occurs within 10 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	trip_multiplexer
QA 2.2	CBR2.trip occurs within 10 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	trip_multiplexer

RQ3	for IED50_51.curve_type > 0 then
	whenever IED50_51.Icurrent > IED50_51.Ipickup_current
	for time > IED50_51.calc_trip_time then
			IED50_51
			overcurrent
			E_DELAY

QA 3.1	CBR1.trip occurs within 10 ms and	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	trip_multiplexer
QA 3.2	CBR2.trip occurs within 10 ms.	4.2.2 Performance Efficiency 4.2.2.1 Time behaviour	trip_multiplexer

The breaker opening time of 10 ms specified in for the QAs is the nominal switching time needed to open a solid-state circuit breaker (Antonova, Nardi, Scott & Pesin, 2012). Feng et al. (2018) explain that for modern devices, this now ranges from 0.5 microseconds to 10 seconds and is dependent on the magnitude of the current being switched. This reaction time is in addition to the Instantaneous minimum or Time Overcurrent trip time delay calculated by using the appropriate time-curve.

10.3.1 Designing an IED 50 and 51 with IEC 61499 function blocks

Figure 10.7 details the internal structure of the `IED50_51` Composite Function Block. The design looks similar to the `IED64` however the internal operation of the block is quite different. The IED composite function block has been designed to be configurable, providing either

Instantaneous or Time Overcurrent protection. The type of protection is specified by the input parameter `curve_type` which an enumerated type from IEC/ANSI standard 60255-151 (ISO, 2009). The equations it uses to calculate the trip time are similar to those specified in the IEEE-242-2001. For the purposes of this research, the IEC/ANSI standard 60255-151 was chosen based on the depth of the information and examples provided for that standard in Kuphaldt's engineering guide (Kuphaldt, 2019).

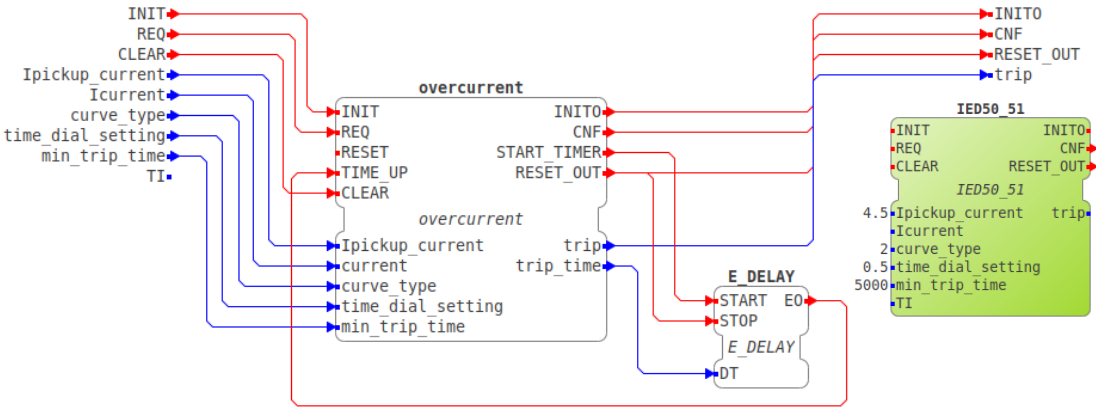

Figure 10.7: IED 50 and IED 51 Overcurrent Fault Composite Function Block.

10.3.2 Diagnosing faults during design and operational stages

Diagnosing faults in an Overcurrent IED requires the agent to determine the expected trip time dynamically, based on its configuration parameters. It performs the calculation as soon as the input current it measures is greater that the limit specified for the IED. The IED Type 50 and 51 devices were used to evaluate how accurately the engine could capture the timing of function block events to compare the performance of the IED with the calculated trip times. Assuming that the 4diac timing blocks are accurate to millisecond resolutions, the experiments measured how repeatable the capturing of the timing was by the engine. The Time Overcurrent formula from IEC/ANSI standard 60255-151 is shown in Definition 10.3.1. Note how the engineering terms used for parts of smart grid devices have preserved many of the conventional names that have been in use for generations to describe the original electromechanical devices.

Definition 10.3.1 (Inverse Time Overcurrent) $t = TDS(C_1 + \frac{C_2}{Mvalue^P - 1})$ *where*

- t *is the trip time in seconds,*
- TDS *is the Time-Dial setting, typically a value between 0.5 and 15. This is traditionally used to fine-tune the response of mechanical protection devices,*
- C_1 *is a constant representing the equivalent inertia of the mechanical measurement equipment,*
- C_2 *and* P *are constants specific to the time curves specified in IEC/ANSI 60255-151, and*
- $Mvalue = \frac{Imeasured}{Ipickup}$*, the proportional overcurrent measure where* $Imeasured$ *is the current measured from the current transformers and* $Ipickup$ *is the current limit that should not be exceeded, referred to as the pickup-current.*

The expected Time Overcurrent protection trip times calculated by the agent in the diagnostic package script are shown on the graph in Figure 10.8. Note that the vertical time axis is logarithmic. All IED type 51 trip time calculations are based on inverse curves since the IEDs must trip faster for higher $Imeasured$ currents. If the IED does not trip in this time, or trips after the overcurrent has subsided, the device has failed. The IED can be configured as a Type 50 for Instantaneous Overcurrent protection by specifying a `curve_type` of zero and a `min_trip_time` in milliseconds. A function within the script uses code that replicates the C++ function used by the `IED_50_51` function block to calculated the expected trip time when the `REQ` event is triggered. Ideally, the function in the diagnostic script should be written from the requirements by a different developer to the one who creates the IED C++ function. This would enable a more robust code-comparison later for this critical function by implementing a blind separation of the developers. This is similar to the team separation techniques that are used to develop the Airbus double-redundant flight control systems (Goupil et al., 2014).

Table 10.3 details the results from sets of 100 consecutive readings for a range of overcurrent values. The $Ipickup$ was set to 4.5 Amps and the time curve used was the Type 2 IEC/ANSI Very Inverse (U3 curve). Note that the minimum and maximum readings are within three ms in

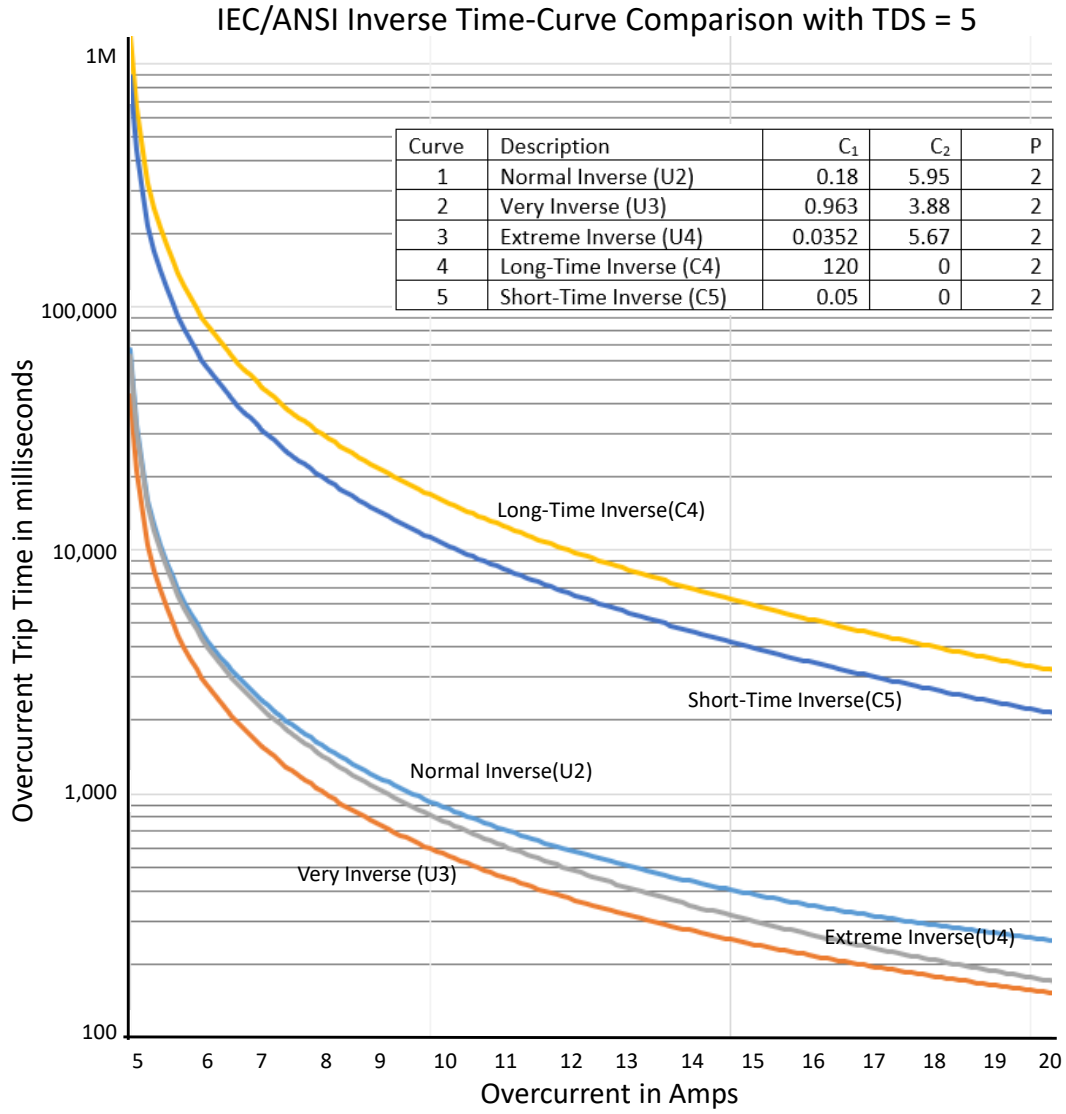

Figure 10.8: IED Type 51 IEC/ANSI Time Overcurrent Inverse Curve Characteristics.

each case, which correlates with the Population Standard Deviation μ that was always less than one. This implies that across all currents, the engine was reliably capturing short-duration events down to 48 ms with a margin of error of three ms of the expected value. The deviation may be partially due to other Linux operating system processes interfering with the function block execution cycle. However, this verifies the ability of the diagnostic points to capture timing events asynchronously and repeatably with a high-degree of accuracy.

10.3.3 Diagnostic techniques for Composite Function Blocks

Both the `IED64` and the `IED50_51` IEDs are constructed from Composite Function Blocks. These provide a convenient and efficient way of encapsulating multiple function blocks into a single, re-usable unit. However, Composite Function Blocks only expose the event, input, and output ports that the designer wishes to make available. For `IED64`, this means that the internal `earth_fault` and `E_DELAY` function blocks are private instances that are not visible to other parts of the function block application. Therefore, they are also hidden from the agents who cannot monitor the internal data exchange occurring. This implies that the `E_DELAY.START` event is not visible to the diagnostic points and cannot be monitored to check for faults that may occur if the `earth_fault` or `overcurrent` function blocks do not start the timer correctly.

However, the Composite Function Block Type Definition file provides all the information needed to instantiate the composite block and its components at run-time. This suggested a novel way for the agents' *rewire()* method to be extended to allow diagnostic packages to specify DPs

Table 10.3: Timing results for the IEC/ANSI Very Inverse (U3) IED configuration.

Icurrent (Amps)	Expected trip time (ms)	Min (ms)	Max (ms)	Mean (ms)	Standard Deviation μ
30.00	92.80	91.00	93.00	92.00	0.43
35.00	80.75	79.00	81.00	80.00	0.68
50.00	63.99	62.00	65.00	63.00	0.67
75.00	55.16	53.00	57.00	55.00	0.82
100.00	52.08	51.00	54.00	52.00	0.68
250.00	48.78	47.00	49.00	48.00	0.61
1,000.00	48.19	47.00	50.00	48.00	0.73

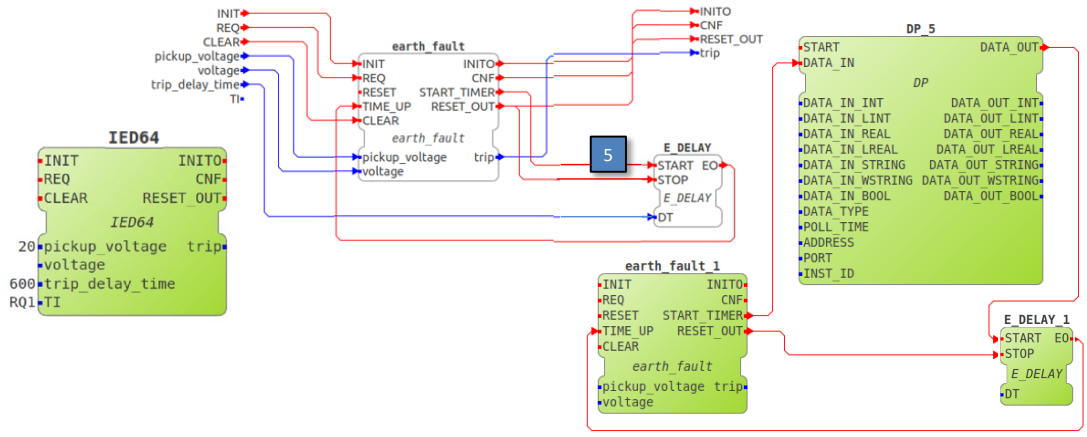

Figure 10.9: Instantiating a Diagnostic Point within a Composite Function Block.

that exist within composite blocks rather than being restricted to only operate externally to them. Figure 10.9 shows the internal structure of the IED64 block. The blue square shows where the engineer has indicated that a diagnostic point should be inserted. The agents *rewire()* function has recognised that the location of the block the diagnostic point is to capture is not already part of the function block application System-Under-Diagnosis belief structure. However, the IED64 has a type definition that classifies it as a Composite Function Block. Figure 10.10 shows the new attribute named FB in the IED64.dpg file. This specifies the name of the function block within the composite structure that allows the event to be resolved during the rewiring.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBDiag SYSTEM "DiagnosticElement.dtd">
<FBDiag Name="IED64" Comment="Earth Fault diagnostic information for IED64 ver 1">
    <DPS>
        <DP FB="earth_fault" Event="START_TIMER"/>
    </DPS>
```

Figure 10.10: Diagnostic Point specification for a Composite Function Block component.

The attributes identify that the block called earth_fault has START_TIMER event to be monitored is internal to the composite block. The *rewire()* method then loads the composite elements specified in IED64.fbt so they become additional nodes in the System-Under-Diagnosis belief structure. The right-hand side of the diagram shows that the earth_fault and the E_DELAY instances have been renamed with a unique suffix when creating the diagnostic harness and wiring in DP_5. This renaming ensures that there is no conflict with other function block instances. The script pre-compiler will note the new instance names when resolving the mappings for the diagnostic points referenced as objects in the script.

While the example shown is simplistic, this functionality becomes important when a Composite Function Block is hiding a lot of internal functionality where faults might occur. The method has been partially tested with the current engine and will be further refined as part of the future work outlined in the final chapter.

10.4 Investigating multi-function IED configurations

The ASEA Brown Boveri (ABB) REG650 is a multi-function IED that offers a range of configurable protection modes in a single unit (ABB, 2019). The engineering technical manual for this device was used to model the function block application that combined the two IEDs into a single unit. This provided an ideal environment in which to deploy a team of co-operating agents. Each agent monitored a different part of the application, ensuring that the timing constraints were met and watching for disturbances in their designated areas. This configuration also provided a way to examine how well the agents coped with multiple or simultaneous failures.

Figure 10.11: ABB REG650 IED

Figure 10.12 shows the team of five named agents and the groups of DPs they have been assigned to manage. The colour used for each agent matches the colour of the DPs they are connected to, shown as rectangles near the input or output port they are capturing.

The Team Manager agent co-ordinates the goals of each agent, assigning them a section of the application to work on. Agent *alpha* executes a script that injects a set of test currents and voltages that simulate both normal operating conditions and a set of representative faults. Agent *beta* is initially given the goal of monitoring `IED50_51`. Figure 10.13 details the skills it uses and its belief management. In a similar way, agent *gamma* watches the operation of `IED64` while agent *delta* observes the `trip_multiplexer` subsystem that transmits the trip command to the circuit breakers CBR1 and CBR2.

The interaction between the Team Manager agent and the *beta* agent is modelled on the monitoring techniques developed for the NASA Curiosity Mars Rover described by Benowitz (Benowitz, 2014). Section 2.4.1 earlier explained how the rover's Fault Protection Engine implements discrete monitor programs whose responsibility is to observe a discrete Curiosity

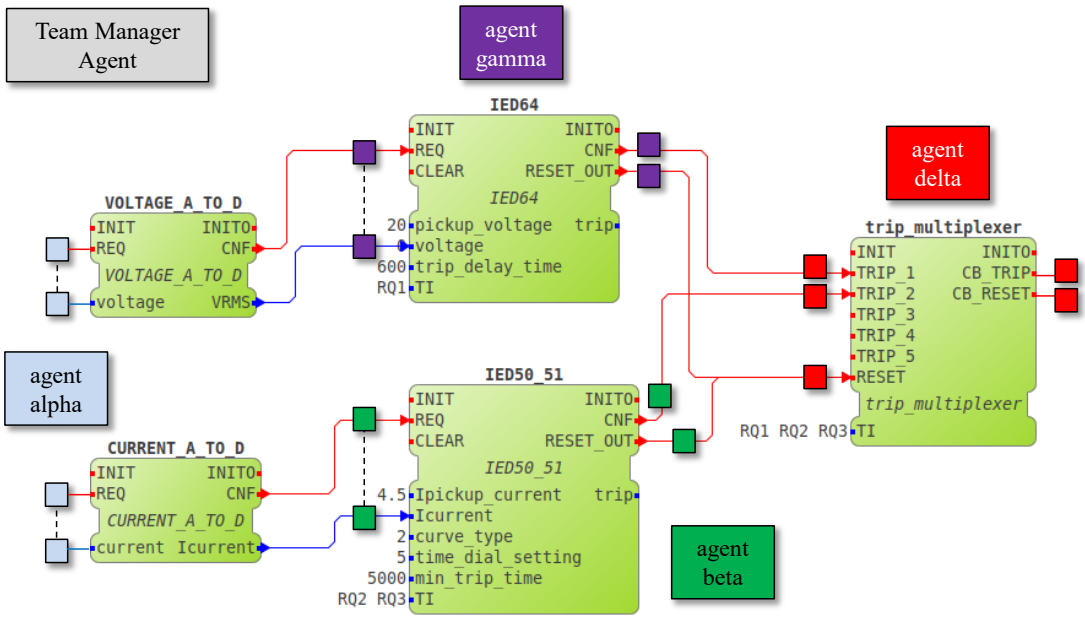

Figure 10.12: REG650 application diagnostic configuration with multiple agents.

subsystem or component. When anomalous behaviour is observed, the monitor code raises an error flag which latches until it is cleared. This allows the monitor to alert the Rovers Fault Diagnostic Engine executive functions that a problem has been detected. This approach scales well since the executive, which operates similarly to the Team Manager agent, has to prioritise the way it manages multiple, cascading failures.

The script also shows how the monitor script can dynamically load static configuration parameters such as `curve_type`, `Ipickup`, and `time_dial_setting`. These are available in the function block application System-Under-Diagnosis belief structure the Team Manager agent constructed and then shared with its team of agents. When the monitor observes the `REQ` event, it first determines if it should expect the current on `Icurrent` to trip the IED. If the IED is likely to trip, the agent delays for 10 ms more than the calculated trip time. If the IED has tripped successfully, it returns a belief with a v set true. The belief also carries information about the trip event. Alternatively, if the IED does not trip in the required time, the belief is returned with a v of false. This has the effect of latching a flag that is detected by the Team Manager agent the next time it performs its *MONITOR_AGENT* goal. By that time, agent *beta* will have switched to its idle state to await further instructions. Similar monitor and gimbal scripts were

```
Belief monitor() {
    double Icurrent = 0;
    double tripTime = 0;
    double calcTripTime = 0;

    // Retrieve the configuration settings of this IED.
    int curve_Type = IED50_51.curve_type;
    double Ipickup = IED50_51.Ipickup_current;
    double time_dial_setting = IED50_51.time_dial_setting;

    if (IED50_51.REQ.read) {
        // The IED has received a new current read event.
        Icurrent = IED50_51.REQ.value();
        if (Icurrent > Ipickup) {
            // This should trip the IED.
            start_overcurrent = IED50_51.REQ.timestamp();
            calcTripTime = calcInverseOvercurrent(curve_type, Ipickup, Icurrent, time_dial_setting);
            delay((int) (calcTripTime + 10));
            if (IED50_51.CNF.read(IED50_51.REQ.timestamp())) {
                end_overcurrent = IED50_51_CNF.timestamp();
                tripTime = (end_overcurrent - start_overcurrent);
                if (Math.abs(calcTripTime - tripTime) < 3) {
                    belief.Veracity(VeracityTypes.TRUE);
                    belief.add("IED50_51 tripped correctly for Icurrent " + Icurrent + "A in "
                            + tripTime + "ms.");
                } else {
                    belief.Veracity(VeracityTypes.FALSE);
                    belief.add("IED50_51 tripped incorrectly in" + tripTime + "ms for Icurrent "
                            + Icurrent + "A.");
                }
            }
        }
    }
    return belief;
}
```

Figure 10.13: The script used by agent *beta* to monitor the overcurrent IED

developed and evaluated for the `IED64` and `trip_multiplexer` function blocks that were monitored by agents *gamma* and *delta*.

The Team Manager agent then determines which team member should be assigned to investigate and diagnose the fault, depending on where the fault is located. Each agent maintains its own set of Dynamic Diagnostic Beliefs. When instructed to diagnose a section of the application it was previously monitoring, it first examines the past beliefs to locate previous fault evidence. Figure 10.14 shows the script that agent *beta* chose from the diagnostic resources to gimbal the `IED50_51`. To demonstrate the functionality, an error was introduced into the `IED50_51` time curve algorithm to simulate faults. For specific overcurrents, the trip-time calculation was made to fail.

```
Belief gimbal() {
    double Icurrent = 0;
    boolean inTrip = false;
    long start_overcurrent = 0;
    long end_overcurrent = 0;
    double calcTripTime = 0;
    double tripTime = 0;
    int errorCount = 0;

    // Retrieve the current configuration settings of this IED.
    int curve_Type = IED50_51.curve_type;
    double Ipickup = IED50_51.Ipickup_current;
    double time_dial_setting = IED50_51.time_dial_setting;

    // Isolate the IED so it can be gimbaled.
    IED50_51.REQ.gateClose();
    IED50_51.CNF.gateClose();
    IED50_51.RESET_OUT.gateClose();

    Icurrent = Ipickup - 2;
    for {cnt = 0; cnt < 100; cnt++) {
        if (Icurrent >= Ipickup) {
            // This should trip the IED.
            tripTime = calcInverseOvercurrent(curveType, Ipickup, Icurrent, timeDialSetting);
            IED50_51_REQ.trigger(Icurrent);
            start_overcurrent = IED50_51_REQ.timestamp();
            inTrip = true;
        } else if (Icurrent < Ipickup){
            // This should not trip the IED.
            IED50_51_REQ.trigger(Icurrent);
            start_overcurrent = IED50_51_REQ.timestamp();
            inTrip = false;
        }

        if (IED50_51.CNF.read(IED50_51.REQ.timestamp())) {
            if (!inTrip) {
                end_overcurrent = IED50_51.CNF.timestamp();
                // The IED should not have tripped.
                IED50_51_REQ.trigger(Icurrent);
                tripTime = IED50_51_REQ.timestamp() - start_overcurrent;
                errorCount++;
                belief.add("IED50_51 tripped incorrectly for Icurrent " + Icurrent + "A in "
                        + tripTime + "ms.");
                isTesting = false;
            } else {
                // Did the IED trip correctly?
                tripTime = IED50_51_REQ.timestamp() - start_overcurrent;
                if (Math.abs(calcTripTime - tripTime) < 3) {
                    belief.add("IED50_51 tripped correctly for Icurrent " + Icurrent + "A in "
                            + tripTime + "ms.");
                } else {
                    errorCount++;
                    belief.add("IED50_51 tripped incorrectly in" + tripTime + "ms for Icurrent "
                            + Icurrent + "A.");
                }
            }
        }
        Icurrent = Icurrent + 1.0;
    }
```

```
    if (errorCount == 0)
        belief.Veracity(VeracityTypes.TRUE);
        belief.add("IED50_51 is performing nominally");
    } else {
        belief.Veracity(VeracityTypes.FALSE);
    }

    // Restore the links to the other function blocks.
    IED50_51.REQ.gateOpen();
    IED50_51.CNF.gateOpen();
    IED50_51.RESET_OUT.gateOpen();
    return belief;
}
```

Figure 10.14: The script used by agent *beta* to diagnose the faulty IED50_51

Figure 10.15 shows the log created during one of the sessions that displays the beliefs recorded by the agents that captured the fault evidence and passed it to the Team Manager agent.

