future internet

MDPI

Article

Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems

Barry Dowdeswell *, **Roopak Sinha** and **Stephen G. MacDonell**

Department of Computer Science and Software Engineering School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Private Bag 92006, Auckland 1142, New Zealand; roopak.sinha@aut.ac.nz (R.S.); stephen.macdonell@aut.ac.nz (S.G.M.)
* Correspondence: barry.dowdeswell@aut.ac.nz

Abstract: IEC 61499 is a reference architecture for constructing Industrial Cyber-Physical Systems (ICPS). However, current function block development environments only provide limited fault-finding capabilities. There is a need for comprehensive diagnostic tools that help engineers identify faults, both during development and after deployment. This article presents the software architecture for an agent-based fault diagnostic engine that equips agents with domain-knowledge of IEC 61499. The engine encourages a Model-Driven Development with Diagnostics methodology where agents work alongside engineers during iterative cycles of design, development, diagnosis and refinement. Attribute-Driven Design (ADD) was used to propose the architecture to capture fault telemetry directly from the ICPS. A Views and Beyond Software Architecture Document presents the architecture. The Architecturally-Significant Requirement (ASRs) were used to design the views while an Architectural Trade-off Analysis Method (ATAM) evaluated critical parts of the architecture. The agents locate faults during both early-stage development and later provide long-term fault management. The architecture introduces dynamic, low-latency software-in-loop Diagnostic Points (DPs) that operate under the control of an agent to capture fault telemetry. Using sound architectural design approaches and documentation methods, coupled with rigorous evaluation and prototyping, the article demonstrates how quality attributes, risks and architectural trade-offs were identified and mitigated early before the construction of the engine commenced.

Data Set: 10.17632/5skpwjh3sw.2

Data Set License: CC BY 4.0

Keywords: industrial cyber-physical systems; IEC 61499; function blocks; quality-driven architectures; fault diagnostics; multi-agent systems

Citation: Dowdeswell, B.; Sinha, R.; MacDonell, S.G. Architecting an Agent-Based Fault Diagnosis Engine for IEC 61499 Industrial Cyber-Physical Systems. *Future Internet* **2021**, *13*, 190. https://doi.org/10.3390/fi13080190

Academic Editors: Agostino Poggi, Ivana Budinská, Martin Kenyeres and Ladislav Hluchy

Received: 29 June 2021
Accepted: 16 July 2021
Published: 23 July 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Embedded Control Systems (ECSs), augmented with communications and sophisticated sensors, have led to the development of powerful new mechanisms known as Industrial-scale Cyber-Physical Systems (ICPS). ICPS control their electro-mechanical parts by using on-board processors and transducers, the sensors and mechanical actuators that enable them to sense and interact with their physical environment. This blending of their "cyber" software parts and their "physical" mechanical and environmental aspects extends them beyond the capabilities of earlier generations of ECSs. It is at this intersection, not the union, between the cyber and the physical that we must understand these entities more deeply [1].

This paper presents the software architecture of our Fault Diagnostic Engine, referred to in this paper as "*the engine*".

The scale, diversity and complexity of ICPS used in transportation, medical and manufacturing systems demands innovative approaches to fault diagnostics. The engine

is a stand-alone, multi-platform application that can identify and diagnose faults in applications created with IEC 61499 Function Blocks (FBs) [2]. The architecture employs software agents that are hosted within the GORITE (Goal ORIented TEams) Multi-Agent Framework [3–5]. The Belief-Desire-Intention (BDI) paradigm embodied in GORITE allows teams of agents to respond to changes in the FBs under examination, adopting a range of analysis strategies when faults are uncovered.

Current IEC 61499 Integrated Development Environments (IDEs) have limited design-time fault-finding capabilities. While some IDEs allow single FBs to be exercised, sets of connected blocks cannot be diagnosed together [6,7]. Engineers need to be able to quickly build semi-automated fault identification capabilities that can be reused throughout the life-cycle of an IEC 61499 Function Block application (FB application). The engine described here addresses that need by facilitating a Model-Driven Development with Diagnostics methodology that supports engineers while they are creating or reusing FBs to construct their ICPS with IEC 61499 [8].

This paper focuses primarily on the architectural design of the software agents and the way in which they have been provided with domain-specific knowledge of IEC 61499. This equips them with the abilities they need to interact with FB applications that are operating nominally, are failing, or have failed. GORITE provides a framework upon which to create new agent designs, but it does not include templates of pre-built agents. The architectural design of the agents, their skills and beliefs was therefore framed by four research questions:

RQ1: *How do we equip an agent with the ability to distinguish between normal function block activities and misbehaviour that indicates that a fault may have occurred?* A scoping survey on industrial-scale fault diagnostics for ICPS was published before the architectural design commenced [9]. This examined fault identification and diagnostic techniques across a range of domains where robust fault management has become critical. The findings that relate to the design of the engine are discussed in Section 2, including the underlying concepts of ICPS, function blocks, faults and agents.

RQ2: *How do we architect a software agent to observe the data that is flowing between function blocks so that it can recognise fault evidence?* Agents exhibit intentional behaviors that enable them to pursue goals and make decisions that modify their actions. To reason about faults, agents need to be able to capture information flowing between FBs. Section 3.4 explores the Diagnostic Point (DPs) that are a key feature of the engine, explaining how the agents autonomously deploy these custom FBs into an FB application to capture real-time fault telemetry.

RQ3: *How can a fault diagnostic engine that implements the features identified in RQ1 and RQ2 be architected?* Section 3 profiles the architecture in detail, showing how the Views and Beyond methodology [10] was used to guide and document the design.

RQ4: *How do we evaluate the architectural decisions made to satisfy RQ1 and RQ2?* The ATAM evaluation and the prototyping of key parts of the engine are discussed in Section 4. These processes refined both the agents themselves and the way they exchange information with the FBs being examined. This led to a more robust and versatile architecture. The ATAM presents scenarios that show how the features of the architecture satisfy both the functional and quality requirements for the engine. It demonstrates how the risks, sensitivities and trade-offs uncovered drove the architectural design choices that were made.

This paper illustrates how Attribute-Driven Design (ADD) [11] and the Views and Beyond methodology were coupled with prototyping to architect the engine through iterative design steps. By creating a range of complementary artefacts that present features as distinct *views*, architects make it easier for stakeholders to understand the reasons why features should be implemented in the ways proposed. Thinking about non-functional requirements as Quality Attributes (QAs) that can be evaluated empirically led to the creation of a more robust architecture. The complete Software Architecture Document (SAD) is available on Mendeley Data [12] to complement the exploration of the architecture presented in this article. Each recommended section of the Views and Beyond document

template was carefully replicated as an exemplar to help other practitioners. The resulting document illustrates the scope and maturity of ADD and Views and Beyond, which has continued to guide the on-going development of the engine.

The primary contribution of this paper is its presentation of the architecture for a fault detection engine in a well-structured Software Architecture Document that draws on recognized industry standards. It describes the way the agents apply domain-specific knowledge during fault finding for ICPS. When agents are deployed by the engine, they first explore the IEC 61499 application definition files of the ICPS that will be diagnosed. They then build sets of beliefs about the FBs and how they are connected to each other. This creates a navigable in-memory representation of the FB application that captures what they believe. This later guides their search through fault paths while they determine if each FB is performing nominally. The agents deploy DPs by wiring them dynamically into the data and event streams of a FB. Each DP operates as a native FB, passing data transparently through itself while also relaying that data back to the agent that is monitoring the DP. When commanded by an agent, sets of DPs can isolate or *gate* sections of the FB application. The agents then examine either individual or logical groups of FBs that are exhibiting fault symptoms, exercising them with pre-defined diagnostic tests. The DPs exhibit low-latency and impose a minimal overhead on the FB application. This approach illustrates how agents can dynamically investigate systems, reconfiguring the control layer from the agent's execution layer. This represents a significant enhancement of the interaction scheme employed in Kalachev et al. [13], Jarvis et al. [14] and Christensen [15]. In their approaches, execution layer agents only initiated function block controlled actions. Our approach was chosen to satisfy the quality attributes that the presence of DPs should not adversely impact the normal operation of the individual FBs. Hence, the agents are responsible for managing the fault-finding activities while the primary role of the DPs is to gather telemetry and inject test values into function blocks.

2. Background

The design of fault diagnostic engines has to address many of the challenges common to ICPSs [16]. By the start of the 1960s, the first general-purpose ECSs had been developed for General Motors' automotive assembly lines [17,18]. Referred to as Programmable Logic Controllers (PLCs), they controlled individual machines in isolation. Industrial plants rapidly became *systems-of-systems* [19], with connected production planning and management that demanded up-to-date information from the factory floor. Increasingly sophisticated manufacturing processes required more complex PLC capabilities. However, connecting multiple PLCs together and co-ordinating them was in itself complex. The term Cyber-Physical System (CPS) originated in the work of Bagheti and Gill in 2006 to describe the new device architectures that were emerging to network devices together to better control their environments [20]. As the scale of CPS architectures expanded, the term Industrial Cyber-Physical System (ICPS) was adopted to distinguish them from simpler consumer CPSs [21].

ICPSs bridge the divide between their "cyber"computational parts and their "physical" real-world environments. At this boundary, the discrete behavior of computational programs has to co-exist with the non-discrete, time-critical behaviors that drive real-world activities [22]. Figure 1 shows a typical entity in an ICPS, a Franka Emika Panda Collaborative Robot [23]. Collaborative Robots or "cobots" often share production line assembly operations with humans. Cobots perform a wide range of pre-programmed tasks but can autonomously modify their behavior in real-time. They detect the presence of nearby objects in their work area and capture feedback from their environment. They make control decisions, often exchanging telemetry over networks. Sensors convert physical characteristics such as orientation, proximity and velocity into electrical signals that can be processed by their control computers [24].

