TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Pekka Alho

Service-Based Fault Tolerance for Cyber-Physical Systems:
A Systems Engineering Approach

Julkaisu 1353 • Publication 1353

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1353
Tampere University of Technology. Publication 1353

Pekka Alho

Service-Based Fault Tolerance for Cyber-Physical Systems: A Systems Engineering Approach

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Konetalo Building, Auditorium K1702, at Tampere University of Technology, on the 11th of December 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3642-7 (printed)
ISBN 978-952-15-3660-1 (PDF)
ISSN 1459-2045

Abstract

Cyber-physical systems (CPSs) comprise networked computing units that monitor and control physical processes in feedback loops. CPSs have potential to change the ways people and computers interact with the physical world by enabling new ways to control and optimize systems through improved connectivity and computing capabilities. Compared to classical control theory, these systems involve greater unpredictability which may affect the stability and dynamics of the physical subsystems. Further uncertainty is introduced by the dynamic and open computing environments with rapidly changing connections and system configurations. However, due to interactions with the physical world, the dependable operation and tolerance of failures in both cyber and physical components are essential requirements for these systems.

The problem of achieving dependable operations for open and networked control systems is approached using a systems engineering process to gain an understanding of the problem domain, since fault tolerance cannot be solved only as a software problem due to the nature of CPSs, which includes close coordination among hardware, software and physical objects. The research methodology consists of developing a concept design, implementing prototypes, and empirically testing the prototypes. Even though modularity has been acknowledged as a key element of fault tolerance, the fault tolerance of highly modular service-oriented architectures (SOAs) has been sparsely researched, especially in distributed real-time systems. This thesis proposes and implements an approach based on using loosely coupled real-time SOA to implement fault tolerance for a teleoperation system.

Based on empirical experiments, modularity on a service level can be used to support fault tolerance (i.e., the isolation and recovery of faults). Fault recovery can be achieved for certain categories of faults (i.e., non-deterministic and aging-related) based on loose coupling and diverse operation modes. The proposed architecture also supports the straightforward integration of fault tolerance patterns, such as FAIL-SAFE, HEARTBEAT, ESCALATION and SERVICE MANAGER, which are used in the prototype systems to support dependability requirements. For service failures, systems rely on fail-safe behaviours, diverse modes of operation and fault escalation to backup services. Instead of using time-bounded reconfiguration, services operate in best-effort capabilities, providing resilience for the system. This enables, for example, on-the-fly service changes, smooth recoveries from service failures and adaptations to new computing environments, which are essential requirements for CPSs.

The results are combined into a systems engineering approach to dependability, which includes an analysis of the role of safety-critical requirements for control system software architecture design, architectural design, a dependability-case development approach for CPSs and domain-specific fault taxonomies, which support dependability case development and system reliability analyses. Other contributions of this work include three new patterns for fault tolerance in CPSs: DATA-CENTRIC ARCHITECTURE, LET IT CRASH and SERVICE MANAGER. These are presented together with a pattern language that shows how they relate to other patterns available for the domain.

Preface

This study was carried out at the Department of Intelligent Hydraulics and Automation (IHA) at Tampere University of Technology.

ITER is an ambitious scientific endeavour that aims to demonstrate the scientific and technological feasibility of fusion energy. As such, it faces many uncertainties and challenges, including the funding and organization of the project. On the other hand, it is an opportunity to learn and develop technologies for, for example, superconductors, radiation resistant materials and maintenance of high-energy physics facilities, providing improvements beneficial for domains outside of fusion research. In this thesis, ITER is used as a target of application and source of requirements, but it has also provided an inspiring framework and research community for my work. Therefore I am indebted to Professor Jouni Mattila who has provided this opportunity and acted as the supervisor of this thesis. Thank you for the guidance, insightful comments and trust.

Jouni's research group at IHA has been crucial in the creation of this thesis. Especially Jukka Väyrynen, Liisa Aha, Antti Hahto, Mikko Viinikainen, Janne Tuominen, Teemu Rasi, Ari Aaltonen, Reza Oftadeh, Zahra Ziaei and Janne Koivumäki have contributed through discussions, work efforts and co-operation. I would also like to thank everybody at IHA for providing a relaxed workplace.

Before I started this work at IHA, I had the opportunity to learn about automation domain and research methodologies in the Automation Software research group; thanks for this opportunity go to Professor Seppo Kuikka and members of the group, and especially Jari Rauhamäki who has been my co-author and introduced me to the world of patterns.

The comments and criticism provided by the pre-examiners Professor Tor Stålhane and Professor Valeriy Vyatkin are greatly appreciated. I'm also grateful to the GOT-RH trainee group for the interesting discussions and co-operation, all the people who have organized GOT-RH and the CCFE staff for their hospitality during my mobility to JET. Thanks also for Jim Coplien, Robert Hanmer and Pierre Bonnal for helpful comments.

This work has been financially supported by Tekes and EFDA (now EUROfusion). Being able to focus on a single research project for four years is a rare luxury in today's academia. This has given me the possibility to consider different approaches to the topic and find my own way through the research questions.

Finally I have to present my sincerest thanks to my family; my parents have always been very supportive which was important especially in the early phases of my doctoral studies. And last but not least, thanks to my beautiful wife Katriina for being so patient and providing the motivation to get through the occasional moment of self-doubt.

Helsinki, Oct. 8, 2015

Pekka Alho

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALARP	as low as reasonably practicable
AUTOSAR	AUTomotive Open System ARchitecture
C4G	Comau robotics control unit
CAT	computer assisted teleoperation
CBSE	component-based software engineering
CoAP	Constrained Application Protocol
COTS	commercial off-the-shelf
CORBA	Common Object Request Broker Architecture
CPS	cyber-physical system
DCOM	Distributed Component Object Mode
DDS	Data Distribution Service for Real-Time Systems
DTP2	Divertor Test Platform 2
DOF	degrees of freedom
EC	equipment controller
ECU	electronic control unit
EFDA	European Fusion Development Agreement
ESB	enterprise service bus
F4E	Fusion for Energy
FMECA	failure modes, effects and criticality analysis
FOSS	free and open source software
GOT-RH	Goal Oriented Training programme on Remote Handling
GSB	global service bus
GSN	Goal Structuring Notation
GUI	graphical user interface
HLCS	high-level control system, includes operator interfaces, virtual reality, etc.
HTTP	Hypertext Transfer Protocol
IDC	input device controller
IDL	OMG Interface Definition Language (or interface description languages in general)
IHA	Department of Intelligent Hydraulics and Automation
IoT	Internet of Things
IP	Internet Protocol
ITER	the Latin word for way or journey (originally International Thermonuclear Experimental Reactor, but the acronym is no longer used)
JET	Joint European Torus
JSON	JavaScript Object Notation
LLCS	low-level control system, includes equipment controllers, etc.
LSB	local service bus
M2M	machine-to-machine communications
MTBF	mean time between failures
MTTF	mean time to failure
MTTR	mean time to repair
NVP	N-version programming
OMG	Object Management Group
OMS	operations management system
OPC	Open Platform Communications (previously Object Linking and Embedding for Process Control)
OPC UA	OPC Unified Architecture
OWL-S	Semantic Markup for Web Services

P2P	peer-to-peer
PLC	programmable logic controller
RAMI	reliability, availability, maintainability and inspectability
REST	Representational State Transfer
RFID	Radio Frequency Identification
RH	remote handling
RHCS	remote handling control system
ROS	Robot Operating System
RPC	remote procedure call
RT	real-time
RTOS	real-time operating system
SIL	safety integrity level
SOA	service oriented architecture
SOAP	Simple Object Access Protocol
SOUP	software of uncertain pedigree
SPoF	single point of failure
SRD	system requirements document
TCP	Transmission Control Protocol
UAV	unmanned aerial vehicle
UI	user interface
UML	Unified Modeling Language
VR	virtual reality
WSDL	Web Services Description Language
XML	Extensible Markup Language

List of Publications

This thesis is a compendium, consisting of a summary and the following six publications, referred to as [P1]-[P6].

[P1] P. Alho and J. Mattila, Dependable Control Systems Design and Evaluation, Ninth Annual Conference on Systems Engineering Research, 2011, Redondo Beach, USA, http://urn.fi/URN:NBN:fi:tty-2011060814695

[P2] P. Alho and J. Mattila, Breaking down the requirements: Reliability in remote handling software, Fusion Engineering and Design, vol. 88, no. 9-10, pp. 1912–1915, 2012, DOI: 10.1016/j.fusengdes.2012.11.008

[P3] P. Alho and J. Mattila, Real-Time Service-Oriented Architectures: A Data-Centric Implementation for Distributed and Heterogeneous Robotic System, Proceedings of the 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 2013, IFIP Advances in Information and Communication Technology, vol. 403, pp. 262-271, 2013, DOI: 10.1007/978-3-642-38853-8_24

[P4] P. Alho and J. Mattila, Software fault detection and recovery in critical real-time systems: an approach based on loose coupling, Fusion Engineering and Design, vol. 89, no. 9-10, pp. 2272-2277, 2014, DOI: 10.1016/j.fusengdes.2014.04.050

[P5] P. Alho and J. Rauhamäki, Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems, [In press], LNCS Transactions on Pattern Languages of Programming

[P6] P. Alho and J. Mattila, Service-oriented approach to fault tolerance in cyber-physical systems, Journal of Systems & Software, vol. 105 (July), pp. 1-17, 2015, DOI: 10.1016/j.jss.2015.03.041

1 Introduction

ITER is an international nuclear fusion engineering and research project that is currently building the world's largest experimental tokamak-type fusion reactor in Cadarache, South of France. ITER is playing a significant role in the transition from plasma physics experiments to full-scale fusion power plants by demonstrating the availability and integration of technologies essential for the reliable generation of electricity [1]. One of these technologies is remote maintenance. Nuclear fusion releases high-energy ionizing radiation that causes the degradation of in-vessel components of the ITER machine, necessitating several replacements during the project's lifetime. However, due to the activation and heavy weight of the reactor components, hands-on inspections and maintenance are not possible. Remote handling (RH) systems are, therefore, used to perform maintenance tasks inside the machine, without the potential for direct human intervention.

RH systems include heavy robotic transporters and manipulators that are used to inspect and replace reactor components using man-in-the-loop teleoperation. The systems must be capable of moving heavy components, such as nine-tonne divertor cassettes, in pitch-black maintenance tunnels with clearances of only a few millimetres. Mechanical and electrical design constraints are introduced by need to maintain in-vessel cleanliness and tolerate radiation levels of approximately 100 Gy/h. The challenges of designing reliable mechanical structures and operating procedures for this environment are immediately obvious. Designing and creating the software to control this equipment is similarly challenging; however, the related risks are significantly more difficult to identify due to the abstract nature and complexity of software. Further challenges are introduced by flexibility requirements of ITER's RH systems.

The supervisory RH system must integrate seven RH systems, containing a heterogeneous mix of equipment, parts and software, into a coherent plant system that is operated from standardized work cells in the RH control room. This plant system enables central monitoring and coordination among different RH systems, while facilitating its integration into the ITER central control and data access system. The seven RH systems of the ITER machine are shown in Figure 1. These include a blanket RH system for the replacement of 440 blanket modules, a divertor RH system for the replacement of 54 divertor cassettes, a neutral beam RH system for the maintenance of three neutral beam injectors, a multi-purpose deployer for in-vessel tokamak operations, a cask and plug RH system to provide a platform for the transportation and deployment of in-vessel systems, and a hot cell RH system for the maintenance and commissioning of all the other RH systems. These systems are put out to tender for different consortiums by the national fusion agencies or the ITER Organization. While the role of the ITER Organization is to standardize interfaces and common components, the ultimate implementation of systems is carried out by a large number of independent suppliers and contractors. The environment, which combines several engineering disciplines, specialized technologies and stakeholder organizations, including all of the different vendors, is typical of cyber-physical systems (CPSs) (i.e. networked embedded feedback systems interacting with the physical environment and operating in open computing environments). The ITER RH systems will utilize a mix of specially developed and off-the-shelf technologies, including cranes, transporters, robotic manipulators and tooling, controllers, data management, structural simulators, visualizations and operational subsystems, plant-level supervisory systems, etc. To maximize a project's chance of success, the reuse of software and the capability to integrate various RH technologies is necessary. For these ends, software architecture plays a key role, defining variation points and how flexible or rigid the integration will be. The architecture should enable the independent deployment of software organized into components around system operational capabilities and avoid forcing specific technology platforms.

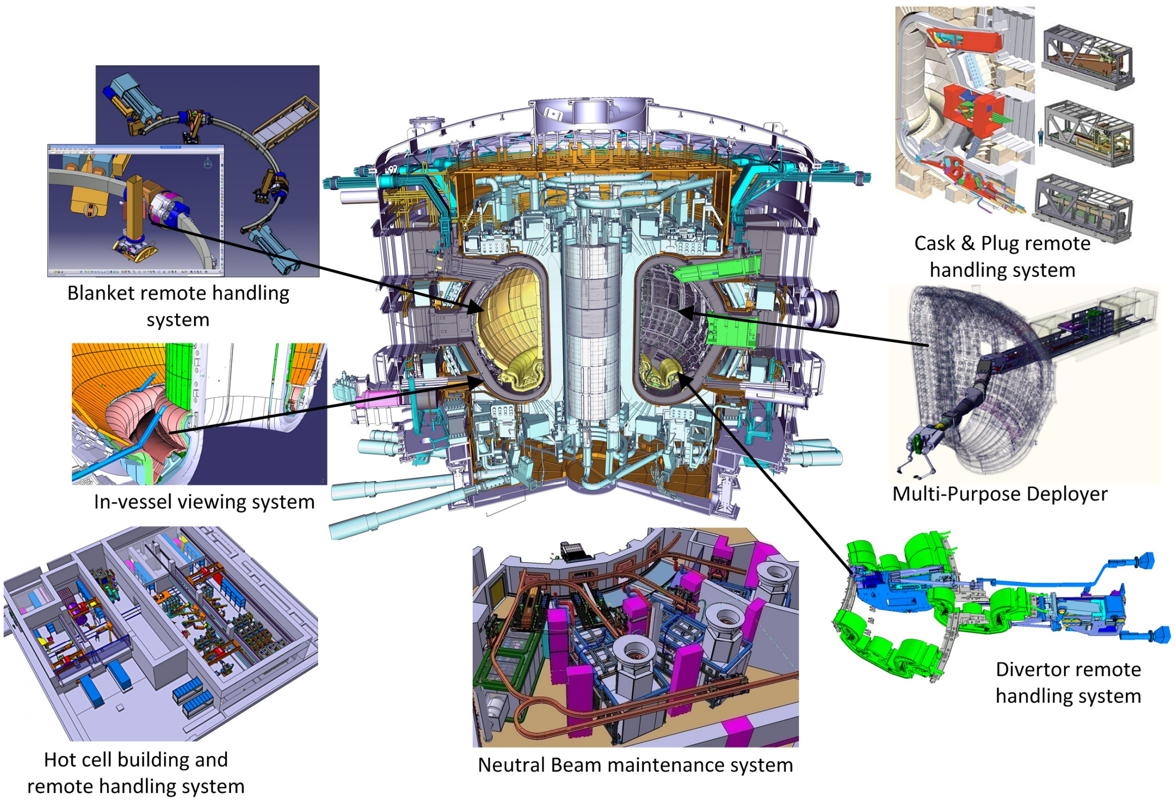

Blanket remote handling system
In-vessel viewing system
Hot cell building and remote handling system
Neutral Beam maintenance system
Cask & Plug remote handling system
Multi-Purpose Deployer
Divertor remote handling system

Figure 1. Some of the different physical systems that need to be managed and operated from the ITER remote handling control room; the ITER tokamak is shown in the middle

The heterogeneous and distributed operating environment of ITER RH systems is characteristic for CPSs, the study of which combines elements from embedded systems, networking and control engineering. CPSs have emerged from a combination of several technological trends: namely, increasingly powerful computing capabilities, declining instrumentation costs, the prevalence of Internet Protocol (IP)-based communication technologies and increasing control of physical dynamics. These trends are moving embedded control systems towards co-operative models and composable components. The utilization of open standards and ecosystems can potentially enable CPSs to provide intelligent and value-adding composite services over networks, made available through dynamic connectivity capabilities. Key issues for the integration of networked systems—both in ITER and CPSs in general—include a scalable management of complexity, an interoperability (to function in heterogeneous environments) and a capability to gracefully handle unexpected faults, since failures of components in both cyber and physical domains must be either tolerated or contained [2]. To contribute to the management of these challenges, we use the ITER Divertor RH system as a target CPS for the evaluation of new approaches to the development of dependable and flexible control systems. These include the integration of existing prototype software into service-based teleoperation systems, as well as the empirical evaluation of contributions using an electric industrial manipulator.

Failures in any of the complex, distributed, real-time systems shown in Figure 1 could cause unexpected maintenance delays, reducing the availability of the ITER device for scientific experiments. The successful integration and operation of RH systems, therefore, plays a significant role in illustrating the feasibility of remote maintenance of a large-scale fusion device. ITER has an expected minimum operational lifetime of 30 years, making changes in the operational environment or in system functionality very likely. Consequently, the long-term success of remote maintenance systems depends on the capability to standardize the architecture, while maximizing its flexibility.

The target level of reliability and the use of rigorous development methods are balanced against the risks of failures, including financial losses and setbacks on project goals. However, evaluating probabilities of failures and failure modes for software-based systems is challenging, especially in dynamic environments. The complexity of CPSs, such as the ITER RH systems shown in Figure 1, is not sufficiently matched with best practices of building dependable embedded systems (e.g., a cleanroom mentality, verifying every line of code, utilizing redundant channels, etc.) alone. Use of current best practices is a viable approach if the system can be considered to be a closed box, with limited interaction with the surrounding world; however, it is impossible to foresee and test all possible combinations of how a software program written for a CPS will be used. Therefore, a systems engineering approach is needed to combine the most effective methods to build a dependable and resilient system.

The main contribution of this compendium thesis is the description and empirical evaluation of the fault-tolerant service-based architecture, which is verified using ITER RH system requirements to represent a demanding, real-world test case. In addition to evaluating the use of a real-time service-oriented architecture (RTSOA) in a CPS, this thesis contributes to the field by demonstrating a systems engineering approach to dependability to show how results are related to the development of CPSs, analysing ITER remote handling control system (RHCS) requirements, evaluating "let it crash" approach for real-time fault detection and recovery, confirming the RTSOA style's applicability for building resilient real-time systems, and identifying three patterns for the fault tolerance of CPSs.

The remainder of this section describes the research background, motivations, objectives, scope, methodology and contributions of this thesis. The current state of the art for dependability and systems engineering is analysed in Section 2. Software architectures for building CPSs are discussed in Section 3. A general overview of ITER RH systems is presented in Section 4. Section 5 presents the systems engineering approach for achieving dependability, as well as the contributions of the papers, including the architectural design and an outline of the roles of the author in each paper. Finally, a discussion of the results and conclusions is presented in Section 6.

1.1 Background

RH plays an important role in the ITER plant because deuterium-tritium fusion produces high-energy neutrons, which are absorbed by the components inside the reactor vessel, leaving the components in an activated state. Since humans do not have access to the reactor, all maintenance tasks are performed indirectly through remotely operated robots using man-in-the-loop operations, making RH critical for the maintenance of the ITER.

RHCSs consist of heterogeneous subsystems, including equipment controllers, operator interfaces and supervisory systems that are used to carry out complex, multi-phase operating scenarios in coordination with other work cells and RH systems using several types of equipment, ranging from cameras to manipulator arms, specialized end-effectors and hydraulic movers. Some of the subsystems used in this study can be seen in Figure 2, which shows a prototype RHCS being used to test divertor remote handling operations at Divertor Test Platform 2 (DTP2)[1]. The RHCSs must be combined into a single cohesive RH plant system that can be used to perform all maintenance operations flexibly and reliably from multiple work cells located in the RH control room. This kind of diverse operation environment is a typical challenge for CPSs.

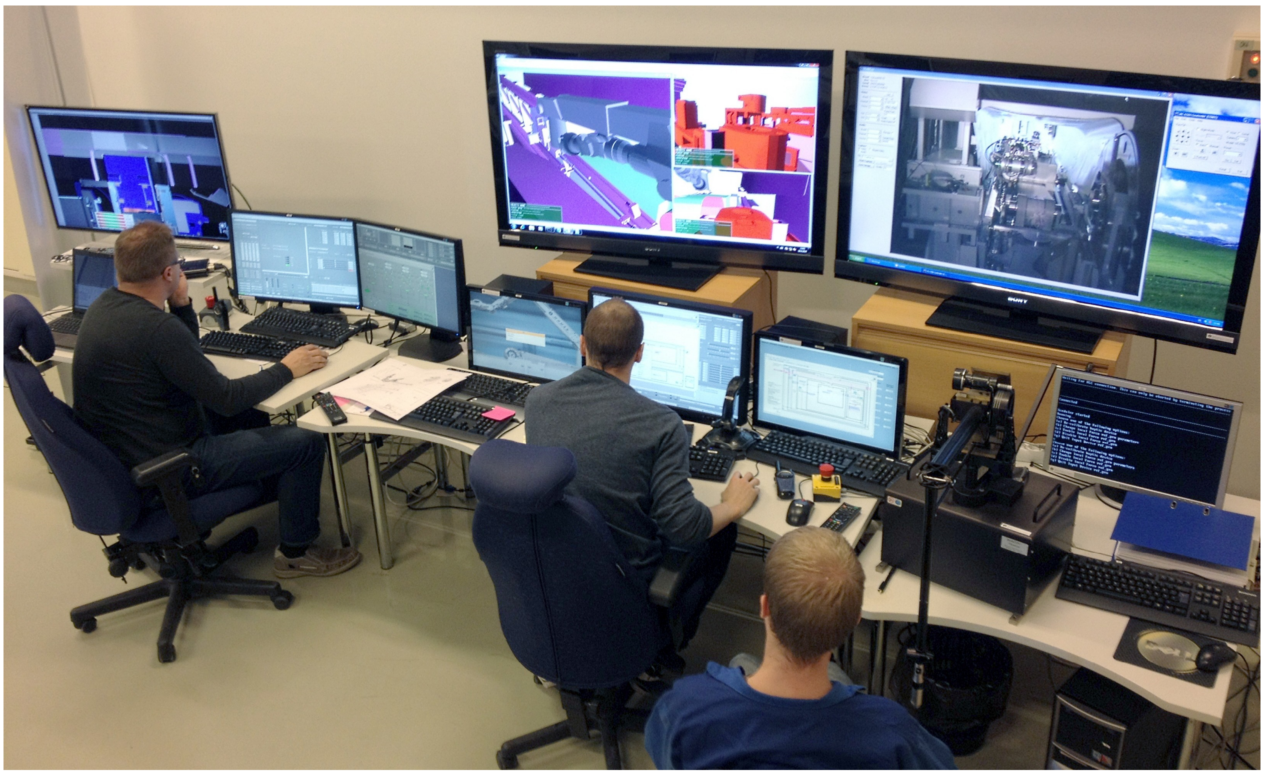


Figure 2. High level control system for ITER divertor remote handling operations at DTP2; shown: operator interfaces, input devices, virtual reality and camera feed

Due to radiation, RH systems and equipment must be protected against single points of failure (SPoF). They must also be recoverable, since the loss of equipment inside the reactor could have catastrophic consequences for the ITER research project. Due to the potential financial losses caused by failures, RH systems can be considered to be safety- or mission-critical. However, the complexity and scale of such systems makes fail-operate fault tolerance techniques, which are typically used in nuclear power plants and aviation systems, financially impractical. The basic approach and requirement, therefore, in the case of a hazardous event is a fail-safe behaviour: The system is transferred to a safe state, such as a power off, through an interlock or safety systems, if

[1] A full scale model of a bottom region of the ITER fusion reactor, located at Tampere, Finland, and used for developing equipment, methods, and software for ITER RH operations.

necessary. The problem with this approach is that a complex system, such as RHCS, is likely to have residual faults, especially since flexibility and interoperability increase factors of uncertainty. Without fault tolerance and resilience, these faults will have a negative impact on availability especially early in the operational lifecycle or when the operational profile of the system changes.

Most of the work presented in this thesis has been carried out through the Goal-Oriented Training programme on Remote Handling (GOT-RH) project, which is supported by the European Fusion Development Agreement (EFDA). Objectives for the GOT-RH include implementing a structured RH system design and development-oriented tasks in a multidisciplinary systems engineering framework. This project served as the starting point for this thesis, providing the objective of developing a fault-tolerant concept architecture that could be used to evaluate and suggest approaches for the actual ITER RHCS design.

The ITER RH systems, which are still under development, were used as sources of requirements and evaluation criteria to evaluate the developed design. Due to the conceptual nature of the design, it is not possible for the prototype to be fully compliant with ITER RHCS requirements (e.g., with regard to radiation resistance). The relevant features of the RH systems selected for this research include a heterogeneous environment, a high availability requirement for RH systems and for the plant, a need to manage complexity, a long operational phase (expected to last 20 to 30 years), a safety-oriented/mission-critical nature of operations, a high level of system distribution and real-timeliness (i.e., time-sensitive, low latency and high throughput).

1.2 Need for Dependability

This thesis uses the definition for dependability given by Avizienis et al. [2], in which dependability is defined as the ability to deliver service that can be justifiably trusted. Dependability encompasses attributes of reliability, availability, maintainability, integrity and safety, and it is closely connected to reliability, availability, maintainability and safety (RAMS) requirements in systems engineering. A fault is a defect in a system, whereas a failure is an instance in time during which a system displays behaviour that is contrary to its specifications [2]. A fault, when activated, can lead to an error (i.e., an invalid state), and the invalid state generated by an error may lead to another error or a failure. Means for achieving dependability can be categorized into four groups: fault prevention, fault tolerance, fault removal and fault forecasting [2]. While the key results in the publications of this thesis focus on fault tolerance, an inclusive systems engineering approach to dependability is necessary. Connections of dependability and the systems engineering approach are analysed in Section 2.1.

One of the main threats to dependability is complexity. Functional safety standards tend to guide the development of safety-related systems towards full-separation and closed systems in order to minimize the potential for error propagation and limit complexity. However, this kind of approach has limited applicability for the open environments of CPSs, in which the system consists of a large number of intercommunicating nodes, and there may be no single organization with sole ownership of the end-to-end system.

The typical approach for achieving a high availability level is based on the use of active or passive redundancy, with hot or cold spares [3]. For example, in telecommunication systems, the availability requirements are very high: Availability levels of six or even nine 9s are not unheard of

[4] [5]. An availability of six 9s (99.9999%) corresponds to downtime of just 31.5 seconds per year. These systems are designed to tolerate single failures with duplicated components, so that the failure of a switching unit does not cause a system failure. This demonstrates how failures can also occur as partial failures, in which the system has capability to provide a subset of services or degraded performance to users. The Mars rovers Spirit and Opportunity have achieved remarkable levels of reliability against system-level failures, lasting several times longer than their planned 90-day mission durations, despite partial failures [6].

Redundancy has served as a central tenet of reliability engineering for over 50 years; however, it is important to understand its limitations. For example, it can be argued that even the hard numbers used for quantitative proof of aeroplane reliability are based largely on numerous immeasurable factors, such as "isolation" and "independence" [7]. Redundancy also increases complexity due to the extra elements required to manage and implement the redundant system—and, if implemented improperly, may detract from the reliability of a system [7]. Therefore, new approaches that support the development of dependable CPSs are needed. The rest of this section explores the motivation for the research and outlines the research problem.

1.2.1 Software Failure Case Studies

The following four short case studies of software failures aim to highlight some of the challenges encountered in CPS-relevant applications.

F-22 Raptor Squadron Shot Down by the International Date Line

F-22 Raptor is a tactical fighter introduced into service in 2007. The F-22's avionics software has some 1.7 million lines of code. While attempting the aircraft's first foreign deployment to the Kadena Air Base in Okinawa, Japan, on 11 February 2007, six F-22s of 27th Fighter Squadron flying from Hickam Air Force Base, Hawaii, experienced multiple software-related system failures, including failures of navigations and parts of communications, while crossing the International Date Line. The fighters had no communication or navigation and could not reset their systems. Serious consequences were avoided, however, because the aircraft were able to return to Hawaii by following tanker aircraft. [8]

Germany's Super Train Arrives Two Years Late

At end of 2013, Siemens AG delivered the first four Intercity-Express 3 (ICE3) trains to Deutsche Bahn [9]. The initial goal had been to deliver the trains more than two years earlier; however, delays occurred when problems with the control system for the brakes prevented the project from receiving authorization from rail authorities. The trains had communication problems in their software, which manifested during the testing of the ICE3 double trains. Specifically, the command to stop an ICE3 double train was delayed by about one second by the computer until it was executed, adding 70 meters to the stopping distance (given a speed of 250 km/h). [10]

Packets of Death

In 2014, Start2Star Communications, LLC adopted a new Ethernet controller for a piece of on-premises equipment, but then noticed that units seemed to be failing randomly. Only a power cycle, requiring a physical visit to the equipment, would bring a failed device back. However, when a faulty unit was shipped back, it would perform correctly when tested. The company began to work with a reseller to debug the problem and eventually found out that the specific content in a Session Initiation Protocol packet was crashing the controller—even though, theoretically, the packet contents should not have affected the operation of the network adapter. Eventually the company determined that the interface shutdown was being triggered by a specific byte value at a specific offset—a condition that occurs only intermittently on a network. The issue was caused by misconfigured read-only memory chip of the network controller, resolved with a memory update. [11]

U-2 Caused Widespread Shutdown of US Flights
In April 2014, a U-2 spy plane flying at a high altitude caused a system failure at a California air traffic control centre, leading to the halting of take-offs at several airports in the southwestern United States. The plane's altitude and route apparently overloaded a computer system called En-Route Automation Modernization (ERAM), which searches for potential conflicts between aircraft. The flight plan data lacked planned altitude, so it was manually entered by an air traffic controller as 60000 feet (about 18 km). However, the system ignored these data and began evaluating all possible altitudes along the U-2's planned flight path for potential collisions with other aircraft, causing the system to exceed the amount of memory allotted to handling the flight's data. This, in turn, resulted in system errors and restarts, eventually crashing the ERAM look-ahead system and affecting the conflict handling of the Federal Aviation Administration for all the other aircraft in the zone. [12]

***

A shared theme in these cases is that systems often fail in unexpected ways. In the situations described above, would it have been possible to recover failures or limit fault propagation to a specific subsystem? The case of the ICE3 trains demonstrates the difficulties that can be encountered when integrating complex distributed control systems. Redundancy (e.g., triple modular redundancy) is an effective form of protection against foreseeable failures with known probabilities, but it is less effective against common mode failures and so-called "black swan" events, which are almost impossible to foresee before the actual incident. Black swan events can render all redundant versions useless at the same time, as in the infamous Jakarta incident of 1982, when all engines of a British Airways 747 failed simultaneously when the passenger jet flew into a cloud of volcanic ash [7]. The aircraft was able to glide out of the ash cloud, restart its engines and land safely. In such cases, resilience [13], or the ability of the system to cope with unforeseen disturbances and events, becomes a key system attribute. This thesis studies how systems can be designed to provide service regardless of faults, by improving system resilience through fault tolerance and diversity.

1.2.2 Cyber-Physical Systems

A CPS integrates computing and communication capabilities with the monitoring and control of entities in the physical world through a communication network, and it does so dependably, safely, securely, efficiently and in real-time [14]. Based on such a broad definition, CPSs are almost synonymous with networked digital control systems and embedded systems. To emphasize that the computational elements of a CPS collaborate and interact across the network to achieve system goals, we complement this definition with the one from CPSoS – Towards a European Roadmap on Research and Innovation in Engineering and Management of Cyber-physical Systems of Systems support action project from the European Union's Seventh Framework Programme for

Research, which takes CPSs to mean large, complex physical systems that are interacting with a considerable number of distributed computing elements for monitoring, control and management which can exchange information between them and with human users [15]. Although the manipulation of the physical world occurs locally, control and management tasks are not carried out in a centrally authorized manner (Figure 3). Due to the large number of computational units with independent decision making, the connections made by a system or its subsystems are expected to change. This increases the need for systems to cope with unexpected situations. There are challenges in developing software for such systems, since the systems' physical components introduce safety and reliability requirements, while their distributed nature introduces complexity and randomness. The ITER RH plant system can be considered an example of such a system, involving a need to interface with the RH equipment, including tools, heavy movers, manipulators, cameras, virtual reality (VR) software, control systems, etc. provided by different suppliers.

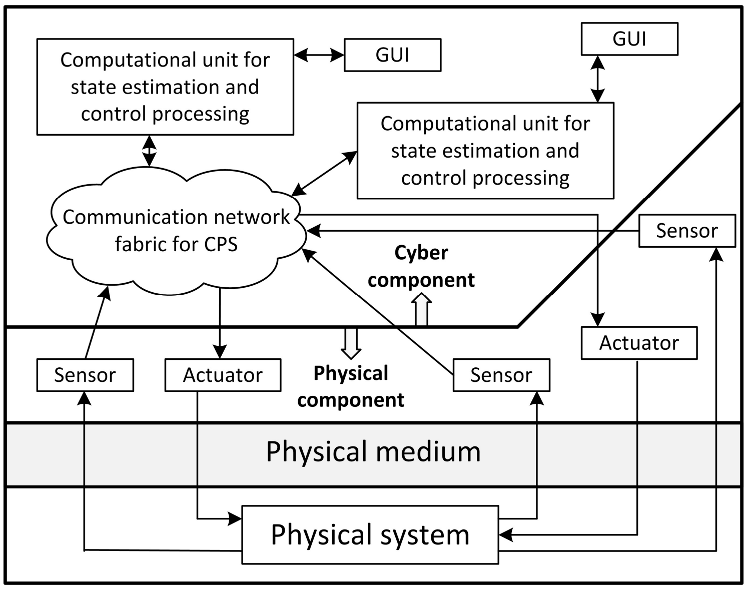

Computational unit for state estimation and control processing
GUI
GUI
Computational unit for state estimation and control processing
Communication network fabric for CPS
Sensor
Cyber component
Actuator
Sensor
Actuator
Physical component
Sensor
Physical medium
Physical system

Figure 3. CPSs are composed of computational units that communicate and collaborate over the network topology and interact with the physical system through sensors and actuators (adapted from [16])

Current real-time embedded systems are built using programming abstractions with few or no temporal semantics [17]; however, they must cope with the unpredictability introduced by networks and software, which may affect the stability and dynamics of their physical subsystems. Their interaction with physical processes, on the other hand, means that these systems are typically mission- or safety-critical. Therefore, CPSs must be capable of balancing dynamic behaviours with a need for dependability and temporal predictability. This capacity to function, despite unexpected changes in the environment—also known as resilience—is a fundamental requirement for CPSs. Resilience is needed to survive unexpected changes without failures and to maintain correctness in the context of improperly coordinated controls of cyber and physical resources [18]. A resilient system should be capable of providing other levels of utility besides binary "broken" and "working" states through diversity, dynamic resource allocation or some other means, so that a single failure does not bring down the whole system. Examples of such design include the limp-home pattern [19] and other alternative operation modes, such as the fault-tolerant automobile steer-by-wire system by Hayama et al. [20] based on diversity. This system uses braking and acceleration to provide limited steering capability as an emergency backup in the event of a failure of the actual steer-by-wire system.

Another integral feature of CPSs is their flexibility to evolve, including adapting to changes in computing environments and being cost-effective to maintain, reconfigure and re-target. Flexibility is seen as essential for systems to successfully interact and survive in a world of on-going and increasing digitalization. The digitalization trend has the potential to improve the efficiency of existing processes; however, it requires intelligent devices capable of providing useful services for people and software alike. However, there are challenges in completing systems in time and fault-free using the current "rigid" system designs; thus, achieving this for the even more complex systems of systems in the Industrial Internet will be a challenging task. Benefits such as utilization rate optimization, improved usability, condition monitoring and support for business decisions will not be realized in practice if the bases for these functionalities do not exist, since they will introduce more complexity to be managed in a dynamic computing environment.

From a technological viewpoint, CPSs are closely related to the concepts of the Internet of Things (IoT) and the Industrial Internet (see Sections 1.3.3 and 3.2). Wireless and Internet communication mechanisms provide increased connectivity options for distributed systems, making CPSs part of the IoT trend. The main drivers behind the IoT are the predicted cost-savings achieved through improved efficiency and decision-making capabilities enabled by advanced analytics, data sharing for collaboration, the ability of smart machines to provide more sophisticated services for users, inter-machine co-operation, remote operation capabilities, systems customizable to customer needs on-the-fly, etc. To achieve these capabilities, a fundamental requirement for automation systems is the provision of real-time access to information for interested parties. This transformation can be supported by moving from monolithic designs with proprietary interfaces to more open and modular architectures. One potential approach to implementing such flexible systems is service-oriented architecture (SOA)—an architectural style that has emerged as a dominant architectural paradigm for creating enterprise and consumer systems capable of cost-effectively adapting to carry out new business tasks by utilizing existing services.

1.2.3 CPSs and Dependability

It is possible to verify or extensively test the code of closed control systems with limited connectivity. For example, the automotive industry uses an approach in which each function of a vehicle has its own electronic control unit (ECU), which can be developed and tested separately (Figure 4 on the following page). This is necessary in order to manage the large amounts of software coming from multiple vendors that are required in modern cars to control safety, emissions, communications, convenience and entertainment functions, among others. Retaining test coverage over growing functionalities of new ECUs leads to longer test periods [21] and the complexity of integrating tens of ECUs means that the system integration phase becomes the major source of challenges. The downsides of increasing the number of ECUs include also the increased component costs, weight, heat and power-consumption. Therefore, it is unclear how well the historical approach of using separate hardware modules to isolate safety-certified applications scales to the need for integrating more networked and complex functionalities in CPSs.

To manage software and hardware complexity in vehicles, the automotive industry has developed the open AUTomotive Open System ARchitecture (AUTOSAR) standard, which can be used to

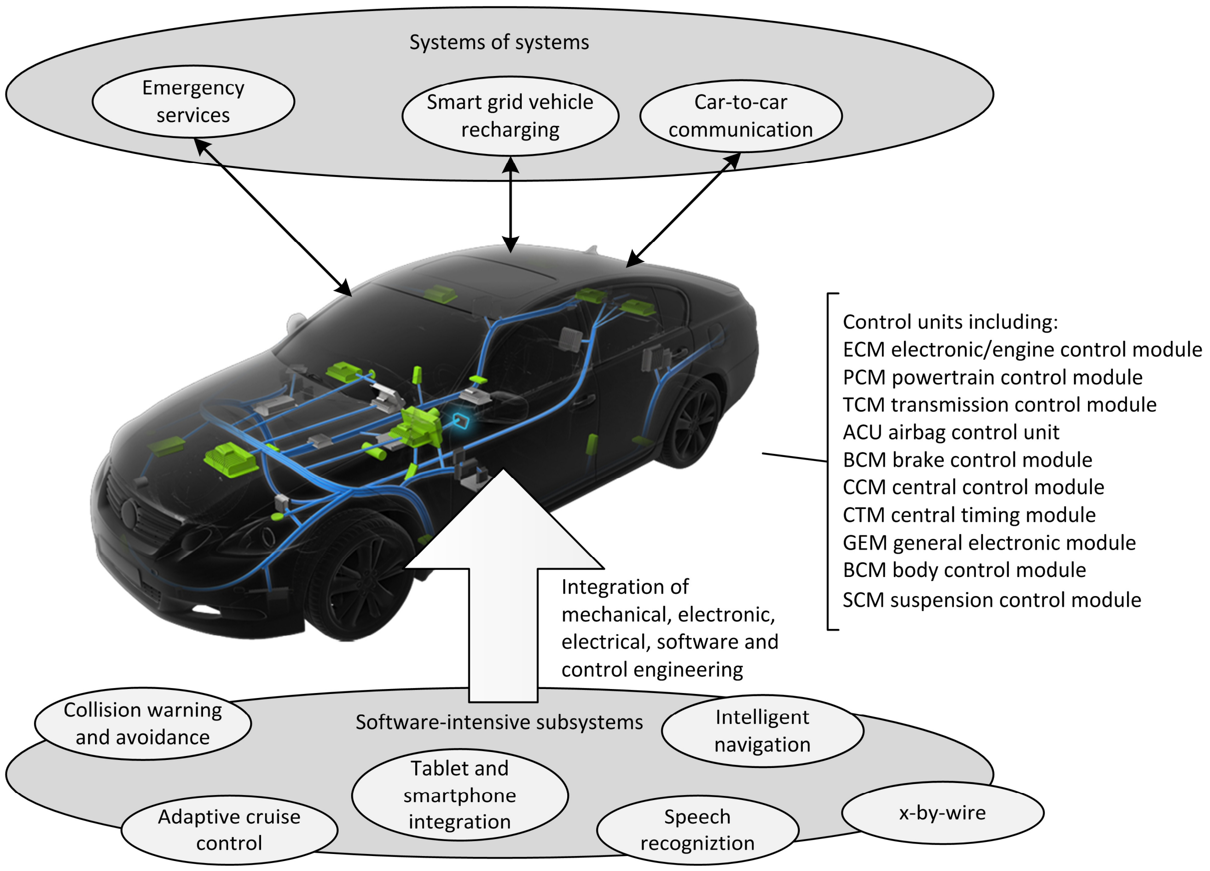

Systems of systems
Emergency services
Smart grid vehicle recharging
Car-to-car communication
Control units including:
ECM electronic/engine control module
PCM powertrain control module
TCM transmission control module
ACU airbag control unit
BCM brake control module
CCM central control module
CTM central timing module
GEM general electronic module
BCM body control module
SCM suspension control module
Integration of mechanical, electronic, electrical, software and control engineering
Software-intensive subsystems
Collision warning and avoidance
Intelligent navigation
Tablet and smartphone integration
Adaptive cruise control
Speech recogniztion
x-by-wire

Figure 4. A smart car as a CPS functioning within a larger ecosystem (system of systems) (adapted from [160]; vehicle image courtesy of Mobile Devices)

integrate a high amount of functionality into a single ECU in a controlled way; however, the migration process to adopt it is still underway [22]. Other proposed approaches in the industry include using language-, location- and platform-independent middleware (e.g., Common Object Request Broker Architecture (CORBA)), to connect and manage tasks [22].

