UK Atomic Energy Authority

UKAEA-CCFE-PR(21)20

Ipek Caliskanelli, Matthew Goodliffe, Craig Whiffin, Michail Xymitoulias, Edward Whittaker, Swapnil Verma, Craig Hickman, Chen Minghao, Robert Skilton

CorteX: A software framework for interoperable, plug-and-play, distributed robotic systems-of-systems

CorteX: A software framework for interoperable, plug-and-play, distributed robotic systems-of-systems

Ipek Caliskanelli, Matthew Goodliffe, Craig Whiffin, Michail Xymitoulias, Edward Whittaker, Swapnil Verma, Craig Hickman, Chen Minghao, Robert Skilton

This is a preprint of a paper submitted for publication in RoboSoft

CorteX: A software framework for interoperable, plug-and-play, distributed, robotic systems-of-systems

Ipek Caliskanelli, Matthew Goodliffe, Craig Whiffin, Michail Xymitoulias, Edward Whittaker, Swapnil Verma, Craig Hickman, Chen Minghao and Robert Skilton

Abstract In the worlds of nuclear energy, mining, petrochemical processing, and sub-sea, robots are being used for an increasing number and range of tasks. This is resulting in ever more complex robotics installations being deployed, maintained, and extended over long periods of time. Additionally, the unstructured, experimental, or unknown operational conditions frequently result in new or changing system requirements, meaning extension and adaptation is necessary. Whilst existing frameworks allow for robust integration of complex robotic systems, they are not compatible with highly efficient maintenance and extension in the face of changing requirements and obsolescence issues over decades-long periods. We present CorteX that attempt to solve the long-term maintainability and extensibility issues encountered in such scenarios through the use of a standardised, self-describing data representations and associated communications protocols. Progress in developing and testing the CorteX framework, as well as an overview of current and planned deployments, will be presented.

1 Introduction

Systems that will be used to maintain and inspect future fusion powerplants (e.g. ITER [3] and DEMO [18]) are expected to integrate hundreds of systems from multiple suppliers with a lifetime of several decades, over which requirements evolve and obsolescence management is required. There are significant challenges associated with the integration of such large systems from multiple suppliers, each operating using bespoke interfaces and having their training requirements.

Nuclear robotics in complex facilities, comprising hundreds of interoperating systems, will become increasingly commonplace as safety and productivity require-

Remote Applications in Challenging Environments (RACE), Culham Science Centre, Abingdon, OX14 3DB e-mail: ipek.caliskanelli, matthew.goodliffe, craig.whiffin, michail.xymitoulias, edward.whittaker, swapnil.verma, craig.hickman, chen.minghao, robert.skilton@ukaea.uk

ments drive facility design and operation. In order to be efficient and operate for years or decades without significant downtime, future systems will be dependent on automation and the sharing of information. Within current nuclear robotic applications, there are fundamental limitations regarding long-term maintainability, extensibility, dependability, reliability, security, and compatibility with regulatory requirements. Existing networked and cloud-based robotics lack interoperability, heterogeneity, security, multi-robot management, common infrastructure design, Quality-of-Service (QoS), and standardisation. There is a need for a new robotic framework, specifically tailored for such applications, that can adapt to new challenges, new user requirements, and new advances in technology. RACE is working towards providing a future-proof communications and control framework, capable of meeting these requirements, essential for future large-scale nuclear robotic facilities.

Development has been focused around five key design principles, namely reusability, extensibility, modularity, standardisation, and integrated user interfaces. To guarantee reusability, control solutions must be implemented in a generic fashion in order to be agnostic of their application. These control solutions must also be implemented in such a way as to allow them to be extended by later applications, which may require additional functionality. Modularity ensures the ability to replace components of a control system with newer or alternative counterparts, providing the ability to adapt to changing requirements. Interoperability is essential for large-scale integration applications and can only be achieved through the use of agreed standards. However, these standards should not restrict the capability of a platform and therefore need to be customisable while maintaining backwards compatibility. If these four principles are followed, it is then possible to create standard integrated user interfaces to all systems, providing users and developers with a standardised experience.

CorteX is built around a self-describing protocol that contains both the data in the system and supporting metadata. The structure of data within the system is similar to that found in other robot control system platforms, such as ROS [40]. The key difference is that CorteX communication messages are standardised; there is only one message type. In the proposed interoperable solution, data belonging to a particular component is complimented with a combined architecture and inheritance type structure of the component, allowing for interpretation and discovery of previously unknown systems. The implementation of this solution takes the form of a core library containing standardised structures for storing the data and metadata. Additionally, a selection of supporting libraries have been developed to allow communication between instances of these data structures to create a distributed system.

The proposed solution delivers a long-term maintainable and extensible robotic framework including an interoperable communication standard, control methods applicable to current robotic technologies, and validation routines to test the stability of the developed platform.

The chapter is organised as follows. Section 2 provides background information on the sectoral requirements of the nuclear industry, reviews the related academic work, and available market products in the areas of robotic middlewares and control platforms. Our specific problem definition is presented in Section 3. Section 4 covers future-proofing, interoperable implementation of the proposed framework.

Section 5 describes the evaluation techniques and analysis of the experimental results. The chapter is closed with software infrastructure quality and provides information on software maintainability in Section 6 and the main conclusion of this study is presented in Section 7.

2 Background and Related Work

2.1 Software-engineering Requirements in the Nuclear Industry

High energy physics research devices are often large, and complex environments that are hazardous to humans due to temperature, radiation and toxic materials. Joint European Torus (JET) [50] experimental fusion reactor is the only active one in Europe, whereas a few others including the ITER [3], the European Spallation Source (ESS) [13] and DEMO [18] are currently under construction across the world.

Systems that will operate future fusion powerplants are expected to integrate hundreds of systems from multiple suppliers with multiple bespoke interfaces and training requirements. The lifespans of these reactors are expected to exceed 30 years, where maintenance and reconfiguration requirements evolve and obsolescence management is required. Such operations are typically conducted using tele-robotic devices and remotely controlled tools and equipment. Tele-robotic systems used for the remote handling and maintenance themselves require frequent reconfiguration to adapt to the evolving needs of the facility and maintenance operations (i.e. tasks required to perform the maintenance activities). Furthermore, operations carried out in these facilities are often time, safety or mission-critical, causing an additional burden on the control systems in terms of high-fidelity, real-time performance to successfully meet the time, safety or mission goals.

In the scientific context, sectoral requirements and challenges presented above can be translated into software requirements for long-term maintainability, extensibility, high-fidelity and interoperability. However, the existing control system architectures used for remote handling and maintenance are highly coupled, often monolithic systems, which fail to adapt to the changing nature of hazardous nuclear environments and often become unmanageable due to ad-hoc, bespoke adaptations causing them to become prohibitively expensive to rectify. An extensive literature review is presented in the next section on the existing market products and control systems explaining why these systems are not well suited for nuclear applications.

The most important criteria for future-proofing long-lived nuclear applications are long-term maintainability, extensibility and interoperability. The design principles for CorteX development have been mostly focused on these three key factors. Nevertheless, performance measures including fidelity and real-time performance are also important for time or mission-critical nuclear applications. First, we define functional requirements that affect performance measures, namely, fidelity and real-time character. Later, we follow defining three non-functional design principles in this section.

In generic terms, *fidelity* is the extent to which the appearance and behaviour of a system / simulation reflect the appearance / behaviour of the real world [42, 15, 44]. The concept of fidelity has two distinctive types: physical and functional fidelity. Physical fidelity refers to the degree of similarity between the equipment, materials, displays, and controls used in the operational environment and those available in the simulation (e.g. EtherCAT, TCP, serial). Functional fidelity refers to how the processes are implemented (e.g. how information requirements are mapped onto response requirements).

Real-time performance of a system can be defined as the systems reaction time to apply environmental changes [9]. Real-time systems guarantee a response within a defined period and missing a deadline have varied affects depending on the constraints on the system. Real-time systems are divided into two categories based on their constraints as hard, soft deadlines [9]. Most nuclear applications contain safety or mission critical (sub-)system (i.e. hard constrained) where a missed deadline is greater damage than any correct or timely computation. Although the solution provided in the chapter is not a real-time system, going forward further development and research is required to make the provided solution real-time given that most nuclear system contain safety or mission criticality.

Maintainability is a concept that suggests how easily the software can evolve and change over time [37]. Measuring maintainability is not a straight-forward task, however, cohesion, coupling and granularity are measurable metrics that collectively provide information about the level of maintainability of the software. Cohesion and coupling are attributes of software that summarise the degree of connectivity or interdependence among and within subsystems. *Cohesion* refers to the strength of association of elements within a system [33, 24]. Gui and Scott defined cohesion as the extent to which the functions performed by a subsystem are related [22]. If a subcomponent is responsible for a number of unrelated functions, then the functionality has been poorly distributed to subcomponents. Hence high cohesion is a characteristic of a well-designed subcomponent. *Coupling* is a measure of independence among modules in a computer program [1]. Coupling can also be described as inter-relatedness among components [24]. High cohesion and loose coupling is a tactic to enhance modifiability of a complex system. As Bass et al. so concisely described it: "high coupling is an enemy of modifiability" [4]. *Granularity* is associated with the size or the complexity of system components [24, 21]. As the granularity size of a module increases, the probability of interchangeability decreases. On the other hand, as the granularity size of a module decreases, the maintenance of the system gets more complex. Low coupling between modules and high cohesion of a fine-grained module are desired properties in the modular architecture design [21].

Extensibility is a design principle that provides for future growth. Extensibility is a measure of the ability to extend a system and the level of effort required (i.e. cost, development time, development effort) to implement the extension [31]. Extensions can be through the addition of new functionality or modification of existing functionality. The principle provides for enhancements and increases in software consistency, through reusing system components where possible, without

impairing existing system functions. Literature in the field of software engineering collectively suggests a strong relationship between extensibility and highly cohesive, loosely coupled, highly granular software.

Interoperability is defined as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform [46]. In the context of software design, modularity or modular software components/programs help improve software reliability, allows multiple uses of common designs and programs, make it easier to modify programs (i.e. improve modifiability) and support extensibility.

Component-based software engineering is a field that focuses on creation and maintenance of software at a lower cost with increased stability through the reuse of approved components in flexible software architecture [23]. We believe the key to achieving long-term maintainability and extensibility whilst developing modular, reusable, interoperable architecture that holds high-fidelity, deterministic functional characteristics is through developing a component-based software solution. Using modern object-orientated design paradigms, encapsulation, low coupling between components and high cohesion of a fine-grained components will help us to achieve the desired functional and non-functional sectoral requirements that long-lived nuclear facilities require.

2.2 Related Work

There are several existing middleware, communication frameworks, and control systems in the market able to perform some of the tasks required for remote handling robotic systems and that could be considered for use in the nuclear sector. Whilst existing frameworks allow for robust integration of complex robotic systems, they are not compatible with highly efficient maintenance and continues extension in the face of changing requirements and obsolescence issues over decades-long periods.

Simulators play a critical role in robotics research as these tools facilitate rapid and efficient testing of new concepts, strategies, and algorithms [32]. The Player/Stage project [19] began in 1999 as one of the first distributed multi-robot platforms, and has widely been used among the research community. Player [20] is a robotic device server allowing offline development of robot control algorithms and is also capable of interfacing with robotic hardware. Although Player does not provide a high level of fidelity, or determinism, it has been well accepted as an open-source, general-purpose, multi-lingual robotic development and deployment platform. Stage is a configurable, lightweight 2D robot simulator capable of supporting large multi-robot simulations. Given that scalability is one of the most important aspects of the studies involving multi-robot research, Player/Stage provide a balance between fidelity and abstractions for its users. 3D dynamics simulator Gazebo [32] is also developed within the Player/Stage Project at first as one of the components; later it became independent. Gazebo is integrated with the Open Dynamics Engine (ODE) [48], Bullet [11] and a few other high-performance physics engines, and therefore is

capable of simulating rigid body dynamics. To facilitate fast and complex visualisation, Gazebo chose OpenGL and GLUT (OpenGL Utility Toolkit) [30] as the default visualisation tools for 3D rendering. Today, Gazebo continues to gain popularity not only in the robotic research community, but also in industry and it can simulate complex, real-world scenarios with high-quality graphics.

Robot Operating System (ROS) [40] came after the Player/Stage Project as a follow-up project with the intension of implementing a more modular, tool-based, re-usable system. ROS is an open-source, multi-lingual platform, primarily used within the academic community. Over the years, it has gained popularity and has also been accepted in industry, for non-critical applications where time, mission, safety-criticality, and QoS are not required. ROS is a powerful tool. It provides a structured communications middleware layer which is designed around commonly used sensory data (e.g. images, inertial measurements, GPS, odometry). Although the structured messages promote modular, re-usable software, ROS messages do not cope well with the continuously evolving nature of software, causing compatibility issues. The highly coupled solutions created in ROS create issues for lasting maintainability and extensibility, crucially important factors for large scale industrial systems. Integration of ROS components is fairly easy for small-scale projects, but they are not practical solutions for large-scale engineering problems due to the efforts required for integration and modification when the system configuration changes (i.e. not extensible easily).

The second generation of Robot Operating System, ROS2 [36], provides deterministic real-time performance in addition to the existing ROS features. Proprietary ROS message formats are converted into Distributed Data Service (DDS) [45] participants and packages; thus providing a high-performing, reliable communication backbone which helps to achieve determinism at the communication layer. ROS2 is backwards compatible with ROS via message converters. Although ROS2 has resolved the reliability, timeliness, determinism and high-fidelity issues ROS previously contained, it has not resolved the maintainability and limited re-usability issues for large-scale engineering problems, as there is no change in message structures.

Evolution of Player/Stage to ROS2, took over 20 years, with thousands of contributors to the open-source platform developing extraordinary, state-of-the-art robotic research on it. Player/Stage, ROS and ROS2 frameworks are important milestones in today's robotic community and technologies. These three platforms are some of the shining examples of visionary research, leading into the development of ROS2 platform for wide industrial use. Despite all the useful tools ROS and ROS2 contain (e.g. rviz, relaxed_ik, Gazebo), the structured message types offer limited capacity to support interoperable, future-proof, modular architecture required/desired in long-lived nuclear facilities.

Fig. 1 illustrates modules of a control stack on the left of the image and lists the associated products capable of carrying out the set of tasks required in each module on the right. Hardware is placed at the bottom of the stack, and the applied control principles (therefore the modules drawn) become more abstract going further away from the hardware. Hardware communications are the protocols that interact with the hardware directly. Fieldbus protocols (e.g. EtherCAT, Modbus, PROFIBUS, Con-

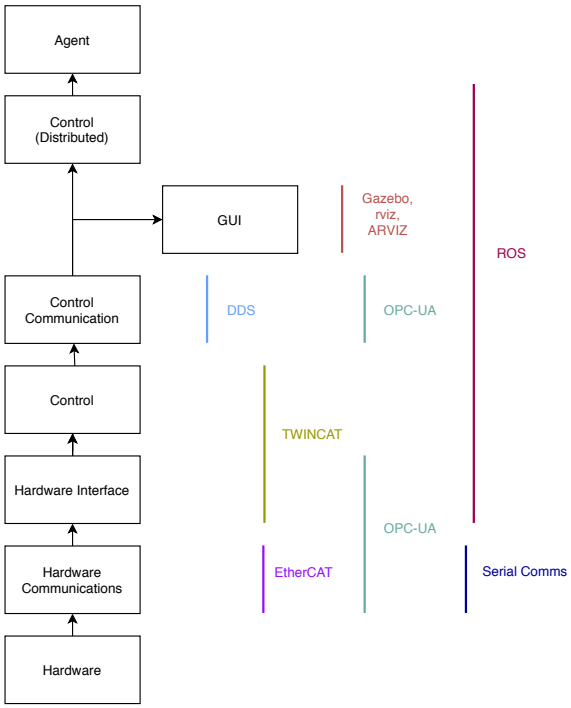

Fig. 1 Comparison between market products on control system stack.

trol Area Network (CAN) bus, serial communications) that are standardised as IEC 61158 for industrial use, are listed in this category. The hardware interface module represents the fieldbus network (e.g. TwinCAT, Modbus, PROFINET, Control Area Network (CAN) open, OPC-UA). TwinCAT contains an operating system that hosts control systems. It also has a built-in EtherCAT master that interfaces to the fieldbus network. Modbus can run over TCP, UDP or RS485. OPC Unified Architecture (OPC-UA) [26] is a machine-to-machine communication protocol used in industrial automation under IEC 62541 specification. OPC-UA TSN [7] is a specific version of OPC-UA that implements the IEEE 802.1 standards for time-sensitive networking (TSN). These add deterministic behaviour to standard Ethernet and therefore guarantee Quality-of-Service (QoS). Unlike regular OPC-UA, which is a client-server application, OPC-UA TSN introduces a publish-and-subscribe (Pub/Sub) model for OPC-UA. The Control module represents particular control algorithms and tools used on a local machine. TwinCAT, ROS, ROS2 have control ability integrated and can be used to operate hardware that is connected to a local machine running the control algorithm. In order to achieve a distributed control system, the information from a local machine has to be distributed over a network.