```
DIAGNOSTIC LOG 08.07.2021 20:08

AppName        type:SuD   v: true  desc: Smart_Grid_02
AppPath        type:SuD   v: true  desc: /home/badger/Development/4diac/Smart_Grid_02/
LoadStatus     type:SuD   v: true  desc: Application loaded successfully
DeployedStatus type:SuD   v: true  desc: Application deployed successfully

alpha: IED50_51         s: MONITOR  v: true  desc: trigger 4A.
beta:  IED50_51         s: MONITOR  v: true  desc: IED did not trip.
alpha: IED50_51         s: MONITOR  v: true  desc: trigger 5A.
alpha: IED50_51         s: MONITOR  v: true  desc: IED50_51 tripped correctly for Icurrent 5A in 8318ms.
delta: trip_multiplexer s: MONITOR  v: true  desc: CB_TRIP triggered.

alpha: IED64            s: MONITOR  v: true desc: trigger 30V.
gamma: IED64            s: MONITOR  v: true desc: IED64 tripped correctly for voltage 30V in 602ms.

alpha: IED50_51         s: MONITOR  v: true desc: reset IED50_51.
alpha: IED64            s: MONITOR  v: true desc: reset IED64.

delta: trip_multiplexer s: MONITOR  v: true  desc: CB_RESET triggered.
delta: trip_multiplexer s: MONITOR  v: true  desc: CB_RESET triggered.

alpha: IED50_51         s: MONITOR v: true  desc: trigger 51A.
beta:  IED50_51         s: MONITOR v: false desc: IED50_51 tripped incorrectly in 100ms for Icurrent 51A.