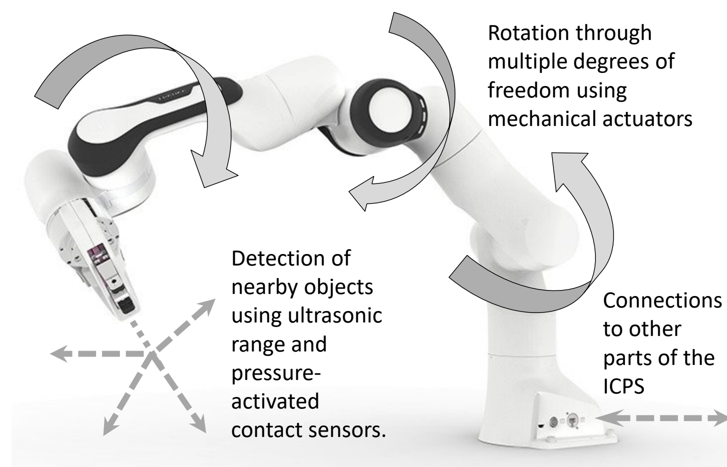

Figure 1. A Franka Emika Panda Collaborative Robot operating as part of an ICPS.

Mechanical actuators are transducers that enable an ICPS to move physical objects in its environment [1]. Motors are special classes of actuators that create translational or rotational movement. They allow cobots to rotate through multiple degrees of freedom with high precision. At all times, cobots have to honor hazard prevention protocols when working in close collaboration with humans or other cobots. This ability to operate in safety-critical environments is a core requirement demanded of many ICPSs. Alur comments that the emergence of distributed ICPSs and CPSs is far more ubiquitous [25]. The same cyber-physical concepts that implement large-scale ICPSs also apply to smaller, less complex consumer and medical CPSs.

2.1. The IEC 61499 Function Block Reference Architecture

At present, no single reference software architecture for the creation of ICPSs dominates. The capabilities of PLCs grew rapidly as industry began to find innovative ways of automating manufacturing operations. That demanded more robust, fault-tolerant software architectures to run PLC applications on. The IEC 61499 standard introduced in 2005 addresses the limitations of the earlier IEC 61131 standard [26] by providing an object-oriented, event-driven architecture that scales well [2]. Figure 2 shows a custom FB that has been developed from a standard Basic Function Block (BFB) type definition. This FB processes a temperature reading in degrees Fahrenheit that is received from another FB that interfaces with a temperature sensor. The IEC 61499 standard defines a convention that *Data Inputs* such as `TEMP_F` are drawn on the left side of the function block symbol. When new data have been made available from another connected FB to the input `TEMP_F`, the *Input Event* `REQ` is triggered. Each function block implements its own Moore-type finite state machine [27]. The *Output Events* and *Data Outputs* shown on the right of the FB symbol depend only on the current state and the values of the Data Inputs it has received. The IEC 61499 *Execution Control Chart* (ECC) [28] shown in Figure 2 details the states and state transitions possible for this FB. 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 an FB facilitates loose-coupling and scalability. However, event-driven systems present issues when interactive diagnostic approaches are used since the actions of the diagnoser can interfere with the timing of the ICPS.

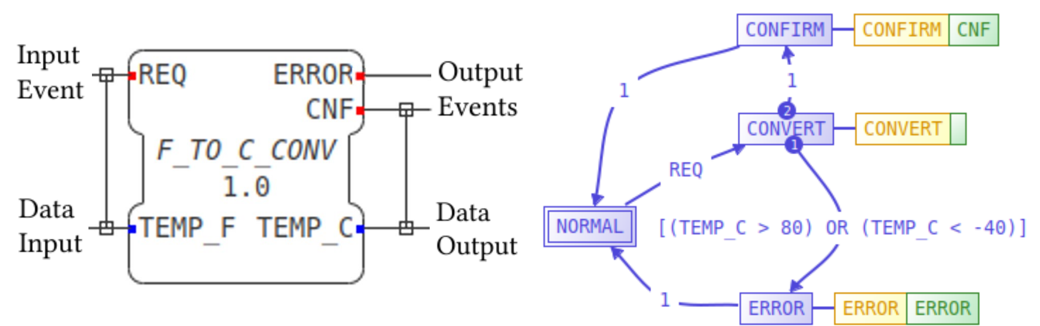

Figure 2. IEC 61499 Function Block that converts temperatures with its ECC.

Hehenberger comments that one of the advantages of ICPSs is the cost saving that modularity, adaptability and re-usability offer when implementing larger, distributed systems [29]. IEC 61499 provides a way to encapsulate capabilities in reusable units that are wired together to create event-driven logic and control. Encapsulation encourages component re-use by enabling developers to craft new FB applications from well-proven resources. The standard function block type definitions provides templates that are used to craft specialized FBs from. The custom Service Interface Function Block template interfaces well to physical sensors and actuators. In this way, trusted libraries evolve as generations of FBs share and refine field-proven resources. IEC 61499 also supports partitioning of applications to work collaboratively on separate, distributed platforms, providing scalability and redundancy. The IEC 61499 architecture has gained wide acceptance, being used in PLCs for industrial automation [30] and in the Intelligent Electronic Devices (IEDs) of Smart Electricity Grids [31,32].

2.2. Fault Identification and Diagnosis in ICPS

ICPS demand sophisticated fault management if they are to operate for the maximum possible uptime without interruption [33,34]. Self-managing fault diagnostic engines now ensure the reliability of interplanetary spacecraft and smart factories, often in situations where humans are not able to be present when things go wrong [35,36]. A *fault* is defined as any change in the operation of an ICPS that leads to unacceptable behavior or degraded performance [37]. Diagnostic strategies must enable fault-management software to be able to recognise what constitutes both normal and abnormal behavior by the devices that operate in the ICPS. If the cobot in the example above pauses because it cannot complete a task when an obstruction is blocking its path, that is not a fault. It is the ICPS adapting appropriately to a change in its environment. However, failing to halt is a fault, possibly caused by a damaged sensor or a programming error.

Sophisticated smart sensors contain in-built electronics to pre-process readings and transmit them to the ICPS they are connected to. Faults in such sensors can include failures in electronic components, both intermittent and permanent, as well as calibration errors that cause them to return false readings. Actuators such as those that move aircraft flight control surfaces rely on feedback from multiple position sensors to determine if they are orientated correctly. Since many sensors and actuators have their own embedded processors, software faults can lead to operational failures. The recent failure of the airspeed and angle of attack sensors on the Boeing MAX 8 led to the incorrect operation of the on-board MCAS flight stability software, resulting in catastrophic behavior of the aircraft [38]. Zolghadri et al. [39] note how the number of sensors being monitored for faults affects both the design and worst-case performance of diagnostic approaches. Dearden et al. [40] comment that many ICPS devices are designed for low power consumption and have limited computational capabilities. Hence, external fault diagnostic systems rather than built-in fault-finding ensures that these ICPSs are not overloaded by fault management and diagnostic processes.

Physics-Based Modeling of faults relies on models of behavior that can be used to recognise misbehavior. Model-invalidation techniques recognize deviations in the ICPS telemetry from the values predicted in the model [41,42]. Noise discrimination is

required since the differences may be small when the device is appearing to perform nominally [43,44]. However, creating models is labor-intensive and they require careful calibration to enable them to determine when the ICPS has deviated from expected behaviors [41,45].

In contrast, Model-Free approaches use Artificial Intelligence (AI) where the correct operation of the ICPS is learnt by observation during a training period. While the effort to create a pre-defined model is not required, fault engines that rely on AI techniques must be trained on a system that is able to demonstrate what nominal operation is. In the prior scoping survey [9], variations of Model-Based approaches in the aerospace sector were encountered in 85% of the studies while AI featured in 30% of the studies. Hybrid techniques that combine aspects of both approaches were found in 36% of these studies. This predominance of Model-Based techniques reflects a conservative approach typical of safety-critical environments.

Christensen [15] proposed a model of fault diagnosis applicable to IEC 61499 that extends the Model-View-Controller (MVC) pattern to include diagnostics and fault recovery. The approach proposed an additional layer of custom FBs that monitored activities of the other blocks it is connected to in that region. Hametner et al. propose a Unit-Test framework for IEC 61499 using FBs that surround the block to be examined [46]. They generate a test framework automatically; however, the scope of this approach is design-time testing rather than Christensen's goal of longer-term fault detection.

2.3. Intentional Agents for Fault Finding

Agents were originally conceived as a way of computerizing tasks and processes in the way that humans typically might address them. Milis et al. discuss cognitive agents who are able to apply expert reasoning, mimicking the activities of human investigators [45]. Like humans, agents are able to adapt their strategies autonomously 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 ICPSs [47,48]. Bratman [49] and Wooldridge [50] provided an early background to intentional agents, introducing the concepts of Beliefs, Desires and Intentions (BDI). They described how agents form beliefs about the environment they are observing. Reasoning about those beliefs leads to the formation of desires to achieve goals that are fulfilled by pursuing intentional tasks. While ICPSs are deterministic, the world they operate in is not. Agents are one approach to working with devices that must bridge that temporal divide.

Diagnostic approaches encountered employed agents in both Model-Based [41,45] and Model-Free Artificial Intelligence fault-finding techniques [44,51]. The most common scenarios describe agents capturing diagnostic data, reasoning about evidence and then facilitating fault management [52,53]. Modest and Thielecke discuss encapsulating avionics domain knowledge for agents to create standard libraries of resources and techniques [54]. 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.