Frameworks	Open Source	Dev. Language	Distributed	Real-Time	Hardware Intr.	GUI	Scalability	Extensibility	Interoperability	Security	Physics Engine	Control System
DDS [45]	Yes	C++	Yes	Yes	No	No	High	High	High	Compatible	No	No
OPC-UA [26]	Yes	C,C++,Java,Python	Yes	Yes	Yes	Yes	High	Medium	High	Built-in	No	Limited
ZeroC ICE [25]	Yes	C++,C#,Java	Yes	Yes	No	Yes	High	Low	High	High	No	No
Player [20]	Yes	C,C++	Yes	Yes	Yes	Compatible	Medium	Low	Low	Low	No	Internal/Limited
Stage [19]	Yes	C,C++	No	No	No	Yes	High	Low	None	Low	No	External
ROS [40]	Yes	C++,Python,Lisp	Yes	No	Yes	No	Medium	Medium	Low	Low	No	Internal
ROS2 [36]	Yes	C++	Yes	Yes	Yes	No	Medium	Medium	Low	Built-in	No	Internal
Gazebo [32]	Yes	C,C++	No	No	No	Yes	Low	Low	Low	Low	Yes	External
Hop [47]	Yes	Bigloo+JavaScript	Yes	Yes	Yes	Yes	High	High	Medium	Medium	No	Internal/Limited
MSRS [29]	No	C#	Yes	No	Yes	Yes	Medium	Low	No	Low	Yes	No
ARIA [5]	Yes	C++	No	No	Yes	Yes	High	High	No	Low	No	Internal/Limitted
ASEBA [34, 35]	Yes	C	Yes	No	Yes	No	Medium	Low	No	Low	No	Internal/Limited
Carmen[52, 51]	Yes	C	Yes	No	Yes	Yes	Low	Low	No	Low	No	Internal/Limited
CLARAty [53]	Yes	C++	Yes	No	Yes	No	Medium	Medium	High	Low	No	Internal/Limited
CoolBOT [16, 17]	Yes	C++	Yes	Yes	Yes	No	Medium	Medium	Low	Low	No	Internal/Limited
Orocos [8]	Yes	C++	Yes	Yes	Yes	Yes	Low	Low	Low	Low	No	Internal
MOOS [39]	Yes	C++	No	No	No	No	Low	Medium	None	No	No	Internal/Limited
Iris	No	C++,C#	Yes	External	External	Yes	High	High	High	External	External	External
CorteX	No	C++	Yes	Yes	Yes	Yes	High	High	High	Compatible	Compatible	Internal

Table 1 Comparison Between Control Systems and Middleware Frameworks.

The Control Communications module refers to middlewares (e.g. DDS, OPC-UA, MQTT, ZeroC ICE) that can be used to drive the information out from a local machine to networked devices. The data-centric Pub/Sub protocol Data Distribution Service (DDS) OpenSplice [6] offers highly dynamic, timely, reliable QoS. Device-centric OPC-UA [26] standardises the communication of acquired process data, alarm and event records, historical data and batch data to multi-vendor enterprise systems. The standardised communication process allows users to organise data and the semantics in a structured manner, which makes OPC-UA an interoperable platform unique for multi-vendor, industrial systems. To ensure interoperability and increase re-usability, standardised but extensible base message types are provided by the OPC-UA Foundation. The Message Queuing Telemetry Transport (MQTT) protocol [2] provides a lightweight and low-bandwidth approach which is more suitable for resource-constraint internet-of-things (IoT) applications and machine-to-machine communications and is orthogonal to OPC-UA, but not interoperable like OPC-UA. ZeroC ICE [25] provides a remote procedure call (RPC) protocol that can use either TCP/IP or UDP as an underlying transport. Similar to DDS, MQTT and OPC-UA, ZeroC ICE is also a client-server application. Although asynchronous, event-driven nature of ZeroC ICE makes it unsuitable for real-time applications where QoS and durability are key; the same characteristic helps improve scalability. Its neatly packaged combination of a protobuf-like compact IDL, an MQTT-like architecture, some handy utility executables to run brokers, autodiscovery features, and APIs in half a dozen languages make ZeroC ICE a popular middleware choice for non-real-time applications. Createc Robotics has been developing Iris [27], an open platform for deployment, sensing and control of robotics applications. Iris combines 3D-native visualisation, a growing suite of ready to use robotics applications and system administration tools for application deployment. As a platform, Iris intends to introduce an open standard designed to enable high-level interoperability of robotics and telepresence system modules. Iris is not a framework or middleware in itself, but aims to provide an abstraction layer to a growing number of middlewares such as ZeroC Ice, and ROS.

Table 1 compares and categorises existing frameworks and middleware products in terms of source openness, development language, distributed or centralised control, deterministic real-time characteristic, interface to hardware features, existing graphical user interface (GUI), scalability, extensibility, interoperability, security, physics engine and control system capabilities. Source openness encourages transparency, correctness and repeatability and is extremely beneficial in academia. The openness of Player/Stage, ROS and ROS2 to the robotics community is invaluable in this regard. Within the nuclear industry open-source applications are not preferred due to the perceived security risks that arise. OPC-UA and ROS2 have built-in security authentication, whereas DDS and IRIS are compatible with secure protocols. Single points of failure should be avoided at all cost. Distributed and scalable architectures are desired for the nuclear industry. Modularity encourages reusability; a key to achieving maintainable and extensible software is through highly granular, loosely-coupled and highly cohesive design. Although ROS, ROS2 and IRIS are modular and encourage reusability, integration efforts to put in these frameworks

suggests that they are not easily extensible. This results in lasting maintainability issues which are far from being ideal for long-lived nuclear applications.

Gazebo is a 3D dynamics simulator, whereas rviz is a visualisation tool. CorteX comes with several integrated GUIs, and example screens are illustrated in Section 4.6. Orocos [8], Open Motion Planning Library (OMPL) [49], OpenRave [14] and Robotics Library (RL) [41] are robotic calculation libraries providing kinematics, motion planning including 3D paths and trajectory estimations, vector translations, etc. Orocos and ROS were integrated to combine the deterministic real-time aspect of Orocos with large ROS control paradigms.

The Experimental Physics and Industrial Control System (EPICS) [12] was designed by Los Alamos National Laboratory, and ITER has identified EPICS as a standard operating framework for all of ITER control systems [43, 10]. EPICS is a distributed process control system built on a software communication bus. As such, it provides brook-less communications where computer processes run EPICS databases that represent system units. EPICS databases held records of functional algorithms used for the system. EPICS is capable of running on RTOS and provides deterministic real-time performance with fairly high-fidelity. ITER has identified EPICS as its control system framework due to EPICS wide range of functionality, rapid development and modifiability and extensibility characteristic. As such, ITER's choice of control system clearly shows the importance of maintainability, extensibility, and high-fidelity for nuclear operations.

ASEBA [34, 35] is an event-driven, distributed, lightweight simulation platform for mobile robotics. CLARAty [53] developed by JPL and has a multi-layer abstraction model to ensure interoperability. The control architecture promotes reusable components and modularity through using layers of abstraction, thus assuring extensibility. Lower-levels of abstraction focus on integration of devices, motors, and processors, whereas high-level of abstraction integrate the lower-level abstractions to provide higher-levels of functionality such as manipulation, navigation, trajectory planning, etc. From this perspective, CLARAty aims to achieve maintainable and extensible frameworks, where CorteX provides timeliness and determinism which CLARAty lacks.

RACE UKAEA is a UK government-funded lab, pioneering technologies on industrial problems, providing state-of-the-art solutions to niche problems in the nuclear industry. CorteX attempts to solve the main problems associated with interoperable, plug-and-play, distributed robot systems-of-systems, at least from a data/communications perspective. CorteX is designed from the ground up to work as a decentralised, distributed control system compatible with Pub/Sub, Service-Oriented application. Although DDS is used to distribute information across the CorteX network, the middleware agnostic nature of CorteX allows it to interface to ZeroC ICE, OPC-UA or ROS. This means that unlike ROS or custom solutions using plain TCP or other transport layers, there's no central 'CorteX server' to set up, configure, and act as a potential single point of failure.

3 Problem Statement

The problems facing the development of control system software, or in this a framework, that is capable of supporting an application such as nuclear remote handling can be grouped into four main categories: interoperability, maintainability, extensibility and performance.

Interoperability In order for a control system to support the various bespoke interfaces that are required when integrating hardware from a range of suppliers, while concurrently reducing integration efforts, a high level of interoperability is required.

The framework should have an architecture and supporting middleware that aims to abstract application specific data and function, in order to allow as much of the system to be application agnostic as possible. Part of this abstraction is the standardisation of interfaces, which can be applied both at the architectural level and within the middleware.

A standard interface within the middleware will allow the various components of the system to be interoperable at the fundamental level, i.e. discovery and exploration of data and functionality within the system.

As much standardisation as possible within the architecture, i.e. in the contents of data and functionality, will maximise interoperability between components of the system and allow them to be reused and replaced. This also increases maintainability.

Maintainability The lifespan of robotic applications in the nuclear sector, and their supporting control systems, is usually in terms of years and decades. This creates a significant challenge supporting a list of requirements that will evolve as the tasks the control system is to perform change over time. The introduction of new technologies required to support these tasks and avoid obsolescence, requires the control system software to be easily modifiable while ensuring the effort required to do so remains low.

A high level of modularity in the design and structure enforced by the framework architecture will ensure components of the system can be swapped out while minimising the impact on the rest of the system. As previously mentioned, this is also strongly coupled to the standardisation of the interfaces between components so as to abstract as much of the control solution from the hardware as possible.

The modular nature of a system with this structure also allows testing at the component level. This is advantageous when replacing or altering components as usually little to no modification is required to re-run the test with the new changes. This means changes can be made with a higher level of confidence as the test ensures the same level of functionality is maintained.

Extensibility A change in requirements often requires an increase in functionality. Where possible, the additional functionality should be added without altering the previous components of the system. Therefore, any errors introduced are unlikely to be at the expense of the original functionality, and result in a higher level of reliability. The ability to extend the capabilities of a system, while minimising the components affected can be achieved with a highly cohesive, loosely coupled, high granularity architecture.

Performance While the previously described features are beneficial to control systems for nuclear applications, they cannot come at the expense of performance. Robotic systems must be reliable, predictable, and responsive in order to meet their critical system requirements.

To ensure performance is maintained the proposed control system software is evaluated against the following metrics:
Loop Cycle Duration - the time between two consecutive tick signals.
Loop Cycle Jitter - the deviation in the loop cycle duration from a set frequency.
Loop Cycle Duty Cycle - the percentage of the loop cycle duration spent executing the job.
Overflow Count - the number of ticks missed because the previous job has not been completed.
Command Latency - the time between a simplex sending a command to another and the command being executed.

In the next section, we describe our proposed solution, CorteX, in detail and explain how we address these challenges.

4 CorteX Design

CorteX attempts to solve the main problems associated with interoperable, plug-and-play, distributed robot systems-of-systems, at least from a data and communications perspective.

CorteX could be thought of as:

- A standardised graphical data representation for robotic systems that can be modelled as directed graphs;
- A method for communicating this representation;
- A software framework that implements the above;
- Additional software tools to add functionality.

To achieve a highly modular, loosely coupled, highly granular system infrastructure, CorteX implements a building blocks methodology. Required functionality is provided by bringing together plug-and-play building blocks. CorteX consists of several modules, namely CorteX Core, CorteX CS, CorteX Toolkit, CorteX Explorer, VirteX, and many other potential future extensions.

CorteX applies an inherently concurrent architectural design through the use of object-orientated design methodologies and implements the imperative paradigm within the single methods only (i.e. control system functions), and for main executables. The concept of distributed objects and actor model design methodology is used. CorteX Core and CorteX CS, the communication backbone and the control system module, are set up to work with real-time constraints, where possible. The

main network middleware layer (i.e. DDS) is chosen specifically for its real-time capabilities (DDS is able to operate as a real-time publish-subscribe protocol).

In the rest of this section, we describe the modules of CorteX in detail. We will start by exploring the standardised interface in section 4.1 and follow-up with an explanation on how we achieve interoperability within CorteX using the standardised interface in section 4.2. Section 4.3 will describe CorteX Core that is the data, and the control system module CorteX CS will be described in section 4.5. This section will end with several examples on CorteX's human-machine interfaces in section 4.6.

4.1 Standard Interface

A system in the CorteX environment can be represented with a graph consisting of nodes and edges, similar to the graph presented in Fig. 6. Every node in a CorteX environment is called a *simplex* and every edge denotes information flow between simplexes. Each simplex uses the same format for internal data representation and has the same external interface. This data representation can be used as part of a communications protocol to allow distributed components of a single control system to exchange data without prior knowledge of each other. This means a CorteX control system can grow to incorporate new hardware and control features without modifying other distributed components.

The term *simplex* is used to describe units of data within CorteX, facilitating functionality within the control system, including: defining data, describing input and output relationships, calling functionality, representing (a part of) a system, controlling an active element (i.e. controller), or communicating with other simplexes within the system.

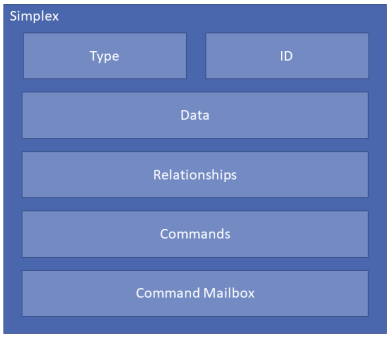

Fig. 2 Simplex

A typical *simplex* $s_m \epsilon$ S, a self-contained unit of information, is the tuple: $s_m =< type_m, id_m, data_m, rlsp_m, cmd_m, cmdbox_m >$

$type_m$: Set based to the ontological structure;
id_m: Unique identifier, including path;
$data_m$: Either an int, float, bool or string. May be a single element or an array. Auto-set by the architecture based on the ontological type;
$rlsp_m$: A link to another *simplex*, in order to obtain information or call a command. It may be a single element, or an array. This is automatically generated and set based on the ontological type;
cmd_m: A function which can be called in order to perform a task or change behaviour. Commands are declared with a list of request parameters, which are sent with the call, and response parameters, which are returned upon completion.
$cmdbox_m$: A standardised interface to exchange commands.

simplexes are common building blocks in a CorteX environment, at a level of granularity associated with minimal reusable units. Each simplex has a type and in the next section, we describe types in detail.

4.2 Interoperablility

The basic concept behind the self-describing, distributed data model is built upon simplexes with types that are associated with a software ontology and morphology techniques. We have used software ontologies to provide semantic meaning to robotic and control systems components, and have implemented morphological rules to associate syntax with the components represented in the system. In the remainder of this section, we describe applied ontology and morphology techniques using a case study to illustrate the principles described.