beta:  IED50_51         s: GIMBAL  v: false desc: IED50_51 tripped incorrectly for Icurrent 51A in 100ms.
                        s: GIMBAL  v: false desc: IED50_51 tripped incorrectly for Icurrent 53A in 202ms.
                        s: GIMBAL  v: false desc: IED50_51 tripped incorrectly for Icurrent 60A in 305ms.
```

Figure 10.15: Diagnostic log of beliefs that captures a fault with the IED50_51

The first section of the log shows the beliefs formed by the Team Manager agent while analysing the function block application, followed by creating and then deploying the diagnostic test harness. Agent *alpha* then generates test currents and voltages for the IEDs. The final section of the log shows the beliefs generated after *alpha* generated an overcurrent of 51A, one of the values which generates a fault in the IED. Agent *beta* detects that the trip time is incorrect. It then returns a failed goal status to the Team Manager agent. It is then assigned a new goal to investigate the fault, whereupon it begins executing the gimbal script shown in Figure 10.14.

Note that only a single belief reporting multiple failures is returned at the end of this script. It has executed a total of 100 different test current readings after isolating the `IED50_51` function block with *gateClose()* commands. The three faulty time calculations for 51A, 53A and 60A were correctly identified. During the development and evaluation of the engine, belief information was returned from scripts as human-readable free text. However, the belief data structure will be extended in later work to hold packed, named data fields. These would make it possible for the agents to perform a more structured analysis of the results semi-autonomously when forming their diagnosis.

10.5 Benchmarking the Smart Grid components

Techniques for evaluating telemetry, accurate timing, reliability, and fault-tolerance are crucial when validating smart grid protection devices. Table 10.4 summarises the benchmarks demonstrated by the engine and the diagnostic methods in this chapter.

Table 10.4: Benchmarking the engine and the diagnostics for Smart Grids.

Benchmark	Requirement	Size and complexity	Description
B6	Capturing and evaluating telemetry.	Large, complex.	Demonstrate the reliable capture of events and data being exchanged between function blocks.
B7	Accurately capture and compare the timing of events.	Large, complex.	Enable the DPs to accurately capture the time that they either triggered or observed an event. Return the event times asynchronously back to the agents. Store the event times so that they can be retrieved and used to calculate the elapsed time between different input and output events.
B8	Diagnose events occurring within Composite Function Blocks.	Moderate, moderately complex.	The function blocks inside a Composite Function Block are now visible to the fault diagnostic engine as elements within the System-Under-Diagnosis belief structure. This now allows them to have DPs wired-into their data and event connections.
B9	Collaboration and task sharing between agents.	Large, complex.	The instantiation of multiple, independent agents that execute tasks on their own threads required methods to enable the GORITE Team Manager functionality to co-ordinate the goal assignment, evaluation, and re-assignment for teams of agents as they shared diagnostic tasks.

This chapter has demonstrated practical ways of capturing accurate time measurements from individual function blocks asynchronously. The DPs receive trigger commands and generate

their own epoch timestamps, returning them to the engine reliably. DPs that capture events generate their own timestamps and return them in the same way. This method ensures that the time values are captured as close as possible to the moment the function block event fired. The DPs provide a way for these timestamps to be transmitted back to the engine asynchronously rather than relying on the engine or agent to estimate the event time. There is no evidence that this additional functionality introduced any significant overhead into the operation of the DPs or the function block application.

This work proved to be complex, especially those parts related to matching the correct input event to a corresponding output event, including those that occurred on different function blocks. The architecture of the FIFO packet queues provided a robust structure that could be used to carry the timestamps and their identities reliably.

The smart grid example also provided a way to evaluate the characteristics and capabilities of multiple agents as they shared diagnostic tasks. While the internal engine and GORITE functionality was complex to redevelop, it did not introduce additional complexity into the diagnostic scripts. The new code also provides ways for agents to exchange information directly between themselves. This functionality was discussed in Section 7.1.1 where agent communication was introduced as an Interaction Belief skill. This was not explored in depth during this set of experiments, but it is discussed further in the conclusions and future work in Section 11.2 in the context of coordinating more complex diagnostic tasks.

Chapter 11

Evaluation, Reflections, and Future Directions

"I learned very early the difference between knowing the name of something and knowing something."

Richard Feynman (1918-1988)

This thesis began with a quotation from Richard Feynman; it seems fitting that it should end with one from him. It was stated in the Introduction that his observation "*what I cannot create, I do not understand*" has had a profound influence on this research (Feynman, 1988). It led to the decision to learn and then refine a methodology, coupled with the construction of the artefacts profiled in the preceding chapters. In doing so, the research sought to better understand the nature of faults, what that implies for ICPS designers, and how they could address these issues earlier in their design phases by applying a Model-Driven Development methodology augmented with diagnostics.

The previous chapters presented significant aspects of the fault diagnostic engine by considering the intricacies of modelling and fault-finding. Each chapter explored how the different parts of the Software Architecture proposed in Chapter 4 were realised to support the research aims. The components of the engine were then developed, evaluated, and refined by considering how well they satisfied the functional and quality metrics specified for them in the architectural

documents.

This final chapter draws those individual evaluations together. It considers the way the research was conducted, discussing aspects of both internal and external validity. It revisits the Design Science Research methodology, addressing how well it supported the research and development processes. It also evaluates the Model-Driven Development with Diagnostics methodology that resulted from this work and the role the engine plays in supporting it. The final sections then examine how closely the engine conforms to the criteria for reaching NASA Technology Readiness Level 5 (Mankins, 2009). The definition of TRL 5 compliance that was adopted and then refined for this research was detailed earlier in Table 2.6. This level marks the point where a technology has been trialled successfully in a relevant environment. TRL 5 specifies that the engine must be able to search for and identify faults in a relevant or representative physical environment. Coupled with these conclusions, the further development that is proposed for the methodology, the engine, and the agents is presented in the same context.

11.1 The framework for considering research validity

Validity in research is concerned with the truth and usefulness of the claims made for it when it is evaluated (Trochim & Donnelly, 2001). This presupposes that validity must not be considered to be a property of the theory, its artefacts or its design. Rather, it is a characteristic based on what can be inferred from what the research presents and the outcomes the study claims (Cook, Campbell & Shadish, 2002). In that sense, validity is a by-product created by performing the research well after establishing a clear and robust research protocol.

When evaluating the outcomes from a research program, two key perspectives must be considered. The *Internal Validity* is concerned with how the conclusions can be warranted while the *External Validity* focuses on how deeply the work could be generalised. For each of these aspects of validity, McGrath and Brinberg (1985) posit that research is undertaken within three interrelated but analytically distinct domains: the *substantive*, the *conceptual*, and the *methodological*. The following subsections examine each of these concepts in-context with the

aspects of the research validity they seek to quantify.

11.1.1 Internal Validity

Internal validity is concerned with the extent to which the research establishes a dependable cause-and-effect relation between the intentions and design of the research and its outcomes (Wright, Kim & Perry, 2010). The degree of internal validity attained is therefore largely dependent on how well the research was formulated, designed, and how rigorously the work was performed (Wright et al., 2010). McGrath and Brinberg (1985) propose that validity cannot be thought of as a binary measure. Rather, it is a measure of confidence. In that sense, validity is a measure of confidence that the findings are robust with a lower level of ambiguity. The degree of internal validity achieved will be heavily dependent on the procedures that were agreed upon when the research design was established. This aspect of validity is also influenced by how well those procedures and processes were adapted during the research to accommodate the inevitable changes demanded as findings emerge and are better understood.

In the case of this research, the Design Science research plan sought to derive empirical measures of conformance with the aims of the engine via a rigorous software architecture design phase. Protocols such as Attribute-Driven Design (Wojcik et al., 2006) and the Architectural Trade-off Analysis Method (Kazman et al., 2000) provided standards-driven frameworks which are both well-proven and widely-accepted in the software engineering domain. Trochim and Donnelly (2001) propose that internal validity should rely on statistical evaluations to provide empirical measures. Hence the analysis of the timing of the function block events captured by the diagnostic points relied on statistical measures such as the standard deviation of the telemetry across multiple inverse curve configurations as shown previously in Table 10.3. The establishment of the evaluation framework was critical to the research. It provided a stronger empirical foundation for determining how closely the artefacts met their design criteria.

11.1.2 External Validity

In contrast, External Validity is a measure of how well the findings can be generalised to apply to an environment outside the internal confines of the research. Can the laboratory outcomes or research field trials be applied to other practical situations that may differ in scale or scope to those examined during the research? Findlay et al. argue that there has been a long-standing credibility argument between the design-based perceptions of the perceived reality of internal validity measures versus how well those same results hold when applied beyond the immediate objects of their investigation (Findley, Kikuta & Denly, 2021). Campbell's pioneering work on external validity established the understanding that robust sets of evaluation criteria not only provided guidance to avoid research pitfalls, but that they also ensure that all artefacts are held to the same standard (Campbell, 1957). Hence, external validity can only be established when the scope of the claims is clearly defined (Goertz & Mahoney, 2012). No research can justify claiming that everything was met adequately or applies universally; there are practical limits to how much a methodology or technique can be generalised (Cook et al., 2002).

11.1.3 The Substantive, Conceptual, and Methodological domains

McGrath and Brinberg (1985) proposed a framework in which to consider their three domains: the *substantive*, the *conceptual*, and the *methodological*. Their Validity Network Schema (Brinberg & McGrath, 1983) was used in the following sections to consider both the internal and the external validity those differing perspectives present. Figure 11.1 illustrates the relationship between their domains:

11.1.4 The Substantive domain

The Substantive domain perspective was used to consider the substance of the deliverables that a body of research has produced. Artefacts, insights, and methodologies the research presents are substantive only if they have a firm basis in reality that leads to them being considered important or meaningful (Lesko et al., 2017). The arrow A links Design as a key process that leads to

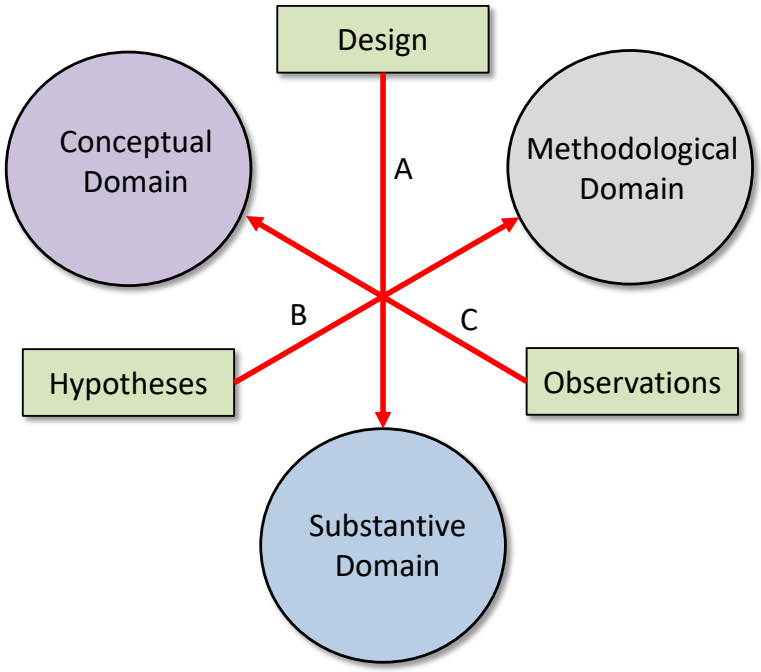

Figure 11.1: McGrath and Brinbergs Validity Network Schema.

substantive outcomes in this domain from their Validity Network Schema. Internal validity was sought first by addressing RQ1 via a literature review, seeking to establish fault-finding techniques that were already being used in the wild that might be applicable to IEC 61499. That also encompassed RQ2, which sought to identify gaps in the diagnostic methodologies already used for function blocks. Some of those gaps are manifested in the IDEs used to develop function block applications. It was also evident that there were limitations in Test-Driven Design (Christensen, 2017) and Unit Testing (Hametner et al., 2014) approaches currently being applied. These did not translate into methods that could be used to perform longer-term fault finding throughout the life-cycle of IEC 61499 applications. It also became evident that Built-In-Self-Test methods such as those discussed by Hale et al. (2018) and Smith et al. (2009a) were intrusive and inflexible. The approach developed in this research led to the creation of light-weight Diagnostic Points that were themselves function blocks. This made it easier for them to operate within the function block runtime environment since they were able to operate seamlessly within the event-driven cycles that are core to the IEC 61499 architecture. Off-loading the diagnostic decision making to agents within the engine helped to realise the quality

Table 11.1: Mapping the thesis chapters to publications from this research

Thesis chapters	**Peer-reviewed publication from the work in these chapters**	**Review rounds**	**Citation count**
Reviewing the Current Literature.	*Finding Faults: a scoping study of Fault Diagnostics for Industrial Cyber-Physical Systems*. In Elsevier *Journal of Systems and Software*, Volume 168, May 2020.	4	12
Architecting the Engine.	*Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems.* In MDPI *Future Internet* in their Special Issue *Modern Trends in Multi-Agent Systems*, July 2021	3	0
Creating Diagnostic Points *and* Model-Driven Development in Practice.	*Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems.* In *IECON 2020 46th International Conference of the IEEE Industrial Electronics Society*, October 2020.	1	2
What an Agent Believes.	*Employing Agent Beliefs during Fault Diagnosis for IEC 61499 Industrial Cyber-Physical Systems.* In *IECON 2020 46th International Conference of the IEEE Industrial Electronics Society*, October 2020.	1	0
Model-Driven Development in Practice *and* Smart Grids and Agents.	*TORUS: Scalable Requirements Traceability for Large-Scale Cyber-Physical Systems.* In ACM *Transactions on Cyber-Physical Systems*, Vol.3, No.2, October 2018.	2	9

attributes of being light-weight, fast-acting, and unobtrusive that were mandated in the software architecture. The essence of that approach is reflected in the arrow labelled A in McGrath and Brinberg's diagram. This links the Design element to the Substantive Domain by implementing the set of substantive elements described here.

Internal validity was also sought by publishing the designs, methodologies, and findings in Q1 and Q2-ranked journals and in A-ranked conferences. This exposed the research to repeated rounds of peer-review, leading to re-consideration and refinements that proved to be both challenging and rewarding. Table 11.1 maps the thesis chapters to the publications that resulted. It also records the cycles of peer-review each publication went through as well as the citations they have gained to date.

The two exemplar systems that were designed and developed examined how well the diagnostic points and the agents could perform while fault-finding. This work refined the diagnostic point function blocks, demonstrating that they could accurately capture the timing of events to verify performance quality attributes. It also led to a novel way of diagnosing

function blocks that operated inside Composite Function Blocks. At present, IDEs such as 4diac (Strasser et al., 2008) can only test Basic Function Blocks. This work extends the fault diagnostic paradigm to allow all types of function blocks, including Service Interface Function Blocks, to be diagnosed in real-time.

The Open Source version of GORITE does not support teams of agents who operate independently on their own execution threads. This feature of the engine extended GORITE to allow agents to be instanced on their own threads. This necessitated a re-evaluation of the standard GORITE Goal state hierarchy to include an asynchronous *Executing* state that allowed the Team Manager Agent to detect when an agent was busy working on its assigned goal and should not be disturbed. That alleviated potential synchronisation issues related to Goal re-assignment that might have compromised the state management of the individual agents. Extending GORITE in this way also provides a template for other practitioners who are considering using the Open Source version to implement their own sets of independent agents.

More significant external validity could be established in the substantive domain if the diagnostic points and the agent methodology could be applied in different environments beyond the initial scope defined for this research. The nxtCONTROL (nxtControl GmbH, 2016) function block development system is IEC 61499 compliant and implements function blocks and applications in C#. Adapting the 4diac C++ function blocks to operate under .Net in C# is reasonable from an architectural perspective since the function blocks have to be designed in compliance with the IEC 61499 standard (IEC, 2013a, 2013b, 2006). However, C# relies on the Windows .Net runtime environment that introduces differences in the implementation of the TCP/IP and other APIs that were implemented to run under Linux during the current research. At this stage, it is unclear how complex some of those changes might be. Applying the engine to diagnose applications designed for other ICPS or IoT architectures would be contingent on finding ways for the agents to implement alternative diagnostic points that could operate within those architectures. The novel rewiring performed for the 4diac/FORTE runtime actually makes the deployment of diagnostic points much easier after the agents have analysed the way the target function block application has been constructed. A similar analysis method would have to

be developed to replicate similar functionality for these alternative architectures. It is reasonable to infer that the complexity involved is heavily dependent on the characteristics of the particular architecture.

11.1.5 The Conceptual domain

McGrath and Brinbergs' Validity Network Schema proposes arrow C as a way to explain observations made about the concepts, novel patterns, frameworks, abstractions, and theories the research contributes. These observations reason about the levels of internal and external validity the artefacts have achieved within this domain. With respect to internal validity, this is established via the primary conceptual aspect of this research which is the multi-agent fault diagnostic engine that was architected, constructed, and evaluated using representative exemplar systems. The research conceptualised agents who embrace three distinct types of beliefs that they rely on while fault hunting and later diagnosing the evidence they uncover. The development of the agent belief mechanisms addressed RQ3 which sought to understand to what extent Belief-Desire-Intention agents could be used to perform diagnostic tasks. The first type, their Interaction belief structure, was presented in Chapter 7. It provides an agent with an immutable set of domain-specific skills and abilities that they need to enable them to interact with elements of an IEC 61499-compliant application. This research has always avoided anthropomorphising the agents. They are not sentient, but thinking about their Interaction skills in human terms is a useful abstraction. Those skills enable them to *rewire* the event and data connections between function blocks to allow them to tap into the data stream being exchanged between the blocks. The novel diagnostic information packages created in this research enable them to identify points of interest that will provide them with the telemetry they need to make decisions. Other Interaction skills allow them to close off parts of the function block application, isolating it so it can be exercised when problems are suspected in that region.

The agents wield their Interaction skills while holding a second belief structure that they constructed themselves. The System-Under-Diagnosis belief structure provides an agent with information about each function block instance that is visible to them. The concepts that

underlie Model-Driven Development were outlined in the background sections of Chapter 2. Provan (2014) and Milis et al. (2016) both noted that creating practical models that are sufficiently information-rich is labour-intensive. To be useful, these models must support Model-Based fault identification techniques that rely on detecting situations where some aspect of a model is invalidated. However, purely static models require effort to keep them up-to-date during the development phases. For this reason, the engine implements methods that allow the agents to dynamically create their own models. These are consistent with but distinct from the models proposed for Model-Driven Development with Diagnostics in Chapter 9. As the design changes, they use their skills to update their System-Under-Diagnosis belief structure which continues to provide them with a conceptual model of the application they are observing. While the agents can share a single System-Under-Diagnosis belief structure, the engine also supports multiple structures to identify and isolate distributed or partitioned IEC 61499 applications. In these situations, the engine assigns the applicable System-Under-Diagnosis belief structure to the agent that is working in that partition or region of the distributed application. This addresses stability and resource quality requirements for the engine.

The third belief structure defines beliefs about what is happening, based on the behaviours the agents have observed. Dynamic Diagnostic Beliefs capture the veracity or degree of truth about propositions the agents make. These are most often judgement calls about the performance of an individual function block. As the agents consider these beliefs as a whole, they conceptualise an evidence-based diagnosis. Since these belief constructs are part of the engine and not the diagnostic points, it is likely that the engine and the agents can be generalised to other domains more easily than the diagnostic points themselves could. The engine and GORITE are written in Java, which provides a degree of portability across multiple operating systems. These factors contribute to the degree of External validity this research claims for the engine architecture and the belief paradigms the agents embody. In doing so, this aspect of the research addressed RQ4 which asked how a fault diagnostic engine could be architected and developed to satisfy the requirements of RQ1, RQ2 and RQ3.

11.1.6 The Methodological domain

McGrath and Brinberg (1985) argue that all methods have limitations. Their Validity Network Schema arrow B proposes that the hypotheses that underpin the research's core theories can be evaluated by applying appropriate methods. Two methodological aspects of the fault diagnostics research were considered. The Design Science methodology outlined in Chapter 3 provided the framework in which to conduct the research. The second aspect considered is the Model-Driven Development with Diagnostics methodology, executed via a phased V-Model model, that this research proposes.

A Design Science methodology was employed in the previous Master's research (Dowdeswell, 2016). Due to limitations inherent in the version of the nxtCONTROL function block development system available at the time, it was difficult to produce executable prototypes of the function block applications during that research program. The decision was therefore made during the early stages of the doctoral research to switch to 4diac (Strasser et al., 2008). 4diac, coupled with its IEC 61499-compliant runtime framework FORTE, offers a rich, Open-Source development environment that is supported by a vibrant research community (4diac, 2019). Despite the limitations of nxtCONTROL, the Master's work successfully addressed gaps related to requirements traceability for IEC 61499 which led to novel contributions, including the creation of the TORUS (Traceability of Requirements using Splices) framework (Sinha et al., 2018). TORUS was used and extended further in the context of the current research beginning in Section 9.6.1, where it helped to facilitate the traceability that became a key component of the V-Model methodology employed for building the example ICPSs.

The Introduction chapter quoted Richard Feynman's encouragement to "*know how to solve every problem that has been solved*". Eichenlaub explains what Feynman meant by stressing the need to be able to solve problems from first principles for yourself that have already been solved by others (Eichenlaub, 2018). Feynman believed that, starting with a blank piece of paper and the knowledge he already had, he could take any existing theoretical result in his field and re-derive it himself. That practice provides a way of working through and acquiring the core, foundational knowledge that is essential before a researcher can identify and make novel

contributions in that field. This need was addressed via a carefully planned and constrained Design Science phases provided a working framework to enable the building of prototypes that led to a much deeper understanding of IEC 61499. The knowledge and experience gained became vital, especially during the design and creation of the complex diagnostic points. These have to operate unobtrusively within the IEC 61499 architecture without causing unwanted side-effects. In a similar way, crafting agents from first principles, supported by the GORITE Multi-Agent framework, provided a deep understanding of agents, their capabilities, and how they might be used to find faults. Without these opportunities for practical experimentation, it would have been difficult to confidently deliver the depth of functionality and empirical analysis presented in Chapters 9 and 10.

The Software Architecture document produced in the phase immediately after the Literature Review provided concrete evaluation criteria which helped to focus, constrain, and continuously evaluate each of the Design Science phases where the engine, the agents, and the diagnostic points were crafted. The research methodology design detailed in Chapter 3 highlighted concerns voiced by Easterbrook and Singer (2008) about the dangers of drifting while building such artefacts. Design Science must not degenerate into creating code for the sake of coding while losing sight of the purpose of crafting prototypes. The architecture-led approach ensured that it was clear at all stages whether or not the artefact being constructed was meeting the quality requirements defined for it. That led to a more measured, iterative approach to re-working features until the component or subsystem fitted properly into the engine architecture and performed in ways that were able to be evaluated empirically. That practice often had the effect of short-circuiting approaches far more quickly that would not lead to desirable outcomes.

Nevertheless, there were clearly times where it seemed like the development in a phase was taking too long or had lost a degree of focus. On reflection, that was more often an issue of depth, where more was being attempted than was actually needed. It took time to establish a balanced research perspective and, in the words of the primary supervisor, "*to own my own research*". The evaluation criteria were carefully crafted and refined during the long year that was required to document the architecture presented in Chapter 4 and in Appendix A. It also

took time and practice to apply the criteria during the ATAM evaluations. An iteration of an architecture is never complete until it has begun to demonstrably satisfy at least some of its stated aims; iteration takes time. Many times, it seemed as if some deep inner pragmatic voice was echoing the cry of Pope Julius II to Michaelangelo "*When will you make an end?*" during the painting of the Sistine Chapel in Rome. I came to understand Michelangelo's reply to the frustrated Pope: "*It will be finished when it is done...*" (Reed, 1965).

The order of the phases presented in Table 3.2 accurately reflects the overall flow of the work over the four and a half years of the doctoral program. However, Design Science also provided opportunities to jump between phases to address specific questions. The first agent was crafted working alongside the creator of GORITE, Associate Professor Dennis Jarvis, during the research posting at the Luleå Institute of Technology long before the literature review was complete. Early-stage prototypes of the HVAC system components were created while developing the diagnostic points before the work detailed in Chapter 9 used them to illustrate the V-Model processes. This suggests a high degree of Internal Validity in this research since RQ3, which asks to what extent Belief-Desire-Intention multi-agent paradigms can be applied to fault-identification have been clearly demonstrated. The experimental chapters present techniques for monitoring and diagnosing faults using multiple semi-autonomous yet co-operating agents. Further, their ability to capture accurate timing asynchronously demonstrates that they can bridge the temporal divide that is inherent in an ICPS discussed by Lee and Seshia (2016).

A number of things could have been done differently. Situating the research within an organisation that was using IEC 61499 to develop their production systems could have provided more relevant example systems to profile. The evaluations after trialling the engine with their engineers would have been valuable and contributed towards reaching higher TRL levels. Unfortunately, it was not possible to identify local organisations who were using IEC 61499. The architecture phase could also have been split into a shorter initial phase that was followed by an initial build and evaluation phase before the architecture was refined and documented in detail. *Views and Beyond* (2013) does not address prototyping explicitly, but Nord et al. (2009) propose that it is often difficult or impossible to perform ATAM evaluations on large-scale

system architectures with architectural descriptions alone. They suggest that some aspects may be more readily addressed via a prototype.

The Design Science program was established for this research to first propose adaptions to a Model-Driven Development with Diagnostics methodology, and then to refine it. The program included the delivery of a fully-functional multi-agent fault diagnostic engine, building on the GORITE framework that was evaluated in the early conceptual stages of this research. The V-Model outlined in Chapter 9 was founded on a well-established protocol that addresses many of the limitations of the traditional Waterfall (Bröhl & Dröschel, 1995; Anitha et al., 2013). The Agile techniques proposed by Graessler and Hentze (2020) offer a way to capitalise on the advantages that Sprint protocols offer while retaining the gate-driven phased approaches demanded by ICPS development. Provan's (2014) concerns about the failure to integrate diagnostic modelling early enough in the requirements process were addressed first by creating well-formed requirements in Phases 1 and 2 of the V-Model. Fault diagnostics was incorporated into the Phase 4 Component Design phase, building on the Risk Analysis that was performed in Phase 3. This approach worked well since function blocks are discrete objects that feature high internal cohesion and low external coupling. Each function block or composite function block provides a separate unit of functionality. Diagnostic tests can therefore be created for each of these entities independent of other blocks that may already exist or will be developed later. This divide-and-conquer approach is highly advantageous since it facilitates incremental development that can be tracked easily within the V-Model phases. Transition criteria between the phases are easy to define and follow. It is also possible to implement sprints of design and development, as long as they remain bounded within a single V-Model phase. The V-Model is also symmetrical, driving verification to ensure the requirement and design specifications are adhered to on the left-hand arm with component and system validation on the complementary right-hand arm. The methodology was refined to split some of the traditional phases into smaller units to better accommodate the component development, verification and development. A field-evaluation phase was also introduced as Phase 8 to complement the previous integration phases.

Simulation also played a key role in the V-Model. Large scale ICPS often require hardware components with long lead-times, especially in the aerospace and automotive domains. Hence, making simulation an option during the Phase 4 Component development and the Phase 5 Development phase ensures that the component designs are verified early and that diagnostic resources can be developed and tested early. Finally, creating diagnostic resources that are applicable right across the life-cycle of the product is a strong incentive to adopt Model-Based approaches to ICPS development. These have the potential to reduce costs by addressing problems early in the development cycle.

11.2 Considering Technology Readiness Level 5