A number of alternative agent frameworks have emerged since Bratman and Wooldridge's original work. Jones and Wray [55] provide a comparison of multi-agent frameworks including GOMS [56] and Soar [57]. Both of these architectures approach agent interactions from a computational intelligence perspective. Later work by Jarvis, Rönnquist and Jain led to the development of the GORITE (Goal ORIented TEams) Multi-Agent Framework [3,5]. GORITE implements intentions as explicit activities and makes no distinction between goals and plans. This approach simplifies the specification of goals that can be shared by multiple agents and then executed independently. However, representing environment information in a format agents can use is complex [58,59]. Jarvis et al. comment that there is a need to be able to describe the activities required from agents more concisely using appropriate Domain-Specific Languages (DSLs) [3].

3. Architecting the Engine

RQ3 asks how a fault diagnostic engine can best be architected. Attribute-Driven Design is a systematic methodology for designing the architecture of software-intensive systems [11]. It 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*" ([60], p. 65). 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 [61–63]. Architects address this by identifying and focusing on those functional requirements that are Architecturally-Significant Requirements (ASRs) [10]. ASRs describe the functionality that will have a significant influence on or constrain the architectural design choices that can be made [64].

Chen et al. outline an approach for identifying ASRs among the requirements [65]. Table 1 details two of the ASRs for the engine categorized using their criteria. ASRs often affect multiple parts of the system rather than just the functionality they are describing. A characteristic that makes them significant is the high cost of changing that feature later [66]. 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. Chen et al. give 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 [67]. 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.

Table 1. Chen et al. 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

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 DPs and how they gather telemetry. This requirement is also deemed to be difficult to achieve, leading to the decision that this requirement is an ASR. In contrast, the performance of the DPs is critical, but does not have a wide-impact; the DPs are self-

contained. However their development is complex, leading to them also being classified as an ASR.

While functional requirements are goals that define what a feature must do, *quality attributes* describe qualities the feature must possess. They specify characteristics such as how suitable, how modifiable, or how reliable that feature must be. ISO Standard 25010 defines and standardizes categories of quality attributes so that they can be used consistently by architects and understood by stakeholders [67]. While architecting the engine, the desired quality attributes were established by working alongside stakeholders in structured Quality Attribute Workshops (QAWS) [68]. The architecture was then evaluated through scenarios that were created using the Architectural Trade-off Analysis Method (ATAM) [69]. The ATAM provides a step-by-step method of validating an architecture empirically.

The IEEE Standard 610 qualifies this definition of a requirement further, asserting that a requirement should be a "documented representation" of a need ([60], p. 65). Clements et al. caution that all the effort put in by the architects is wasted if the documentation they produce cannot be understood later by developers [10]. 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 [66]. Hence, the role of a software architect is to propose and document a solution by blending different, well-proven architectural styles. The Views and Beyond approach was used to create the Software Architecture Document (SAD) [70].

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. Figure 3 illustrates how in Views and Beyond, Bass et al. extend the 4 + 1 Views Model of Architecture proposed by Kruchten [71] to incorporate Rozanski and Woods seven distinct architectural viewpoints [72]. The Viewpoint Catalogue of Rozanski and Woods is explored in more detail in Section 1.3 of the SAD. The diagram shows how the stakeholders' viewpoints are considered in the light of the ASRs. The *View Packets* are the smallest piece of information a stakeholder requires, presented from a particular viewpoint [73]. They are detailed on the right of the diagram, showing the view packets contained in the SAD that detail an aspect of the architecture. Inside each view packet, the rationale behind each feature is explained, providing diagrams that present each element used in the architecture [64].

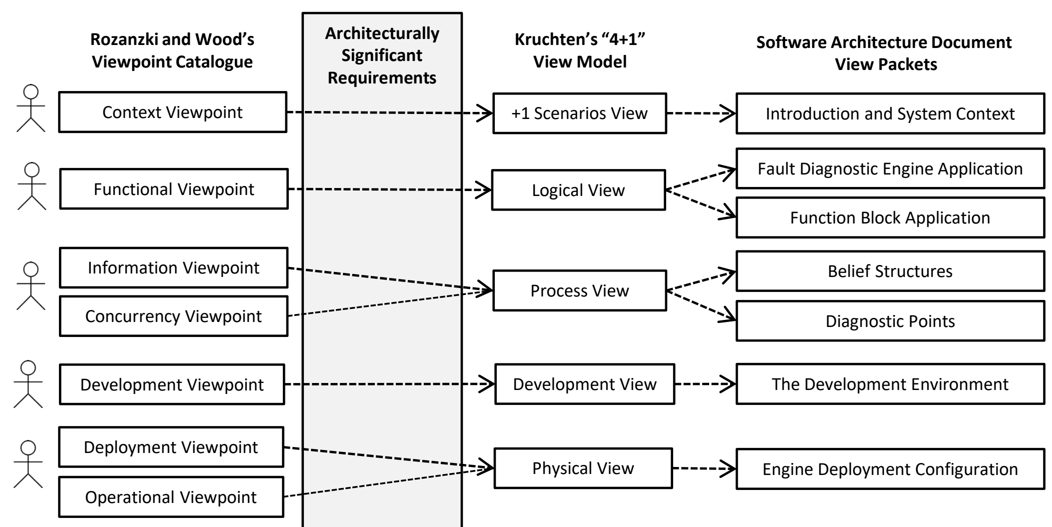

Figure 3. From viewpoints, architecturally-significant requirements and views to view packets.

SysML diagrams were adopted as a way of of presenting the elements of an architectural feature in a consistent way. SysML is an extension of the UML modeling language defined in the ISO 19514 standard [74,75]. SysML extends the range of UML diagrams to add Activity, Requirement and Block diagram types which are ideal for describing systems-of-systems architectures such as the engine. 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 Catalogues in the SAD that describe each view. For every view in the Kruchten 4 + 1 model, there are well-known UML or SysML diagrams that have been adopted to document the elements appropriately in that view. The SAD is intended to provide a single-source of truth to capture the needs of the stakeholders and propose architectural solutions to those needs. In doing that, the architects carefully tread a fine line between providing just enough detail without over-architecting the design. Doing so would constrain the ability of the developers to make their own decisions about how the code should implement the design. Architecture is concerned with the external characteristics of elements. In contrast, developers create the internal implementation of elements to realize those external characteristics. The SAD should therefore present an unambiguous architectural design, without unnecessary internal implementation details.

3.1. The System Context View of the Engine

The introductory sections of SAD present the agents and the engine in-context with the diagnostic needs they satisfy. Figure 4 illustrates the activities the agents can perform and the subsystems the engine is constructed from in this context. The view does this by presenting the relations, dependencies and interactions between the engine subsystems. Doberkat [76] 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 view corresponds to the +1 Scenario View proposed by Kruchten [71]. The Context View is a high-level overview, designed to present the purpose and business drivers that underpin the architectural decisions made about the engine. The view also describes the needs of the stakeholders who either use or will develop the engine. Stakeholders have differing concerns, so alternative view packets 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 FB 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.

The engine is a stand-alone system that interacts with a separate FB application. Design resources created in the IDE during the creation of the FB application are available to the agents. They use this structural information to configure the telemetry connections they need to interact with the FBs 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 in each context. Views and Beyond creates a framework in which each view or view packet includes a description of each architectural element the subsystem relies on as well as the interfaces they provide.

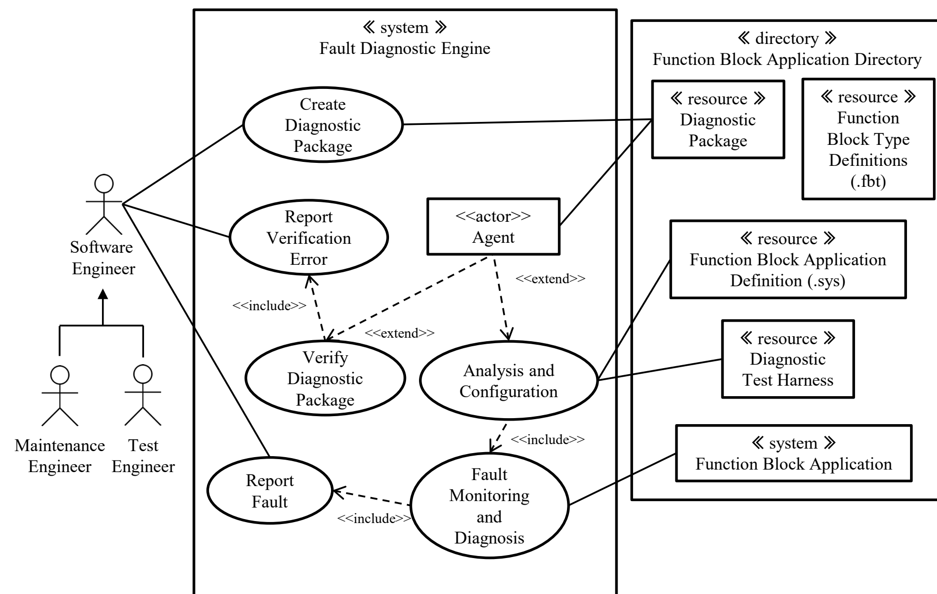

Figure 4. The system context view of the fault diagnostic engine.

3.2. The Logical View

Figure 5 presents the Logical View of the engine from the Rozanski and Woods Functional Viewpoint. A layered architectural style was chosen since the self-contained GORITE Team Manager subsystem provides the primary interface to the diagnostic agents. The refinement of the agent subsystem is discussed further in Section 3.3. 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 FB application layer. Sub-applications of the FB application can run on their own hardware platforms as distributed device instances in this lower layer.