Software ontologies have been around since the early 1990s and applied broadly in multiple disciplines. Software ontologies aim to build the structure of information of a certain domain, to share a common understanding of the information available in that specific domain. Computer agents can extract and aggregate information shared within a structure to make the best use of the presented information in an ontology. Ontologies, by creating a structure for information, help increase common understanding in the following ways:

- to enable reuse of domain knowledge;
- to make domain assumptions explicit;
- to separate domain knowledge from the operational knowledge;
- to analyse domain knowledge.

CorteX makes use of an ontology to ensure common and consistent use of domain knowledge and to make domain assumptions explicit. CorteX shares the domain-specific structure (the robotic and control system ontology) with every agent that runs CorteX, before execution. At runtime, distributed CorteX agents can make explicit

assumptions using the knowledge represented in the ontology, thus ensuring the consistent reuse of distributed domain knowledge among agents. Fig. 3 illustrates some elements of an example CorteX ontology relating to robotic and control systems.

In Fig. 3, the *Manipulator* type model is split into four categories. In addition to traditional serial and parallel manipulator categorisation, the ontology presented in this chapter also considers gripper and one degree-of-freedom (DOF) categories under the *Manipulator* subgroup. *Kuka LBR* is placed under the *Serial Manipulator* category. Using the ontology built for the type inheritance model structure, the CorteX system is capable of discovering that the *Kuka LBR IIWA* type also inherits the *Serial Manipulator* interface, which can be used instead, if needed. Careful examination can detect that *Robotiq 2-Finger Gripper* is classified under *1 DOF Manipulator* as well as being listed under the *Gripper* type model. This is not a mistake; one can use the same technique to operate a 1DOF manipulator and a Robotiq 2-finger gripper. Therefore, we decide to allow multiple occurrences in this ontology-based knowledge structure. The effects of multiple inheritance and polymorphism on the ontology and interoperability is out of the scope of this chapter and will be analysed in the future separately.

The *Concept* type in this case study represents pure data modules, and they are used as the inputs and outputs for processing simplexes. CorteX applies standardised (but extensible) data blocks and assures standardised data exchange between blocks. The standardised information exchange is guaranteed by the standardised simplex interface. In Fig. 3, Concept data types are categorised into three example categories: axis, cartesian and digital concept type models.

The ability to use a simplex of a given type at various levels of its inheritance hierarchy allows for standardisation but also extension, in the form of polymorphism. This is one of the advantages of using an ontological type system.

The second feature of the CorteX type model is that it does not require prior knowledge. The type model is distributed at runtime and can be completely customised for the particular application, rather than using predetermined types. If two systems are required to interoperate then the types within their type models must not contradict each other, but the overall models do not need to be identical.

Consider the following example: Agent A declares an Axis type, and it contains position, velocity, and acceleration data. Agent B declares an Axis type but states that it contains position, velocity, and torque data. These two agents are not interoperable as they do not agree on the definition of the Axis type.

However, now consider this example: Agent A declares the Axis type as before. Agent B declares an Axis type with a definition that matches Agent A, but *also* declares another type that inherits from Axis, states it contains torque data, and calls it AxisWithTorque. These two are now interoperable as they do not conflict in their definitions of any types, but Agent B can extend the Axis type to include their

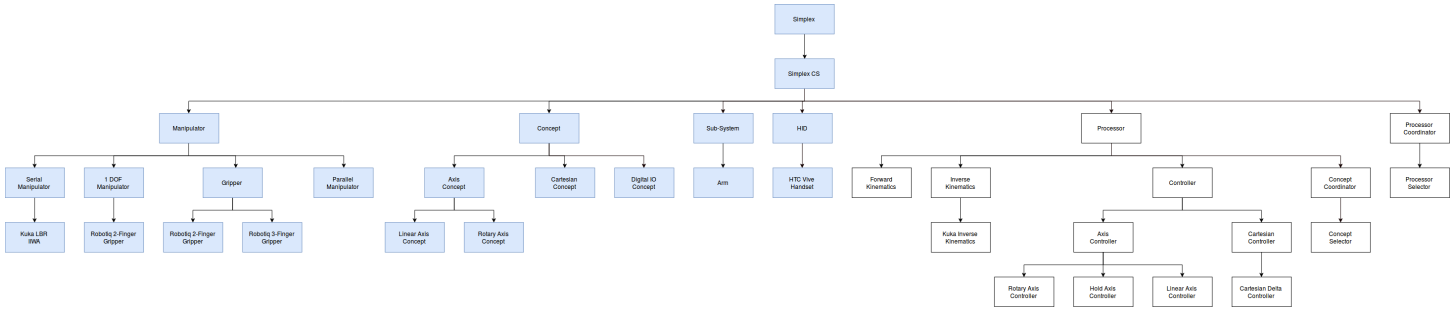

Fig. 3 Robotic and control system ontology - type model. Within our implementation, type modules fall into two main categories: descriptive shown in blue) and active shown in white. Descriptive modules, as the name implies, are there to describe the state of the system, either currently (which we refer to as an observation), or as we wish it to be (which we refer to as a modification). In terms of hardware, an observation contains data read *from* the hardware, and a modification would contain data to be written *back to* the hardware as a demand. Active modules, on the other hand, are used in full CorteX control systems (CorteXCS) in which functionality has been added, in order for the modules to manipulate the data during its loop cycle task. In a system where the functionality was provided by some other means, for example a ROS implementation, the CorteX system would likely be comprised of only descriptive modules. Values from ROS would then be injected into the simplex data, in order for it to be communicated. Although the "role" of these two categories of modules differ, this difference is purely implicit and is not reflected in their structure or contents.

required data. Not only does this mean that Agent B can add their data to the system, but Agent A can also use the simplex of type AxisWithTorque with any interface designed for a simplex of type Axis as one inherits from the other, and therefore must contain all the required information from the Axis type.

Arm is listed under the *Sub-System* category, whereas *HTC Vive Headset* is placed under *Human Interaction Device (HID)*. One can think that UAVs and UGVs categories are entirely missing, or ask why different brands of headsets and robotic arms are not represented. Fig. 3 does not illustrate the full ontology CorteX implements; the figure only contains the required knowledge to represent the case study presented towards the end of this subsection, shown in Fig. 6. Please get in touch with the authors to find out details of the full ontology.

Inverse and forward kinematic modules, different types of controllers and concept (data) coordinators are all categorised under the *Processor* type model, whereas *Processor Coordinator* has associated only with the *Processor Selector* type model in the provided hierarchy. Fundamentally, kinematics modules are responsible for the motion of objects and the forces required to provide the motion.

Morphology is the study of form and structure. In linguistics, it generally refers to the study of form and structure of words. In the fields of computer science, computational linguistics have been used to analyse complex words to define their component parts or to analyse grammatical information.

Within the CorteX framework, we use ontology to build a common structure of the domain-specific information, to distribute and reuse to make explicit assumptions. Therefore, a robotics and control system ontology is used to provide *semantic* meaning to components of robotic and control system elements, represented by simplexes. On the other hand, morphology is used to provide a set of rules to enforce the *syntactic* meaning and a binding structure to the component types represented in the ontology to achieve operational success among the distributed components, and allow explicit assumptions to be made with regard to the structure of a given system, to facilitate interoperability.

The types that are provided in the ontology, not only define functionality and structure, but also data represented, and external interfaces. Relationships in the context of components presented in the ontology define connections to other components. Types can define not only the relationships a component must have (to be considered of that particular Type), but also how many (minimum, maximum, or absolute) components must be related to it, and define a particular Type that the related components must be. The result of these relationship rules is that a system develops a particular *morphology*, which is consistent between all systems using common types. These morphologies tend to fall into one of two distinct groups: structural as shown in Fig.4 and behavioural as shown in Fig.5.

Structural morphologies are either used to describe how the physical counterparts of descriptive simplexes are connected in reality, or to compose several granular descriptive simplexes under another. Some structural morphologies can be seen in Fig.4. The *Arm* (Fig.4, top left) is an example of how relationships can be used to describe physical assemblies. In this example an *Arm* is physically comprised of a *Serial Manipulator* and a *Gripper*. This is achieved using two relationship rules:

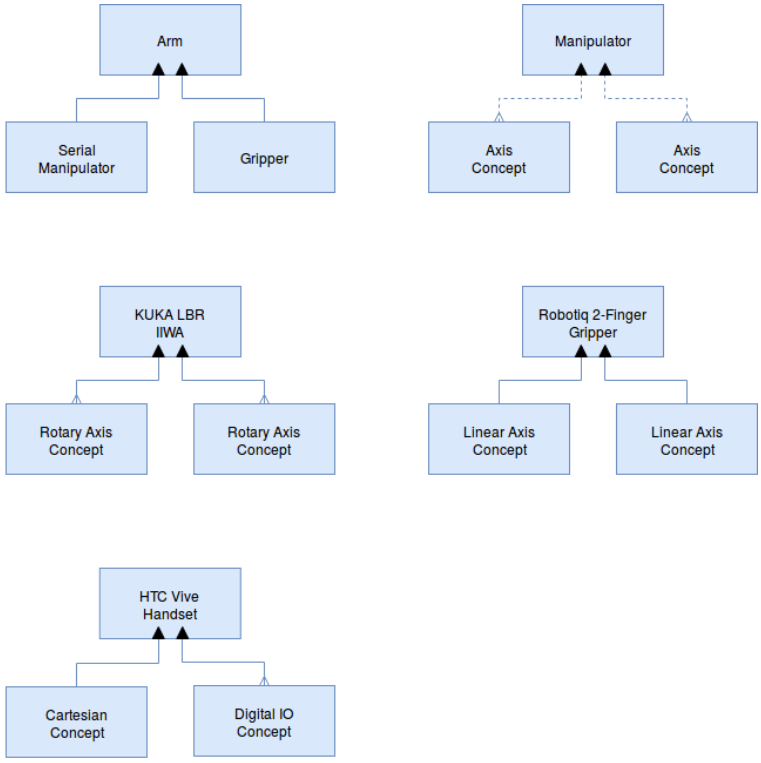

Fig. 4 Robotic and control system morphology - Morphological rules.

the first is an *Arm* must have an input relationship of type *Serial Manipulator*, the second is an *Arm* must have an input relationship of type *Gripper*. This means that an *Arm* component cannot be added to a CorteX system without a *Serial Manipulator* component and a *Gripper* component and the relationships configured to connect them. This makes CorteX systems and their component structure highly discoverable and navigable, as these structures can be recognised and explored.

The *HTC Vive Handset* (Fig.4, bottom left) is an example of composition via relationships. In this case, the *HTC Vive Handset* is described by combining (via relationships) a single *Cartesian Concept* and one or more (signified by the many-to-one symbol) *Digital IO Concepts*. These components are used to describe an HTC Vive Handset's position in 3D space (which is provided by a tracking system) and the state of its buttons. These relationships are both inputs of the *HTC Vive Handset* component as the state of the individual components must be evaluated first, before we have the complete knowledge that makes up the *HTC Vive Handset*. Similar to the *Arm* example, these relationship rules allow us to create recognisable and expected structures within the CorteX component model, which can be used to contextualise groups and derive semantic meaning.

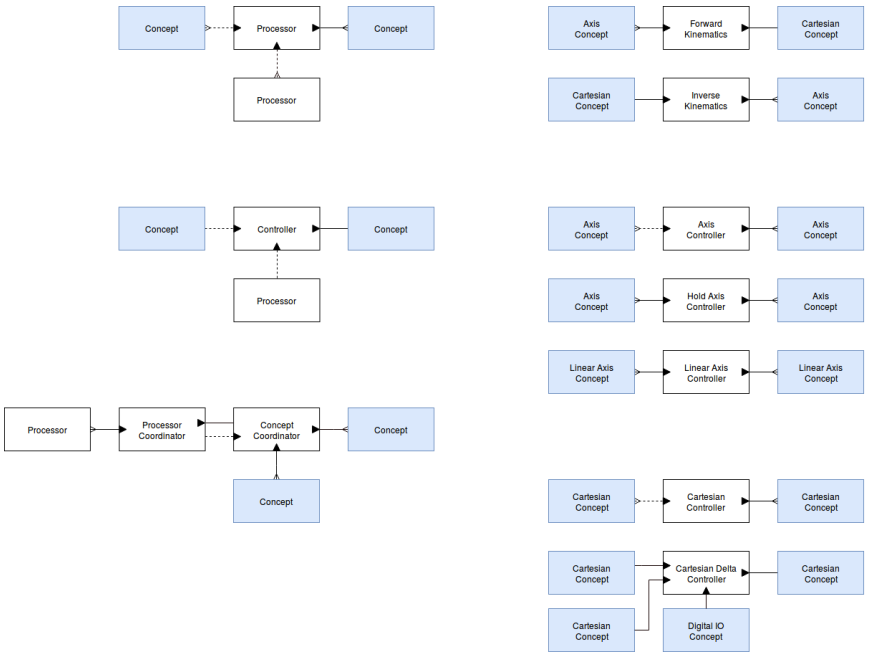

Fig. 5 Robotic and control system morphology - Morphological rules.

In Fig. 4, *Arm*, *Manipulator*, *KUKA LBR*, *Robotiq 2-Finger Gripper* and *HTC VIVE handset* behavioural morphological rules are illustrated visually. Arrowheads in the boxes denote the ownership of the rule; boxes that contain the arrowheads are the owners of the morphological rules and are responsible for implementing the associated rule. Behavioural morphologies are created when combining descriptive and active components, and are therefore common in full CorteX control systems. The *Processor* example (Fig.5, top left), is one of the simplest examples of how descriptive and active components can be used together. This example uses the basic types of *Concept* and *Processor* and has no functional purpose, but is used to establish a template that specific types can follow (similar to a purely abstract class in C++). The *Processor* type defines three relationship rules: 1) A processor *may* have one or more (signified by the dotted line and one-to-many symbol) input relationships of type *Concept*. 2) A processor *may* have one or more input relationships of type *Processor*. 3) A processor *must* have at least one (signified by a solid line and one-to-many symbol) output relationships of type *Concept*. These three relationship rules exist for behavioural reasons, and exist to ensure the processor is able to function. The first rule is to provide input to the processor. This may not be required if the process is "open loop" and is therefore optional. The second rule is designed to allow for sub-processes that may need to be completed first for the processor to function. This may also not be required and is therefore also optional. The final rule is to provide an output for the processor. This is essential, as without an output, the processor is

not making a contribution to the state of the system, and is therefore redundant. In the given examples, the *Arm* type model has two components: a serial manipulator and a gripper. In CorteX, this translates as a robotic component is considered as a robotic arm (see Fig. 4) if it consists of a serial manipulator (e.g. KUKA LBR) and a gripper (e.g. Robotiq 2-Finger gripper). A Serial Manipulator on the other hand, would input and output multiple axis data information, that implement positional move. Therefore, in Fig. 4 the *Serial Manipulator* type model is visualised with axis concepts. *KUKA LBR* which is a 7DOF serial manipulator that contains rotary joints only is therefore represented with rotary axis concepts. *Robotiq 2-Finger Gripper* is a 1DOF linear axis slider and is visualised with linear axis concepts. An HTC Vive Handset can output cartesian position data and digital data, separately. Therefore, in the figure *HTC Vive Handset* type model is visualised as a model containing *Cartesian* and *Digital IO* concepts.

Fig. 5 illustrates some of the morphological rules of control system components. Morphology rules for the *Processor* type model and its sub-categories, including variations of different *Controller* and kinematic type models are presented as well as for the *Processor Coordinator*.

The *Forward Kinematics* example (Fig.5, top right) is a less abstract example of the *Processor* morphology. In this case, the generic *Concept* types have been inherited by more concrete *Axis Concept* and *Cartesian Concept* types. Please note, this change still satisfies the rules established by the base *Processor* type, as both *Axis Concept* and *Cartesian Concept* inherit from the *Concept* type (see Fig.3). As those familiar with control theory will be aware, the purpose of forward kinematics is to derive the tip position of a manipulator using its joint positions. This functionality is partly evident in the morphology, as the *Forward Kinematics* simplex takes one-or-many *Axis Concept* as input (in this case the joint positions) and produces an output of a single *Cartesian Concept* (in this case the tip position). Similarly to the structural morphologies described earlier, these behavioural morphologies help create consistent, discoverable, and navigable structures within the CorteX simplex model.