Before an engineer will consider adopting a re-configurable tool into their software arsenal, it must satisfy two key criteria. First, the underlying functionality it provides them with must be robust and dependable, while being both simple to learn and easy to use. That usually implies that they have already recognised that this technology will fill a gap in their tool set. Secondly, the interface that allows them to configure and interact with the tool must be intuitive. It is important to consider the current version of the engine, the agents, and the Model-Driven Development with Diagnostics methodology from both of these user perspectives. This also provided another way to address RQ5, which asked how the diagnostic approaches provided by the engine could be evaluated.

When the evaluation criteria for this tool set was being discussed prior to the final evaluation commencing, it became clear that the NASA Technology Readiness Level 5 was an ideal benchmark level to consider. This level identifies components and technologies that have been validated in a relevant production environment beyond the laboratory trials that characterise TRL 4. Technologies that reach TRL 5 are half-way up the scale that leads to TRL 9, where technologies are deemed to be flight-proven and can be deployed in mission-critical space environments. However, the TRL scale is not linear. Terrile et al. (2015) note that the steepest steps are in the TRL range 6 to 8. This granularity distinguishes more clearly between artefacts

that are purely theoretical and those that are able to perform well in live environments.

On the first criteria, the engine technology the experiments discussed in Chapters 9 and 10 was shown to be comprehensive and robust. The way the agents operate and interact with a function block application mimics many of the ways a team of humans might hunt for faults by gathering evidence and then considering its implications. The concept of diagnostic points mimics technologies such as hardware Built-In-Test-Points that engineers are familiar with (Hale & Bollas, 2018). The diagnostic packages allow function block operations to be considered individually, assigning likely data capture test points in an intuitive way during design. The belief structures provide a way to reliably capture discrete fault information while the agents monitor and observe a running function block application, often over extended periods of time. The agents do not require extensive supervision once their tasks are defined and assigned. This creation of dynamic models by the agents addresses some of the barriers inherent in Model-Driven Design. Further, these diagnostic resources can be combined together as the application is built as well as being shared in repositories for later re-use. The concept of adding a `TI` traceability identifier to each function block instance also addresses long-standing needs for requirements traceability that is both intuitive and is easy to use within all of the current IEC 61499 IDEs. Using this criteria, it can be argued that both the engine technologies and the V-Model methodology were demonstrated to make a potent combination that should work equally well in the production environments mandated by TRL 5.

However, by considering the second criteria of ease-of-use, the engine technologies cannot be deemed to have reached TRL 5. How the agents currently present diagnostic information back to the engineers is still reliant on command-line displays and console dumps. Consider how in teams of humans, both verbal discussions and graphic diagrams play a significant role in their shared deliberations. Paradigms for communicating easily with agents and facilitating human-agent collaboration remain a challenge that has not been addressed in the present work. Therein lie the most intriguing avenues for future research.

The 4diac IDE was shown to provide an ideal modelling environment in Chapter 9. It provides a number of interfaces that visualise function block applications and their interactions

that were more appropriate than alternative UML or SysML diagrams. Hence it is likely that many of the human-to-agent interactions might be facilitated by replicating some of the functionality these interfaces provide to create a GUI for the engine. Figure 11.2 extends a diagram from Chapter 10 to propose a potential GUI that shows the function block application, the agents, and the diagnostic points they are marshalling in-context.

At present, the Diagnostic Package (.dpg) XML-format files are created manually. However, the GUI points to a way to implement a point-and-click method for creating diagnostic points interactively. Given that the engine is able to work outside the 4diac IDE, it would be able to dynamically refresh its System-Under-Diagnosis belief structure to maintain the model of the function block application. This would offer instant validation of the existence of events and ports that can be selected to attach diagnostic points to. Developing the diagnostic script language further would lead to a script editor that was hosted inside the same GUI to allow the steps of a diagnostic task to be specified. Finally, the definition of agents and their goals should

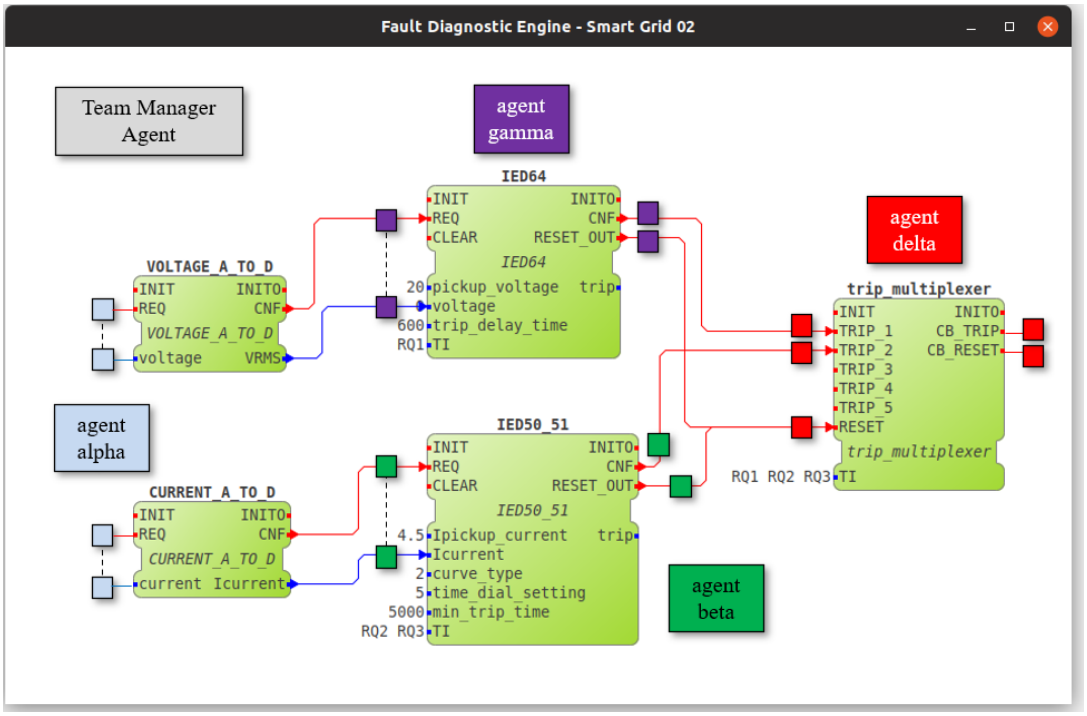

Figure 11.2: Potential function block application GUI interfaces for the engine and agents.

be performed in-context with the application while it is visible. At the same time, making the TORUS traceability functionality available in a complementary tool to enable requirements to be specified and managed along with quality attributes would help to foster and support the adoption of the V-Model approach extensions.

This leads naturally to what is likely to be the largest scope of future research into the application of multi-agent systems for the engine. This is concerned with the creation of *Pragmatic Agents.* Given that the agents have a well-defined set of three belief structures, how could they begin to consider their Dynamic Diagnostic Beliefs captured while fault-finding in intelligent ways? One would expect a human team member to form opinions about the evidence presented to them; what is the potential and what are the limits of the agents' abilities to form a meaningful diagnosis, adopt alternative strategies, and sift through large amounts of fault evidence. Might they also point to ways of not only detecting faults but also predicting them before they happen? How could we blend machine learning into the dynamic Model-Driven approaches investigated in this research to apply the best of these techniques in a hybrid solution? The agent actions performed within the engine do not at present address safety-critical aspects of fault diagnosis. Gating a section of a function block application that is controlling a critical transducer may not always be a wise course of action while fault finding. Hence one area that an investigation of pragmatic agents should address is how they might become risk-aware and consider the implications of some of their actions more carefully. The current GORITE team paradigm that was implemented relies on the Team Manager to co-ordinate the high-level actions of agents. However, it is reasonable for agents to co-operate with each other to complete shared goals together, especially if they are operating across distributed function block application boundaries. This would require inter-agent communication, an Interaction Skill that is supported within the current implementation.

11.3 Final thoughts

By the time the *Final Thoughts* section of the Master's thesis was being written during July 2016, there was a clear feeling that the research was in some way, complete. However, this Doctoral journey is ending on a different note. In so many ways, this research feels like it has reached a fulfilling way-point rather than being finished. The previous section summarised many hours of thinking, both alone and in discussion with my supervisors, that explored where these artefacts might go next. It was intriguing to see the requirements traceability functionality that was developed for TORUS during that prior research re-emerge and find new purpose here as part of the Model-Driven Development with Diagnostics methodology.

The words of Richard Feynman have been inspirational throughout this research. However, the writings of Winston Churchill have also provided many opportunities for reflection late at night during this journey. Perhaps this observation from one of his wartime speeches sums up these feelings in a more succinct way. It was written when the war and his part in it had reached a significant way-point:

"This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."

"*The Unrelenting Struggle*", Winston Churchill (1942)

References

4diac. (2019). *4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment* [Internet web page]. PROFACTOR Produktionsforschungs GmbH. Retrieved 28 January, 2019, from `https://www.eclipse.org/4diac/`

ABB. (2019). Generator Protection Version 1.1 ANSI REG650 Application Manual.

Abel, A., Adir, A., Blochwitz, T., Greenberg, L. & Salman, T. (2013). Development and verification of complex hybrid systems using synthesizable monitors. In *Haifa verification conference* (pp. 182–198). doi: 10.1007/978-3-319-03077-7_13

Agarwal, A. & Lang, J. (2005). *Foundations of analog and digital electronic circuits*. Elsevier.

Agency, D. A. R. P. (1981). *RFC 793 Transmission Control Protocol.* DARPA.

Ahmad, M. O. (2019). Agile Methods and Cyber-Physical Systems Development?A Review with Preliminary Analysis. In *International conference on big data and security* (pp. 274–285). doi: 10.1007/978-981-15-7530-3_20

Angeslevä, J., Bähr, B., Beckmann-Dobrev, B., Eichmann, U., Exner, K., Gengnagel, C., ... Stark, R. (2016). The results of rethinking prototyping. In *Rethink! prototyping* (pp. 201–210). Springer. doi: 10.1007/978-3-319-24439-6_11

Anitha, P., Savio, D. & Mani, V. (2013). Managing requirements volatility while "Scrumming" within the V-Model. In *2013 3rd international workshop on empirical requirements engineering (empire)* (pp. 17–23). doi: 0.1109/EmpiRE.2013.6615211

Antman, E., Lau, J., Kupelnick, B. et al. (1992). A comparison of results of meta-analyses of RCTs and recommendations of clinical experts. Treatments for myocardial infarction. *Journal of the American Medical Association*, *268*, 240–248.

Antonova, G., Nardi, M., Scott, A. & Pesin, M. (2012). Distributed generation and its impact on power grids and microgrids protection. In *2012 65th annual conference for protective relay engineers* (pp. 152–161).

AOSONG. (2020). *AM2302/DHT22 Temperature and Humidity Sensor Application Note* [Internet web page]. Aosong (Guangzhou) Electronics Co.,Ltd. Retrieved 13 July 2020, from `http://akizukidenshi.com/download/ds/aosong/AM2302.pdf`

Applegate, L. M. & King, J. (1999). Rigor and Relevance in MIS Research–Introduction. *MIS Quarterly*, *23*(1), 1–2.

Arksey, H. & O'Malley, L. (2005). Scoping studies: towards a methodological

framework. *International journal of social research methodology*, *8*(1), 19–32. doi: 10.1080/1364557032000119616

Arpinen, T., Hämäläinen, T. & Hännikäinen, M. (2011). Meta-model and UML profile for requirements management of software and embedded systems. *EURASIP Journal on Embedded Systems*, *2011*(1), 1. doi: 10.1155/2011/592168

ASHRAE. (2004). ASHRAE Standard 55-2004. *Thermal environmental conditions for human occupancy*, *3*.

Avison, D. E., Lau, F., Myers, M. D. & Nielsen, P. A. (1999). Action Research. *Communications of the ACM*, *42*(1), 94–97.

Awad, H. & Bollen, M. H. (2003). Power electronics for power quality improvements. In *2003 ieee international symposium on industrial electronics (cat. no. 03th8692)* (Vol. 2, pp. 1129–1136). doi: 10.1109/ISIE.2003.1267983

Azam, M., Pattipati, K., Allanach, J., Poll, S. & Patterson-Hine, A. (2005). In-flight fault detection and isolation in aircraft flight control systems. In *2005 ieee aerospace conference* (pp. 3555–3565). doi: 10.1109/AERO.2005.1559659

Baheti, R. & Gill, H. (2011). Cyber-physical systems. *The impact of control technology*, *12*(1), 161–166.

Bajracharya, M., Maimone, M. W. & Helmick, D. (2008). Autonomy for Mars Rovers: Past, Present, and Future. *Computer*, *41*(12). doi: 10.1109/MC.2008.479

Banerjee, T. P. & Das, S. (2013). Intelligent Fault Tracking by an Adaptive Fuzzy Predictor and a Fractional Controller of Electromechanical System–A Hybrid Approach. In *International conference on swarm, evolutionary, and memetic computing* (pp. 574–582). doi: 10.1007/978-3-319-03756-1_51

Barbacci, Ellison, Lattanze, Stafford & Weinstock. (2003). *Quality Attribute Workshops (QAWA)* (Tech. Rep.). Carnegie-Mellon University, Pittsburgh.

Barcelos & Travassos, G. H. (2006). Evaluation Approaches for Software Architectural Documents: a Systematic Review. In *Cibse* (pp. 433–446).

Basha, N. M. J., Moiz, S. A. & Rizwanullah, M. (2012). Model based software development: issues & challenges. *Special Issue of International Journal of Computer Science & Informatics (IJCSI), ISSN (PRINT)*, *2*(1), 2.

Bass, L., Clements, P. & Kazman, R. (2013). *Software Architecture in Practice, Third Edition*. Addison-Wesley.

Benowitz, E. (2014). The Curiosity Mars Rover's Fault Protection Engine. In *2014 IEEE International Conference on Space Mission Challenges for Information Technology* (pp. 62–66). doi: 10.1109/SMC-IT.2014.16

Bertolini, M., Mezzogori, D., Neroni, M. & Zammori, F. (2021). Machine Learning for industrial applications: a comprehensive literature review. *Expert Systems with Applications*, 114820. doi: 10.1016/j.eswa.2021.114820

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, *5*(2), 21–24.

Bizjak, G. & Baruti, G. (2017). Overcurrent protection. *Industrial Power System Protection*, 1–20.

Black, G. & Vyatkin, V. (2010). Intelligent component-based automation of baggage handling systems with IEC 61499. *Automation Science and Engineering, IEEE*

Transactions on, *7*(2), 337–351.

Bo, Z., Lin, X., Wang, Q., Yi, Y. & Zhou, F. (2016). Developments of power system protection and control. *Protection and Control of Modern Power Systems*, *1*(1), 1–8. doi: 10.1186/s41601-016-0012-2

Boehm, B. W., Brown, J. R. & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on software engineering* (pp. 592–605).

Boeing. (1998). *Space Shuttle Main Engine Orientation* [Internet web page]. Rocketdyne Propulsion and Power Group. Retrieved 25 May 2021, from `http://www.lpre.de/p_and_w/SSME/SSME_PRESENTATION.pdf`

Bolbot, V., Theotokatos, G., Bujorianu, M. L., Boulougouris, E. & Vassalos, D. (2018). Vulnerabilities and safety assurance methods in Cyber-Physical Systems: A comprehensive review. *Reliability Engineering & System Safety*.

Boules, N., Douglas, K., Feldman, S., Fix, L., Hager, G., Hailpern, B., … others (2016). The future of computing research: industry-academic collaborations. *arXiv preprint arXiv:1606.09236*. doi: arXiv:1606.09236v1

Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D. & Uchitel, S. (2015). Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering* (pp. 9–16). doi: 10.1145/2804337.2804339

Bradatsch, C., Ungerer, T., Zalman, R. & Lajtkep, A. (2011). Towards runtime testing in automotive embedded systems. In *Industrial embedded systems (sies), 2011 6th ieee international symposium on* (pp. 55–58). doi: 10.1109/SIES.2011.5953679

Brand, K., Brunner, C. & Wimmer, W. (2011). Design of IEC 61850 based substation automation systems according to Customer requirements. *Indian Journal of Power and River Valley Development*, *61*(5), 87.

Bratman. (1987). *Intention, plans, and practical reason* (Vol. 10). Harvard University Press Cambridge, MA. doi: 10.2307/2185304

Bratman, M. E., Israel, D. J. & Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational intelligence*, *4*(3), 349–355. doi: 10.1111/j.1467-8640.1988.tb00284.x

Braun, P., Broy, M., Houdek, F., Kirchmayr, M., Müller, M., Penzenstadler, B., … Weyer, T. (2014). Guiding requirements engineering for software-intensive embedded systems in the automotive industry. *Computer Science-Research and Development*, *29*(1), 21–43. doi: 10.1007/s00450-010-0136-y

Bregon, A., Daigle, M., Roychoudhury, I., Biswas, G., Koutsoukos, X. & Pulido, B. (2014). An event-based distributed diagnosis framework using structural model decomposition. *Artificial Intelligence*, *210*, 1–35.

Brinberg, D. & McGrath, J. E. (1983). A Validity Network Schema.

Bröhl, A.-P. & Dröschel, W. (1995). *Einführung in das V-Modell*. na.

Brooks, F. P. (1975). *The Mythical Man-Month* (Vol. 1995). Addison-Wesley Reading, MA.

Brooks, F. P. (1987). No silver bullet: essence and accidents in software engineering. *IEEE Computer*, *20*, 10–19.

Brooks Jr, F. P. (1996). The computer scientist as toolsmith ii. *Communications of the ACM*, *39*(3), 61–68. doi: 10.1145/227234.227243

Brosgol, B. (2011). DO-178C: The next avionics safety standard. *ACM SIGAda Ada Letters*, *31*(3), 5–6. doi: 10.1145/2070336.2070341

Brown, A. W. & McDermid, J. A. (2007). The Art and Science of Software Architecture. In *European conference on software architecture* (pp. 237–256). doi: 10.1007/978-3-540-75132-8_19

Broy, M., Kruger, I. H., Pretschner, A. & Salzmann, C. (2007). Engineering Automotive Software. *Proceedings of the IEEE*, *95*(2), 356–373.

Buchanan, R. (1992). Wicked problems in design thinking. *Design issues*, *8*(2), 5–21.

Cabasino, M. P., Seatzu, C., Mahulea, C. & Silva, M. (2010). Fault diagnosis of manufacturing systems using continuous Petri nets. In *2010 ieee international conference on systems, man and cybernetics* (pp. 534–539). doi: 10.1109/ICSMC.2010.5642021

Cabrera, A. A., Foeken, M., Tekin, O., Woestenenk, K., Erden, M., De Schutter, B., … Tomiyama, T. (2010). Towards automation of control software: A review of challenges in mechatronic design. *Mechatronics*, *20*(8), 876–886. doi: 10.1016/j.mechatronics.2010.05.003

Cacchione, P. Z. (2016). *The evolving methodology of scoping reviews.* SAGE Publications Sage CA: Los Angeles, CA. doi: 10.1177/1054773816637493

Calegari, R., Ciatto, G., Mascardi, V. & Omicini, A. (2021). Logic-based technologies for multi-agent systems: a systematic literature review. *Autonomous Agents and Multi-Agent Systems*, *35*(1), 1–67. doi: 10.1007/s10458-020-09478-3

Calvaresi, D., Marinoni, M., Sturm, A., Schumacher, M. & Buttazzo, G. (2017). The challenge of real-time multi-agent systems for enabling IoT and CPS. In *Proceedings of the international conference on web intelligence* (pp. 356–364). doi: 10.1145/3106426.3106518

Campbell, D. T. (1957). Factors relevant to the validity of experiments in social settings. *Psychological bulletin*, *54*(4), 297.

Card, S. K., Newell, A. & Moran, T. P. (1983). The Psychology of Human-Computer Interaction.

Carden, F., Jedlicka, R. P. & Henry, R. (2002). *Telemetry systems engineering*. Artech House.

Carvallo, A. & Cooper, J. (2015). *The advanced smart grid: Edge power driving sustainability*. Artech House.

Castleberry, A. & Nolen, A. (2018). Thematic analysis of qualitative research data: Is it as easy as it sounds? *Currents in Pharmacy Teaching and Learning*, *10*(6), 807–815. doi: 10.1016/j.cptl.2018.03.019

Charette, R. N. (2009). This car runs on code. *IEEE Spectrum*, *46*(3), 3.

Chen, L., Babar, M. A. & Nuseibeh, B. (2012). Characterizing architecturally significant requirements. *IEEE software*, *30*(2), 38–45. doi: 10.1109/MS.2012.174

Chen, Y. (2011). Applications of Bayesian network in fault diagnosis of braking system. In *Intelligent human-machine systems and cybernetics (ihmsc), 2011 international conference on* (Vol. 1, pp. 234–237). doi: 10.1109/IHMSC.2011.63

Chen, Z., Yang, Y. & Hu, Z. (2012). A technical framework and roadmap of embedded diagnostics and prognostics for complex mechanical systems in prognostics and health management systems. *IEEE Transactions on Reliability*, *61*(2), 314–322. doi: 10.1109/TR.2012.2196171

Christensen, J. H. (2017). Design Patterns, Frameworks, and Methodologies. *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, 27. doi: doi.org/10.1201/b19391

Cicchetti, A., Ciccozzi, F. & Pierantonio, A. (2019). Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233. doi: 10.1007/s10270-018-00713-w

Clements, P., Garlan, D., Little, R., Nord, R. & Stafford, J. (2003). Documenting software architectures: Views and Beyond. In *25th international conference on software engineering, 2003. proceedings.* (pp. 740–741).

Cook, T. D., Campbell, D. T. & Shadish, W. (2002). *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin Boston, MA.

Coughlan, R. (1954). Dr Edward Teller's magnificent obsession. *LIFE magazine*, 61–74.

Council, N. R. (2001). *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.* The National Academies Press. doi: 10.17226/10193

Cremona, F., Lohstroh, M., Broman, D., Lee, E. A., Masin, M. & Tripakis, S. (2019). Hybrid co-simulation: it's about time. *Software & Systems Modeling*, *18*(3), 1655–1679. doi: 10.1007/s10270-017-0633-6

Cruzes, D. S. & Dyba, T. (2011). Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement* (pp. 275–284). doi: 10.1109/ESEM.2011.36

DARPA. (1980). *RFC 768 User Datagram Protocol.* Author.

Dearden, R., Willeke, T., Simmons, R., Verma, V., Hutter, F. & Thrun, S. (2004). Real-time fault detection and situational awareness for rovers: Report on the mars technology program task. In *2004 ieee aerospace conference proceedings (ieee cat. no. 04th8720)* (Vol. 2, pp. 826–840). doi: 10.1109/AERO.2004.1367683

Delligatti, L. (2013). *SysML distilled: A brief guide to the systems modeling language*. Addison-Wesley.

Deloitte. (2020). 2020 Global Automotive Consumer Study. Retrieved 18 December, 2020, from `http://www.deloitte.com/autoconsumers`

Dennett, D. (2009). Intentional systems theory. *The Oxford Handbook of Philosophy of Mind*, 339–350.

Dillon, T., Chang, E., Singh, J. & Hussain, O. (2012). Semantics of cyber-physical systems. In *International conference on intelligent information processing* (pp. 3–12). doi: 10.1007/978-3-642-32891-6_3

Doberkat, E.-E. (2003). Pipelines: Modelling a software architecture through relations. *Acta Informatica*, *40*(1), 37–79. doi: 10.1007/s00236-003-0121-z

Dos Santos, P. S. M. & Travassos, G. H. (2011). Action research can swing the balance in experimental software engineering. *Advances in Computers*.

Dowdeswell. (2016). TORUS: Tracing complex requirements for large Cyber-Physical Systems. *Master's Thesis: AUT Research Elements*, 173. Retrieved from https://openrepository.aut.ac.nz/bitstream/handle/10292/10227/DowdeswellB.pdf;jsessionid=454787F2C7B37902BA696AC2D7092586?sequence=3

Dowdeswell, Sinha, Jarvis, D., Jarvis, J. & MacDonell. (2020). *Employing agent beliefs during fault diagnosis for iec 61499 industrial cyber-physical systems.* doi: 10.1109/IECON43393.2020.9254877

Dowdeswell, Sinha & MacDonell. (2020a). *Diagnosable-by-design model-driven development for iec 61499 industrial cyber-physical systems.* doi: 10.1109/IECON43393.2020.9254620

Dowdeswell, Sinha & MacDonell. (2020b). Finding faults: A scoping study of fault diagnostics for Industrial Cyber-Physical Systems. *Journal of Systems and Software*, *168*, 110638. doi: 0.1016/j.jss.2020.110638

Dowdeswell, B., Sinha, R. & Haemmerle, E. (2016). TORUS: Tracing Complex Requirements for Large Cyber-Physical Systems. In *Master's thesis from the auckland university of technology* (pp. 1–173).

Dowdeswell, B., Sinha, R. & MacDonell, S. G. (2021). Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems. In (Vol. 13, p. n). MDPI. Retrieved from https://www.mdpi.com/1999-5903/13/8/190 doi: 10.3390/fi13080190

Dragojević, M., Stević, S., Stupar, G. & Živkov, D. (2018). Utilizing iot technologies for remote diagnostics of next generation vehicles. In *2018 ieee 8th international conference on consumer electronics-berlin (icce-berlin)* (pp. 1–4). doi: 10.1109/ICCE-Berlin.2018.8576249

Dubinin, V. & Vyatkin, V. (2008). On definition of a formal model for IEC 61499 function blocks. *EURASIP Journal on Embedded Systems*, *2008*, 7.

Dwyer, M. B., Purandare, R. & Person, S. (2010). Runtime verification in context: Can optimizing error detection improve fault diagnosis? In *International conference on runtime verification* (pp. 36–50). doi: 10.1007/978-3-642-16612-9_4

EARTO. (2014). *The trl scale as a research and innovation policy tool, earto recommendations.* [Internet web page]. European Association of Research and Technology Organisations. Retrieved 27 July 2019, from https://www.earto.eu/wp-content/uploads/The_TRL_Scale_as_a_R_I_Policy_Tool_-_EARTO_Recommendations_-_Final.pdf

Easterbrook, S. (1999). Verification and validation of requirements for mission critical systems. In *Proceedings of the 21st international conference on software engineering* (pp. 673–674). doi: 10.1145/302405.302951

Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285–311). Springer.

Ebert, C. & Favaro, J. (2017). Automotive software. *IEEE Software*(3), 33–39. doi:

10.1109/MS.2017.82

Efkemann, C. & Hartmann, T. (2008). Specification of conditions for error diagnostics. *Electronic Notes in Theoretical Computer Science*, *217*, 97–112. doi: 10.1016/j.entcs.2008.06.044

Eichenlaub, M. (2018). What did Richard Feynman mean when he said, "What I cannot create, I do not understand"? *Quora*, 1. Retrieved from https://www.quora.com/What-did-Richard-Feynman-mean-when-he-said-What-I-cannot-create-I-do-not-understand

Eifert, T., Eisen, K., Maiwald, M. & Herwig, C. (2020). Current and future requirements to industrial analytical infrastructure - part 2: Smart Sensors. *Analytical and bioanalytical chemistry*, *412*(9), 2037–2045. doi: 10.1007/s00216-020-02421-1

ESA. (2009). *ESA Software Engineering and Standardization.* [Internet web page]. European Space Agency Software and Standards Group. Retrieved 11 July 2017, from http://http://www.esa.int/TEC/Software_engineering_and_standardisation/TECP5EUXBQE_0.html

Exner, K., Lindow, K., Stark, R., Ängeslevä, J., Bähr, B. & Nagy, E. (2015). A transdisciplinary perspective on prototyping. In *2015 ieee international conference on engineering, technology and innovation/international technology management conference (ice/itmc)* (pp. 1–8). doi: 10.1109/ICE.2015.7438659

Fang, H., Shi, H., Dong, Y., Fan, H. & Ren, S. (2017). Spacecraft power system fault diagnosis based on DNN. In *2017 prognostics and system health management conference (phm-harbin)* (pp. 1–5). doi: 10.1109/PHM.2017.8079271

Feiler, Gluch, D. & Mcgregor, J. (2016). An architecture-led safety analysis method. In *8th european congress on embedded real time software and systems (erts 2016), toulouse.*

Feiler, P. H., Lewis, B. A. & Vestal, S. (2006). The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In *2006 ieee conference on computer aided control system design, 2006 ieee international conference on control applications, 2006 ieee international symposium on intelligent control* (pp. 1206–1211). doi: 10.1109/CACSD-CCA-ISIC.2006.4776814

Feldman, S. P. (2000). Micro matters: The aesthetics of power in NASAâ€™s Flight Readiness Review. *The Journal of Applied Behavioral Science*, *36*(4), 474–490. doi: 10.1177/0021886300364005

Feng, Y., Zhou, Y. & Shen, Z. J. (2018). Sic solid state circuit breaker with an adjustable current-time tripping profile. In *2018 ieee applied power electronics conference and exposition (apec)* (p. 1968-1973). doi: 10.1109/APEC.2018.8341287

Ferrell, T. K. & Ferrell, U. D. (2017). RTCA DO-178C/EUROCAE ED-12C. *Digital Avionics Handbook*.

Feynman, R. (1964). The Character of Physical Law. The Messenger Lectures, 1964.

Feynman, R. (1988). *The final blackboard comments left by Richard Feynman* [Internet web page]. Caltech Image Archive - used with permission TN 4276. Retrieved 29 June, 2020, from http://archives-dc.library.caltech.edu/islandora/object/ct1%3A483

Findley, M., Kikuta, K. & Denly, M. (2021). External Validity. *Annual Review of Political Science*, *24*(1), 365-393. doi: 10.1146/annurev-polisci-041719-102556

Fleischmann, H., Kohl, J. & Franke, J. (2016). A Modular Architecture for the Design of Condition Monitoring Processes. *Procedia CIRP*, *57*, 410–415. doi: 10.1016/j.procir.2016.11.071

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Friedman, N. (1997). *Modeling beliefs in dynamic systems* (Unpublished doctoral dissertation). Citeseer.

Fritzson, P. & Engelson, V. (1998). Modelica - A unified object-oriented language for system modeling and simulation. In *European conference on object-oriented programming* (pp. 67–90). doi: 10.1007/BFb0054087

Fürst, S. & Bechter, M. (2016). AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In *2016 46th annual ieee/ifip international conference on dependable systems and networks workshop (dsn-w)* (pp. 215–217). doi: 10.1109/DSN-W.2016.24

Ganzha, M. & Jain, L. C. (2012). *Multiagent Systems and Applications: Volume 1: Practice and Experience* (Vol. 45). Springer Science & Business Media.

Ganzha, M., Jain, L. C., Jarvis, D., Jarvis, J. & Rönnquist, R. (2013). *Multiagent Systems and Applications*. Springer. doi: 10.1007/978-3-642-33323-1

Garcia, A., Silva, V., Chavez, C. & Lucena, C. (2002). Engineering multi-agent systems with aspects and patterns. *Journal of the Brazilian Computer Society*, *8*(1), 57–72.

García, M. V., Irisarri, E., Pérez, F., Estévez, E. & Marcos, M. (2017). An Open CPPS Automation Architecture based on IEC-61499 over OPC-UA for flexible manufacturing in Oil&Gas Industry. *IFAC-PapersOnLine*, *50*(1), 1231–1238. doi: 10.1016/j.ifacol.2017.08.347

Garousi, V., Petersen, K. & Ozkan, B. (2016). Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, *79*, 106–127. doi: 10.1016/j.infsof.2016.07.006

Gartner. (2019). *Gartner Top 10 Strategic Technology Trends 2019* [Internet web page]. Gartner, Inc. (US). Retrieved 19 June 2021, from `https://www.gartner.com/smarterwithgartner/gartner-top-10-strategic-technology-trends-for-2019`

Gelernter, D. (1999). *Machine Beauty: Elegance and the Heart of Computing (Repr ed)*. Basic Books.

Genesereth, M. R. & Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, *37*, 48–53.

Georgeff, Pell, B., Pollack, M., Tambe, M. & Wooldridge, M. (1998). The belief-desire-intention model of agency. In *International workshop on agent theories, architectures, and languages* (pp. 1–10). doi: 10.1007/3-540-49057-4_1

Georgeff, M. & Ingrand, F. (1989). Decision-making in an embedded reasoning system. In *International joint conference on artificial intelligence*.

Georgeff, M. P. & Ingrand, F. F. (1990). Monitoring and control of spacecraft systems using procedural reasoning. In *Nasa, lyndon b. johnson space center, third annual workshop on space operations automation and robotics (soar 1989)*.

Georgeff, M. P. & Lansky, A. L. (1987). Reactive reasoning and planning. In *Aaai* (Vol. 87, pp. 677–682).

Ghadhab, M., Kuntz, M., Kuvaiskii, D. & Fetzer, C. (2015). A controller safety concept based on software-implemented fault tolerance for fail-operational automotive applications. In *International workshop on formal techniques for safety-critical systems* (pp. 189–205). doi: 10.1007/978-3-319-29510-7_11

Gilchrist, A. (2016). *Industry 4.0: The Industrial Internet of Things*. Apress.

Gill, H. (2008). From vision to reality: cyber-physical systems. In *Hcss national workshop on new research directions for high confidence transportation cps: automotive, aviation, and rail*.

Giordano, V., Gangale, F., Fulli, G., Jiménez, M. S., Onyeji, I., Colta, A., ... others (2011). Smart Grid projects in Europe: lessons learned and current developments. *JRC Reference Reports, Publications Office of the European Union*.

Glatz, B., Schuster, H., Horauer, M., Rauscher, T. & Obermaisser, R. (2016). Fault injection for IEC 61499 applications. In *2016 ieee 21st international conference on emerging technologies and factory automation (etfa)* (pp. 1–3). doi: 10.1109/ETFA.2016.7733676

Goertz, G. & Mahoney, J. (2012). *A tale of two cultures*. Princeton University Press. doi: 10.1515/9781400845446

Goupil, P., Boada-Bauxell, J., Marcos, A., Cortet, E., Kerr, M. & Costa, H. (2014). AIRBUS efforts towards advanced real-time fault diagnosis and fault tolerant control. *IFAC Proceedings Volumes*, *47*(3), 3471–3476. doi: 10.3182/20140824-6-ZA-1003.01945

Graessler, I. & Hentze, J. (2020). The new V-Model of VDI 2206 and its validation. *at-Automatisierungstechnik*, *68*(5), 312–324. doi: 10.1515/auto-2020-0015

Green, S., Hurst, L. & Nangle. (1997). Software agents: A review.

Greenberg, L. S. (1991). Research on the process of change. *Psychotherapy research*, *1*(1), 3–16.

Gregor, S. & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS quarterly*, 337–355.

Gribbin, J. & Gribbin, M. (1998). Richard Feynman: a life in science.

Grimm, S., Watzke, M., Hubauer, T. & Cescolini, F. (2012). Embedded Reasoning on Programmable Logic Controllers. In *International semantic web conference* (pp. 66–81). doi: 10.1007/978-3-642-35173-0_5

Griss, M. L. (2001). Software agents as the next generation software components. *Component-based software engineering*, 641–657.

Guessi, M., Cavalcante, E. & Oliveira, L. B. (2015). Characterizing architecture description languages for software-intensive systems-of-systems. In *Proceedings of the third international workshop on software engineering for systems-of-systems* (pp. 12–18).

Haghighatkhah, A., Banijamali, A., Pakanen, O.-P., Oivo, M. & Kuvaja, P. (2017). Automotive software engineering: A systematic mapping study. *Journal of Systems and Software*, *128*, 25–55. doi: 10.1016/j.jss.2017.03.005

Hale, W. T. & Bollas, G. M. (2018). Design of Built-In Tests for Active Fault Detection and Isolation of Discrete Faults. *IEEE Access*, *6*, 50959–50973. doi: 10.1109/ACCESS.2018.2869269

Hall, E. C. (1972). Reliability history of the Apollo Guidance Computer.

Hametner, R., Hegny, I. & Zoitl, A. (2014). A unit-test framework for event-driven control components modeled in iec 61499. In *Proceedings of the 2014 ieee emerging technology and factory automation (etfa)* (pp. 1–8). doi: 10.1109/ETFA.2014.7005209

Hametner, R., Hegny, I. & Zoitl, A. (2017). Unit test framework for iec 61499 function blocks. *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*, 123.

Hamilton, M. & Zeldin, S. (1980). The relationship between design and verification. *Journal of Systems and Software*, *1*, 29–56.

Harirchi, F. & Ozay, N. (2018). Guaranteed model-based fault detection in cyber–physical systems: A model invalidation approach. *Automatica*, *93*, 476–488. doi: 10.1016/j.automatica.2018.03.040

Hayes, S., Hou, D. & Fischer, N. (2019). Understanding Ground Fault Detection Sensitivity and Ways to Mitigate Safety Hazards in Power Distribution Systems. In *46th annual western protective relay conference.*

Hazzan, O. (2002). The reflective practitioner perspective in software engineering education. *Journal of Systems and Software*, *63*(3), 161–171. doi: 10.1016/S0164-1212(02)00012-2