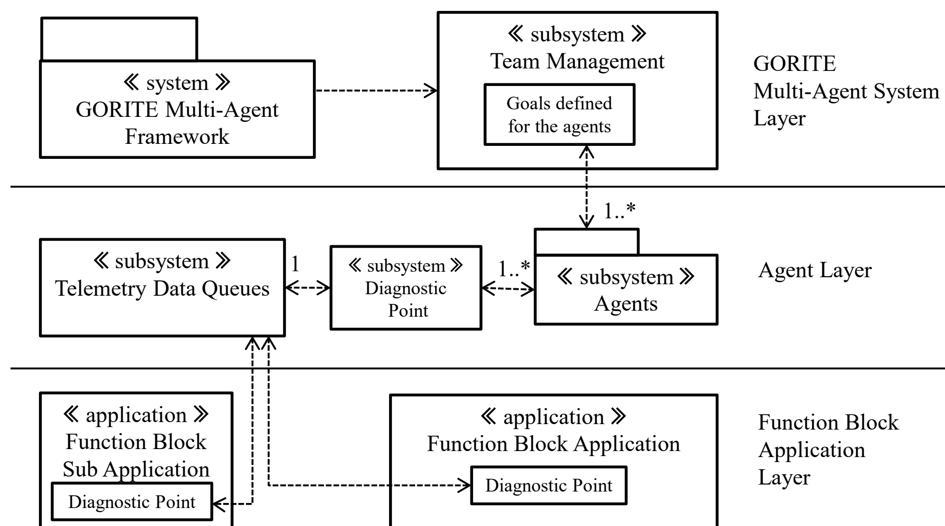

Figure 5. The layered architectural style of the fault diagnostic engine.

While agents are often created using object-oriented design patterns, there is a distinct difference in the way agents interact with other components in multi-agent frameworks. 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 [50,77]. An *active agent* encompasses its own thread of control [50]. Unlike object instances which can only undergo a change of state when one of their methods is called, agents can initiate a state change based on their own deliberations. Adapting the notation of Kidney and Denzinger [78], an active agent ag can be defined as:

Definition 1 (Active Agent). $ag = \langle Sit, Act, Dat, f_{sit} \rangle$ *is a tuple where:*

1. *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.*
2. *Act is the set of actions that an agent can perform in that environment.*
3. *Dat is the set of internal data the agent maintains about its state and the environment it is situated in.*
4. *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 $f_{sit} : Sit \times Dat \rightarrow Act$.*

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 Ga that the agent can perform to work towards a desired outcome. Jarvis et al. make no distinction between the traditional BDI concept of a plan and a goal; a plan in GORITE is just an explicit goal [3]. 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 2 (Behaviour). $Beh_{task} = \{act_1, act_2, \ldots, act_t\}$ *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.*

Since agents have the ability to choose from a number of alternatives what the appropriate course of action might be in a given situation or environment, and it is possible that a number of different behaviors could be exhibited that still achieve the same outcome.

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

Definition 3 (Goal). *Given an agent ag, a goal is specified as the tuple $Goal = \langle N, Ga, gs_w \rangle$ where:*

1. *N is a unique identifier that names the goal, and*
2. *Ga is the set of actions the agent can perform while pursuing the 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.*

GORITE defines the following goal states that can be returned from the customized $execute()$ function defined for a goal:

- 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.

3.3. The Process View

The Process View of the architecture focuses on processes that execute in the engine that support fault-finding. The view corresponds to Rozanski and Woods Information and Concurrency Viewpoint, detailed in Section 1.5 of the SAD. The SAD presents the operation of the agents and the DPs in two separate view packets.

Prototyping the agents and DPs during the ATAM evaluation of the architecture led to significant refinements. The *DiagnosticTeam* component is responsible for creating the team of agents and assigning 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. Each goal is implemented as a class with an $execute()$ function that is designed to operate until the status of the goal changes. That implies that different agents cannot perform their goals in parallel if they are both operating on the current GORITE thread. The prototype single agent performed three goals successfully that included the configuration of the FB application, watching for simple faults and then reporting them. The ATAM identified that this did not address the quality attributes including Concurrency and Scalability. These could not be satisfied by executing agents on a single execution thread. The agents also need to be able to persist their own private copies of resource objects, each with their own state management requirements. Instantiating agents on their own threads demanded a Trade-Off that led to a different goal structure for the Team manager. Figure 6 details the revised interaction between the Team Manager Agent and the Diagnostic Agents.

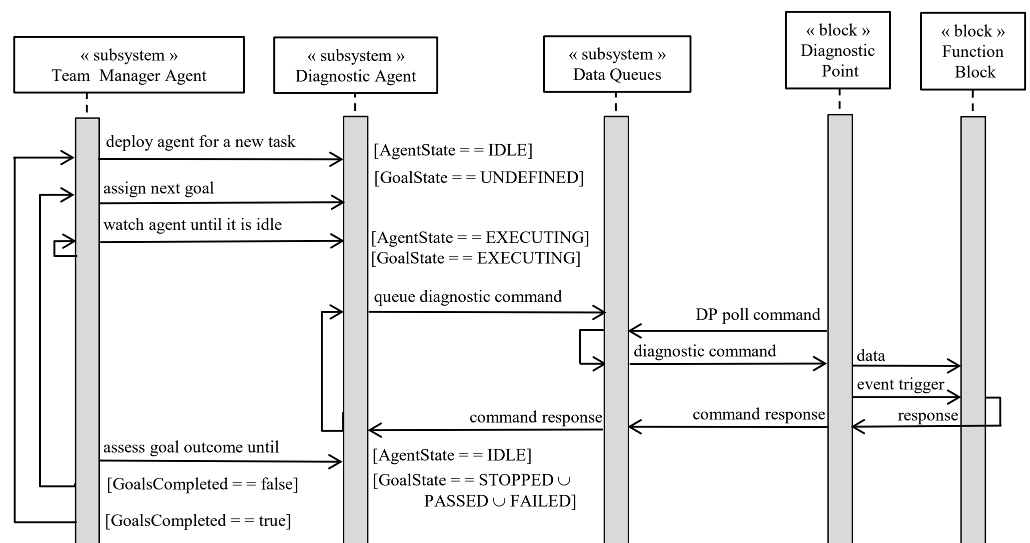

Figure 6. GORITE Team, agent, and diagnostic point processes across the execution layers.

This included the extension of the set of GORITE Goals states to include a new $EXECUTING$ status. The Team Manager Agent executes a set of GORITE Sequence Goals named $Manage$ where the role of the team manager is to assign goals to available agents and monitor their progress. Each named agent instance provides a set of thread-safe synchronized methods that ensure the setting and reading of the $AgentState$ and $GoalState$ agent properties are interleaved appropriately. Agents execute goals such as $Configure$, $Monitor$, $Diagnose$ and $Analyse$, transitioning from first configuring the FB application,

then watching the telemetry streams and intervening whenever fault evidence is observed. Each transition to a new goal is initiated by reporting via the *AgentState* that the current goal is no longer being executed, whereupon the Team Manager Agent evaluates the goal status reported by the agent in *GoalState* and assigns a new goal. Five agents were prototyped for an ATAM scenario and evaluated with GORITE on its own thread and its own set of management goals. No collisions or conflicts were observed. This new configuration now positions the Team Manager as an agent in its own right, co-ordinating multiple agents and freeing each agent to manage the state of the resources it instantiates. ATAMs do not usually involve a prototyping phase; however, the prototypes provided valuable quantitative feedback to justify the architectural changes needed that the ATAM had identified.

3.4. Diagnostic Points and Telemetry

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 [79]. The agents treat FBs as black-boxes whose behavior or misbehavior can be inferred from the data they exchange with each other. In most practical instances, the engine is deployed on its own hardware, separate from the platform the FB application is executing on. The rational for creating stand-alone fault management engines is discussed in depth in Benowitz's description of the NASA Curiosity Mars Rover [36]. Goupil et al. present a similar rational for the Airbus fault engine [80]. These engines have to keep functioning even when the applications they are managing are failing. Hence, the DPs in our engine need to provide remote data capture and interaction points that can operate as software-in-the-loop entry points within the FB application, exchanging telemetry via managed network connections to the platform that the agents are operating on.

Figure 7 shows the construction of the DP function block `DP`. This is a Composite Function Block (CFB) that contains an instance of the `AGENT_GATE` Service Interface Function Block (SIFB). The `DP` block is used to break and re-connect the data and event lines that connect FBs together.

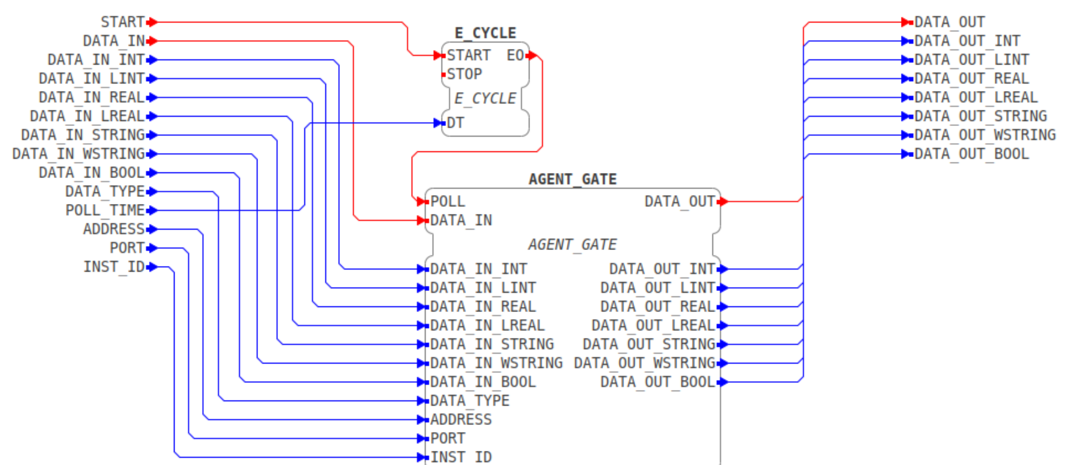

Figure 7. Diagnostic point composite function block.