However, the added intelligence and advanced features, such as x-by-wire applications and machine-to-machine (M2M) communications, mean that CPSs become more vulnerable to software faults caused by, for example, timing or integration issues that have potential to cause severe damage or economic losses; in such cases, the smart car in Figure 4 could just as well be a forest harvester or an ITER RH robot. The networked CPSs require new approaches to real-time fault tolerance and reasoning about consequences of faults because the fault tolerance of CPSs cannot be solved solely as a software problem—since, by their nature, these systems function on the tight coordination among hardware, software and physical elements.

Further challenges faced in the development of CPSs are introduced by their cross-disciplinary natures, since engineers must define requirements, interfaces and error management for components that bridge engineering disciplines both horizontally (subsystems and functions) and vertically (software, computers, physical platform). For a hydraulic divertor cassette mover, these subsystems would include, at a minimum, electrical, pneumatic, hydraulic, mechanical, power unit and user interface subsystems. Components and subsystems are likely coming from multiple vendors from diverse engineering disciplines with specific domain expertise; thus, they must be somehow integrated to an effective and dependable system.

CPSs are networked computing units that monitor and control physical processes (often in feedback loops). This is a particularly cogent issue when considering that, compared to classical control theory, these systems include dynamics introduced by networks and software (e.g., the timing jitter in communications and computation, packet losses in networks and resource contention) that may affect the stability and dynamics of the physical subsystems. If timing affects the system, small changes in environment, software or hardware can cause unexpected changes in timing for a brittle design. As a consequence, manufacturers of embedded systems with long lifecycles need to stockpile components. They cannot take advantage of improvements in computing capabilities because the testing costs for new hardware would be too high. This can lead to an outdated system architecture, obsolete hardware and unsupported software.

The claim that software does not decay is a fallacy, as pointed out by the 1st law of Lehman [23], which states that, since the environment of the real world is constantly changing, in order for a system that performs real-world activity to continue to be relevant, it must adapt and evolve as the world as well. The alternative is for the system to become progressively less applicable and useful. The 2nd law of Lehman states that, as an evolving program is continually changed, its complexity increases unless work is done to maintain or reduce it. These laws are highly relevant for automation systems with long expected lifetimes, especially when moving to increasingly distributed designs. Since the systems will need to be maintained (changed) in order to retain functionality and security, the system dependability may degrade. Therefore, the evolvability of the architecture, which can be supported through a high level of decoupling between subsystems and components, is needed to facilitate the capability to maintain software.

1.2.4 Safety-Critical Software and Related Standards

Software and control systems are usually categorized in the following way, using a vehicle application as an example [24]:

- *Safety-critical* software (e.g., brake control).
- *Safety-related* software, intended to maintain safe conditions or prevent risks (e.g., seatbelt pre-tensioning and airbag control).
- *Non-safety-related* software (e.g., air conditioning control).

As can be seen from the categorization, not all control software is safety-critical. The scope of what is considered safety-critical depends from the context; for example, it can cover the whole control system software if a fault can cause economic losses (e.g., losing a satellite) or endanger human lives (e.g., through haywire mobile machinery). Moreover, a non-critical component that can influence the safety-critical components becomes safety-critical itself [25]. *Mission-critical systems* may have a combination of the abovementioned software modules; therefore, partitioning (i.e., the separating of different software units) is usually used so that a failure in a multimedia system does not cause a crash in brake control software [24].

Standards can be used to show compliance with legal requirements, which in the case of the ITER, may include the European Union machinery directive, French nuclear laws, etc. IEC 61508 is a standard for developing software-based safety-related systems. It sets out the requirements for ensuring that these systems are designed, implemented, operated and maintained to provide the required safety integrity level (SIL), which ranges from 1 to 4 (highest reliability). The standard also has domain-specific variants, such as IEC 62061 for machinery. In the United States, the "DO-178C Software Considerations in Airborne Systems and Equipment Certification" document and the "MIL-STD-882E Standard Practice for System Safety" military standard are commonly used references. According to a report completed for U.S. National Academies [26], certification of the dependability of software-based systems is considered to typically rely more on assessments of the development processes than on properties of the system itself, due to the difficulty of assessing the dependability of software.

Restrictions set by the safety standards can conflict with the vision of creating CPSs that can function in dynamic computing and physical environments, perform autonomously or in co-operation with humans, etc. It is still unclear how software-based safety functions could be implemented cost-efficiently; current approaches rely on separating safety and control systems [27], but this approach prevents the utilization of the increasingly larger amounts data available in the control systems of the machines to support the safety systems. Currently, such reuse of information in the safety system is avoided in order to prove that the safety system is independent from other systems and not affected by fault propagation. However, this kind of information could provide an improved level of safety as part of a layered safety approach.

1.3 Objectives and Scope

1.3.1 Objectives

The objective of this thesis is to develop a fault-tolerant control system software architecture and to provide contributions to cost-effective practices for a dependable and resilient design based on the requirements of CPSs. The architecture should be standards-based and implemented with open and interoperable platforms. ITER RHCS is used as a target CPS that has requirements for fail-safe and recoverable design against any SPoF.

SOAs are used to build highly interoperable systems, since services can act as uniform and ubiquitous information distributors for a wide range of computing devices [28]. Services must be technology-neutral, loosely coupled and supportive of location transparency [28], making them ideal building blocks for CPSs. This thesis investigates the use of SOA as a real-time and fault-tolerant platform, since there is limited research available on the subject of the fault tolerance capabilities of RTSOAs, despite the fact that loose coupling and modularity are recognized as integral parts of fault tolerance.

In addition to the fault tolerance capabilities of an RTSOA, a systems engineering methodology is used and investigated to achieve a dependable design. This approach is used to gain an understanding of the research problem, develop a concept design and prototypes, and evaluate the applied solutions.

1.3.2 Research Questions

Based on the research objectives, the following research questions are presented. These are the refined research questions that this thesis aims to answer within the scope described in the next section. The questions are numbered and referred to as Q1-Q7.

- Q1: What methods and practices are used to achieve dependable designs for control system software?
- Q2: How can CPSs be designed to provide service, regardless of faults—and, especially, how can fault tolerance be implemented without the extensive use of diverse redundancy (cf. N-version programming [29])?
- Q3: What requirements for system dependability exist for RH systems? How can architecture support the implementation and verification of these requirements?
- Q4: Can a modular and loosely coupled (service-based) architecture support the dependability of distributed and heterogeneous real-time systems?
- Q5: What mechanisms can be applied to detect and recover faults in such loosely coupled systems?
- Q6: How can fault propagation between services be stopped?
- Q7: What is the overhead of using fault tolerance based on loose coupling?

1.3.3 Scope

The main challenges of CPSs are defined by Kim and Kumar [30], as follows: 1) modelling and analysis of dynamics, 2) real-time computing and networking, 3) wireless sensor networks, 4) security and 5) design and development. Contributions of this thesis relate mainly to the topic of design and development, as well as, to some extent, real-time computing and networking. Since the performance, reliability, and production costs of control systems are becoming more dependent on those of software systems, managing complexity to make it easy to design and implement software systems for CPSs is becoming an important software technology research issue [30].

Figure 5 illustrates the relationships among CPSs, consumer-oriented IoT devices and fail-operate safety-critical systems. The Industrial Internet emphasizes top-down development, where enterprise business processes drive connectivity to subsystem and equipment levels. The Industrial Internet, therefore, incorporates, not only industrial automation, distributed robotics and "things that spin", but also the systems that interact with them, such as resource management and planning systems, into systems of systems.

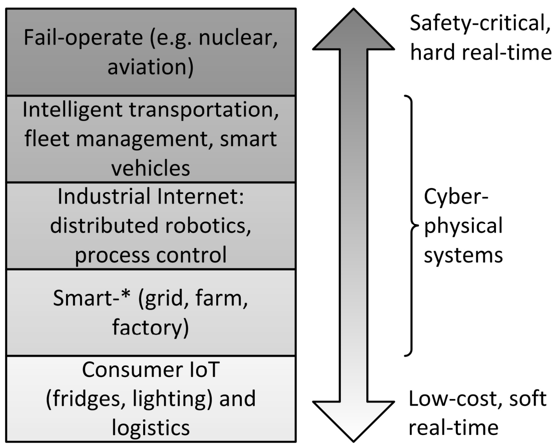

Fail-operate (e.g. nuclear, aviation)
Intelligent transportation, fleet management, smart vehicles
Industrial Internet: distributed robotics, process control
Smart-* (grid, farm, factory)
Consumer IoT (fridges, lighting) and logistics
Safety-critical, hard real-time
Cyber-physical systems
Low-cost, soft real-time

Figure 5. CPSs compared to consumer IoT devices and fail-operate safety-critical systems

The applicability of the results in this thesis is aimed at the middle layers, since these represent systems that have high dependability requirements but typically include separate interlock and safety systems providing fail-safe functionality or can be reconfigured to have simpler functionality. IoT devices, on the other hand, represent a bottom-up approach to using connectivity, sensors and data analysis to provide new product features and services for consumers. However, mass-marketing-oriented consumer devices tend to be low-margin and low-cost, with limited viability to use high-performing CPUs, operating systems and middleware solutions. More powerful and expensive consumer IoT devices have limited value for customers, especially if the connectivity does not provide sufficient meaningful value to offset the increased price.

Many cyber-physical and information systems can be considered safety-critical because significant financial losses or even the loss of lives could result from their failures and downtime. Since the difference between safety-critical and mission-critical systems is a fine line, for the research purposes of this thesis, the exact type of the control system (i.e. high availability, safety-related, embedded, etc.) is not considered. Instead, the focus is on finding methods and technologies that have potential to improve trade-off between system-level dependability and costs, while acknowledging the need to support the verification of the safety and timeliness aspects of control systems. The test case for the research is an RH system for a scientific facility with some safety related aspects; however, the primary object of the research is to study distributed and open real-time systems (i.e. CPSs) in general. The applicability of the results is analysed in Section 6.

Cost efficiency is used to narrow the scope of the research by leaving, for example, formal methods and the so-called roll-forward fault tolerance techniques outside. The use of formal methods can be cost-efficient for systems with high dependability demands [31]; however, their use would limit the potential to use commercial off-the-shelf (COTS) solutions, 3rd party libraries, etc., that are almost impossible to formally verify. A key problem for CPSs is that traditional real-time programming models specify timing properties indirectly, by associating priorities with tasks. The development of programming models for timed systems is an essential research topic, but it is also outside the scope of this research.

From the means to attain dependability, fault prevention and fault removal are considered to be part of the systems engineering approach, although the study's main contributions are to the topics of architecture and fault tolerance. Fault forecasting is covered, since an understanding of failure modes (fault taxonomies, Section 5.1.2) is essential to focus dependability efforts on the most serious and probable failures. Fault forecasting is also related to the estimation of the present number of faults.

System lifecycle scope is limited to the concept and development stages in the application of the systems engineering approach—that is, the operations period, including maintenance and disposal activities, is not evaluated in this thesis. However, system evolvability and the reusability of services affect requirements and system design, and are, therefore, considered in this thesis.

1.4 Methodology

The research methodology of this thesis is based on design science and systems development, including prototype implementations. Moreover, the research is carried out using a systems engineering process (shown in Figure 6), derived from the ITER quality assurance processes to develop and research new approaches to the research questions. The figure also shows how the publications [P1] through [P6] are related to the systems engineering process.

The process consists of following steps:

- Pre-project analysis of the research problem and domain.
- System requirements—research of the subject matter.
- Concept study—initial system architecture, prototyping and verification of the prototype vs. requirements.
- Final design—refining and extending the architecture and the prototype, evaluation of the system.

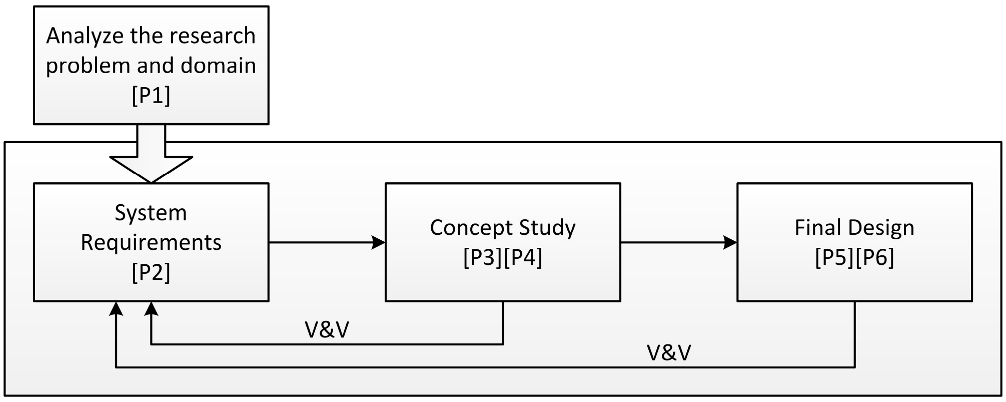


Figure 6. Major phases in the applied systems engineering process and the publications included in this thesis

Software engineering research, including the study of software architectures, is affected by factors that complicate the scientific evaluation of experimental results of software systems [32]. The skill factor means that the result of an experiment is vulnerable to subject and experimenter bias. The lifecycle factor means that the behaviour of the system may change once the system is deployed. Similar problems are encountered with systems engineering research, in which the application of systems engineering methods and expertise typically happens out in the field [33]. This makes objective evaluation of the specific methods difficult, since every situation is different. Due to these challenges, the research in this thesis uses a constructive research methodology of systems development, instead of the empirical research methods (e.g., case study, action research, controlled experiment, survey [34]) usually applied in software engineering research.

The idea of constructive research involves the construction of artifacts based on existing knowledge used in novel ways [35]. Construction proceeds through design thinking into the envisioned solution, and it fills conceptual and other knowledge gaps by purposefully tailoring building blocks to support the whole construction [35]. The artifacts—constructs, models, methods or instantiations—are then tested in laboratory and experimental situations to evaluate the utility, quality and efficacy of the design artifact [36].

The systems development research methodology used in this thesis is similar to the research process for constructing artifacts proposed by Nunamaker et al. [37]. The process, shown in Figure 7, is similar to conventional systems and software engineering development models. The key differences between design science and the practice of building IT systems would, thus, be the scientific evaluation of the artifacts and the rigorous constructive design method [36].

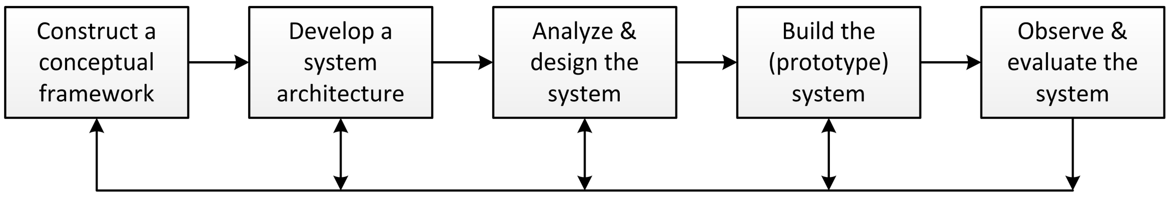

Construct a conceptual framework
Develop a system architecture
Analyze & design the system
Build the (prototype) system
Observe & evaluate the system

Figure 7. Process for systems development research [37]

To use the constructive research method, it is necessary to first gain an understanding of the domain and the problem. Therefore, the initial phases of the systems engineering process and the papers [P1] and [P2] focus on understanding the problem and the requirements, as well as on setting the research questions. Then, the system architecture is developed and analysed ([P3] and [P4]). [P3] evaluates capability (feasibility) of real-time services as building blocks for the architecture and includes an implicit hypothesis that the platform can be used to implement an RTSOA. [P4] has a working hypothesis that mission-critical real-time systems can use service management to recover from transient faults. After building and evaluating the initial prototypes in papers [P3] and [P4], the final prototype system is built and empirically evaluated in [P6], based on the requirements and research goals, stating explicit null hypothesis that let-it-crash approach cannot support real-time fault detection and recovery, which is disproved. The patterns presented in [P5] can be seen as additional artifacts, which document the insights gained from the system development research process.

1.5 Contributions

The research contributions of this thesis focus on presenting an architectural approach for building resilient CPSs using fault tolerance, evolvability and diversity. The architecture has been designed to meet the needs for interoperability and the timely dissemination of data in open environments. Furthermore, this thesis shows how the fault-tolerant architecture can be incorporated into a holistic systems engineering approach for the building of dependable systems.

System resilience is improved by utilizing loosely coupled architecture, dynamic reconfiguration and recovery based on the fine-grained restarting of processes (i.e., services), rather than focusing on robustness and the mean time to failure (MTTF). Loose coupling is achieved by relying on a service-oriented architectural style and dedicated communication buses. Based on the results presented, SOA can support the fault tolerance of real-time distributed systems through the use of services as units of fault isolation and recovery without significant overhead, diversity (i.e., utilizing existing diverse operating modes and backup services) and the straightforward application of fault tolerance patterns.

The proposed systems engineering approach is based on combining evidence from qualitative and quantitative evaluations, VR model-based dynamic testing, and the use of proven-in-use components and domain-specific fault taxonomies to build and maintain a dependability case for system under development. Cost-effectiveness is supported through the use of COTS components, reuse of services and subsystems, and the ability to flexibly incorporate appropriate fault tolerance techniques and patterns into the architecture. In detail, the claimed contributions are the following.

Service-oriented architectural style (Q4, Q5, and Q6):

- Architectural specification of RTSOA based on the use of middleware and real-time operating system (RTOS), including more detailed specifications of services, fault tolerance and service management when compared to the reference RTSOA (Sulava architecture publications [38] [39]).
- RTSOA implementation for a CPS application to evaluate its fault tolerance capabilities by allowing partial failures.
- Application of distributed microservice design principles in a demanding mission/safety-critical real-time application, which is verified empirically through the implementation of a distributed closed-loop control system.

Fault tolerance of CPSs (Q5, Q6, and Q7):

- Documentation of the patterns DATA-CENTRIC ARCHITECTURE, SERVICE MANAGER and LET IT CRASH and the pattern language for CPS fault tolerance.
- Evaluation of fault tolerance (i.e., detection and recovery) based on loose coupling and services as units of fault isolation and recovery for CPSs, including evaluation of overhead for the approach.
- Evaluation of fault tolerance based on applying the "let it crash" and supervision approach to real-time fault detection and recovery for CPSs.
- Implementation of resilience based on diversity (i.e., diverse operation modes and backup services) and services as fine-grained units of fault tolerance for CPSs.

Systems engineering of CPSs (Q1, and Q2):

- Evaluation of cost-efficient methods and practices for developing dependable CPSs.
- Proposed systems engineering approach for developing CPSs by combining different dependability methods into a dependability case.
- Support for verification of the system, based on loosely coupled and autonomous services that can be used as units for dependability methods and techniques, including dependability case, and can be tested with VR models.

Systems engineering of RH systems (Q3):

- Analysis of the role of software requirements for RH system reliability, including a comparison of ITER's RAMI process and general software dependability requirements.

- Categorization of RH faults using fault and failure taxonomies for RH systems, developed by analysing control system architectures and real operational data (i.e., bottom-up approach) from Joint European Torus[2] (JET) RH systems.

[2] Fusion research facility located in Culham in the United Kingdom.

2 Dependability and Systems Engineering

This state of the art review aims to answer the research question Q1, how to achieve dependability for control systems. The approach is to analyse how the dependability and resilience of CPSs can be improved from the systems engineering perspective, with focus on ability to build systems that can dependably function in open and networked environments. Fault tolerance and architectural solutions are emphasized, which reflects the results and the contents of the publications. However, since the approach of this thesis is based on achieving dependability, which relies on having an overall solid development process, fault prevention, fault removal and fault forecasting are also evaluated from the systems engineering perspective.

Dependability and especially fault prevention methodologies (categorized as "part of general engineering" in [2]) have much in common with systems engineering approach (Section 2.1), while fault forecasting has strong ties to safety analysis (Section 2.4). Fault tolerance is used to avoid service failures when some part of the system fails (Section 2.2), which is essential for guaranteeing the reliability of safety-related subsystems and limiting fault propagation. Fault removal (Section 2.3) is tightly coupled with the development process, and thus also with fault prevention. Dependability case and safety case are systematic approaches that can be used to collect evidence for proving the dependability or safety of the system; they are studied in Section 2.5.

2.1 Systems Engineering

Systems engineering is defined by the International Council on Systems Engineering to be an interdisciplinary approach and means to enable the realization of successful systems [40]. This is achieved through processes on enterprise, technical and system life cycle levels. In this thesis the focus is on the technical processes and especially development-related activities, that is, requirements definition and analysis, architectural design, V&V and integration (e.g., interface design) [40]. These phases are captured in the V-model, which is an idealized model of the development process.

An example is shown in Figure 8 on the following page of how the different methods to achieve dependability could be applied in the different development phases in the V-model. The qualitative fault forecasting methods are typically used for hazard and risk analyses before the realization of the system [41], whereas other fault forecasting methods, such as fault injection, can be used to support testing efforts and collecting data from the actual system. Results of the fault forecasting evaluations affect system requirements and needed level of fault tolerance. Adoption of fault tolerance methods is a design issue to be solved in the design phases [25, p. 114]. Fault prevention and fault removal methods can have potential uses during all phases (e.g. validation of requirements), but are also associated strongly with implementation and testing phases, correspondingly. Finally, data collected from testing and field experience can be used to quantitatively estimate reliability of the system (e.g., number of found and fixed faults).

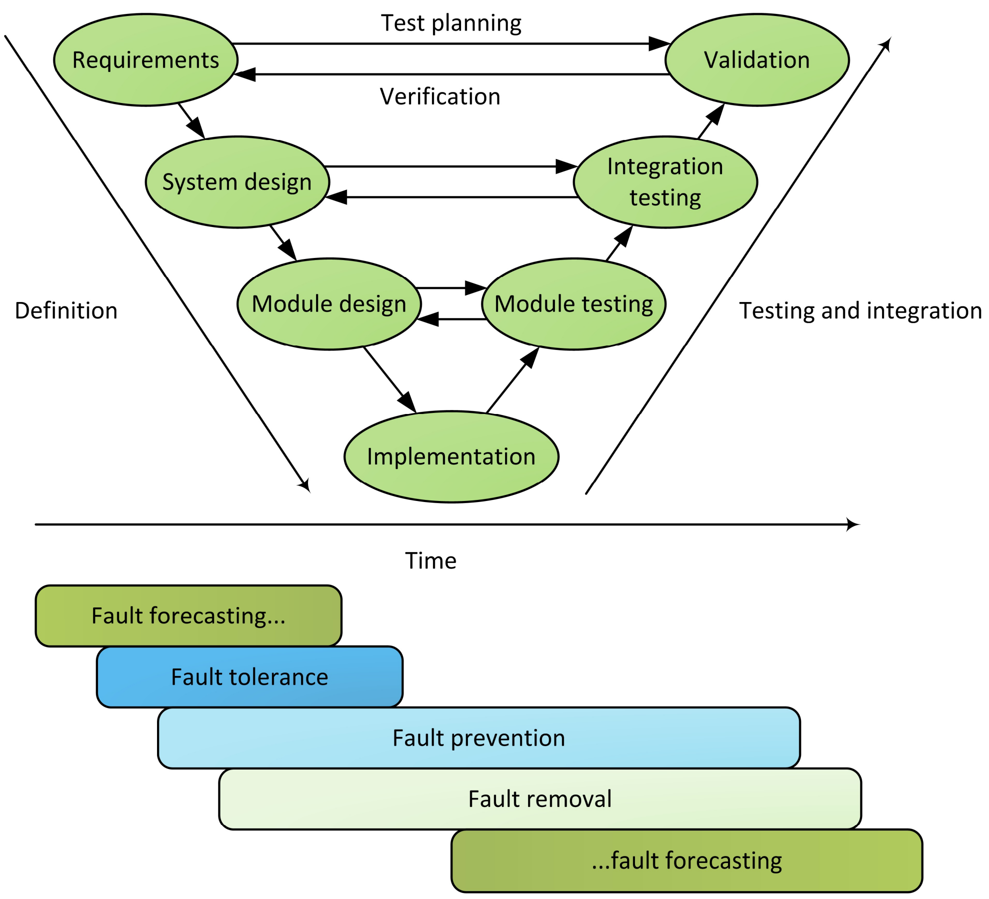


Figure 8. V-model compared with methods for achieving dependability

The objective that was set in the beginning of the research was how to improve system fault tolerance cost-efficiently. The objective of systems engineering is to see that the system accomplishes its purpose safely in a cost-effective manner, taking into account factors such as performance, cost, schedule and risks [42]. Figure 9 shows the trade-off space for costs and effectiveness. The graph illustrates the maximum achievable effectiveness of designs with current technologies [42].

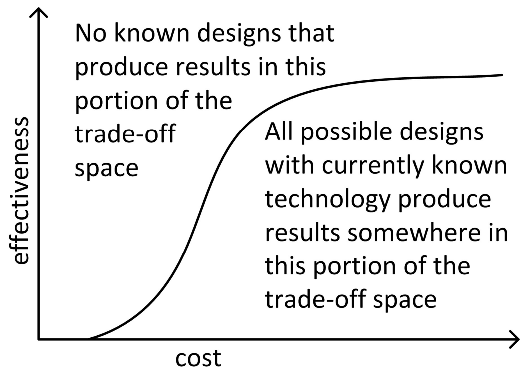


Figure 9. Enveloping surface of non-dominating designs [42]

Exact positioning of designs would be arbitrary and application-specific, but the graph is useful as a conceptual tool for comparisons. For example, a fault-tolerant design based on N-version programming (NVP) would be further right on the costs axis of the graph, compared to a simpler monitor-actuator-based fail-safe design. Goal of this research is to find solutions that exist on the low-cost part of the effectiveness axis and push the envelope, if possible. Alternatively, cost efficiency could also be improved by reducing costs of the existing solutions, but applicability of such results would likely be limited, even if verifiable cost reductions for a design were achieved.

2.2 Fault Tolerance

Fault tolerance is used to make systems more resilient to unanticipated faults, such as residual software faults (i.e., bugs). A fault tolerance solution consists of error detection and recovery [2], although some techniques address only one of these phases. According to Somani and Vaidya the hardware is getting more reliable and fault-tolerant but there is an increasing demand for tolerance of design, operator, environmental and reconfiguration faults [43]. This development is hardly surprising, since hardware faults are mostly physical faults whereas software faults are typically design faults, which are harder to visualize, classify, detect, and correct [44].

2.2.1 Fault-tolerant Architectures

Architecture and system structure have key roles in how the final system fulfils the quality attribute requirements, such as dependability and its sub-attributes. Examples of architectural design approaches that can be used to improve dependability include redundancy and modularity (see [41, pp. 91, 93, 107] or [3]). Modular approach entails design principles to implement information hiding, well-defined interfaces, loose coupling and strong cohesion, and to limit module size and structural complexity. Highly decoupled architectural styles and patterns (e.g., SOA and publish-subscribe) could improve dependability in control systems, since the designers of the consumer services cannot make the assumption that a service provider is available all the time, thus forcing them to take this situation into account in the design and implementation [45]. Decoupling limits fault propagation and also increases simplicity of the system, thus improving dependability [31]. These aspects are looked into more closely in Section 3.

Fault tolerance techniques applied in the architectural level make use of redundancy in one form or another, to the extent that redundancy is almost synonymous to fault tolerance. In order to make the system able to tolerate faults, it needs to be able to detect and react to errors and implementing these features adds redundancy to the system (e.g., code to detect faults). However, this thesis also uses redundancy as a shorthand for forward recovery techniques based on multiple versions (e.g., NVP [29]) or active/passive node redundancy [3].

Effectiveness of software redundancy techniques requires that redundant modules must be diverse—otherwise the software faults are replicated in all of the modules. This is typically avoided by using design diversity. NVP is an example of a well-known fault tolerance approach based on design diversity, in which two or more functionally equivalent programs are independently created from the same initial specifications and run on separate hardware channels [46]. This form of redundancy can be especially effective at tolerating intermittent faults. However, most of the software faults are design faults [47] and NVP does not solve the problems related to incorrect specifications. NVP has been criticized, for example, in [48] because programmers tend to make same kinds of mistakes. This may result in the different versions having the same types of faults, regardless of independent development. Faults do not need to be identical, it is sufficient that they are coexistent to cause a failure. Diverse software redundancy can create further problems due to slightly different behaviours between versions, especially when an abnormal input occurs into the system. Although Laprie et al. [49] show that an N-variant software module is less costly than N times a non-fault-tolerant software module, the costs of using design diversity are often hard to justify, unless necessitated by a fail-operate application (e.g., flight controller) or required by the functional safety standards. For example, in the IEC 61508 standard, diverse redundancy is "recommended" on SIL 3 and "highly recommended" on SIL 4.

Recovery blocks scheme, another commonly used form of design diversity, is based on backward recovery instead of forward recovery. Recovery blocks have better overall performance than NVP [50]. This is because NVP has to always wait for the slowest variant, whereas recovery block-based solution causes execution time overhead only when the system is recovering from an error. In hard real-time systems it should be noted that this overhead can cause the system to miss execution deadlines, whereas NVP has more consistent performance. Another similar solution to recovery blocks is retry blocks in which the fault tolerance is based on re-expressing input data on error detection. Unlike NVP, recovery blocks and retry blocks do not use multiple hardware channels for redundancy and they are recommended as cost efficient fault tolerance schemes in [51].

Triple modular redundancy (used as a hardware fault tolerance solution), NVP and other redundant systems also need a voting algorithm (e.g., 2 out of 3). The voter acts as an adjudicator that determines the correct output. This part of the system, implemented either in software or hardware, needs to be as simple as possible in order to avoid the need for introducing fault detection for the voter ("Who watches the watchmen?").

Dunn [52] presents a practical approach to implementing fail-safe systems without extensive redundancy. If a fault is detected, system can be guided to a known non-operating safe state. The fail-safe system can be implemented, for example, with the monitor-actuator pattern, in which a separate sensor channel watches over the actuator channel, looking for an indication that the system should be commanded into its fail-safe state [53, pp. 432-438]. NASA calls this type of systems failure-tolerant [25, pp. 114-115]. Even though the fail-safe approach can improve safety, the system reliability may decrease compared to fault masking techniques. Another threat for reliability with the fail-safe approach is false positive errors.

Erlang [5] programming language/runtime system uses a low-cost process based model to implement loosely coupled communication for processes, supporting a large number of small processes to carry out tasks. Initially developed for telecommunication systems, it has been used to achieve high system-level availability. Even if a single process terminates, it does not affect availability of the whole system. For example, Erlang-based implementation in web server has been shown to be more resilient and to have better availability than a similar C++ implementation. Erlang has been used for mobile robot control [54], to provide separation of processes and ability to recover crashed drivers. However, this research does not give information about system real-time or dependability performance.

Sotirovksi [55] uses object-oriented design approach based on encapsulation and exception mechanism for developing a fault-tolerant software architecture. The downside of using objects as units for fault containment is that a failure affects the execution of the whole application, including other objects.

Fault tolerance solutions have been documented also in the pattern format. Hanmer [4] covers patterns for fault-tolerant software in general, whereas Eloranta at al. [19] focus on distributed control systems and have documented fault tolerance patterns applicable for this domain. Nygard [56] documents patterns targeted for design and deployment of enterprise applications and includes patterns that can be used to improve system stability.

2.2.2 Approaches to Achieving Fault Tolerance Cost-efficiently

Single version fault tolerance techniques offer cost-efficient ways for error detection, confinement and handling without extensive replication of system functionality. Single-version fault tolerance can be efficient at detecting faults and subsequently moving the system to a fail-safe state (i.e., maintaining safety at the cost of system availability). Examples of single version fault tolerance include system structuring, checkpoint & restart, atomic actions, error detection codes and exception handling [57]. The downside of implementing fault tolerance without multi-version redundancy is naturally that the system is likely to be less capable at tolerating faults while meeting real-time requirements. Reliability requirements can thus be in conflict with real-time constraints in control systems [58] and more research is needed on how real-time systems can handle transient service failures in software components and subsequently restore the full level of service in a safe manner.

In [59], Kreutzfeld & Neese present a methodology called Data Fusion Integrity Process (DFIP) which aims to be a simple but effective technique by being designed to tolerate inconsistent errors with simplified recovery mechanisms. However, another paper did not manage to show any meaningful results achieved with DFIP [60]. This shows that it can be hard to verify the effectiveness of fault tolerance solutions in actual use and also that a system with ineffective fault tolerance is probably better without that solution because of the increased complexity and costs.

Hayama et al. [20] present a solution to implement fault-tolerant automobile steering using a diversified design. Cost and volume increases that are usually associated with diverse redundancy are avoided by using an existing subsystem (breaking) to perform functionality of another subsystem (steering). Their paper considers only a very specific case, but the diversified design principle could be generalized by considering how to reproduce the service provided by a system. In SOAs, diverse design can be achieved with alternative service compositions, especially if the services are deployed on different nodes.

Tai et al. [61] propose message and confidence-driven error containment and recovery to mitigating the effect of software design faults in distributed embedded systems. Basically, the approach uses checkpointing and rollback as a low-cost solution for error recovery. Confidence refers to the long onboard execution times for backups that are used for recovery. The approach enables implementation with generic middleware and application of the fault tolerance only to critical software components, allowing different integrity levels coexist in a system. Although the performance overhead of the algorithm is relatively low (roughly 5-35% depending on algorithm variant and message sending rate), potential challenges are related to using dynamically changing applications and reliable implementation of confidence tracking for complex systems.

2.2.3 Use of COTS Solutions

COTS solutions are non-developmental hardware or software items that are sold commercially and have a substantial user base, as defined by the U.S. Federal Acquisition Regulation. This definition covers also free software if it has commercial support. Free and open source software (FOSS) are commonly available for use as off-the-shelf solutions, typical use cases including software development tools, server software, operating systems and libraries. In the last 15 years, FOSS RTOS solutions based on general purpose operating systems such as RT_PREEMPT Linux and Xenomai-Linux have become viable alternatives for commercial RTOSs [62]. Use of general purpose operating systems has benefits that can potentially bring cost-savings: lower acquisition costs, large user base (increased likelihood of continued support), number of available tools and libraries.

Quality-wise open source software are on similar level with commercial software—many open source applications have been developed and maintained by expert-level developers for a long time, meaning that most of serious flaws and bugs have been fixed. The 2012 Coverity Scan report shows a defect density of 0.69 per thousand lines of code (kloc) for 118 open source software projects, which is same level as proprietary code used for comparison (only counting confirmed and potential defects identified by the Coverity software) [63]. In the 2013 report, open source code quality (0.59 per kloc) surpassed proprietary code quality (0.72 per kloc) in C/C++ projects [64].

New embedded multicore processors can also support use of different operating systems on separate processor cores, enabling configurations in which, for example, the RTOS executes timing critical code and Linux runs web server and other non-critical applications (Figure 10). This execution model could be used to provide separation of safety/mission-critical software from non-critical software, especially when combined with a hypervisor or other trusted execution environment technologies, enabling combination of connectivity requirements for the Industrial Internet with the mission and safety-critical nature of CPS on a single physical node.

Use of COTS-based software solutions is usually associated with improvements in responsiveness, reduced time-to-market and affordability with respect to custom-made software [65]. However, in the domain of certifiable safety-critical applications the benefits are not clear since extra efforts and

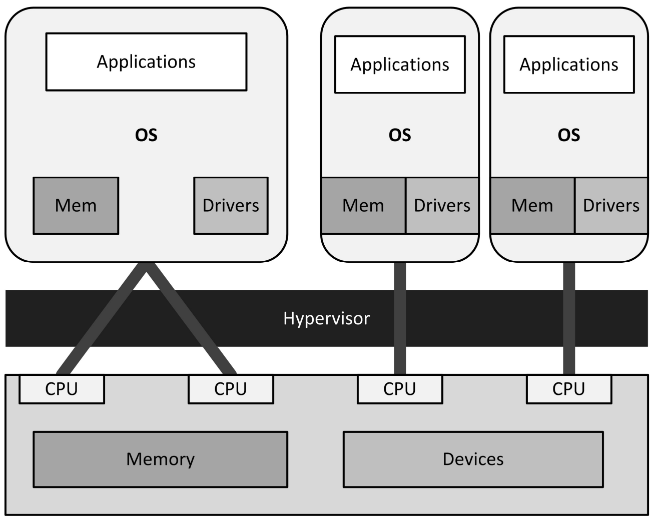


Figure 10. Supervised asymmetric multicore processing (adapted from [159])

costs should be factored for certification of the COTS components. In a case study about civil air transport systems done in 2007, it was found out that development and certification of COTS-based software is possible in a safety-critical domain, but the previously mentioned benefits were not observed for this particular case [65]. Another possible drawback of using common, inexpensive or free off-the-shelf software packages is increased security threats, since they can make the control system an easy target for cyber threats. According to Leveson [66], COTS software components are not suitable for safety-critical systems because using custom-built software, developed to fulfil the specific system safety properties from the beginning, would be cheaper and safer. However, there are safety-certified operating systems, middleware, etc. available, enabling developers to focus on core functionality and providing a potentially cost-efficient solution even for software that aims to fulfil safety certification requirements.

A research study completed for the UK Health and Safety Executive focused in the use of software of uncertain pedigree (SOUP), i.e. off-the-shelf software components, in safety-related applications [67]. The authors saw SOUP as an effective way to reduce development costs and also showed that extensive use will result in increased reliability (see Figure 11) with long term use strongly correlating with improved MTTF, even reaching a level that corresponds to IEC 61508 SIL 1. The challenge comes from identifying the proven components and demonstrating the reliability in new systems. To accomplish this they propose an approach based on evidence-based safety case (see Section 2.5).

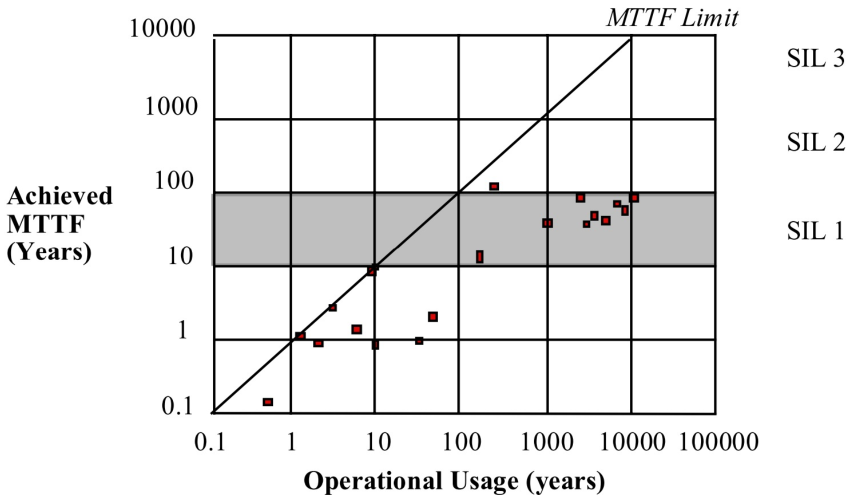


Figure 11. Software failure data from nuclear, chemical and aerospace (control and protection) industries [67]

Other notable publications and uses of COTS components include:

- For use in mission-critical systems, COTS software components can be seen as black boxes that usually do not have any guarantees for their behaviour and may have serious faults, etc. Integration of such components in the architecture would, therefore, benefit from additional fault tolerance mechanisms. In [68] Guerra et al. present an architectural solution of turning COTS components into "idealised COTS components" by adding a protective abstraction layer to them. They have developed and tested this approach with a self-developed layer-based architectural style.

- The JET facility uses COTS products designed for industrial applications to implement the unique remote handling control systems. This has the benefits of using commercially proven motion control software, easily upgradable hardware and ability to develop new features more quickly and confidently [69].
- Generic fault-tolerant computer architecture GUARDS was developed to tackle the cost and rapid obsolescence issues in real-time systems [70]. This architecture model, which uses fault containment to deal with faults, is based on the use of COTS components and allows the designer to choose redundancy dimensions for the system and determine differing fault tolerance mechanisms for components of different criticality level.

2.3 Fault Prevention and Removal

Fault prevention is a combination of general engineering expertise and use of specific development methodologies, that is, it resembles the systems engineering approach. The use of fault prevention to achieve dependability relies on rigour in the design and implementation to prevent introduction and occurrence of faults [2]. Fault prevention approaches include good programming practices and principles, such as information hiding and modularization, as well as development methodologies, such as use of modelling or formal methods. Building of the dependability case for a system—supported in [26]—is also a fault prevention approach.

Another way to affect cost efficiency of control system development would be to adopt modern and efficient platforms (including operating systems and programming languages), development infrastructure, better processes and tools to the development process, all of which can dramatically improve quality of software [31]. Agile development methods can also be applied to the implementation of safety-critical systems [71] [72], supporting cost efficiency indirectly with iterative and incremental methodology.

Formal methods (e.g., Z notation) are mathematically based techniques that can be used to describe system properties unambiguously. The use of formal methods might not be cost efficient if lower levels of confidence suffice, but with higher dependability demands, formal methods could be a better alternative than traditional software development [31]. Regardless of promised increases of dependability and possible cost savings, formal methods have yet to make a breakthrough. Some researchers, such as Sotirovski [55], suggest that the trend of increasing complexity will beat the advances in development of formal methods. Especially the use of a large number of COTS software or 3rd party libraries can potentially hinder formal verification. Nevertheless, use of formal methods is recommended for safety-critical systems in many standards, including IEC 61508 (recommended for SIL 2-3 and highly recommended for SIL 4).