Héder, M. (2017). From NASA to EU: The evolution of the TRL scale in Public Sector Innovation. *The Innovation Journal*, *22*(2), 1–23.

Hedges, L. (2012). Data analysis validity. *Research Methods and Methodologies in Education*, 27.

Hegde, R., Mishra, G. & Gurumurthy, K. (2011). An insight into the hardware and software complexity of ECUs in vehicles. In *International conference on advances in computing and information technology* (pp. 99–106). doi: 10.1007/978-3-642-22555-0_11

Hegny, I., Hummer, O., Zoitl, A., Koppensteiner, G. & Merdan, M. (2008). Integrating software agents and IEC 61499 realtime control for reconfigurable distributed manufacturing systems. In *2008 international symposium on industrial embedded systems* (pp. 249–252). doi: 10.1109/SIES.2008.4577710

Hehenberger, P., Vogel-Heuser, B., Bradley, D., Eynard, B., Tomiyama, T. & Achiche, S. (2016). Design, modelling, simulation and integration of cyber physical systems: Methods and applications. *Computers in Industry*, *82*, 273–289. doi: 10.1016/j.compind.2016.05.006

Herbsleb, J. & Roberts, J. (2006). Collaboration in software engineering projects: A theory of coordination. *ICIS 2006 Proceedings*, 38.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105. doi: 10.2307/25148625

Hewitt, C. (1971). Procedural Embedding of knowledge in Planner. In *Ijcai* (pp. 167–182).

Hewitt, C. (2008). Development of Logic Programming: What went wrong, What was done about it, and What it might mean for the future. In *What went wrong and why: Lessons from ai research and applications: Papers from the 2008 aaai workshop.*

Hill, K., Menk, D. & Swiecki, B. (2016). *Just How High-Tech is the Automotive Industry?* [Internet web page]. Center for Automotive Research (CAR). Retrieved 11 July 2016, from `http://www.cargroup.org/?module=Publications&event=View&pubID=103#sthash.T5d2ik2u.dpuf`

Howden, N., Rönnquist, R., Hodgson, A. & Lucas, A. (2001). Jack intelligent agents-summary of an agent infrastructure. In *5th international conference on autonomous agents* (Vol. 142).

Huang, X.-F., Zhou, C.-J., Huang, S., Huang, K.-X. & Li, X. (2014). Transient fault detection in networked control systems. *International Journal of Distributed Sensor Networks*, *10*(11), 346269. doi: 10.1155/2014/346269

IBM. (2016). *Rational Rhapsody.* Inc.

IEC. (2003). Function blocks–Part 1: Programmable Controllers. General information. *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*, 61131–1.

IEC. (2006). Function blocks–Part 4: Rules for compliance profiles. *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*.

IEC. (2013a). Function blocks–Part 1: Architecture. *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*.

IEC. (2013b). Function blocks–Part 2: Software Tool Requirements. *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*.

IEC, I. (2001). 9126-1 (2001). Software Engineering Product Quality-Part 1: Quality Model. *International Organization for Standardization*.

IEEE. (1991). IEEE Standard 610 Glossary of Software Engineering Terminology. *Office*, *121990*(1), 1.

IEEE. (2008). Ieee standard electrical power system device function numbers, acronyms, and contact designations. *IEEE Std C37.2-2008 (Revision of IEEE Std C37.2-1996)*, 1-48. doi: 10.1109/IEEESTD.2008.4639522

Ingenieure, V. D. (2004). Design methodology for mechatronic systems. Retrieved 19 January 2021, from `https://www.techstreet.com/standards/vdi-2206?product_id=1163026`

Isa, S. S., Liem, A., Steinert, M. et al. (2015). The value of prototypes in the early design and development process. In *Ds 80-5 proceedings of the 20th international conference on engineering design (iced 15) vol 5: Design methods and tools-part 1, milan, italy, 27-30.07. 15* (pp. 235–242).

ISO. (2009). ISO 60255-151 (2009). Functional requirements for under/over current protection. *International Organization for Standardization*.

ISO. (2011a). *DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification*. Author.

ISO. (2011b, July). ISO/IEC 42010:2011, Systems and software engineering Recommended practice for architectural description of software-intensive systems. *ISO/IEC/IEEE 42010-2011*, 1-37.

ISO. (2019a). ISO 60255-181 (2019). Measuring relays and protection equipment: Functional requirements for frequency protection. *International Organization for Standardization*.

ISO. (2019b). *ISO Standard 19501:2005 Information Technology -The Object Management Group Systems Modeling Language (OMG SysML)*. Retrieved 28 February, 2019, from `https://www.iso.org/standard/65231.html`

ISO. (2019c). *ISO Standard 19501:2005 Information Technology - The Unified Modeling Language (OMG UML)*. Retrieved 28 February, 2019, from `https://www.iso.org/standard/32620.html`

ISO. (2021, February). ISO/IEC 61850:2021 Communication networks and systems for power utility automation. *ISO/IEC 61850:2021*, 1-64.

ISO/BSS. (2019). *Space systems. Definition of the Technology Readiness Levels (TRLs) and their criteria of assessment*. Author.

ISO/IEC. (2000). *(E)-Road Vehicles Diagnostic Systems Keyword Protocol 2000*. Author.

ISO/IEC. (2011a). *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. Author.

ISO/IEC. (2011b). *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. Author.

Iverson, D. L., Martin, R., Schwabacher, M., Spirkovska, L., Taylor, W., Mackey, R., ... Baskaran, V. (2012). General purpose data-driven monitoring for space operations. *Journal of Aerospace Computing, Information, and Communication*, *9*(2), 26–44. doi: 10.2514/1.54964

Jamro, M. (2014). SysML modeling of POU-oriented unit tests for IEC 61131-3 control software. In *Methods and models in automation and robotics (mmar), 2014 19th international conference on* (pp. 82–87).

Janasak, K. M. & Beshears, R. R. (2007). Diagnostics to Prognostics-A product availability technology evolution. In *Reliability and maintainability symposium, 2007. rams'07. annual* (pp. 113–118). doi: 10.1109/RAMS.2007.328051

Jarvis, D., Jarvis, J., Kalachev, A., Zhabelova, G. & Vyatkin, V. (2018). PROSA/G: An architecture for agent-based manufacturing execution. In *2018 ieee 23rd international conference on emerging technologies and factory automation (etfa)* (Vol. 1, pp. 155–160). doi: 10.1109/ETFA.2018.8502598

Jarvis, D., Jarvis, J., Rönnquist, R. & Jain, L. C. (2013). Multi-Agent Systems. In *Multiagent systems and applications* (pp. 1–12). Springer. doi: 10.1007/

978-3-642-33320-0_1

Jazdi, N. (2014). Cyber physical systems in the context of Industry 4.0. In *Automation, quality and testing, robotics, 2014 ieee international conference on* (pp. 1–4). doi: 10.1109/AQTR.2014.6857843

Jones, R. M. & Wray, R. E. (2006). Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI magazine*, *27*(2), 57. doi: 10.1609/aimag.v27i2.1880

Kalachev, A., Zhabelova, G., Vyatkin, V., Jarvis, D. & Pang, C. (2018). Intelligent mechatronic system with decentralised control and multi-agent planning. In *Iecon 2018-44th annual conference of the ieee industrial electronics society* (pp. 3126–3133). doi: 10.1109/IECON.2018.8591390

Kane, A., Fuhrman, T. & Koopman, P. (2014). Monitor based oracles for cyber-physical system testing: Practical experience report. In *2014 44th annual ieee/ifip international conference on dependable systems and networks* (pp. 148–155). doi: 10.1109/DSN.2014.28

Kang, H. S., Lee, J. Y., Choi, S., Kim, H., Park, J. H., Son, J. Y., ... Do Noh, S. (2016). Smart manufacturing: Past research, present findings, and future directions. *International Journal of Precision Engineering and Manufacturing-Green Technology*, *3*(1), 111–128. doi: 10.1007/s40684-016-0015-5

Karnouskos, S., Leitao, P., Ribeiro, L. & Colombo, A. W. (2020). Industrial agents as a key enabler for realizing industrial cyber-physical systems: Multiagent systems entering Industry 4.0. *IEEE Industrial Electronics Magazine*, *14*(3), 18–32. doi: 10.1109/MIE.2019.2962225

Käßmeyer, M., Berndt, R., Bazan, P. & German, R. (2016). Product Line Fault Tree Analysis by Means of Multi-valued Decision Diagrams. In *International gi/itg conference on measurement, modelling, and evaluation of computing systems and dependability and fault tolerance* (pp. 122–136). doi: 10.1007/978-3-319-31559-1_11

Kazman, R., Klein, M. & Clements, P. (2000). *ATAM: Method for architecture evaluation* (Tech. Rep.). Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Keiner, W. L. (1990). A navy approach to integrated diagnostics. In *Ieee conference on systems readiness technology,'advancing mission accomplishment'*. (pp. 443–450).

Kernel. (2018). GPIO Sysfs Interface for Userspace. Retrieved 11 July, 2021, from `https://www.kernel.org/doc/html/v5.5/admin-guide/gpio/sysfs.html`

Ketter, W., Collins, J., Saar-Tsechansky, M. & Marom, O. (2018). Information systems for a smart electricity grid: Emerging challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, *9*(3), 1–22. doi: 10.1145/3230712

Khlif, M. & Shawky, M. (2008). Enhancing diagnosis ability for embedded electronic systems using co-modeling. In *Novel algorithms and techniques in telecommunications, automation and industrial electronics* (pp. 143–149). Springer.

Khoukhi, A. & Khalid, M. H. (2015). Hybrid computing techniques for fault detection and isolation, a review. *Computers & Electrical Engineering*, *43*, 17–32. doi: 10.1016/j.compeleceng.2014.12.015

Kidney, J. & Denzinger, J. (2006). Testing the limits of emergent behavior in MAS using learning of cooperative behavior. *Frontiers in Artificial Intelligence and Applications*, *141*, 260.

Kim, M. H., Lee, S. & Lee, K. C. (2011). A fuzzy predictive redundancy system for fault-tolerance of x-by-wire systems. *Microprocessors and Microsystems*, *35*(5), 453–461. doi: 10.1016/j.micpro.2011.04.003

Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O. P., Turner, M., Niazi, M. & Linkman, S. (2010). Systematic literature reviews in software engineering–a tertiary study. *Information and Software Technology*, *52*(8), 792–805. doi: 10.1016/j.infsof.2010.03.006

Klar, D. & Huhn, M. (2012). Interfaces and models for the diagnosis of cyber-physical ecosystems. In *Digital ecosystems technologies (dest), 2012 6th ieee international conference on* (pp. 1–6). doi: 10.1109/DEST.2012.6227948

Kochte, M. A. & Wunderlich, H.-J. (2017). Self-Test and Diagnosis for Self-Aware Systems. *IEEE Design & Test*, *35*(5), 7–18. doi: 10.1109/MDAT.2017.2762903

Koitz, R., Lüftenegger, J. & Wotawa, F. (2017). Model-based diagnosis in practice: interaction design of an integrated diagnosis application for industrial wind turbines. In *International conference on industrial, engineering and other applications of applied intelligent systems* (pp. 440–445). doi: 10.1007/978-3-319-60042-0_48

Kritzinger, W., Karner, M., Traar, G., Henjes, J. & Sihn, W. (2018). Digital Twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, *51*(11), 1016–1022. doi: 10.1016/j.ifacol.2018.08.474

Kruchten, P. B. (1995). The 4+1 View Model of Architecture. *IEEE software*, *12*(6), 42–50.

Kuechler, B. & Vaishnavi, V. (2008). On theory development in design science research: anatomy of a research project. *European Journal of Information Systems*, *17*(5), 489–504.

Kunst, N., Judkins, J., Lynn, C. & Goodman, D. (2009). Damage propagation analysis methodology for electromechanical actuator prognostics. In *2009 ieee aerospace conference* (pp. 1–7). doi: 10.1109/DSN.2014.28

Kuphaldt, T. (2019). Lessons in Industrial Instrumentation. , 3013.

Kurz, D., Kaspar, J. & Pilz, J. (2011). Dynamic maintenance in semiconductor manufacturing using Bayesian networks. In *Automation science and engineering (case), 2011 ieee conference on* (pp. 238–243). doi: 10.1109/CASE.2011.6042404

Laird, J. E., Newell, A. & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial intelligence*, *33*(1), 1–64.

Lander, A. (2019). Programmable Logic Controllers: The Evolution of a Disruptive Technology. Retrieved 13 July, 2020, from `https://new.engineering.com/eng/story/programmable-logic-controllers-the-evolution-of-a-disruptive-technology`

Langer, F., Eilers, D. & Knorr, R. (2009). Fault detection in discrete event based distributed systems by forecasting message sequences with neural networks. In *Annual conference on artificial intelligence* (pp. 411–418). doi: 10.1007/978-3-642-04617-9_52

Lapira, E. R., Bagheri, B., Zhao, W., Lee, J., Henriques, R. V., Pereira, C. E., … Guimarães, C. (2013). A systematic approach to intelligent maintenance of production systems with a framework for embedded implementation. *IFAC Proceedings Volumes*, *46*(7), 23–28. doi: 10.3182/20130522-3-BR-4036.00092

Laughton, M. A. & Say, M. G. (2013). *Electrical engineer's reference book*. Elsevier.

Lee & Seshia, S. A. (2016). *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press.

Lee, E. A. (2006). Cyber-physical systems-are computing foundations adequate. In *Position paper for nsf workshop on cyber-physical systems: research motivation, techniques and roadmap* (Vol. 2, pp. 1–9).

Lee, J., Bagheri, B. & Kao, H.-A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, *3*, 18–23. doi: 10.1016/j.mfglet.2014.12.001

Lee, J., Jin, C. & Bagheri, B. (2017). Cyber physical systems for predictive production systems. *Production Engineering*, *11*(2), 155–165.

Leitão, P. & Karnouskos, S. (2015). *Industrial agents: emerging applications of software agents in industry*. Morgan Kaufmann.

Lesko, C. R., Buchanan, A. L., Westreich, D., Edwards, J. K., Hudgens, M. G. & Cole, S. R. (2017). Generalizing study results: a potential outcomes perspective. *Epidemiology (Cambridge, Mass.)*, *28*(4), 553. doi: 10.1097/EDE.0000000000000664

Levac, D., Colquhoun, H. & O'Brien, K. K. (2010). Scoping studies: advancing the methodology. *Implementation science*, *5*(1), 69. doi: 10.1186/1748-5908-5-69

Lindgren, P., Lindner, M., Lindner, A., Vyatkin, V., Pereira, D. & Pinho, L. M. (2015). A real-time semantics for the IEC 61499 standard. In *Emerging technologies & factory automation (etfa), 2015 ieee 20th conference on* (pp. 1–6). doi: 10.1109/ETFA.2015.7301558

Lu, Y., Dong, Y., Wei, X. & Xiao, M. (2018). A hybrid method of redundancy system reliability analysis based on aadl models. In *2018 ieee international conference on software quality, reliability and security companion (qrs-c)* (pp. 294–300). doi: 10.1109/QRS-C.2018.00060

Mäder, P. & Egyed, A. (2015). Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, *20*(2), 413–441.

Mader, P., Gotel, O. & Philippow, I. (2008). Enabling automated traceability maintenance by recognizing development activities applied to models. In *2008 23rd ieee/acm international conference on automated software engineering* (pp. 49–58). doi: 10.1109/ASE.2008.15

Mankins, J. C. (1995). Technology readiness levels. *White Paper, April*, *6*.

Mankins, J. C. (2009). Technology readiness assessments: A retrospective. *Acta Astronautica*, *65*(9-10), 1216–1223. doi: 10.1016/j.actaastro.2009.03.058

Mansour, K., Farag, W. & ElHelw, M. (2012). AiroDiag: A sophisticated tool that diagnoses and updates vehicles software over air. In *Electric vehicle conference (ievc), 2012 ieee international* (pp. 1–7). doi: 10.1109/IEVC.2012.6183181

Marik, V., Gorodetsky, V. & Skobelev, P. (2020). Multi-agent technology for industrial applications: Barriers and trends. In *2020 ieee international conference on systems, man, and cybernetics (smc)* (p. 1980-1987). doi: 10.1109/SMC42975.2020.9283071

Marzat, J., Piet-Lahanier, H., Damongeot, F. & Walter, E. (2009). A new model-free method performing closed-loop fault diagnosis for an aeronautical system. In *7th workshop on advanced control and diagnosis, acd'2009* (p. 6). doi: 10.3182/20090706-3-FR-2004.00032

May, N. (2005). A survey of software architecture viewpoint models. In *Proceedings of the sixth australasian workshop on software and system architectures* (pp. 13–24).

Mays, N., Roberts, E., Popay, J. et al. (2001). Synthesising research evidence. *Studying the organisation and delivery of health services: Research methods*, *220*.

McCarthy, J., Minsky, M. & Rochester, N. (1955). A proposal for the dartmouth summer research project on artificial intelligence.

McGovern, S. M., Cohen, S. B., Truong, M. & Fairley, G. (2007). Kinematics-based model for stochastic simulation of aircraft operating in the national airspace system. In *2007 ieee/aiaa 26th digital avionics systems conference* (pp. 3–B). doi: 10.1109/DASC.2007.4391876

McGrath, J. E. & Brinberg, D. (1985). External validity and the research process: A comment on the calder/lynch dialogue. *Journal of Consumer Research*, *10*(1), 115–124.

McGregor, J. D., Gluch, D. P. & Feiler, P. H. (2017). Analysis and Design of Safety-critical, Cyber-Physical Systems. *ACM SIGAda Ada Letters*, *36*(2), 31–38. doi: 10.1145/3092893.3092899

McKinsey. (2014). What's Driving the connected car? Retrieved 29 June, 2016, from `http://www.mckinsey.com/industries/automotive-and-assembly/our-insights/whats-driving-the-connected-car`

Mellon, C. (2018). *Views and Beyond: The SEI Approach for Architecture Documentation* [Internet web page]. Carnegie Mellon University Software Engineering Institute. Retrieved 23 January, 2019, from `https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513862`

Merriam-Webster. (2021). Veracity dictonary definition. Retrieved 08 January 2021, from `https://www.merriam-webster.com/dictionary/veracity`

Metcalfe, J. S. & Miles, I. (1994). Standards, selection and variety: an evolutionary approach. *Information Economics and Policy*, *6*(3-4), 243–268.

Milis, G. M., Eliades, D. G., Panayiotou, C. G. & Polycarpou, M. M. (2016). A cognitive

fault-detection design architecture. In *Neural networks (ijcnn), 2016 international joint conference on* (pp. 2819–2826). doi: 10.1109/IJCNN.2016.7727555

MIMOSA. (2019). Machinery Information Management Open Systems Alliance (MIMOSA). Retrieved 29 June, 2019, from https://www.mimosa.org

Mitsubishi. (2021). Mitsubishi PAR-33MAA Technical Reference. Retrieved from https://www.mitsubishitechinfo.ca/sites/default/files/TM_PAR-33MAA_M-E0734-SIZ1803_201803.pdf

Modest, C. & Thielecke, F. (2016). SPYDER: a software package for system diagnosis engineering. *CEAS Aeronautical Journal*, *7*(2), 315–331. doi: doi.org/10.1007/s13272-016-0189-0

Mohrle, F., Zeller, M., Hofig, K., Rothfelder, M. & Liggesmeyer, P. (2015). Automated compositional safety analysis using component fault trees. In *Software reliability engineering workshops (issrew), 2015 ieee international symposium on* (pp. 152–159). doi: 10.1109/ISSREW.2015.7392061

Monostori, L. (2014). Cyber-physical production systems: Roots, expectations and R&D challenges. *Procedia Cirp*, *17*, 9–13. doi: 10.1016/j.procir.2014.03.115

Moore, E. F. et al. (1956). Gedanken-experiments on sequential machines. *Automata studies*, *34*, 129–153.

Mugan, J. (2014). The Curiosity Cycle.

Munn, Z., Peters, M. D., Stern, C., Tufanaru, C., McArthur, A. & Aromataris, E. (2018). Systematic review or scoping review? guidance for authors when choosing between a systematic or scoping review approach. *BMC medical research methodology*, *18*(1), 143. doi: 10.1186/s12874-018-0611-x

Murphy, B., Wakefield, A. & Friedman, J. (2008). *Best practices for verification, validation, and test in model-based design* (Tech. Rep.). SAE Technical Paper. doi: 10.4271/2008-01-1469

NASA. (1995). RX-25 Space Shuttle Gimbal Test. *NASA Technical Report*. Retrieved May, 2021, from http://dayton.hq.nasa.gov

Nasri, O., Lakhal, N. M. B., Adouane, L. & Slama, J. B. H. (2019). Automotive decentralized diagnosis based on can real-time analysis. *Journal of Systems Architecture*. doi: 10.1016/j.sysarc.2019.01.009

Neto, L., Madsen, A. L., Silva, R., Reis, J., McIntyre, P., Gonçalves, G. et al. (2018). A component framework as an enabler for industrial cyber physical systems. In *2018 ieee industrial cyber-physical systems (icps)* (pp. 339–344). doi: 10.1109/ICPHYS.2018.8387682

Newton, I. (1687). Philosophiæ naturalis principia mathematica (Mathematical principles of natural philosophy). *London (1687)*.

NOJA. (2015). NOJA Power Smart Grid Automation software simplifies enhanced network implementations. Retrieved 10 December 2020, from http://www.nojapower.com/press/2015/iec-61499-recloser-function-blocks-sga-increases-distribution-automation-flexibility-smart-grid.html

Nord, R. L., Clements, P. C., Emery, D. & Hilliard, R. (2009). Reviewing architecture documents using question sets. In *2009 joint working ieee/ifip conference on*

software architecture european conference on software architecture (p. 325-328). doi: 10.1109/WICSA.2009.5290679

Nord, R. L., McHale, J. & Bachmann, F. (2010). *Combining architecture-centric engineering with the team software process* (Tech. Rep.). Carnegie-Mellon University Pittsburg PA Software Engineering Institute.

Norman, D. A. (2013). *The design of everyday things: Revised and expanded edition*. Basic books.

Noy, N. F., McGuinness, D. L. et al. (2001). *Ontology development 101: A guide to creating your first ontology*. Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880.

Nuñez, D. L. & Borsato, M. (2017). An ontology-based model for prognostics and health management of machines. *Journal of Industrial Information Integration*, *6*, 33–46. doi: 10.1016/j.jii.2017.02.006

nxtControl GmbH. (2016). *The nxtStudio Development Environment.* Retrieved 8 January 2016, from `http://www.nxtcontrol.com/en`

OMG. (2020). What's Driving the connected car? Retrieved 31 March 2020, from `https://www.omg.org/omgmarte/`

O'Raghallaigh, P., Sammon, D. & Murphy, C. (2012). Using Focus Groups to Evaluate Artefacts in Design Research. In *Proceedings of the 6th European Conference on Information Management and Evaluation, edited by T. Nagle, University of Cork, Ireland* (pp. 251–257).

Osswald, S., Matz, S. & Lienkamp, M. (2014). Prototyping automotive cyber-physical systems. In *Proceedings of the 6th international conference on automotive user interfaces and interactive vehicular applications* (pp. 1–6).

Oster, C. V. & Strong, J. S. (2017). *Managing the Skies: Public Policy, Organization and Financing of Air Traffic Management*. Routledge. doi: 10.1002/pam.20418

Palmer, K. A., Hale, W. T. & Bollas, G. M. (2018). Active fault identification by optimization of test designs. *IEEE Transactions on Control Systems Technology*, *27*(6), 2484–2498. doi: 10.1109/TCST.2018.2867996

Parr, E. A. (1998). *Industrial control handbook*. Industrial Press Inc.

Paz, A. & El Boussaidi, G. (2017). Landing Gear Control Software.

Pecheur, C., Simmons, R. & Engrand, P. (2006). Formal verification of autonomy models. In *Agent technology from a formal perspective* (pp. 311–339). Springer.

Pesce, M. (2011). Software Takes On More Tasks in Today's Cars. Retrieved 29 June, 2016, from `http://www.wired.com/2011/04/the-growing-role-of-software-in-our-cars`

Petersen, K., Feldt, R., Mujtaba, S. & Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. In *Ease* (Vol. 8, pp. 68–77).

Pons, R., Subias, A. & Travé-Massuyès, L. (2015). Iterative hybrid causal model based diagnosis: Application to automotive embedded functions. *Engineering Applications of Artificial Intelligence*, *37*, 319–335. doi: 10.1016/j.engappai.2014.09.016

Poslad, S. (2007). Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *2*(4), 15–es. doi: 10.1145/1293731.1293735

Pretschner, A., Broy, M., Kruger, I. H. & Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *Future of software engineering (fose'07)* (pp. 55–71). doi: 10.1109/FOSE.2007.22

Preuße, S., Missal, D., Gerber, C., Hirsch, M. & Hanisch, H.-M. (2010). On the use of model-based IEC 61499 controller design. *Int. J. of Discrete Event Control Systems (IJDESC)*.

Procter, S. & Feiler, P. (2020). The AADL Error Library: An Operationalized Taxonomy of System Errors. *ACM SIGAda Ada Letters*, *39*(1), 63–70. doi: 10.1145/3379106.3379113

Provan, G. (2014). A Contracts-Based Framework for Systems Modeling and Embedded Diagnostics. In *International conference on software engineering and formal methods* (pp. 131–143). doi: 10.1007/978-3-319-15201-1_9

Ragheb, M. (2019). Fault Tree Analysis and Alternative Configurations of Angle of Attack (AOA) Sensors as Part of Maneuvering Characteristics Augmentation System (MCAS). *mragheb. com*.

Ramos, A. V., Delamer, I. M. & Lastra, J. L. (2011). Embedded service oriented monitoring, diagnostics and control: Towards the asset-aware and self-recovery factory. In *Industrial informatics (indin), 2011 9th ieee international conference on* (pp. 497–502). doi: 10.1109/INDIN.2011.6034930

Rao, A. S., Georgeff, M. P. et al. (1995). BDI agents: from theory to practice. In *Icmas* (Vol. 95, pp. 312–319).

Reed, C. (1965). *The agony and the ecstasy.* Twentieth Century Fox Home Entertainment.

Reijonen, V., Koskinen, J. & Haikala, I. (2010). Experiences from scenario-based architecture evaluations with ATAM. In *European conference on software architecture* (pp. 214–229). doi: 10.1007/978-3-642-15114-9_17

Ribeiro, C. & Berry, D. (2020). The Prevalence and Severity of Persistent Ambiguity in Software Requirements Specifications: Is a Special Effort Needed to Find Them? *Science of Computer Programming*, 102472. doi: 10.1016/j.scico.2020.102472

Ribeiro, L. & Barata, J. (2011). Re-thinking diagnosis for future automation systems: An analysis of current diagnostic practices and their applicability in emerging IT based production paradigms. *Computers in Industry*, *62*(7), 639–659. doi: 10.1016/j.compind.2011.03.001

Richardson, D. J. & Wolf, A. L. (1996). Software testing at the architectural level. In *Joint proceedings of the second international software architecture workshop (isaw-2) and international workshop on multiple perspectives in software development (viewpoints' 96) on sigsoft'96 workshops* (pp. 68–71).

Rizzoni, G., Onori, S. & Rubagotti, M. (2009). Diagnosis and prognosis of automotive systems: motivations, history and some results. *IFAC Proceedings Volumes*, *42*(8), 191–202. doi: 10.3182/20090630-4-ES-2003.00032

Rodríguez, J. C. (2017). Design-based Research. *The Handbook of Technology and Second Language Teaching and Learning*, 364–377.

Rönnquist, R. (2007). The Goal Oriented Teams (GORITE) framework. In *International workshop on programming multi-agent systems* (pp. 27–41).

Roos, N., ten Teije, A., Bos, A. & Witteveen, C. (2002). An analysis of multi-agent diagnosis. In *Proceedings of the first international joint conference on autonomous agents and multiagent systems: part 2* (pp. 986–987).

Rozanski, N. & Woods, E. (2011). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

Rozanski, N. & Woods, E. (2019). *Viewpoints and Perspectives on-line resources* [Internet web page]. Carnegie Mellon University Software Engineering Institute. Retrieved 23 January, 2019, from `https://www.viewpoints-and-perspectives.info`

Rubey, R. J. & Hartwick, R. D. (1968). Quantitative measurement of program quality. In *Proceedings of the 1968 23rd acm national conference* (pp. 671–677). doi: 10.1145/800186.810631

Runeson, P. & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, *14*(2), 131–164.

Sabriye, A. o. J. & Zainon, W. M. N. W. (2018). An Approach for Detecting Syntax and Syntactical Ambiguity in Software Requirement Specification. *Journal of Theoretical & Applied Information Technology*, *96*(8).

Samad, T., Parisini, T. & Annaswamy, A. (2011). Systems of systems. *The Impact of Control Technology*, *12*(1), 175–183.

Sampigethaya, K. & Poovendran, R. (2013). Aviation cyber–physical systems: Foundations for future aircraft and air transport. *Proceedings of the IEEE*, *101*(8), 1834–1855. doi: 10.1109/JPROC.2012.2235131

Sankavaram, C., Kodali, A. & Pattipati, K. (2013). An integrated health management process for automotive cyber-physical systems. In *2013 international conference on computing, networking and communications (icnc)* (pp. 82–86). doi: 10.1109/ICCNC.2013.6504058

Saracco, R. (2016). Guess what requires 150 million lines of code. *EIT Digital. Internet. Available: https://www. eitdigital. eu/news-events/blog/article/guess-what-requires-150-million-lines-of-code/(13 Jan. 2016)*.

Sargolzaei, A., Crane, C. D., Abbaspour, A. & Noei, S. (2016). A machine learning approach for fault detection in vehicular cyber-physical systems. In *Machine learning and applications (icmla), 2016 15th ieee international conference on* (pp. 636–640).

Schoeller, M., Roemer, M., Leonard, M. & Derriso, M. (2007). Embedded reasoning supporting aerospace IVHM. In *Aiaa infotech@ aerospace 2007 conference and exhibit* (p. 2820). doi: 10.2514/6.2007-2820

Schulte, P. Z. (2018). *A State Machine Architecture for Aerospace Vehicle Fault Protection* (Unpublished doctoral dissertation). Georgia Institute of Technology.

Schut, M., Wooldridge, M. & Parsons, S. (2002). On partially observable MDPs

and BDI models. In *Foundations and applications of multi-agent systems* (pp. 243–259). Springer. doi: 10.1007/3-540-45634-1_15

Schwabacher, M., Aguilar, R. & Figueroa, F. (2009). Using decision trees to detect and isolate simulated leaks in the J-2X rocket engine. In *2009 ieee aerospace conference* (pp. 1–7). doi: 10.1109/AERO.2009.4839691

Schwabacher, M. & Goebel, K. (2007). A survey of artificial intelligence for prognostics. In *Aaai fall symposium* (pp. 107–114).

Schwabacher, M., Martin, R., Waterman, R., Oostdyk, R., Ossenfort, J. & Matthews, B. (2010). Ares IX ground diagnostic prototype. In *Aiaa infotech@ aerospace 2010* (p. 3354). doi: 10.2514/6.2010-3354

Schwabacher, M. & Waterman, R. (2008). Pre-launch diagnostics for launch vehicles. In *2008 ieee aerospace conference* (pp. 1–8).

Schweppe, H., Zimmermann, A. & Grill, D. (2009). Flexible on-board stream processing for automotive sensor data. *IEEE Transactions on Industrial Informatics*, *6*(1), 81–92. doi: 10.1109/TII.2009.2037145

Segal, S. (2017). A framework for removing ambiguity from software requirements. *IIOAB Journal*, *8*, 43–46.

Shah, A. A., Kerzhner, A. A., Schaefer, D. & Paredis, C. J. (2010). Multi-view modeling to support embedded systems engineering in SysML. In *Graph transformations and model-driven engineering* (pp. 580–601). Springer. doi: 10.1007/978-3-642-17322-6_25

Sharp, D. C., Bell, A. E., Gold, J. J., Gibbar, K. W., Gvillo, D. W., Knight, V. M., ... others (2010). Challenges and solutions for embedded and networked aerospace software systems. *Proceedings of the IEEE*, *98*(4), 621–634. doi: 10.1109/JPROC.2009.2039631

Shehory, O. M. (1998). *Architectural properties of multi-agent systems*. Carnegie Mellon University, The Robotics Institute.

Shi, W. & Dustdar, S. (2016). The Promise of Edge Computing. *Computer*, *49*(5), 78–81. doi: 10.1109/MC.2016.145

Shin, S. Y., Nejati, S., Sabetzadeh, M., Briand, L. C. & Zimmer, F. (2018). Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis* (pp. 49–60). doi: 10.1145/3213846.3213852

Shraim, H., Awada, A. & Youness, R. (2018). A survey on quadrotors: Configurations, modeling and identification, control, collision avoidance, fault diagnosis and tolerant control. *IEEE Aerospace and Electronic Systems Magazine*, *33*(7), 14–33. doi: 10.1109/MAES.2018.160246

Simon, H. A. (1996). *The Sciences of the Artificial 3rd ed.* MIT Press.

Simpson, W. R. & Sheppard, J. W. (1991). System complexity and integrated diagnostics. *IEEE Design & Test of Computers*, *8*(3), 16–30. doi: 10.1109/54.84239

Sinha, Dowdeswell, Zhabelova & Vyatkin. (2018). Torus: Scalable requirements

traceability for large-scale cyber-physical systems. *ACM Transactions on Cyber-Physical Systems*, *3*(2), 15.

Smith, J. & Lowenstein, D. (2009a). Built in Test-coverage and diagnostics. In *Autotestcon, 2009 ieee* (pp. 169–172). doi: 10.1109/AUTEST.2009.5314056

Smith, J. & Lowenstein, D. (2009b, Sept). Built in Test - coverage and diagnostics. In *2009 IEEE AUTOTESTCON* (p. 169-172). doi: 10.1109/AUTEST.2009.5314056

Song, Y. H., Kim, M. H., Lee, S. & Lee, K. C. (2010). Implementation of a fuzzy predictive redundancy system for tolerance of x-by-wire systems. In *Iecon 2010-36th annual conference on ieee industrial electronics society* (pp. 3141–3145). doi: 10.1109/IECON.2010.5675027

Sparx. (2020). *Sparx Systems Enterprise Architect.* Sparx Inc.

Std, I. (2001). IEEE recommended practice for protection and coordination of industrial and commercial power systems. *IEEE Std 242-2001, (Revision of IEEE Std 242-1986)*, 1-710. doi: 10.1109/IEEESTD.2001.93369

Steinkamp, N. (2018). 2018 Automotive Warranty and Recall Report. Retrieved 15 October, 2018, from `https://www.stoutadvisory.com/insights/report/2018-automotive-warranty-recall-report`

Steinkamp, N. (2020). 2020 Automotive Warranty and Recall Report. Retrieved 15 January, 2020, from `https://www.stout.com/en/insights/report/2020-automotive-defect-and-recall-report`

Steinkamp, N., Levine, R. & Roth, R. (2019). 2019 Automotive Defect and Recall Report. Retrieved 18 October, 2019, from `https://www.stout.com/en/insights/report/2019-automotive-defect-and-recall-report`

Stephenson, D. (2006). Boeing: The Airplane Doctors. Retrieved 2 March, 2018, from `http://www.boeing.com/news/frontiers/archive/2006/august/ts_sf09.pdf`

Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sünder, C., Valentini, A. & Martel, A. (2008). Framework for distributed industrial automation and control (4DIAC). In *Industrial informatics, 2008. indin 2008. 6th ieee international conference on* (pp. 283–288).

Straub, J. (2015). In search of technology readiness level (TRL) 10. *Aerospace Science and Technology*, *46*, 312–320.

Tanveer, A., Sinha, R. & MacDonell, S. (2018). On Design-time Security in IEC 61499 Systems: Conceptualisation, Implementation, and Feasibility. doi: 10.1109/INDIN.2018.8472093

Technavio. (2018). *Global commercial vehicle remote diagnostics market 2018-2022*. [Internet web page]. Author. Retrieved 27 July 2019, from `https://www.technavio.com`

Terrile, R. J., Doumani, F. G., Ho, G. Y. & Jackson, B. L. (2015). Calibrating the technology readiness level (TRL) scale using NASA mission data. In *2015 ieee aerospace conference* (pp. 1–9). doi: 10.1109/AERO.2015.7119313

Thombare, T. R. & Dole, L. (2014). Review on fault diagnosis model in automobile. In

2014 ieee international conference on computational intelligence and computing research (pp. 1–4).

Thramboulidis, K., Frey, G. et al. (2011). Towards a model-driven IEC 61131-based development process in industrial automation. *Journal of Software Engineering and Applications*, *4*(04), 217. doi: 10.4236/jsea.2011.44024

Townsend, J. D., Montoya, M. M. & Calantone, R. J. (2011). Form and function: A matter of perspective. *Journal of Product Innovation Management*, *28*(3), 374–377. doi: 10.1111/j.1540-5885.2011.00804.x

Tremblay, M. C., Hevner, A. R. & Berndt, D. J. (2010). Focus groups for artifact refinement and evaluation in design research. *Cais*, *26*(27), 599–618. doi: 10.17705/1CAIS.02627