CFBs expose a single input and output interface that encapsulates a more complex internal configuration. The configuration parameters include a *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. DPs initially pass all data and events through transparently after the agent has instructed them to operate in their *PASSTHROUGH_ENABLED* mode. Each data value is also sent back to the agent as it passes through the DP. The `AGENT_GATE` block implements a TCP client connection to a single named non-blocking I/O gateway component called *NIOserver* that is hosted on the engine. A TCP server listener socket

handover then assigns a unique socket to maintain the connection between the agent and the client DP. The agent communicates with the DP wired into the FB application via an in-memory object interface, also referred to as a *DiagnosticPoint*, established within the engine. The agents exchange telemetry with the DP using First-In-First Out (FIFO) data queues and the packet structure detailed in Figure 8.

Start of packet character	Packet type or command	DP Instance ID	Time stamp	Data length	Variable-length data value	End of packet character

Figure 8. Definition of the Data Packet structure used to exchange telemetry.

The start 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. The data type is inferred by the agent since it was established during the set up of the DP. The input `DATA_TYPE` is used by the block to determine 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 on input parameters. Table 2 details the packet commands supported by the DP and the agents. The DP provides input and output data ports for each defined IEC 61499 data type. These range from Boolean types through to long and short Integers, Real Numbers and Strings. An efficient type conversion in C++ with rounding to a specified number of decimal places was implemented in the blocks to ensure that data were processed rapidly and packed efficiently into the data packets.

Table 2. Packet Type Commands used to control and exchange data with Diagnostic Points.

Packet Type Enum	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 input and event ports.

The $gate()$ functionality allows the agent to dynamically isolate either a single FB or a set of related blocks to exercise them. Data and events are blocked from passing into or out of a ring-fenced area to allow the blocks to be tested in isolation, unaffected by other parts of the FB 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, listening, probing, and then moving on to the next section of interest. This improvement was driven by the ATAM evaluation and resulted in a more dynamic and resilient approach that allows the agents to change the type of interventions they choose to make without needing to re-wire additional DPs at runtime.

The Reliability quality attribute was addressed by making the DP function blocks fault-tolerant to network issues by ensuring that they manage their own TCP client connections. 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 $NIOserver$ component and the $DiagnosticPoint$ subsystems. In this sense, the $DiagnosticPoint$ instance shown in the Agent layer of Figure 5 acts like a Digital Twin of the DP FB within the FB application [81]. Later, this made the creation of the diagnostic scripts the agents use to execute their goals easier to construct since the instance references to the collection of DPs marshaled by the agent do not change in the event of a network issue.

3.5. Managing Agent Beliefs

The Process View also details the beliefs an agent forms. These are stored in an in-memory structure that provides a model of the domain the agents operate in [4]. 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. 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 alters the agents' perceptions. Dennett defines a First-Order Intentional system as one that has beliefs and desires, but no beliefs and desires about its beliefs and desires [82]. This corresponds to the initial BDI concept proposed by Bratman [83]. 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 beliefs formed by each agent in the engine are First-Order. Definition 4 is the primary definition of a belief that all the subsequent beliefs used by the agents are built upon:

Definition 4 (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.*

The veracity v is defined as the degree to which a concept or unit of information conforms to the truth or known facts [84]. The quantities assigned to v can range from true, false, or undetermined, 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.

Interaction beliefs capture an agent's knowledge about an ability they possess to interact with either other agents or the FB application to perform goals.

Definition 5 (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 a skill is the ability of an agent to communicate with the DPs, open and close control gates, and trigger inputs on an FB of interest. This belief structure

models IEC 61499-specific domain knowledge needed to interact with the FB application being diagnosed.

Figure 9 shows a set of DPs that an agent has wired between `Z_TEMPERATURE` and the `F_TO_C_CONV`. When the agent performs a $rewire()$ interaction, the DP1 and DP2 FB 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. These are captured further down the path by DP2 using the $read()$ command. Note that, if a DP is capturing the input side of a FB, the $gateClose()$ function must close only the inbound side. Conversely, when the DP is capturing an output, only the outbound data and events should be blocked from being sent onto the next connected FB.

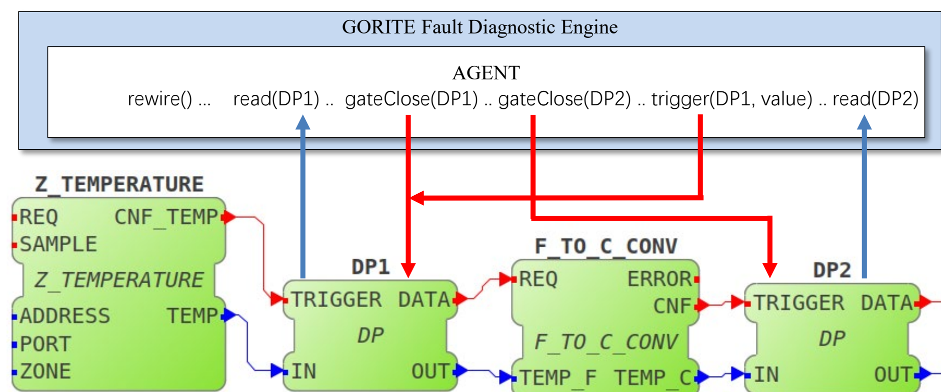

Figure 9. Capturing and triggering Diagnostic Points.

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 unless for some reason the agent is blocked from wielding that ability. Before the diagnostic goals are assigned, a second belief structure is made available to the agent that provides knowledge about the FB application and its structure.

Figure 10 shows the implementation of this belief structure created by restructuring the FB application as a Directed Graph. This is referred to as the System-Under-Diagnosis belief structure.

Definition 6 (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 $\langle FB_1, trg, FB_2 \rangle$. FB_1 where:*

- *FB_2 are function block instances in the system under diagnosis, and*
- *trg represents the conditions (events and variable values) under which a transition can be triggered by the agent from FB_1 to FB_2.*

The fault diagnostic engine adopts a model-based rather than a machine-learning approach. Manually creating models is labour and time-intensive, since models need to be calibrated [41,45]. IEC 61499 offers a formally-defined application structure file that allows the engine to autonomously construct an in-memory model of the function block application it has been assigned to monitor and diagnose. As engineers change their designs, the engine is able to update its model using its domain knowledge of the IEC 61499 architecture.

IEC 61499 Application Definition files are optimized to work with development IDEs such as 4diac [6] and nxtCONTROL [85]. However, the XML structure of these files is hard for an agent to navigate dynamically since the parameters for each FB are stored in different parts of the file. The five FB instances are stored as *FunctionBlock* nodes and the connections between them as directed edges stored in the *Connections* object. The name of an edge corresponds to the output event or data output on the FB the node describes. This structure is easier for the agents to navigate when rewiring the FBs to insert DPs. Agents

are also able to access this belief structure during fault finding by referencing a single node where all its details are available as properties.

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 block nodes can be updated. This architectural approach using an indexed list of objects means that new methods and data can be added to the structure without complicated changes to the interface.

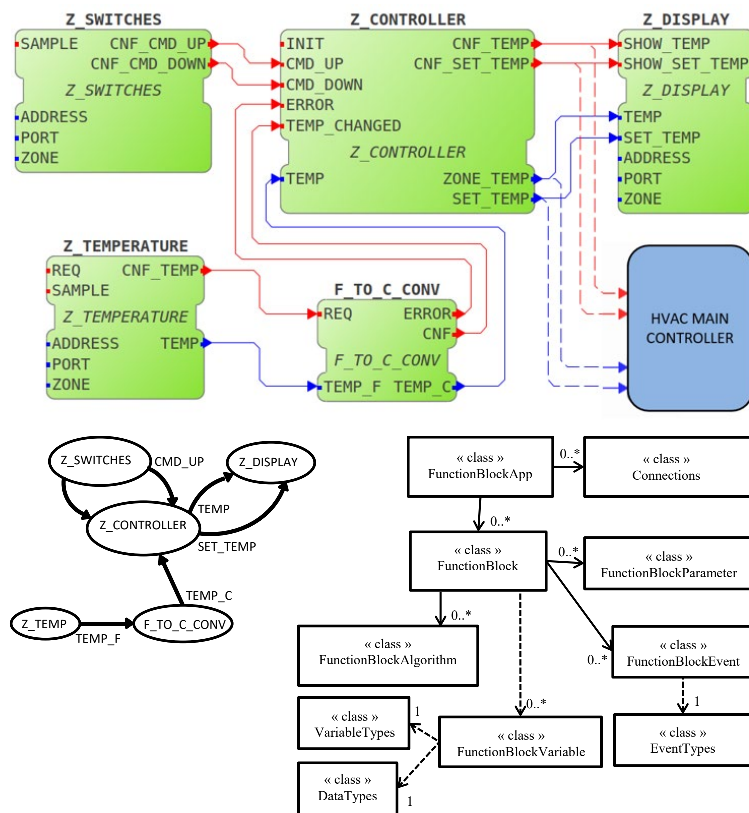

Figure 10. An FB application as a Directed Graph and the Function Block Node structure.

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

During the design phase, an engineer will create a package of information that identifies the DPs that are available for each FB. These *Diagnostic Packages* contain sets of test values associated with data pathways through the FB that can be used during diagnosis to determine if the FB is performing correctly. The agent interprets these diagnostic packages while iterating each FB as it builds its application belief structure. A *Diagnostic Harness* is then created by inserting all the DP instances into the function block using the agents' *rewire*() interaction belief.

Model-based approaches rely on techniques of fault-finding that involve invalidating one or more aspects of the model. The engine architecture facilitates this with interaction belief skills and domain knowledge of the IEC 61499 function block architecture, enabling the agents to build their own model of the application they are examining. This addresses some of the concerns that creating models is labor-intensive [41,45]. For fault diagnosis,

First-Order beliefs such as these are sufficient to form this model of the ICPS that the agents are examining.