Fault removal can be carried out during the development or use phases of a system. Former is verification and validation, latter maintenance. Software maintenance has an important role in improving performance and reliability, but by fixing faults this late in the development process is expensive. Nevertheless, components and software that has seen extensive use, in the form of long period of use and/or large amount of users, can be said to be proven in use [67]. Verification includes comparing the system to specifications to find out the quality of the system. Static verification (e.g., model checking) does not include execution of code whereas dynamic verification (i.e., traditional software "testing") does [2].

Verification can also be performed on specifications to ensure that requirements are correct, concise, traceable etc. Validation tries to find out if the system—or its specifications—matches the actual needs and constraints of the customer. Requirements validation can uncover faults in requirements, and is thus extremely useful since the cost of finding and removing faults is more expensive the later it is done. According to Boehm and Basili, fixing a fault after delivery is often 100 times more expensive than finding and fixing the same fault during the requirements and design phase [73]. After the requirements have been validated, the software can be verified against them.

For systems operating in dynamic environments, pre-deployment V&V may not provide sufficient trust for the dependability of the system in the live environment. Therefore, run-time assessability becomes an important feature for both fault forecasting and fault removal [13]. Simulators can support testing of service compositions and system configurations with full scale service deployments prior to moving to a live environment testing. For example, Devices Profile for Web Services, a specification for implementing Web services with minimal set of implementation constraints, has been used for simulated testing of an intelligent conveyor system in [74]. Vepsäläinen and Kuikka have used model-in-the-loop simulations to support model-driven development of industrial control systems, demonstrating the technique with a simulated crane system [75]. In enterprise systems, service virtualization can help companies to develop testing environments, based on using virtual services to emulate production services. Testing environments can be used to increase the frequency of integration testing and decrease testing costs [76].

2.4 Fault Forecasting and Fault Taxonomies

Fault forecasting is conducted by evaluating dependability of the system, which includes estimating the present number, the future occurrence, and likely consequences of the faults. Fault forecasting methods can be used for requirements specification, design analysis and V&V activities, with different types of methods suitable for each purpose. The methods for forecasting faults can be divided into two approaches: qualitative evaluation that aims to identify failure modes (e.g., failure mode and effects analysis FMEA) and quantitative evaluation that aims to evaluate probabilities (e.g., Markov chains), with some methods like fault-trees combining both aspects. [2]. Especially the qualitative methods are commonly used as part of safety analysis processes, but the probabilistic risk assessments are also used, for example, in the nuclear and aviation industries, to evaluate how likely some undesirable consequences are. Research of quantitative software reliability methods is related to the topic of this thesis in the sense that they could, at least theoretically, be used to evaluate reliability of services. If the architecture could guarantee fault isolation, it would directly benefit software reliability estimation in probabilistic risk assessments.

Risk assessment (sometimes called hazard assessment and/or analysis) is used in the design process of safety-critical and safety-related systems to produce safety requirements and proposals for design changes based on the analysed risks. The methods for risk assessment (e.g., failure mode, effects and criticality analysis (FMECA), hazard and operability study, etc.) can be quantitative or qualitative or these can be used in combination. The use of risk assessment techniques is not common outside of safety or mission-critical systems, but the application of the

FMEA process has been suggested for business critical services in SOAs [77]. Challenges of applying FMEA to such systems include dynamic service compositions, analysis of the effects of the failure modes, lack of knowledge on services and environment prior to implementation, SOA complexity, analysis results becoming quickly outdated and ability to agree to common scales for occurrence, severity and detectability of failures between teams developing different services.

Fault and failure taxonomies can be used to document domain-specific failure mechanisms and support fault-based analysis methods to prevent and identify faults prior to implementation. Domain-specific fault taxonomies are necessary because the failure modes of a system can be complicated to analyse. For example, feedback from the plant needs to be considered when developing control system failure modes [78]. Fault taxonomies can also support test planning of the system, especially for fault injection [79]. Taxonomies have been developed, for example, for SOAs [77] [80], NASA [81] and digital instrumentation and control systems of nuclear power plants [78], but not for RH systems according to the author's knowledge. In robotics, a systematic approach to choosing fault tolerance methods based on fault types expected in a given context and situation has been suggested [82].

According to Melchers [83], the problem with qualitative risk assessments, such as As Low As Reasonably Practicable (ALARP), is that the applied expressions are very subjective (low, reasonably, etc.) and they are best suited to dealing with established technologies in which potential problems have been already identified. Modelling and quantifying software failures and quantitative methods can be used to quantify software failure rate and probabilities for use in probabilistic risk assessments [84], but there is no consensus for modelling of digital systems in probabilistic risk assessments. Current software reliability estimates in the probabilistic assessments for nuclear power plants are often engineering judgments without proper justification [85]. Similarly, there is a lack of systematic reporting of software-related system failures in general which makes it more difficult to evaluate the risks and costs of software failures [26].

Reliability requirements presented for software in the format of probabilistic measurements can be a challenge to verify, if applied to software at all. Current approaches for quantifying software reliability rely on a combination of expert judgment, experience of similar systems and real-world data, if available. Requirements concerning the design procedure, coding practices and other qualitative requirements can be found from related standards, but addressing reliability requirements in a numerical format is an open challenge. Although SIL type requirements have been presented for systems that include software, those are always more or less accurate estimates. These estimates, however, are usually the best quantitative reliability requirement that can be given for software and, since they are presented as a number of allowed failures in a unit of time, they can be considered a probabilistic measure.

Software reliability prediction is usually based on stochastic methods, assuming that a population of supposedly identical systems, operating under similar conditions, fail at different points in time, thus necessitating use of probability theory. Probabilistic approaches to evaluating software reliability include:

- Bayesian networks, i.e. directed acyclic graphical models [86].
- Execution time based models, such as Musa's basic execution time model [87] and logarithmic Poisson model developed with Okumoto [88].
- Markov models—a type of stochastic model—including Markov Chains [89].
- Weibull or exponential distribution based estimation models.

Whether analysing software with stochastic methods (e.g., with Poisson process) is a reasonable approach is a valid question since software failures are often deterministic and faults are dormant until a specific functionality or sequence of events is executed, in the case of an internal fault. Dick et al. [90] analysed this question and found supporting evidence to their claim that software failures are a deterministic, chaotic process instead of a stochastic one. They analysed datasets from different software testing processes, and concluded that no standard distribution describes them well, thus assuming that the data sets had emerged from a deterministic process. The results showed improvements of 25% over the best stochastic models for two datasets.

Unfortunately, the models resulting from probabilistic analysis are complex to analyse accurately, so reliability modelling is not always a viable option [43]. Research on reliability models has been conducted over three decades and numerous models have been proposed, but not many software developers actually utilize them. In a survey conducted in the late 90s, the number of positive answers was around 4% when asking participants if they could use a software reliability model [91, p. 5]. According to Immonen and Niemelä [92], the biggest shortcomings for reliability and availability prediction methods are lack of support for tools, weak reliability analysis of software components and weak validation of the methods. Similar problems exist for component reliability prediction and certification in component-based software engineering, with a need for a way to predict component properties and a component quality model, to establish what kind of component properties can be certified [93].

2.5 Dependability and Safety Cases

The difficulty of predicting software failure rates has led to many certification standards adopting the approach of emphasizing the use of specific development methods and practices to achieve sufficient level of dependability. Jackson [31] considers this approach to be inefficient, since following the specified techniques imposes burdensome demands that produce large amounts of documentation and may prevent the choice of optimal development methods. The journal article is based on a larger report by a committee of dependability experts from academia and industry [26]. As a solution to more cost-efficient dependability, the committee proposes three things. First, a change of approach to "goal-based", in which developer must provide direct evidence (e.g., by testing, analyses, inspections and formal methods) that the system satisfies its dependability goals by building a dependability (assurance) case for the system. Second, making dependability claims explicit, since no system can be dependable under all conditions, and third, use of expertise to increase flexibility in development process by allowing experts the freedom to employ new techniques or deviate from best practices if necessary. Jackson argues that since the approach based on building the dependability case gives developers the incentive to use whatever tools are most economic and effective, it rewards innovation, especially when compared to conservative certification schemes and standards.

Dependability case is defined as "a structured argument providing evidence that a system meets its specified dependability requirements" that can be used to communicate information inside the organization or to a third party (e.g., certification authority) [94]. The concept is a generalisation of safety case, addressing attributes related to the dependability of the system, such as reliability, safety, security, real-time performance, interoperability, etc. A key difference is that the safety case is required for the certification of safety-critical systems, for example, in automotive, railway and defence sectors [95], whereas dependability cases have not yet seen widespread use.

To communicate a comprehensive and defensible argument that the system is acceptably dependable for a specific context, a structured argument notation can be used. The assurance case consists of three principal elements: requirements (claims about the system), arguments and evidence—arguments are used to link the evidence to the dependability requirements and objectives [96]. The goal of the notation is to show how objectives are successively broken down into sub-goals until claims can be supported by direct reference to available evidence [96]. Two major graphical notations for assurance cases are Goal Structuring Notation (GSN) [97] and Claims, Arguments and Evidence notation [98]. Object Management Group (OMG) has developed a standardized meta-model for assurance cases called Structured Assurance Case Metamodel, making both notations interchangeable [99].

A simple example of using GSN notation to develop a goal structure is shown in Figure 12 on the following page, including most of the principal symbols of the notation. G1 is the top level goal, C1 and C2 are contexts, S1 and S2 strategies, G_1, G1_2 and G_1_1_2 goals and S1_1_2 a solution. Furthermore, S2 and G1_2 are marked as "undeveloped". Additional notation includes justifications and assumptions, marked with ellipses accompanied with a letter J or A, correspondingly.

Arguments are necessary to link the evidence to the requirements; they provide the rationale how the evidence supports fulfilling of requirements. Arguments can be related to, for example, fault elimination and quantification, error activation, failure containment or failure effects estimation [97]. Evidence to support arguments could be based on the development process (documentation, testing, formal methods, etc.), system design or field experience [97].

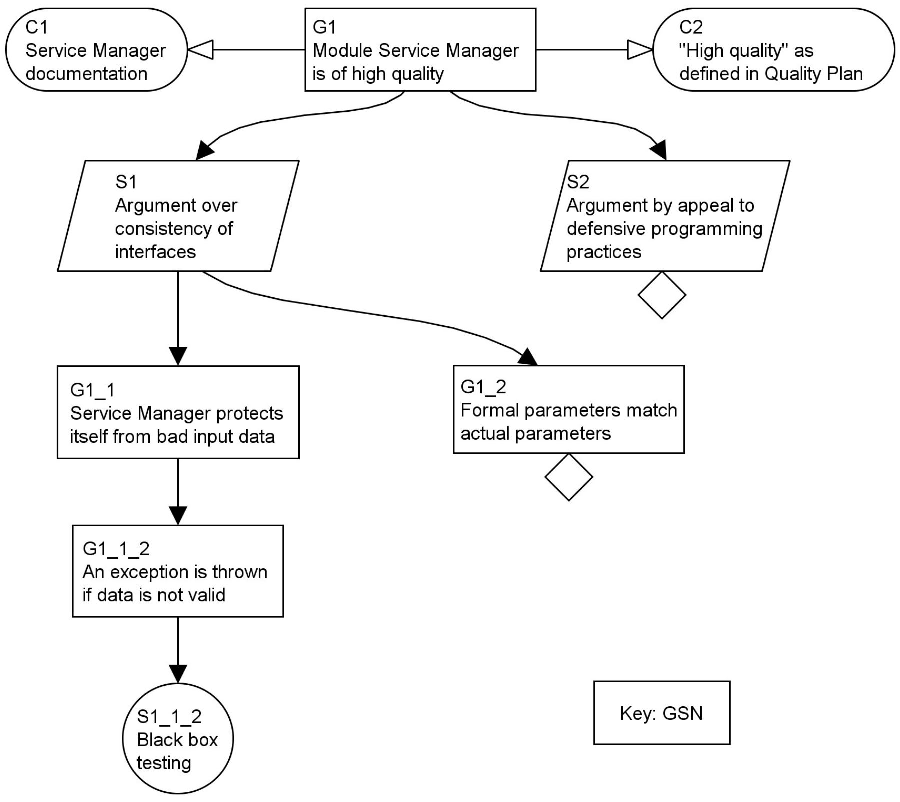

C1
Service Manager documentation
G1
Module Service Manager is of high quality
C2
"High quality" as defined in Quality Plan
S1
Argument over consistency of interfaces
S2
Argument by appeal to defensive programming practices
G1_1
Service Manager protects itself from bad input data
G1_2
Formal parameters match actual parameters
G1_1_2
An exception is thrown if data is not valid
S1_1_2
Black box testing
Key: GSN

Figure 12. An example of using GSN to assure the quality of a critical software module (adapted from [96])

Weaver argues that GSN provides a lightweight approach to determining what evidence is required to satisfy the requirements [96]. Claimed benefits of using GSN originate from identifying specific low-level requirements for evidence and combining different types of evidence to meet the requirements. The notation is used to develop the assurance case throughout the system development, allowing the low-level requirements for evidence to be identified and the need for testing and analysis determined during system design phases [96].

3 Architecture for CPSs: Towards Resilient and Dependable Software

Enterprise systems have already moved from siloed applications to open architecture [100] and this change is likely to affect embedded real-time systems as the prices for devices and communication costs are diminishing [101] and customers come to expect features enabled by collection and processing of data. The move to open architectures for devices is generally seen as part of the evolution of machine to machine (M2M) communications to IoT [102]. M2M refers to both wireless and wired technologies, but typically M2M communication standards are wireless with emphasis on low power [101, pp. 4, 14, 254].

CPSs benefit from the availability of low-cost, low-power, high-capacity small form-factor computing capabilities of general-use computing devices [18] and can potentially facilitate higher abstraction level architectures, middleware and operating systems even if they introduce some overhead. Transmission Control Protocol/Internet Protocol (TCP/IP) stacks and web server capabilities can be added even to low-cost embedded systems, enabling integration of CPSs into other systems with IP-based protocols. This section aims to answer research question Q2, how to implement fault tolerance in CPSs.

3.1 Challenges

Increasing computing power means that embedded systems have more processing capability for "intelligence" in the form of processing data into more meaningful information before sending it forward (e.g., smart sensors) to conserve bandwidth and support more flexible connectivity. Processing of data can reduce network load and improve efficiency of bandwidth usage, an essential feature for devices and systems deployed to remote locations, such as unmanned aerial vehicles (UAVs). Complexity of the interactions increases when communications change from basic actuation signals to complete movement sequence commands and from point-to-point measurement samples to data structures combining diagnostics data, especially since the logic is not necessarily centralized to back-end servers. However, this enables system experts to implement their know-how in a reusable form, for example, by sending notifications about predicted failures and need for maintenance from a device. To support implementation of intelligent services, systems should be extensible and facilitate independent deployment of new services, whereas currently changes in embedded system software typically require recompilation and re-programming of the complete software to read-only memory.

Resources are also not without limits, especially in real-time applications: running out of bandwidth, memory or CPU may result in consequences in the physical world. Development of safety-critical CPSs could move to the same direction as automotive industry so that units would be verified independently. However, in automotive applications the ECUs are separated on the hardware level. Independent verification of software components requires a high degree of independence and design for failure unless interfaces and potential interactions between nodes are known beforehand. Verification of software units is, therefore, still an open research question.

Challenges for software architecture of CPSs include the following:

- sharing of large amounts of data to optimize processes, utilization levels and predictive maintenance while having predictable timing for end-to-end latencies
- V&V, especially security and safety, of evolving systems
- scale increase from closed network to integrated system of systems
- flexible allocation of resources needed for scalability
- stakeholders want systems to be easy to use, build, maintain and repurpose [103]
- wide range of timing requirements [103]
- stakeholders want to use whatever communication network best meets application specific requirements
- increasingly large and complex systems (e.g., Airbus A380 and big science projects such as Large Hadron Collider) that need more people and organizations to develop them; systems will, therefore, be composed from more fragmented modules (cf. Conway's law) delivered at various time points
- support for evolvability, including modifiability and maintainability
- product variation for mass-produced systems (e.g., mobile working machine variants and options),
- mixed criticality embedded systems (e.g., infotainment system combined with rear view camera display in vehicles, similar challenges exist also in RHCSs, see [P2])
- rise of browser-based computing—integrating operational systems and Web-based applications
- architectural style should not limit the logical architecture (e.g., sense-plan-act or layered robot control architectures) or technology platforms

3.2 Approaches

Scalability and modularity of the architecture have significant roles when addressing almost all of the challenges mentioned in the previous section. Modularity of the architecture has been recognized early as an elementary part of building dependable systems. For example, software modularity and fault containment through fail-fast software modules are identified by Grey [104] in 1986 as key factors for fault tolerance. Modular design itself is a long-running development trend in the software engineering field, with the goals of managing software complexity, increasing software reusability and shortening time-to-market. Examples of this trend range from using modules for information hiding in 1972 by Parnas [105] to object-orientation, component-based software engineering (CBSE) and SOA.

CBSE emphasises use of reusable off-the-shelf and custom-built components to compose a software system, enabling systematic reuse of existing solutions, and the SOA paradigm is in a sense an extension of CBSE to more distributed computing environments. SOA can mean a variety of things—there are claims that the term has become too ambiguous [106]—but in this thesis we use SOA to describe an architectural style that consists of self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications [28]. Services must be loosely coupled, technologically neutral (comply with widely accepted standards) and support location transparency [28]. The technological neutrality means that the services can be accessed without knowledge of their underlying technologies. Services are typically implemented as autonomous applications that can be composed into collections (compositions) to carry out complex tasks, enabling efficient reuse. The status of SOA as de facto standard approach for building enterprise IT systems can be at least partly attributed to its capability to support integration of the applications built using different technologies, both on implementation platform and communication levels. Platform integration is supported in SOAs through open communication standards, such as Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML), whereas different communication platforms can be integrated with enterprise service bus (ESB) and gateway solutions.

M2M communication and IoT are technological trends that are related to CPSs; this relationship is illustrated in Figure 13, which extends the model from Figure 5 and emphasizes the viewpoint that CPSs and M2M communication are enabling technologies for IoT [102]. Other similar notions such as Internet of Everything, Industrial IoT, etc. have been proposed in the recent years, but ISO/IEC JTC 1 IoT workgroup predicts that is unlikely that these other notions will survive because they are too similar to IoT [103]. M2M includes technologies that can be used to provide connectivity for a group of devices with decentralized, mesh networking or telecommunications networks. Typically, M2M connections are wireless, with some of the devices acting as gateways and providing connectivity to business applications. CPSs can utilize M2M technologies to support mobile machines, integrate wireless sensors, provide remote connection capabilities, improve reliability by omitting central servers, etc.

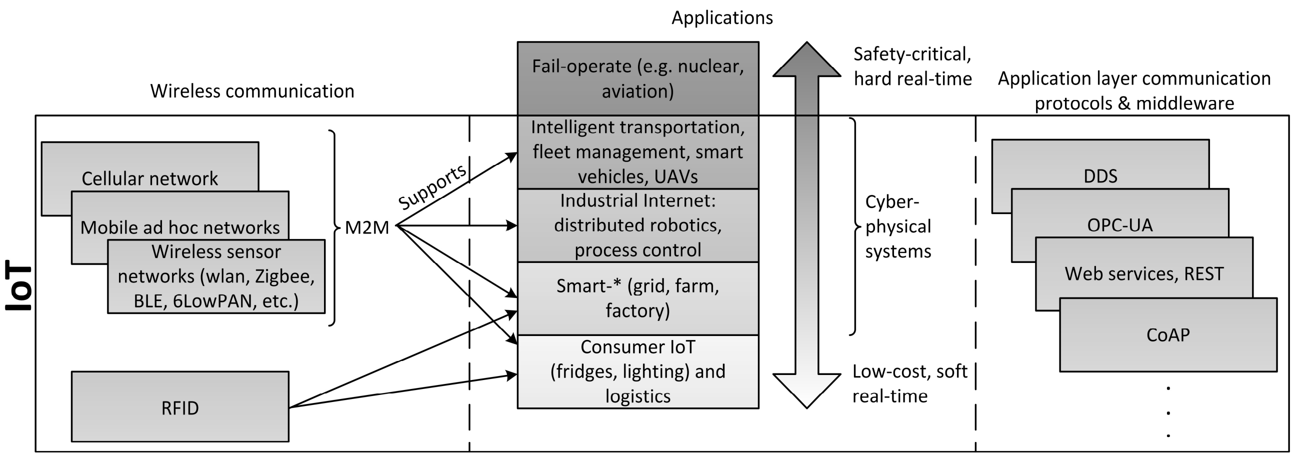

Applications
Wireless communication
Fail-operate (e.g. nuclear, aviation)
Safety-critical, hard real-time
Application layer communication protocols & middleware
IoT
Cellular network
Mobile ad hoc networks
Wireless sensor networks (wlan, Zigbee, BLE, 6LowPAN, etc.)
M2M
Supports
Intelligent transportation, fleet management, smart vehicles, UAVs
Industrial Internet: distributed robotics, process control
Smart-* (grid, farm, factory)
Consumer IoT (fridges, lighting) and logistics
Cyber-physical systems
Low-cost, soft real-time
RFID
DDS
OPC-UA
Web services, REST
CoAP

Figure 13. M2M technologies and CPSs in the scope of IoT

As a trend, beginnings of the IoT are in the possibility to connect things into Internet through Radio Frequency Identification (RFID) technology in the end of 1990s [107]. Over the time and through the advancements in technology, the concept of IoT has extended to cover the connectivity further to a "connect anything and anywhere" level, enabling physical objects to integrate seamlessly into communication networks and Internet. Industrial Internet is application of the Internet protocols to enable communication and cooperation between uniquely identifiable embedded devices and information systems in the industry sector, but the devices connected to the IP network may not always be visible to the global network for obvious reasons. CPSs can be seen as part of this trend, integrating physical processes and their control (computation) into networks.

3.3 Robotic Frameworks, Component Systems, Middleware and Communication Protocols

A basic need in software development is to create maintainable software for new systems by composing it from reusable blocks. This goal can be achieved, for example, with CBSE and component frameworks. Component frameworks can be used to build the system from smaller pieces obtained in various ways and to manage the complexity of the system by breaking it down into more manageable chunks.

In the robotics domain there are several frameworks available that aim to reduce the time and complexity of developing robotics systems by providing often-used functionality, such as kinematics, filtering, vision, planning, etc. as libraries. Other typically provided functions include process support and messaging interfaces for inter-process communications. These frameworks provide a way to organize the code into modules, while not dictating the arrangement of the modules (i.e., the logical architecture); although, the framework may affect the architecture indirectly, based on how modularity is supported or what middleware solution (if any) is used to implement the communication. Some of the more established robotic frameworks include Orca, Orocos, Player 2.0 and Robot Operating System (ROS). The frameworks also typically include tools to help development and a concurrency model for scheduling of the components. The concurrency model affects, for example, interoperability and data-sharing between components. While some frameworks support reusability with component-based or object-oriented approaches, others like ROS use processes as a unit of reuse and deployment [108]. Processes in ROS communicate by sending messages to topics with publish-subscribe semantics and could potentially be utilized to implement a domain-specific RTSOA model. Finally, the framework may provide capabilities to execute components in hard real-time, but not all robotic frameworks have this feature because for some applications, such as small mobile research robots, soft real-time is adequate.

Distributed generic component frameworks, such as the proprietary Distributed Component Object Model (DCOM) technology by Microsoft and the OMG standard CORBA provide a distributed component model. These days, some people consider CORBA to be a niche technology, largely due to its complexity [109], but it still sees use in real-world systems. DCOM technology was used to implement OLE for Process Control (OPC), providing real-time communication for control systems. OPC had limited configurability and support limited to Microsoft platforms. Some of the distributed component frameworks also have strong coupling between communicating components (component holds a direct reference to another component), which limits their usability as a basis for service-based system development. In order to stay competitive, OPC Foundation released an updated, service-based specification OPC Unified Architecture (OPC UA) [110] in 2008, which provides an SOA model targeted at process control, adding security and cross-platform support to OPC.

Middleware is a technology for enabling collaboration between nodes of a distributed system, abstracting the low-level infrastructure details for developers. By providing an abstracted communication layer for applications, middleware allows developers to focus on the application logic instead of platform specific communication systems or network topology. The raised level of abstraction also aims to isolate the system software from processor and network changes. There are various ways to categorize middleware based on the key abstractions provided, including transactional, message-oriented (which can be based on queues or publish-subscribe model), remote procedure calls (RPC), object/component oriented and service oriented [111]. DCOM and CORBA are object-based middleware, whereas Java Message Service (JMS) is a message-based middleware that supports both point-to-point and publish-subscribe communication models.

Data-centric middleware, such as Data Distribution Service for Real-Time Systems (DDS) [112], extend the message-based approach by making messages "transparent" for the middleware, so that it has more control over their delivery and prioritization. This enables, for example, extension of topics with new data types, while retaining backwards compatibility. When comparing message-based and data-centric models, data-centric middleware enables new subsystems joining the network to directly read the latest relevant data, whereas in a message-centric system messages have to specifically be resent to the new participant. The data publisher does not need information about the subscribers and vice versa. In other words, publishers are not required to know anything about receivers, only the specific data types being communicated. The data-centric approach is decoupled and scalable, since publishers and subscribers can join and leave the communication channel at runtime. DDS is also decentralized and supports explicit configuration of resource usage and reliability with QoS policies. Cross-vendor compatibility is achieved by using an interoperable wire transfer protocol.

TCP/IP is widely used as a basis for implementing messaging protocols, such as AMQP and MQTT (binary publish-subscribe protocols), Representational State Transfer (REST), SOAP (RPC protocol for Web services) and Devices Profile for Web Services (DPWS), which defines Web services implementation for resource constrained devices [113]. The protocols are complementary, serving different purposes and can be combined if necessary, for example, using JMS to send SOAP packages or using MQTT for sending telemetry and DDS to provide M2M connectivity. REST typically focuses on resources and state (data), but uses request-reply operations (client-server model), most commonly over HTTP. REST with Constrained Application Protocol (CoAP) can also be used to provide SOA capabilities for resource-constrained devices [113].

Middleware can be wrapped as an ESB to provide a communication platform for SOAs. For example, Mule ESB supports JMS in addition to several other protocols [114]. The ESB provides additional functionality when compared to a middleware platform, such as management and service orchestration capabilities, commodity services for protocol translations, etc. This has the effect of abstracting some of the complexity and message protocols from developers, but introduces coupling and risk of vendor lock-in to the system. An approach to building service-oriented system without complex ESBs and central orchestration (e.g., WS-Choreography) using REST or lightweight messaging buses from small and independently deployable services is called "microservices" [115]. Microservices refer more to the implementation style, rather than technology, and can be seen as evolution of SOA. The approach emphasizes ability of multiple teams to work on services independently (vs. monolithic server applications) and deploy them using continuous delivery, allowing frequent updates to services while keeping the rest of the system available. This enables rapid prototyping and lean development of services with less consideration paid to extending them in the future, avoiding over-engineering. However, this approach might be in conflict with safety-critical CPSs, thereby necessitating identification of safety-critical system components as suggested in [P2]. Also, the complexity is pushed to a higher level in terms of managing a large number of services, necessitating good operations support and automation for testing, deployment and orchestration of services, which might require custom-built tooling. SERVICE MANAGER [P5] pattern and its implementation in [P3], [P4] and [P6] can be seen as examples of solving the management problem of microservices in a real-time environment.

3.4 RTSOA

Generally, SOA style is associated with a number of potential benefits, including [116]:

- Ability to develop new function combinations rapidly.
- Organizational flexibility—the service-based architecture is easier to change than large, monolithic programs.
- Configuration flexibility—service communication is decoupled through messages. Services can, therefore, be moved between computer systems and compositions to meet changed business needs.
- Ability to integrate existing assets.
- Improved co-operation capabilities—use of fine-grained services supports information flow within and between enterprises when compare to the use of monolithic applications.
- Improved reliability through service virtualization.

These potential benefits have been recognized in CPS-related research and commercial sectors, including domains such as industrial automation, robotics and embedded devices. Applying of SOA in these domains can enable operation in the open and networked IoT environments and faster reconfiguration of the system for different products or tasks. However, to be applicable for CPSs, service architecture communications and service management must be dependable and predictable for interacting with the physical world in a timely manner. Therefore, the implementation technology should support QoS parameters to set and monitor priorities, deadlines and resending of the messages, and use standards-based, efficient encoding of the transmitted data. As standard SOA technologies have not been designed with strict timing limitations or QoS needs in mind, a distinction is made between SOA and RTSOA technologies. Despite the differences, there are common foundations especially in the design principles of the services, including service granularity, use of concurrency, deployment, etc.

Service orientation has the potential to support evolvability of the system, since services are loosely coupled components that can be analysed, replaced, reused, distributed and parallelized autonomously. Evolvability is defined as a system's ability to easily accommodate changes [117], having strong similarity to the concept of resilience, although from the development and maintenance point of view, whereas resilience has an operational emphasis. Benefits of SOA to system evolvability are clear in the enterprise applications sector where SOA has seen wide-spread adoption, but one of the research topics in this thesis is how it can support resilience of the system [P6].

An integral part of implementing service-orientation is the use of dynamic discovery and binding between services [118]. The dynamic nature of services supports reusability, discoverability and composability, at the cost of predictability. An SOA implementation targeting a CPS application needs, therefore, to provide means for application developers to support predictability of service-to-service communications, for example by allowing for negotiation of the quality of service (QoS).

Most of the current RTSOA approaches, such as [119], are based on the existing message-based Web service standards or the more streamlined REST style. Web services and REST style services are often implemented using standardized application layer protocols for message transmission, exposing the capabilities of the service over the standard Internet languages and protocols. Web services face challenges when used in embedded systems, since messages need to be serialized in real-time [120], and QoS must be managed at transport layer. Other challenges include the complexity of networking with HTTP, XML and SOAP, constraints imposed by embedded system architecture and the verbosity of HTTP and XML. For example, in a study applying Web service-based SOA to a wireless sensor network, the return time average was 1.3 s [121]. Although there are a number of factors that affect the performance (e.g., use of wireless technology, network topology and additional encoding for Efficient XML Interchange), the performance was sufficient for the target application of district heating substations. However, the general consensus seems to be that Web services are not suitable for applications with significant time constraints, such as RH.

The SOA style can support event-driven approach (i.e., asynchronous and non-blocking sending of events (messages) to interested parties), although an SOA is not automatically even-driven. Typically in the SOA services make calls for specific service providers in a point-to-point fashion, whereas using an event-driven approach the services publish events but do not know what service consumers are interested about that specific event. The event-driven communication needs a middleware or ESB to take care of notifying the interested parties about events (cf. event-driven architecture). Event-driven approach is suitable for event and state based interactions (publish events on state changes, sensor data, commands, etc.), whereas request/reply transactions are more challenging. They key benefits of the event-driven approach for implementing SOA are the support for scalability and resilience, due to the lack of service coupling.

3.4.1 Service Design

Service is the key abstraction in both SOA and RTSOA, acting as a basic building block for software reuse and enabling further architectural abstractions, such as service composition and service buses. Service is defined here as a self-contained and independent software implementation supporting a specific functionality or a task. Independence means that the services are units of deployment and integration that they can be developed, deployed, managed and maintained as separate entities. Due to the independent nature of services, implementation can be done using different languages and environments, moving some of the complexity from writing the services to configuring, monitoring and managing the service compositions.

Key difference to components is that instead of remote procedure calls, which imply some level of coupling between components, communication is based on asynchronous messages. The consequence of building systems using loosely coupled services is that the services need to be designed so that they can tolerate the failure of other services. This necessitates additional logic (and complexity) to handle these situations which may seem like a disadvantage. However, this has the outcome of making the overall system more robust, which is a fundamental requirement for a CPS. Some of the logic may also be externalized to middleware, libraries and service templates, helping to keep the core service logic simple.

The functionality of the service can represent a device (e.g., sensor, actuator), hardware abstraction (e.g., fieldbus), a basic computation task (e.g., kinematics, trajectory generation), or a complete (sub)system through a composite service, depending on the granularity level of services. Granularity has high impact on reusability of services, since smaller services are easier to reuse, but increase the communication and configuration overheads.

An integral part of the service architecture is the interface for the syntax and the semantics of message exchange. For example, in enterprise SOA implementations services typically use Web Services Description Language (WSDL) for Web services [122]. For services implemented with REST style, service interfaces may be documented manually or with the help of tools and metadata formats, such as WSDL 2.0 or JavaScript Object Notation (JSON) based Swagger [123] for better scalability.

RTSOA technologies are still in immature development phase and there are no standardized or widely adopted RTSOA-specific interface description languages. Possible alternatives include, for example, OMG Interface Definition Language (IDL) used with DDS and CORBA, SOA standards (WSDL and WS-*) and ontology based technologies. The challenge is describing the real-time properties of services, which may require V&V of these properties and strict specification of the environment. Cucinotta et al. have extended WS-Agreement protocol to support QoS and real-time attributes [124], but manual documentation in combination with existing standards (as in [P3]-[P6], see also Section 4.2) seems to be a common option.

Embedded systems are characterized by an application-inherent parallelism, since they consist of many concurrent, independent processes [125]. This is true especially for CPSs in which the system may need to interact and sense the physical world through a number of sensors and actuators in addition to communicating with peripherals and other systems in real-time. Traditionally small-to-medium scale embedded systems have utilized programming models based on shared memory [126]. However, recent introduction of multicore systems-on-chip favour use of message-based communication infrastructure for systems with high computation/communication ratio, typical for many industrial systems [126] [125]. Use of message-based architecture provides better function encapsulation and fault containment, and enables implementation of additional dynamic power management techniques when compared to shared memory or monolithic architectures. This thesis considers application of SOA style architecture in CPSs, building the communication over message-based infrastructure and loosely coupled components (i.e., services). Embedded systems represent a large application sector of CPSs which can potentially benefit from the RTSOA style, enabling the full exploitation of hardware capabilities.

3.4.2 RTSOA Research Challenges and Implementations

Research on RTSOAs ranges from the application of the service-based architectural style to industrial automation to the development of real-time composition and reconfiguration algorithms. This section presents key research challenges and research results relevant to development of CPSs. Different communication standards available for RTSOA implementation have been covered in the previous sections.

Application of SOA in industrial automation for reconfiguration of intelligent devices to support flexibility and agility for manufacturing has been suggested by Cândido et al. [127] to overcome issues such as centralized implementations, lack of fault tolerance, vendor and legacy system incompatibility issues and time-consuming reconfiguration. They list middleware, devices &

services setup, control, self-* capabilities, dynamic reasoning, simulation/validation and IT integration as necessary research topics for implementing SOA-based industrial automation devices. Several similar topics are identified by Rajkumar et al. [18], according to whom the key challenges for CPS research include composition, architecture, computational abstractions, real-time embedded system abstractions, robustness (cf. resilience), safety and security of CPSs, model-based development, control of hybrid time/event-based systems, V&V and certification of CPSs. There is overlap in these proposed research topics and several of them are touched on in this thesis, although the resilience and fault tolerance issues are mainly approached from the architectural point.

Tsai et al. [128] argue that service reconfiguration needs to be carried out in real-time to satisfy the timing constraints and propose an RTSOA framework including a search algorithm for service compositions using pre-verified services. The algorithm is verified with simulations.

Time-bounded reconfiguration for service-oriented distributed systems is achieved in iLand middleware, developed by Gárcia Valls et al. [129]. The middleware achieves time-bounded reconfiguration of services using a graph-based algorithm, which is empirically evaluated with a high-definition video application.

De Deugd et al. [130] present the results of a workshop for service-oriented device architecture for integrating embedded devices with distributed enterprise systems. The architecture provides monitoring and controlling capabilities for physical environment and is based on applying a device adapter layer to encapsulate device-specific programming interfaces for the services and using a bus adapter for presenting the device data as SOA services over an ESB. Similar model is applied in the Sulava RTSOA reference architecture (see Section 5.2).

Tiderko et al. [131] present a service-based framework for a wireless multi-robot systems controller over wireless networks using a specialized publish-subscribe message protocol. Although the research focuses on achieving an efficient throughput over a mobile ad hoc network and has limited scalability, it presents a practical application of SOA in robotics. Accomplishments in the paper include:

- Use of a service manager to control services on a single host
- Centralized configuration management with a configuration manager
- Robot locomotion details abstracted with motor control services
- Odometry/location estimation and sensor services
- Robot simulation service

Cucinotta et al. [124] propose RTSOA for industrial automation to enable building the automation system out of flexible autonomous components. The architecture allows for negotiation of the QoS requested by Web service clients and provides temporal encapsulation of individual activities. Supported APIs include a common API based on Web services and a non-standard custom API for real-time services. To provide guarantee for the service real-time properties, a priori analysis is used. The architecture has been demonstrated in a scenario in which Modbus messages are sent from a programmable logic controller (PLC) and translated into DPWS messages to be sent to the recipient of the original Modbus frame. Empirical evaluation was performed with object detection and image rotation in a video application. System reconfiguration capabilities were not evaluated.

Panahi et al. [132] present the RT-Llama RTSOA framework for real-time enterprise systems. The architecture includes a global resource manager for scheduling and orchestration of services in business processes. RT-Llama allows end-to-end deadline guarantees for business processes through advance reservation of resources.

Moussa et al. [119] propose a three-phase composition approach for Web services to address the issues of admission control and estimating communication latencies between services in the service composition phase. Admission control is needed since a service may be selected for workflow, but cannot cope sufficiently with the workload when deployed. Latency estimation provides an alternative to obtaining latencies during composition (which can be costly) or maintaining a full table of latencies for all service pairs. The three-phase approach consists of using an efficient algorithm to reduce the number of candidate compositions for a more accurate algorithm which selects the components for the final grounding phase. The work is based on using and extending Semantic Markup for Web Services (OWL-S) for the specification of service compositions and timing-related specifications.

3.5 Identified Gaps in the Existing Knowledge

Many of the challenges in the development of CPSs are related to managing the system complexity, size and number of interacting components. These factors drive development time and costs up, especially for safety-critical systems because proving that the system fulfils necessary requirements becomes more and more challenging. SOA has been identified as an approach that has potential to support building of CPSs correctly, affordably and flexibly. Although the dynamic nature of SOA may not seem to be suitable for safety-critical systems, there has been some research on applying the architectural style in safety-critical embedded systems. Rodriguez et al. have applied Web services in non-critical parts of UAV avionics [133] and mission execution [134], but the topic has still relatively little research available.

Research on fault-tolerant architectures is typically based on different forms of redundancy, such as checkpointing or active/passive redundancy, which do not solve the abovementioned issues. On the other hand, RTSOA can provide architectural basis for CPSs, but fault tolerance capabilities of SOA, especially in real-time applications, is a significant gap in existing research. As CPSs are typically composed of interacting nodes, the units of the system must be able to operate in heterogeneous computing environments, without relying on point-to-point connections. This enables, for example, the reconfiguration of the system and sharing of data to interested parties. However, to achieve this, research on CPSs faces challenges in configurability, ability to guarantee real-time performance and fault tolerance of real-time services. Effects of failover on real-time deadlines, consequences of faults for CPSs controlling complex physical processes with stochastic behaviour, fine-grained services in control system architecture, auto-scaling and optimization of (distributed) resources, guaranteeing schedulability and physical stability are some of the topics related to fault tolerance which need further research.

Common theme with the RTSOA frameworks is that they emphasize benefits of SOA as a tool to provide system reconfigurability and flexibility for achieving improved agility and more efficiency, typically in the form of factory floor level reconfiguration [124] [127] [135]. Although these evolvability-related aspects are some of the key drivers for RTSOA, their potential benefits for fault tolerance and resilience have not been explored.

Decoupling and modularity support fault tolerance without diverse redundancy, as discussed for example in 1985 by Gray [104]. Without modularity, a failure in any component implies a total system failure. Potential solution for fault tolerance could, therefore, be provided by architectures that promote loose coupling between software components and enable system to survive partial failures. The modular approach can be implemented, for example, with standardized component frameworks or service-orientation. BULKHEADS pattern [56] is an example of partitioning enterprise systems and allowing partial failures without losing the service provided by the whole system. However, there is not enough research about the actual effects of modular architecture on reliability in distributed real-time systems. SOA can be implemented without a central server, which would be a SPoF, having a negative impact on reliability. On the other hand, reliability estimation of SOA-based systems is difficult, because services might be used in unseen ways which complicates testing [136]. Evaluation of SOA overall impact on reliability needs, therefore, to be evaluated in real-time systems, including exploring how failures and restoring of services could be handled.

Existing research on time bounded reconfiguration in service-oriented systems has focused in resource management, latency estimation and choosing a solution from tentatively pre-scheduled solutions, as noted in the previous section. However, there is a knowledge gap in fault tolerance capabilities of RTSOAs, as noted earlier. In the event of a detected error, time-bounded operation of the system cannot necessarily be guaranteed, since the very definition of an error is that the external state of the system deviates from correct service state [2] (i.e., it is operating outside its specifications). The existing approaches presume the availability of an alternative service composition already in execution and the capability to react to the faults before they activate on the service or system levels. The research presented in papers [P4] and [P6] presents an alternative approach, evaluating the use of architectural properties of RTSOA for providing resilience for the system and using the services as a unit of fault tolerance, instead of service graphs or compositions. The proposed approach does not try to guarantee recovery within specific deadlines, leaving partial responsibility of detecting and reacting to (service) failures to the services themselves. Further insight could be gained by extending this research approach to service compositions.

There is also a need to bridge the gaps between the different models, methods and tools that are used to improve the design and the operation of dependable systems, especially when being adapted to control systems. Development of a dependability case in the context of systems engineering would seem a promising approach for combining fault prevention, removal, tolerance and forecasting methods to collect evidence for the case of a CPS. However, existing research on use of dependability cases is limited to a small number of case studies including an e-learning system [137], IP network assessment [138] and US Department of Defense projects [139]. For CPSs, the case could include architectural design choices (e.g., redundancy), requirements as a dependability argument about timing properties of the system and models (control model, formal models, etc.).