Trochim, W. M. & Donnelly, J. P. (2001). *Research methods knowledge base* (Vol. 2). Atomic Dog Pub.

UIDAHO. (2018). Time Overcurrent Basic Equations. Retrieved 27 July 2021, from `https://webpages.uidaho.edu/ece/ee/power/ECE525/Lectures/L11/L11.pdf`

Uschold, M., Gruninger, M. et al. (1996). Ontologies: Principles, methods and applications. *Technical Report - University of Edinburgh Artificial Intelligence Applications Institute AIAR TR*.

Van Heesch, U., Avgeriou, P. & Hilliard, R. (2012). A documentation framework for architecture decisions. *Journal of Systems and Software*, *85*(4), 795–820. doi: 10.1016/j.jss.2011.10.017

Vassev, E. & Hinchey, M. (2014). Software Engineering for Aerospace: State of the Art. In *Autonomy requirements engineering for space missions* (pp. 1–45). Springer. doi: 10.1007/978-3-319-09816-6_1

Vogel-Heuser, B., Schütz, D., Frank, T. & Legat, C. (2014). Model-driven engineering of manufacturing automation software projects–A SysML-based approach. *Mechatronics*, *24*(7), 883–897. doi: 10.1016/j.mechatronics.2014.05.003

Vyatkin, V. (2011). IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *IEEE transactions on Industrial Informatics*, *7*(4), 768–781. doi: 10.1109/TII.2011.2166785

Wang, J., Ye, L., Gao, R. X., Li, C. & Zhang, L. (2018). Digital twin for rotating machinery fault diagnosis in smart manufacturing. *International Journal of Production Research*, 1-15. doi: 10.1080/00207543.2018.1552032

Wang, J., Zhang, L., Duan, L. & Gao, R. X. (2017). A new paradigm of cloud-based predictive maintenance for intelligent manufacturing. *Journal of Intelligent Manufacturing*, *28*(5), 1125–1137. doi: 10.1007/s10845-015-1066-0

Waszecki, P., Lukasiewycz, M. & Chakraborty, S. (2015). Decentralized diagnosis of permanent faults in automotive e/e architectures. In *Embedded computer systems: Architectures, modeling, and simulation (samos), 2015 international conference on* (pp. 189–196). doi: 10.1109/SAMOS.2015.7363675

Weyns, D., Omicini, A. & Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, *14*(1), 5–30. doi: 10.1007/s10458-006-0012-0

Wiener, N. (1948). *Cybernetics or Control and Communication in the Animal and the Machine* (Vol. 25). MIT press.

Wilson, J. R. (2002). Responsible authorship and peer review. *Science and Engineering Ethics*, *8*(2), 155–174. doi: 10.1007/s11948-002-0016-3

Windmann, S. & Niggemann, O. (2015). Efficient fault detection for industrial automation processes with observable process variables. In *2015 ieee 13th international conference on industrial informatics (indin)* (pp. 121–126). doi: 10.1109/INDIN.2015.7281721

Wohlin, C. (2014). Writing for synthesis of evidence in empirical software engineering. In *Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement* (p. 46). doi: 10.1145/2652524.2652559

Wohlin, C. & Aurum, A. (2015). Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, *20*(6), 1427–1455.

Wohlin, C., Runeson, P., Neto, P. A. d. M. S., Engström, E., do Carmo Machado, I. & De Almeida, E. S. (2013). On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, *86*(10), 2594–2610. doi: 10.1016/j.jss.2013.04.076

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R. & Wood, B. (2006). *Attribute-driven design (ADD), version 2.0* (Tech. Rep.). Carnegie-Mellon University Pittsburg PA. Software Engineering Institute (SEI).

Woods, E. & Rozanski, N. (2009). The system context architectural viewpoint. In *2009 joint working ieee/ifip conference on software architecture & european conference on software architecture* (pp. 333–336).

Wooldridge, M. J. (2009). *An introduction to multiagent systems*. John Wiley & Sons.

Wooldridge, M. J. & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The knowledge engineering review*, *10*(2), 115–152.

Workers, W. (2020). *Franka Emika Panda Research Robot Manual* [Internet web page]. Author. Retrieved 20 November, 2020, from `https://www.wiredworkers.io`

Wray, R. E. & Jones, R. M. (2005). An introduction to Soar as an agent architecture. *Cognition and multi-agent interaction: From cognitive modeling to social simulation*, 53–78.

Wright, H. K., Kim, M. & Perry, D. E. (2010). Validity Concerns in Software Engineering Research. In (p. 411-414). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1882362.1882446

Wu, D., Liu, S., Zhang, L., Terpenny, J., Gao, R. X., Kurfess, T. & Guzzo, J. A. (2017). A fog computing-based framework for process monitoring and prognosis in cyber-manufacturing. *Journal of Manufacturing Systems*, *43*, 25–34. doi: 10.1016/j.jmsy.2017.02.011

Wüest, D., Seyff, N. & Glinz, M. (2012). Flexisketch: A mobile sketching tool for software modeling. In *International conference on mobile computing, applications, and services* (pp. 225–244). doi: 10.1007/978-3-642-36632-1_13

Yang, C.-W., Dubinin, V. & Vyatkin, V. (2019). Automatic Generation of Control

Flow from Requirements for Distributed Smart Grid Automation Control. *IEEE Transactions on Industrial Informatics*. doi: 10.1109/TII.2019.2930772

Yang, X. & Chen, L.-j. (2009). Design and fault diagnosis of Petri net controllers for Petri nets with uncontrollable and unobservable transitions. *Journal of Manufacturing Systems*, *28*(1), 17–22.

Yen, I.-L., Zhang, S., Bastani, F. & Zhang, Y. (2017). A framework for IoT-based monitoring and diagnosis of manufacturing systems. In *2017 ieee symposium on service-oriented system engineering (sose)* (pp. 1–8). doi: 10.1109/SOSE.2017.26

Youn, W. K., Hong, S. B., Oh, K. R. & Ahn, O. S. (2015). Software certification of safety-critical avionic systems: DO-178C and its impacts. *IEEE Aerospace and Electronic Systems Magazine*, *30*(4), 4–13. doi: 10.1109/MAES.2014.140109

Young, W., Boebert, W. & Kain, R. (1988). Proving a computer system secure. *Advances in Computer System Security., 1988,*, *3*.

Zave, P. (1982). An operational approach to requirements specification for embedded systems. *IEEE transactions on Software Engineering*(3), 250–269. doi: 10.1109/TSE.1982.235254

Zhabelova, Yang, Vyatkin, Etherden & Christoffersson. (2016, Nov). Open architecture for cost effective protection and control of power distribution networks. In *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)* (p. 729-735). doi: 10.1109/SmartGridComm.2016.7778848

Zhabelova, G. & Vyatkin, V. (2012). Multiagent smart grid automation architecture based on IEC 61850/61499 intelligent logical nodes. *Industrial Electronics, IEEE Transactions on*, *59*(5), 2351–2362.

Zhang, W., Shen, G., Huang, Z., Yang, Z. & Xue, L. (2017). An Analysis Tool Towards Fault Tolerance Systems based on AADL Error Model. *International Journal of Performability Engineering*, *13*(6), 844–853. doi: 10.23940/ijpe.17.06.p6.844853

Zhang, Y. (2004). Test-driven Modeling for Model-Driven Development. *IEEE Software*, *21*(5), 80–86.

Zhou, X., Gou, X., Huang, T. & Yang, S. (2018). Review on testing of cyber physical systems: Methods and testbeds. *IEEE Access*, *6*, 52179–52194. doi: 10.1109/ACCESS.2018.2869834

Zoitl, A. (2009). *Real-time Execution for IEC 61499*. Instrumentation, Systems, and Automation Society Research Triangle Park, NC.

Zolghadri, A., Cieslak, J., Efimov, D., Henry, D., Goupil, P., Dayre, R., ... Leberre, H. (2015). Signal and model-based fault detection for aircraft systems. *IFAC-PapersOnLine*, *48*(21), 1096–1101. doi: 10.1016/j.ifacol.2015.09.673

Zowghi, D. & Gervasi, V. (2002). The three cs of requirements: consistency, completeness, and correctness. In *International workshop on requirements engineering: Foundations for software quality, essen, germany: Essener informatik beitiage* (pp. 155–164).

Index

Appendix A

The Software Architecture Document

Robust, well-documented architectures remain one of the most significant factors driving the delivery of successful projects (Brown & McDermid, 2007). However, deciding on an appropriate format for a software architecture document is not a trivial task. Clements et al. (2003) caution that all the effort put in by the architects is wasted if the documentation they produce cannot be understood later by the developers who have to build it.

The Software Architecture Document (SAD) presented here was a core driving guide for all of the development and evaluation phases of the fault diagnostic engines creation that followed. It is based on the *Views and Beyond* document template (Mellon, 2018) that captures the design in each of the steps of the Attribute-Driven Design process.

The document faithfully re-creates each of the sections of the Carnegie-Mellon template to illustrate the maturity and scope of the Views and Beyond approach.

A Software Architecture for a Fault Diagnostic Engine

The software architecture for a Multi-Agent based Fault Diagnostic Engine that can identify and diagnose faults in distributed IEC 61499 Function Block Applications.

Barry Dowdeswell, Roopak Sinha and Stephen G. MacDonell.

Document version 2.0

Contents

List of Figures

1 Document Roadmap

This document presents a software architecture for a Fault Diagnostic Engine, referred to in this document as the *engine*. The engine is able to diagnose faults in IEC 61499 Function Block Applications [1]. Based on the Carnegie-Mellon Software Architecture Document template [2], it presents different views of the system, profiling its structure and the architectural relationships between the elements that implement it.

All of the Carnegie Mellon sections are replicated as exemplars to show the full scope that a commercial software architecture document should address. However, remember that this document was created for use in a research environment, not a commercial company. All stakeholders should therefore be considered to be archetypes of people who would typically be encountered in real-world environments. Hence some parts, such as Section 2.3, may seem irrelevant in this non-commercial context. They are included since they are illustrative of the scope that the Carnegie Mellon *Views and Beyond* methodology encompasses [3].

The Document Roadmap section details the organization of this software architecture document. It allows readers to quickly locate the parts of the documentation that are of interest to them.

The sub-sections of this document include:

Section 1.1 - Document Management and Configuration explains how this architectural document has evolved through each of the design phases that were used to create the engine.

Section 1.2 - Purpose and Scope of the Software Architecture Document details the purpose and scope of this document.

Section 1.3 - How this Software Architecture Document is Organized explains what information you should expect to find in each section.

Section 1.4 - Stakeholder Representation profiles each of the stakeholders whose needs this system is designed to address. It explains how each of these users might use this documentation to verify that the system design will satisfy their needs.

Section 1.5 - Viewpoint Definitions explains the Viewpoints defined by IEEE Standard 42010 [4] that have been used in this document. Subsequent sections explore individual Views in detail, each one presented from one of the Viewpoints defined in this standard. Note that ISO 42010 supersedes the earlier IEEE 1471 standard [5] mentioned in the Carnegie Mellon documentation.

Section 1.6 - How a View is Documented explains the organization of the views of the architecture presented in this document. It explains which section contains the view you should look at to find the information you are seeking.

1.1 Document Management & Configuration Control

This section identifies the version, release date, and other relevant management and configuration control information:

Revision Number: 1.0

- *Revision Release Date:* 13th May, 2019.
- *Purpose of Revision:* Initial release of the Software Architecture Document for comment.
- *Scope of the Revision:* The initial software architecture for the fault diagnosis system was created during the first design phase. It was used to guide a number of experiments that evaluated the design of agents and the function block diagnostic interfaces.

Revision Number: 1.1

- *Revision Release Date:* 10th January, 2020
- *Purpose of Revision:* Minor updates to the agent design.
- *Scope of the Revision:* Updates to the design of the agent and their supporting classes.

Revision Number: 2.0

- *Revision Release Date:* 20th November, 2020.
- *Purpose of Revision:* Updates to the software architecture as a result of several design and experimental phases.
- *Scope of the Revision:* Building example agents in version 1.0 refined the operation and interactions of the agents under the GORITE Multi-Agent Framework. Enhanced diagnostic function blocks, known as Diagnostic Points (DPs), were also developed during this phase. This release introduces the agent belief structures, Diagnostic Packages, and the dynamic runtime loading of diagnostic code for agents.

1.2 Purpose and Scope of the SAD

This document presents a software architecture for a semi-automated fault diagnostic engine. Since IEC 61499-compliant function block applications are often partitioned to run on multiple, separate devices, any practical fault diagnostic system must be able to co-ordinate fault-finding tasks across spatially-separated instances of the application being examined.

The decision to use a multi-agent approach, and to only target IEC 61499-compliant applications, are constraints imposed partly by our research interests. We sought to

create a fault-finding system that could change its diagnostic tactics autonomously, based on its own evaluation of the fault tests it has just performed. Hence the use of IEC 61499 architectures, coupled with the choice of the GORITE Multi-Agent Framework to support reasoning agents, should be viewed as architectural constraints imposed during the initial requirements elicitation.

Hence this document also has to present enough detail about GORITE to address the architectural features of the agents that are possible. For this reason, the viewpoints and views of agent functionality are the primary focus of the architecture. While they are not strictly stakeholders, agents can also be considered as actors who use the system to perform tasks. These structures also enable separate instances of the engine to be run on the computers that host distributed parts of the target Function Block Application. Inter-agent communication across distinct instances of the engine to share diagnostic tasks and telemetry is an architecturally-significant requirement that this analysis should address.

1.3 How the SAD is Organized

This section provides a narrative description of the major sections of this software architecture document and its overall contents. Readers seeking specific information can use this section to help them locate it more quickly.

Section 1 - Document Roadmap provided information about this document and its intended audience. Section 1 describes how this document is organized, which stakeholder viewpoints are represented, how stakeholders are expected to use it, and where information particular to any view may be found.

Section 2 - Architecture Background explains why the architecture is the way it is. It provides a system overview to establishing the context and goals for the development. In so doing, it describes the background and rationale for the software architecture chosen. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been used in the architecture. It also includes information about the evaluation performed on the architecture to provide assurance it will meet its goals.

Section 3 - Views and Section 4 - Relations Among Views specify the software architecture in detail, presenting views of the systems from the viewpoint of each stakeholder.

Section 5 - Referenced Material provides information about relevant publications or subsystems whose information is not included directly in this document.

1.4 Stakeholder Representation

This section identifies the stakeholders who have an interest in the Fault Diagnostics System. It profiles their concerns and explains how the architecture will address their needs. The primary stakeholders include:

Software Engineers, who will specify diagnostic tests for the function block application they are designing. In our proposed Model-Driven Design with Diagnostics (MDDD) methodology, diagnostic tests are created early by software engineers [6]. Like other tests and resources, they are created at appropriate points during the design and development phases. MDDD introduces the concept of *Diagnostic Packages*, created by the engineers, that follow the function blocks through their life-cycle.

Test Engineers, who provide support for the application during early-stage testing and later during go-live commissioning. The set of diagnostic tests they employ may be focused on differing needs. Examples include longer duration burn-in trials that would not necessarily be performed by Software Engineers. Diagnostic Packages can contain multiple test resources to address different phases of the function blocks life-cycle.

Maintenance Engineers, who provide long-term support for the system. Their investigations may influence later maintenance or design changes as the product line evolves. At any time during the products lifetime, especially if it fails, the whole system or the part of it can be switched to diagnostic mode. Their immediate need is to identify and isolate the fault with assistance from the fault diagnostic engine and its agents. Their work focuses on restoring the function block application to a fully-operational state. When the engine is deployed in routine monitoring mode, the agents may be the entities that detect misbehavior first, alerting Maintenance Engineers to the problem.

Developers who will implement and refine the software architecture while developing the Fault Diagnostic Engine. They will use representative function block applications and Diagnostic Packages to evaluate the design and performance of the engine as it is being developed.

1.5 Viewpoint Definitions

An architectural *Viewpoint* is a collection of patterns, templates and conventions for constructing and presenting a particular architectural *View* or perspective of our application. Viewpoints are defined with reference to the IEEE 42010 Recommended Practice standard [4]. The Carnegie Mellon *Views and Beyond* approach builds on Kruchten's original 4+1 View Model of Architecture [7] by adopting Rozanski & Wood's seven distinct viewpoints [8, 9], which in turn are based on IEEE 42010.

This section provides a short description of each of the applicable viewpoints available to the architects that can be used to define the views presented in Section 3.

1.5.1 Context Viewpoint Definition

1.5.1.1 Abstract. Views conforming to the Context viewpoint present the relationships, dependencies and interactions between the individual subsystems that the engine is constructed from. Views created from this viewpoint correspond to the Scenarios view that forms the +1 view of Kruchten's 4+1 View model of architecture [7]. They present a high-level overview showing how the system in constructed, how it interacts with the stakeholders who use it, and the environment it operates in.

Rozansi and Woods comment that the context of the system is often defined too loosely during analysis [9]. Since the rest of the views focus on *interior* or internal constructs, the Context viewpoint is needed to present a wider perspective of subsystems, their boundaries, and their interfaces.

1.5.1.2 Stakeholders and their Concerns Addressed. The Context view is relevant to all stakeholders since it presents a high-level view to show how their concerns are addressed in the architecture. The stakeholders include the software, test and maintenance engineers who formulate diagnostic plans and make them available for the agents to use while hunting for faults. The Architecturally Significant Requirements identified are also of interest to the developers since they identify concerns that must be addressed while the engine is being developed.

1.5.1.3 Elements, Relations, Properties, and Constraints. Elements of the Context Viewpoint are self-contained sub-systems, including the agents that hunt for faults in the function block applications and the components that allow them to interact with the individual function block that are being examined.

1.5.1.4 Languages used to Model and Represent Conforming Views. SysML Internal Block diagrams are used to create Extended System Context Diagrams [10]. These diagrams present the core systems and their relationships to each other.

1.5.1.5 Applicable Evaluation, Consistency and Completeness Criteria. The Context Viewpoint is evaluated in terms of its completeness. All systems and sub-systems that contribute to fulfilling the system goals and requirements should be visible in this perspective. The naming of elements should be consistent, since the Context diagrams present a roadmap to place all other parts of the system in-perspective.

1.5.1.6 Viewpoint Source. This viewpoint was developed from the definition presented by Rozanski & Woods [9, 11].

1.5.2 Functional Viewpoint Definition

1.5.2.1 Abstract. The Functional Viewpoint describes the operation of the parts that perform the tasks that enable the system to fulfill its purpose. This viewpoint presents the relationships, dependencies and interactions between the individual sub-systems that the application is constructed from. The design choices made for

these views of the application are often the ones most influenced by the architectural decisions made earlier.

1.5.2.2 Stakeholders and their Concerns Addressed. Since this view details the functionality the engine provides, it is applicable to all stakeholders. The software, test and maintenance engineers need to be aware of the capabilities the engine provides. The developers also need to understand the proposed functionality in terms of the components that will be required.

1.5.2.3 Elements, Relations, Properties, and Constraints. Elements of the Functional Viewpoint are self-contained subsystems. These will later be specified down to the level of classes, interfaces and methods in the Development views.

1.5.2.4 Languages used to Model and Represent Conforming Views. SysML Block Definition and Use Case diagrams are used to represent the relationships between all high-level subsystems.

1.5.2.5 Applicable Evaluation, Consistency and Completeness Criteria. The Functional views are evaluated by tracing them back to the requirements that they address. The existence of every functional part must be able to be justified with reference to the requirements it fulfills. This traceability also provides a measure of the completeness of the requirements coverage. The quality of a functional part is also assessed later in the Development views in terms of how well it passes the unit and other tests prescribed for it.

1.5.2.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Bass et al., [3] and Rozanski & Woods [11].

1.5.3 Information Viewpoint Definition

1.5.3.1 Abstract. Views from the Information Viewpoint describe how the architecture stores, processes and distributes information. This viewpoint is designed to capture a complete but high-level view of both static data structures and information flows. This view also addresses issues such as content, how that content is structured, and who owns the information that flows between different parts of the system.

1.5.3.2 Stakeholders and their Concerns Addressed. This viewpoint addresses issues that are relevant to all stakeholders since its emphasis is on what information is available, not just on how it is persisted.

1.5.3.3 Elements, Relations, Properties, and Constraints. Elements of the Information Viewpoint are often high-level groupings of data needed for different sub-systems within the application. Properties can include the accuracy of numeric values while the constraints often relate to what information individual subsystems have access to. These constraints can also address security concerns and access

rights for different users and components of the system.

1.5.3.4 Languages used to Model and Represent Conforming Views. Information Views present models of the information held, often as static information structure models. Other models commonly used include information flow, lifecycle, ownership and quality analysis models.

1.5.3.5 Applicable Evaluation, Consistency and Completeness Criteria. Information views are evaluated by how well the resulting data structures capture all the information defined in the requirements. Requirements traceability is used to ensure that the information needs of each requirement is realized in the proposed data structures.

1.5.3.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Rozanski & Woods [12], and Bass et al., [3]

1.5.4 Concurrency Viewpoint Definition

1.5.4.1 Abstract. Historically, concurrency has been a key issue with control systems. Systems created with function blocks are inherently distributed so they can run as separate instances on different hardware. They interface with a myriad of external sensors and actuators. Hence, synchronization of data flows and decision processing demands appropriate concurrency control. Issues addressed in this viewpoint can include state management, startup and shutdown synchronization, task failure and reentrancy.

1.5.4.2 Stakeholders and their Concerns Addressed. This viewpoint addresses the concerns of developers, testers and system administrators. All other users expect concurrency to have been addressed transparently by these stakeholders.

1.5.4.3 Elements, Relations, Properties, and Constraints. Elements of the Concurrency viewpoint are those that have to honor the order and precedence of other systems and the sub-systems they interact with.

1.5.4.4 Languages used to Model and Represent Conforming Views. State models and system-level concurrency models. SysML diagrams such as Activity and Sequence Diagrams are common ways of modeling data exchange and process synchronization.

1.5.4.5 Applicable Evaluation, Consistency and Completeness Criteria. Concurrency problems are often evidenced by deadlocks, resource contention and race conditions. After identifying those parts of the system where concurrency is required, it is important to determine if the application provides sufficient overhead to allow for correct process switching. The design should ensure that tasks are prioritized correctly and that there is sufficient communications bandwidth [13, 14].

1.5.4.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Rozanski & Woods [12].

1.5.5 Development Viewpoint Definition

1.5.5.1 Abstract. After first understanding the views from the Context and Functional viewpoints, the views that present the Development viewpoint become the most important to the developers. These views are usually refined after the other preliminary views have been created and as the project progresses. A well-written set of development view packets provide the foundation for later documentation that is written to support the system.

1.5.5.2 Stakeholders and their Concerns Addressed. The designers and developers are the primary stakeholders whose concerns are addressed via this viewpoint.

1.5.5.3 Elements, Relations, Properties, and Constraints. Elements of the Development Viewpoint are coding structures, dependencies, interfaces and system-wide standards. Each element is evaluated from the perspective of how it contributes to the integrity of the system.

1.5.5.4 Languages used to Model and Represent Conforming Views. SysML diagrams are used to represent the relationships between system elements, including Activity and Sequence diagrams. SysML Class diagrams model classes, their interfaces, functions and methods.

1.5.5.5 Applicable Evaluation, Consistency and Completeness Criteria. Development views are evaluated by how well the resulting structures and relationships between the subsystems fulfill the requirements. Requirements Traceability is a key indicator of integrity, helping to ensure that the presence of any element in the system can be justified.

1.5.5.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Hillard [15], Rozanski and Woods [12], and Bass et al., [3]

1.5.6 Deployment Viewpoint Definition

1.5.6.1 Abstract. The Deployment viewpoint describes the environment that the system will be deployed into. This includes an understanding of the dependencies the system has on its runtime environment as well as what tasks are required to deploy it into production.

1.5.6.2 Stakeholders and their Concerns Addressed. The designers, developers and those who will deploy the system are the primary stakeholders whose concerns

are addressed in this viewpoint.

1.5.6.3 Elements, Relations, Properties, and Constraints. The elements profiled in this view are packages of functionality and the confguration information necessary to install runtime components and databases. Views from this viewpoint are often presented as structured lists or text documents.

1.5.6.4 Languages used to Model and Represent Conforming Views. Deployments can be modeled with Runtime Platform Models, Network Models and component dependency matrices.

1.5.6.5 Applicable Evaluation, Consistency and Completeness Criteria. Evaluation criteria to measure successful deployments should be based on empirical performance measurements made against pre-defined criteria. Data Migration plans and models address data needs when a system is replacing a pre-existing solution that requires legacy data to be imported.

1.5.6.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Rozanski & Woods [12] and Scratchly [14].

1.5.7 Operational Viewpoint Definition

1.5.7.1 Abstract. The Operational viewpoint describes how the system will be operated, maintained and supported after it has been deployed into its production environment. Rozanski & Woods comment that this viewpoint is often the one that is least well defined [9]. It needs both elaboration and refinement while the system is being constructed.

1.5.7.2 Stakeholders and their Concerns Addressed. The designers, developers and the production users are the primary stakeholders whose concerns are addressed this viewpoint addresses.

1.5.7.3 Elements, Relations, Properties, and Constraints. The elements of this viewpoint consist of all the parts of the system that comprise its final deployed configuration.

1.5.7.4 Languages used to Model and Represent Conforming Views. Configuration management and administration models can be used to model the operational needs of the system.

1.5.7.5 Applicable Evaluation, Consistency and Completeness Criteria. All evaluation criteria for successful operation are derived from the original requirements. These should include non-functional specifications for qualities such as performance and availability.

1.5.7.6 Viewpoint Source. This viewpoint was developed from the definitions presented by Rozanski & Woods [12], and Bass et al., [3]

1.5.8 Stakeholders and relevant Viewpoints

Stakeholder	*Viewpoint(s) that apply to those stakeholder's concerns.*
Software Engineers	Context, Operational.
Test Engineers	Context, Operational.
Maintenance Engineers	Context, Operational.
System designers and developers	Context, Functional, Deployment, Development, Information, and Operational.

1.6 How a View is Documented

All views have been documented using UML diagrams [16] and SysML diagrams [17], an extension of UML that introduces Requirements, Sequence and Activity diagrams that were not originally a part of UML. Used together, UML and SysML brings a rigor and consistency that is not always present in free-form diagrams.

1.7 Relationship to Other SADs

This software architecture document for the engine is self-contained: it does not reference any other architectural document. Since this architecture focuses on the design and integration of the diagnostic agents into GORITE, details of the internal architecture of the GORITE framework are discussed only where necessary.

1.8 Process for updating this document

In a commercial software architecture document, this section should outline processes and procedures for rectifying errors found in the documentation. During this research, all significant changes originated from evaluations performed at the end of each research phase. Where applicable, the version number of the software was incremented and changes were documented in Section 1.1 as part of a new release.

2 Architecture Background

The architectural background section provides a description of the engine, helping to explain the architectural decisions made and the problems they address. This section includes an overview of the engine in terms of its goals and the context it operates in. The requirements which are architecturally significant are also identified.

2.1 Problem background

Traditional Model-Driven Developement (MDD) has concentrated on establishing appropriate software models to justify the architectural and design decisions made by the architects. However, even if diagnostic test information was created and captured during the modeling stages, IEC 61499 function block applications cannot use it easily. Current Integrated Development Environments (IDE's) such as 4diac [18] and nxtSTUDIO [19] do not provide the ability to execute diagnostic tests across multiple, connected function blocks.

The engine is a stand-alone application that addresses the need for both interactive and unattended analysis of IEC 61499 applications at runtime. It is able to exercise and examine running, failing, or fully-failed IEC 61499 applications. By applying tactics to isolate sections of the function block network and verify functionality, the engine can narrow down and identify a set of candidate faults and their locations. Interactive analysis promotes good Test-Driven Design techniques by encouraging the developer to refine their function blocks iteratively, supported by empirical evaluations of both suitability and performance.

Function block application development typically involves the creation of new function blocks based on templates. Where pre-existing libraries of blocks exist from previous projects, a designer may decide to re-use a proven block. As each block is developed, it is wired into the function block application, creating an increasingly more complex solution. The scope of diagnostic analysis should include both the testing of individual function blocks followed by repeated integration testing until the application is complete. Then, longer-duration evaluations are required to determine how well each function block fulfills its role in the system.

Like Unit Tests for classes, diagnostic resources that have been developed for each function block should remain available as *Diagnostic Packages*. They can be included in the libraries of re-usable function blocks that organizations develop. This contributes to the rigor of MDD by enabling the creation of comprehensive diagnostic resources that can accompany function blocks application though their complete life-cycle as they are developed and later re-used across multiple projects.