The third belief structure captures dynamic diagnostic beliefs. As an agent investigates a fault, it forms new beliefs as it observes the FB application that is executing and interacts with it. It may also modify existing beliefs to better align them with new evidence.

Definition 7 (**Dynamic diagnostics belief**). *A belief* $b = \langle \triangle, v \rangle$ *is a dynamic diagnostics belief if* $\triangle$ *is represented as a pair* (FB_i, f_q) *where*

- FB_i *is a function block instance of the system under diagnosis, and*
- f_q *is a valid fault code for* FB_i*, obtained from a set of fault codes* F*.*

In this belief structure, a belief about what is happening is a fuzzy logic opinion about a particular FB, sensor, actuator, or algorithm in the FB application. The set $B = \{b_0, b_1, \ldots, b_n\}$ captures the agent's beliefs, where b_i represents either a single BFB or a network of BFBs connected as a CFB. A BFB or CFB represents the smallest unit of functionality an agent can test and hence beliefs are atomic at this level.

Before the Team Manager shown in Figure 6 instructs the agent to begin operating, it provides a set of goals that can include monitoring for fault signatures and executing diagnostic plans. The agent is also provided with a definition of what constitutes normal behavior for each FB via the set of diagnostic packages. One example of normal behavior is the appearance of temperature readings at 500 ms intervals from `Z_TEMPERATURE TEMP`. These propagate through the controller to appear at `Z_CONTROLLER ZONE_TEMP`. The agent then pursues its *Monitor* goal, watching for normal temperature fluctuations. All DP instances pass events and data through transparently to other FBs as well as back to the agent. The agent continues pursuing this goal until one or more of its primary beliefs about normal operation are invalidated. The agent establishes an initial belief for the *Monitor* goal such that:

$$b_0 = \langle FB_{\texttt{Z_TEMPERATURE}}, v_{\text{undetermined}}, f_0 \rangle$$

For an agent, the pursuit of its *Monitor* goal is the repeated re-evaluation of each of its beliefs by observing or performing prescribed tests at defined intervals. Whenever the agent determines that the FB is operating within tolerance, it reinforces its belief so that $v_{\text{undetermined}} \rightarrow v_{\text{true}}$. Other primary beliefs track and remember what is being observed in different parts of the FB application.

Invalidating any one of these beliefs in B causes the agent to signal the Team Manager Agent that it has completed its primary *Monitor* goal successfully by finding a potential fault. The Team Manager Agent will then assign the agent a *Diagnostic* goal. The agent then adopts a divide-and-conquer strategy for diagnosing faults in the temperature sensor subsystem described earlier in Figure 9. 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 F) should trigger the `ERROR` event, captured by another DP. 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 fault path checking each subsequent FB, updating its beliefs until all FBs 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.

4. Evaluating the Architecture of the Engine

Software Engineering, and Software Architecture in particular, are reflective disciplines [86]. Once an architecture is proposed, it should be presented in a well-structured document. The design decisions made should then be considered 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.

An ATAM [69] 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 SAD visualizes the application with a high degree of abstraction [87]. 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 usually less costly than remediation work later [66]. 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 [88]. 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 SAD by the prototyping relevant parts of the architecture to investigate issues uncovered during the ATAM.

4.1. Constructing the ATAM Utility Tree

RQ4 was partially addressed by constructing a Utility Tree. These are used during the ATAM to present quality attributes in a hierarchical tree structure that highlights the importance and risk of each attribute. Figure 11 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 SAD. Each of the sections gives examples that evaluate the SAD views against these qualities, identifying the risks, sensitivities and trade-offs and proposing scenarios to evaluate suggested changes.

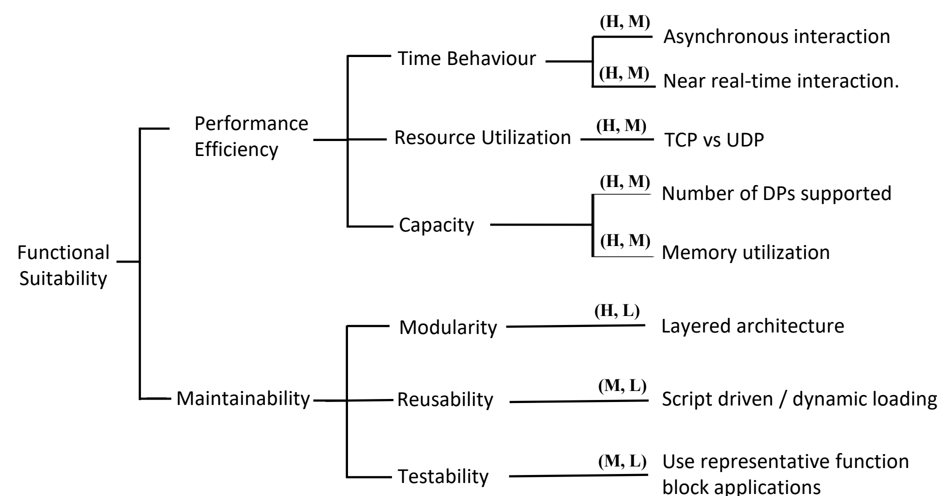

Figure 11. The ATAM utility tree for the engine.

4.2. 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 he SAD requires that the agents must be able to operate faster than the FB 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 FB 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 [36,80]. Where 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.

4.3. 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 a desktop, server and embedded system platforms. The DPs are custom FBs implemented in C++ for compatibility with the FORTE function block runtime [89]. DPs were converted to C# to run with the nxtCONTROL runtime [85]. The engine is considered low risk for these qualities. This also partially satisfies RQ2 by helping to ensure that the DPs can operate in the same way in multiple environments.

4.4. 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 FB application is failing or has failed. The sub-category of Recoverability is addressed by the DP ability to re-connect automatically if they lose their connection to the engine. Data is exchanged asynchronously between the agent and the DPs it is capturing data from or sending test data to. A First-In, First-Out (FIFO) queue structure is used by the engine. A non-blocking socket TCP socket protocol was implemented rather than use UDP. Transmission Control Protocol (TCP) [90] is a connection-oriented protocol that provides guaranteed packet delivery and error correction. In contrast, the User Datagram Protocol (UDP) [91] is a raw, connectionless protocol that provides no error checking or guarantee of delivery. 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 DPs to evaluate the latency of high-traffic environments.

4.5. 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 DPs and the agents are 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. DPs 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. [92] propose a secure-by-design approach that implements encrypted data streams using custom function blocks and security keys that are applicable to this architecture.

5. Conclusions and Future Work

The model-based invalidation approach presented here addresses RQ1 by providing the agent with a way of distinguishing normal operation from misbehavior. The use of domain-specific agents in the GORITE framework also provides a scope for implementing model-free or machine-learning approaches during subsequent research. The proliferation of hybrid techniques encountered in the literature suggests that a blend of fault detection methods is viable. The use of the DPs addresses RQ2 by providing feeds of not only telemetry from FBs but also control over their operations during fault investigations.

The application of Views and Beyond coupled with ADD was used to address RQ3. The ATAM evaluations that addressed RQ4 are usually performed after the first version of the architecture has been documented. Our approach demonstrated the value of performing brief, focused and iterative ATAMs that led to the findings presented in the evaluation section. Adhering to a sound architectural documentation methodology, rigorous evaluation with the ATAM and validation with prototypes led to important architectural refinements. These are continuing to improve the resilience, performance and reliability of the engine during the subsequent on-going development.

Model-Driven Development with Diagnostics suggests a more collaborative and resilient approach to fault management needs early in the architectural design. The engine offers a way to automate many of the fault diagnostic tasks that are not possible in the current IEC 61499 IDEs. The dynamic nature of DPs capitalises on the existing reconfigurability features of the 4diac FORTE runtime environment. Running the engine on its own platform with multiple agents addresses scalability concerns that traditional Built-In-Self-Test (BITS) approaches to fault diagnostics and testing do not. However, engines such as these need to deliver practical advantages if they are to see adoption in the IEC 61499 community. At present, the diagnosis presented by the agents to the engineers does not include deep reasoning about the evidence captured. The belief structures presented here suggest ways of doing that which is being investigated in our current research. This is leading to the implementation of *pragmatic* agents that reason more deeply about the faults they have observed using their domain knowledge as a basis for their deliberations. There is also further work on a Domain Specific Language used to write the diagnostic routines [3]. A level of compliance with products such as Selenium would help to standardize the way diagnostic commands are expressed by the agents, making adoption easier for users [93].

The architecture will continue to evolve through subsequent iterations as the engine is being implemented by our team. We intend to publish further articles that include empirical evaluations of the architecture supported by appropriate FB applications that illustrate the full scope of the engines' capabilities.

Author Contributions: Conceptualization, B.D. and R.S.; methodology, B.D., R.S. and S.G.M.; software, B.D.; validation, B.D. and R.S.; writing—original draft preparation, B.D.; review and editing, B.D. and R.S.; supervision, R.S. and S.G.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Glossary

ADD	Attribute-Driven Design method [11].
ATAM	Architectural Trade-off Analysis Method [69].
CFB	Composite Function Block [2].
DP	Diagnostic Point [8].
FB	Function Block [2].
ICPS	Industrial Cyber-Physical System.
PLC	Programmable Logic Controller.

QAW	Quality Attribute Workshop
ASR	Architecturally-Significant Requirement.
BFB	Basic Function Block [2].
CPS	Cyber-Physical System [20].
ECS	Embedded Control System.
FDE	Fault Diagnostic Engine.
IDE	Integrated Development Environment.
QA	Quality Attribute.
SAD	Software Architecture Document.