4 Main Test Case: Remote Handling Control System

4.1 RHCS in ITER

Remote handling has an important role in the maintenance and operation of the ITER plant since humans will not have access to the reactor and all maintenance tasks will be performed indirectly with remotely operated robotic manipulators and movers using man-in-the-loop teleoperation. In this respect ITER RHCSs have similar requirements to nuclear and space systems, needing either capability to recover equipment regardless of single failures or to provide adapted operability under failures. However, RH system computers are not located in the hostile environment, unlike Mars rovers or satellites, and do not have the fail-operate requirement. This is a significant difference, since there is no need for (diversely) redundant channels. Similarities to plant automation systems in general include the long expected operational period, since the RHCSs need to be designed with a minimum of operational lifetime of 30 years.

Development of the RHCS in a waterfall-like development process, as is typical for large-scale automation projects, can pose challenges for software integration. Some of the risks are related to the large number of control systems that need to be managed by the ITER Organization. The control systems should form a coherent and consistent system that is maintainable and usable by the RH operators. This was an issue in the early years of JET RH system development, as the development of control systems was left to external companies after the definition of the functional requirements [69]. The approach led to a situation in which every device had a custom-built control system with different architecture and user interface, complicating maintenance and operation. However, at low level the controls required in RH systems are similar to industrial applications. Therefore, widely available commercial products could be configured to handle motion control and integrated in a uniform control system framework. The current version of the control software (Mascot 4.5) was developed in-house in the late 90s using a new, unified approach based on shared high-level control software and use of COTS controllers [69].

For ITER RHCSs, development of a dependable and flexible control system architecture that can be integrated in the plant systems to control maintenance robots and equipment is essential. The plant system must integrate several systems provided by domestic agencies in-kind and developed by different suppliers. For example, Fusion for Energy (F4E) is the domestic agency of the European Union, responsible for securing the delivery of Divertor RH system, among others. ITER will feature several RH systems, all of which will have their own equipment controllers (ECs). In order to be economically viable, these controllers must be based on a common architecture that is able to deliver key features, such as configurability, maintainability during the long expected life-cycle, reliability and interoperability between different subsystems. The different RH equipment need to be operated flexibly from multiple work cells in the RH control room, which means that the subsystems must be capable of forming connections dynamically and in ad hoc manner when using the system, without needing an intervention by system administrators.

Erratic or uncontrolled movements have potential to damage the expensive and delicate components within the reactor, causing significant financial losses and maintenance period to exceed schedule. This may cause cascading delays and force changes to experimental operations, reducing operational time and lowering the plant availability, which is crucial for showing that ITER and magnetic fusion devices in general can be efficiently maintained. The ITER project needs to achieve a high level of dependability to succeed in its mission to produce the expected scientific data, meaning that it needs to be safe to operate and available for experiments as scheduled. Since a failure in a RHCS may cause or be a contributing factor in damage or loss of property, parts of the system fulfil the criteria for being safety or mission-critical.

Incorrect movement can be caused by software, especially on contact and near singularities. This could be a problem especially during the commissioning testing when people could be working near the equipment. Another typical cause for unexpected movement is incorrect parametrization of configuration, which can easily result from human error, parameter reset to default values, etc. Also a non-recoverable "no movement" failure can lead to a loss of the mission, forcing a rescue operation for the equipment. In such case, most likely cause would be a hardware failure. In the case of any failure, leaving the RH equipment in the reactor is not an option, since this would cause a critical or catastrophic unavailability or potentially violate ALARP principles which aim minimizing the risk of radioactive exposure for workers.

Against hazardous mechanical failures, system design should avoid SPoFs, although a casket-based rescue system is planned for use in recovery situations. However, rescue scenarios themselves increase demands for flexibility in the control system, since collaboration might be required, for example, between an RH system manipulator, Multi-Purpose Deployer and a rescue cask. Although ITER is a nuclear machine with need for a high availability, it is also an ambitious experimental device which will likely see several upgrades, emphasizing the need for evolvability.

ITER has a reference architecture for the RHCSs, provided as the ITER RHCS design handbook. This reference architecture is used as a baseline for implementing the RTSOA-based prototype control systems. IHA also has experience of developing software subsystems for RH operations based on the ITER reference architecture and some of these applications were used to demonstrate interoperability of the RTSOA in papers [P3], [P4] and [P6]. Therefore, a brief overview of the reference architecture is given in Appendix A.

4.2 Information Modelling and Interface Design

Service-based software development can be seen to emphasize interface design over component design, since the emphasis shifts from describing the components to designing on how the components interact with each other. In order to use the interfaces between the diverse applications and systems, a common information model must be shared, either implicitly or explicitly. Information models can be used to define the semantics of data that are exchanged over component interfaces, capturing the relevant concepts from the domain [140].

The information model is a conceptual, abstract model that should not depend on protocols or implementation details [141]. Information models specify relationships between objects with techniques ranging from natural language to formal structured languages, as in the partial information model for DTP2 shown in Figure 14 using Unified Modeling Language (UML) class diagram. The full information model is given in Appendix B.

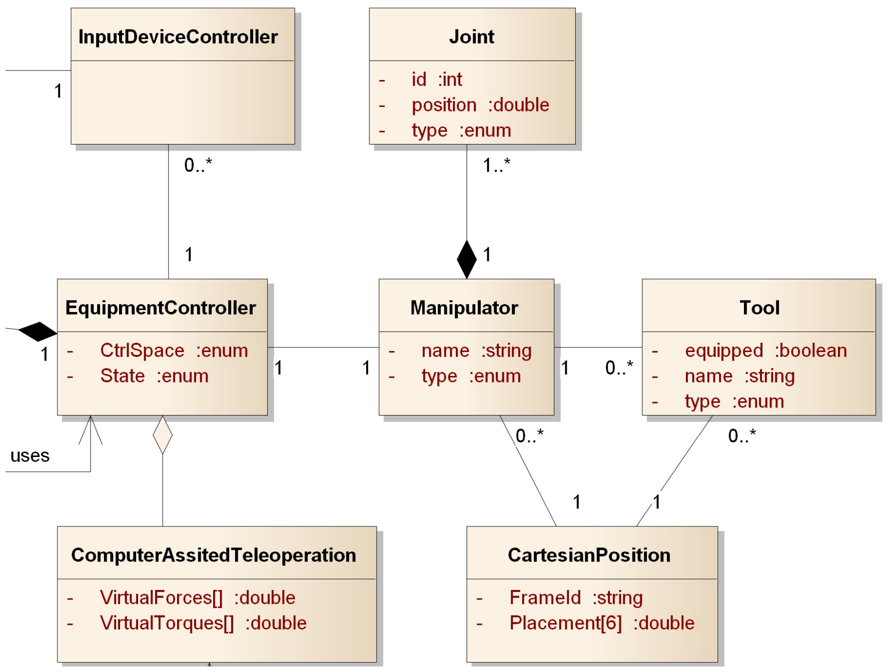


Figure 14. Part of the information model for DTP2

The data models, on the other hand, are used to describe implementation of the information model for different protocols and typically include implementation details. Since the conceptual models can be implemented in different ways, multiple data models can be derived from an information model [141]. A data model based on a well thought-out information model can be expected to have good consistency, enabling it to survive system modifications with minimal changes. The data model can decouple the system from hardware, supporting implementation of technical refresh cycles with less effort and cost, enabling to keep the system up-to-date.

An excerpt of the DTP2 information model translated to a data model, expressed with IDL, is given in Listing 1 on the following page. The example is used in the DDS-based GSB implementation in the prototype RH system. The IDL listing defines the structures and data types for the operations management system (OMS) commands topic used to send movement commands to EC from OMS; the corresponding objects in the information model from Figure 14 are EquipmentController, Manipulator and CartesianPosition. A minor disadvantage of a data model designed for general use is demonstrated by the example, as there are some data types that are not used in this prototype application. In the case of OMS commands the overhead is insignificant because of the low rate of commands being sent, whereas for more timing-critical topics it can be worthwhile to optimize the size and number of data types. The DTP2 IDL data model and the information model have been developed as part of the DTP2 research. The full IDL description is given in Appendix B.

Listing 1. Example of defining a structure for OpenSplice DDS IDL Pre-processor

```
struct OmsCommand
{
      long cmd_id;
      long cmd_type;
      long object_id;
      long parent_id;
      Location3D location;         // x, y, z
      Orientation3D orientation;  // roll, pitch, yaw
      boolean flags[5];
      string msg;
};
#pragma keylist OmsCommand cmd_id
```

For a RHCS, most important communication needs include sending motion commands, configuration data, status updates, position and sensor data, virtual guiding forces, diagnostics data, events and alarms. This translates to large amounts of data being transferred, some of it cyclic and some acyclic. Especially VR, computer assisted teleoperation (CAT) and input device require real-time position data from the equipment. Asynchronous communication is especially suited for building distributed control systems, which need to react to external stimuli and events [142, p. 28]. Using synchronous RPCs for sending of commands might be more straightforward to implement compared to designing commands as part of the data-model, sending them as messages and then monitoring updates to published status updates. However, RPC is a tightly coupled technology, reducing the benefits from the loosely coupled data-centric approach. The location transparency of blocking RPCs can also lead the programmer to neglect the performance impact of the remote calls, resulting in unresponsive applications.

The requirement to be able to control more than one type of RH device adds to the complexity of the system design, especially if all detailed requirements for these devices are not fully known at the design time [143], as is the case with ITER RH systems. A well-designed information model can be used to facilitate deployment of new RH devices as the new equipment can reuse existing implementations and provide explicit extensions and adaptations by modifying data models derived from the information model.

5 Results

This section summarizes the results in the papers [P1]-[P6] and shows how they are related to the overall view of building dependable CPSs by combining the contributions to a top-down systems engineering approach which considers issues related to identifying and managing of design information relevant to the dependability of the architecture, including failure modes, configuration management, identified critical requirements/subsystems, architectural decisions and so on. While the systems engineering approach is crucial for managing dependability systematically throughout the system lifecycle, the main contributions of this thesis are for service-based software architecture and fault tolerance of CPSs. For the software architecture and its prototype implementation, a bottom-up approach to dependability based on evaluating fault tolerance capabilities of the implementation is presented.

Section 5.1 outlines the systems engineering approach for building dependable CPSs, originally proposed in [P1]. This includes the analysis of dependability-related requirements for RH systems [P2] and fault taxonomies for RH systems, developed during a two-month visit to JET at Culham, UK. The approach proposes combining the development elements affecting system dependability to a dependability case. Section 5.2 describes the fault-tolerant RTSOA developed and implemented within the GOT-RH systems engineering framework (Figure 6 on p. 15). Results of the prototype design and implementation include publications based on the concept study [P3], more refined implementation for fault detection and recovery in [P4] and the design description [P6]. Architectural insight gained from the concept design was published as patterns in [P5]. Finally, Section 5.3 presents a paper-by-paper summary of the results.

5.1 Systems Engineering Approach to Dependability

5.1.1 Development of a Dependability Case

Goal of the research, as stated in Section 1.3, was to use the top-down GOT-RH systems engineering framework to develop architecture and methods for a fault-tolerant ITER RH equipment control system. This section describes how the results of the research fit to the general systems engineering practices. One result of the research is analysing how the RAMI and software dependability approaches can be combined with systems engineering using fault taxonomies, probabilistic methods, qualitative methods, architectural design decisions, component reuse, COTS components, etc. to collect evidence for the arguments in a dependability case.

Major components of system development that impact dependability include:

- Requirements (fault prevention)
- Architectural design, use of patterns (fault tolerance)
- Components and services proven in use
- Analyses and operational data (fault forecasting)
- Failure modes and effects (domain-specific)
- Verification and validation (fault removal)
- Development process & methods (fault prevention and removal)

Combining these design artefacts as requirements, arguments and evidence to develop a dependability case is demonstrated in the context of a systems engineering process model in Figure 15. The process model is based loosely on the V-model (cf. Figure 8), showing the integration of additional design elements to the condensed "requirements → design → implementation → testing → validation" development process. Development process, design, tools and techniques (e.g., VR-based verification) have significant impact on the dependability and should be part of the documentation in the dependability case; this aspect is present through the system development and verification plans if the figure below. The building of a dependability case consists of dependability analysis, dependability goal definition, argument construction and evidence collection [144], marked with dark grey in the figure.

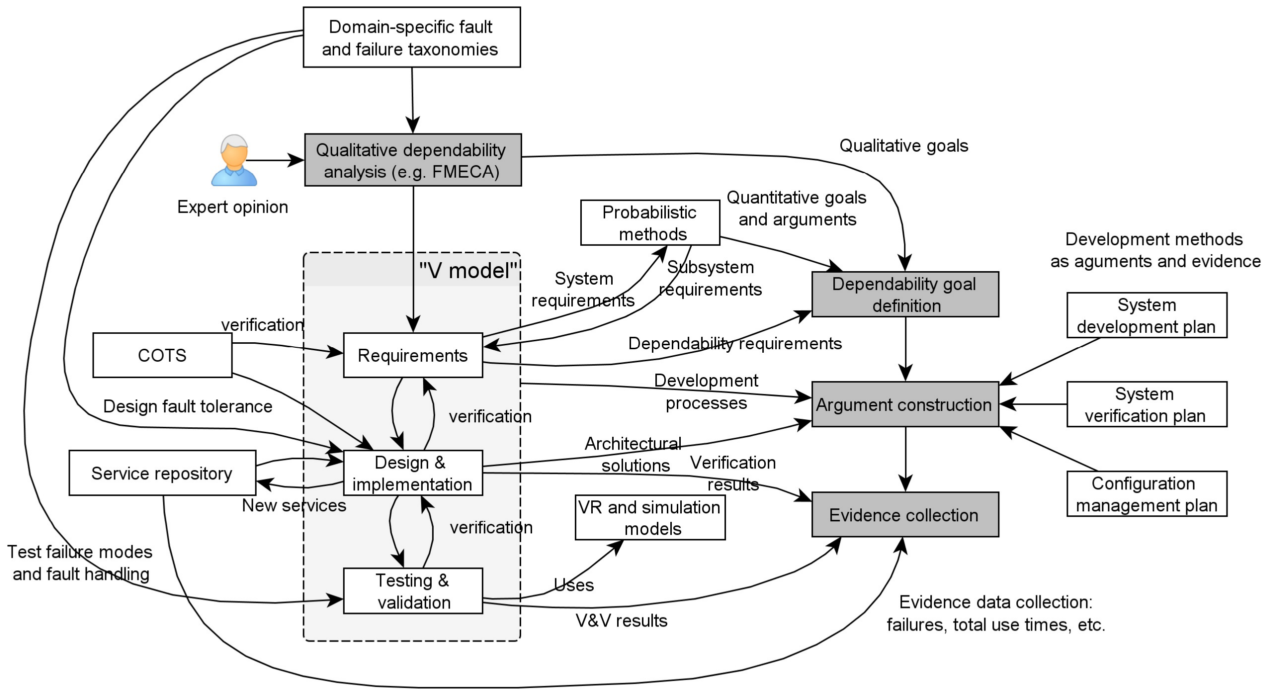


Figure 15. Process model for building the dependability case in the context of the systems engineering V-model and dependability methods

The components of the model include:

- Fault taxonomies (**fault forecasting**) can be used to support failure analyses and design of fault tolerance solutions (see Section 5.1.2).
- Qualitative approach to fault prediction (**fault forecasting**), such as hazard analysis and FMECA, are used in systems engineering and in the ITER RAMI process, thereby providing a possibility to integrate software development processes to the overall systems engineering based dependability processes, since FMEA/FMECA can also be applied to software [145].
- Probabilistic methods (**fault forecasting**) such as reliability block diagrams, fault trees, etc. if needed to evaluate reliability or availability.
- Role of the architecture and platform (**fault tolerance**): architectural design decisions, trade-off point decisions, architectural solutions (patterns, interfaces, etc.)

- Use of COTS solutions need to be considered not only against the system requirements, but also against life-cycle costs and especially maintenance. COTS or open-source components may require adaptation (e.g., custom-patched kernel [62]) or device drivers for the operating system that need to be maintained.
- Repository of existing services and components with explicit dependability claims, including intended use conditions.
- Verification (**fault removal**) based on empirical testing using simulated and/or virtual models and taxonomies. The service-oriented approach supports this goal, since the services interfacing with the real (physical) devices and plants can straightforwardly be replaced with simulated ones. Simulated plants and virtual models can, therefore, be dynamically connected to the service deployment being assessed.
- Used development, verification and configuration management methods (**fault prevention**).

For building the dependability case, the dependability requirements need to be identified; following aspects of control system software that affect these requirements were identified originally in [P2]:

- dependability objectives—MTBF, hazard analysis
- operation modes (level of autonomousness: manual-supervisory-automatic, open vs. closed loop, limp home, etc.)
- timing requirements: latencies, execution deadlines, scheduling
- fault tolerance and responses to undesired events (e.g., fail-safe behaviour)
- identification of safety-critical functions and subsystems—all software running physically on the same platform as safety-critical software is also considered safety-critical

Additionally, the following aspects were identified later during the research process: operational profiles, security requirements, data quality, variability, resource usage (bandwidth, CPU, memory, power, etc. [P4]) and resilience requirements for CPS [P6].

Operational profile is a quantitative representation of how software will be used (e.g., by assigning occurrence probabilities for system modes and functions [146]). The operational profile can then be used to support reliability engineering by focusing testing efforts on the most used operations.

Security: As a downside for increased connectivity of CPSs, attack surface of a system is also increased, setting a need to maintain a level of cyber security during the intended lifecycle of the system. This is in contrast with the commonly seen approaches of relying on security through obscurity and separation of critical networks in automation projects. Automation systems have had backdoors, non-encrypted network traffic, etc. even in safety-critical systems, such as Supervisory Control and Data Acquisition [147] or traffic light control [148]. Designing the system to fail into a safe state is of little use if an adversary can reconfigure the system or use this for denial of service attacks. Using common off-the-shelf components and operating systems means that the current approaches to security become even less sufficient, since the systems will have to incorporate processes and means for security updates against common vulnerabilities, such as Shellshock or Heartbleed, especially if the devices are to be connected to IP infrastructure. Any

CPS must, therefore, include security architecture to minimize security risks and ensure system integrity.

Data quality: sufficient precision and temporal correctness must be ensured to guarantee that the system operates within specifications. Implementation may require setting and monitoring QoS parameters, calibration of measurements, redundant sensors, etc.

Variability and expected types of change: variability points should be identified beforehand, in order to accommodate changes in the system architecture, including potential maintenance needs (cf. 1st and 2nd laws of Lehman). On the other hand, resilience and loosely coupled architecture support these goals on the architectural level, as services can be used as variability points.

Figure 16 shows an example of a quantitative argument that could be used to support the claimed level of dependability using GSN [97] for a critical service in the RHCS. By focusing the assurance efforts for critical services, dependability argument can be built over service dependability and focus testing on critical services for evidence.

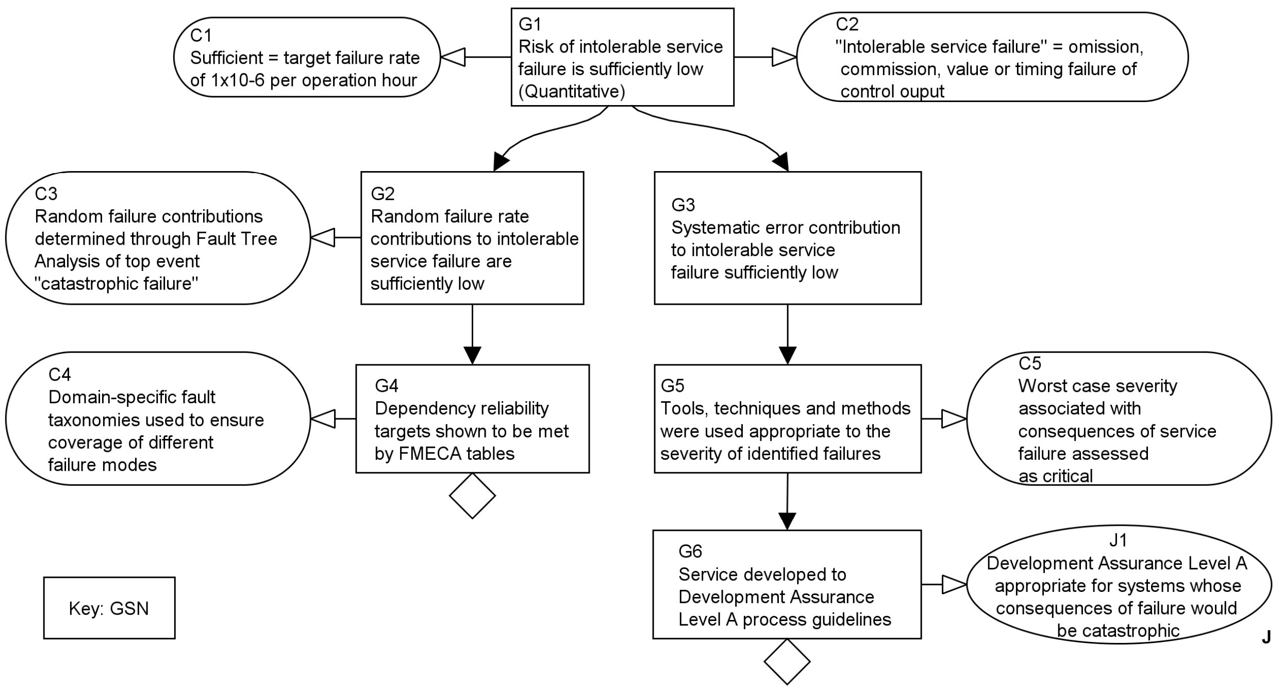

C1
Sufficient = target failure rate of 1x10-6 per operation hour
G1
Risk of intolerable service failure is sufficiently low (Quantitative)
C2
"Intolerable service failure" = omission, commission, value or timing failure of control ouput
C3
Random failure contributions determined through Fault Tree Analysis of top event "catastrophic failure"
G2
Random failure rate contributions to intolerable service failure are sufficiently low
G3
Systematic error contribution to intolerable service failure sufficiently low
C4
Domain-specific fault taxonomies used to ensure coverage of different failure modes
G4
Dependency reliability targets shown to be met by FMECA tables
G5
Tools, techniques and methods were used appropriate to the severity of identified failures
C5
Worst case severity associated with consequences of service failure assessed as critical
G6
Service developed to Development Assurance Level A process guidelines
J1
Development Assurance Level A appropriate for systems whose consequences of failure would be catastrophic
J
Key: GSN

Figure 16. Dependability argument example adapted for RTSOA from [97]

5.1.2 Domain-Specific Fault and Failure Taxonomies for Remote Handling

Domain-specific fault taxonomies have potential applications in the implementation of fault tolerance, since it is typically a very domain specific subject and depends on understating failure mechanisms, fault propagation and consequences of faults. Fault taxonomies can be useful in documenting this kind of knowledge. Currently there are no RH-specific fault taxonomies available. RH fault taxonomy could be used to develop methods for fault detection, isolation and mitigation specific to RH domain [149]. Another significant use for fault taxonomies is related to verification & validation of the system since testing is one of the key techniques for finding faults, regardless of its limitations. Fault taxonomy could be used in fault analysis methods (e.g., FMECA) and as an aid for designing software requirements, fault tolerance, test suites and analysing coverage of aforementioned items.

To gain better understanding of software faults and their effects in control systems, a two-month visit to JET in Culham, UK was carried out as part of the GOT-RH project. During this researcher visit, the control system architecture and operational logbooks of the JET RH systems were studied. Based on the data obtained during this visit, the fault taxonomies presented in this section were developed. Regarding the potential use of the taxonomies, it should be noted that no complete coverage of failure modes or fault types is claimed by this thesis.

There are various possible approaches to categorizing faults (see e.g. [2]). In RH systems, these approaches could include:

- Failure consequences: unwanted movement, fail-safe tripped, incorrect measurement values, incorrect value for gain, etc.
- Failure causes: operator error, developer error, complex timing error, aging fault, faults caused by incorrect assembly/manufacturing/maintenance, mechanical wear, etc.
- (Software) fault types: aging related faults, bohrbugs, mandelbugs, heisenbugs, byzantine failures, etc.
- Level of detail: system, division, unit, processor, module, component, etc.
- Software vs. hardware, plant vs. controller, operator station faults vs. server-side faults, etc.

We chose to focus in three areas that were considered to be most interesting from the CPS point of view, namely system failures, controller faults and software faults:

- *System failures* cover the failure modes of the system that are related to remote handling systems (i.e., situations in which a fault causes a disruption in the service provided by the system, such as movement and operation of the manipulator and tools). The consequences of a system failure may cause downtime and/or damage. The fault that has caused the failure may have occurred in hardware or software, and all faults could be permanent/transient, internal/external, etc. as categorized in [2].
- *Controller faults*: hardware (mechanical and electrical components), control theory, environment.
- *Software faults* focus specifically in RH control systems. Assumptions: system does not use multi-version redundancy, since it is not a protection/safety-related system, so no voters are needed. Controller hardware failures are not covered, except for timing and communications, where hardware failure may trigger software faults if fault handling is not correctly implemented.

For example, fault injection used in [P6] to kill the IDCom service would be classified as a "dependency not available" software fault and could cause system failure "incorrect movement" if not handled properly, instead of just "failure to move".

Typically mechanical and hardware faults are covered well by FMECA and similar bottom-up analysis methods used for fault prediction. Therefore, they were left outside of the scope. In order to track relationships between faults, taxonomies use a notation in which faults can be marked as being related to each other or a fault causing another one.

System Failures

Failure is an event in which the service delivered by the system deviates from the correct service, either because it does not comply with the functional specification or because the specification did not adequately describe the system function [2]. System failures can be used to evaluate system failure modes. We have grouped system failures into three main categories that are expanded in Figure 17:

- Failures to move/operate manipulator at all.
- Failures caused by inadequate performance. System can be used, but operations are slowed down and/or risk level is increased.
- Failures caused by incorrect movements/operations. Use of system might be possible in some cases, but at a very high risk level.

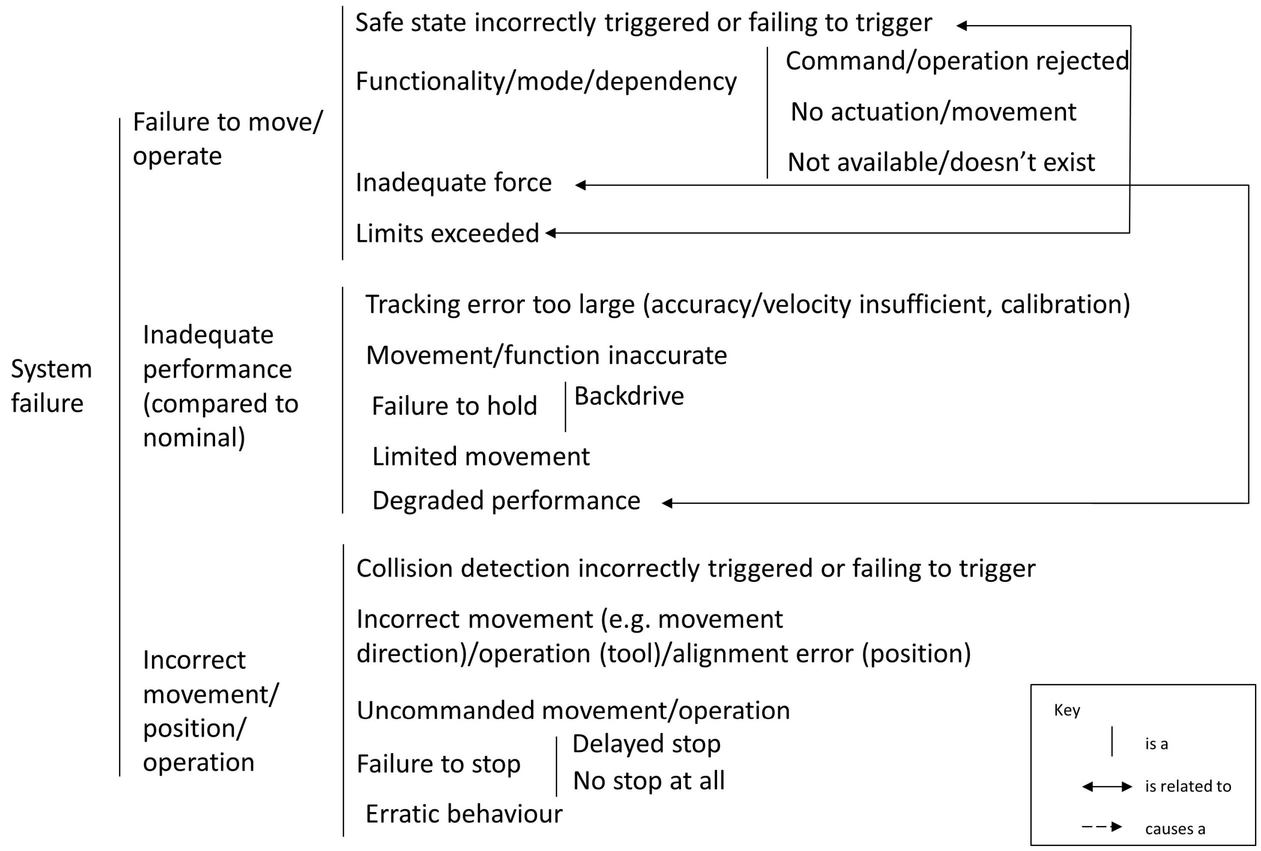

System failure
Failure to move/ operate
Safe state incorrectly triggered or failing to trigger
Functionality/mode/dependency
Command/operation rejected
No actuation/movement
Not available/doesn't exist
Inadequate force
Limits exceeded
Inadequate performance (compared to nominal)
Tracking error too large (accuracy/velocity insufficient, calibration)
Movement/function inaccurate
Failure to hold
Backdrive
Limited movement
Degraded performance
Incorrect movement/ position/ operation
Collision detection incorrectly triggered or failing to trigger
Incorrect movement (e.g. movement direction)/operation (tool)/alignment error (position)
Uncommanded movement/operation
Failure to stop
Delayed stop
No stop at all
Erratic behaviour
Key
is a
is related to
causes a

Figure 17. Remote handling system failure taxonomy

Control System Faults

The cause of an error is a fault [2]. Control system specific faults have been divided into four categories which are expanded in Figure 18 on the following page:

- Hardware faults: problems with the hardware that the control system relies on.
- Operator function faults that lead to system failures (see previous section).
- Control faults: specifically related to control algorithms and their tuning.
- Disturbances: inability to take possible disturbances into account or operating in non-nominal environment (e.g., temperature).

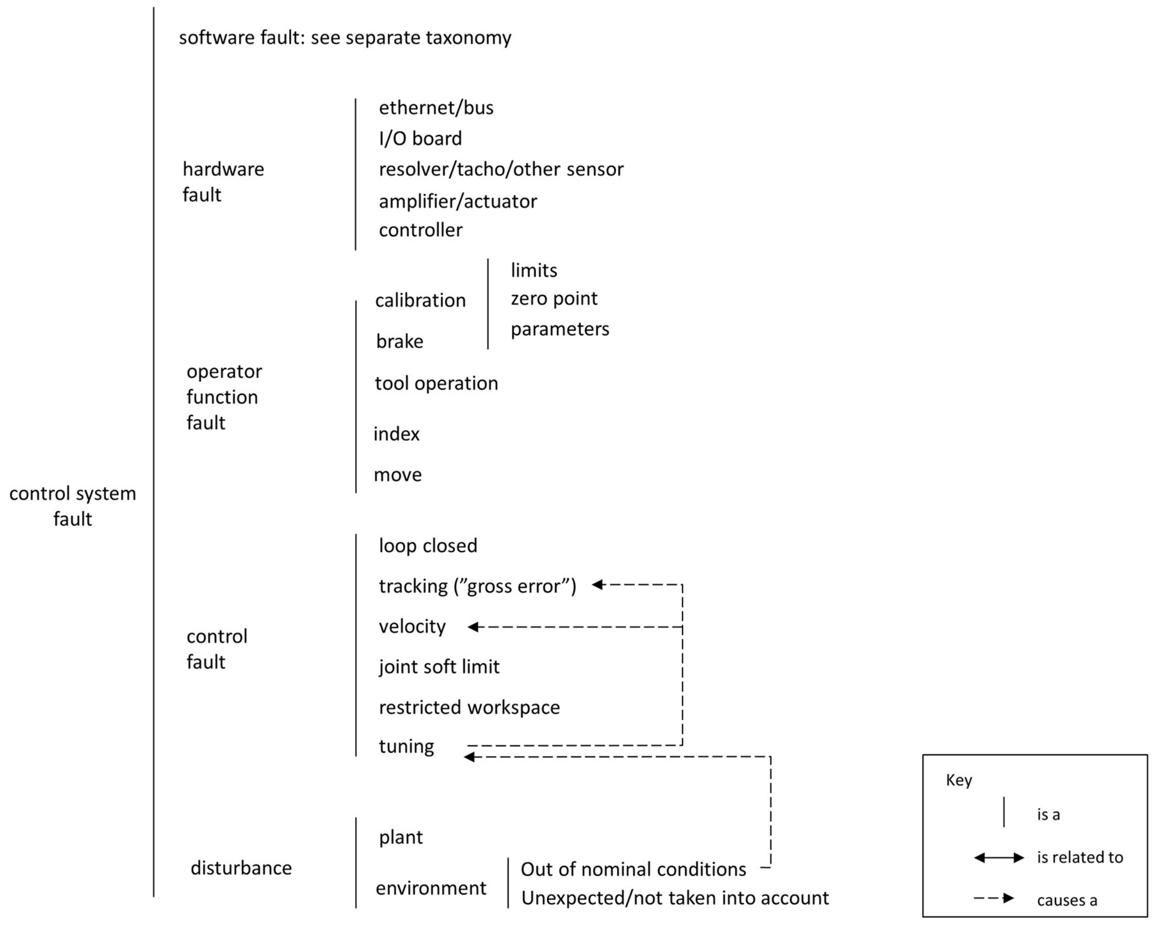


Figure 18. Remote handling control system fault taxonomy

In JET, controller failures (e.g., in Power PMAC motion controllers) were frequently solved with controller restart or, in severe cases, a factory reset and reloading of the configuration file. Highly decoupled system design enables application of this approach automatically and more granularly if a node or service has crashed. Crashes can be detected, for example, with HEARTBEAT [11]. Module/node can be restarted with SYSTEM MONITOR [11] or SERVICE MANAGER [P5].

Software Faults

The range of possible software faults is considerable, so the taxonomy focuses in faults that are most relevant or typical for RH systems (e.g., categories for configuration and command faults). Software faults are especially interesting in the sense that they can be a potential source of common-cause failures (failure of two or more systems/components due to a single cause), e.g. due to a fault in the shared software platform [150]. The taxonomy for remote handling software faults is presented in Figure 19 on the following page.

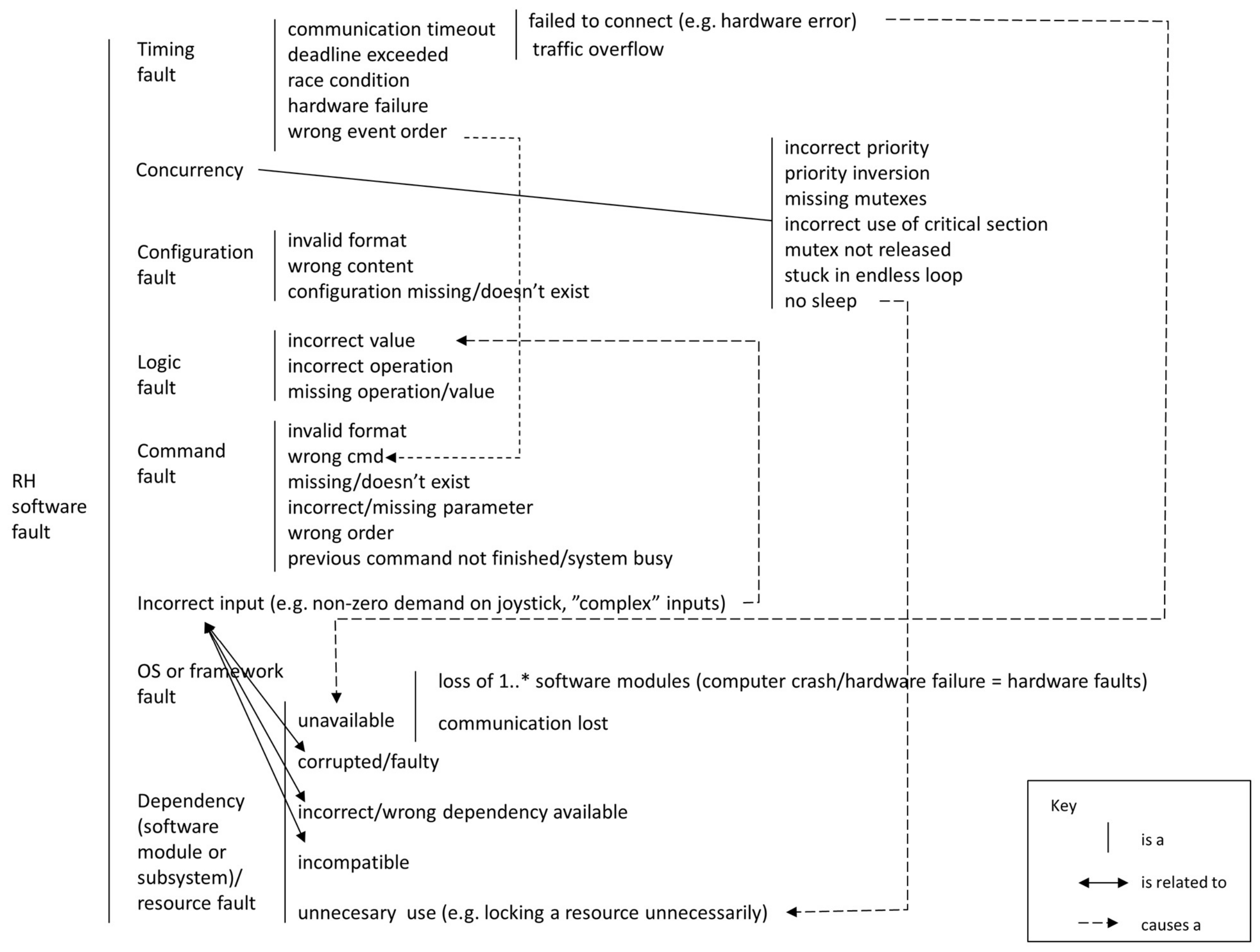


Figure 19. Remote handling software fault taxonomy

As an example of plausible software fault, parameters could be lost and recovered to default configuration values. This kind of fault could cause incorrect movement failure and can be hard to notice beforehand, unless taken into account in the design, for example, by using invalid or zero values as defaults.

Software faults were not a major problem in the current JET RH system compared to downtime caused by mechanical failures, although exact rate of software failures cannot be accurately estimated as root cause analysis was not included in most of the logged failures. Dominance of hardware failures is to be expected, since the current system has been in use for almost 20 years. For example, based on the analysed logs, failure entries containing the string "arm crashed" have gone from having over 100 instances in 1996-1999 to 3 instances in 2000-2013. This is a typical failure in which the failure is reasonably likely caused by software. Naturally system updates could introduce new faults and change the failure rate.

An Example of Using the RH Fault Taxonomies

The taxonomies can be used to identify critical components that need dependability assurance. For example, incorrect movement is a severe system failure that can be caused by a fault in the C4G[3]

[3] C4G is the control unit of the Comau Smart NM45-2.0 industrial robot used in this thesis. C4G service is responsible for abstracting the interface for the control unit.

service, as shown in Table 1. Furthermore, when analysing failure modes of a critical component, such as the equipment controller or C4G service, the taxonomy can be used to identify susceptibility of the component to specific faults and coverage of fault tolerance.

Table 1. Failure analysis with taxonomies

Unit	C4G service
Failure mode	Incorrect movement
Possible causes	Logic fault, incorrect input, configuration fault, command fault
Detect with	Human operator, VR collision detection, soft limits exceeded.
Action needed to take	Fail-safe (emergency stop)
Mitigation (arguments for the dependability case)	Extensive testing, input assertion, use of invalid default values, configuration validation with VR models, periodic command validity checks, no caching of commands

5.2 Architecture and Prototype Implementation

The architecture described in the included publications builds on the Sulava RTSOA for embedded heterogeneous machine control, introduced in a short paper [39] and demonstrated for hydraulic boom control using a single box configuration in [38]. Sulava is a high level reference architecture, originally designed for mobile machines. Key features of the architecture include the use of P2P networking, microservices running as Xenomai tasks and communicating with local (LSB) and global communication buses (GSB) that are implemented with Xenomai queues, shared memory and DDS middleware. The Sulava architecture is used as a reference architecture for the concept study design, implemented as an automatic teleoperation system based on real-time services for use in ITER RH scenarios, as shown in Figure 20.