RACE, within the Robotics and AI in Nuclear Hub (RAIN) project, has been working on a set of complex research problems to find state-of-the-art engineering solutions for decommissioning gloveboxes for the nuclear industry. The tele-operated dual-arm platform, shown in Fig. 7 has been used as a development and deployment platform for decommissioning tasks. The CorteX control system designed for one of the tele-operated KUKA LBRs is illustrated in Fig. 6 and provided in this section as a case study to explore the ontology and morphology principles described above. Fig. 6 illustrates an HTC Vive Handset tele-operated KUKA LBR and its working principle. When the operator clicks on the digital output button and starts moving the HTC Vive handset, the KUKA LBR mimics the move of the handset and moves accordingly. When the operator is not moving the handset, KUKA LBR is been kept on hold using the *Hold Axis Controller*. The *Processor Selector* picks one of the controllers based on the registered delta of the move (i.e. if there is move) and the activity on the *Digital IO Concept* (i.e. if the operator passed the button). *Robotiq 2 Finger Gripper* is operated independent to the manipulator and shown separately.

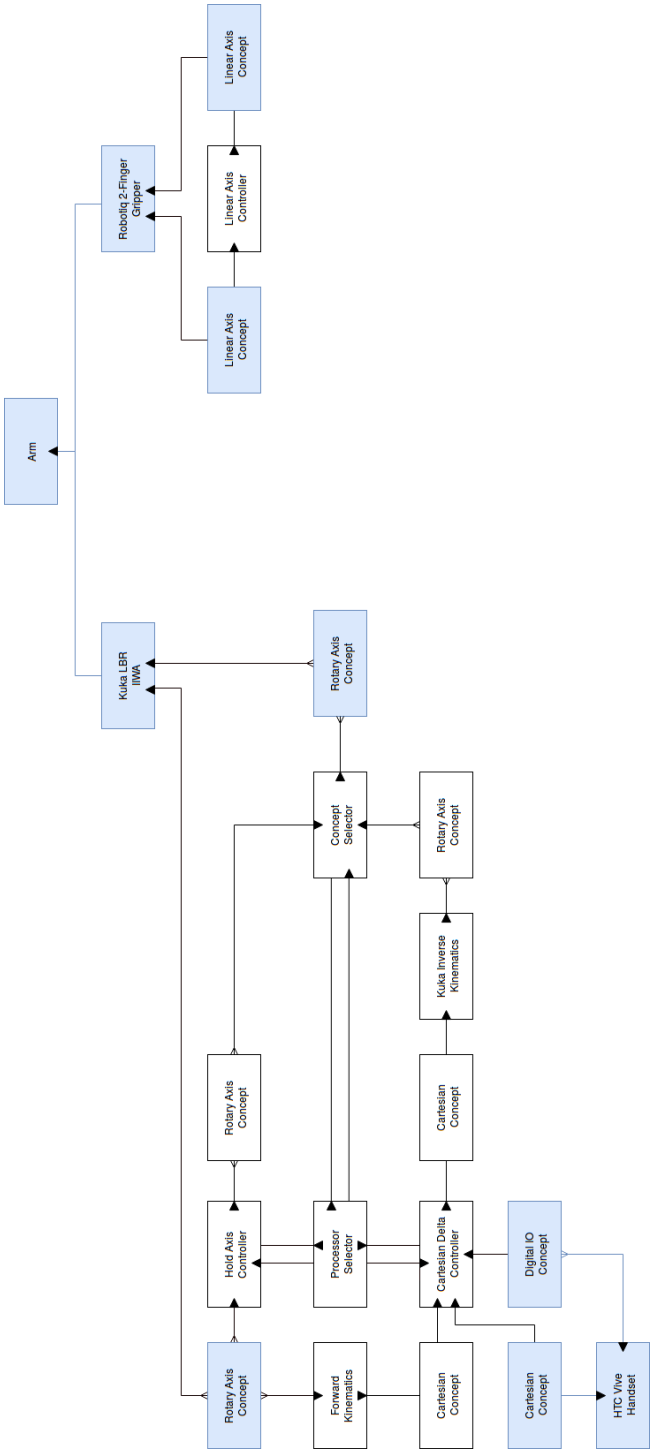

Fig. 6 Case study: System built using the building blocks represented in the ontology and assembled according to the morphological rules.

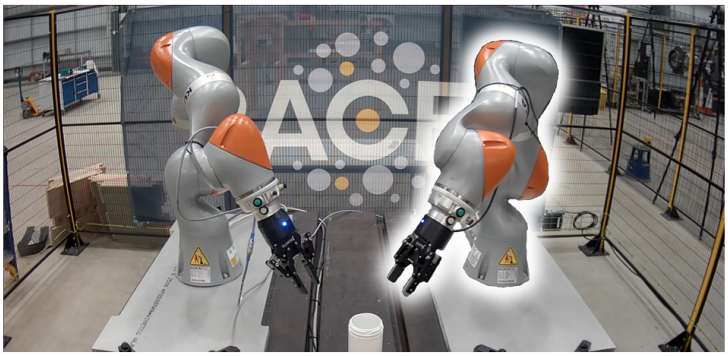

Fig. 7 Tele-operated dual-arm used in the RAIN project and case-study.

In summary, to achieve interoperability, CorteX applies a self-describing model of the system. We decided to use the following concepts to achieve a modular, self-describing and distributed CorteX control system:

- One standard building block type;
- Standardised (but extensible) data blocks;
- Standardised data exchange between blocks;
- These standard structures of the ontology must be distributed.

The common building block types are called simplexes. Each common building type, simplex, have a type, which is the information described in the ontology. In the next section, we will describe simplexes in detail. A small part of the ontology tailored for the case-study has illustrated in Fig. 3. Standardised, but extensible data blocks are the data type model represented under *Concept* sub-category in the ontology. Standardised data exchange between blocks are enforced and implemented through the morphological rules. Some of the rules used in the case-study, are presented in Fig. 4 and 5. In our current implementation, we implement the ontology and morphology in the XML format and distribute it across computer agents that run CorteX to enable reuse of the domain knowledge.

4.3 Core Architecture

CorteX Core is the communication component of CorteX. The CorteX method for communicating data (simplexes), are kept in data tables. The creation of these data tables is initiated by a validation process. Fig. 8 shows the fundamentals of the Core architecture using a UML diagram. Ontology, morphology and the type model that are explained in the previous section are validated in the CorteX Core against a given use-case. For example, the case study presented in Fig. 7 and the associated types of each simplex are validated against the ontology and the morphological rules.

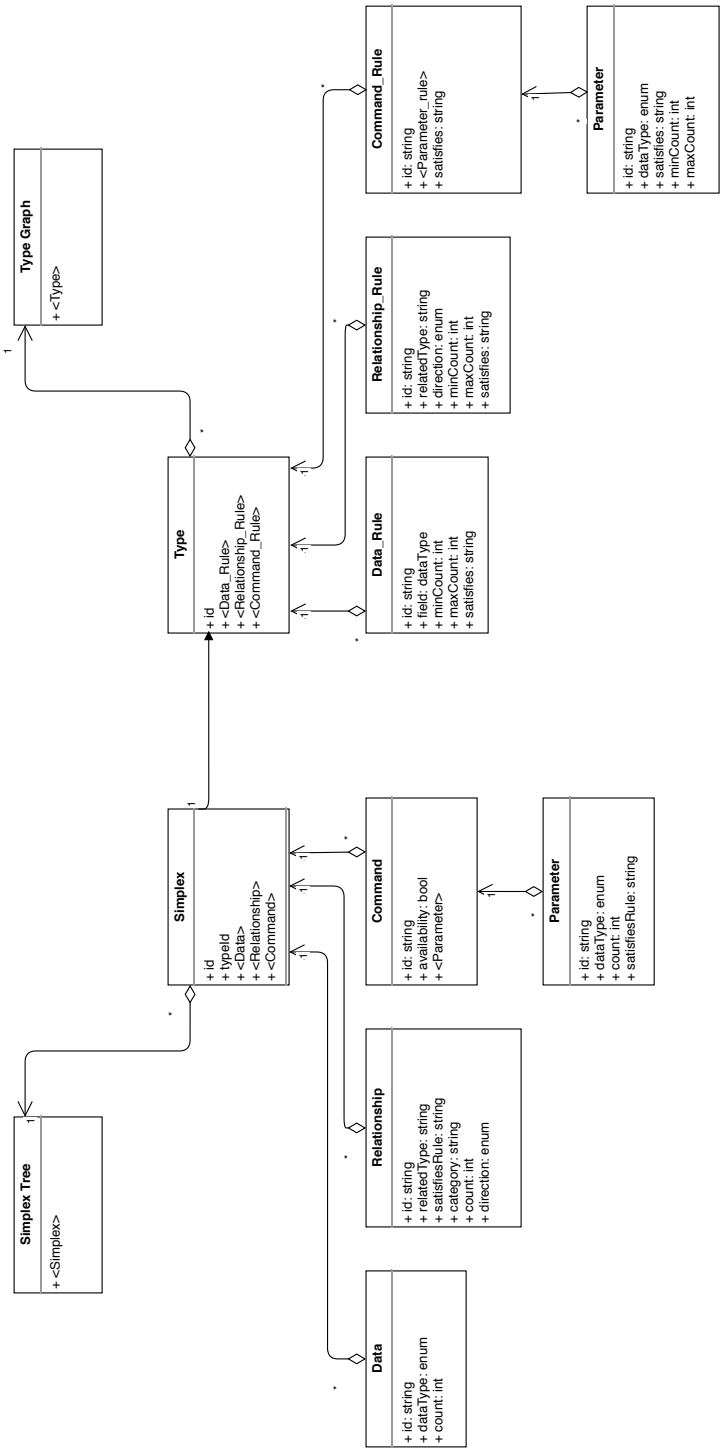

Fig. 8 Core Architecture

This process is performed at the initialisation stage. The initialisation process achieves success if, and only if, all the simplexes can compile correctly; thus require construction of data, relationships, commands and command parameters to match with the associated simplex type and the morphological rules. If the provided application by the user contains simplexes that do not match the ontological structure and morphological rules, the CorteX system does not compile.

Fig. 9 gives an abstract give on the CorteX Core components and illustrates the information flow between these components step-by-step. simplexes update the system and type tables in CorteX core, illustrated with step 1.1. Domain-information is parsed through the XML file, and tables are updated in step 2.1 and step 2.2, respectively. The type model (Fig. 3) and the system model (Fig. 6) informs system tables in step 3.3. For example, when a simplex type, $type_m$, is set to Kuka LBR, as in step 1.1, CorteX auto-generates a *Rotary Axis Concept* in the simplex data field, $data_m$ (step 4.1). The knowledge that a 7DOF Kuka LBR contains *Rotary Axis Concept*s is embedded into the ontology, illustrated in step 2.2 and 3.2, using the morphological rules shown in step 2.1 and 3.1.

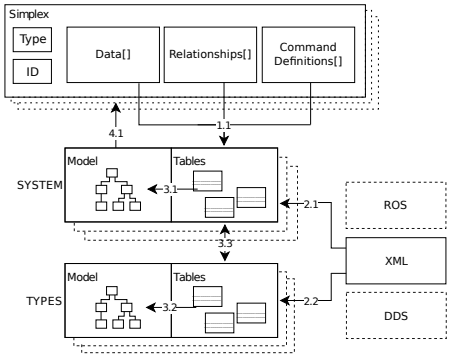

Fig. 9 Core data tables and interfaces.

4.4 Distributing CorteX Data

CorteX agents (i.e at least one cluster of simplexes running on a single PC) can publish their current states to a shared network via the real-time publish-subscribe protocol. This protocol is based on a standard that is maintained by the Object Management Group, and is more commonly known as the Data Distribution Service (DDS). CorteX abstracts the DDS communications and allows users to write new transport methods using an API to access the data inside CorteX core. CorteX also abstracts this interface and allows its users to use an API to construct simplex models in CorteX either programmatically, or as a result of a parsed interface of their choosing.

DDS allows a DDS "participant" to publish tables of data in a broadcast message to all other DDS "participants". DDS participants can filter these broadcast messages and process them as required. This means that as long as two or more CorteX agents are on the same network, and are using the DDS interface, their contents are automatically transmitted to each other. New CorteX agents (Clusters) may also be discovered at runtime, so when new CorteX participants 'come online', they will appear in the DDS interface automatically. CorteX provides fully distributed data transmission and communication via DDS, in (almost) real-time manner. Therefore, unlike MQTT or ROS, there is no need for a central server, as each individual CorteX agent is able to broadcast messages to all other agents in a fully decentralised, distributed manner.

CorteX has interfaces to ROS, which make use of the ROS API, therefore CorteX can read and write to ROS nodes. CorteX converts Interface Definition Language (IDL) files that describe ROS messages into CorteX system model-based structure, which is then accessible and can be processed in a CorteX environment. CorteX currently is only able to read ROS messages and does not translate CorteX commands into ROS service calls, although this is a planned future development.

In summary, CorteX Core is responsible for encoding and extracting system information, making use of the *semantic* meaning of the domain-specific information shared in XML format using ontologies and handling interoperable communications between systems. CorteX CS carries functionality of the control system to meet functional and performance requirements of the robotic control application. CorteX CS is discussed in the next section.

4.5 Control System Architecture

An obvious extension of the CorteX Core paradigm is the introduction of functionality. In addition to simply describing a system, by augmenting the ability to manipulate the data within each simplex, we can produce a system capable of control.

Building on the architecture described in Section 4.3, a full CorteX Control System (CS) extends the original simplex by adding a *task* ($task_m$) resulting in the following *simplexCS* tuple:

$scs_m =< type_m, id_m, data_m, rlsp_m, cmd_m, cmdbox_m, task_m >$

The definitions of $type_m$, id_m, $data_m$, $rlsp_m$, cmd_m, and $cmdbox_m$ are provided in Section 4.3, and used accordingly in simplexCS, therefore we only describe $task_m$ in this section.

Each simplex task is broken up into four main parts. The first is the processing of any commands. These commands may have been sent from a simplex task executed previously in the current job, from a simplex task executed in the previous loop cycle job, or from another distributed agent of the control system. These commands may request to change the behaviour of a simplex, or change the value of a piece of data, or start and stop the functionality of the simplex entirely.

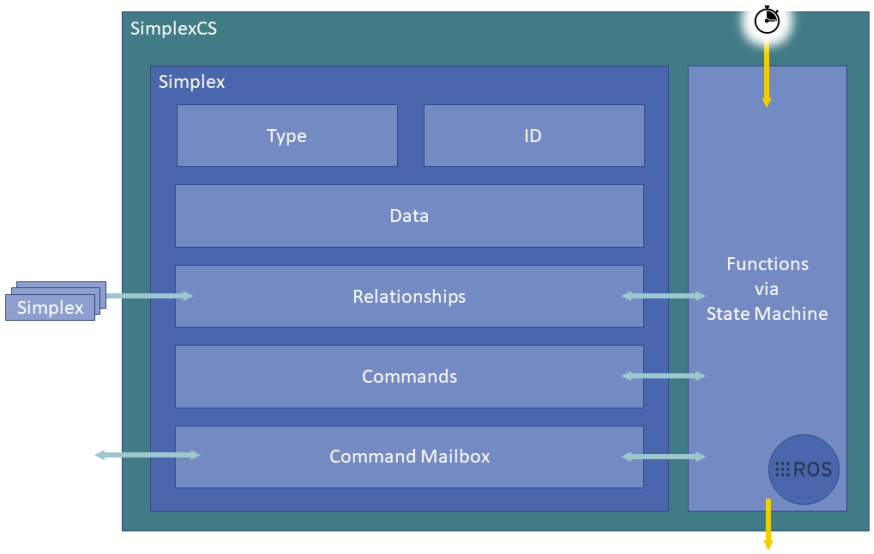

Fig. 10 SimplexCS.