References

1. Lee, E.A.; Seshia, S.A. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*; MIT Press: Cambridge, MA, USA, 2016.
2. IEC. *Function Blocks–Part 1: Architecture*; IEC: Geneva, Switzerland, 2013.
3. Jarvis, D.; Jarvis, J.; Rönnquist, R.; Jain, L.C. Multi-Agent Systems. In *Multiagent Systems and Applications*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 1–12. [CrossRef]
4. Ganzha, M.; Jain, L.C.; Jarvis, D.; Jarvis, J.; Rönnquist, R. *Multiagent Systems and Applications*; Springer: Berlin/Heidelberg, Germany, 2013. [CrossRef]
5. Rönnquist, R. The Goal Oriented Teams (GORITE) framework. In *International Workshop on Programming Multi-Agent Systems*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 27–41.
6. Strasser, T.; Rooker, M.; Ebenhofer, G.; Zoitl, A.; Sünder, C.; Valentini, A.; Martel, A. Framework for distributed industrial automation and control (4DIAC). In Proceedings of the 2008 6th IEEE International Conference on Industrial Informatics, Daejeon, Korea, 13–16 July 2008; pp. 283–288.
7. Zoitl, A.; Strasser, T.; Ebenhofer, G. Developing modular reusable IEC 61499 control applications with 4DIAC. In Proceedings of the 2013 11th IEEE International Conference on Industrial Informatics (INDIN), Bochum, Germany, 29–31 July 2013; pp. 358–363.
8. Dowdeswell, B.; Sinha, R.; MacDonell, S.G. Diagnosable-by-Design Model-Driven Development for IEC 61499 Industrial Cyber-Physical Systems. In Proceedings of the IECON 2020 46th International Conference of the IEEE Industrial Electronics Society, Singapore, 18–21 October 2020.
9. Dowdeswell, B.; Sinha, R.; MacDonell, S.G. Finding faults: A scoping study of fault diagnostics for Industrial Cyber-Physical Systems. *J. Syst. Softw.* **2020**, *168*, 110638. [CrossRef]
10. Clements, P.; Garlan, D.; Little, R.; Nord, R.; Stafford, J. Documenting software architectures: Views and Beyond. In Proceedings of the 25th International Conference on Software Engineering, Portland, OR, USA, 3–10 May 2003; pp. 740–741. [CrossRef]
11. Wojcik, R.; Bachmann, F.; Bass, L.; Clements, P.; Merson, P.; Nord, R.; Wood, B. *Attribute-Driven Design (ADD), Version 2.0.*; Technical Report; Software Engineering Institute (SEI), Carnegie-Mellon University: Pittsburg, PA, USA, 2006.
12. Dowdeswell, B.; Sinha, R.; MacDonell, S. Mendeley Dataset: A Software Architecture for a Fault Diagnostic Engine. *Mendeley Data* **2020**. [CrossRef]
13. Kalachev, A.; Zhabelova, G.; Vyatkin, V.; Jarvis, D.; Pang, C. Intelligent mechatronic system with decentralised control and multi-agent planning. In Proceedings of the IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society, Washington, DC, USA, 21–23 October 2018; pp. 3126–3133. [CrossRef]
14. Jarvis, D.; Jarvis, J.; Kalachev, A.; Zhabelova, G.; Vyatkin, V. PROSA/G: An architecture for agent-based manufacturing execution. In Proceedings of the 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), Turin, Italy, 4–7 September 2018; Volume 1, pp. 155–160. [CrossRef]
15. Christensen, J.H. Design Patterns, Frameworks, and Methodologies. In *Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499*; CRC Press: Boca Raton, FL, USA, 2017; p. 27. [CrossRef]
16. Samad, T.; Parisini, T.; Annaswamy, A. Systems of systems. *Impact Control. Technol.* **2011**, *12*, 175–183.
17. Laughton, M.A.; Say, M.G. *Electrical Engineer's Reference Book*; Elsevier: Amsterdam, The Netherlands, 2013.
18. Parr, E.A. *Industrial Control Handbook*; Industrial Press Inc.: New York, NY, USA, 1998.
19. Boem, F.; Ferrari, R.M.; Parisini, T. Distributed fault detection and isolation of continuous-time nonlinear systems. *Eur. J. Control.* **2011**, *17*, 603–620. [CrossRef]
20. Baheti, R.; Gill, H. Cyber-physical systems. *Impact Control. Technol.* **2011**, *12*, 161–166.
21. Leitao, P.; Karnouskos, S.; Ribeiro, L.; Lee, J.; Strasser, T.; Colombo, A.W. Smart agents in industrial cyber–physical systems. *Proc. IEEE* **2016**, *104*, 1086–1101. [CrossRef]
22. Cremona, F.; Lohstroh, M.; Broman, D.; Lee, E.A.; Masin, M.; Tripakis, S. Hybrid co-simulation: its about time. *Softw. Syst. Model.* **2019**, *18*, 1655–1679. [CrossRef]
23. Workers, W. Franka Emika Panda Research Robot Manual. 2020. Available online: https://www.franka.de/ (accessed on 21 July 2021).
24. Jazdi, N. Cyber physical systems in the context of Industry 4.0. In Proceedings of the 2014 IEEE International Conference on Automation, Quality and Testing, Robotics, Cluj-Napoca, Romania, 22–24 May 2014; pp. 1–4. [CrossRef]
25. Alur, R. *Principles of Cyber-Physical Systems*; MIT Press: Cambridge, MA, USA, 2015.
26. IEC. *Function Blocks–Part 1: Programmable Controllers. General Information*; IEC: Geneva, Switzerland, 2003; p. 61131.

27. Moore, E.F. Gedanken-experiments on sequential machines. *Autom. Stud.* **1956**, *34*, 129–153.
28. Lindgren, P.; Lindner, M.; Lindner, A.; Vyatkin, V.; Pereira, D.; Pinho, L.M. A real-time semantics for the IEC 61499 standard. In Proceedings of the 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 8–11 September 2015; pp. 1–6. [CrossRef]
29. Hehenberger, P.; Vogel-Heuser, B.; Bradley, D.; Eynard, B.; Tomiyama, T.; Achiche, S. Design, modelling, simulation and integration of cyber physical systems: Methods and applications. *Comput. Ind.* **2016**, *82*, 273–289. [CrossRef]
30. Atmojo, U.D.; Blech, J.O.; Vyatkin, V. A Plug and Produce-inspired Approach in Distributed Control Architecture: A Flexible Assembly Line and Product Centric Control Example. In Proceedings of the 2020 IEEE International Conference on Industrial Technology (ICIT), Buenos Aires, Argentina, 26–28 February 2020; pp. 271–277. [CrossRef]
31. Yang, C.W.; Zhabelova, G.; Vyatkin, V.; Nair, N.K.C.; Apostolov, A. Smart Grid automation: Distributed protection application with IEC61850/IEC61499. In Proceedings of the IEEE 10th International Conference on Industrial Informatics, Beijing, China, 25–27 July 2012; pp. 1067–1072. [CrossRef]
32. NOJA. *N*OJA Power Smart Grid Automation Software. 2015. Available online: https://www.nojapower.com.au/tags/smart-grid-automation-software (accessed on 21 July 2021).
33. Khairullah, S.S.; Elks, C.R. Self-repairing hardware architecture for safety-critical cyber-physical-systems. *IET Cyber-Phys. Syst. Theory Appl.* **2020**, *5*, 92–99. [CrossRef]
34. Jackson, S. A multidisciplinary framework for resilence to disasters and disruptions. *J. Integr. Des. Process. Sci.* **2007**, *11*, 91–108.
35. Holzmann, G.J. Mars Code. *Commun. ACM* **2014**, *57*, 64–73. [CrossRef]
36. Benowitz, E. The Curiosity Mars Rover's Fault Protection Engine. In Proceedings of the 2014 IEEE International Conference on Space Mission Challenges for Information Technology, Laurel, MD, USA, 24–26 September 2014; pp. 62–66. [CrossRef]
37. Thombare, T.R.; Dole, L. Review on fault diagnosis model in automobile. In Proceedings of the 2014 IEEE International Conference on Computational Intelligence and Computing Research, Coimbatore, India, 18–20 December 2014; pp. 1–4. [CrossRef]
38. Ragheb, M. Fault Tree Analysis and Alternative Configurations of Angle of Attack (AOA) Sensors as Part of Maneuvering Characteristics Augmentation System (MCAS). 2019. Available online: https://www.mragheb.com (accessed on 20 July 2021).
39. Zolghadri, A.; Cieslak, J.; Efimov, D.; Henry, D.; Goupil, P.; Dayre, R.; Gheorghe, A.; Leberre, H. Signal and model-based fault detection for aircraft systems. *IFAC-PapersOnLine* **2015**, *48*, 1096–1101. [CrossRef]
40. Dearden, R.; Willeke, T.; Simmons, R.; Verma, V.; Hutter, F.; Thrun, S. Real-time fault detection and situational awareness for rovers: Report on the mars technology program task. In Proceedings of the 2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No. 04TH8720), Big Sky, MT, USA, 6–13 March 2004; Volume 2, pp. 826–840. [CrossRef]
41. Provan, G. A Contracts-Based Framework for Systems Modeling and Embedded Diagnostics. In Proceedings of the International Conference on Software Engineering and Formal Methods, Grenoble, France, 1–5 September 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 131–143. [CrossRef]
42. Harirchi, F.; Ozay, N. Guaranteed model-based fault detection in cyber–physical systems: A model invalidation approach. *Automatica* **2018**, *93*, 476–488. [CrossRef]
43. Koitz, R.; Lüftenegger, J.; Wotawa, F. Model-based diagnosis in practice: interaction design of an integrated diagnosis application for industrial wind turbines. In Proceedings of the International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, Arras, France, 27–30 June 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 440–445. [CrossRef]
44. Sankavaram, C.; Kodali, A.; Pattipati, K. An integrated health management process for automotive cyber-physical systems. In Proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC), San Diego, CA, USA, 28–31 January 2013; pp. 82–86. [CrossRef]
45. Milis, G.M.; Eliades, D.G.; Panayiotou, C.G.; Polycarpou, M.M. A cognitive fault-detection design architecture. In Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN), Vancouver, BC, Canada, 24–29 July 2016; pp. 2819–2826. [CrossRef]
46. Hametner, R.; Hegny, I.; Zoitl, A. 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), Barcelona, Spain, 16–19 September 2014; pp. 1–8. [CrossRef]
47. Calvaresi, D.; Marinoni, M.; Sturm, A.; Schumacher, M.; Buttazzo, G. The challenge of real-time multi-agent systems for enabling IoT and CPS. In Proceedings of the International Conference on Web Intelligence, Leipzig, Germany, 23–26 August 2017; ACM: New York, NY, USA, 2017; pp. 356–364. [CrossRef]
48. Braberman, V.; D'Ippolito, N.; Kramer, J.; Sykes, D.; Uchitel, S. Morph: A reference architecture for configuration and behaviour self-adaptation. In Proceedings of the 1st International Workshop on Control Theory for Software Engineering, Bergamo, Italy, 31 August 2015. [CrossRef]
49. Bratman, M.E.; Israel, D.J.; Pollack, M.E. Plans and resource-bounded practical reasoning. *Comput. Intell.* **1988**, *4*, 349–355. [CrossRef]
50. Wooldridge, M. *An Introduction to Multiagent Systems*; John Wiley & Sons: Hoboken, NJ, USA, 2009.
51. Wu, D.; Liu, S.; Zhang, L.; Terpenny, J.; Gao, R.X.; Kurfess, T.; Guzzo, J.A. A fog computing-based framework for process monitoring and prognosis in cyber-manufacturing. *J. Manuf. Syst.* **2017**, *43*, 25–34. [CrossRef]
52. Janasak, K.M.; Beshears, R.R. Diagnostics to Prognostics—A product availability technology evolution. In Proceedings of the 2007 Annual Reliability and Maintainability Symposium, Orlando, FL, USA, 22–25 January 2007; pp. 113–118. [CrossRef]