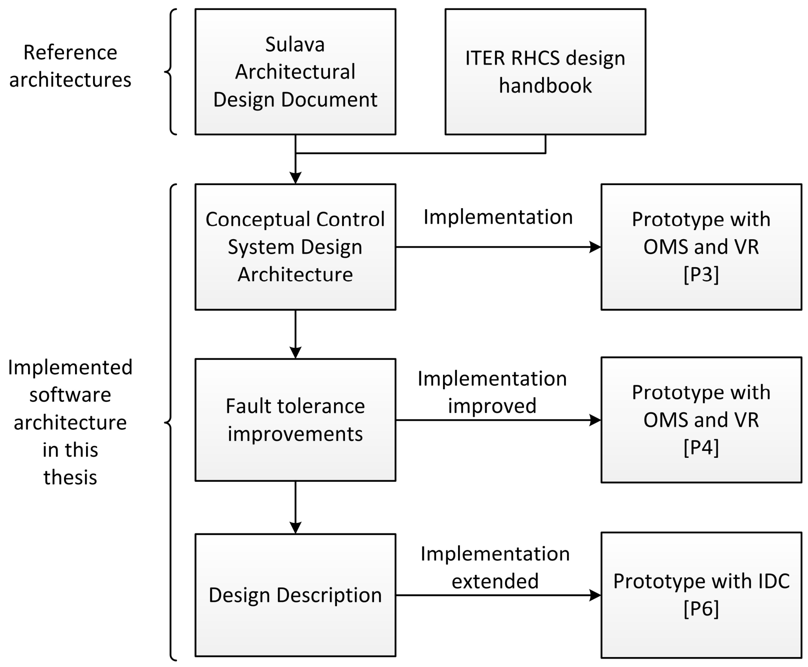

Reference architectures
Sulava Architectural Design Document
ITER RHCS design handbook
Implemented software architecture in this thesis
Conceptual Control System Design Architecture
Implementation
Prototype with OMS and VR [P3]
Fault tolerance improvements
Implementation improved
Prototype with OMS and VR [P4]
Design Description
Implementation extended
Prototype with IDC [P6]

Figure 20. Role of reference architectures for the implemented architecture

The concept design was extended with service management to provide fault detection and recovery. The detailed design (final design in the research methodology, see Section 1.4) focused especially on adding and implementing fault tolerance features. For the detailed design, input device controller (IDC) was also implemented to provide bilateral teleoperation mode for manual operation. The architecture prototypes were evaluated after each phase using deployment to distributed and heterogeneous environment, which was not part of the original Sulava architecture research.

5.2.1 Concept Design and Target Platform

The concept design based on the Sulava RTSOA reference architecture was planned and implemented as a prototype to evaluate benefits of service-orientation in CPSs. This was carried out using automatic operations with OMS and VR as described in the publication [P3]. The implementation used the 6 DOF Comau Smart NM45-2.0 industrial robot as a test platform, which was also used in the papers [P4] and [P6]. The robot is equipped with an open control system option "OpenC4G" which enables an external PC to participate in the low-level control of the manipulator joints, essentially enabling PC-based control from outside of the manipulator's own C4G control system. OpenC4G comes with a library for RTAI Linux, which was ported to Xenomai in order to use it on the standard platform of the RTSOA. The OpenC4G extension supports several modes of operation, in which the PC provides position, velocity and/or current contributions to affect target position and velocity control loops in the joint space. Position and velocity control mode of joints was used, in which the PC provides the absolute target position for each joint.

OMS is used to provide Cartesian target position and orientation for the robot end-effector in the operational space. In order to move the manipulator to the target position, the inverse kinematics problem must be solved in order to calculate the corresponding joint space values [151]. OpenC4G control modes, Denavit-Hartenberg parameters, forward and inverse kinematics for the manipulator are given in Appendix C.

Inverse kinematics are calculated in the TrajectoryGenerator service, which generates the desired position x_{ref} and trapezoidal velocity profiles $\dot{x}_{\text{ref}}$ for each joint according to the specified acceleration profiles, based on the movement command received from OMS through the GSB. Position and velocity reference values are provided constantly in real-time to the C4G service through the LSB as the TrajectoryGenerator runs through the trajectories. Trajectory generation and reading of trajectory commands from GSB are separated into two services (OmsCom and TrajectoryGenerator) in order to ensure real-time performance of the trajectory generation. Finally, position of the manipulator is updated constantly to the GSB for the subscribed nodes—such as the VR system which is used to visualize the position of the robot—through another GSB communication service C4GJointDataPublish. A full deployment diagram of the services is provided in the following section.

As noted in the beginning of Section 5.2, starting point for the architecture development was the Sulava RTSOA reference architecture, which was further developed and implemented for the RHCS, focusing especially on the fault tolerance features of the architecture. OmsCom, C4GJointDataPublish, VR subsystem (including IHA3D integration with a DDS plugin) and OMS subsystem were implemented by the IHA personnel, whereas concept design, implementation of the Service Manager and C4G service and the adaptation of TrajectoryGenerator service were carried out by the candidate.

Figure 21 shows the class diagram for the C4GJointDataPub service to demonstrate the practical implementation of the service concept. The service class is derived from the abstract base class EskoaService and uses the JointDataGSB class to implement GSB communication. In the current implementation, the logic related to the service manager is included in the service main function. Services are implemented as autonomous processes which can be executed as applications or forked by a parent process, as in the case of service manager. Typical service sizes in the implemented RH system were approximately 1000-1500 lines of C++ code, which includes boilerplate code for encapsulating DDS, basic service logic, state management, etc., but not auto-generated DDS code or other libraries.

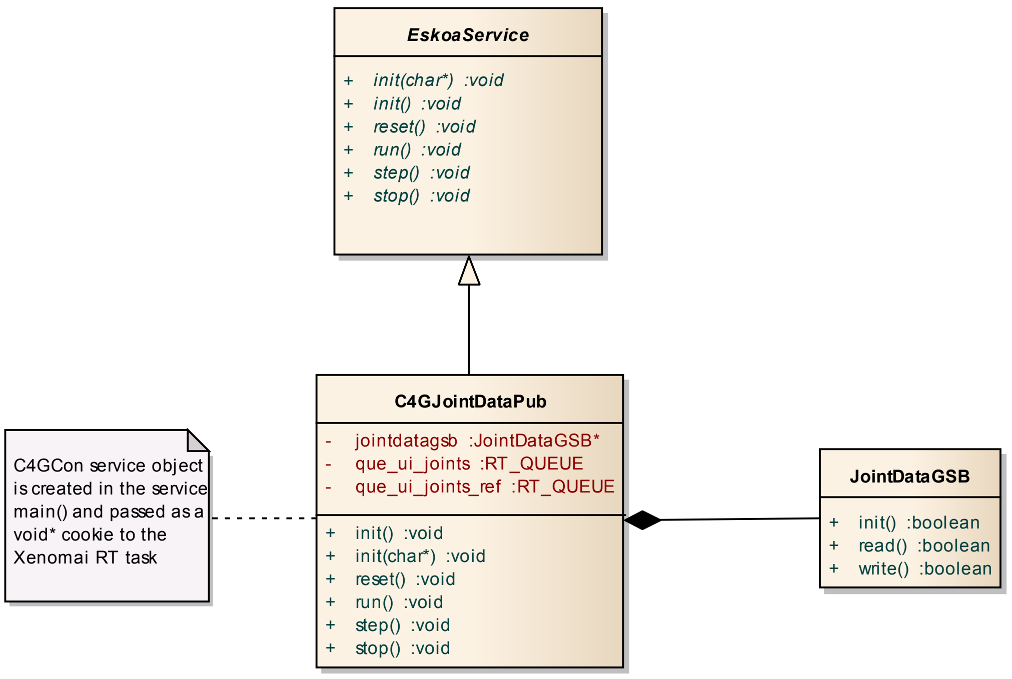


Figure 21. Class diagram for the C4GJointDataPub service (key: UML)

Based on the analysis of the quality attributes of interoperability, evolvability (focusing on changeability, extensibility and portability), real-time performance, fault tolerance, cost efficiency and ease of development, the implementation successfully demonstrated integration of heterogeneous subsystems with the service-based control system. Patterns DATA-CENTRIC ARCHITECTURE, SERVICE MANAGER and LET IT CRASH were published first as a conference paper and later in [P5], based on the development of the concept design. The latter publication includes the pattern language for fault tolerance for CPSs, showing one possible way to combine these patterns with other fault tolerance patterns to support the development of dependable CPS designs. Short descriptions of patterns are provided in Table 2 on the following page.

Table 2. Patterns and descriptions

Pattern	Description
DATA-CENTRIC ARCHITECTURE	How to implement a reliable and scalable distributed control system? Build the system from autonomous modules that communicate by sharing data that is based on a well-designed and consistent data model.
SERVICE MANAGER	How to detect faults and restart modules or processes after a failure? Implement a service manager that can monitor, start and stop the modules.
LET IT CRASH	How to react to failures without crashing the whole system? Terminate the process if it has failed in order to detect failures quickly, instead of trying to handle the fault. Processes must be designed to tolerate failures of other services. Terminated process can be recovered, for example, by automatically restarting it.

5.2.2 Control System Architecture Design Description and Verification

Based on the promising results of the concept design, fault tolerance capabilities (especially the service manager, heartbeats, fault detection and fault recovery) were refined for the next deliverable of the GOT-RH systems engineering process, the control system architecture design description (detailed design). Improved fault tolerance capabilities were analysed in [P4]. To provide a more thorough estimation of overhead, scalability, fault detection and recovery capabilities of the service-based approach, the prototype RH system was extended to support bilateral teleoperation with a 6 DOF haptic device Phantom Omni. The results are presented in [P6], which also includes a more detailed description of the architecture. This section presents evaluation of the applied fault tolerance methods and their effects, and levels of safety, service-orientation, real-time performance and computing overhead achieved.

Position-position bilateral teleoperation architecture was chosen to provide haptic feedback for the operator. Position of the master device (Phantom Omni) is used to set the target position for the slave device (Comau) and the position of the slave device is used to generate force feedback for the master device (i.e., manipulators track the positions of each other). Position-position control was chosen because it is straightforward to implement and requires no additional expensive force sensors, although it has limited transparency (sensing of the remote environment) [152]. Because of the large size difference between the manipulators, forces and positions are heavily scaled. Furthermore, Phantom Omni only has 3 DOF force feedback, providing no torque for the wrist. Although these limitations affect usability of the manipulator in delicate manipulations, the setup nevertheless provides a demanding environment for testing of the fault tolerance capabilities. Extending the system to support more advanced bilateral teleoperation architectures is fairly straightforward, for example, by installing a force sensor to the slave manipulator's end-effector and publishing force measurements to the GSB.

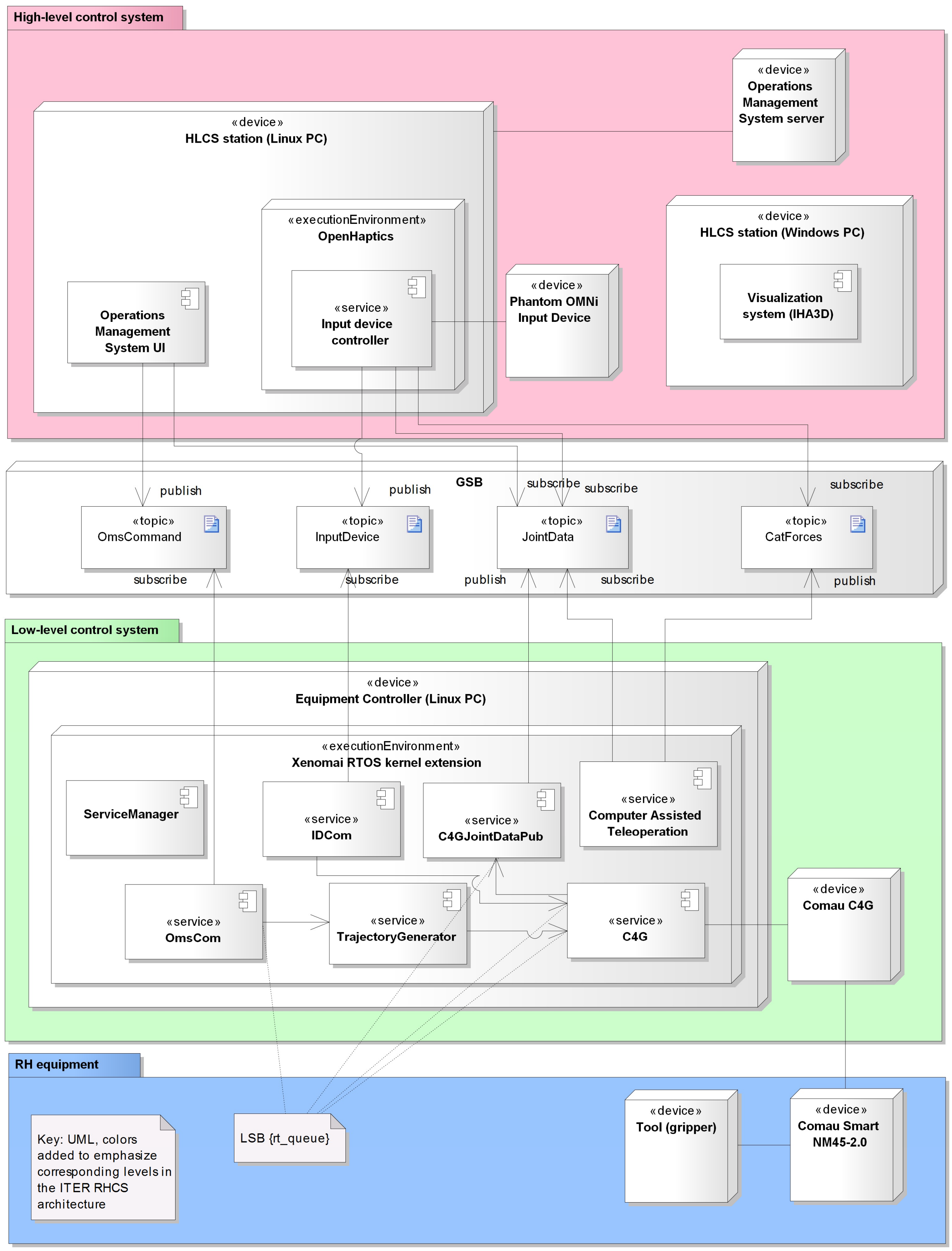

High-level control system
«device» HLCS station (Linux PC)
«device» Operations Management System server
«executionEnvironment» OpenHaptics
«service» Input device controller
«device» Phantom OMNi Input Device
«device» HLCS station (Windows PC)
Visualization system (IHA3D)
Operations Management System UI
GSB
publish
publish
subscribe
subscribe
subscribe
«topic» OmsCommand
«topic» InputDevice
«topic» JointData
«topic» CatForces
subscribe
subscribe
publish
subscribe
publish
Low-level control system
«device» Equipment Controller (Linux PC)
«executionEnvironment» Xenomai RTOS kernel extension
ServiceManager
«service» IDCom
«service» C4GJointDataPub
«service» Computer Assisted Teleoperation
«service» OmsCom
«service» TrajectoryGenerator
«service» C4G
«device» Comau C4G
RH equipment
Key: UML, colors added to emphasize corresponding levels in the ITER RHCS architecture
LSB {rt_queue}
«device» Tool (gripper)
«device» Comau Smart NM45-2.0

Figure 22. Full service deployment diagram (cf. Figure 23 in Appendix A)

Figure 22 above shows the complete service deployment view with both OMS and IDC deployed at the same time. The figure also shows the principle for integration of the CAT subsystem as a service to demonstrate extensibility of the platform. IDCom and InputDeviceController services were implemented by the author—previous services were re-used without needing further modifications, besides the improvements to the fault tolerance-related functionality.

Results published in [P6] indicate that RTSOA is suitable for implementing CPSs by providing sensors, actuators, state estimation and control processing as services. The architecture supports extension of the system by introducing, for example, force sensors or new input devices (master arms or joysticks) in a straightforward manner, as they could be integrated with the system directly through existing interfaces (i.e., GSB topics and LSB queues).

Focus of the development was on the fault tolerance aspects, including analysis of how services should react in the case of failures—e.g., not receiving data inside the deadline—without cascading failures, but still triggering fail-safe functionality correctly. This was implemented so that services move to a fail-safe waiting state (starting state in [P6]) if all dependencies are not available, whereas a separate error state is used in the case of errors (and which may be used to terminate service in the case of let it crash error handling), as described in [P6]. Therefore, the architecture can be considered to use fail-stop approach [153] on the service level, while recovery is done on best-effort basis.

Best-effort reconfiguration differs from the typical reconfiguration algorithms used in real-time systems which aim to provide deterministic reconfiguration capabilities, such as those covered in Section 3.4.2. Best-effort approach is used because deterministic behaviour cannot be guaranteed in service failure situations and also because of verification of service real-time properties before deployment for heterogeneous and dynamic CPSs is challenging. In best-effort recovery, service failure scenario actually uses the same logic when the system is initialized or a service is upgraded on-the-fly. Other services react to the missing service as a normal failure situation (dependency not available) and resume operation after the upgraded or recovered service is running and available. As an example, if a communication service detects a fault and moves to the error state, the service manager chooses what action to take based on the escalation level and, for example, restarts the service. As shown in [P4], if the service restarts before any deadlines are exceeded, other services do not need to take any action. On the other hand, if any deadlines are exceeded, other services will move to the starting state to wait for their dependencies.

The fault handling approach had two primary goals, to avoid failure on the system level and to improve mean time to repair (MTTR) by rapid recovery in order to maintain system level availability. Methods applied to achieve these goals included automatic restarts and escalation, configurable through the service manager. Fault handling is escalated after a specified number of faults have been triggered (detected using e.g. missing heartbeats) through fault handling levels of soft and hard service restarts, switching to a backup service and finally a complete removal from execution. The approach of recovering faults with restarts aims to restart only the necessary part of the system to reduce impact on the availability of the system. As the restarts are applied on the service level, impact depends directly from the service granularity.

Effects of the employed fault tolerance methods are evaluated in the Table 3. From the dependability-related attributes, reliability and availability can be improved with fault detection and recovery mechanisms, but effects on safety are analysed separately. Effects on the resilience of the system are evaluated as a whole with one column. For example, service manager enables detection and recovery of failed services, but can be considered to be a SPoF. Service manager improves resilience by providing a tool for managing a large number of services, but also adds more complexity and a potential external risk for the services. From a pure safety perspective, many of the methods providing partial level of functionality can be seen as risks, since they introduce uncertainty to the system, albeit for the benefit of resilience.

Table 3. Fault tolerance methods and their estimated effects ('+' positive effect, '-' negative effect, 'o' neutral)

Method	Fault detection	Fault recovery	Safety	Resilience
service manager	++	++	- (SPoF)	+/-
heartbeat	++	o	++	o
resource monitoring	+	o	+	+
loose coupling	+ (locating the fault)	o	+ (limit propagation)	++
fine-grained recovery	+	+	+/-	+
let it crash	+	o	+	o
hot standby	o	+	-	+
escalation	o	++	-	+
switch to fall-back service	-	+	-	+
switch to alternative service composition	o	+	-	++
fail-safe	+	o	+	-

Evaluation of other design criteria, performance, service-orientation and safety [P6]:

- Computing performance—small overhead for service management (estimated 5-10%, around 5% for the prototype with the bilateral teleoperation service composition active [P6]). Performance gains can likely be gained on multi-core CPUs due to the possibility to execute services in parallel (Amdahl's law).
- Real-time performance—although 100% real-time performance for networked communications cannot be guaranteed, GSB and LSB communications enable monitoring of the deadlines inside the services. Therefore, services can react accordingly and move to the fail-safe state if necessary. In practice, system performs without spurious safety trips.
- Service-orientation—the proposed architecture generally adheres to service-orientation principles (see Principles of Service Design by T. Erl [154] and [P6]), but when part of a composition, the services participating in a control loop are typically not stateless, especially in robotic or machine automation applications in which machinery cannot be allowed to simply start along with the services. Instead, services can utilize state machines to facilitate equipment initialization, safe state, possible simulation modes and controlled transitions to operational mode. Use of hierarchical state machines enables composition of states, which may be necessary to facilitate different control mode combinations.

- Safety—in case of a failure, caching of commands is not desirable as late commands become potentially hazardous. The physical world state could have changed before the command activates, resulting in unexpected behaviour. Caching may also lead to a burst of commands when service becomes available after recovery. Middleware QoS parameters can be used to control behaviour of safety-related data and commands. For example, in DDS with reliability (reliable vs. best effort), history (keep last n samples), ownership (exclusive vs. shared), liveliness of publishers, etc. Generally, fail-silent response is preferred over stale data, default values or "best guesses" for CPS applications, making faults visible fast and in a predictable manner.

5.3 Summary of Papers

In this section the included publications are summarized. The candidate is the main author in all included papers, responsible for planning and writing them. A list of related publications in which the author of this thesis is either the main author or a co-author is also presented.

Included Publications

[P1] discusses the current best practices and fault prevention, tolerance, removal and forecasting methods for building dependable systems. An approach for the development process of dependable systems based on systems engineering principles is proposed, with the goal of combining dependability and systems engineering approaches. Related research question: Q1.

[P2] considers the role of software in RH system dependability and compares it to the risk management codes and processes, which in the case of ITER seem to overlook software aspects of systems. The paper analyses practices for developing dependability-related requirements and considers the role of safety-critical functions and their effect on software requirements and architecture. Related research question: Q3.

[P3] presents the concept design derived from the Sulava RTSOA and the evaluation of its prototype implementation in order to assess the benefits of data-centric approach to service-based system development for CPSs. Fault tolerance based on heartbeat, resource limits, let it crash approach and service manager is proposed to demonstrate the capability of the architecture to support cost-efficient fault tolerance solutions. The paper demonstrates viability of the architecture for a heterogeneous and distributed system with real-time requirements. Related research questions: Q2, Q3 and Q4.

[P4] analyses the fault tolerance mechanisms of the architecture in the concept design (i.e., fault detection and recovery mechanisms in a loosely coupled system). The approach is based on isolating and recovering the faults on a service (or component) level. For transient faults, restarting the failed unit can be an effective strategy. The fault tolerance capabilities of the architecture are extended and implemented into the prototype (e.g., action to be taken on a service failure). Related research questions: Q5 and Q6.

[P5] presents three software patterns for improving control system dependability with light-weight fault tolerance and decoupled design. The patterns are DATA-CENTRIC ARCHITECTURE, SERVICE MANAGER and LET IT CRASH. In order to show how the patterns fit the gaps in the existing pattern literature and how they address CPS-specific needs, a pattern language for fault tolerance in CPSs is also outlined. Related research questions: Q5 and Q6.

[P6] empirically evaluates the approach described in the previous papers by extending the prototype to use a 6 DOF input device with force feedback (bilateral teleoperation). The prototype is extended to support fault detection based on exit signals and service status updates, and fault handling based on fault escalation, including switching to a backup service. The evaluation results show that the service manager can successfully detect failed services with all implemented methods without significant overhead (around 5% CPU usage for the test scenario). Fault handling and recovery is tested with all implemented methods as well, including recovery of a critical service participating in a closed-loop control running at 500 Hz. Based on the results, services can be recovered without cascading failures and can, therefore, function as units of fault isolation. Related research questions: Q6 and Q7.

Related Publications
The author has been the main author or a contributor for the following publications that are related to the topic of this thesis, but have not been included:

Pekka Alho, Jouni Mattila, Antti Hahto, and Liisa Aha, "Planning of Dependable Remote Handling Control System Architecture for ITER", Proceedings of the European Nuclear Young Generation Forum 2011 (ENYGF), May 17-22, 2011, Prague, Czech Republic. http://urn.fi/URN:NBN:fi:tty-2011061714709

Pekka Alho and Jari Rauhamäki, "Patterns for Fault Tolerant and Cost-Efficient Design of Distributed Control Systems", Proceedings of the VikingPLoP 2013 conference, April 21-24, 2013, Ikaalinen, Finland. http://URN.fi/URN:ISBN:978-952-15-3167-5

Janne Tuominen, Mikko Viinikainen, Pekka Alho and Jouni Mattila, "Using a Data Centric Event-Driven Architecture approach in the integration of real-time systems at DTP2", Fusion Engineering and Design 89(9–10) 2014, pp. 2289–2293, Elsevier. http://dx.doi.org/10.1016/j.fusengdes.2014.04.040

Mikko Viinikainen, Janne Tuominen, Pekka Alho and Jouni Mattila, "Improving the performance of DTP2 bilateral teleoperation control system with haptic augmentation", Fusion Engineering and Design 89(9-10) 2014, pp. 2272-2277, Elsevier. http://dx.doi.org/10.1016/j.fusengdes.2014.04.045

Pekka Alho, "Data-centric middleware: Data Distribution Service (DDS) for real-time systems", in Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen, Ville Reijonen, "Designing Distributed Control Systems: A Pattern Language Approach", 2014, pp. 44-47, Wiley.

6 Summary and Discussion

6.1 Summary

Residual faults, including faults showing intermittent behaviour, are unavoidable in complex, large-scale CPSs, such as ITER RHCSs, that are expected to provide dynamic functionality. A control system failure could lead to an equipment breakdown and a loss of operation time, affecting the operations of a whole plant. However, the use of fail-operate fault tolerance is not economically viable for these systems and cannot protect against common-cause failures. Moreover, an RH system consists of a wide range of heterogeneous subsystems, necessitating a high level of interoperability and an ability to adapt to changes in the computing environment (i.e., resilience). Flexibility and reusability are needed to improve the cost efficiency of large-scale systems with long lifecycles. These goals are supported through modular architecture and open platforms.

The main goal of this thesis was to develop a fault-tolerant control system architecture based on the requirements of the ITER RH systems. To achieve this goal, the thesis presents an approach for building resilient CPSs using fault tolerance based on loosely coupled architecture and diversity. The resilience is achieved by relying on the properties of the service-oriented architectural style, a real-time capable computing platform and a dedicated communication middleware. The architecture has been designed to meet the needs of interoperability and the timely dissemination of data in open environments. Each software module used within a CPS is considered to be a service, which:

- communicates with other services by means of well-specified service contracts over peer-to-peer communication buses, with support for real-time communication;
- can be in a specific state, such as stopped, started, running or error;
- can receive a set of commands (e.g., start, restart, stop) for managing its state within the whole life cycle; and
- cannot make assumptions about the availability states of other services.

By utilizing bus-based communication, a set of generic services, called service managers, can control the lifecycles of all services within the nodes and can be used to manage compositions and to start, restart and stop services within the CPS.

This thesis has evaluated and discussed the management of fault tolerance within a service-oriented CPS based on a prototype implementation of the described architecture. The proposed approach builds on the concept of a service as a unit for managing fault isolation, transient fault detection (via a combination of heartbeat and resource monitoring at run-time) and transient faults recovery (via a system of fault escalation levels). This approach has been demonstrated in practice via an empirical evaluation performed in the context of a project in the area of remotely operated robots. The performed experiments show that the let-it-crash approach, based on terminating and restarting services, can support real-time, service-based fault detection and recovery.

Applying fail-operate fault tolerance or software development according to the rigid recommendations given in safety standards may not be economically viable within the scope of

CPSs, which typically are dynamically changing systems of systems. A systems engineering approach is, therefore, essential for building dependable CPSs; moreover, to support dependability, other means besides fault tolerance were also considered. In particular, dependability assurance through the building of a dependability case was seen to be a promising approach for describing and amalgamating various methods to achieve dependability in the context of systems engineering. The systems engineering approach presented in this thesis to achieve dependability for CPSs presented includes:

- Fault forecasting: fault and failure taxonomies for RH.
- Fault prevention: the role of safety-critical requirements and modules considered, support for dependability case development.
- Fault tolerance: loosely coupled architecture supporting fault tolerance patterns, fault isolation and fault recovery.
- Fault removal: V&V and run-time assessment using virtual 3D models (or simulated plants).

A dependability assurance approach would enable developers to focus on providing evidence for the critical parts of the system, which would be essential for the cost-efficient development of mixed-criticality systems. Key factors for using such an approach, as identified in this thesis, include the identification of critical subsystems and modules and a focus on the V&V efforts on these components, including software requirements and demonstrations of software dependability and safety (i.e., evidence-based assurance). Moreover, there is a need to improve the resilience of these systems so that they can dependably function as parts of future large-scale, interconnected and evolving information infrastructures [13]. To support dependability and resilience, this thesis has illustrated how to use a systems engineering approach to building dependable CPSs—that is, open and interconnected systems that control physical plants—in the specific context of RH systems. This approach is combined with fault tolerance solutions that support intrinsic system resilience through modularity, loose coupling and diversity.

In order to take into account the systems engineering oriented dependability approach, the results are presented as a systems engineering approach. The approach is not intended as a development process to follow to be followed, but rather to show how the proposed approaches, COTS components, reuse of components, quantitative and qualitative reliability analysis for software, verification using VR models, etc. fit into the general systems engineering processes and enable integration of fault tolerance approaches to system development.

6.2 Validity

It is important to discuss the generality of the results, since the research was carried out in the specific context of ITER RH systems. For the results to be applicable to CPSs, ITER RHCSs must be accepted as CPSs; argumentation supporting this point is presented in Sections 1 and 4. RH is a very specific domain, which is why this thesis chose to focus on CPSs in general. The results in this thesis have been obtained using a constructive research methodology, which emphasizes the building and evaluation of artefacts. The major contributions in this thesis, therefore, are related to fault tolerance and software architecture; the systems engineering perspective is used primarily to link the gained knowledge to the existing. To these aims, this thesis has analysed the use and performance of the architecture, methods and models in order to understand, explain and improve the design of CPSs, which represent a significant development trend for embedded real-time systems in general. However, it should be noted that no statistical data analysis has been performed on the results presented in the papers.

The applicability of the results is analysed in relation to the scope of CPSs, since the research focuses on solving problems encountered when developing and applying a service-based architecture with real-time requirements set by interactions with physical processes and the environment. In a sense, the specific control task the services are carrying out does not affect the basic needs for dependability and resilience; thus, the choice of architecture and design decisions are made based on the balancing of forces for a specific application. This is also reflected in the patterns identified in [P5].

As pointed in the review of the state of the art, the fault tolerance capabilities of SOA—especially RTSOA—represents a gap in the existing knowledge, and the publications in this thesis provide evidence for the applicability of the service-based approach to building resilient CPSs. Further research is needed to verify the generality of the results by comparing them to another implementation (based on, for example, a robotic framework), even though the applicability of such results would be limited by the environment and skill factors.

Instead of developing domain-specific frameworks or protocols, an approach utilizing generic COTS and FOSS components was used to maximize interoperability and to obtain a compromise between dependability and affordability. The dependability of these components is based on the "proven in use" argument, which can be applied to the dependability case; however, this factor was not extensively researched (i.e., a complete dependability case for the RTSOA-based RHCS was not developed).

Simulations (e.g., VR IHA3D and CAT) enable the safe testing of advanced functionality and failure modes, such as incorrect movement on contact or near singularities. Loosely coupled services support V&V of individual services and service deployment integration testing. Such testing can be carried out with VR models (as in the case of the developed prototypes) or with simulated plant models (e.g., by changing a robot service to a mock version that simulates the manipulator). However, in empirical evaluations using VR models, some functionality related directly to the physical properties of the plant (e.g., force sensing on physical contact), cannot be fully tested with simulations. Finally, prior verification of all possible service chains and combinations may not be possible due to the number of potential combinations, which means that testing new service compositions is still a potential issue, if this step is not included as part of the normal testing and deployment process.

The objective of this thesis was to develop of cost-effective practices for dependable and resilient designs for CPSs. Although the cost efficiency of the contributions is a subjective matter that depends on context (e.g., downtime costs vs. development costs), some reasoning for why the presented results might prove to be cost-efficient in actual use has been provided. Modularity does not require extensive development costs the way that diverse redundancy does, and it supports several system quality attributes related to evolvability, such as maintainability and scalability.

The Sulava reference architecture has been used in [38] and [39] for embedded machine control. The research in this thesis extends the fault tolerance capabilities and carries out an empirical evaluation with the RH system, using different operation modes and heterogeneous subsystems. Although the system can be considered to feature typical CPS challenges, the author does not claim that the results can be generalized to *all* CPSs. Rather, the research shows the viability of using a loosely coupled architectural model for implementing fault tolerance and improving system resilience in timing-constrained CPS scenarios with non-deterministic service recovery. The goal was to prove that an approach based on loose coupling and modularity could be extended to build dependable and resilient real-time systems. Some tasks in CPSs have significantly more relaxed timeliness requirements—and can use, for example, Web services to implement the SOA model—whereas some tasks have strict hard real-time requirements, necessitating solutions with verifiable real-time properties.

Limitations

Fault taxonomies are RH-specific, based on data collected from a single facility over a long period of time (over 10 years). Dependability case development using the systems engineering approach has not developed empirically. The presented approach is based on research into dependability and safety cases, research in [P2], and experiences from the GOT-RH program. The systems engineering approach is not a new, scientifically evaluated result; however, it shows how the results of this thesis fit into the overall field of dependability in CPSs.

Communication and real-time determinism have been empirically evaluated and rely on proven algorithms; however, the automatic configuration or verification of real-time properties is not supported. The design has not been formally verified, since formal verification for this kind of system is not feasible due to the large number of external dependencies. Some parts of the work, such as state models for services, could be used together with formal verification methods and code generation in the future, while formal methods are generally straightforward to integrate into a dependability case.

The architecture described in this thesis has not been developed to implement safety functions. Such functions are theoretically feasible, if implemented with safety-certifiable middleware[4] and operating systems, but the evaluation of the cost efficiency of such an approach is beyond the scope of this thesis. Moreover, some of the proposed functionality may not be desirable in safety-critical systems. For example, dynamic reconfiguration is not recommended on SIL 2 or higher in the IEC 61508 [41].

One of the goals of this thesis was to design a system that was recoverable against any SPoF. The architecture presented in this thesis enables the use of real-time services as units of fault containment and recovery, and the system can be operated with diverse operating modes and subsystems. However, the architecture cannot *guarantee* the fault isolation of services, since a faulty service could crash a whole node or send erroneous messages to other services. The service manager, EC and some services (e.g., C4G) are SPoFs in the prototype system that require

[4] For example, TwinOaks offers CoreDX DDS—SE Safety Critical Edition, although this is aimed at the DO-178B aviation-related certification.

dependability assurance, but services can also be operated independently without the service manager, enabling a true “peer-to-peer” architecture with no SPoF. An obvious downside to this approach is a lack of service composition management, which results in the need for other release automation and management tools.

Self-adaptation can be used to build resilient systems that can adapt to a variety of situations, including environment changes and faults. A self-adaptive system will change its behaviour and structure without human intervention. For example, the automatic loading of a backup service can be considered to be a limited form of self-adaptation. The automatic loading of alternative service compositions would also be a potential approach to self-adaptation; however, this raises some questions about the dependability of such a system. This would be an interesting topic for further research—however, in the current form, no extensive self-adaptation capabilities are claimed.

6.3 Discussion

In the beginning of this thesis, four software failure cases relevant to CPSs were analysed. Software complexity, common-cause failures, and communication among software modules (and, thus, fault propagation) are all major challenges for CPSs. Systems must be evolvable, recoverable and robust against incorrect data. Modularity supports dependability through the separation of concerns, limiting fault propagation and enabling the flexible use of fault tolerance patterns; for these reasons, an approach based on the development of a loosely coupled, real-time capable architecture was chosen for this research. However, for CPSs, dependability is not sufficient to describe the requirements that these systems must fulfil. This thesis argues that resilience is also a key requirement for CPS, since there is a need to make systems more robust to unexpected changes. In order to implement resilience with diversity-based fault tolerance, the design of the system should be modular and loosely coupled. To achieve this goal, the research proposes and empirically verifies an RTSOA in a critical real-time application built through the reuse of existing and COTS software and hardware to develop an open platform. The developed open RTSOA has been designed to support the rapid prototyping and verification of control system components, such as force sensors, master arms, control algorithms, etc.

A publish-subscribe, data-centric communication based on real-time microservices has been used to achieve a fail-safe and recoverable system. The publish-subscribe model removes direct references from modules, whereas the data-centric communication makes the data payload “visible” to the middleware, enabling the application of advanced QoS policies based on, for example, the urgency of the data compared to message-based communication. For the sending of commands, procedure calls provide a convenient way to ensure that commands and requests are addressed (i.e., carried out). In a data-centric communication model, this is directly possible; however, status updates can be published that reflect the effects of the received commands. If no data are published, the sender of the command should presume that the service is in an incorrect or non-active state.

The service-based approach enables teams to choose the internal architecture of the service and implementation tools based on their expertise and on the specific needs of the task. This may result in a need to support a wide range of platforms; however, on the positive side, small and loosely coupled services are straightforward to replace. The obvious trade-off points include possible restrictions based on safety (e.g., limitations related to applicable tools and environments) and the verification of new service compositions. Therefore, service development benefits from the use of supporting methods (e.g., test-driven development [155], deployment automation or DevOps). The key point is that the service-based approach supports utilizing the expertise of development teams and integration of various services/subsystems developed by different teams into a single system—a benefit that may support the development of large systems, such as the ITER RHCS.

On the other hand, the SOA design principle of loose coupling conflicts with the requirement for time-bounded operation in CPSs. Automatic service contract negotiations between service provider and consumer require knowledge of timing properties in order to decide whether an acceptable level of QoS can be provided, thereby introducing a level of coupling between the provider and consumer. In the approach used in this thesis, this question is left to the developer. Similarly, the dynamic discoverability and composition of services would not currently be admissible in safety-related systems.

Regardless of its benefits for the development of distributed systems, SOA is not a "silver bullet" that will solve the integration of heterogeneous nodes in IoT. The diversity of such applications is huge, and the requirements related to security, reliability, costs, performance, power, etc. can differ significantly from one application to another. Although the success of the Internet can be attributed to its compatibility, which is achieved through IP-based technologies and the relative simplicity of protocols, the Industrial Internet may not necessarily have such a convergent technology path. IP communications provide a common basis for the Industrial Internet, but there are many different approaches, protocols and platforms available, especially on the upper stack levels. Since many of these technologies are complementary, serving different requirements related to energy efficiency, deterministic communication, etc., there may be no single "winner". This reality may be a strength, in that it will provide suitable solutions for applications with different needs, or it may result in a highly segmented, domain-specific IoT. In such a case, there will be a need to support several protocols for different communication purposes to ensure sufficient flexibility of communication.

6.3.1 Cost Efficiency

One of the main benefits of the SOA approach is that the system can be cost-effectively evolved by using existing services to implement new tasks. This thesis used SOA design principles to develop a service-based architecture suitable for CPSs, enabling the reuse of existing software "as is" or with minor modifications. This service-based approach makes the system flexible to adapt, for example, from automatic teleoperation to manual teleoperation (i.e., a change from [P3] to [P4]) or to interface with subsystems. Further changes, such as the introduction of force sensors to the system, should be straightforward, although such changes were not conducted within the scope of this thesis.

The results of this thesis indicate that a loosely coupled and modular design can support a resilient CPS. However, these same features also support an evolving system. Therefore, it appears that these features have a high level of synergy for the building of CPSs.

To summarize, solutions supporting cost efficiency in this thesis include:

- Interoperable architecture, which supports choosing the most applicable and cost-efficient alternatives available for each service and subsystem. Services can be developed utilizing suitable tools and the special expertise of the development team.
- An autonomous nature of services, which facilitates relatively independent development by separate teams.
- Resilience based on the intrinsic features of architecture, utilizing diversity and proven-in-use services as backups.
- Modular applications that share expensive and/or limited resources and peripherals, including RH work cells and master arms.
- Support for the reusability of existing services and special software subsystems (e.g., CAT, OMS, VR, etc.).
- The use of open, off-the-shelf communication solutions (e.g., middleware) instead of proprietary, in-house development.
- The flexible use of patterns to utilize existing design knowledge.

6.3.2 Patterns

SERVICE MANAGER together with loosely coupled services provides fault tolerance and enables the management of services (e.g., deployment, fault detection and recovery). SERVICE MANAGER answers the specific needs of CPSs by providing an automated approach to the management of the system state through services and the monitoring of overall system health. The number of services running on a deployed CPS could range from a few to hundreds per node, depending on the service granularity. For an autonomous system or a system deployed in remote locations, response time for a human operator (or a system administrator) in transient failure situations could lead to a loss of production time, particularly in cases of numerous services. SERVICE MANAGER solves the problem of manually managing a large number of services by minimizing human intervention.

LET IT CRASH, on the other hand, approaches the dependability and resilience problem from the service point of view, seeking to stop the propagation of errors to the service interfaces. In a way, the service leaves the responsibility of reacting to a failure to other parties. This not only leverages the strength of loose coupling, but also increases this strength. Naturally, this pattern is not mutually exclusive with the normal practice of checking for errors and exception handling (i.e., defensive programming). LET IT CRASH can be also said to support the fail-safe functionality of a safety-critical system, since it aims to expose faults rapidly and in a well-defined manner, rather than through more complex, out-of-specification failures.

The LET IT CRASH and SERVICE MANAGER patterns are forms of temporal redundancy that do not require redundant hardware or software. Instead, they require time for the code to re-execute. This approach is identified as a cost-efficient solution in [51]. Embedded systems that typically have hard real-time constraints are not usually considered to be well-suited for this type of fault tolerance; however, based on empirical results, a suitable architectural design can mitigate some of the drawbacks.

6.3.3 Benefits

CPSs present a type of system for which dependability is critically important. Since this software has the power to directly affect and alter the physical world, the consequences of faults can be costly. Due to the interaction between software and the physical world, the systems engineering approach, which aims to combine different engineering disciplines, becomes even more important, particularly when pursuing safe system design and integrating software modules into system design processes, which commonly neglect any software reliability aspects beyond the safety-related parts of a system (as in the case of the ITER RAMI process) [P2].

Overall, CPSs represent an important part of the on-going digitalization trend. Commonly mentioned examples of applications of the IoT include preventive maintenance, remote operation and system optimization to save energy; however, digitalization also has the potential to open up completely new business models and methods of interaction with the physical world through control systems. Although CPSs may not necessarily support a "connect everything" attitude, an open and flexible architecture is a prerequisite for supporting this development. Such an architecture enables rapid responses to changing markets and customer demands, whether this involves the integration of a new type of manipulator into fusion reactor remote maintenance systems or the reconfiguration of a factory production line.

The use of RTSOA enables a system to be built out of loosely coupled and fine-grained services that focus on accomplishing a single task well. The resilience of the loosely coupled software architecture provides a flexible platform for implementing fault tolerance techniques, such as defence in depth and diversity, thereby providing a complementary approach to testing and development process-centric techniques. SOA fault tolerance is already utilized in practice by such companies as Netflix[5], which seek to build systems that are prepared for inevitable failures. Loose coupling furthermore supports the integration of simulations or VR models into systems to enable fast prototyping and early concept validation.