The final three parts of the simplex task ($task_m$) are executed as follows: first, a *loopStart* function which is executed every loop cycle regardless of state; second a state-specific function based on the current state of the state machine (e.g. *stateStandby* or *stateActive*); finally, a *loopEnd* function which is executed every loop cycle regardless of state. The various states of the *simplexCS* state machine and its transitions are shown in Fig.11.

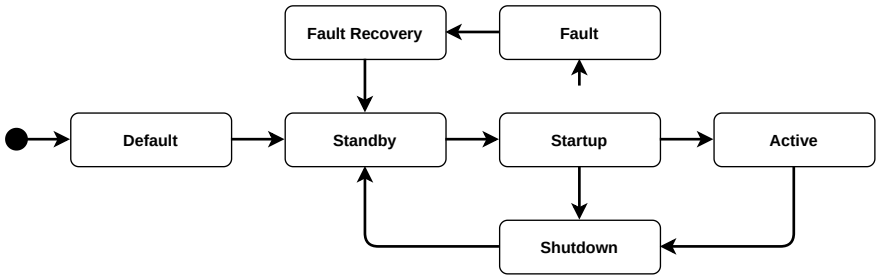

Fig. 11 simplexCS State Machine

The fundamental properties of a control system, including *Loop Cycle, Task, Job, Tick signal*, are defined in Section 3. The control system *loop cycle* is intended to be run periodically with minimised jitter, and features are available to halt or send warnings if this is breached. A CorteX environment may contain multiple CorteX agents (pc running CorteX), and each CorteX agent should at least have one cluster.

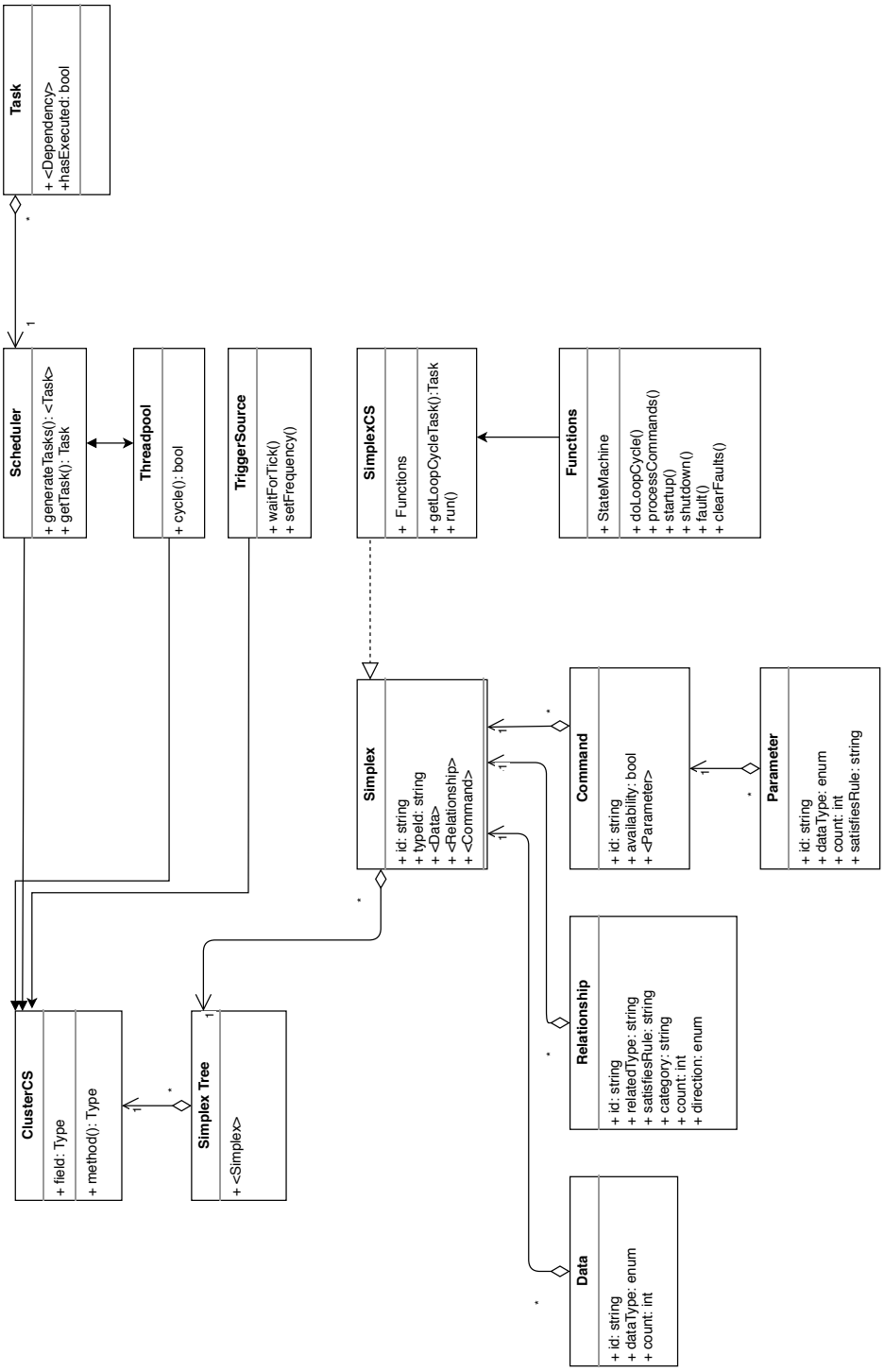

Fig. 12 CorteX CS UML diagram.

Each *ClusterCS* contains *simplexCS*s. Fig. 12 shows the attributes of a *ClusterCS* and their interactions in the UML format.

The *Loop Cycle Scheduler* constructs the execution sequence of simplexCS tasks. As part of a running CorteX agent, it is also responsible for generating an internal *tick signal*. The *Thread Pool* is used to optimise the execution of each simplexCS loop cycle, by using parallel threads. Threads, configured by the user, can be used to pull tasks from the loop cycle scheduler, while still obeying any dependency driven order (often in the form of Directed Acyclic Graph (DAG)), and execute them in parallel where possible. This ensures tasks are executed quickly and efficiently, whilst still maintaining the dependencies as a result of data flow requirements.

To provide a big picture, our implementation of a CorteX-compatible control system framework (which is one example way of creating a CorteX CS) is illustrated using a sequence diagram in Fig. 13, showing a sequence diagram implemented on CorteX CS.

When a CorteX environment is created at runtime, the executable (main.cpp), constructs an instance of a *ClusterCS*, a *Simplex Tree*, a number of *SimplexCS*s, a *Trigger Source*, and a *Communication Interface* (in this case, DDS). The executable then calls the populate() function on the simplexCS instances (which constructs the various data, relationships, and command items. i.e. a simplex) before adding them to the simplex Tree instance via the addSimplex() function.

The next step is to add the collection of *simplexCS*s to the *ClusterCS* using the addSimplexTree() function. The executable sets the *Trigger Source* by calling the setTriggerSource() function upon itself, passing the TriggerSource as a parameter. With the system model constructed, the main.cpp calls the start() function of the *ClusterCS*.

The *ClusterCS* then executes its constructLoopCycle() function, which creates the *Scheduler* and *Threadpool*. With these constructed, the *Scheduler* must be populated with each *simplexCS*'s task so that they can be used during the loop cycle. This is done via a call of the *Scheduler*'s generateTasks() function, which iterates through each *simplexCS* and retrieves its task via the getLoopCycleTask() function. Finally, with the contents of the *simplexCSs* established the System Model and Type Model can be communicated via a write() call on the *Communication Interface(s)*.

The *ClusterCS* is now ready to begin the loop cycle. At this point, the *Cluster* creates a thread on which to run the loop cycle, due to it being a blocking function. The first call is to the waitForTick() function of the *TriggerSource*, which blocks until a trigger is received. In the case of the default internal clock, this is at a regular interval set by a configured frequency. Upon receiving a trigger, the function continues to the following steps:

1. Read any updates from the Communications Interface(s) via the read() function. This will update any information required from simplexes running on another agent.
2. Call cycle() on the *threadpool*. Execution of the cycle() function is an initiation of several actions carried out by the *threadpool*. The first is to get the list of tasks from the *Scheduler*, which returns each simplexCS task and the order they must

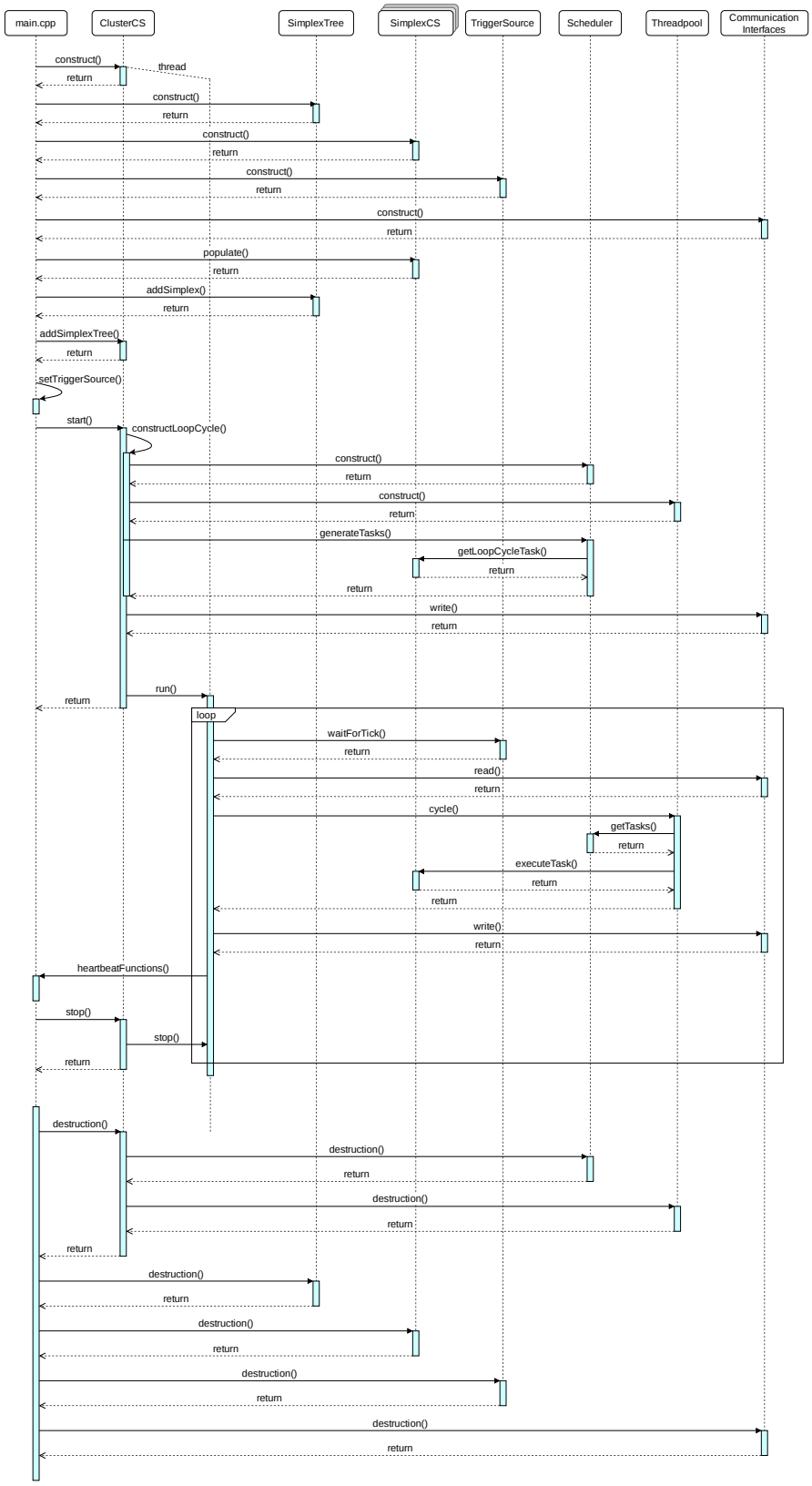

Fig. 13 CorteXCS Run Sequence.

be executed, as derived by input and output relationships. The *threadpool* then calls executeTask() on simplexCS instances so that each simplexCS executes its task ($Task_m$).
3. Write any updates to the Communications Interface(s) via the write() function. This will update any subscribed agents.

ClusterCS calls heartbeatFunctions() on the main executable to signal activity at the end of each loop cycle. When a user terminates the main executable, main.cpp calls stop() function on ClusterCS and breaks the loop cycle loop. The destruction() function on ClusterCS initialises the destruction phase and ClusterCS calls destruction() on Scheduler and Threadpool. The main executable then calls destruction() on *Simplex Tree, simplexCS, Trigger Source and Communications Interface(s)*.

4.6 CorteX Explorer

The CorteX Explorer is a feature of CorteX that collects all the graphical user interfaces (GUI) and human-machine interfaces (HMI) under the same umbrella. It is an essential module of a control system that is designed to be used in the nuclear industry, where the majority of people in operations teams are likely to consist of non-technical operations engineers. CorteX GUIs and HMIs are fundamental to guarantee high operational success. Therefore, there is a great need for highly reliable, responsive and efficient visualisation and reporting features.

To provide the required high level of reliability, responsiveness, and efficiency, an event-driven methodology is used for the Explorer, where the front-end interaction with the back-end (i.e. pulling for updates) is on a time-based loop. The loop frequency is parametrised and can be changed depending on the user requirement. Custom designed GUIs have been implemented using freely available QT libraries. Cortex visualises system parameters using a tabular view to display the content of each simplex in a CorteX environment to help the operator in monitoring the system activities. In addition to the monitoring features, The CorteX Explorer also renders the content of the (selected) simplexes and provides control screens. Some of CorteX HMIs are provided below as an example. Performance evaluation such as reliability, responsiveness, and efficiency of the CorteX Explorer is out of the scope of this chapter, therefore there are no results presented in this chapter on these. Through this approach, it is possible to automate adherence to industrial and accessibility standards and guidelines.

CorteX's self-describing protocol allows any connecting agent to discover and read the system model and its contents without prior knowledge. This functionality also extends to the CorteX Explorer GUI allowing an operator, user or a developer to explore the contents of any simplex in the system. To provide an intuitive interface to do this we have developed the *Base View*.

Fig.14 illustrates the Base View. It displays the list of simplexes in a tree down the left side of the screen in much the same way as a file browser. Upon selecting a simplex in the tree, the right side of the screen is populated with the simplex's

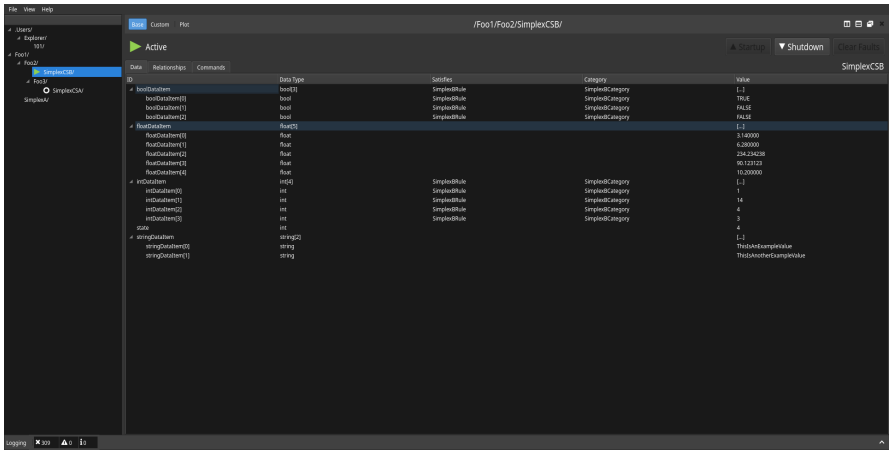

Fig. 14 CorteX Explorer: Base View

contents. Data, relationships, and command definitions are shown in a tab view along with each item's value, path, and availability respectively. Arrays are shown in collapsible rows (as shown) to avoid over-populating the screen. In the case of a simplexCS being selected, the simplex's state is shown in the top left, and command buttons to *Startup*, *Shutdown*, *Clear Faults* are provided.