2.1.1 System Overview

Contents of this Section. At any stage of development, the engine can be launched to perform both fault monitoring and diagnostic intervention tasks. The engine schedules these tasks, assigning them to teams of semi-autonomous *diagnostic agents*, hereafter referred to as *agents*. Agents posses sufficient domain knowledge of the IEC 61499 software architecture. This enables them to identify the relevant input data ports, output data ports, and the matching event ports on each function block instance. Using information available in the diagnostic packages, the agents reconfigure the wiring between connected function blocks to gain access to telemetry being exchange at runtime. These access points, referred to as *Diagnostic Points* (DPs) allow the agent to monitor data that is sent into a function block and watch the data that is sent out of the block after processing. Events can also be watched

independently. Figure 1 illustrates the high-level context of the three interacting entities that are participating in fault-finding activities.

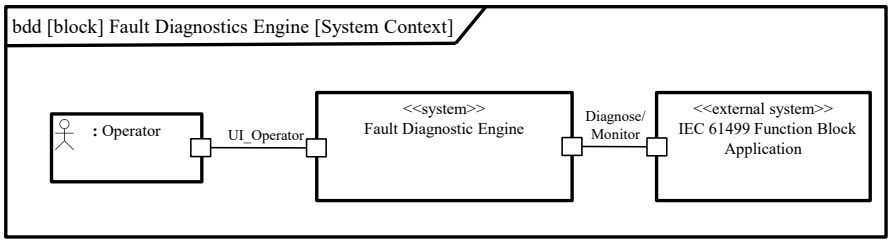

Figure 1: Fault Diagnostic Engine Context View.

When an agent operating withing the engine observes evidence of faulty behavior, the DPs can be instructed to isolate groups of function blocks so they can be tested interactively. Each diagnostic package carries detailed test procedures and representative test data that can be used to evaluate a particular function block type. Agents can inject test values, via a DP, into the input port of a function block. Using other DPs, the agent can then capture data that is output from the function block as well as subsequent events that are triggered. Where a group of function blocks are wired together to accomplish a task, the agent can step through each potential fault pathway, hunting for evidence of misbehavior.

2.1.2 Goals and Context

Parts of a larger IEC 61499 function block application are often partitioned along logical boundaries so that they can be run on separate, connected hardware instances. These instances exchange data and status information as they interact with each other. Diagnosing faults in such environments involves tapping into points in the system to monitor activities or inject data. Hence the proposed architecture addresses the need to deploy entities that can perform these tasks co-operatively at appropriate locations.

Multi-Agent Systems support multiple instances of semi-autonomous agents who can work co-operatively. The engine has been designed to operate using a team of discrete agent instances, hosted within the GORITE (Goal ORiented TEams) multi-agent framework [20, 21].

We can therefore define the "engine" as being the collection of components that includes the agents, their knowledge management structures, and the custom-designed functionality that supports their fault-finding activities. This software architecture document therefore focuses only on the design of the agents themselves and their components. Details of the internal architecture of the GORITE framework itself is included only when necessary.

2.1.3 Significant Driving Requirements

Not all requirements drive or influence the architectural decisions made by the architects. The requirements that do are referred to as Architecturally Significant Requirements (ASRs). Chen et al. comment that when architecturally significant requirements are wrong or incomplete, they can lead to architectures that contain errors [22]. ASRs are the subset of the Primary Functional Requirements that influenced the key architectural decisions made while designing the engine. These are presented as a set of Quality Attribute Scenarios that describe qualities that the system must possess to achieve its goals. The quality classifications are derived from the ISO 25010 Standard Systems and software Quality Requirements and Evaluation Standard [23].

QAS 1 - ISO 4.2.1 Functional Suitability Quality Attribute Scenario

Quality requirement: The agents must be able to interact with function block applications designed to run under the 4diac FORTE [18, 24] and nxtSTUDIO [19] runtime environments.

Source: The agent or its components that are interacting with the function block data inputs, outputs and events within the target function block application.

Stimulus: A function block output is being monitored or an input is being supplied by the agent.

Artifact: The Function Block Application.

Environment: The platform that the function block application is running on.

Response: The agent can supply an appropriate test data value to a function block input or capture the event or data from an output.

Response Measure: Each agent is able to interact with multiple function blocks simultaneously. It must be able to reliably capture 95% of all available telemetry from each diagnostic point that is being monitored (a significance level of $\chi = 0.05$).

Comments: FORTE and nxtSTUDIO create function block applications that are compiled and deployed differently. The method of capturing output activity and stimulating inputs must be compatible with those two runtime environments.

QAS 2 - ISO 4.2.2 Performance Efficiency Quality Attribute Scenario

Quality requirement: Agents must be able to exercise or monitor function blocks which are operating with events that are being triggered at the highest rate defined by the particular function block application specification.

Source: External, event timing limits imposed by the function block application

design constraints.

Stimulus: An event generated by a function block that is being monitored or a data input event that is being triggered by the agent.

Artifact: A agent that is monitoring the event or a function block that is being triggered.

Environment: The function block application when a diagnostic point is being monitored or a data value is being supplied to the diagnostic point.

Response: The agent watching for a corresponding data and event output from the input event and data that was triggered determines that the event has occurred.

Response Measure: The agent must be able to interact with function blocks through the diagnostic points within the time interval available. This requires an agent to be operating at least twice as fast as the function block application is cycling to satisfy the Nyquist-Shannon sampling constraint [25, 26].

Comments: A sample FORTE application runtime was tested to determine how fast the function block application could execute in the development environment. A standard 4diac E_TIMER function block was configured to trigger the test function block at 100ms intervals. Events and data were captured reliably within the requirements of QAS 1.

QAS 3 - ISO 4.2.2 Performance Efficiency Quality Attribute Scenario

Quality requirement: The operation of diagnostic points must not adversely impact the performance of the individual function blocks or the overall performance of the function block application itself.

Source: The activities of the agents and the diagnostic points.

Stimulus: One or more diagnostic point marshaled by an agent are capturing outputs from or supplying inputs to a function block.

Artifact: The diagnostic points and the function blocks that are being monitored.

Environment: An agent is interacting with one or more function blocks via a set of diagnostic points.

Response: The agents shall have no adverse effect on the performance of the function block application. They may cause the application to run more slowly while the system is being monitored, but they must not break specified timing constraints or

detrimentally affect the overall throughput of the system.

Response Measure: The function block application continues to operate within the timing constraints expected of it when it was not being monitored. The monitoring functionality should not cause more than a 5% decrease in the performance of the function block application being diagnosed.

QAS 4 - ISO 4.2.5 Reliability Quality Attribute Scenario

Quality requirement: The engine and its agents must continue to operate reliably even while interacting with a malfunctioning or failed function block application.

Source: A failure of the function block application or one of its component parts.

Stimulus: A failure has occurred within the function block application.

Artifact: The function block application, the diagnostic points and the agent.

Environment: While an agent is running a diagnostic test or monitoring one or more function blocks.

Response: The agent detects a potential fault.

Response Measure: The engine and its components continue to operate reliably and within tolerance, capturing all data to correctly within the bounds of the other QAS. The monitoring functionality continues to operate, reliably capturing 95% of all available telemetry from each diagnostic point that is being watched.

Comments: This QAS does not apply to engines that are operating on the same hardware as the function block application. Under those circumstances, processor hardware failures may render this QAS invalid. Reasonable risk management suggests that the engine should operate on a separate, redundant platform of its own wherever possible. Refer to QAS 5 - ISO 4.2.8 later in this document.

QAS 5 - ISO 4.2.8 Portability Quality Attribute Scenario

Quality requirement: The engine and agents must be able to run natively on the target device that is running the function block application for some fault-finding or monitoring scenarios.

Source: The engine and agents.

Stimulus: The run-time requirements of the engine and the agent components.

Artifact: The host operating system for the function block application. The runtime environment of the function block application: FORTE or nxtSTUDIO.

Environment: The engine, its components, and the agents are required to run on the same hardware and operating system as the target function block application.

Response: The agents are able to run and perform their diagnostic tasks correctly.

Response Measure: While operating, the engine and agents running on the target platform shall not cause more than 15% decrease in the performance of the function block application being diagnosed.

2.2 Solution Background

The engine provides a platform to host a team of agents who exercise and examine running, failing or fully-failed IEC 61499 function block applications.

Agents are able to manage their own fault-finding activities, following pre-defined plans supplied by users. The agents possess a degree of domain knowledge about the architecture and operation of IEC 61499 applications. While hunting for faults, they build and maintain a belief structure about what the function block application has done during the current analysis run. The beliefs capture the evidence trail that the agents will later present as their fault diagnosis and evaluation.

This evidence-based belief structure allows agents to reason about what they have observed. Influenced by the domain knowledge they hold, they interpret fault evidence and choose to change their tactics to execute alternative paths through the diagnostic task list. Test plans can include detailed example test values or methods that the agents can apply at their own discretion. Agents can also co-operate with other agents, forming teams to share tasks and information. They can also agree on shared tactical decisions.

2.2.1 Architectural Approaches

This section discusses the architectural design decisions that have been proposed to address both the functional and quality requirements.

2.2.2 Analysis Results

After the completion of version 1.0 of the architecture document, a series of prototypes were created to evaluate critical parts of architecture. These included a number of different types of specialized interface function blocks that could be used to interact with a representative set of IEC 61499 function block applications. This helped to refine the Transmission Control Protocol (TCP) [27] communications architecture, confirming TCP as the preferred transport protocol between the agents and the diagnostic points. User Datagram Protocol (UDP) [28] was considered, but initial performance testing demonstrated that the overhead of the TCP protocol was justified since it provided guaranteed delivery of complete data packets. The choice of

TCP also simplified the asynchronous queue structure that is used by the engines server component. Since TCP only delivers complete packets, the mechanism used for queuing each packet did not require the component to perform any additional buffering or packet assembly for slower connections.

Initial evaluations were performed with a single agent executing with two diagnostic points. The agent issued commands reliably to the DPs on a cycle time of approximately 10ms while the function block applications were executing with a 100ms cycle time. The engines server FIFO packet queues supported reliable asynchronous packet exchange in both directions between the diagnostic point and the agent. No packet loss was detected in either direction. These evaluations partially validate the QAS 2 Performance Efficiency scenario.

2.2.3 Requirements Coverage

ASR	*View Packets that address that architecturally-significant requirement.*
QAS 1	View Packet 2 - Creating the Diagnostic Harness page 33. View Packet 3 - Agent interactions with Diagnostic Points page 35.
QAS 2	View Packet 1 - Fault Diagnostic Engine Application page 23. View Packet 2 - Creating the Diagnostic Harness page 33. View Packet 3 - Agent interactions with Diagnostic Points page 35.
QAS 3	View Packet 3 - Agent interactions with Diagnostic Points page 35. View Packet 2 - Creating the Diagnostic Harness page 33.
QAS 4	View Packet 1 - Engine Deployment Configuration page 40. View Packet 1 - The Development Environment page 38.
QAS 5	View Packet 1 - The Development Environment page 38. View Packet 2 - Creating the Diagnostic Harness page 33. View Packet 3 - Agent interactions with Diagnostic Points page 35.

2.2.4 Summary of Changes Reflected in this Version

This is version 2.0 of the software architecture document. All changes to the architecture that were made during design phases are reflected in the current document. Significant changes include:

Implementation of Diagnostic Packages. While function blocks are being designed, Software Engineers identify potential monitor points that can be used to capture telemetry. This information is now available in an XML-format file that accompanies each function block. This Diagnostic Package also contains sets of tests specific to this function block which can be run by the agents when they are investigating a fault. This release of the engine also implements the dynamic loading and compiling of the diagnostic tests before the agents begin watching for faults.

Refinement of the diagnostic points. Version 1.0 implemented the diagnostic points

as two separate Service Interface Function Blocks (SIFBs). The AGENT_SEND function block was responsible for transmitting data and events back to an agent. A separate AGENT_RECV function block received test data from an agent to inject into an input port.

This release introduces a single type of SIFB called DP. This block can be reconfigured dynamically by the agent at runtime. In its normal state, a DP instance is configured in the middle of the datapath between two function blocks of interest to create a Diagnostic Point. It passes all events and data through itself passively. Whenever an event is detected, it also transmits a copy of the data back to the agent as shown in Figure 7 in Section 3.3.5.3. During fault finding, the agent can instruct the diagnostic point to block the flow of inbound or outbound data. This functionality, referred to as *gating*, allows the agent to isolate one of more functions blocks of interest so that they can be exercised by the agent directly. The operation of the diagnostic points is discussed in more detail in Section 3.3.5.3.

2.3 Product Line Reuse Considerations

In its first iteration, the engine was primarily concerned with performing after-the-fault analysis on a completed system. However, the engine is also applicable to fault diagnosis during other phases of a product's lifecycle. Once an application has moved into the testing stages beyond its initial development, the engine and its agents can facilitate longer duration automated burn-in trials using similar monitoring techniques. In this mode, the emphasis is on diagnosing catastrophic failures as well as proving quality requirements such as performance, reliability and fault-recovery.

Diagnostic activities in these phases can be facilitated by using specific diagnostic test plans and configurations that are appropriate to the system at its present stage of development. This approach to testing is similar in intent to current DevOps continuous integration approaches [29]. While the system may not be complete, significant sections may already be available that can be stress-tested by the engine. The diagnostic tests would recognize that the scope of their fault-finding activities is restricted and hence, that a full spectrum of analysis tests is not necessarily possible.

3 Views

This section provides distinct yet complementary views of the system, each presented from one of the viewpoints defined previously in Section 1.5. Where multiple views are used to illustrate a viewpoint, each has been presented as a separate *View Packet* [2, page 17].

3.1 System Context View

3.1.1 View Description: The System Context View presents a high-level view of the engine, placing it in context with the views that follow [11]. Similar to the Kruchten +1 View [7], it describes the scenarios that the engine operates in while agents are

hunting for faults.

3.1.2 View packet overview: This view contains a single View Packet.

3.1.3 Architectural background: The engine works in conjunction with the users who are the stakeholders identified in Section 1.5.8. The primary architectural style implemented is a layered architecture with GORITE in the primary layer and the agents in a separate layer [30]. QAS 2 requires that the agents must be able to operate faster than the function block application. While the diagnostic point only has to mediate the flow of data to and from the engine, the agent also has to analyze the data streams from multiple diagnostic points and execute strategies preemptively. Communications between the agents and the diagnostic points is via a client-server architecture. Hence the overall architecture style for the system is a blended one.

3.1.4 Variability mechanism: The engine operates in the same way for 4diac and nxtSTUDIO. Differences in the diagnostic test harness are discussed in Section 3.3.5.2.4.

3.1.5 View packets:

3.1.5.1 View Packet 1 - System Context

3.1.5.1.1 Primary presentation: This view presents the users in the context of the Use Cases for the tasks they accomplish with the engine.

3.1.5.1.2 Element catalogue:

3.1.5.1.2.1 Elements:

Stakeholders and Users. These are the users identified in Section 1.5.8. The user's roles differ only in the way that they interact with the engine and the phase of the function block applications life-cycle where the engine is being used.

Create Diagnostic Package The users create diagnostic packages during the creation of the function blocks that will ultimately be used to assemble the function block application. These are made available to the engine for use during configuration and fault finding.

Agent. One or more agents perform configuration and diagnostic tasks, hosted within the GORITE framework that the engine is created from. The tasks the agents perform include the creation of the diagnostic test harness for either 4diac or nxtSTUDIO function block applications.

Reporting. Two primary reporting phases are supported by the engine. The status of the configuration of the diagnostic test harness is reported after the configuration

before a fault-finding run is completed. A second report that may contain a fault diagnosis is available at the end of a fault-finding session. Real-time status updates are also created by the agents during fault-finding runs.

3.1.5.1.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.1.5.1.2.3 Interfaces: The engine provides an interface to both the users, the resources the function block application is created from, and the running instance of the function block application.

3.1.5.1.2.4 Behavior: Refer to Section 3.3 for details of the behavior of each part of the engine.

3.1.5.1.2.5 Constraints: There are no constraints implicit in this view packet. All constraints are detailed in the relevant views that follow this Context View.

3.1.5.1.3 Context diagram:

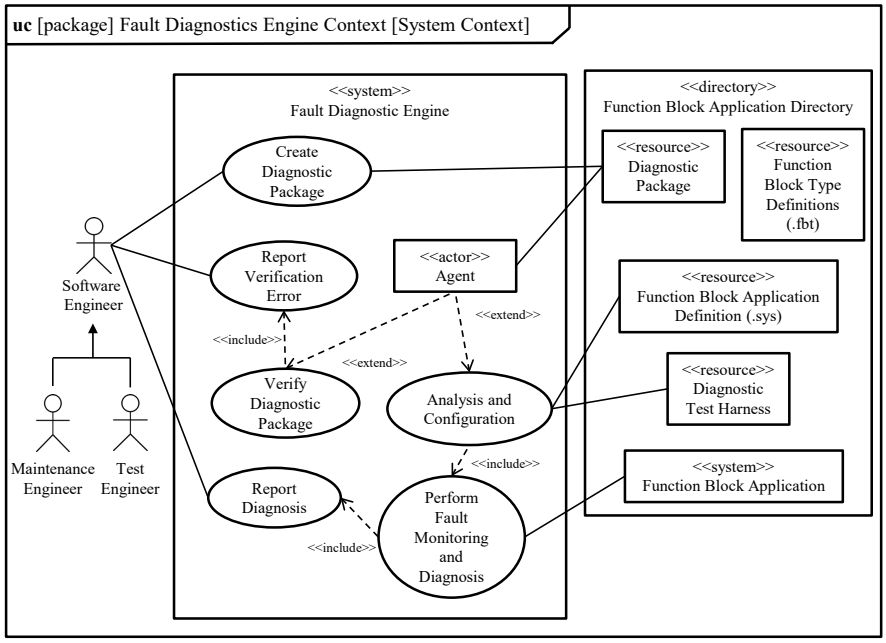

Figure 2: System Context View: The Fault Diagnostic Engine and its Users.

3.1.5.1.4 Variability mechanisms: There are no variabilities in this view.

3.1.5.1.5 Architectural background: The engine and the function block applica-

tion operate asynchronously, exchanging telemetry via network packets. Domain knowledge of the IEC 61499 function block architecture is therefore important to the agents. A brief discussion of the IEC 61499 architecture is included in Section 3.2.2 as part of View Packet 2.

3.1.5.1.6 Related View Packets: Architectural details of each part of the system are presented in more detail in the views and view packets that follow.

3.2 Logical View

3.2.1 View Description: This view presents the architecture of the engine and its subsystems from the Functional Viewpoint. It describes the the agents and the classes that implement their functionality. The engine operates within the GORITE Multi-Agent Framework but as noted in Section 1.7, the internal architecture of GORITE is only discussed where relevant.

3.2.2 View packet overview: This view contains two View Packets. The first presents the functional structure of the engine. The second provides a brief background overview of the IEC 61499 Function Block architecture used to construct the function block applications that the engine interacts with. It illustrates the role of the diagnostic points that facilitate the connection between these two systems.

3.2.3 Architectural background: The teams of agents that operate under GORITE are arranged in top-down, hierarchical functional layers. This architecture is typical of multi-agent systems where the responsibility for performing tasks is delegated to discrete, semi-autonomous agents while team-wide decisions are managed higher up at the team level [20, 31].

3.2.4 Variability mechanism: 4diac and nxtStudio use different approaches to deploying function block applications. To create a runtime environment, 4diac compiles all the custom application function block types and library components into the FORTE executable. It then deploys an application by instructing FORTE to wire together blocks dynamically and launch them. In contrast, nxtStudio statically links all required function blocks and connections into a single executable. While the engine operates similarly for both types of applications, this difference requires the engine to deploy diagnostic points differently during the engines deployment phase.

3.2.5 View packets:

3.2.5.1 View Packet 1 - Fault Diagnostic Engine Application

3.2.5.1.1 Primary presentation: This view presents the component classes used in the engine and the interfaces they present. The diagrams highlight the hierarchy layers and the dependencies between the sub-systems and the elements they are

composed of.

3.2.5.1.2 Element catalogue:

3.2.5.1.2.1 Elements: Figure 3 shown in Section 3.2.5.1.3 presents the structure of the engine application that supports the activities of the agents. The elements of this view are classes that implement the functionality used by the agents:

GORITE is a self-contained set of classes that manage the instantiation and deployment of the plans, goals and tasks of each agent.

Diagnostic Team is the data structure that allows GORITE to co-ordinate the goals and activities of a team of agents who share some or all of their goals in common. The engine supports one generic type of agent that is capable of performing any of the predefined diagnostic abilities, also referred to as *skills*. However, at any time different agents can operate in different parts of the function block application pursuing goals specific that that region of the application. The GORITE Team construct facilitates the agent interactions such as inter-agent communications and task sharing.

Diagnostic Agent. Each agent is deployed on its own thread. When an agent is assigned a goal to execute, it operates asynchronously, reporting back to the Diagnostic Team controller when the status of a goal has changed. Typical status changes reported by an agent include the completion, suspension or the failure of its current goal. GORITE then re-prioritizes the list of goals defined for the agent before issuing new directions. Each agent instantiates its own copy or shares a reference to each of the following resources:

Beliefs enable an agent to "remember" a discrete set of logical data packets that carry information about the nature and status of an element that is part of the function block application. The most common logical unit of information captured in a belief is s a premise about the current status of a single function block. Each belief has a *veracity* which represents the current truthfulness of the premise. An agent holds that a belief may be true, false or undetermined. Agent beliefs for this engine are discussed in more detail in [32].

Capabilities or skills are domain-specific abilities that an agent can employ to perform diagnostic tasks. They enable the agent to interact with a function block in a function block application to observe and interact with it. The Capabilities class provides a set of functions that include the ability to create diagnostic test harnesses, instantiate diagnostic points, and compile diagnostic scripts.

Diagnostics is a class that is used by agents to load and parse the diagnostic package for a specified function block. Each package is encoded as XML-format file.

Function Block Application. 4diac and nxtStudio create XML-format files that define each function block application. The FunctionBlockApp class loads these files to form

a list of function block instances. The function block type definition file is also loaded and parsed for each function block to create a list of data ports, algorithms and events.

Diagnostic Points are classes that track each diagnostic point instance that has been re-wired into the function block application. The diagnostic point object instantiated within the engine maintains the TCP/IP connection between the diagnostic point function block and the agent that is controlling it.

NIOserver. The agents hosted in the engine and the function blocks in the function block application operate on their own asynchronous time cycles. The Non-Blocking TCP/IP server component maintains a set of First-In, First Out (FIFO) packet queues that buffer all interactions between the agents and the function blocks. Using a post/response mechanism, the NIOserver component ensures that the diagnostic point function blocks only need to manage a light-weight TCP/IP client. This ensures that they can meet the QAS for performance and reliability.

HMI (Human-Machine Interface). This class provides methods that enable agents to provide information to the display devices used by the engine operators during diagnostic sessions.

3.2.5.1.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.2.5.1.2.3 Interfaces: The engine and the agents form a self-contained system that does not interface to any other external system except the function block application that is being analysed.

3.2.5.1.2.4 Behavior: IEC 61499 Function Block Applications are event-driven. Agents interact with the function blocks they are analysing by generating input events when presenting test data to the target function block application. Similarly, they capture telemetry generated by the function blocks by recognizing when function blocks have generated output events.

3.2.5.1.2.5 Constraints: Their are no additional constraints within this view packet.

3.2.5.1.3 Context diagram:

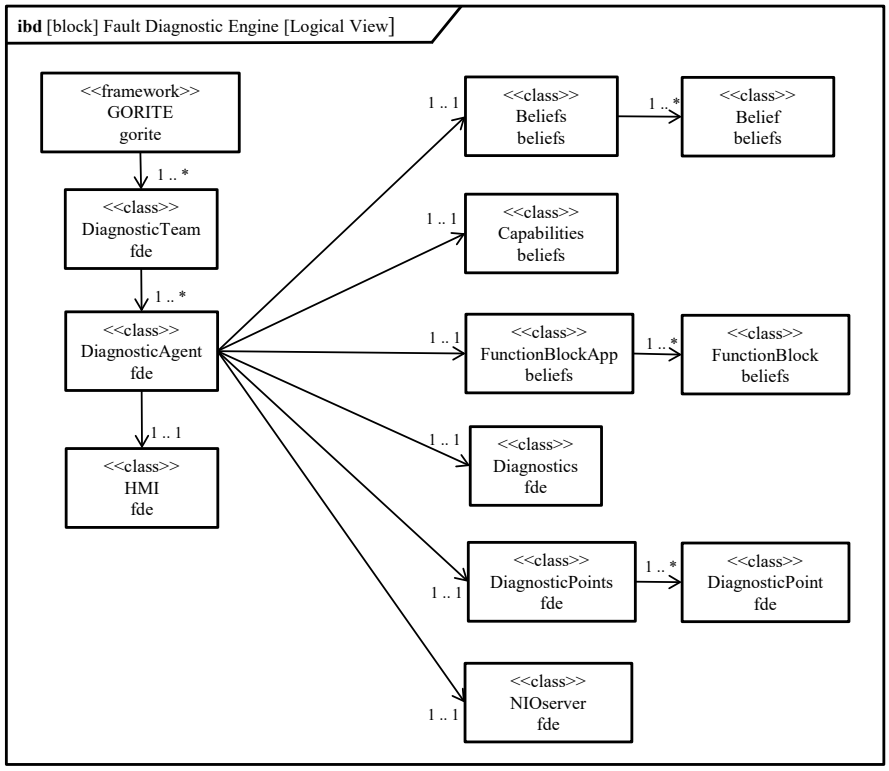

Figure 3: Fault Diagnostic Engine Application Logical View.

3.2.5.1.4 Variability mechanisms: Once the target function block application has been re-wired to include appropriate diagnostic points, the operation of the diagnostic plan is the same for function block applications developed in either 4diac and nxtSTUDIO. Hence there are no significant variations present in this view.

3.2.5.1.5 Architectural background: The GORITE Team structure allows multiple agents to be deployed that can manage their own state. Agents are deployed on their own threads to ensure that they can operate asynchronously, interacting with function blocks in near real-time. Components such as the NIOserver are designed to buffer data packets to assist the agents to stay in sync with the function block application and meet appropriate timing constraints.

3.2.5.1.6 Related View Packets: The engine and the function block application operate asynchronously, exchanging telemetry via network packets. Domain knowledge of the IEC 61499 function block architecture is therefore important to the agents. A brief discussion of the IEC 61499 architecture is included in Section 3.2.5.2 as part of

View Packet 2.

3.2.5.2 View Packet 2 - Function Block Application

3.2.5.2.1 Primary presentation: A Function Block Application is a program that has been written according to the guidelines of the IEC 61499 Function Block Reference Architecture [1]. These applications provide computing functions for cyber-physical systems.

3.2.5.2.2 Element catalogue:

3.2.5.2.2.1 Elements: Figure 4 shown in Section 3.2.5.2.3 illustrates the classes that are used to create function block applications:

Basic Function Block. This is the fundamental type used to create **Custom Function Blocks** from. Each function block can be customized to create specific input and output data ports, events, and processing algorithms.

Composite Function Block. Multiple Basic and Custom Function Blocks can be encapsulated into a single Composite Function Block type. Composite Function Blocks expose input and output data ports from their component function blocks. Events are sourced and sunk from the individual function blocks contained within the Composite Block. This approach to function block design encourages modularity through encapsulation, promoting re-usability.

Service Interface Function Blocks provide a mechanism for interfacing to physical devices such as sensors and actuators. They also provide input and output data ports, events, and processing algorithms. They often link in custom code libraries to facilitate interactions with devices.

Diagnostic Point Function Blocks are specialized Service Interface Function Blocks that provide a bridge between the agents in the engine and the data and event streams within the function block application. They were designed specifically to work with the agents during the development of the engine.

Function Block Runtime Library. All Basic, Composite, Service Interface, Custom and Diagnostic Point function block types are compiled into the runtime library. The runtime library also contains utility and support classes that provide services to the function block instances.

Function Block Application. Function block applications are constructed by connecting networks of function blocks together. The data and event inputs and outputs for a function block can be connected to another function block to allow them to exchange data, triggered by events. Function block applications can be partitioned or *distributed* so that parts of them run on across multiple, separate devices. These

devices then work co-operatively as if they were a single, homogeneous application.

3.2.5.2.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.2.5.2.2.3 Interfaces: The engine maintains interfaces to external systems through network connections. Discrete physical devices are accessed using pre-defined protocols applicable to each device. Usually, these interfaces are mediated via Service Interface Function Blocks.

3.2.5.2.2.4 Behavior: IEC 61499 Function Block Applications are event-driven as defined in the IEC 61499 Standard [1].

3.2.5.2.2.5 Constraints: Their are no additional constraints within this view packet.

3.2.5.2.3 Context diagram:

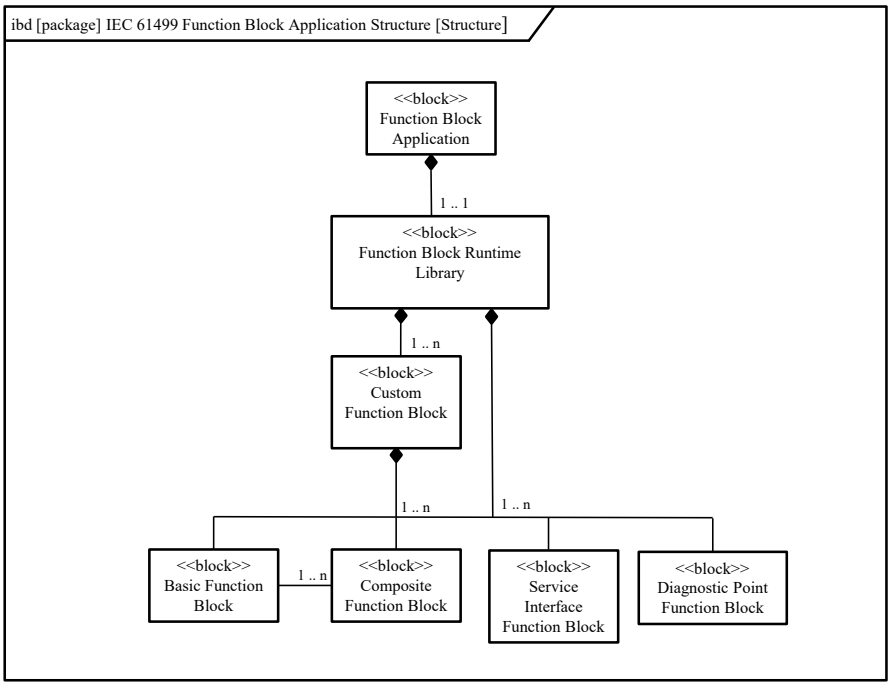

Figure 4: IEC 61499 Function Block Application Context View.

3.2.5.2.4 Variability mechanisms: There are no variabilities in this view packet. 4diac and nxtStudio implement function block types and libraries that may differ internally while still complying with the IEC 61499 standard.

3.2.5.2.5 Architectural background: The Diagnostic Point is the primary architectural feature that allows the agents to exchange telemetry with the function blocks in the function block application.

3.2.5.2.6 Related View Packets: The interface between a function block application and the engine is discussed in Section 3.2.5.1.6.

3.3 Process View

3.3.1 View Description: This section presents a set of view packets that detail the processes that run within the engine and its agents while it is hunting for faults. It includes details of the configuration and re-wiring process that the agents execute to create diagnostic points and the test harness. This view is presented from the Information and Concurrency viewpoints.

3.3.2 View packet overview: This view contains three View Packets. It presents the processing of the function block application belief structures, the configuration re-wiring processes, and the interactions between the agents and the diagnostic points.

3.3.3 Architectural background: The belief structures are established and maintained while the agent is configuring and interacting with the function block application. The belief structures are architecturally significant since they address both domain-specific features of IEC 61499 and the constraints that distributed function block applications place on the ways the engine can exchange data with function blocks. A client-server architectural pattern was used to model the way agents and the diagnostic points maintain connections and manage concurrency.

3.3.4 Variability mechanism: The processes that the engine executes are the same for both 4diac and nxtSTUDIO. Differences in the deployment and re-wiring processes are documented where relevant in the View Packets that follow in this view.

3.3.5 View packets:

3.3.5.1 View Packet 1 - Function Block Application Belief Structures