53. Klar, D.; Huhn, M. Interfaces and models for the diagnosis of cyber-physical ecosystems. In Proceedings of the 2012 6th IEEE International Conference on Digital Ecosystems and Technologies (DEST), Campione d'Italia, Italy, 18–20 June 2012; pp. 1–6. [CrossRef]
54. Modest, C.; Thielecke, F. SPYDER: A software package for system diagnosis engineering. *CEAS Aeronaut. J.* **2016**, *7*, 315–331. [CrossRef]
55. Jones, R.M.; Wray, R.E. Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI Mag.* **2006**, *27*, 57. [CrossRef]
56. Card, S.K.; Newell, A.; Moran, T.P. *The Psychology of Human-Computer Interaction*; CRC Press: Boca Raton, FL, USA, 1983.
57. Laird, J.E.; Newell, A.; Rosenbloom, P.S. Soar: An architecture for general intelligence. *Artif. Intell.* **1987**, *33*, 1–64. [CrossRef]
58. Fröhlich, P.; Móra, I.; Nejdl, W.; Schröder, M. Diagnostic agents for distributed systems. In *ModelAge Workshop on Formal Models of Agents*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 173–186. [CrossRef]
59. Santos, F.; Nunes, I.; Bazzan, A.L. Model-driven agent-based simulation development: A modeling language and empirical evaluation in the adaptive traffic signal control domain. *Simul. Model. Pract. Theory* **2018**, *83*, 162–187. [CrossRef]
60. IEEE. IEEE Standard Glossary of Software Engineering Terminology. *Office* **1990**, *121990*, 1.
61. Ribeiro, C.; Berry, D. The Prevalence and Severity of Persistent Ambiguity in Software Requirements Specifications: Is a Special Effort Needed to Find Them? *Sci. Comput. Program.* **2020**, *195*, 102472. [CrossRef]
62. Sabriye, A.O.J.; Zainon, W.M.N.W. An Approach for Detecting Syntax and Syntactical Ambiguity in Software Requirement Specification. *J. Theor. Appl. Inf. Technol.* **2018**, *96*, 2275–2284.
63. Segal, S. A framework for removing ambiguity from software requirements. *IIOAB J.* **2017**, *8*, 43–46.
64. Van Heesch, U.; Avgeriou, P.; Hilliard, R. A documentation framework for architecture decisions. *J. Syst. Softw.* **2012**, *85*, 795–820. [CrossRef]
65. Chen, L.; Babar, M.A.; Nuseibeh, B. Characterizing architecturally significant requirements. *IEEE Softw.* **2012**, *30*, 38–45. [CrossRef]
66. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*, 3rd ed.; Addison-Wesley: Boston, MA, USA, 2013.
67. ISO. *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*; International Organization for Standardization: Geneva, Switzerland, 2011.
68. Barbacci, M.R.; Ellison, R.J.; Lattanze, A.J.; Stafford, J.A.; Weinstock, C.B. *Quality Attribute Workshops (QAWA)*; Technical Report; Carnegie-Mellon University: Pittsburgh, PA, USA, 2003.
69. Kazman, R.; Klein, M.; Clements, P. *ATAM: Method for Architecture Evaluation*; Technical Report; Software Engineering Institute, Carnegie-Mellon University: Pittsburgh, PA, USA, 2000.
70. Mellon, C. *Views and Beyond: The SEI Approach for Architecture Documentation*; Carnegie Mellon University: Pittsburgh, PA, USA, 2018.
71. Kruchten, P.B. The 4 + 1 View Model of Architecture. *IEEE Softw.* **1995**, *12*, 42–50. [CrossRef]
72. Rozanski, N.; Woods, E. *Software Systems Architecture: Working with Stakeholders Usin Viewpoints and Perspectives*; Addison-Wesley: Boston, MA, USA, 2011.
73. May, N. A survey of software architecture viewpoint models. In Proceedings of the Sixth Australasian Workshop on Software and System Architectures, Brisbane, Australia, 29 March 2005; pp. 13–24.
74. ISO. *ISO Standard 19514:2017 Information Technology—The Object Management Group Systems Modeling Language (OMG SysML)*; ISO: Geneva, Switzerland, 2019.
75. ISO. *ISO Standard 19501:2005 Information Technology—The Unified Modeling Language (OMG UML)*; ISO: Geneva, Switzerland, 2019.
76. Doberkat, E.E. Pipelines: Modelling a software architecture through relations. *Acta Inform.* **2003**, *40*, 37–79. [CrossRef]
77. Shehory, O.M. *Architectural Properties of Multi-Agent Systems*; The Robotics Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 1998.
78. Kidney, J.; Denzinger, J. Testing the limits of emergent behavior in MAS using learning of cooperative behavior. *Front. Artif. Intell. Appl.* **2006**, *141*, 260.
79. Carden, F.; Jedlicka, R.P.; Henry, R. *Telemetry Systems Engineering*; Artech House: Norwood, MA, USA, 2002.
80. Goupil, P.; Boada-Bauxell, J.; Marcos, A.; Cortet, E.; Kerr, M.; Costa, H. AIRBUS efforts towards advanced real-time fault diagnosis and fault tolerant control. *IFAC Proc. Vol.* **2014**, *47*, 3471–3476. [CrossRef]
81. Kritzinger, W.; Karner, M.; Traar, G.; Henjes, J.; Sihn, W. Digital Twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine* **2018**, *51*, 1016–1022. [CrossRef]
82. Dennett, D. Intentional Systems Theory. In *The Oxford Handbook of Philosophy of Mind*; Oxford University Press: Oxford, UK, 2009; pp. 339–350.
83. Bratman, M. *Intention, Plans, and Practical Reason*; Harvard University Press: Cambridge, MA, USA, 1987; Volume 10. [CrossRef]
84. Merriam-Webster. Veracity Dictonary Definition. 2021. Available online: https://www.merriam-webster.com/dictionary/veracity (accessed on 20 July 2021).
85. nxtControl GmbH. The nxtCONTROL Development Environment. 2020. Available online: https://www.nxtcontrol.com/en/engineering/ (accessed on 20 July 2021).
86. Hazzan, O. The reflective practitioner perspective in software engineering education. *J. Syst. Softw.* **2002**, *63*, 161–171. [CrossRef]

87. Barcelos, R.F.; Travassos, G.H. *Evaluation Approaches for Software Architectural Documents: A Systematic Review*; CIbSE: London, UK, 2006; pp. 433–446.
88. Reijonen, V.; Koskinen, J.; Haikala, I. Experiences from scenario-based architecture evaluations with ATAM. In Proceedings of the European Conference on Software Architecture, Copenhagen, Denmark, 23–26 August 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 214–229. [CrossRef]
89. 4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment. 2019. Available online: https://www.eclipse.org/4diac/en_rte.php (accessed on 20 July 2021)
90. Defense Advanced Research Projects Agency. *RFC 793 Transmission Control Protocol*; Defense Advanced Research Projects Agency: Arlington County, VA, USA, 1981. Available online: https://datatracker.ietf.org/doc/html/rfc793 (accessed on 20 July 2021).
91. DARPA. *RFC 768 User Datagram Protocol*; DARPA: Arlington County, VA, USA, 1980. Available online: https://www.ietf.org/rfc/rfc768 (accessed on 20 July 2021).
92. Tanveer, A.; Sinha, R.; MacDonell, S.G. On Design-time Security in IEC 61499 Systems: Conceptualisation, Implementation, and Feasibility. In Proceedings of the 2018 IEEE 16th International Conference on Industrial Informatics (INDIN), Porto, Portugal, 18–20 July 2018; pp. 778–785. [CrossRef]
93. The Selenium Testing Environment. 2020. Available online: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Your_own_automation_environment (accessed on 20 July 2021).