As any value within a CorteX system can be read and updated over time, it is also possible to plot any numerical value on a graph. Within the Base View, there is the option to select any number of numerical data items within CorteX and place them onto a graph to be plotted over time. This can be very useful when debugging or watching for certain signals within a system.

The CorteX Explorer GUI also allows users to develop custom views for certain simplex types, or create views for entire systems comprised of both static and dynamic elements, examples of which are given in Fig. 15.

Fig. 15 shows a view used to observe and control a dual-axis EtherCAT DS402 (motor drive) device. The view is split into two halves (top and bottom) to display both axes: Axis A and Axis B, respectively. Although only the *DS402 Processor* simplex is selected in the tree on the left, the view not only uses data from that particular simplex, but also uses its relationships to pull data from both the input *DS402 Observation* simplex and the output *DS402 Modification* simplex. The left side of each axis view shows the current state of the axis from the *DS402 Observation*, the centre drive control area shows state values and control buttons for the *DS402 Processor*, and the right side shows the demand state of the drive from the *DS402 Modification*. This shows how using the standard processor morphology (see 5) we can design views around these structures and pull data from multiple simplexes to give clear contextualised information.

Similar to Fig. 15, Fig. 16 shows a view that encompasses data from multiple simplexes. This is the view for the *Jog Controller* simplex, but again pulls data from a related input *Axis Concept* and output *Axis Concept*. The current axis values

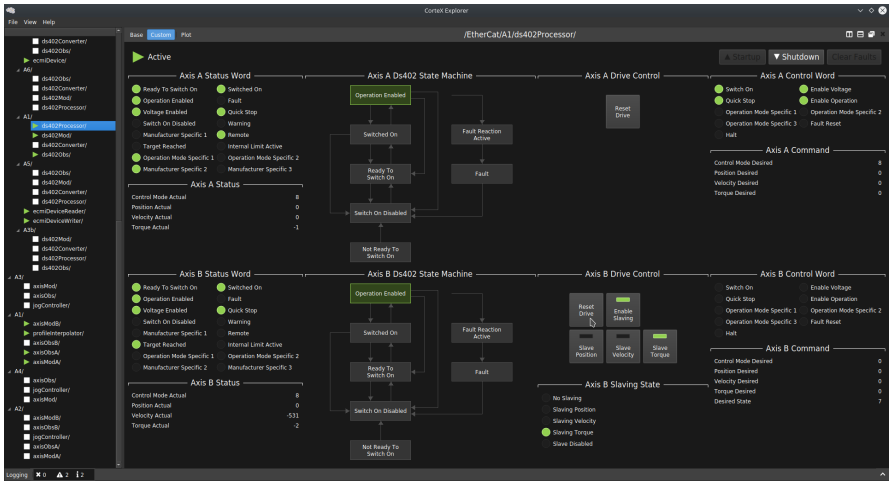

Fig. 15 CorteX Explorer: Dual DS402 Processor View.

Fig. 16 CorteX Explorer: Jog Controller View

from the input *Axis Concept* are shown on the graph and in the displays on the top row. The operator can enter demand values and change the operating mode of the *Jog Controller* using the bottom row controls. Demand values from the output *Axis Concept* are also displayed on the graph as targets.

Fig. 17 demonstrates several capabilities of the CorteX GUI framework. First, the view is split into two halves. The left side shows a representation of the physical TARM manipulator, posed to show not only the current position (in white) but also the target position (in green). This pulls data from a number of *Axis Concepts*, both observations and modifications, to pose the TARM image. This side of the view is

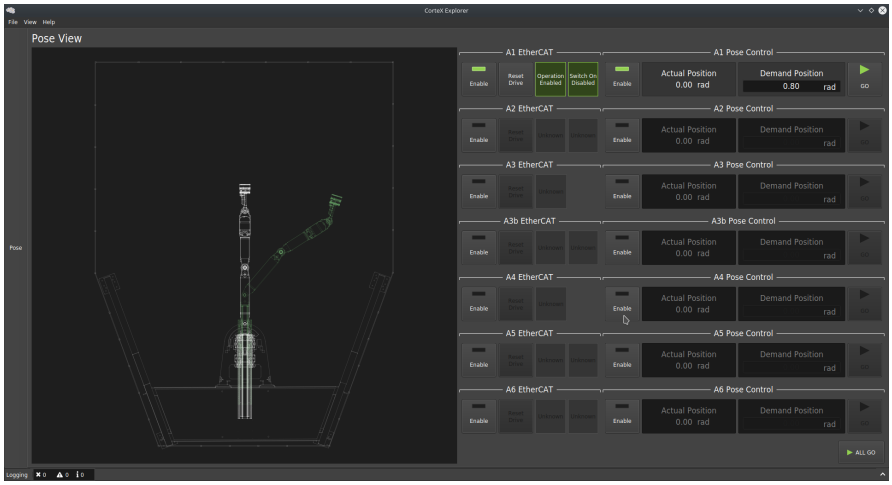

Fig. 17 CorteX Explorer: TARM Pose View

specific to the TARM manipulator as the robot joints are preloaded. However, the right side of the view is generic and can be used for any multi-axis manipulator. This selection of controls is generated dynamically by searching through the simplex model for any *Axis Controllers*, and creating a set of controls for each type of controller - in this case, they are all *Axis Pose Controllers*. You will also notice that to the left of each Pose Control area is an EtherCAT control area. The controls in this area were also auto-generated, using the morphology to discover the device *Processor* related to each *Axis Controller*, and then generating a view for the specific device processor type - in this case, an *EtherCAT DS402 Processor*. Notice how the EtherCAT control area discovers the number of axes each processor is controlling, producing two status displays for axes A1, A2, A3B, A5 and A6, but only one for A3 and A4.

CorteX Explorer is the manifestation of CorteX's discoverability (via the use of the morphology, the type model), and the self-describing nature (through the use of standardised simplex interface) that provides auto-generating, dynamic, contextual, and universal GUIs for exploring any and all CorteX systems.

5 Performance Evaluation

In the previous section, the *CorteX* framework is described in detail. In this section, we discuss performance evaluation and the configuration parameters used to evaluate *CorteX*. Section starts with a typical CorteX system definition in Section 5.1, followed-up with performance analysis of *CorteX* in terms of memory allocation and real-time characteristics, in Section 5.2 and Section 5.3, respectively.

5.1 A Typical CorteX System

There are 50-70 simplexes in an average CorteX control system. Referring back to Fig. 7, 44 simplexes (each rotary axis has 7 joints therefore is equal to 7 simplexes) are shown in the figure to illustrate the ontology and morphology concept. The communication (e.g. Fieldbus protocol) and diagnostic functionality also uses an additional 20 simplexes. Appropriately, between a third and a half of the entire system consists of Concepts. Referring back to Fig. 7 again, 30 out of 44 simplexes are Concept types. This means a minimum of a third of the simplexes in the CorteX system do not send commands, but do contain numerous pieces of data and some relationships. An average Concept type simplex contains 5-10 pieces of data; some Cartesian Concepts used for the inverse kinematics or arms, contain up to 20 data item each. Therefore, the performance analysis presented in this chapter does not represent a typical control system; the examinations on CorteX are performed for stress-testing purposes.

5.2 Memory Footprint

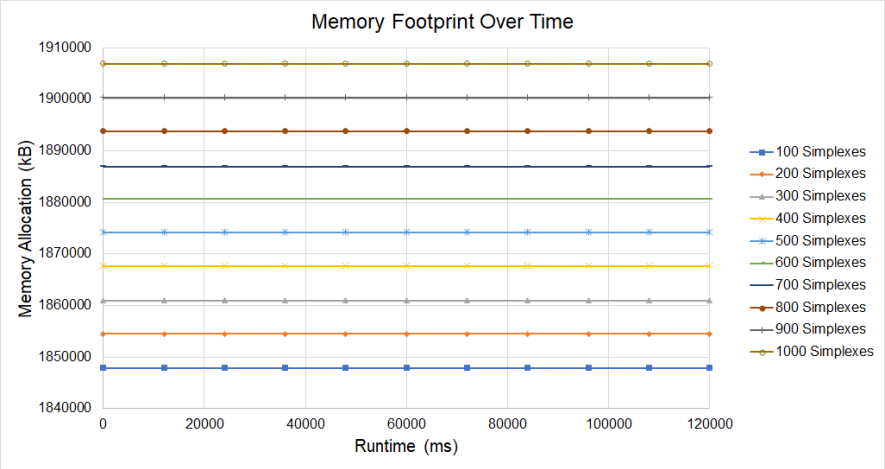

Fig. 18 Memory footprint over time for increased number of simplexes.

Dynamic memory allocation is permitted within CorteX simplex functions but should be avoided if real-time performance is required (see section 5.3), due to how long it can take. Within the internal CorteX Core and CS infrastructure, dynamic memory allocation is avoided wherever possible to produce a fixed memory footprint. Fig. 18 illustrates the allocated memory for two minutes for the increased number of simplexes. For this test, we start analysing allocated memory for 100 empty simplexes

where there is no data, relationships or commands defined. We increase the number of simplexes by 100 each time and log the memory change over the total run time. Fig. 18 provides evidence for zero memory allocation by the CorteX framework over a run time of two minutes. The same experimental data is also used to show the linear increase of the allocated memory for an increased number of empty simplexes in Fig. 19. These two figures hint at the scalability of the platform, without providing further information.

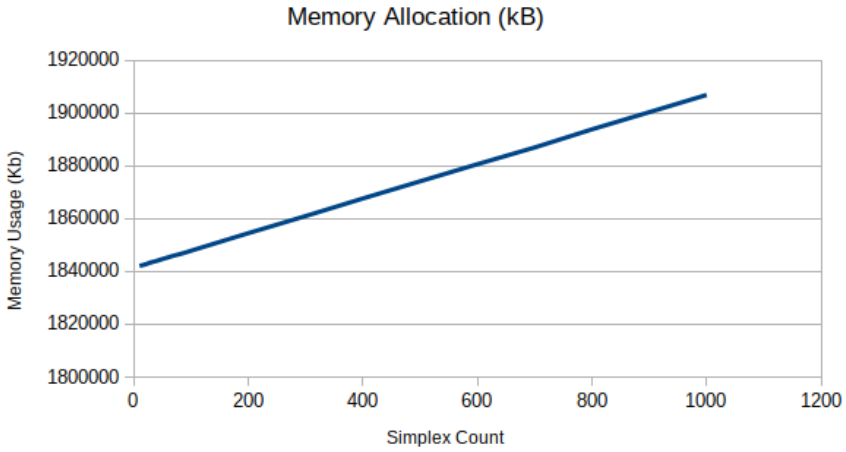

Fig. 19 *CorteX* memory profile over the increased number of simplexes.

Fig. 20 illustrates memory allocation for an increased number of data items over a fixed number of simplex. We gradually increased the number of data items by 10 per Simplex over the 25 Simplex and record the memory allocation. These 25 Simplex do not have any relationship or command values; they only contain data items. As such, Fig. 20 shows the linear increase in allocated memory as the data items increase for a fixed 25 Simplex. Again, hinting at the ability of the platform to scale linearly.

Fig. 21 shows memory allocation for an increased number of relationship items over a fixed number of simplexes. We increase the number of relationship items by 10 per simplex over a total of 25 simplexes and record the memory allocation. These 25 simplexes do not have any data or command values; they only contain relationship items. Fig. 22 presents the allocated memory for the increased number of commands over a fixed number of simplexes. Increasing the number of defined commands by 10 per simplex over 25 simplexes and recording the memory allocation. In this set, 25 simplexes do not contain any data or relationship items; they only contain command items. As such, Fig. 21 and Fig. 22 show a linear increase in the allocated memory for relationship and command items, hinting at the system's ability to scale linearly.

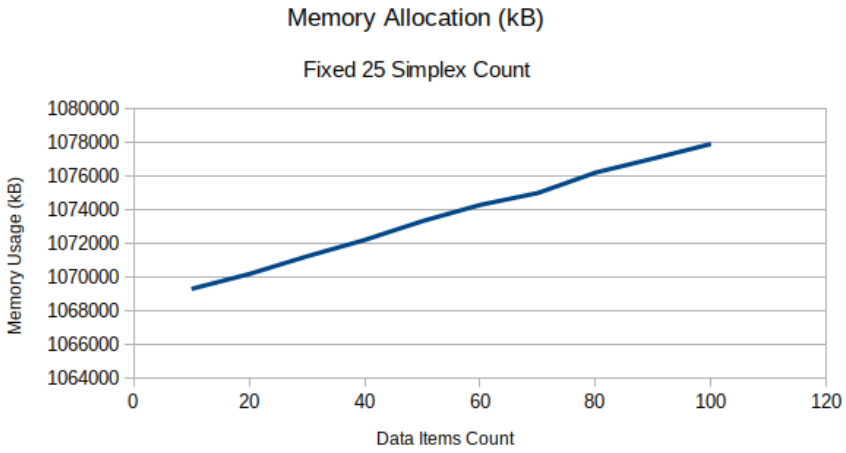

Fig. 20 *CorteX* memory profile over the increased number of data.

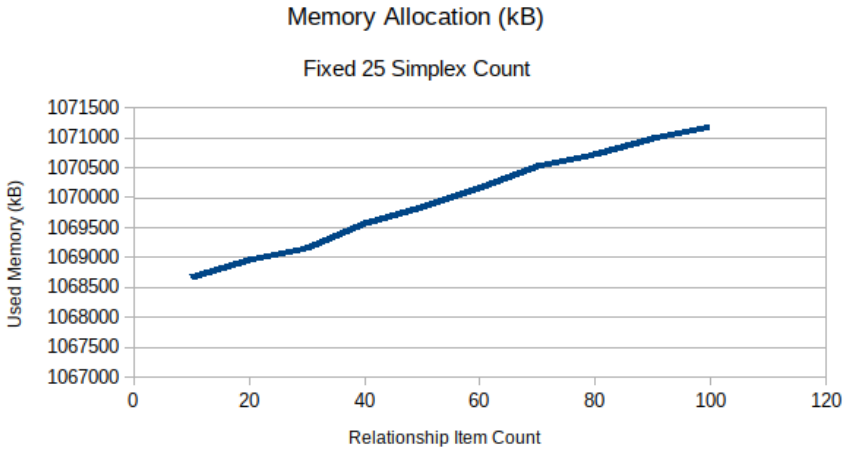

Fig. 21 *CorteX* memory profile over the increased number of relationships.

5.3 Real-time Characterisation

The importance of timeliness, Quality of Service (QoS), and high-fidelity within control systems for the nuclear industry was previously explained in Section 2. The examinations of CorteX in this section use a loop cycle frequency of 1kHz and a sample size of 5000 loop cycles. The simplexes also use a parallel dependency structure, meaning all simplex tasks are capable of executing concurrently. However running all simplex tasks on their thread would require heavy context switching,

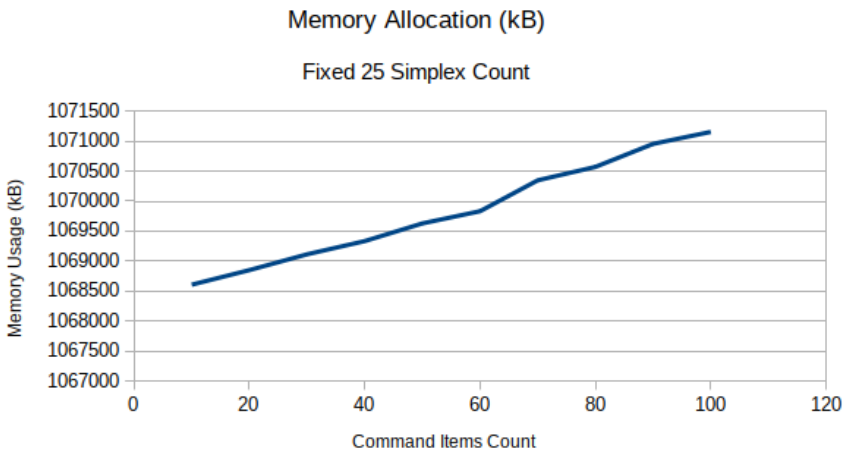

Fig. 22 *CorteX* memory profile over the increased number of commands send.

which is not an efficient method for parallel systems. For this reason, the tasks are distributed by the *Scheduler* among 4 reserved threads owned by the *Threadpool*, each running on one core of the CPU of the CI machine, see Section **??**.