Benefits of this research include the evaluation and development of the RTSOA for embedded machine control, which provides a basis for implementing future research projects that require an open platform that supports rapid prototyping. These experiments prove the viability of using microservices in mission-critical real-time systems. Since humans can only consider a limited number of things at a time, decoupled and small units improves understanding and increases system maintainability and evolvability. Another potential benefit of building a system out of small, independent services is support for rapid experimentation with various solutions that combine diverse platforms, although this requires more effort in creating, collecting and utilizing data in smarter and finely grained ways. The proposed RTSOA model, which is based on a well-defined information model as a basis of communication, adds another layer of support to system evolvability. First, the hardware and computing environment can be changed, since services are not configured or verified to be dependent on these factors. Second, the individual services can be replaced in a straightforward manner, since new services only need to comply with the information model and with possible requirements for deterministic communication. This enables the rapid development of the architecture and its service prototypes, according to lean principles.

[5] See http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

6.4 Research Questions Revisited

These research questions were originally presented in Section 1. They seek to explore and further explain the broader question of how to achieve dependable system operations in open computing environments with CPSs.

Q1: What methods and practices are used to achieve dependable designs for control system software?

A systems engineering approach can be used to support and provide fault prevention, fault tolerance, fault removal and fault forecasting methods to achieve dependability. A systems engineering approach can also be used to collect evidence for a dependability case that can support the dependability validation of the software.

Q2: How can CPSs be designed to provide service, regardless of faults—and, especially, how can fault tolerance be implemented without the extensive use of diverse redundancy?

Loosely coupled and modular architectures based on services support the recovery of faults without diverse redundancies. Single faults can still cause system failure in service-based systems; however, the fault can be isolated and recovered or circumvented through the use of alternative services or service compositions, utilizing diverse operation capabilities. Therefore, the fault tolerance implementation still uses both diversity and redundancy—which are, after all, integral parts of fault tolerance. The difference is that, instead of creating multiple redundant and diverse versions of the whole application, these approaches can be applied in a more fine-grained, lean manner on the service level.

Q3: What requirements for system dependability exist for RH systems? How can architecture support the implementation and verification of these requirements?

A key requirement is the identification and separation of critical services and subsystems from the non-critical. Other identified requirement categories involve dependability and resilience objectives (i.e., responses to undesired or unexpected events), operation modes, timing requirements, operational profiles, resource usage, security, data quality and variability.

Architecture supports verification through the use of virtual or simulated models, since the modular nature of the system suggests an easy way to connect simulated subsystems and environments for virtual or hardware-in-the-loop simulation testing. To handle the challenge of multiple time scales, communication has been separated into two separate communication mechanisms: LSB and GSB.

Q4: Can a modular and loosely coupled (service-based) architecture support the dependability of distributed and heterogeneous real-time systems?

Pre-verification of services and their real-time properties is challenging, since such systems are affected by the locations of the service provider and the consumer, by resource availability, etc. This reality degrades system flexibility and means that an exponential effort is required to increase the number of services. Previous research on topic has developed time-bounded reconfiguration algorithms, but this thesis presents an alternative approach that provides best-effort availability, utilizing loosely coupled and independent natures of services to recover a system from failures. The approach combines deterministic operation with minimal assumptions about other services, connecting services in an ad hoc manner. Therefore, a service failure does not differ from normal service shutdown, as in the case of recovery-oriented computing [156]. A heterogeneous and distributed real-time system was used to empirically evaluate the service-oriented approach with a mission-critical application: however, the approach has not been formally proven.

Q5: What mechanisms can be applied to detect and recover faults in such loosely coupled systems?
Fault tolerance patterns provide a selection of commonly used approaches. For example, the SERVICE MANAGER pattern is based on the provision of an external fault detection and recovery entity. The services themselves are not designed to make assumptions about the states of other services; however, they do need to implement additional checks for deadlines and the availability of other services. The recovery of a service uses the same path as the normal operation start-up, and error recovery paths are executed regularly to verify their proper functionality.

Q6: How can fault propagation between services be stopped?
Fault propagation can be limited to very small modules, such as services that can act as bulkheads (cf. BULKHEADS pattern [56]). Sanity checking, assertions and other verification methods can be used for input values, but the architecture cannot guarantee that a service will not appear to be alive, while providing erroneous values within expected value ranges. Failed services can also potentially crash the computing node, which may affect other services. Critical services may, therefore, need to be deployed on separate nodes. If fault propagation is limited to loosely coupled services, the origin of the fault is easier to trace and fix (although complex interaction faults between services may still be challenging to debug).

Q7: What is the overhead of using fault tolerance based on loose coupling?
Paper [P6] presents an evaluation of operational overhead, including CPU usage and the scalability of the fault tolerance solution. For the implementation effort, the overhead is significantly more challenging to estimate, especially since system requirements may call for a loosely coupled design anyway. On the other hand, some complexity is related to the management of services or modules (i.e., a service manager, in the case of the prototype).

References

[1] R. Aymar, V. Chuyanov, M. Huguet and Y. Shimomura, "Overview of ITER-FEAT - The future international burning plasma experiment," *Nuclear Fusion,* vol. 41, no. 10, pp. 1301-1312, 2001.

[2] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concpets and Taxonomy of Dependable and Secure Computing," *Transactions on Dependable and Secure Computing,* vol. 1, no. 1, 2004.

[3] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, Addison-Wesley, 2013.

[4] R. Hanmer, Patterns for Fault Tolerant Software, John Wiley & Sons, 2007.

[5] J. Armstrong, Making reliable distributed systems in the presence of software errors, Stockholm, Sweden: Royal Institute of Technology, 2003.

[6] NASA/JPL, "Mars Exploration Rover - Opportunity," [Online]. Available: http://www.jpl.nasa.gov/missions/mars-exploration-rover-opportunity-mer/. [Accessed 29 September 2014].

[7] J. Downer, "When Failure is an Option: Redundancy, reliability and regulation in complex technical systems," London School of Economics and Political Science , 2009.

[8] Defense Industry Daily, "F-22 Squadron Shot Down by the International Date Line," 1 March 2007. [Online]. Available: http://www.defenseindustrydaily.com/f22-squadron-shot-down-by-the-international-date-line-03087/. [Accessed 30 October 2014].

[9] The Local, "Germany's super train arrives - two years late," 18 February 2014. [Online]. Available: http://www.thelocal.de/20140218/germanys-new-super-train-ice-three-arrives-two-years-late-deutsche-bahn-siemens. [Accessed 30 October 2014].

[10] Der Spiegel 48/2012, "Eine Sekunde und 70 Meter," 26 November 2012. [Online]. Available: http://www.spiegel.de/spiegel/print/d-89801822.html. [Accessed 30 October 2014].

[11] K. Kielhofner, "Packets of Death," [Online]. Available: http://blog.krisk.org/2013/02/packets-of-death.html. [Accessed 30 October 2014].

[12] S. Gallagher, "U-2's flight plan was like malware to FAA computer system," Ars Technica, 12 May 2014. [Online]. Available: http://arstechnica.com/information-technology/2014/05/u-2s-flight-plan-was-like-malware-to-faa-computer-system/. [Accessed 30 October 2014].

[13] J.-C. Laprie, "From Dependability to Resilience," in *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, 2008.

[14] National Science Foundation, "Call for Position Papers," NSF Workshop On Cyber-Physical Systems, October 2006. [Online]. Available: http://varma.ece.cmu.edu/CPS/. [Accessed 30 October 2014].

[15] S. Engell, "Cyber-physical Systems of Systems – Definition and core research and innovation areas," Towards a European Roadmap on Research and Innovation in Engineering and Management of Cyber-Physical Systems of Systems, 2014.

[16] P. Guturu and B. Bhargava, "Cyber-Physical Systems: A confluence of Cutting Edge Technology Streams," in *International Conference on Advances in Computing and Communication ICACC-11*, Hamipur, India, 2011.

[17] J. Eidson, E. Lee, S. Matic, S. Seshia and J. Zou, "Distributed Real-Time Software for Cyber–Physical Systems," *Proceedings of the IEEE (special issue on CPS),* vol. 100, no. 1, pp. 45 - 59, 2012.

[18] R. R. Rajkumar, I. Lee, L. Sha and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th Design Automation Conference*, 2010.

[19] V.-P. Eloranta, J. Koskinen, M. Leppänen and V. Reijonen, Designing Distributed Control Systems: A Pattern Language Approach, Wiley, 2014.

[20] R. Hayama, M. Higashi, S. Kawahara, S. Nakano and H. Kumamoto, "Fault-tolerant automobile steering based on diversity of steer-by-wire, braking and acceleration," *Reliability Engineering and System Safety,* vol. 95, pp. 10-17, 2010.

[21] P. Caliebe, C. Lauer and R. German, "Flexible integration testing of automotive ECUs by combining AUTOSAR and XCP," in *Computer Applications and Industrial Electronics (ICCAIE), 2011 IEEE International Conference on*, Penang, 2011.

[22] R. Hegde, G. Mishra and K. Gurumurthy, "An Insight into the Hardware and Software Complexity of ECUs in Vehicles," in *Advances in Computing and Information Technology*, Springer, 2011, pp. 99-106.

[23] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE,* vol. 68, no. 9, pp. 1060-1076, 1980.

[24] D. Herrmann, Software Safety and Reliability, IEEE, 1999.

[25] NASA, NASA Software Safety Guidebook, 2004.

[26] D. Jackson, M. Thomas and L. Millett, "Software for Dependable Systems: Sufficient Evidence?," National Academies Press, Washington, D.C., 2007.

[27] J. Rauhamäki, T. Vepsäläinen and S. Kuikka, "Patterns in Safety System Development," in *International Conference on Performance, Safety and Robustness in Complex Systems and*

Applications, Venice, 2013.

[28] M. P. Papazoglou, “Service-oriented computing: concepts, characteristics and directions,” in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, 2003.

[29] L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, 2001.

[30] K.-D. Kim and P. Kumar, “Cyber–Physical Systems: A Perspective at the Centennial,” *Proceedings of the IEEE,* vol. 100, no. Special Centennial Issue, pp. 1287-1308, 2012.

[31] D. Jackson, “A Direct Path to Dependable Software,” *Communications of the ACM ,* vol. 52, no. 4, 2009.

[32] B. Kitchenham, T. Dyba and M. Jorgensen, “Evidence-based software engineering,” in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004.

[33] G. Muller, “Systems Engineering Research Methods,” in *2013 Conference on Systems Engineering Research*, 2013.

[34] S. Easterbrook, J. Singer, M. A. Storey and D. Damian, “Selecting empirical methods for software engineering research.,” *Guide to advanced empirical software engineering,* pp. 285-311, 2008.

[35] G. D. Crnkovic, “Constructive Research and Info-computational Knowledge Generation,” in *Model-Based Reasoning in Science and Technology*, Springer, 2010, pp. 359-380.

[36] A. Hevner and S. Chatterjee, Design Research in Information Systems: Theory and Practice, Springer, 2010, pp. 9-22.

[37] J. F. Nunamaker Jr, M. Chen and T. D. Purdin, “Systems development in information systems research,” *Journal of Management Information Systems,* vol. 7, no. 3, pp. 89-106, 1991.

[38] A. Hahto, T. Rasi, T. Kivelä, J. Honkakorpi and J. Mattila, “Service-oriented ad-hoc method for unforeseen scenarios,” in *International Workshop on Robotics for Risky Interventions and Environmental Surveillance-Maintenance IARP RISE, C-ELROB*, Brussels, 2011.

[39] A. Hahto, T. Rasi, J. Mattila and K. Koskimies, “Service-oriented architecture for embedded machine control,” in *International Conference on Service-Oriented Computing and Applications*, Irvine, CA, USA, 2011.

[40] INCOSE, Systems Engineering Handbook, version 3, 2006.

[41] IEC, “SFS-EN 61508-3 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems - Part 3: Software requirements,” Suomen Standardisoimisliitto SFS, 2010.

[42] NASA, "NASA Systems Engineering Handbook," 2007.

[43] A. Somani and N. Vaidya, "Understanding Fault Tolerance and Reliability," *Computer,* vol. 30, no. 4, 1997.

[44] M. Lyu, Handbook of Software Reliability Engineering, IEEE Computer Society Press and McGraw-Hill, 1995.

[45] B. Hargrave and P. Kriens, "OSGi Best Practices!," in *First OSGi Alliance Community Event*, Munich, 2007.

[46] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on* , 1995.

[47] C. Jones, "Software quality in 2011: a survey of the state of the art," 31 August 2011. [Online]. Available: http://sqgne.org/presentations/2011-12/Jones-Sep-2011.pdf. [Accessed 14 August 2012].

[48] J. Knight and N. Leveson, "An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Programming," *Transactions on Software Engineering,* vol. 12, pp. 96-109, 1986.

[49] J.-C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *Computer,* vol. 23, no. 7, pp. 39-51, 1990.

[50] S. Chiaradonna, A. Bondavalli and L. Strigini, "On Performability Modeling and Evaluation of Software Fault Tolerance Structures," in *EDCC-1 Proceedings of the First European Dependable Computing Conference on Dependable Computing* , Berlin, 1994.

[51] M. Hiller, "Software Fault-Tolerance Techniques from a Real-Time Systems Point of View: combination of real-time systems and fault-tolerance," Chalmers University of Technology, Göteborg, 1998.

[52] W. Dunn, "Designing Safety-Critical Computer Systems," *Computer,* vol. 36, no. 11, pp. 40-46, 2003.

[53] B. Douglass, Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison-Wesley Professional, 2002.

[54] S. Szominski, K. Gadek, M. Konarski, B. Blaszczyk, P. Anielski and W. Turek, "Development of a cyber-physical system for mobile robot control using Erlang," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, Kraków, 2013.

[55] D. Sotirovski, "Towards fault-tolerant software architectures," in *Software Architecture, 2001.*

Proceedings. Working IEEE/IFIP Conference on , Amsterdam, Netherlands, 2001.

[56] M. T. Nygard, Release It!: Design and Deploy Production-Ready Software, Pragmatic Bookshelf, 2007.

[57] W. Torres-Pomales, Software Fault Tolerance: A Tutorial, NASA, 2000.

[58] C. Moron, "Designing adaptable real-time fault-tolerant parallel systems," in *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, 1996.

[59] R. J. Kreutzfeld and R. E. Neese, "Methodology for cost-effective software fault tolerance for mission-critical systems," *Aerospace and Electronic Systems Magazine,* September 1997.

[60] D. Winter and B. Rest, "Open system Ada technology demonstration program," in *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE* , Irvine, CA , USA, 1997.

[61] A. Tai, K. Tso, L. Alkalai, S. Chau and W. Sanders, "Low-cost error containment and recovery for onboard guarded software upgrading and beyond," *Computers, IEEE Transactions on,* vol. 51, no. 2, pp. 121-137, 2002.

[62] J. Brown and B. Martin, "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications," in *Twelfth Real-Time Linux Workshop*, Nairobi, 2010.

[63] Coverity, "Coverity Scan: 2012 Open Source Report," 2013. [Online]. Available: http://softwareintegrity.coverity.com/register-for-the-coverity-2012-scan-report.html. [Accessed 27 April 2015].

[64] Coverity, "Coverity Scan: 2013 Open Source Report," 2013. [Online]. Available: http://softwareintegrity.coverity.com/register-for-scan-report-2013.html. [Accessed 27 April 2015].

[65] E. Kesseler, "Assessing COTS software in a certifiable safety-critical domain," *Information Systems Journal,* vol. 18, no. 3, 2008.

[66] N. Leveson, "Using cots components in safety-critical systems," in *RTO Meeting on COTS in Defense Applications*, Brussels, 2000.

[67] P. Bishop, R. Bloomfield and P. Froome, "Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications," UK Health and Safety Executive Contract Research Report, CRR 336/2001, 2001.

[68] P. Asterio, C. Guerra, C. Mary and F. Rubira, "A Fault-Tolerant Software Architecture for COTS-Based Software Systems," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* , 2003.

[69] D. Hamilton and B. Haist, "A Rational Approach to Remote Handling Equipment Control System Design," EFDA, 2001.

[70] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac and A. Wellings, "GUARDS: a generic upgradable architecture for real-time dependable systems," *Transactions on Parallel and Distributed Systems,* vol. 10, no. 6, 1999.

[71] M. Vuori, "Agile Development of Safety-Critical Software," Tampere University of Technology, Department of Software Systems, 2011.

[72] A. Sidky and J. Arthur, "Determining the Applicability of Agile Practices to Mission and Life-Critical Systems," in *SEW '07 Proceedings of the 31st IEEE Software Engineering Workshop*, 2007.

[73] B. Boehm and V. Basili, "Software Defect Reduction Top 10 List," *Computer,* vol. 34, no. 1, 2001.

[74] R. R. Milagaia, DPWS middleware to support agent-based manufacturing control and simulation, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2009.

[75] T. Vepsäläinen and S. Kuikka, "Integrating model-in-the-loop simulations to model-driven development in industrial control," *Simulation,* 13 October 2014.

[76] M. Kaufman and J. Hurwitz, Service Virtualization For Dummies, IBM Limited Edition, John Wiley & Sons, Inc., 2013.

[77] C. Areias, N. Antunes and J. C. Cunha, "On Applying FMEA to SOAs: A Proposal and Open Challenges," in *6th International Workshop, SERENE 2014, Proceedings*, Budapest, Hungary, 2014.

[78] T.-L. Chu, M. Yue and W. Postma, "A Summary of Taxonomies of Digital System Failure Modes Provided by the DigRel Task Group," in *11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference 2012 (PSAM11 ESREL 2012),* Helsinki, 2012.

[79] S. Bruning, S. Weissleder and M. Malek, "A Fault Taxonomy for Service-Oriented Architecture," in *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE,* Plano, TX, 2007.

[80] S. Bruning, S. Weissleder and M. Malek, "A Fault Taxonomy for Service-Oriented Architecture," in *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE,* Plano, TX, 2007.

[81] J. H. Hayes, "Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project," in *ISSRE '03 Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[82] B. Shim, B. Baek, S. Kim and S. Park, “A Robot Fault-Tolerance Approach Based on Fault Type,” in *Quality Software, 2009. QSIC '09. 9th International Conference on*, Jeju, 2009.

[83] R. Melchers, “On the ALARP approach to risk management,” *Reliability Engineering & System Safety,* vol. 71, no. 2, 2001.

[84] T.-L. Chu, M. Yue, G. Martinez-Guridi and J. Lehner, “Review of quantitative software reliability methods,” Brookhaven National Laboratory, 2010.

[85] O. Bäckström and J.-E. Holmberg, “Use of IEC 61508 in nuclear applications regarding software reliability,” in *11th International Probabilistic Safety Assessment and Management Conference & The Annual European Safety and Reliability Conference*, 2012.

[86] H. Langseth and L. Portinale, “Bayesian Networks in Reliabiltiy,” *Reliability Engineering and System Safety,* vol. 92, pp. 92-108, 2007.

[87] J. D. Musa, “Validity of execution-time theory of software reliability,” *Reliability, IEEE Transactions on,* vol. 28, 1979.

[88] J. D. Musa and K. Okumoto, “A logarithmic Poisson execution time model for software reliability measurement,” in *ICSE '84 Proceedings of the 7th international conference on Software engineering*, 1984.

[89] Y. Ting, F. Shan, W. Lu and C. Chen, “Implementation and evaluation of failsafe computer-controlled systems,” *Computers & Industrial Engineering,* vol. 42, no. 2-4, pp. 401-415, 2002.

[90] S. Dick, C. L. Bethel and A. Kandel, “Software-Reliability Modeling: The Case for Deterministic Behavior,” *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on,* vol. 37, no. 1, pp. 106-119, 2007.

[91] H. Pham, System Software Reliability, Springer, 2007.

[92] A. Immonen and E. Niemelä, “Survey of reliability and availability prediction methods from the viewpoint of software architecture,” *Software and Systems Modeling,* vol. 7, no. 1, pp. 49-65, 2008.

[93] T. Cucinotta and D. Faggioli, “An exception based approach to timing constraints violations in real-time and multimedia applications,” in *Industrial Embedded Systems (SIES), 2010 International Symposium on* , Trento, 2010.

[94] C. B. Weinstock, J. B. Goodenough and J. J. Hudak, “Dependability Cases,” Software Engineering Institute, 2004.

[95] G. Despotou and T. Kelly, “Design of Dependability Case Architecture during System Development,” in *Proceedings of the 25th International System Safety Conference (ISSC),*

Baltimore, MD, 2007.

[96] R. Weaver, G. Despotou, T. Kelly and J. Mcdermid, "Combining Software Evidence: Arguments and Assurance," in *Proceedings of the 2005 workshop on Realising evidence-based software engineering*, 2005.

[97] T. P. Kelly, Arguing Safety - A Systematic Approach to Managing Safety Cases, PhD Thesis: Department of Computer Science, University of York, 1998.

[98] Y. Matsuno, K. Taguchi, Y. Nakabo and A. Ohata, "Iterative and simultaneous development of embedded control software and dependability cases for consumer devices," in *Society of Instrument and Control Engineers Annual Conference (SICE), 2012 Proceedings of*, Akita, 2012.

[99] Object Management Group, "Structured Assurance Case Metamodel (SACM)," [Online]. Available: http://www.omg.org/spec/SACM/. [Accessed 24 10 2014].

[100] D. Hinchcliffe, "A CIO's guide to the future of work," ZDNet, 29 October 2014. [Online]. Available: http://www.zdnet.com/a-cios-guide-to-the-future-of-work-7000035220/. [Accessed 4 November 2014].

[101] D. Boswarthick, O. Elloumi and O. Hersent, M2M communications: A systems approach, Wiley, 2012.

[102] J. Höller, V. Tsiatsis, C. Mulligan, S. Avesand, S. Karnouskos and D. Boyle, From Machine-To-Machine to the Internet of Things: Introduction to a New Age of Intelligence, Elsevier, 2014.

[103] ISO/IEC JTC 1, "Internet of Things (IoT) Preliminary Report 2014," ISO, Geneva, 2015.

[104] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on reliability in distributed software and database systems*, Los Angeles, 1986.

[105] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM,* vol. 15, no. 12, pp. 1053-1058, 1972.

[106] M. Fowler, "ServiceOrientedAmbiguity," 1 July 2005. [Online]. Available: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html. [Accessed 3 January 2015].

[107] K. Ashton, "That 'Internet of Things' Thing," RFID Journal, 22 June 2009. [Online]. Available: http://www.rfidjournal.com/articles/view?4986. [Accessed 27 December 2014].

[108] Open Source Robotics Foundation, "ROS," 9 May 2015. [Online]. Available: http://wiki.ros.org/. [Accessed 12 May 2015].

[109] M. Henning, "The rise and fall of CORBA," *Communications of the ACM,* vol. 51, no. 8, pp. 52-57, 2008.

[110] OPC Foundation, “Unified Architecture,” [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/. [Accessed 28 November 2014].

[111] V. Issarny, M. Caporuscio and N. Georgantas, “A Perspective on the Future of Middleware-based Software Engineering,” in *FOSE '07 Future of Software Engineering*, Washington DC, 2007.

[112] OMG, “Data Distribution Service Portal,” [Online]. Available: http://portals.omg.org/dds/. [Accessed 28 November 2014].

[113] G. Moritz, E. Zeeb, S. Prüter and F. Golatowski, “Devices Profile for Web Services and the REST,” in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, Osaka, 2010.

[114] MuleSoft, “Transports Reference,” [Online]. Available: http://www.mulesoft.org/documentation/display/current/Transports+Reference. [Accessed 19 November 2014].

[115] J. Lewis and M. Fowler, "Microservices," 25 March 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html. [Accessed 4 September 2014].

[116] SOA Work Group, “SOA Source Book,” The Open Group, 2011.

[117] H. P. Breivold, Software Architecture Evolution through Evolvability Analysis, PhD Thesis: Mälardalen University, 2011.

[118] H. P. Breivold and M. Larsson, “Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles,” in *EUROMICRO '07 Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007.

[119] H. Moussa, T. Gao, I.-L. Yen, F. Bastani and J.-J. Jeng, “Toward effective service composition for real-time SOA-based systems,” *Service oriented computing and applications,* vol. 4, no. Special Issue: RTSOAA, pp. 17-31, 2010.

[120] A. Machado and C. Ferraz, “Guidelines for performance evaluation of web services,” in *WebMedia '05 Proceedings of the 11th Brazilian Symposium on Multimedia and the web*, 2005.

[121] J. Gustafsson, R. Kyusakov, H. Mäkitaavola and J. Delsing, "Application of Service Oriented Architecture for Sensors and Actuators in District Heating Substations," *Sensors,* vol. 14, pp. 15553-15572, 2014.

[122] W3C, “Web Services Description Language (WSDL) Version 2.0 Part 1: Core language,” 26 June 2007. [Online]. Available: http://www.w3.org/TR/wsdl20/. [Accessed 3 January 2015].

[123] Reverb, "Swagger," 2014. [Online]. Available: http://swagger.io/. [Accessed 3 January 2015].

[124] T. Cucinotta, A. Mancina, G. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo and F. Rusina, "A Real-Time Service-Oriented Architecture for Industrial Automation," *Industrial Informatics, IEEE Transactions on* , vol. 5, no. 3, 2009.

[125] H. Kopetz, Real-time systems: Design principles for distributed embedded applications, 2nd ed., Springer, 2011.

[126] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi and M. Poncino, "Energy-Efficient Multiprocessor Systems-on-Chip for Embedded Computing: Exploring Programming Models and Their Architectural Support," *Computers, IEEE Transactions on,* vol. 56, no. 5, pp. 606-621, 2007.

[127] G. Cândido, J. Barata, A. W. Colombo and F. Jammes, "SOA in reconfigurable supply chains: A research roadmap," *Engineering Applications of Artificial Intelligence,* vol. 22, no. 6, pp. 939-949, 2009.

[128] W. Tsai, Y.-H. Lee, Z. Cao, Y. Chen and B. Xiao, "RTSOA: Real-Time Service-Oriented Architecture," in *Service-Oriented System Engineering, 2006. SOSE '06. Second IEEE International Workshop*, Shanghai, 2006.

[129] M. Garcia Valls, I. Lopez and L. Villar, "iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems," *Industrial Informatics, IEEE Transactions on* , vol. 9, no. 1, pp. 228-236, 2012.

[130] S. de Deugd, R. Carroll, K. E. Kelly, B. Millett and J. Ricker, "SODA: service-oriented device architecture," *IEEE Pervasive Computing,* vol. 5, no. 3, pp. 94-96, 2006.

[131] A. Tiderko, T. Bachran, F. Hoeller and D. Schulz, "RoSe - A framework for multicast communication via unreliable networks in multi-robot systems," *Robotics and Autonomous Systems,* vol. 56, pp. 1017-1026, 2008.

[132] M. Panahi, W. Nie and K.-J. Lin, "A Framework for real-time service-oriented architecture," in *Commerce and Enterprise Computing, 2009 IEEE Conference on*, 2009.

[133] D. Rodrigues, R. Melo Pires, J. C. Estrella, M. Vieira, M. Corrêa, J. Batista Camargo Júnior, K. R. L. J. C. Branco and O. Trindade Júnior, "Application of SOA in Safety-Critical Embedded Systems," in *Convergence and Hybrid Information Technology*, Springer, 2011, pp. 345-354.

[134] D. Rodrigues, R. de Melo Pires, J. C. Estrella, E. A. Marconato, O. Trindade and K. R. L. J. C. Branco, "Using SOA in Critical-Embedded Systems," in *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, Dalian, 2011.

[135] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State of the Art Review," *Industrial Informatics, IEEE Transactions on,* vol. 7, no. 4, pp. 768-781, 2011.

[136] T. Parveen and S. Tilley, "A Research Agenda for Testing SOA-Based Systems," in *Systems Conference, 2008 2nd Annual IEEE*, 2008.

[137] V. Patu, "How assurance case technique would justify the dependability of the upload data-flow diagram of the Distributed E-Learning System called KISSEL," in *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2013*, Victoria, Canada, 2013.

[138] I. Norros, P. Kuusela and P. Savola, "A Dependability Case Approach to the Assessment of IP Networks," in *Emerging Security Information, Systems and Technologies, 2008. SECURWARE '08. Second International Conference on*, Cap Esterel, France, 2008.

[139] S. J. Blanchette, "Assurance Cases for Design Analysis of Complex System of Systems Software," Software Engineering Institute, 2009.

[140] J. G. Wijnstra, "Components, Interfaces and Information Models within a Platform Architecture," in *Generative and Component-Based Software Engineering*, Springer, 2001, pp. 25-35.

[141] A. Pras and J. Schoenwaelder, "RFC 3444: On the Difference between Information Models and Data Models," Internet Society, 2003.

[142] F. Buschmann, K. Henney and D. C. Schmidt, Pattern-oriented Software Architecture: A Pattern Language for Distributed Computing, vol. 4, John Wiley & Sons, 2011.

[143] J. Tuominen, M. Viinikainen, P. Alho and J. Mattila, "Using a data-centric event-driven architecture approach in the integration of real-time systems at DTP2," *Fusion Engineering and Design,* vol. 89, no. 9-10, p. 2289–2293, 2014.

[144] Y. Matsuno, K. Taguchi, Y. Nakabo and A. Ohata, "Iterative and simultaneous development of embedded control software and dependability cases for consumer devices," in *SICE Annual Conference (SICE), 2012 Proceedings of*, Akita, 2012.

[145] P. Haapanen and A. Helminen, "Failure mode and effects analysis of software-based automation systems," STUK, 2002.

[146] J. D. Musa, "Operational Profiles in Software-Reliability Engineering," *Software,* vol. 10, no. 2, pp. 14-32, 1993.

[147] R. Santamaria, "Here be backdoors: A journey into the secrets of industrial firmware," in *Black Hat*, 2012.

[148] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek and J. A. Halderman, “Green Lights Forever: Analyzing the Security of Traffic Infrastructure,” in *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT14)*, 2014.

[149] B. Krogh, M. Ilic and S. Sastry, “National Workshop on Beyond SCADA: Networked Embedded Control for Cyber-Physical Systems (NEC4CPS): Research Strategies and Roadmap,” Team for Research in Ubitquitous Secure Technology (TRUST), 2007.

[150] IAEA, “Protecting against Common Cause Failures in Digital I&C Systems of Nuclear Power Plants,” IAEA Nuclear Energy Series, Vienna, 2009.

[151] L. Sciavicco and S. B., Modeling and Control of Robot Manipulators, Springer, 1996.

[152] I. Aliaga, A. Rubio and E. Sanchez, “Experimental quantitative comparison of different control architectures for master-slave teleoperation,” *Control Systems Technology, IEEE Transactions on,* vol. 2, no. 1, pp. 2-11, 2004.

[153] E. A. Strunk and J. C. Knight, “Dependability through Assured Reconfiguration in Embedded System Software,” *Dependable and Secure Computing, IEEE Transactions on,* vol. 3, no. 3, pp. 172-187, 2006.

[154] T. Erl, SOA: Principles of Service Design, Prentice Hall, 2008.

[155] D. Janzen and H. Saiedian, “Does Test-Driven Development Really Improve Software Design Quality?,” *Software, IEEE,* vol. 25, no. 2, pp. 77-84, 2008.

[156] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. a. E. P. Cutler, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman and N. Treuhaft, “Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies,” Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[157] S. Esque, J. Mattila, M. Siuko, M. Vilenius, J. Järvenpää, L. Semeraro, M. Irving and C. Damiani, “The use of digital mock-ups on the development of the Divertor Test Platform 2,” *Fusion Engineering and Design,* vol. 84, no. 2, pp. 752-756, 2009.

[158] M. Viinikainen, Computer-Aided Bilateral Teleoperation of Manipulators, MSc Thesis: Tampere University of Technology, 2014.

[159] I. Nam, “Enabling Multi-OS Embedded Systems with Hypervisor Technology,” Mentor Graphics, [Online]. Available: http://go.mentor.com/3fflj. [Accessed 29 March 2015].

[160] C. Shamieh, Systems Engineering For Dummies, IBM Limited Edition, John Wiley & Sons, Inc., 2011.

Appendix A: ITER RHCS Reference Architecture

The architecture for the ITER RH plant system is shown in Figure 23, representing the logical structure that combines the RH systems shown in Figure 1. The plant system is composed of RH supervisory control system and seven RH systems responsible for controlling different RH equipment (the in-vessel viewing system is not shown in the figure). Each of RH systems consists of high-level control system (HLCS) implementing operator interfaces, low-level control system (LLCS) consisting of device and tool controllers and finally the physical devices. Physically the HLCSs will be deployed to the RH control room, meaning that the implementation needs cyber components of the system to be accessible on any work cell in the control room and connections to desired low-level systems reconfigurable on-the-fly. RH Supervisory system and central coordination systems Control, Data Acquisition and Communication system CODAC, Central Interlock System CIS and Central Safety System CSS are left outside the scope of this work.

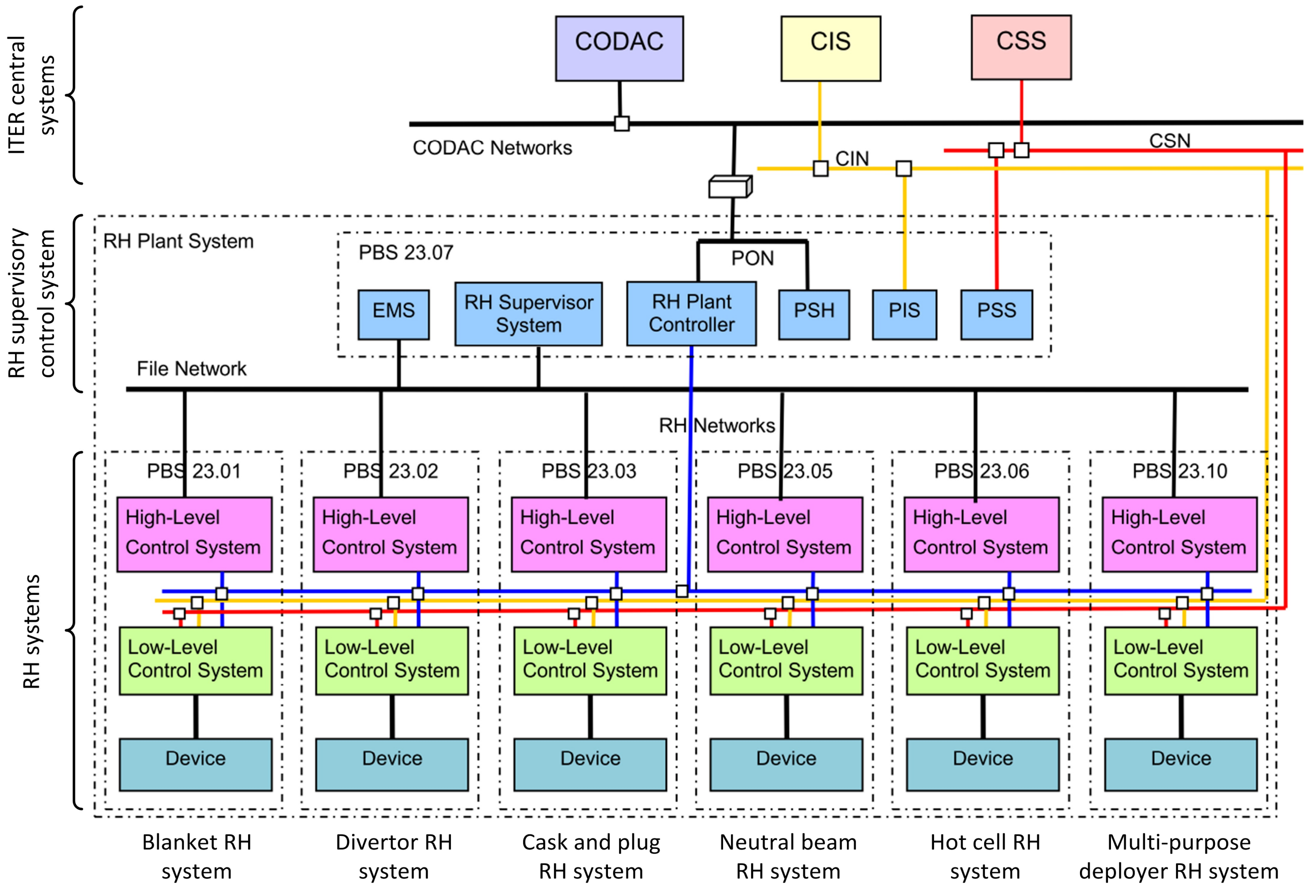


Figure 23. ITER RH plant system (image courtesy of ITER Organization), including Equipment Management System, Plant System Host, Plant Interlock System, Plant Safety System, Plant Operation Network, Central Interlock Network and Central Safety Network

RH systems share common components for supervisory and virtual reality subsystems, as shown in Figure 24 on the following page. Input devices have also been included in the HLCS, although they could be considered to be RH devices. The HLCS consists of following subsystems: supervisory, VR, remote diagnostics and viewing system. Structural simulator, remote diagnostics or viewing systems are not introduced since they are not used in the prototype implementation.

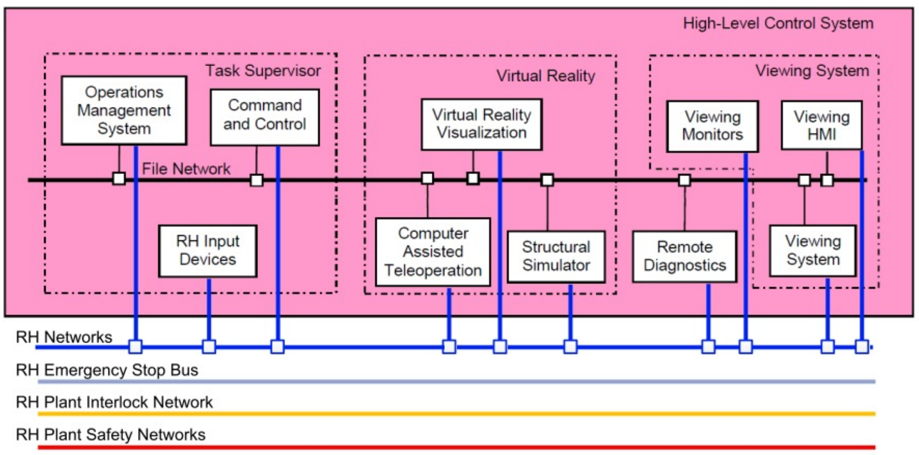


Figure 24. High-Level Control System (supervisory system), see Figure 2 for prototype implementation at DTP2 (image courtesy of ITER Organization)

Supervisor subsystem:

- OMS is used to support operation by planning, helping and instructing execution of RH procedures. Procedures are complete sequences of manual actions required to perform maintenance or testing operations. The IHA prototype OMS can also be used to carry out automatic operations.
- Command & Control provides a GUI that can be used to control RH equipment through its equipment controller.
- Input devices include joysticks and haptic input devices (master manipulators) used to control RH devices.

VR subsystem:

- Visualization software gives operators visual access to the virtual 3D model of the environment. Virtual reality software, such as IHA3D [157], can be used for task planning, practicing and simulation in addition to aiding the task execution with collision detection.
- Computer Assisted Teleoperation (CAT) [158] assists operators in teleoperation tasks that require manual operation of remote handling devices by generating virtual forces. The virtual forces can be used to guide the manipulators along pre-set paths or away from pre-set force barriers to avoid collisions.

LLCS controls and powers the RH devices. Figure 25 shows EC, the basic unit for integrating the field equipment, receiving commands from HLCS, translating them for the RH equipment and send data from equipment back to HLCS. ECs represent a translation point between the common HLCS subsystems and the equipment-specific control modules (PLCs, motion controllers, etc.) so they need to be flexible to support the diverse range of equipment, while being consistent in implementation to interface with high-level systems and support maintainability.