3.3.5.1.1 Primary presentation: Function Block Applications are created in IDEs such as 4diac and nxtSTUDIO. Each application is stored in an XML-format file that details the function block instances and the connections between them. While this format is ideal for development systems to use, it is not in a format that the agents can use easily during fault-finding. Information about each function block is recorded in different places in the application file depending on the context of the information needed by the IDEs.

The engine addresses this by first processing the application file, accumulating each item of function block data as it is encountered. The file is normalized by encoding each attribute of a function block instance into a discrete object. The list of function blocks and their connection form a directed graph structure where each node represents a function block and each edge details a data or event connection. During fault-finding, the agents can more easily navigate this structure, loading all the details of a function block instance in a single method call [32].

3.3.5.1.2 Element catalogue:

3.3.5.1.2.1 Elements: Figure 5 shown in Section 3.3.5.1.3 illustrates the processes that analyze the function block application definition file and create the in-memory belief structure used by the agents:

currentCommand: This object node starts the function block application load process. The command is provided in the Goal data supplied to the agent when the task is assigned to them. The command data includes the fully-qualified path to the folder where the function block application definition file is located. This file is an XML-format file with the extension .sys.

parseFile: Loads and parses the application file, verifying that the XML elements are well-formed. This is a belief-type skill executed by the agent. If an error is encountered, the details are reported back to the agent by creating an error belief.

Analyse Function Block: This loop processes each function block as it is encountered in the file. The FunctionBlockApp structure illustrated in Figure 3 in Section 3.2.5.1.3 contains a collection of FunctionBlock instances.

Several error scenarios are possible in this loop. Each function block type is defined in a separate function block type file. This file is named with the function block type and has a .fbt file extension. An error belief is returned if the function block type cannot be found or is not well-formed.

Parse Diagnostic Package: Function blocks may also have diagnostic packages available. These are contained in discrete XML-format files with the extension .dpg where the file name matches the function block type. Note that is is not an error if no matching diagnostic file is present in the application folder.

Diagnostic packages contain two information sections. The first lists each diagnostic test point on the function block. Diagnostic point information is used to create the structure DiagnosticPoints documented in Section 3.3.5.1.3, Figure 5. The second information section contains diagnostic test routines that the agent can execute against those diagnostic points to exercise the function block. These routines are used to create a Java class that is loaded and compiled dynamically at runtime for later use during diagnostic sessions.

3.3.5.1.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.3.5.1.2.3 Interfaces: The FunctionBlockApp belief structure is private to team of agents that has been assigned to monitor and diagnose the function block application. Note that only one agent in the team should be assigned to create the belief structure. A handle to the shared FunctionBlockApp can be assigned by the DiagnosticTeam allowing the structure to be shared by all agents in the team. Where applications are distributed, the scope of the FunctionBlockApp structure is the sub-application that has been assigned to the team of agents.

3.3.5.1.2.4 Behavior: IEC 61499 function block applications are designed to be interchangable between development IDEs.

3.3.5.1.2.5 Constraints: Their are no additional constraints within this view packet.

3.3.5.1.3 Context diagram:

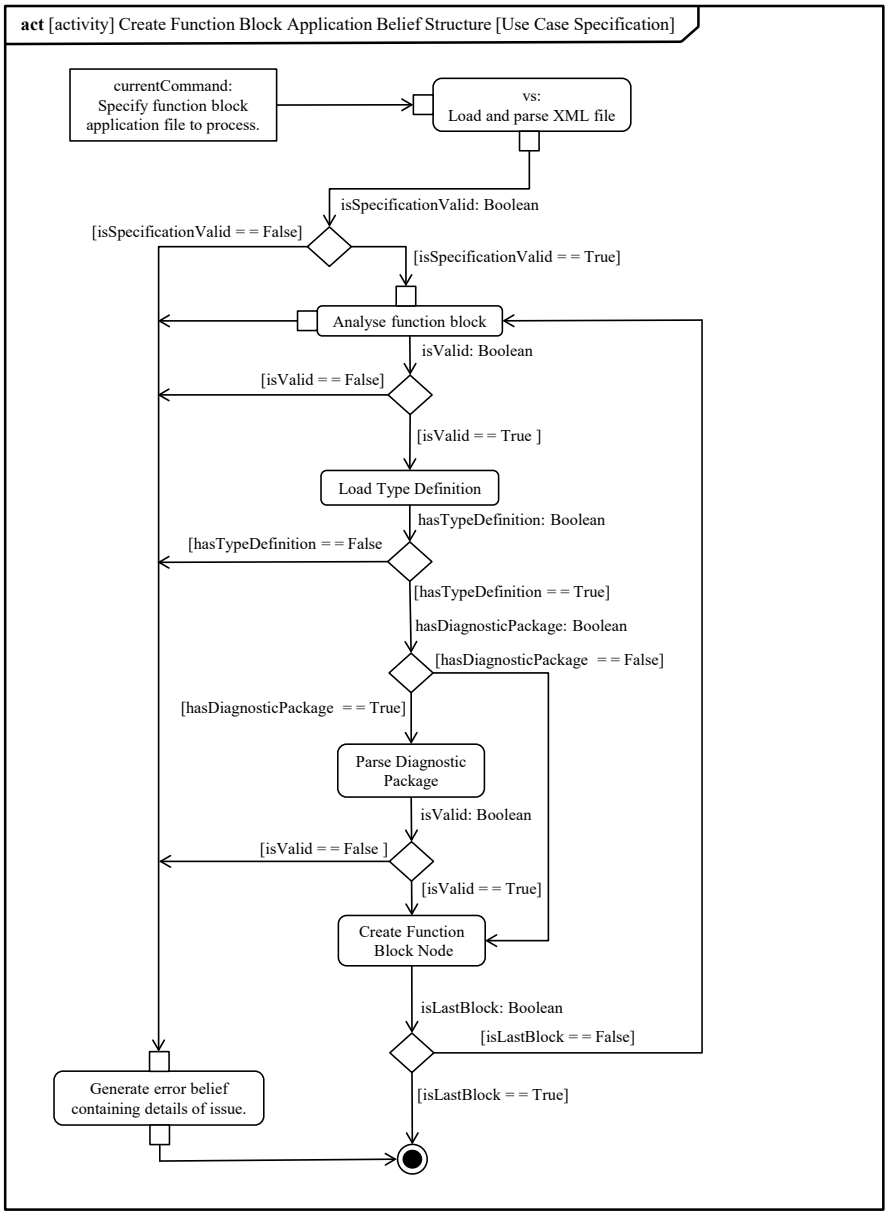

Figure 5: Process View: Creating the Function Block Application Belief Structures

3.3.5.1.4 Variability mechanisms: There are no variabilities in this view packet.

3.3.5.1.5 Architectural background: The FunctionBlockApp object contains an indexed list of FunctionBlock objects. This allows diagnostic programs such as the re-wiring functionality to access each function block node instance by establishing a reference to it. This contributes to the Modifiability QA, making it easier to add new properties to a FunctionBlock node without extensive code modification.

3.3.5.1.6 Relation to other view packets: There are no related view packets except for the FunctionBlockApp structure illustrated in Figure 3 in Section 3.2.5.1.3. That view packet illustrates the hierarchy of the different belief structures managed by the agent.

3.3.5.2 View Packet 2 - Creating the Diagnostic Harness

3.3.5.2.1 Primary presentation: The FunctionBlockApp and the DiagnosticPoint structure profiled in Figure 3 in Section 3.2.5.1.3 provide the information needed to create diagnostic function blocks within the function block application. These are inserted into the data and event paths dynamically to create a test harness.

3.3.5.2.2 Element catalogue:

3.3.5.2.2.1 Elements: Figure 6 shown in Section 3.3.5.2.3 illustrates the classes that are used to create the diagnostic test harness for the function block application:

currentCommand: This object node starts the re-wiring process. The command is provided by the agent when the task is assigned to them.

createHarness is a domain-specific skill possessed by the agent. Each function block in the FunctionBlockApp structure is examined to see if diagnostic information was found during the processed outlined in View Packet 1 in Section 3.3.5.1.

Create DP type Function Block. Diagnostic Points are specialized Service Interface Function Blocks that can interact with the data and event flows. Once created, the existing data and event flows are re-routed through the diagnostic points using the **Rewire input and output ports and events** methods.

Deploying the harness. The agent knows which runtime environment the diagnostic harness is being created for. It deploys either a 4diac forte.fboot file or a modified nxtSTUDIO .sys file. Refer to Section 3.3.5.2.4 for further information about this variability mechanism.

3.3.5.2.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.3.5.2.2.3 Interfaces: There are no additional external interfaces to this view packet.

3.3.5.2.2.4 Behavior: The methods executed while creating the diagnostic harness share a common ErrorHandler object which is returned to the agent at the end of the process. If errors occur during the rewiring, the details are recorded in an agent belief that is presented to the users at the end of the diagnostic test run.

3.3.5.2.2.5 Constraints: Their are no additional constraints within this view packet.

3.3.5.2.3 Context diagram:

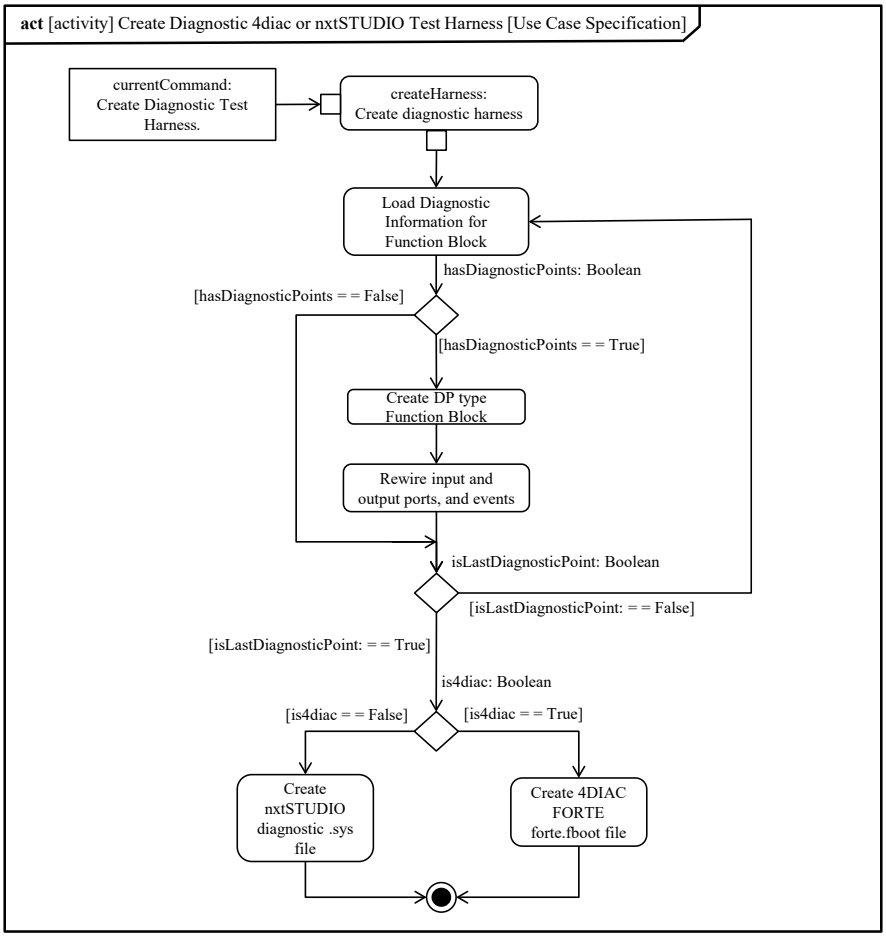

Figure 6: Process View: Create Diagnostic Test Harness for 4diac or nxtSTUDIO.

3.3.5.2.4 Variability mechanisms: The variabilities in this view packet relate to the differences in the runtime constraints of the 4diac and nxtSTUDIO diagnostic

test harnesses. 4diac function block applications compile their custom function blocks and all resources into a single runtime executable file called *forte*. When the application is launched, it processes a configuration file called *forte.fboot* located in the same directory. This allows the runtime to dynamically wire together all the function blocks and then start the function block application. The diagnostic harness created by the engine is a modified version of forte.fboot that seamlessly wires in the diagnostic point function blocks.

In contrast, nxtSTUDIO generates a single executable that statically links the function blocks at compile time. This requires the engine to generate a diagnostic version of the .sys application file that is then compiled by nxtSTUDIO. The original .sys application file is preserved.

3.3.5.2.5 Architectural background: The differences in the diagnostic harness configurations outlined in Section 3.3.5.2.4 are architectural issues that only affect this view packet. Once the diagnostic harness is created, the fault finding processes performed by the engine are identical. The C++ code that is required for 4diac diagnostic point function blocks differs significantly from the C# code used by nxtSTUDIO. This is discussed in more detail in Section 3.4.

3.3.5.2.6 Related View Packets: There are no related view packets in this view.

3.3.5.3 View Packet 3 - Agent interactions with Diagnostic Points.

3.3.5.3.1 Primary presentation: This view details the interaction between the agent and the diagnostic points that are being used to investigate one or more function blocks of interest. The agents have a range of strategies they can use to identify the location of a fault and then categorize it.The view is presented from the Information and the Concurrency Viewpoints.

3.3.5.3.2 Element catalogue:

3.3.5.3.2.1 Elements: Figure 7 shown in Section 3.3.5.3.3 illustrates the communication flow between a representative agent and two diagnostic points. Two function blocks are used to illustrate the flow of data and events.

Agent. Each agent marshals one or more diagnostic points that they have deployed into the function block application via a diagnostic test harness. The creation of the diagnostic harnesses is profiled in View Packet 2 in Section 3.3.5.2.

Diagnostic Points. Diagnostic Points are custom function blocks that manage their own TCP/IP client connection with the NIOserver running in the engine. The diagnostic point on the far right of the diagram is capturing the output of the second application function block.

Function Blocks. Section 3.3.5.2 discusses the connection between the diagnostic

point function blocks and the application function blocks whose data and event streams are being examined.

3.3.5.3.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.3.5.3.2.3 Interfaces: No additional interfaces are represented in this view packet.

3.3.5.3.2.4 Behavior: The agent operates in two distinct modes:

Monitor Mode is the default state of diagnostic function blocks when the function block application starts. All application data and events are passed through transparently while relaying information to the agent.

Trigger and Capture Mode allows an agent to take control of a section of the function block applicaton so that it can inject representative test values into the data and event flow. The *gate* commands allow the agent to instruct the diagnostic points to block the flow of data and events into the region of interest.

3.3.5.3.2.5 Constraints: Their are no additional constraints within this view packet.

3.3.5.3.3 Context diagram:

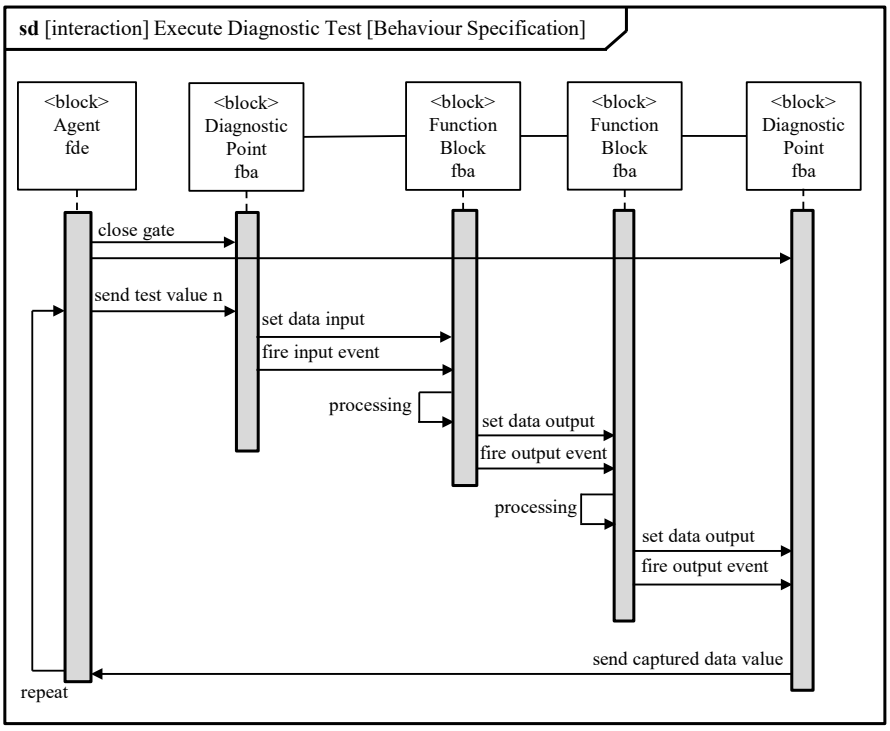

Figure 7: Process View: Exchanging data between the agent and the diagnostic points.

3.3.5.3.4 Variability mechanisms: There are no variabilities in this view packet.

3.3.5.3.5 Architectural background: The interaction between the agents and the diagnostic points is a unique feature of the engines architecture. The diagnostic function blocks are designed to be portable across the 4diac and nxtSTUDIO implementations. They only require different internal libraries to implement their TCP/IP client components.

3.3.5.3.6 Related View Packets: There are no related view packets for this view.

3.4 Development View

3.4.1 View Description: This view describes the configuration of the application development systems used to create the engine. It includes a description of the IEC 61499 IDEs that were used to develop representative test function block applications

that exercise each aspect of the engines functionality.

3.4.2 View packet overview: This view contains a single View Packet.

3.4.3 Architectural background: GORITE is written in Java. Writing the agent code in Java contributed to the performance of the system since no interoperability layer was required. The diagnostic point function blocks are written in C++ and this also provides a degree of portability across the 4diac and nxtSTUDIO runtime platforms.

3.4.4 Variability mechanism: The only point of variability is the language used to write the custom libraries for communications for the diagnostic point function blocks. The 4diac FORTE runtime is written in C++ while nxtSTUDIO relies C# code.

3.4.5 View packets:

3.4.5.1 View Packet 1 - The Development Environment.

3.4.5.1.1 Primary presentation: The Eclipse development environment for Java was used in conjunction with the 4diac and nxtSTUDIO function block development tools.

3.4.5.1.2 Element catalogue:

3.4.5.1.2.1 Elements:

Eclipse is the Java development environment and IDE used to create and debug the fault diagnostic engine source code. It is also responsible for creating the C++ and the C# libraru components used by the diagnostic point function blocks.

4diac 4diac manages the creation of the custom function blocks as well as the configuration of the function block application .sys file. The GCC C++ development toolchain is used to compile the function block source.

nxtSTUDIO nxtSTUDIO manages the creation of custom function blocks and the configuration of the function block application file. The C# compiler is used to create the runtime executable program that implements the function block application.

3.4.5.1.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.4.5.1.2.3 Interfaces: The interfaces between the development tools are mediated by files generated by the IDE. Some of these code resources are passed into the GCC C++ toolchain or the C# toolchain for compilation and linking.

3.4.5.1.2.4 Behavior: The behavior of these systems is inherently that of the IDE designers for all three development tools. They differ in screen layout, key bindings

and development flows.

3.4.5.1.2.5 Constraints: GORITE is written in Java. This constrained the development of the agent code to also be in Java. However, Java is portable across a number of different platforms so this factor contributes to the Quality Attributes for portability. However nxtSTUDIO only runs on Windows platforms.

3.4.5.1.3 Context diagram:

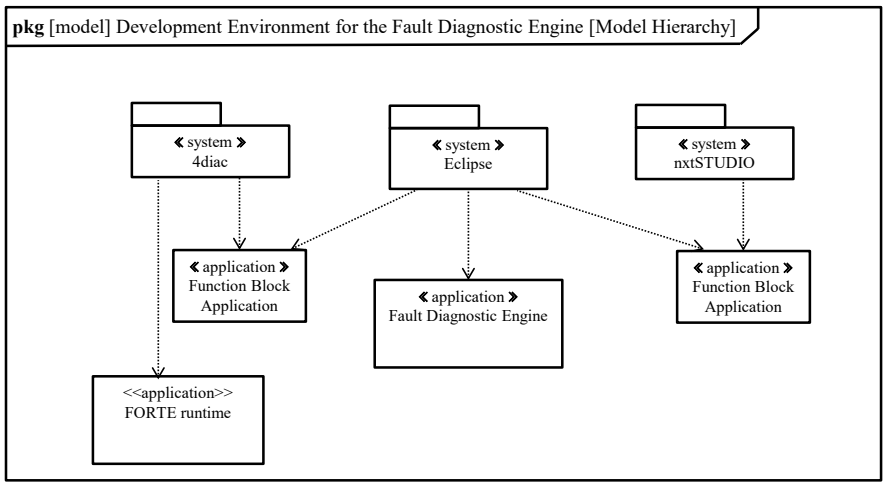

Figure 8: Development View: Fault Diagnostic Engine Development Environment.

3.4.5.1.4 Variability mechanisms: The variabilities in this view relate to the diagnostic point function blocks that are developed for the 4diac and nxtSTUDIO runtimes.

3.4.5.1.5 Architectural background: The Eclipse development environment was extended with the LiClipse C++ development editor so that a single IDE could be used to create both the engine and the custom C++ code for the diagnostic points.

3.4.5.1.6 Related View Packets: View Packet 3.3.5.2 details the differences in the creation of the diagnostic harnesses for 4diac and nxtSTUDIO.

3.5 Physical View

3.5.1 View Description: This section presents a set of view packets that outline the way the engine is configured for use in the field after it has been developed. These views are presented from the Deployment and Operational viewpoints.

3.5.2 View packet overview: This view contains a single View Packet that shows both the local and distributed configurations of the engine.

3.5.3 Architectural background: Function block applications are cyber-physical systems that can operate as either single, monolithic executables or they can spread their functionality across multiple distributed instances. The ability to perform in both scenarios is part of the core engine architecture and functionality, detailed in the previous views. The current version of the engine is designed to run as a single instance, hosting multiple agents who reach out to distributed instances of the function block application via network connections.

3.5.4 Variability mechanism: There is no variability across these scenarios. The engine can be configured to run on any platform that provides a Java runtime.

3.5.5 View packets:

3.5.5.1 View Packet 1 - Engine Deployment Configuration

3.5.5.1.1 Primary presentation: Figure 9 shown in Section 3.5.5.1.3 illustrates the system instances required to support the agents and the function block application during fault-finding.

3.5.5.1.2 Element catalogue:

3.5.5.1.2.1 Elements:
Operator. The operators of the engine are the stakeholders listed in Section 1.5.8.

Fault Diagnostic Engine is the instance of the engine hosted either on the same platform as the function block application or on its own platform. Deploying the engine on its own platform addressed the QAs of reliability and performance when the function block application is exhibiting faults.
indexFault Diagnostic Engine see Engine

Function Block Application. Figure 9 shown in Section 3.5.5.1.3 shows the function block application running configured as either a single application or as multiple, distributed instances of the application.

3.5.5.1.2.2 Relations: There are no additional relations between the elements documented in this view packet.

3.5.5.1.2.3 Interfaces: The interfaces between these systems include the HMI used by the operator to communicate with the engine and the network connections between the engine and the function block application instances.

3.5.5.1.2.4 Behavior: Each system operates asynchronously, exchange data via their

network connections.

3.5.5.1.2.5 Constraints: Their are no additional constraints within this view packet.

3.5.5.1.3 Context diagram:

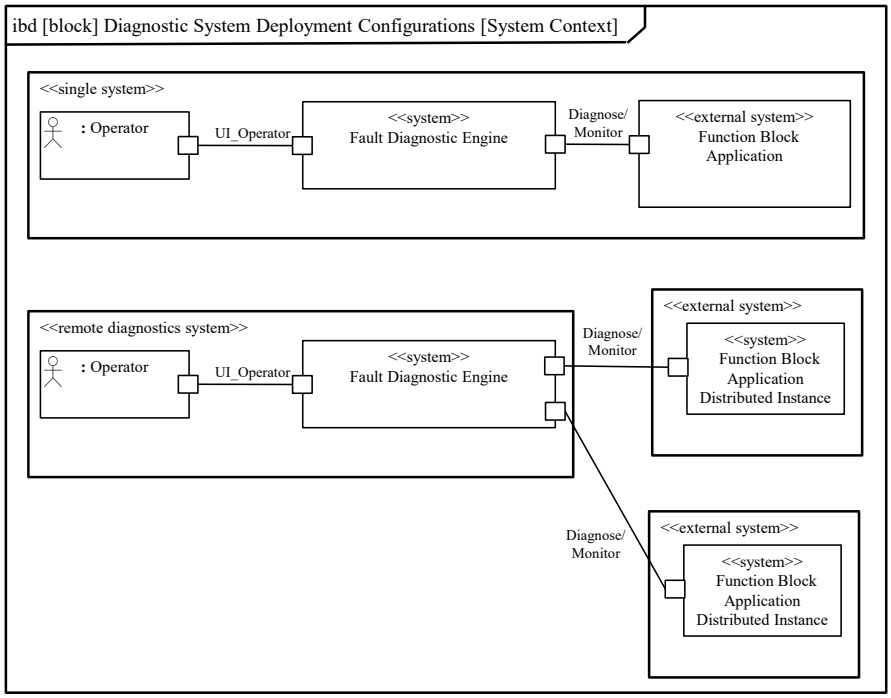

Figure 9: Physical View: Function Block Engine Deployment Configurations.

3.5.5.1.4 Variability mechanisms: The ability to run the engine separately from the function block application also provides a way for diagnosing multiple, distributed function block instances while co-ordinating agent activities from a central place. It may also be possible to connect multiple GORITE instances together to diagnose larger configurations.

3.5.5.1.5 Architectural background: There are no additional architectural features illustrated in this view packet.

3.5.5.1.6 Related View Packets: There are no related view packets in this view.

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective on the design of the fault diagnostic engine. Each is valid and useful in its own right. While providing these different system perspectives from differing viewpoints, they are not independent. Elements in one view may be related to those in different views. It is important to reason about these relationships. In general, mappings between views are many-to-many.

This section describes the relations that exist among the views given in Section 3. As required by IEEE 42010 [4], it also highlights any known inconsistencies among the views.

4.1 General Relations Among Views

4.2 View-to-View Relations

5 Referenced Material

[1] I. E. Commission *et al.*, "Function blocks–Part 1: Architecture," *International Electrotechnical Commission, Geneva, Switzerland, Tech. Rep. IEC*, 2013.

[2] Carnegie Mellon, "Views and Beyond: The SEI Approach for Architecture Documentation," 2018. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513862

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Third Edition*. Addison-Wesley, 2013.

[4] ISO, "ISO/IEC 42010:2007, Systems and software engineering Recommended practice for architectural description of software-intensive systems," *ISO/IEC/IEEE 42010-2007*, pp. 1–37, July 2007.

[5] IEEE, "IEEE Std 1471-2000 Recommended Practice for Architectural Description of Software Intensive Systems," *ISO/IEC/IEEE 1471-2000*, Nov 2000.

[6] B. Dowdeswell, R. Sinha, and S. G. MacDonell, "Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems." in *IECON 2020 - 46th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2183–2188.

[7] P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

[8] Rozanski, Nick and Woods, Eóin, *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2011.

[9] N. Rozanski and E. Woods, "Viewpoints and Perspectives on-line resources," 2019. [Online]. Available: https://www.viewpoints-and-perspectives.info

[10] SYSMOD, "How to model an Extended System Context using SysML," 2019. [Online]. Available: https://mbse4u.com/2012/08/06/how-to-model-an-extended-system-context-with-sysml/

[11] E. Woods and N. Rozanski, "The system context architectural viewpoint," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, 2009, pp. 333–336.

[12] N. R. E. Woods and N. Rozanski, "Software Systems Architecture," 2005.

[13] H. Choi and K. Yeom, "An approach to software architecture evaluation with the 4+ 1 view model of architecture," in *Ninth Asia-Pacific Software Engineering Conference, 2002.* IEEE, 2002, pp. 286–293.

[14] W. C. Scratchley and C. M. Woodside, “Evaluating concurrency options in software specifications,” in *MASCOTS'99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 1999, pp. 330–338.

[15] R. Hilliard, “Views and viewpoints in software systems architecture,” in *Position paper from the First Working IFIP Conference on Software Architecture, San Antonio*, 1999.

[16] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[17] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: Systems Modeling Language*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[18] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sünder, A. Valentini, and A. Martel, “Framework for distributed industrial automation and control (4diac),” in *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*. IEEE, 2008, pp. 283–288.

[19] nxtControl GmbH. (2020) The nxtSTUDIO Development Environment. [Online]. Available: http://www.nxtcontrol.com/en

[20] D. Jarvis, J. Jarvis, R. Rönnquist, and L. C. Jain, “Multi-Agent Systems,” in *Multiagent Systems and Applications*. Springer, 2013, pp. 1–12.

[21] M. Ganzha, L. C. Jain, D. Jarvis, J. Jarvis, and R. Rönnquist, *Multiagent Systems and Applications*. Springer, 2013.

[22] L. Chen, M. A. Babar, and B. Nuseibeh, “Characterizing architecturally significant requirements,” *IEEE software*, vol. 30, no. 2, pp. 38–45, 2013.

[23] O. I. de Normalisation, *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. ISO/IEC, 2011.

[24] 4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment, 2019. [Online]. Available: https://www.eclipse.org/4diac/

[25] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[26] H. Nyquist, “Certain topics in telegraph transmission theory,” *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

[27] Defense Advanced Research Projects Agency, “RFC 793 Transmission Control Protocol,” pp. 1–89, 1981.

[28] DARPA, “RFC 768 User Datagram Protocol,” pp. 1–3, 1980.

[29] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: a large-scale empirical study," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71.

[30] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture, on patterns and pattern languages*. John Wiley & Sons, 2007, vol. 5.

[31] D. Weyns, *Architecture-based design of multi-agent systems*. Springer Science & Business Media, 2010.

[32] B. Dowdeswell, R. Sinha, D. Jarvis, J. Jarvis, and S. G. MacDonell, "Employing Agent Beliefs during Fault Diagnosis for IEC 61499 Industrial Cyber-Physical Systems." in *IECON 2020 - 46th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2189–2194.

6 Directory

6.1 Index

6.2 Glossary

Agent An intelligent agent (IA) refers to an autonomous entity which acts, directing its activity towards achieving goals. Situated Agents operate in a physical environment, basing their activities on decisions made by observing through sensors and acting using actuators. Intelligent agents may also learn or use knowledge to achieve their goals.

Function Block The fundamental building block used to create IEC 61499 Function Block Applications.

IEC 61499 The international standard IEC 61499 [1] defines the function block architecture for industrial process measurement and control systems. It was initially published in 2005. The specification of IEC 61499 defines a generic model for distributed control systems and is based on the IEC 61131 standard.

Software Architecture The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [3]. "Externally visible properties" refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

Software Architecture Document Software Architecture Document. A document template developed by the Carnegie Mellon Software Engineering Institute (SEI) that can be used to document a software architecture.

View A representation of a whole system from the perspective of a related set of concerns standardized in ISO 42010 [4]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A View conforms to a defining Viewpoint.

View Packet The smallest package of architectural documentation that could usefully be given to a stakeholder. The documentation of a View is composed of one or more View Packets.

Viewpoint A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and au-dience for a view, and the techniques for its creation and analysis standardized in ISO 42010 [4]. Identifies the set of concerns to be addressed, and the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view.

Views and Beyond The Carnegie Mellon Software Engineering Institute's *Views and Beyond* method for documenting software architectures, as described in Bass and Clements [3]

6.3 Acronyms

ATAM Architecture Tradeoff Analysis Method

GORITE Goal ORiented TEams

IDE Integrated Development Environment

QAW Quality Attribute Workshop

SAD Software Architecture Document

SDE Software Development Environment

SEI Carnegie Mellon Software Engineering Institute

SysML Systems Modeling Language

UML Unified Modeling Language

Appendix B

The simbIoTe Environmental Simulator

The Heating Ventilation and Air-Conditioning (HVAC) function block application provided an ideal testing ground for the development of the diagnostic points. It also allowed the agents to be exercised with an IEC 61499 application that changed dynamically as it sampled temperature and other environmental changes. Before the real sensors and actuators were developed in Chapter 9, that interactive behaviour had to be simulated. The **simbIoTe** (**sim**ulator for **b**uilding **IoT e**nvironments) application was developed in-parallel with the fault diagnostic engine to fulfil that role. This section briefly outlines the operation and architecture of simbIoTe.

B.1 Simulation for function block applications

Simulators used in research usually adhere to one of two primary design paradigms. MATLAB is a general-purpose development environment and a programming language that is ideal for creating rich data sets that can be used by other applications. It is especially suitable for complex

mathematical simulations and matrix manipulation. In contrast, Simulink and COMSOL Multiphysics focus more on emulators that model hardware designs and the simulation of physical phenomena.

SimbIoTe was designed to provide elements of both these approaches. It is an Open Source Java-based simulator that can model and then run Human-Machine Interfaces (HMIs) for IEC 61499 applications. Figure B.1 shows part of the HMI created for the HVAC room simulator. SimbIoTe provided a panel with touch buttons that capture user interactions to request the room temperature to be changed. Figure B.2 shows the HVACsim simulation with multiple HMI control panels monitoring different rooms. HVACsim is also using the simbIoTe Environment module to model and run a simulation of the building that provides independent temperatures in each zone.

Figure B.1: HMI control panel

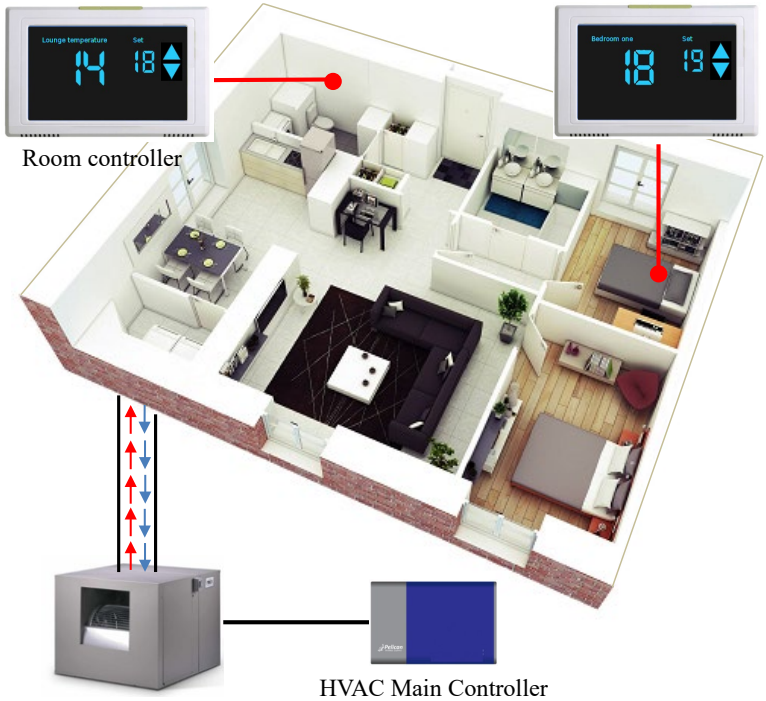

Figure B.2: HVAC room controllers in a *simbIoTe* emulated environment.

B.2 Architectural considerations

Figure B.3 illustrates the classes that implement a simbIoTe application. The HMIui classes implements a Java Swing User Interface that runs on its own thread. The HMI provides windowed elements that are assembled into highly interactive resolution-independent interfaces. In addition, simbIoTe provides a programmable emulation environment. in the HVACsim model, this is multi-threaded environmental simulator that emulates the temperatures of multiple rooms in a building. Algorithms generate realistic changes in the environment based on simple thermal equations with randomised changes. Within this environment, the physics of complete sensor components such as the AM2302/DHT22 temperature probe were modelled. These components form part of a library that includes emulation of their packet communication protocols and realistic errors that can be activated on command. A simple TCP/IP communications protocol was developed to allow IEC 61499 Service Interface Function Blocks (SIFBs) to exchange data directly with simbIoTe. The NIOserver class is similar to the class developed for the fault diagnostic engine.

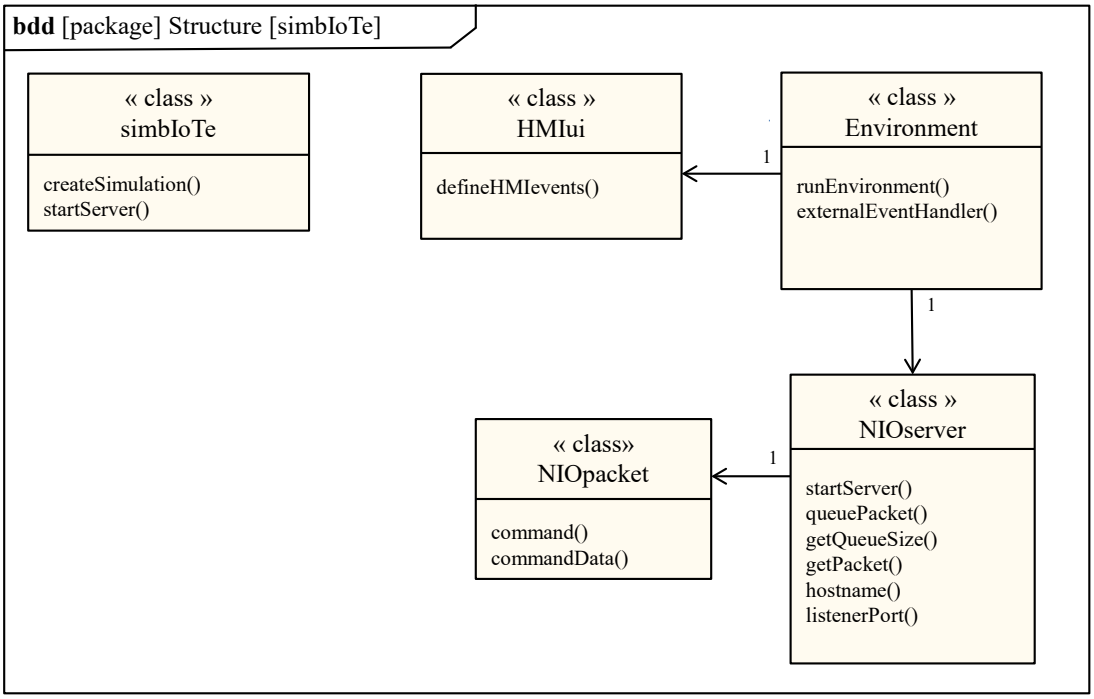

Figure B.3: simbIoTe Block Diagram showing the classes and their relationships.

B.3 Extending simbIoTe further

At present, the HMIs are coded manually to implement the Java Swing elements. The Open Source GUI design tool WindowBuilder (`https://www.eclipse.org/windowbuilder`) will be built into the next version of simbIoTe to make the creation of the HMIs for simulations quicker and more interactive. A way of encapsulating sensors and actuators is being designed that can enable them to be dynamically loaded into simbIoTe on-demand. This would make it easier to create and share libraries of IoT devices between users.

Appendix C

The Source Code

The source code for the Fault Diagnostic Engine and each of the projects discussed is available on the following public Github sites:

The Fault Diagnostic Engine	`https://github.com/badger-dowdeswell/Fault-Diagnostic-Engine`
Diagnostic Point code	`https://github.com/badger-dowdeswell/Diagnostic-Point-Function-Blocks`
The HVAC Room Controller	`https://github.com/badger-dowdeswell/HVACsim`
Smart Grid Protection IEDs	`https://github.com/badger-dowdeswell/Smart-Grid-Protection-IEDs`
simbIoTe	`https://github.com/badger-dowdeswell/simbIoTe`

Appendix D

The Unused Quotations

These are the precious quotations that did not end up being used in the chapters. Over the past four years, the quotations that were found while reading for this research were squirrelled away in my research log book. Many of them were swapped into and out of chapter headings multiple times until one felt like it belonged there. They are included here in the hope that they inspire you in the same way they have encouraged and supported all the late-night thinking that went into this thesis.

"It is the nature of an idea to be communicated: written, spoken, done. The idea is like grass. It craves light, likes crowds, thrives on cross-breeding, grows better for being stepped on."

"The Dispossessed", Ursula K. Le Guin (1974)

"Crude classifications and false generalisations are the curse of the organized life."

"*A Modern Utopia*", page 191, H.G.Wells (1905)

"That isn't the right question; and there are no right answers to wrong questions - as witness the works of those who attempted to discover the properties of Phlogiston."

Introduction to *"Planet of Exile"*, Ursula K. Le Guin (1964)

"The time will come when diligent research over long periods will bring to light things which now lie hidden. A single lifetime, even though entirely devoted to the sky, would not be enough for the investigation of so vast a subject ..."

"Natural Questions", Lucius Annaeus Seneca, circa 65 AD

"I did then what I knew how to do. Now that I know better, I do better."

Interview with Maya Angelou (2011)

"It is clear from the preceding that every 'art' [technique] has its speculative and its practical side. Its speculation is the theoretical knowledge of the principles of the technique; its practice is but the habitual and instinctive application of these principles. It is difficult if not impossible to make much progress in the application without theory; conversely, it is difficult to understand the theory without knowledge of the technique."

Diderot, "Arts" in Encyclopédie (1751-1765) quoted in Gregor and Hevner (2013).

"Write the vision; make it plain upon tablets, so he may run who reads it. For still the vision awaits its time; it hastens to the end - it will not lie. If it seem slow, wait for it; it will surely come, it will not delay."

The Book of Habakkuk, Chapter 2:2-3

"Writing is not some quiet, closet act. Oh, and please ... no matter how we advance technologically, please don't abandon the book. There is nothing in our material world more beautiful than the book."

Patti Smith, writer and musician (2020)

"The instinct of creativity must be followed by the act, the physical act of putting it down for a sense of permanence."

Rod Serling speaking at Ithaca College in 1960.

"The possession of knowledge does not kill the sense of wonder and mystery. There is always more mystery."

Anais Nin (1903-1977)

"You want weapons? We're in a library. Books are the best weapon in the world. This room's the greatest arsenal we could have. Arm yourself!"

Doctor Who speaking in the episode *Tooth and Claw*. Russell T. Davies (2005)

"For me, it is far better to grasp the Universe as it really is than to persist in delusion, however satisfying and reassuring."

The Demon-Haunted World: Science as a Candle in the Dark, Carl Sagan (1934-1996)

"Knowledge comes, but wisdom lingers."

Alfred Lord Tennyson (1809-1892).

"I came in with Halley's Comet in 1835. It is coming again next year (1910), and I expect to go out with it. It will be the greatest disappointment of my life if I don't go out with Halley's Comet. The Almighty has said, no doubt: 'Now here are these two unaccountable freaks; they came in together, they must go out together'."

Mark Twain (Samuel Langhorne Clemens) 1835-1910

"To learn which questions are unanswerable, and not to answer them: this skill is most needful in times of stress and darkness."

The Left Hand of Darkness Ursula K. Le Guin (1969)

"The hardest thing of all is to find a black cat in a dark room, especially if there is no cat."

attributed to Confucius

"I don't think about what people will learn from my books. I do hope that people will like my stories, that they will be entertained. I don't try to teach anything. I try to write engaging stories with interesting characters. I try to write good sentences. It's as simple and difficult as that.."

original version of the quotation by children's author Kevin Henks (2020).

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."

The Long Dark Tea-Time of the Soul, Douglas Adams (1988)

"A designer knows they have achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

Antoine de Saint-Exupery (1900-1944)