Fig. 23 presents how system utilisation is effected as the number of simplexes is increased. As we increase the system workload by inserting more simplexes, the utilisation percentage increases. As each simplex task has an equal duration for completion (in this case), the percentage of time between ticks spent performing simplex tasks increases linearly with the number of simplexes in the system. In this particular test, the CorteX control system achieves almost 100% utilisation with 105 simplexes in the system.

For the next test, we create dependency pairs. Each pair has a sender and a recipient simplex; commands are sent from the sender and received by the recipient. The read() and write() functions are called on both simplexes, which are known to be more time consuming than processing empty tasks and no commands. Fig. 24 illustrates the interval between task executions on the same experimental data. As it can be seen in the figure, Cortex runs with 1 kHz when the simplex count is less than 105. When there are 110 or more simplexes in the system, tasks executions can no longer be complete in one loop cycle period. This causes tasks to overflow into the next loop cycle. Given that each loop cycle has only one command to execute per loop cycle, the interval between the task executions should be close to loop cycle duration when jitter is low in a system. Fig. 24 shows that a CorteX system can run with 1 kHz and execute commands of 105 simplexes within 1 ms interval, whereas as we introduce more simplexes into the system, the loop cycle frequency drops down to 500 Hz and therefore the interval between task executions jumps to 2 ms.

Deviation on the duration between task execution hints at a level of latency and jitter in a system. Fig. 25 and Fig. 26 illustrate the difference between a timely and an

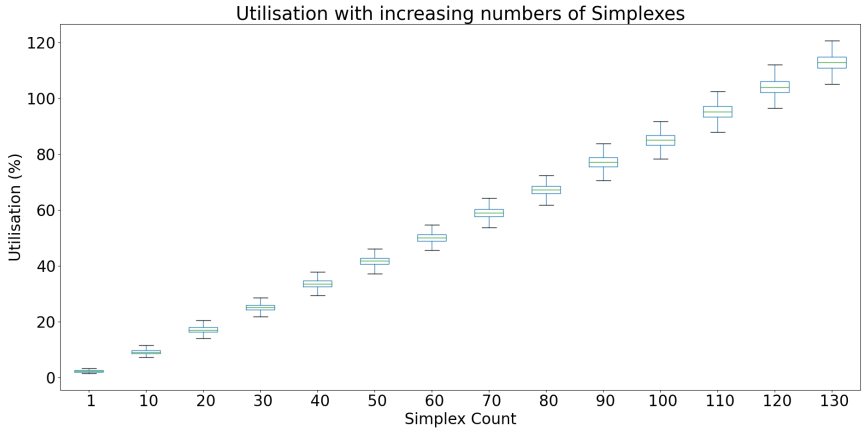

Fig. 23 Duty cycle over the increased number of simplexes each sending 1 command per loop cycle.

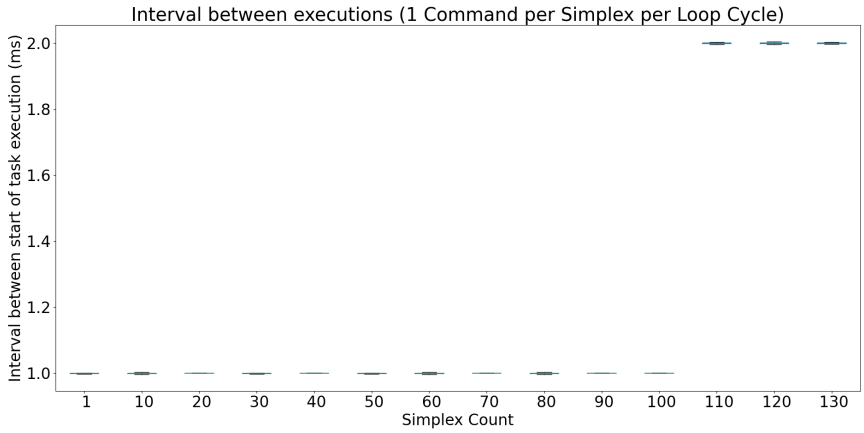

Fig. 24 Duration between task executions over the increased number of simplexes each sending 1 command per loop cycle.

overflown CorteX system. A CorteX system, running at 1 kHz with 105 simplexes without overflowing has a minimal deviation in the interval between the start of task executions. This provides suggestions on the determinism of a system. As shown in Fig. 25 a deviation of a maximum of 3 microseconds illustrates how timely is the CorteX system. The noise introduced into the CorteX system is minimal.

Fig. 27 and Fig. 28 analyse the latency in CorteX. For this set of experiments, we introduce 10 simplexes in a CorteX system and increase the number of commands each simplex sends. A fixed number of simplexes, with increased workloads (commands and tasks) increases the latency in the system. The deviation of latency

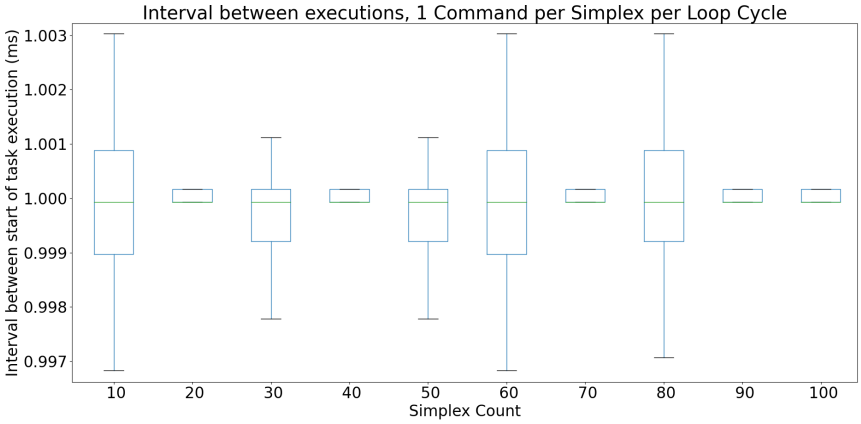

Fig. 25 Deviation of duration between task executions over the increased number of simplexes upto 100, each sending 1 command per loop cycle.

Fig. 26 Deviation of duration between task executions over the increased number of simplexes between 110-130, each sending 1 command per loop cycle.

increases as the workload of the system increases and at the maximum the deviation gets to 14 microseconds without on overflown system running at 1 kHz. The system starts to overflow when there are more than 10 commands per simplex per loop cycle. When the system overflows, it outputs in 500 Hz, latency increases to 2ms and above. The deviation is reduced by 2 ms when there are 13 or more commands per loop cycle.

The experimental results presented in Fig. 29 were performed on a fixed number of simplexes each sending one command per loop cycle. The analysis presented shows the effects of changing loop-cycle frequency over latency in Fig. 29.

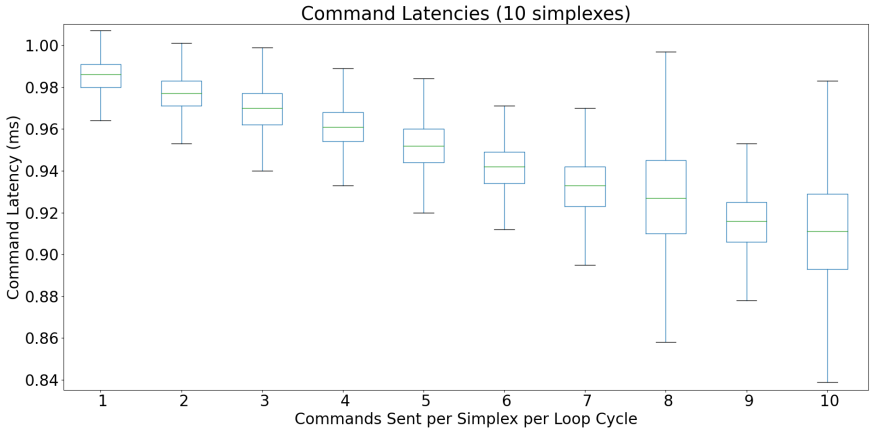

Fig. 27 Latency over the increased number of commands per loop cycle.

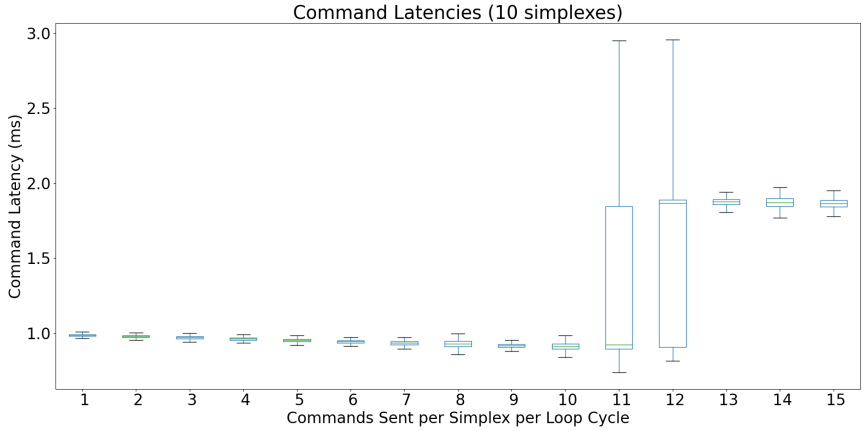

Fig. 28 An overflown analysis of latency over the increased number of commands per loop cycle.

To report jitter of a CorteX system, we use 50 empty simplexes. Fig. 30 shows the jitter in the CorteX system when there is no processing or communication happening, which means there is no injected noise due to execution or communication. This experiment is performed to provide a baseline for the increase in loop-cycle frequency and to improve our perspective on any other noise that the system could have contained. According to the Fig. 30, deviation in jitter vary between -0.02 to 0.01 for loop-cycle frequency less than 400 Hz. Jitter increases and fluctuates more as the loop-cycle frequency increases.

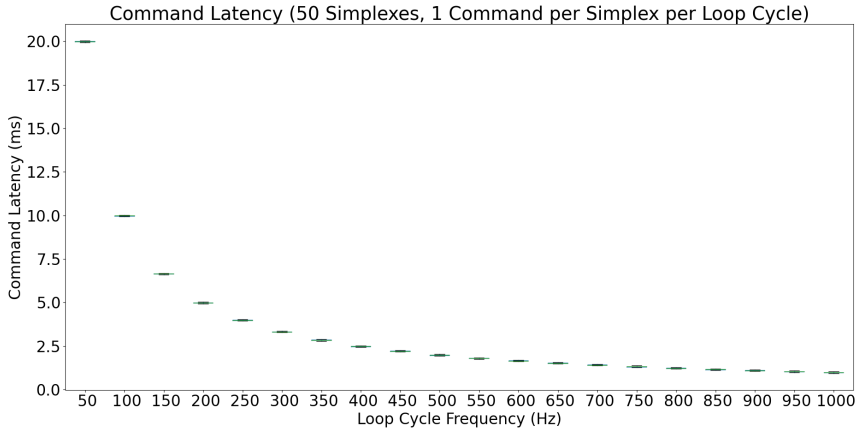

Fig. 29 Latency over the increased loop cycle frequency.

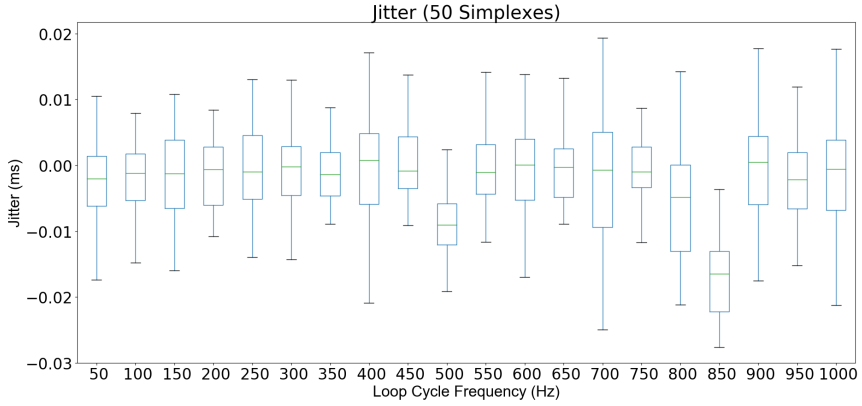

Fig. 30 Jitter over the increased loop-cycle frequency.

6 Software Infrastructure Quality and Maintainability of the High Performing System

Quality of the software is paramount in achieving the high performance required for control systems in long-lived nuclear facilities. In this section we describe maintainability of the *CorteX* codebase over the applied life-cycle management procedures. This section is followed-up with the unit testing that is used to assure the correctness of the required functionalities in section 6.2. In section 6.3 we explain the additional optimisation applied on the CorteX codebase to increase the efficiency of the software.

6.1 Life-cycle Management

Component-based software engineering and life-cycle management techniques are practised during *CorteX* development. Modern, maintainable software infrastructures help us in achieving the required high performance (high fidelity, timeliness, real-time characteristics) over a long period of time. Therefore, the importance of component-based software engineering for long-lived nuclear facilities has been discussed in Section 2. Some of the main components of *CorteX* namely, CorteX Core, CorteX CS and CorteX Explorer are described in the section 4 in detail. CorteX Profiler is the built-in evaluation module within the *CorteX* framework that analyses the system performance as a whole. The results presented in this chapter in section 5 are obtained using CorteX Profiler.

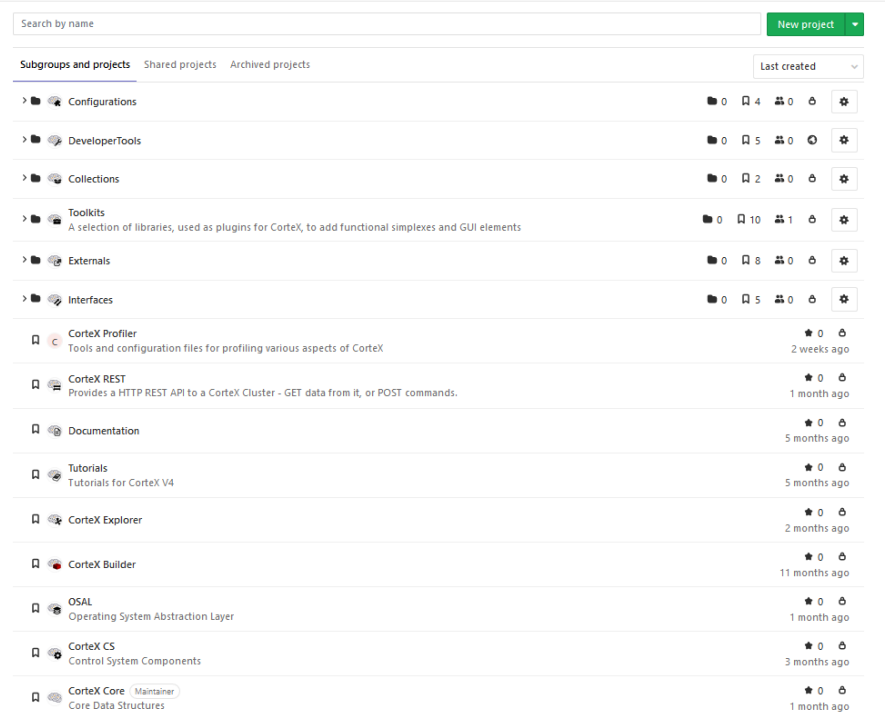

Fig. 31 Life cycle and process management.