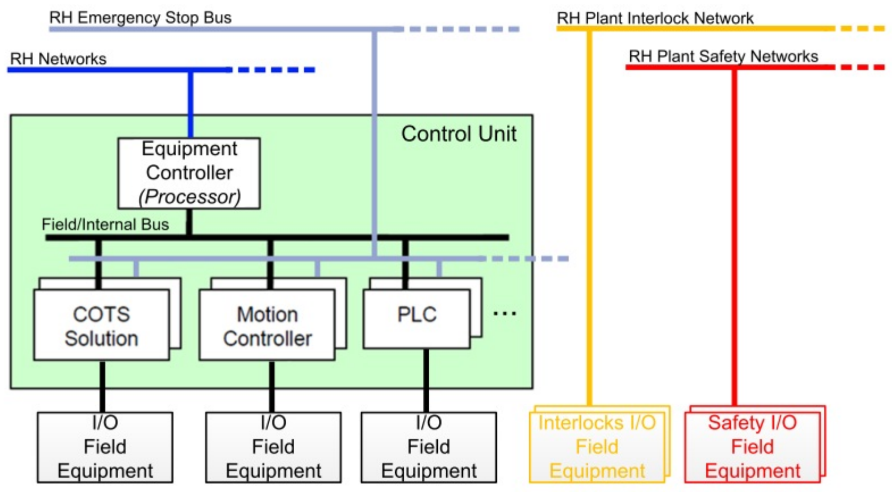
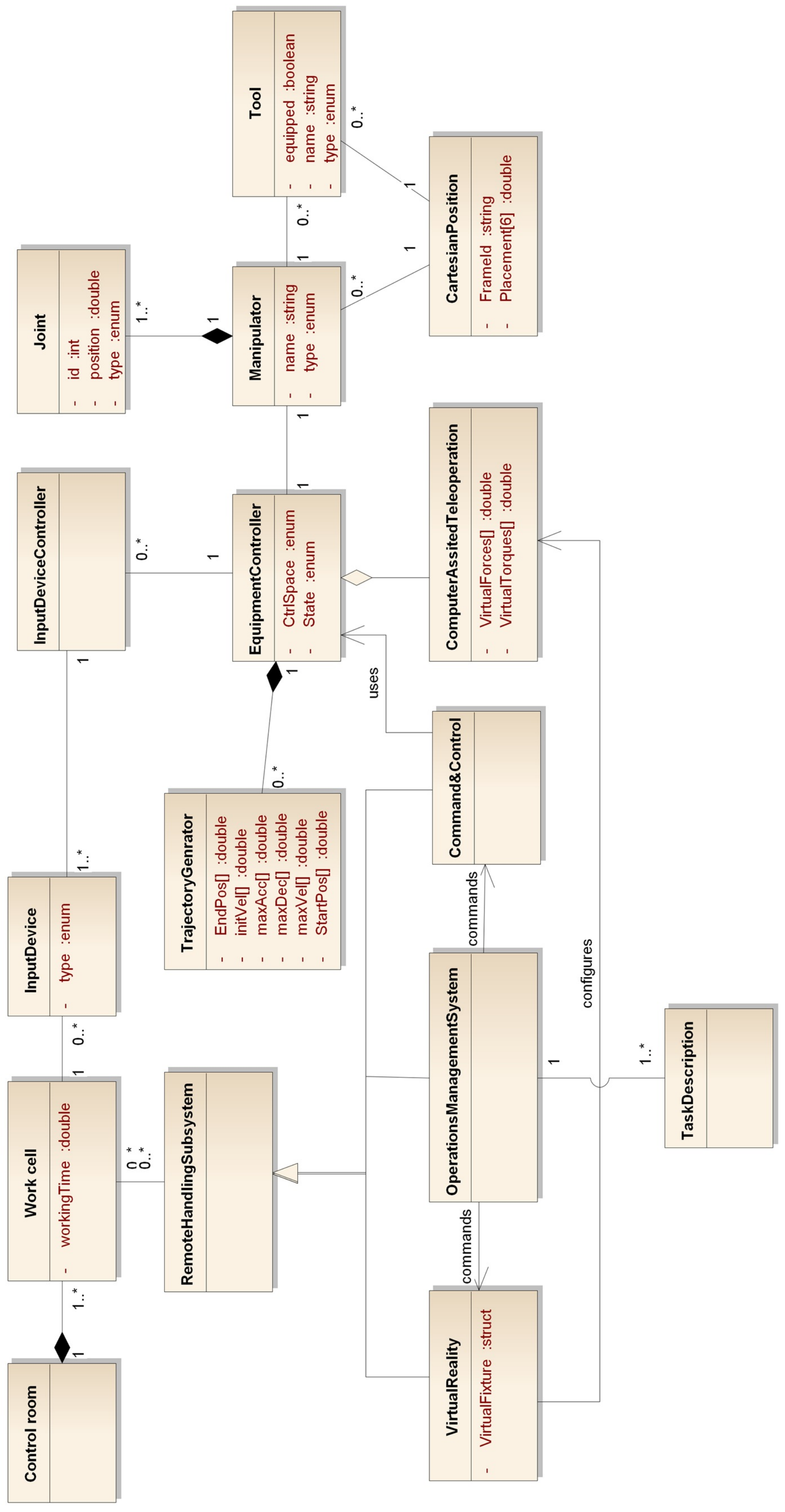


Figure 25. Low-level control system (image courtesy of ITER Organization)

Key design goal for the RHCS is to have no SPoFs, which is typically a requirement for mechanical and electrical designs, since they are more straightforward to design and analyse than software. Combination of concurrent and distributed computing typical for CPSs can introduce seemingly stochastic behaviour and failures. With peer-to-peer middleware and data-centric approach presented in [P5], "no SPoF" approach can be pursued also in software. For example, connecting multiple OMS, VR and other HLCS subsystems to EC is straightforward as there is no point-to-point mapping between these subsystems in the implemented prototype. For the proposed service architecture, service manager can be a local SPoF, although services can also be executed in autonomous mode. Even though the use of peer-to-peer data-centric middleware alone does not remove SPoFs, this design supports reconfiguration flexibility of the system and use of multiple work cells in the control room to operate multiple robots.

Appendix B: Information and Data Models for RHCS



Listing 2 describes the data model used in the LSB (Xenomai queues) and Listing 3 GSB section IDL descriptions for DDS. The GSB IDL module is the same as used in the DTP2, demonstrating reusability of a well-designed information model (and the derived data models) as a basis for communication.

Listing 2. LSB (in C++)

```
Que_tg_command
struct TG_command
{
    int id; // Id has to correspond with existing input
    // triggerType:
    // 0 = sample output, 1 = get scalefactor, 2 = stop profiles,
    // 3 = (NEW!) start timer and write all outputs in a 2 ms loop to rt_queue
    int triggerType;
    double scalefactor;
};
Que_tg_input
struct TG_input
{
    int id;
    double startPos;    // Starting position
    double endPos;      // Desired end position
    double maxVel;      // Maximum velocity (for that joint)
    double initVel;     // Initial velocity
    double maxAcc;      // Maximum acceleration
    double maxDec;      // Maximum deceleration
    bool changed; // Flag to determine if profiling should be reinitialized
};
Que_tg_output
struct TG_output
{
    int id; // 0 = all, 1 = 1st joint, 2 = 2nd joint etc.
    double position[6];
    double speed[6];
    bool finished[6];
};
Que_c4g_joints
struct C4G_joints
{
    double position[6];
};
Que_ref_joints
struct TG_output
{
    int id; // 0 = all, 1 = 1st joint, 2 = 2nd joint etc.
    double position[6];
    double speed[6];
    bool finished[6];
    TG_output() : id(0), position(), speed(), finished() {}
};
```

Listing 3. GSB (Interface Description Language)

```
module Valvomo
{
      struct Location3D
      {
            double x;
            double y;
            double z;
      };
      struct Orientation3D
      {
            double roll;
            double pitch;
            double yaw;
      };
      struct BoxDimensions
      {
            double width;
            double height;
            double depth;
      };
      typedef double RotationMatrix[9];
      struct OmsCommand
      {
            long cmd_id;
            long cmd_type;
            long object_id;
            long parent_id;
            Location3D location;
            Orientation3D orientation;
            boolean flags[5];
            string msg;
      };
      #pragma keylist OmsCommand cmd_id
      struct PointCloud
      {
            long id;
            sequence<Location3D> points;
            boolean enabled;
            double spring;
            double range;
      };
      #pragma keylist PointCloud id
      struct CollisionBoundary
      {
            long id;
            Location3D position;
            RotationMatrix rotation;
            BoxDimensions lengths;
            boolean enabled;
            double spring;
            double damping;
            boolean end_effector;
      };
      #pragma keylist CollisionBoundary id
      struct JointData
      {
            long device_id;
            long jointId[10];
            double jointValue[10];
      };
      #pragma keylist JointData device_id
};
```

Appendix C: Comau Smart NM 45-2.0 Kinematic Parameters and Diagrams

This report outlines the kinematic equations and their solutions necessary for the control of the Comau Smart NM 45-2.0 manipulator. The Smart NM 45-2.0 is a 6 DOF robotic manipulator with 6 revolute joints, including an inline wrist. This manipulator (shown in Figure 26) has a reach of 2000 mm and payload capacity of 45 kg. The actuators of the manipulator are powered by brushless AC motors with encoders for reading motor position. Suitable applications for the manipulator include, for example, handling, assembly and welding.

Kinematics of the manipulator are given based on the Denavit-Hartenberg (D-H) convention described by Sciavicco and Siciliano [151]. Link frames assigned to the manipulator can be seen in Figure 26. The D-H parameters of the manipulator are given in Table 4.

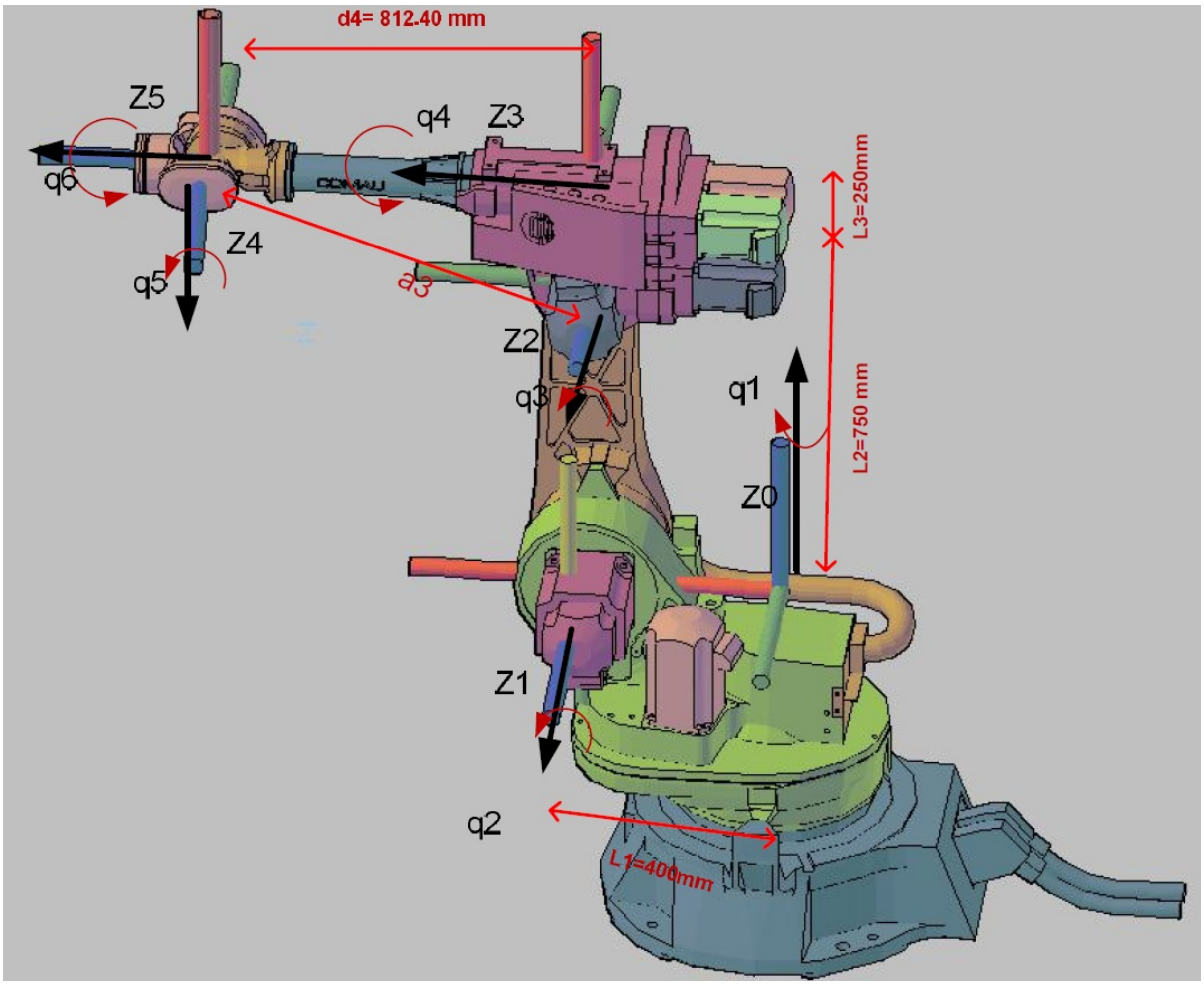


Figure 26. D-H frames and main dimensions of the 6 DOF Comau Smart NM45-2.0 robot

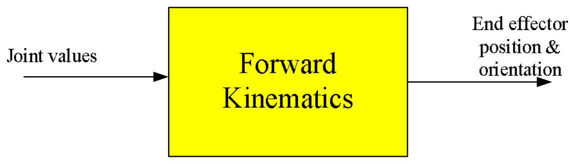
Table 4. Denavit-Hartenberg parameters of the 6 joint manipulator

Link	a_i	α_i	d_i	q_i
1	$L_1 = 0.400\ m$	$\pi/2$	0	q_1
2	$L_2 = 0.750\ m$	0	0	q_2
3	$L_3 = 0.25\ m$	$\pi/2$	0	q_3
4	0	$-\pi/2$	$d_4 = 0.8124\ m$	q_4
5	0	$\pi/2$	0	q_5
6	0	0	0	q_6

The kinematic chain of the manipulator ends at frame {5}, located at the wrist joint. The transformation matrix ${}^{0}_{6}T$ of frame {5} with respect to fixed frame {0} is derived from the above D-H table.

Forward kinematics

The forward kinematics function for the robot calculates the position and orientation of the end-effector as a function of the joint values (q1, q2, q3, q4, q5, q6).



Inputs of the forward kinematics are joint values: q1, q2, q3, q4, q5, q6 (units as radians) and outputs of the forward kinematics are obtained from the transformation matrix ${}^{0}_{6}T$, which is obtained as:

$$ {}^{0}_{6}T = {}^{0}_{1}T \times {}^{1}_{2}T \times {}^{2}_{3}T \times {}^{3}_{4}T \times {}^{4}_{5}T \times {}^{5}_{6}T $$

$$ {}^{0}_{1}T = \begin{bmatrix} \cos(q1) & -\sin(q1) \times \cos(\alpha1) & \sin(q1) \times \sin(\alpha1) & a1 \times \cos(q1) \\ \sin(q1) & \cos(q1) \times \cos(\alpha1) & -\cos(q1) \times \sin(\alpha1) & a1 \times \sin(q1) \\ 0 & \sin(\alpha1) & \cos(\alpha1) & d1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{1}_{2}T = \begin{bmatrix} \cos(q2) & -\sin(q2) \times \cos(\alpha2) & \sin(q2) \times \sin(\alpha2) & a2 \times \cos(q2) \\ \sin(q2) & \cos(q2) \times \cos(\alpha2) & -\cos(q2) \times \sin(\alpha2) & a2 \times \sin(q2) \\ 0 & \sin(\alpha2) & \cos(\alpha2) & d2 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ \vdots $$

$$ {}^{0}_{6}T = {}^{0}_{1}T \times {}^{1}_{2}T \times {}^{2}_{3}T \times {}^{3}_{4}T \times {}^{4}_{5}T \times {}^{5}_{6}T \quad (1) $$

For a given set of joint values (q_1, q_2, q_3, q_4, q_5, q_6), position P_6^0 and orientation R_6^0 of end-effector frame {5}, with respect to base frame {0} are obtained from the transformation matrix ${}_6^0T$ in (1) as follows:

$${}_6^0R = \begin{bmatrix} \cos(\alpha)\cos(\beta)\cos(\delta) - \sin(\alpha)\sin(\gamma) & -\cos(\alpha)\cos(\beta)\sin(\delta) - \sin(\alpha)\sin(\beta) & \cos(\alpha)\sin(\beta) \\ \sin(\alpha)\cos(\beta)\cos(\delta) - \cos(\alpha)\sin(\gamma) & -\sin(\alpha)\cos(\beta)\sin(\delta) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta) \\ -\sin(\beta)\sin(\gamma) & \sin(\beta)\sin(\gamma) & \cos(\beta) \end{bmatrix}$$

$$R(\theta) = R_z(\alpha) \times R_{y'}(\beta) \times R_{z''}(\delta)$$

$${}_6^0P = \begin{bmatrix} {}_6^0Px \\ {}_6^0Py \\ {}_6^0Pz \end{bmatrix}$$

$${}_6^0T = \begin{bmatrix} {}_6^0R & {}_6^0P \\ 0 & 1 \end{bmatrix} \quad (2)$$

Position and orientation of end effector via forward kinematic functions are:

$$P_6^0(x) = {}_6^0T(1{,}4) \quad P_6^0(y) = {}_6^0T(2{,}4) \quad P_6^0(z) = {}_6^0T(3{,}4)$$

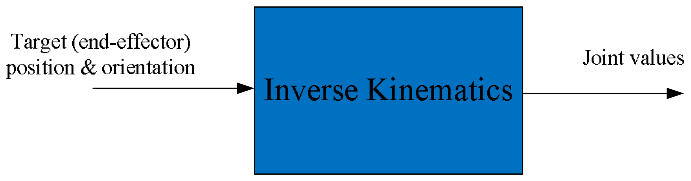
$$R_6^0(\,\theta\mathrm{z}\,) = \operatorname{atan} 2\left({}_6^0T(2{,}3), {}_6^0T(1{,}3)\right)$$

$$R_6^0(\,\beta\ (\theta\mathrm{y}')\,) = \operatorname{atan} 2(\sqrt{\ {}_6^0T(1{,}3)^2 + \ {}_6^0T(2{,}3)}\,, {}_6^0T(3{,}3))$$

$$R_6^0(\gamma\ (\theta\mathrm{z}''\,)) = \operatorname{atan} 2({}_6^0T(3{,}2), -\,{}_6^0T(3{,}1))$$

Inverse kinematics

The inverse kinematics routine provides the joint values (q1, q2, q3, q4, q5, q6) required to achieve a given position and orientation of the end-effector.



Inputs for the inverse kinematics are end-effector position and orientation, in frame {5} with respect to the fixed base frame {0}.

P_6^0: Vector position (x, y, z) of end-effector in frame {5} with respect to frame {0} (units: meters)

R_6^0 : Z (α_z)-Y($\beta_{y\prime}$)-Z($\gamma_{z\prime\prime}$) angles that of frame {5} measured with respect to fixed frame {0}.

Determining anthropomorphic arm position and joint values (q1, q2, q3)

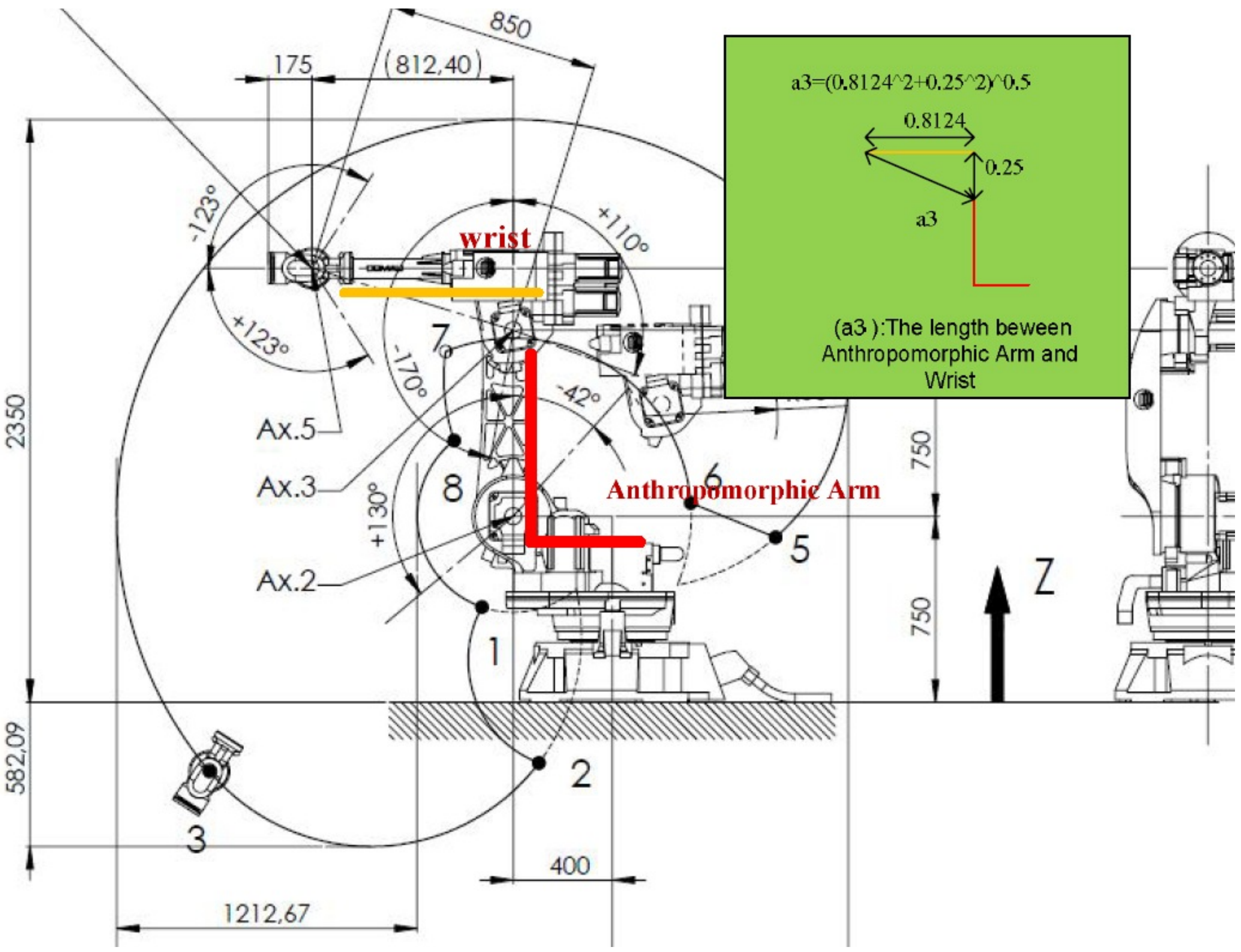
Transformation matrix of the end-effector ${}_6^0T$ in frame {5} is defined by the provided position and orientation targets P_6^0 and R_5^0. Therefore, ${}_6^0T$ is built as in (2).

Position vector *W* of the wrist is defined as:

$$W_x = T_6^0(1.4)\,; \quad W_y = T_6^0(2.4); \quad W_z = T_6^0(3.4) \qquad (3)$$

Joint angle q1 using solution of anthropomorphic arm is determined by W_x and W_y as:

$$q_1 = \operatorname{atan}2(W_y, W_y)$$



The calculation of joint values (q2, q3) is reduced to a planar problem if we set frame {1} as the new origin of coordinates for the end-effector position vector W (see Figure 26):

$$W_x = T_6^0(1.4) - L_1\cos(q_1); \;\; W_y = T_6^0(2.4) - L_1\sin(q_1); \;\; W_z = T_6^0(3.4)$$

The length between frame {2} and frame {3} is:

$$a3 = \sqrt{0.25^2 + 0.8124^2}$$

The solution of the anthropomorphic arm shows:

$$\cos(q_3') = \frac{(P^2{}_{Wx} + P^2{}_{Wy} + P^2{}_{Wz} - L^2{}_2 - a^2{}_3)}{2 \times L_2 \times a_3}$$

$$\sin(q_3') = \mp\sqrt{(1 - \cos(q_3')^2)}$$

$$q3 = atan2(sin(q_3'), cos(q_3')) + \ atan2(812.4{,}250)$$

$$\sin(q_2) = \frac{(L_{2+}a_3 \times \cos(q_3')) \times P_{Wz} - a_3 \times \sin(q_3') \times \sqrt{P^2{}_{Wx} + P^2{}_{Wy}}}{P^2{}_{Wx} + P^2{}_{Wy} + P^2{}_{Wz}}$$

$$\cos(q_2) = \frac{(L_{2+}a_3 \times \cos(q_3')) \times \sqrt{P^2{}_{Wx} + P^2{}_{Wy}} + a_3 \times \sin(q_3') \times P_{Wz}}{P^2{}_{Wx} + P^2{}_{Wy} + P^2{}_{Wz}}$$

$$q2 = atan2(sin(q_2), cos(q_2))$$

Determining spherical wrist position and joint values (q3, q4, q5)

Consider the spherical wrist in Figure 26, whose direct kinematic was given in (2). It is desired to find the joint variable (q4, q5, q6) corresponding to a given end effector orientation (${}^3_6R$). These angles constitute a set of input Euler angles ZYZ (α, β, δ) with respect to frame {3}.

The whole rotation matrix is:

$$R(\theta) = R_z(\alpha) \times R_{y'}(\beta) \times R_{z''}(\delta)$$

$${}^0_6R = \begin{bmatrix} \cos(\alpha)\cos(\beta)\cos(\delta) - \sin(\alpha)\sin(\gamma) & -\cos(\alpha)\cos(\beta)\sin(\delta) - \sin(\alpha)\sin(\beta) & \cos(\alpha)\sin(\beta) \\ \sin(\alpha)\cos(\beta)\cos(\delta) - \cos(\alpha)\sin(\gamma) & -\sin(\alpha)\cos(\beta)\sin(\delta) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta) \\ -\sin(\beta)\sin(\gamma) & \sin(\beta)\sin(\gamma) & \cos(\beta) \end{bmatrix}$$

Also

$${}^0_6P = \begin{bmatrix} {}^0_6Px \\ {}^0_6Py \\ {}^0_6Pz \end{bmatrix}$$

$${}^0_6T = \begin{bmatrix} {}^0_6R & {}^0_6P \\ 0 & 1 \end{bmatrix}$$

Also computing the ${}^0_3T$ using calculated values q1, q2, q3 yields:

$${}^0_1T = \begin{bmatrix} \cos(q1) & -\sin(q1) \times \cos(\alpha1) & \sin(q1) \times \sin(\alpha1) & a1 \times \cos(q1) \\ \sin(q1) & \cos(q1) \times \cos(\alpha1) & -\cos(q1) \times \sin(\alpha1) & a1 \times \sin(q1) \\ 0 & \sin(\alpha1) & \cos(\alpha1) & d1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\vdots$$

$${}^0_3T = {}^0_1T \times {}^1_2T \times {}^2_3T$$

Hence having computed the rotation matrix:

$${}^{0}_{6}T = {}^{0}_{3}T \times {}^{3}_{6}T$$

$${}^{3}_{6}T = inv({}^{0}_{3}T) \times {}^{0}_{6}T$$

q4, q5, q6 are obtained by extracting the ZYZ-Euler angles from ${}^{3}_{6}T$:

$$q_4 = \operatorname{atan2}({}^{3}_{6}T(2,3),\ {}^{3}_{6}T(1,3))$$

$$q_5 = \operatorname{atan2}\left(\sqrt{{}^{3}_{6}T(1,3)^2 + {}^{3}_{6}T(2,3)^2}\ ,\ {}^{3}_{6}T(3,3)\right)$$

$$q_6 = \operatorname{atan2}({}^{3}_{6}T(3,2), -{}^{3}_{6}T(3,1))$$

C4G Open control modes

C4G Open is a technology developed by Comau that allows an external PC to communicate in real-time with the robot control and interact at different levels in the robot control (servo system) and in the trajectory generation. The external contribution collaborates with, adds to or substitutes the internal control.

C4G Open provides the user possibility to select control mode from several different options that are intended for different purposes. They are different combinations of PC contributing in different ways to joint velocity, position or current (or a combination of these). The modes with explanations of the intended purposes are listed in Table 5.

Table 5. C4G Open control modes

Mode	PC role	SMP+[6] role	Purpose / notes
0	-	target and position and velocity loop	Default mode.
0′	target and position and velocity loop	target and position and velocity loop	Debugging.
1	velocity and current loop	-	Gives the external PC full control.
2	velocity and current (motion control)	position target (trajectory generation)	Allows evaluating the coefficients for the control loops on the external PC, while SMP+ takes care of the trajectory generation.
3	velocity control (current contribution)	velocity target (trajectory generation)	Not currently available.
4	absolute position target (trajectory	position and velocity loop (motion	PC directly generates the position referenced trajectory.

[6] SMP+ is the control board where high level processing such user program interpretation, HMI management and trajectory generation take place.

	generation)	control)	
5	relative position target (trajectory generation)	position and velocity loop (motion control)	PC indirectly generates the position Referenced trajectory.
6	target velocity (velocity referenced trajectory generation)	velocity loop (motion control)	Not currently available.
7	position additional contribution	target and position and velocity loop	It is the modality allowing PC to supply the trajectory (generated by SMP+) with an additional contribution.
8	velocity additional contribute	target and position and velocity loop	Not currently available.
9	current additional contribute	target and position and velocity loop	It is the modality allowing adding a current contribution calculated by external PC, while SMP+ generates the trajectory.

Out of the currently available modes, suitable modes for external motion control and trajectory generation seem to be modes 4 and 5.

- Mode 1 seems to be too unstable and/or difficult to use, since PC is required to provide all three of position, velocity and current references. Execution of TestMode1 (a test application provided with the C4G Open Library) resulted in uncontrolled vibration and subsequent shutdown of the communication link.
- Mode 2 can also be used for controlling joint currents and velocity, although it is intended for simple control loops since SMP+ is used for trajectory generation in this mode.
- Another alternative would be to use mode 3 in which PC is used for current contribution, but this mode is currently not available. Note that current-based control would likely be the most difficult mode to control.
- In modes 4 and 5 the PC supplies SMP+ with target position (as motor turns) and target velocity (delta motor turns). Since mode 5 is intended to be used in a configuration in which PC adds its position & velocity values on top of the SMP+ pos. & vel. values, mode 4 seems to be the more appropriate choice for PC-based motion control.
- Mode 6 could be used for velocity control, but it is not currently available. This mode could be the most user-friendly if it would be implemented.

To summarize, mode 4 is the best choice for generic PC-based control purposes.

Publications

Publication I

P. Alho and J. Mattila, Dependable Control Systems Design and Evaluation, Ninth Annual Conference on Systems Engineering Research, 2011, Redondo Beach, USA

© INCOSE-LA. Reprinted with permission.

Dependable Control Systems Design and Evaluation

Pekka Alho
Tampere University of Technology,
Department of Intelligent Hydraulics and Automation, Finland
pekka.alho@tut.fi

Jouni Mattila
Tampere University of Technology,
Department of Intelligent Hydraulics and Automation, Finland
jouni.mattila@tut.fi

Abstract

Remote handling (RH) is a key technology in the ITER fusion reactor. The controller systems used for performing mission-critical RH operations need to be dependable, as the fundamental requirement for the ITER RH system is a fail-safe and recoverable design. Additional design challenges include interoperability with systems and platform independence during ITER life cycle. Contributions are especially needed for development of cost-effective systems engineering (SE) practices and guidelines for fault-tolerant implementation. This paper addresses the issues by presenting a survey of industrial best practices and different fault prevention, tolerance, removal and forecasting methods. Based on the results, key findings to achieve dependable and cost efficient design include development a SE framework that supports reuse of components, models and analysis results; non-redundant fault tolerance; and use of commercial off-the-shelf hardware, operating systems and communication middleware.

1 Introduction

ITER is an experimental nuclear fusion reactor, currently under construction in Cadarache, France and planned to start operations in 2018. The ITER machine operation is based on remote handling (RH) maintenance systems that enable the operators to safely, reliably and repeatedly perform robotic manipulation of items without being in contact with those items. This paper focuses on systems engineering (SE) development process of dependable RH control systems that perform mission-critical operations in this demanding environment and presents objectives of the current research. The research is part of a PhD thesis topic in the Goal Oriented Training Program on Remote Handling (GOT-RH) managed by European Fusion Development Agreement (EFDA). GOT-RH aim is to train engineers for activities to support ITER project. The research in this paper combines SE and dependability approaches to fulfil the ITER RH control system requirements in a cost-efficient manner.

Fig. 1. Divertor Test Platform 2.

A major objective of the ITER project is to demonstrate that a fusion energy device can be maintained efficiently so that the plant availability is retained at sufficient level. During the ITER lifetime reactor components must be inspected and maintained, including replacement of the 9 tonne divertor components. Reactor operation produces high energy neutrons which are absorbed by components inside the reactor vessel, leaving them beta and gamma activated. Therefore RH has to be utilized to perform maintenance tasks instead of manual operations, as there is no human access into reactor. To test the proof-of-concept designs for the replacement of a divertor, a full scale prototype environment, designated 'Divertor Test Platform 2' (Fig. 1), is operational at Tampere, Finland. The facility is hosted by VTT and Tampere University of Technology, Department of Intelligent Hydraulics and Automation (TUT/IHA). TUT/IHA has worked with ITER RH since 1994 developing the ITER divertor maintenance, processes, tools and equipment.

Commonly used fault-tolerance techniques employ redundancy in order to improve reliability, but usually require significant amounts of resources. Use of these techniques is mandatory in safety-critical systems that need to keep operating regardless of failures, such as flight-control or fission reactor management (*fail-operate*), but in systems that can be guided to a safe state (*fail-safe*) the additional costs are more difficult to justify, therefore a balanced solution is needed. A key difference between ITER and fission reactors is that the energy density in ITER reactor cannot cause a catastrophic failure, but the economic losses in the case of an operation failure could be significant nevertheless. Thus the RH system is *safety-critical* and the design of a RH control system must be fail-safe or capable of operating in a *limp-home mode*, which is a form of fail-operational system.

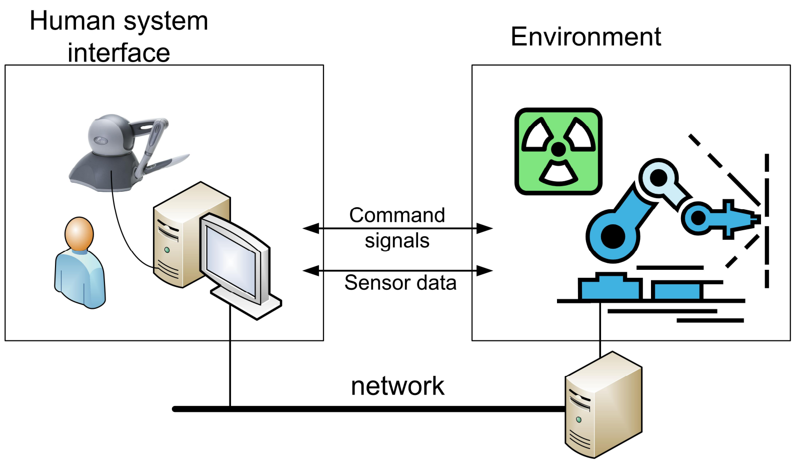


Fig. 2. Bilateral teleoperation system.

Application for our RH control system is a teleoperated bilateral master-slave manipulator system, where the operator controls a remote manipulator working in a hazardous environment (see Fig. 2). The fundamentals of implementing such systems are well-known, with commercial manipulators and components available. Challenges with developing and using such systems in ITER are – in addition to aforementioned environment, dependability, etc. – need for interoperability and platform independency during ITER life cycle. As a whole, ITER RH aims to have one master system which is used to control several heterogeneous slave systems that perform various maintenance tasks, provided by different subcontractors. All these must be able to work harmoniously, regardless of changes to other systems and technology upgrades.

This paper includes a survey of industrial best practices developed by researchers in organizations like IEC, NASA, etc. and compares them against ITER requirements. Our overall target for EFDA GOT-RH is to propose a subset of a generic lean-minded SE framework to support reuse of artifacts (hardware, software, processes, models, etc.) suitable for ITER. Additionally we seek a system design that avoids extensively redundant and tightly coupled solutions. A proof-of-concept implementation of the architecture for Fig. 2 system is being currently developed.

In the next chapter we introduce research background, starting with dependability terminology and then covering related research and the research problem. In chapter 3 we examine how standards together with industry best practices and cost-efficiency affect the development process when compared to ITER requirements. Chapter 4 approaches the problem through systems development process, divided into specification, design and architecture, implementation and evaluation. Finally the conclusions are presented in chapter 5.

2 Background

In the following sections basic dependability concepts, state of the art in dependable systems and research problem are briefly reviewed.

2.1 Dependability

According to Avizienis et al. *dependability* is defined as the ability to deliver service that can justifiably be trusted. It is an umbrella term that consists of several attributes: availability, reliability, safety, integrity and maintainability. Researchers and the ITER requirements emphasize especially safety and reliability. However, all attributes need to be addressed in order to ensure delivery of the correct service – therefore the SE approach is necessary to manage all dependability-related design aspects. *Failures* are events where the delivered service deviates from the correct service. The deviations are called errors and the cause for the error is defined as a fault. It should be noted that not all errors lead to service failures – this depends from the structure and behaviour of the system. (Avizienis et al. 2004). As shown in Fig. 3, service failures can cause new faults for other systems.

$$\begin{tikzcd}
\dots \arrow[r] & \text{fault} \arrow[r, "\text{activation}"] & \text{error} \arrow[r, "\text{propagation}"] & \text{failure} \arrow[r, "\text{causation}"] & \text{fault} \arrow[r] & \dots
\end{tikzcd}$$

Fig. 3. Error propagation (Avizienis et al. 2004).

Applications that have dependability requirements can be categorized as fail-safe or fail-operate, depending from if the system can be brought into a safe state or whether it needs to continue operation in the presence of the faults. (Avizienis et al. 2004). Different means used to attain dependability can be categorized as *fault prevention*, *removal*, *tolerance* and *forecasting* techniques. Fault tolerance techniques, i.e. avoiding service failures in the presence of faults, can address one or more of the following stages of tolerating faults: error detection, damage assessment, and recovery and continued service. A review of different techniques can be found e.g. from NASA report Software Fault Tolerance: A Tutorial (Torres-Pomales 2000). With software (SW) systems duplication of modules replicates errors as well, so redundant components need to be diverse. Even though the development costs of N-variant software are less than N times non-fault-tolerant software (Laprie et al. 1990), it still presents a major cost increase for the development when compared to basic software or single version fault tolerance techniques.

2.2 Related work

Increase of complexity and amount of requirements for modern software systems present us the problem of how to attain and estimate the dependability of these complex systems. Another problem is related to interoperability of systems with long expected lifetimes. E.g. U.S. Navy intentionally sank the Aegis cruiser Valley Forge after 18 years of service – intended service life had been at least three decades but the integration costs of new software and weapon systems were too high (Schneider 2010). Clearly building stovepipe systems, i.e. complex single-purpose ‘soon-to-be-legacy’ systems that consist of inter-related and tightly bound elements, is not a viable solution. ITER will have several subcontractors providing software and has an expected life span of several decades, so integration of distributed real-time systems is a critical design factor.

In software systems service failures can create new faults via causation. To achieve no-single-points-of-failure goal in a software unit, we would need redundancy (Flammini 2010), (Hayama et al. 2010), which again increases development costs (Laprie et al. 1990). In fail-safe systems or systems using graceful degradation, structuring can be used to limit failures inside the architectural unit. If software safety, i.e. execution within system context without contributing to hazards, is considered more important than reliability, i.e. low mean time between failures, then fault tolerance techniques can concentrate on preventing catastrophic failures. Single version fault tolerance techniques are cost-efficient way to achieve safety with possible compromises to reliability when compared to multiversion techniques. Examples of single version fault tolerance include system structuring, atomic actions, error detection and exception handling, among others (Torres-Pomales 2000).

Dependable architecture designs and fault tolerant control methods tend to be too specific to be reusable, at least outside their application domain. There has been some earlier research on generic architectures for real-time dependable systems, e.g. architecture model developed by Powell et al. (Powell et al. 1999) uses fault containment to deal with faults and is based on the use of software components. The focus for our research is not in producing architectural design patterns or domain specific solutions to achieve dependability.

Some architectures have been developed to support use of commercial off-the-shelf (COTS) components (Powell et al. 1999), (Asterio et al. 2003), but reuse needs to be carefully considered to evaluate if possible cost benefits outweigh compromising effects on dependability and possible needs for additional fault tolerance. Based on experiences with the older experimental reactors, COTS components could be used to implement some parts of the control system in ITER; for example, in JET (Joint European Torus) results were positive with implementing the highly critical motion control, and integrating into a uniform control system framework. For fault tolerance JET employed a large number of error checks, which is one of the basic single-version fault tolerance techniques. For severe errors the system was put into safe-state by cutting all power, engaging brakes and opening emergency stop circuit. (Haist & Hamilton 2001).

2.3 Research problem

ITER organization promotes a standardized software-module based design approach and has an equipment controller (EC) architecture draft to improve cooperation of RH systems and higher level systems being developed by several different contractors. However, this architecture only outlines basic features. In addition to standard external safety features, like emergency stops, it does neither provide nor dictate solutions for fault tolerance.

In Fig. 4 area one presents *embedded* hard real-time systems, i.e. ‘standard’ solution to implementing controllers, and area two presents hard real-time systems implemented using *commercial PCs and RT operating systems* (OS) which is rarer alternative (Flammini 2010). Our purpose is to use industrial PCs to test their feasibility in implementing dependable RH systems

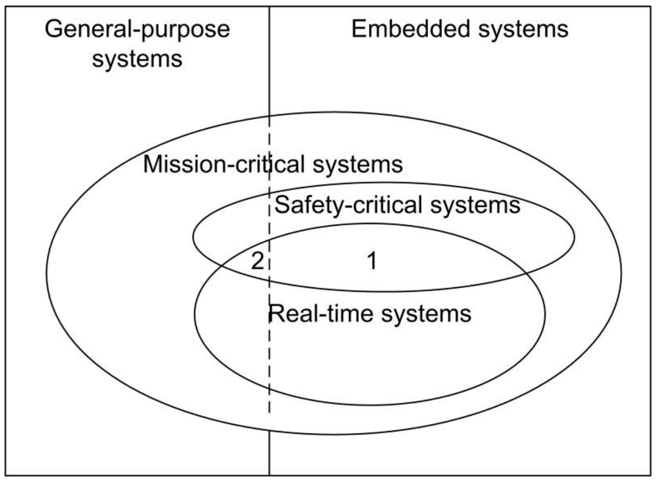


Fig. 2. A classification of critical computer systems, adapted from (Flammini 2010).

with strict performance requirements. If the use of general-purpose systems proves to be a viable alternative, it could be one way to reduce development costs for systems with dependability requirements.

Especially interesting from the perspective of using open source or commercial real-time operating systems (RTOS) in safety-critical applications is the report (Bishop et al. 2001) for Health and Safety Executive: Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications. The report considers the safety assurance of SOUP in the context of IEC61508. Similar evaluation methods could be used with ITER RH systems.

In dependability research the fault-tolerant approach is often promoted over fault prevention. The reasoning behind this being that all faults cannot be prevented or removed, so it is better to concentrate on fault tolerance methods (Elder 2001); or fault prevention is shrugged off being part of general engineering (Avizienis et al. 2004). However, neglecting of fault prevention is short-sighted. First, no single mechanism can cope with all faults and anticipation of unexpected faults can increase costs. Second, the cost of finding and removing faults typically rises by development phase and finally, faulty specifications are major cause of software faults (Avizienis et al. 2004). Hence, additional research is needed on use of fault prevention to minimize the number of faults in the system as early as possible with optimal development methods. Furthermore, as scientific papers usually focus in one or two of the strategies used to achieve dependability, there is a need to bridge the gaps between the different models, methods, and tools that are used to improve the design and the operation of dependable systems, especially when being adapted to control systems (Bondavalli et al. 2001).