Continuous Integration (CI), Continuous Delivery, and Continuous Deployment are commonly used processes within modern software life-cycle management, especially for large-scale engineering solutions. *CorteX* is compliant with sustainable development through continuous delivery routines and continuous deployment procedures. As part of the commit process, *CorteX* CI runners perform several auto-

mated tests on the modified codebase to check that alterations have not introduced any undesired change in functionality or performance.

6.2 Unit Testing

Each *CorteX* repository is unit tested individually, so any errors can be more easily traced to a particular component. These tests are automated for the CI runner and the entirety of the code library is tested module-by-module. These are combined with integration tests that then assemble various modules together and test their overall functionality and integration. As shown in Fig. 32 dev-unstable and dev pipelines reach over 85% tested code coverage for both CorteX Core and CorteX CS modules. The hardware specifications of the CI runner are a 4 core (2 real, 2 virtual) Intel(R) Core(TM) i3-4170 CPU @ 3.70GHz processor and 4GB memory running Debian 9.5 (stretch) OS with the 4.16.8 kernel, RT patched. Soon, we intend to perform the test routines on RTOS like operating systems, which have shown to have greater RT performance [28].

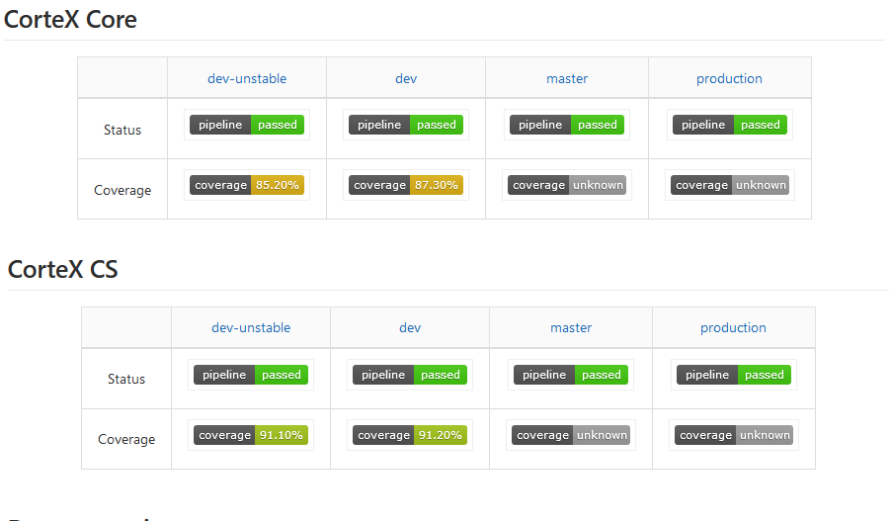

Fig. 32 Percentage of unit tested code coverage over the CI pipelines.

6.3 Optimising the CorteX Codebase

The Valgrind tool suite [38] provides a number of debugging and profiling tools that assist with bug fixing and optimising software. Callgrind is a profiling tool

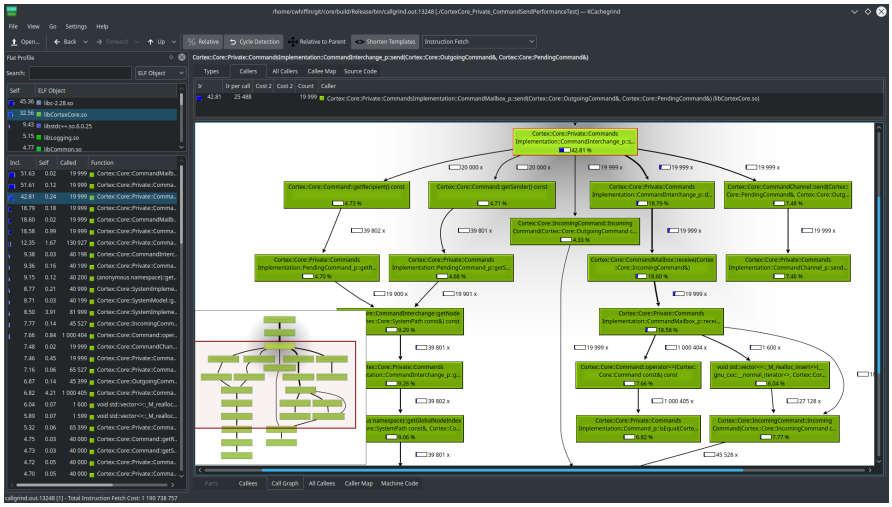

Fig. 33 Valgrind's Callgrind profiling *CorteX*.

within the Valgrind suite that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source code, the caller/callee relationship between functions, and the number of each call. Optionally, cache simulation and/or branch prediction can produce further information about the runtime behaviour of the codebase. Fig. 33 shows a call-graph generated by Callgrind applied to Simplex Commands and the Command Mailbox, to optimise implementation of command exchange in *CorteX*. Analysing the codebase function-by-function, we detect the bottlenecks that can limit the performance over case studies and make the necessary changes to increase the code efficiency. This reflects positively on the performance of CorteX systems and help us understand the interaction among the newly written code versus the excising code from a traceability point of view and inform us on the improvement areas of the entire codebase.

7 Conclusion

This chapter has proposed a novel software framework for interoperable, plug-and-play, distributed robotic systems of systems. We developed CorteX to tackle the implementation of control systems for robotic devices in complex, long-lived nuclear facilities.

CorteX attempts to solve main problems associated with interoperability and extensibility using a self-describing data representation. Standardised but extensible data interfaces are developed to provide interoperability, whilst *semantic* meaning is self-described by the components through typing, where the types are associated with

a software ontology for robotic and control system components. To aid with structural interpretation in data exchange between these interfaces, software morphologies are implemented and used to provide *syntactic* meaning. The robotic and control system knowledge structure is distributed across the CorteX agents before run-time.

Encapsulation combined with low coupling between components and high cohesion of fine-grained components, along with the use of standardised interfaces helps in achieving modularity and testability. Quality and maintainability required from a software platform are achieved by modern life-cycle management processes and effective component-based development techniques. Unit tested components and a high level of code coverage of CorteX is illustrated in Section 6 as part of the software quality control.

Extensive analysis of CorteX memory profiling has been provided in Section 5.2. The results illustrated that CorteX is a lightweight framework. In addition to this, CorteX runs with a constant memory footprint, which translates as no memory leakage. Scalability, which is crucial to achieving extensibility, is briefly illustrated as part of the memory tests with 1000 simplexes.

Timeliness and fidelity are important features of nuclear applications. Real-time characterisation of CorteX is shown in Section 5.3. Although CorteX is not a deterministic system, the deviation in latency, jitter and loop cycle duration is less than 40 microseconds while the CorteX loop cycle is running at 1 kHz. Stress testing on the increased number of command send per simplexes and increased number of simplexes each executing one command per loop cycle identify that error and noise in the system is minimised. Based on the real-time characterisation and the applied software quality management, we believe CorteX promises to deliver the needed control system solutions for the long-lived nuclear facilities.

Future work will focus on the evaluation of CorteX on large-scale, distributed clusters to inspect scalability and fidelity even further. Analysis of interoperability will be part of these tests. In addition to these points, further analysis is required in the fields of metric-based software engineering to characterise the low coupling, high cohesion and fine-granular nature of CorteX. We believe this will be a key to achieve long-term maintainability in software. Furthermore, performance improvements on CorteX that will lead to increase determinism will be carried. We envision further investigation on appropriation on task executions, and methods to improve different approaches on overflow compensation to take place.

Acknowledgement

CorteX is intellectual property of the UKAEA. This work is partly supported by the UK Engineering & Physical Sciences Research Council (EPSRC) Grant No. EP/R026084/1.

References

1. IEEE Standards Association et al. ISO/IEC/IEEE 24765: 2010 systems and software engineering-vocabulary. *Institute of Electrical and Electronics Engineers, Inc*, 2010.
2. Andrew Banks and Rahul Gupta. MQTT version 3.1.1. *OASIS standard*, 29:89, 2014.
3. V. Barabash, The ITER International Team, A. Peacock, S. Fabritsiev, G. Kalinin, S. Zinkle, A. Rowcliffe, J.-W. Rensman, A. A. Tavassoli, and P. Marmy. Materials challenges for ITER–Current status and future activities. *Journal of Nuclear Materials*, 367:21–32, 2007.
4. Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.
5. Esubalew T. Bekele, Uttama Lahiri, Amy R. Swanson, Julie A. Crittendon, Zachary E. Warren, and Nilanjan Sarkar. A step towards developing adaptive robot-mediated intervention architecture (ARIA) for children with autism. 21(2):289–299.
6. Paolo Bellavista, Antonio Corradi, Luca Foschini, and Alessandro Pernafini. Data distribution service (DDS): A performance comparison of opensplice and RTI implementations. In *2013 IEEE symposium on computers and communications (ISCC)*, pages 000377–000383. IEEE, 2013.
7. Dietmar Bruckner, Marius-Petru Stănică, Richard Blair, Sebastian Schriegel, Stephan Kehrer, Maik Seewald, and Thilo Sauter. An introduction to OPC UA TSN for industrial communication systems. *Proceedings of the IEEE*, 107(6):1121–1131, 2019.
8. Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, volume 3, pages 2523–2528. IEEE.
9. Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.
10. Paulo F Carvalho, Bruno Santos, Bruno Goncalves, Bernardo B Carvalho, Jorge Sousa, AP Rodrigues, António JN Batista, Miguel Correia, Álvaro Combo, Carlos MBA Correia, et al. EPICS device support module as ATCA system manager for the ITER fast plant system controller. *Fusion Engineering and Design*, 88(6-8):1117–1121, 2013.
11. Erwin Coumans. Bullet physics engine. *Open Source Software: http://bulletphysics. org*, 1(3):84, 2010.
12. Leo R Dalesio, AJ Kozubal, and MR Kraimer. EPICS architecture. Technical report, Los Alamos National Lab., NM (United States), 1991.
13. C. Darve, M. Eshraqi, M. Lindroos, D. McGinnis, S. Molloy, P. Bosland, and S. Bousson. The ESS superconducting linear accelerator. *MOP004, SRF2013, Paris*, page 168, 2013.
14. Rosen Diankov and James Kuffner. OpenRAVE: A planning architecture for autonomous robotics. 79.
15. Peter Dieckmann, David Gaba, and Marcus Rall. Deepening the theoretical foundations of patient simulation as social practice. *Simulation in Healthcare*, 2(3):183–193, 2007.
16. Antonio C. Domínguez-Brito. CoolBOT: a component-oriented programming framework for robotics.
17. Antonio Carlos Domínguez-Brito, F. J. Santana-Jorge, S. Santana-De-La-Fe, J. M. Martínez-García, Jorge Cabrera-Gámez, J. D. Hernández-Sosa, J. Isern-González, and Enrique Fernández-Perdomo. CoolBOT: An open source distributed component based programming framework for robotics. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 369–376. Springer.
18. G Federici, C Bachmann, L Barucca, W Biel, L Boccaccini, R Brown, C Bustreo, S Ciattaglia, F Cismondi, M Coleman, et al. Demo design activity in Europe: Progress and updates. *Fusion Engineering and Design*, 136:729–741, 2018.
19. Brian Gerkey, Richard T Vaughan, and Andrew Howard. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, volume 1, pages 317–323, 2003.
20. Brian P Gerkey, Richard T Vaughan, Kasper Stoy, Andrew Howard, Gaurav S Sukhatme, and Maja J Mataric. Most valuable player: A robot device server for distributed control.

In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, volume 3, pages 1226–1231. IEEE, 2001.
21. Morteza Ghasemi, Sayed Mehran Sharafi, and Ala Arman. Towards an analytical approach to measure modularity in software architecture design. *JSW*, 10(4):465–479, 2015.
22. Gui Gui and Paul D Scott. Measuring software component reusability by coupling and cohesion metrics. *JCP*, 4(9):797–805, 2009.
23. Wilhelm Hasselbring. Component-based software engineering. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*, pages 289–305. World Scientific, 2002.
24. Wilhelm Hasselbring. Software architecture: Past, present, future. In *The Essence of Software Engineering*, pages 169–184. Springer, Cham, 2018.
25. Michi Henning and Mark Spruiell. Distributed programming with ICE. *ZeroC Inc. Revision*, 3:97, 2003.
26. Robert Henßen and Miriam Schleipen. Interoperability between OPC UA and AutomationML. *Procedia Cirp*, 25:297–304, 2014.
27. http://www.createcrobotics.com. Iris.
28. Benjamin Ip. Performance analysis of VxWorks and RTLinux. *Languages of Embedded Systems Department of Computer Science*, 2001.
29. Jared Jackson. Microsoft Robotics Studio: A technical introduction. *IEEE robotics & automation magazine*, 14(4):82–87, 2007.
30. Mark J Kilgard. The OpenGL utility toolkit (GLUT) programming interface. 1996.
31. Jungho Kim, Sungwon Kang, Jongsun Ahn, and Seonah Lee. EMSA: Extensibility metric for software architecture. *International Journal of Software Engineering and Knowledge Engineering*, 28(03):371–405, 2018.
32. Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.
33. H Leung and Z Fan. In handbook of software engineering and knowledge engineering, 2002.
34. Stéphane Magnenat, Valentin Longchamp, and Francesco Mondada. ASEBA, an event-based middleware for distributed robot control. In *Workshops and tutorials CD IEEE/RSJ 2007 international conference on intelligent robots and systems*. IEEE Press.
35. Stéphane Magnenat, Philippe Rétornaz, Michael Bonani, Valentin Longchamp, and Francesco Mondada. ASEBA: A modular architecture for event-based control of complex robots. 16(2):321–329.
36. Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.
37. Derrick Morris. *Concise encyclopedia of software engineering*, volume 1. Elsevier, 2013.
38. Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. Building workload characterization tools with Valgrind. In *2006 IEEE International Symposium on Workload Characterization*, pages 2–2. IEEE, 2006.
39. Paul Michael Newman. MOOS-mission orientated operating suite.
40. Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
41. Markus Rickert and Andre Gaschler. Robotics library: An object-oriented approach to robot applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 733–740. IEEE, 2017.
42. Douglas T Ross, John B Goodenough, and CA Irvine. Software engineering: Process, principles, and goals. *Computer*, 8(5):17–27, 1975.
43. D Sanz, M Ruiz, R Castro, J Vega, M Afif, M Monroe, S Simrock, T Debelle, R Marawar, and B Glass. Advanced data acquisition system implementation for the ITER neutron diagnostic use case using EPICS and FlexRIO technology on a PXIe platform. *IEEE Transactions on Nuclear Science*, 63(2):1063–1069, 2016.

44. Mark W Scerbo and Steven Dawson. High fidelity, high performance? *Simulation in Healthcare*, 2(4):224–230, 2007.
45. Joseph M. Schlesselman, Gerardo Pardo-Castellote, and Bert Farabaugh. OMG data-distribution service (DDS): architectural update. In *IEEE MILCOM 2004. Military Communications Conference, 2004.*, volume 2, pages 961–967. IEEE.
46. Robert W Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th international conference on Software engineering*, pages 83–92. IEEE Computer Society Press, 1991.
47. Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985.
48. Russell Smith et al. Open dynamics engine. 2005.
49. Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.
50. Jet Team. Fusion energy production from a deuterium-tritium plasma in the JET tokamak. *Nuclear Fusion*, 32(2):187, 1992.
51. Sebastian Thrun. Robotic mapping: A survey. 1(1):1.
52. Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust Monte Carlo localization for mobile robots. 128(1):99–141.
53. Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The CLARAty architecture for robotic autonomy. In *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, volume 1, pages 1–121. IEEE, 2001.