3 Standards and best practices

IEC 61508 is an international safety standard related to functional safety of electric/electrical/programmable systems with part three related to software requirements (IEC 2010). Most of European standards for *safety related* control systems refer to IEC 61508, if the implementation language is C or similar. ITER RH control system will include safety-related systems and some of the safety functions could be implemented by software – this would be safety-related software. Safety standards introduce concept of safety integrity level (SIL) (or performance level, PL), used to present risk reduction offered by safety functions. To achieve target SIL levels, standards include recommendations and requirements. However, especially software systems have the problem that SIL estimation is difficult because of systems complexity.

Even though this paper mostly refers to IEC61508 standard because of its suitable scope and internationality, there are a number of other well-known standards that can be used to contribute to system dependability development and evaluation. Software testing has its own standard, IEEE 829-1998 and some of the American internationally recognized standards include e.g. ISA 84 series and MIL-STD-882D.

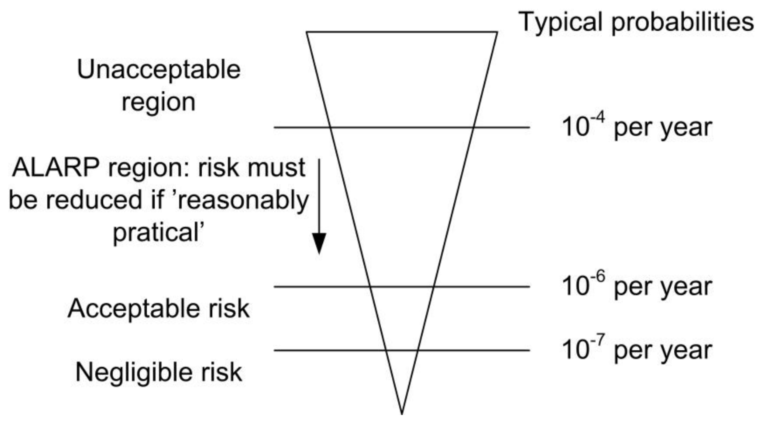


Fig. 3. As low as reasonably practicable (ALARP) (Melchers 2001).

One approach to achieving a tolerable risk is 'as low as reasonably practicable' (ALARP), mentioned e.g. in IEC61508 standard. If the evaluated risk is smaller than 'must be refused altogether', but larger than 'insignificant', ALARP principle together with a cost benefit assessment can be used to determine areas where risks need to be decreased, as shown in Fig. 5. Where the risks are less significant, the fewer resources are needed to be spent to reduce them and vice versa. ALARP is one of the principles used in ITER, and in nuclear project designs in general.

Component reuse and use of commercial off-the-shelf (COTS) components show some promise for achieving cost reductions in development. Especially use of commercial hardware components gives the benefit of utilizing performance and energy efficiency of cutting edge processor technology. Software component reuse has more problems related to it, as there is usually no guarantee that the components have sufficient quality for mission-critical applications and may require additional fault tolerance. Instead, use of commercial OSs has the same potential benefits as hardware, i.e. they include the latest developments in OS technology and have potentially better quality and less bugs than custom made software because of widespread use. For example, QNX Neutrino RTOS kernel has been certified to confirm to IEC61508 at SIL3 (Hobbs 2010) out of maximum level of 4. Even though implementing hard real-time systems using commercial PCs and real-time OSs is still fairly rare, this could be an interesting development path to cost-efficient and dependable systems.

Finally, design patterns are reusable general solutions that present best practice knowledge. They can and should be used to improve fault tolerance, as fault tolerance has patterns of its own – a classic example is the watch dog pattern (Hanmer 2007). However, possible design pattern use must be traceable to requirements and patterns should not be introduced without good reasons as they can add unnecessary complexity to system. A good general rule for architecture design is to keep it as simple as possible, especially for safety-critical components.

4 Development process for dependability

Development process should combine all possible methods – fault prevention, fault tolerance, fault removal and fault forecasting – to achieve sufficient level of dependability with optimal resource use by combining different methods, according to ALARP principles. As stated earlier, most studies focus on one or two methods and do their research within this limited scope, whereas efficient approach would be to combine all different approaches. The role of the development process is similar to quality assurance, i.e. reducing mistakes made by developers and ensuring product quality.

Typically V model, waterfall and other software life cycle models describe only development process (see Fig. 6). However, system life cycle also includes installation, operation, and maintenance. For safety-critical software it is important to also take these phases into consideration to ensure maintainability and interoperability, because of the higher development costs and consequently longer expected life time of the system.

The development process considerations presented in this chapter combine best practices and recommendations and discuss them in the ITER RH context. The process analysis is divided into system definition, design and architecture, implementation, and evaluation.

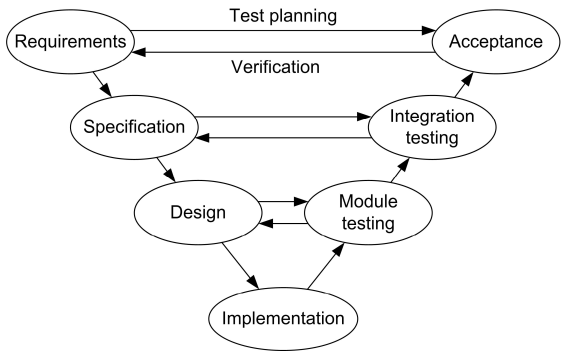


Fig. 4. Life cycle V model.

4.1 Specification

Specification, which in this case is considered to cover system analysis and definition, including hardware and software, has significant role in *fault prevention*. It is a well-known fact that errors made in the requirement specification phase of software cause more problems than coding errors (Pullum 2001). Requirements come from multiple sources and usually change as the project moves on, and the development process should offer support for this. According to Pikkarainen, use of agile methods and practices improved communication and management of requirements, features and project task dependencies (Pikkarainen 2008). However, agile methods are not necessarily suited for development of safety-critical software as such and may need additional emphasis on documentation, architectural design and traceability. IEC 61508 part 7 has a list of development methods IEC considers suitable for safety-related software.

Safety requirements are especially interesting from the dependability point of view, as they contain information about what the system is allowed and not allowed to do, as software should have indications and contraindications especially if reuse is planned for components. In most systems there are many opportunities to enhance safety, e.g. by simple value checks, but often they are not used. Safety requirements could be used to document possible values that can be used for safety checks, e.g. humidity, dust, vibration etc. (Herrmann 1999).

Hazard/risk identification and analysis should always be carried out for safety-critical systems, preceding the finalization of system requirements (Douglass 1999). Risk probability estimates can be made early, even before committing resources to hardware or software (Dunn 2002). Normal methods for risk analysis include fault modes, effects and criticality analysis (FMEA or FMECA), fault tree analysis (FTA) and risk analysis (RA). First two are qualitative and RA is quantitative (Dunn 2003). IEC61508 standard presents risk graph and hazardous event severity matrix as qualitative methods for determination of SILs.

Pre-design hazard identification and measuring reliability of existing system have some shared methods, e.g. FTA. Hazard identification requires significant resources and participation of different shareholders to gain accurate and useful results so reuse of methods and previous results should be considered for cost benefits. For ITER RH systems previously done risk analyses include e.g. manipulator FMECA and FTA for rescue of failed in-cask equipment.

4.2 Design and architecture

High-level design is the realization of quality requirements. It is also the first concrete form of the system that can be analyzed and tested. Architectural choices impact software attributes like availability, security etc., so the chosen architecture should support dependability requirements with appropriate *fault tolerance* techniques and patterns (Bass & Clements 2003). In addition to using these techniques, the actual system architecture should also be designed as fault-tolerant (e.g. with layering and error confinement areas). Safety-critical, safety-related and nonsafety-related software components should be isolated by partitioning the software (Herrmann 1999). Risk probability estimates for components can be made early and designs changed before actual commitments to hardware and software are made.

For ITER RH an initial version of the reference architecture has been developed, and after implementation it will be used as a testing platform for evaluating dependability of PC-based control systems and different fault tolerance methods. Possible fault tolerance components will combine patterns and COTS solutions, including QoS manager**,** network middleware, partitioning of architecture, use of heartbeat/watchdog and more. Project will make use of existing knowledge, hardware and software components etc. to maximum reuse of artifacts across different RH systems and projects.

4.3 Implementation

Implementation methods are generally dictated by development process (iterative, agile etc.), which defines the routines and support tools used in the project. Use of implementation-related methods to ensure dependability of the product is also part of fault prevention. Following good practices and programming methods can prevent generation of faults, e.g. NASA Software Safety Guidebook (NASA 2004) has comprehensively listed principles for implementation of real-time software.

4.4 Evaluation

Evaluation of software-based systems includes traditional software testing, V&V methods, and formal inspections etc., which are considered *fault removal* techniques from the dependability point of view. Another way to approach evaluation is from the *fault forecasting* standpoint by estimating or predicting reliability of the system. Component reliability is an important quality measure for system level analysis, but software reliability is hard to characterize. Post-verification reliability estimates remains a controversial issue (Torres-Pomales 2000).

In a sense, evaluation is risk control – evaluation of the software includes mishap risk assessment of the current implementation (‘is this safe enough?’) and finally acceptance. Risk assessment can be supported with data from testing, e.g. detected and corrected defects, and forecast results. Together these can be used to decide whether additional testing and fault-tolerance techniques are needed or if the product is ‘good enough’ to be finished.

Targets for evaluation include should be on all levels, including requirements, architecture, source code, software units and the complete system. In addition to evaluating the system under development, evaluation can also be done for organization before the project has been started, based on e.g. the Capability Maturity Model Integration (CMMI). Architecture evaluation methods like Architecture Tradeoff Analysis Method (ATAM) are used to determine if the architecture enables realization of key scenarios and identifying of potential risks (Grimán et al. 2007).

5 Conclusions

The research carried out in this paper has compared control system design against the industrial and scientific best practices developed for safety-critical applications. The paper presents the results phase-by-phase according to the SE process in Fig. 6. Considered viewpoints include balancing of requirements (especially dependability vs. cost) and design artefact reuse. Based on the analysis in this paper, the development process for dependable control system design and evaluation has to focus in the following:

1) Fault tolerance based on non-redundancy, i.e. single-version fault tolerance. Most of control systems are not required to be fail operate, so some compromises can be made in achieving dependability cost efficiently, like implementing fail-safe system with single version fault tolerance and reuse of components, as per ALARP principles. Standards for safety-related systems have recommendations about the use of diverse programming techniques (N-version redundancy), but e.g. in IEC61508-6 even on SIL3 they are still only 'recommended'.
2) Avoiding stovepipe systems, i.e. building large custom software, instead developing systems based on well-tested COTS or open source communication middleware, OSs and hardware to maximize interoperability, dependability and cost-efficiency. Only business-critical SW components should be custom built.
3) SE framework that covers requirements, architecture, design and evaluation to support reuse of software and hardware components, processes, models and analysis results (e.g. FMECA and FTA).

The next phase of this research consists of developing a proof-of-concept implementation of the dependable control system design for the bilateral master-slave teleoperation system presented in Fig. 2, utilizing the developed SE framework. After this the overall target is to V&V and propose a subset of generic lean SE framework, such as design models, processes, hardware & software modules, suitable for ITER RH systems.

6 References

Asterio, P., Guerra, C., Mary, C., & Rubira, F., "A fault-tolerant software architecture for COTS-based software systems." *Proceedings of the Joint European Software Engineering Conference (ESEC) and 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)* ACM Press, pp. 375-382, 2003.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C., "Basic concpets and taxonomy of dependable and secure computing." *Transactions on dependable and secure computing , 1* (1). 2004.

Bass, L., & Clements, P., *Software architecture in practice.* Addison Wesley, 2003.

Bondavalli, A., Fantechi, A., Latella, D., & Simoncini, L., "Design validation of embedded dependable systems." *Micro, IEEE , 21* (5). 2001.

Douglass, B., *Doing hard time: developing real-time systems using UML, objects, frameworks, and patterns.* Addison-Wesley, 1999.

Dunn, W., "Designing safety-critical computer systems." *Computer , 36* (11), pp 40-46, 2003.

Dunn, W., *Practical Design of Safety-Critical Computer Systems.* Reliability Press, 2002.

Elder, M., *Fault Tolerance in Critical Information Systems.* University of Virginia, 2001.

Flammini, F., *Dependability assurance of real-time embedded control systems.* New York: Nova Science Publishers, 2010.

Grimán, A., Pérez, M., Mendoza, L., & Méndez, E., "A method proposal for architectural reliability evaluation." *ICEIS 2007 - Proceedings of the Ninth International Conference on Enterprise Information Systems*, pp. 564-568, 2007.

Haist, B., & Hamilton, D., "A rational approach to remote handling equipment control system design." *9th ANS International Topical Meeting on Robotics and Remote Systems.* EFDA, Seattle, 2001.

Hanmer, R., *Patterns for fault tolerant software.* Wiley, 2007.

Hayama, R., Higashi, M., Kawahara, S., Nakano, S., & Kumamoto, H., "Fault-tolerant automobile steering based on diversity of steer-by-wire, braking and acceleration." *Reliability Engineering and System Safety , 95*, pp 10-17, 2010.

Herrmann, D., *Software Safety and Reliability.* IEEE, 1999.

Hobbs, C., *Using and IEC 61508-Certified RTOS Kernel for Safety-Critical Systems.* QNX, 2010.

IEC, *IEC 61508 Edition 2.0.* 2010.

Laprie, J.-C., Arlat, J., Beounes, C., & Kanoun, K., "Definition and analysis of hardware- and software-fault-tolerant architectures." *Computer , 23* (7), pp 39-51, 1990.

Melchers, R., On the ALARP approach to risk management. *Reliability Engineering & System Safety , 71* (2), 2001.

NASA, *NASA Software Safety Guidebook.* 2004.

Pikkarainen, M., *Towards a framework for improving software development process mediated with CMMI goals and agile practices.* VTT Publications, Espoo, 2008.

Powell, D., Arlat, J., Beus-Dukic, L., Bondavalli, A., Coppola, P., Fantechi, A., et al., "GUARDS: a generic upgradable architecture for real-time dependable systems." *Transactions on Parallel and Distributed Systems , 10* (6), 1999.

Pullum, L., *Software fault tolerance techniques and implementation.* Artech House, 2001.

Schneider, S., *What is Real-Time SOA.* RTI, 2010.

Torres-Pomales, W., *Software fault tolerance: a tutorial.* NASA, 2000.

7 Biography

M.Sc. **Pekka Alho** is a researcher and a doctoral candidate at TUT/IHA working with ITER RH control systems in EFDA's GOT-RH trainee program. He graduated with M.Sc. (Eng) in Oct 2009 from TUT. Earlier in 2007-2010 he worked first as research assistant and then as a researcher at Dept. of Automation Science and Engineering.

Professor, Dr. Tech. **Jouni Mattila** received M.Sc. (Eng.) in 1995 and Dr. Tech 2000 both from TUT. He is a TUT program manager in ITER-DTP2-projects on Remote Handling. He is a coordinator of EFDA GOT-RH trainee program with 10 trainees. His research interests include machine automation and preventive maintenance, and fault-tolerant control system development for advanced machines utilizing lean systems engineering framework.

This work supported by European Communities was carried within the framework of EFDA and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of European Commission.

Publication 2

P. Alho and J. Mattila, Breaking down the requirements: Reliability in remote handling software, Fusion Engineering and Design, vol. 88, no. 9-10, pp. 1912–1915, 2012

DOI: 10.1016/j.fusengdes.2012.11.008

© Elsevier. Reprinted with permission.

Fusion Engineering and Design 88 (2013) 1912–1915

Contents lists available at ScienceDirect

Fusion Engineering and Design

journal homepage: www.elsevier.com/locate/fusengdes

Breaking down the requirements: Reliability in remote handling software

Pekka Alho*, Jouni Mattila

Department of Intelligent Hydraulics and Automation, Tampere University of Technology, Finland

HIGHLIGHTS

- ▶ We develop a set of generic recommendations for control system software requirements.
- ▶ We analyze ITER remote handling system requirements.
- ▶ Requirement specifications have major impact on software reliability.
- ▶ Reliability requirements need to be managed as a system measure.
- ▶ Systematically developed requirements can be used to form a dependability case.

ARTICLE INFO

Article history:
Received 13 September 2012
Received in revised form 22 October 2012
Accepted 7 November 2012
Available online 4 December 2012

Keywords:
remote handling, control system, software, requirements, reliability, dependability

ABSTRACT

Software requirements have an important role in achieving reliability for operational systems like remote handling: requirements are the basis for architectural design decisions and also the main cause of defects in high quality software. We analyze related recommendations and requirements given in software safety standards, handbooks etc. and apply them to remote handling control systems, which typically have safety-critical functionality, but are not actual safety-systems–for example the safety-systems in ITER will be hardware-based.

Based on the analysis, we develop a set of generic recommendations for control system software requirements, including quality attributes, software fault tolerance, and safety and as an example we analyze ITER remote handling system software requirements to identify and present dependability requirements in a useful manner. Based on the analysis, we divide a high-level control system into safety-critical and non-safety-critical subsystems, and give examples of requirements that support building a dependable system.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

ITER will feature a large number of remote handling (RH) systems, including divertor, blanket, port handling, viewing, neutral beam, transfer cask and hot cell. Proper maintenance and operation of ITER is not possible without these systems, and their reliable operation is necessary for ensuring that the plant is available for fusion experiments. Achieving this goal requires reliable mechanical components and designs, together with a suitable maintenance strategy. However, software failures have passed hardware as the most common source for computer system outages already in the last century [1], and modern control systems have complex functionality implemented with software. Software failures therefore present a major threat for ITER RH systems, which are safety-critical in the sense that a fault could damage research equipment or cause maintenance outages, potentially reducing experimental time.

Software requirement specifications have an important role in establishing safe and reliable RH operations, especially since the RH systems will be developed by several contractors. This is because: 1) requirements set targets that are used to verify software quality, including reliability; 2) quality attributes (i.e. non-functional requirements for "how well" the system should perform) drive significant architectural and design decisions [2] and 3) requirement specification is the largest source for defects in high-quality software [3]. In order to improve dependability in control systems, our research evaluates effective ways for RH system development teams to present related software requirements.

2. Comparison of RAMI process and dependability

Reliability is defined as the probability of failure-free operation for a specified period of time in specified environment [4]. It is also one of the key attributes in the RAMI process [5] used in the ITER project to manage risks in the facility development and design. In the RAMI process every system undergoes a risk-analysis to evaluate what can go wrong and to recommend spare components,

* Corresponding author.
E-mail address: pekka.alho@tut.fi (P. Alho).

0920-3796/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.fusengdes.2012.11.008

back-up systems, maintenance schedules, etc. to reduce the risk level of breakdown to minimum [6]. RAMI stands for:

- Reliability (continuity of correct service),
- Availability (readiness for correct service),
- Maintainability (ability to undergo modifications and repairs) and
- Inspectability (ability to undergo easy visits and controls) [6,7].

This is similar to the concept of dependability used in computing and communication systems. Dependability has same attributes, except instead of inspectability it has *integrity* (absence of improper system alterations) and *safety* (absence of catastrophic consequences on the user and environment) [7], being more relevant for RH software.

Specifications and standards usually implicitly or explicitly focus on hardware and are largely silent about software reliability and other quality attributes [8], and the RAMI process seems to be no exception. E.g. inspectability is essentially a requirement for mechanical systems. Dependability-related requirements for software-based systems need to take into account that failure mechanisms of software differ from mechanical systems. Hardware usually fails because of physical faults caused by wear and aging, whereas software failures are typically caused by human errors made in the development phase of the system and are deterministic in nature, making software faults harder to predict, locate and correct [4]. Because of these reasons, proving the reliability of software is not as straightforward as for mechanical subsystems and the related requirements for software need to reflect this.

3. System fault tolerance and dependability requirements

In this chapter we analyze the practices of developing dependability requirements and apply them to RH system software. RH systems typically need high reliability and have a combination of safety-critical and non-critical subsystems. Software systems are complex, which makes fault tolerant and dependable software costly: development often includes risk assessments, verification & validation procedures, and restrictions on design choices. However, use of a particular technique or techniques is not evidence of software quality, and even certified systems fail [9].

For high quality software–like control systems–the requirements specification is the most important source of delivered defects [3]. These defects can be due to errors, changes or omissions in requirements. Errors and changes can be usually discovered and managed with inspections (validation of requirements) and tools, but missing requirements can be considerably more difficult to detect. Possible sources for software requirements include system requirements specification (which includes system safety requirements), software hazard & risk analyses, hardware & environmental constraints and customer input [10]. To adequately define dependability and fault tolerance requirements for a system, several aspects of the software must be documented, including quality attributes, intended modes of operation, timing requirements, failure modes, and safety-related functionality which are briefly covered next.

3.1. Dependability objectives

The dependability objectives are documented in quality attributes (reliability, availability etc.). For control systems, important attributes include e.g. interoperability and evolvability (which has longer-term focus when compared to maintainability) because of the long expected lifetimes. Different subsystems may have different target levels of reliability.

Dependability objectives must be defined for a given environment, i.e. operation conditions. No system can be dependable under all conditions, so the claims must be made explicit [11]. These include not only environmental factors, but also expected interaction with external systems and humans.

3.2. Operation modes

Modes of operation are based on operational conditions or mission phase. By specifying operation modes we can limit the amount of functionality that has to be considered at a time.

Operation modes can also affect enabled commands or allowable limits for parameters, which has safety implications. Examples of operation modes important to dependability include automatic & manual, degraded operation and recovery modes.

3.3. Timing requirements

Timing requirements include communication deadlines, sampling rates, time to criticality etc. If the system has timing requirements that include hard deadlines, this has major impact for the system architecture design.

Safety and reliability can also be in odds–reliability can cause non-determinism for communications, as resent information could already be old. Especially in safety-critical systems it is often more important to keep sending up-to-date information.

3.4. Fault tolerance and responses to undesired events

Even though software developers work to create correct requirements and code, software will always have faults–and the number of delivered defects per function point goes up with software size and complexity [3]. Thus we also need to consider responses to undesired events, even if the software has low number of defects. This includes needs for fault tolerance (error detection, recovery, redundancy), specifying failure modes, i.e. how the system should fail, and what the system is not allowed to do in the case of failure.

Another factor that has to be considered in the case of errors is the tradeoff between robustness and correctness: robust software function tries to return some value (even if inaccurate) and correct software will return no results, which is usually better for safety-critical systems since faults will be easier to detect.

3.5. Safety-critical requirements

Reliability focuses in costs of failure and downtime, whereas safety focuses in dangerous failure modes. When a potentially unsafe command is detected, safety system inhibits the hazardous command and initiates transition to a known safe state. E.g. ITER will have a hardware-based plant interlock system which implements investment protection functions [12].

Safety-critical software covers software that has *impact on hazards* (cf. safety systems that are used for avoidance or control of hazards). Plant subsystems like RH may have complex safety-related functionality which must be implement with software. Examples of such functions include stability of machines, anti-collision systems and reduced speed & restricted space for robots [13]. Any such software feature identified as a potential hazard should be designated as safety-critical to ensure that future changes and verification processes can take them into consideration [10]. Candidates for safety-critical items list can be found e.g. with software FMEA.

4. Example: ITER RH software requirements & recommendations

4.1. Safety standards in ITER

Codes and standards applicable at ITER are French standards for a basic nuclear plant. Main standard is IEC61513 (Nuclear power plants–instrumentation and control systems important to safety–general requirements for systems) [14] which has been derived from the functional safety standard IEC61508.

Safety systems in ITER fall into three categories, which are nuclear safety, occupational safety and personal access safety. However, RH systems do not implement safety functions for these categories, even though they may have some safety-related functionality (i.e. economical hazards for research equipment and plant availability). It is therefore seen that RH systems are not "instrumentation and control systems important to safety" as intended in the standards, and therefore not in their applicable domain. The developers therefore have more flexibility to choose most efficient practices, but also need more expertise to do so without compromising dependability of the system.

4.2. Non-functional requirements

In this section we analyze some dependability-related requirements chosen from the system requirements document for remote handling control systems [15].

Reliability:

- "RH operations without causing significant damaging to ITER components shall have a target reliability of greater than 98% over a 120 day operational period."
- "The RH high-level control system shall have target reliability against significant failures of greater than 90% over a 120 day operational period."

Target level for RH operations without significant damaging equals safety integrity level (SIL) 1 in the IEC61508 functional safety standard (probability of failure per hour $\geq 10^{-6}$ to $<10^{-5}$). The high-level control system itself has lower reliability requirements so only safety-critical operations need to be considered for higher reliability level, instead of the whole control system. To be useful, the requirements specification should also define "significant damaging" and "significant failures" precisely.

For hardware components it is often possible to evaluate their reliability based on their historical data and/or subcomponent specification. However, for software components such data is not usually available. Approach used in the IEC61508 is based on giving recommendations on the use of specific techniques in the development based on the target SIL. Other sources recommend building a dependability case for the software, which should explain why the critical properties hold e.g. with (formal) requirements, testing results and verification [11].

Maintainability:

- "The RH high-level control system shall have a maintenance system that ensures that significant failures have an average recovery time of less than 1 day."

For software, maintainability means how easy it is to correct defects and make changes. To achieve high maintainability, software needs to be well documented and easy to understand. Especially the latter can be hard to achieve with a complex system like RH. Maintainability can also be greatly affected by choice of programming environment and language, as e.g. licensed programming languages are generally lacking in 3rd party tool support like version control.

Software updates and fixes may also increase failure rates as new faults can be injected to the system. This is critical for ITER because the RH systems need to be able to accommodate new and changing functional requirements over time, caused by evolving experimental changes.

4.3. Safety-critical subsystems

High-level control system for ITER RH [15] is used to demonstrate division to safety-critical & non-safety-critical subsystems in Fig. 1. Software subsystem is considered safety-critical if it controls hazardous or safety-critical hardware or software or provides information upon which a safety-related decision is made [10]. Systems that monitor safety-critical hardware or software as part of a hazard control are considered as safety-systems and are presumed to be implemented with hardware (interlocks).

Safety critical subsystems include:

- RH input device and computer assisted teleoperation (CAT) which are used for control in manual control.

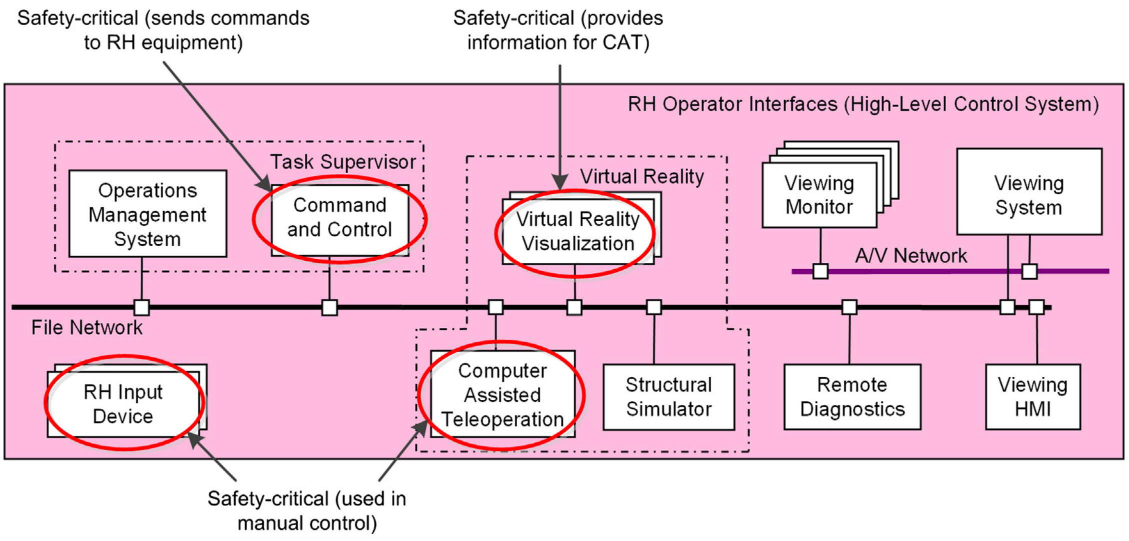

Safety-critical (sends commands to RH equipment)
Safety-critical (provides information for CAT)
RH Operator Interfaces (High-Level Control System)
Task Supervisor
Operations Management System
Command and Control
Virtual Reality
Virtual Reality Visualization
Viewing Monitor
Viewing System
A/V Network
File Network
RH Input Device
Computer Assisted Teleoperation
Structural Simulator
Remote Diagnostics
Viewing HMI
Safety-critical (used in manual control)

Fig. 1. High-level control system [15], showing safety-critical subsystems as identified in this paper.

- Virtual reality which provides information for CAT (dependencies are not shown in the figure).
- Command and control which is used for executing commands in automatic operations.

Structural simulator, remote diagnostics and viewing system are not considered safety-critical in this example because they do not directly impact hazardous operations, but are used as supporting information sources and for data analysis. However, now the system has a combination of safety-critical and non-safety-critical subsystems. Without decoupled system architecture and error detection, there is a danger of fault propagation. Therefore the system will need requirements and constraints like example 2 in the next section to reduce common-mode failures.

4.4. Example requirements

4.4.1. Example 1

Fast controllers (in a plant system) may run Linux as the operating system and have safety-critical functionality, but Linux is not validated for safety-critical use. However, extensively used software may reach reliability levels suitable for SIL1 or even SIL2 [16]. Recommendation is to restrict allowed versions for kernel and distribution to specific, well-tested versions, and gather reliability-related evidence for building a supporting dependability case. Constraint could be e.g.

[CO-1] Linux kernel version must be 2.6.30.8.

4.4.2. Example 2

To prevent fault propagation, add requirements to support modularity and decoupling between subsystems:

[MA-1] No direct inter-module references (function calls, class references etc.) to other software modules are allowed between subsystems.

4.4.3. Example 3

Specify dependability requirements explicitly for different properties. E.g. incorrect “stop” is significantly less dangerous than incorrect “go” [11].

[RE-1] No more than one out of 100 RH operations using collision detection and virtual force functionality for guiding telemanipulation operations shall be incorrectly stopped.

[RE-2] No more than one out of 10 000 RH operations using collision detection and virtual force functionality for guiding telemanipulation operations shall cause significant damage to ITER components.

5. Conclusions

Requirement specification presents a major source of defects for control systems, and thus has major impact on reliability. However, risk management processes and codes often neglect software reliability, even though reliability requirements need to be managed as a system measure that accounts for both hardware and software, and their different characteristics taken into account. Especially proving the reliability of software can be problematic.

Remote handling systems have safety-critical functionality like stability and anti-collision systems implemented with software, whereas actual safety functions are usually implemented with hardware. Software development is therefore guided by dependability requirements instead of strict safety standards, as risks are economical ones. Cost-efficient development needs to take into account that different subsystems therefore need different levels of reliability (e.g. diagnostics is not as critical as command & control). Systematically developed requirements can be used to form a dependability case for the system under development, where requirements, architectural solutions, verification etc. is provided to give sufficient confidence in the reliability of software.

Acknowledgments

This work was carried out under the EFDA Goal Oriented Training Programme (WP10-GOT-GOTRH) and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

References

[1] J. Gray, A census of Tandem system availability between 1985 and 1990, IEEE Transactions. on Reliability (1990) 409–418.
[2] K. Wiegers, Software Requirements, Microsoft, Redmond, 2003.
[3] C. Jones, Software quality in 2011: a survey of the state of the art, 31 August 2011 (accessed 14 August 2012), http://sqgne.org/presentations/2011-12/Jones-Sep-2011.pdf
[4] M. Lyu, Handbook of Software Reliability Engineering, IEEE Computer Society Press and McGraw-Hill, 1995.
[5] K. Chan, et al., Development of a RAMI program for LANSCE upgrade, Particle Accelerator Conference (1995) 822–824.
[6] I. Rona, RAMI, or thinking ahead, 27 Oct. 2008 (accessed 9 August 2012) http://www.iter.org/newsline/55/1193
[7] A. Avizienis, et al., Basic Concepts and Taxonomy of Dependable and Secure Computing, Transactions. on Dependable and Secure Computing 1 (1) (2004) 11–33.
[8] M. Hecht, K. Owens, J. Tagami, Reliability-Related Requirements in Software-Intensive Systems, Reliability and Maintainability Symposium (2007) 155–160.
[9] D. Jackson, A Direct Path to Dependable Software, Communications of the ACM 4 (52) (2009) 78–88.
[10] NASA Software Safety Guidebook, NASA, 2004.
[11] D. Jackson, M. Thomas, L. Millett, Software for Dependable Systems: Sufficient Evidence? National Academy Press, 2007.
[12] ITER, Plant control design handbook, 27LH2V, 2011.
[13] T. Malm, et al., Safety-critical software in machinery applications, VTT, 2011.
[14] L. Scibile, et al., The ITER safety control systems – status and plans Fusion Engineering and Design 85 (2010) 540–544.
[15] D. Hamilton, System Requirement Document SRD-23-07 Remote Handling Control System, ITER, 2008.
[16] P. Bishop, R. Bloomfield, P. Froome, Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications, CRR 336/2001, UK Health and Safety Executive Contract Research Report, 2001.

Publication 3

P. Alho and J. Mattila, Real-Time Service-Oriented Architectures: A Data-Centric Implementation for Distributed and Heterogeneous Robotic System, Proceedings of the 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 2013, IFIP Advances in Information and Communication Technology, vol. 403, pp. 262-271, 2013

DOI: 10.1007/978-3-642-38853-8_24

© Springer. Reprinted with permission.

Real-Time Service-Oriented Architectures: A Data-Centric Implementation for Distributed and Heterogeneous Robotic System

Pekka Alho and Jouni Mattila

Tampere University of Technology, Dept. of Intelligent Hydraulics and Automation, Finland
{pekka.alho,jouni.mattila}@tut.fi

Abstract. Cyber-physical systems like networked robots have benefited from improvements in hardware processing power, and can facilitate modern component and service-based architectures that promote software reuse and bring higher-level functionality, improved integration capabilities, scalability and ease of development to the devices. However, these systems also have very specific requirements such as reliability, safety, and strict timeliness requirements set by the physical world, that must be addressed in the architecture.

This paper proposes a real-time capable service-oriented architecture, based on data-centric middleware and an open real-time operating system. A prototype implementation for a robotic remote handling scenario is used to test the approach. The architecture is evaluated on the basis of how well it fulfils the expectations given for the service-orientation, including: reusability, evolvability, interoperability and real-time performance. In one sentence, the goal is to evaluate the benefits of a data-centric approach to service-orientation in a performance-critical and distributed system.

Keywords: real-time, distributed, SOA, data-centric, middleware, robotics.

1 Introduction

Developing software for cyber-physical embedded systems such as networked robots is a demanding task, due to complex functionality that has to be realised in a distributed and heterogeneous computing environment which typically has requirements for real-time performance and fault tolerance. Many of the challenges in these systems are related to interoperability and growing scale. Typically a distributed control system will consist of several subsystems running on different platforms that produce and consume increasing amounts of data.

On a higher abstraction level, business processes are also becoming strongly networked to improve efficiency by automatically transferring data, task requests etc. between systems. This means that robotic systems must be integrated to operations management systems, open for external connections, and able to connect and cooperate with other machines. These requirements and challenges are not unique to robotics

© IFIP International Federation for Information Processing 2013

– other domains like industrial automation, mobile machines and telecommunication have very similar issues.

Service-oriented software engineering has evolved from component frameworks and object orientation to meet the demands of more open and networked environments. It promotes reuse by decomposing business processes into reusable core services. The main benefits of service-oriented architecture (SOA) include high level of decoupling – provided by the service model – and interoperability which enables service providers and consumers to exist on different platforms. Two major downsides typically associated with SOA are complexity of developing such a system and increased overhead caused by communication mechanisms [6]. The latter is also related to the lack of performance guarantees, and presents a major challenge especially for embedded systems. An SOA implementation for robotic system therefore needs to place heavy emphasis on solving this problem, which is one of the key design goals for the architecture presented in this paper.

Application of SOA design principles to real-time systems (RTSOA) is a research topic that has come up in the last decade, with research including experimental implementations [3], [9], [10] and related key features like service composition [2], [4]. However, most of the current RTSOA approaches are based on the existing message-based Web Service standards. Web services face challenges when used in embedded systems, as messages need to be serialized in real-time [2], and quality of service (QoS) must be managed at the transport layer. Other challenges include complexity of networking with HTTP, XML, and SOAP; constraints imposed by embedded system architecture; and verbosity of HTTP and XML.

We believe that the service-oriented approach may be beneficial for the development of cyber-physical systems, but there is a need to test out different implementation solutions that fulfil the specific limitations and requirements of the target domain, including reliable communications, limited resources, and deterministic behaviour. In this paper we present a data-centric approach to RTSOA and evaluate it by implementing the proposed reference architecture for a robotic remote handling scenario. Remote handling involves human operators remotely controlling robots that perform tasks like maintenance or construction in dangerous environments, so reliability and performance of the system are vital for successful task completion.

2 Real-Time Service Orientation for Robotic Systems

2.1 Design Goals

We see the following as the main design goals for the real-time service-oriented architecture:

- Promote software **reuse** by producing reusable and decoupled software modules.
- Enable **composing** a working system out of reusable and existing services.
- Improve **interoperability** of heterogeneous systems (platform & programming language independence).

- Ensure **evolvability** [8] in the future; the architecture should support changing requirements and operating environments during the system lifecycle.
- Deterministic **real-time performance,** despite dynamically changing environment.
- **Dependable and fault tolerant** operation.
- Improve **cost-efficiency & ease of development;** the implementation should be able to use off-the-shelf solutions for tools, software and hardware, instead of purpose-built applications and devices.

2.2 Reference Architecture

The reference architecture, introduced in [5], is a general proof-of-concept control system platform for machine automation as an alternative to proprietary and specialized solutions. The platform is based on the ideas of real-time service orientation, introduced previously in this section, and emphasizes integrability, interoperability, maintainability and heterogeneity. Service orientation allows software components to be published and located locally or over a network.

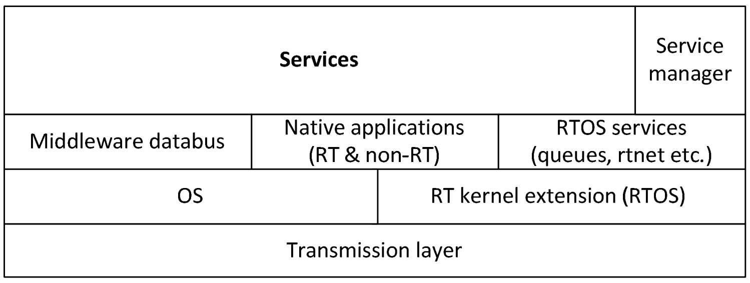

Services
Service manager
Middleware databus
Native applications (RT & non-RT)
RTOS services (queues, rtnet etc.)
OS
RT kernel extension (RTOS)
Transmission layer

Fig. 1. Layered architectural view of the reference architecture

In section 3 we will describe the actual implementation of the reference architecture for a remote handling scenario. A high-level layered view of the reference architecture is shown in Fig. 1. Key concepts of the architecture are services, communication & information sharing mechanisms, composition, and fault-tolerance. These are described next.

2.3 Concurrency Model and Real-Time Performance

The choice of a concurrency model for the architecture directly affects decoupling of the modules and management of real-time constraints. Options for the concurrency model include processes, threads or call-back functions [1]. Each solution has its own pros and cons for ease of development and inter-process communication. For a service-based architecture, the process-based model (services as processes) makes most sense, as it is the most decoupled alternative. This decoupling provided by processes has benefits, including the possibility to more easily manage services at runtime and improved robustness.

A service can be defined as an independently developed, deployed, managed, and maintained software implementation that directly represents business tasks or devices.

A service can be defined by a verb which describes the function it implements, e.g. “generate a trajectory”. Our implementation of services uses object orientation: service is an interface (virtual class) that has methods for starting, stopping, restarting etc. the service, which the service developer must implement. Services can use native applications and services provided by the operating system (e.g. APIs for communication).

2.4 Communication and Information Sharing

In order to communicate, components need some form of visibility or references between the communicating parties. However, this can lead to a tightly coupled system design that scales poorly. Examples of communication methods that impose coupling include sockets, remote method invocation and client-server model in general. A more decoupled solution is to use middleware based on the asynchronous publish/subscribe communication paradigm, which can be implemented as message-based like Java Message Service (JMS), or data-centric like Data Distribution Service (DDS)[1].

Another communication problem in distributed real-time systems is that networking can add unpredictable delays and unreliability to connections. Therefore we need to be able to set and monitor quality of service (QoS) parameters like reliability and how long the data is valid for each topic, so that the system can react appropriately if the QoS is compromised. QoS can be used to define if we want reliable sending (e.g. for commands) or just the most recent value as fast as possible (e.g. sensor measurements).

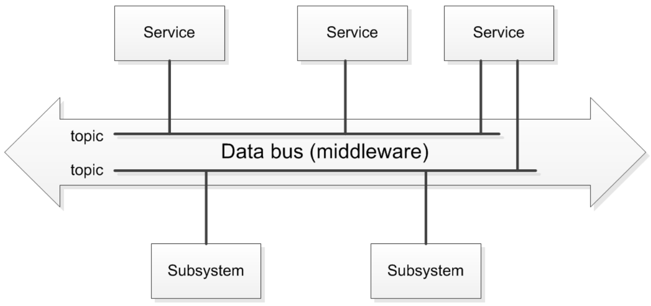


Fig. 2. Bus-based communication in SOA

The data-centric middleware can be used as a data bus between the services, as shown in Fig. 2: this is similar to the use of enterprise service bus (ESB) in enterprise SOAs. Another benefit of using a distributed middleware is a global data space where all data can be accessed; there is no central broker/repository that could act as a bottleneck or a single point of failure.

In an ideal situation we would have total location transparency for the services (no difference between accessing local and distributed services), but in order to achieve optimal real-time performance, the architecture uses separate communication methods

[1] A standard maintained by Object Management Group, `http://portals.omg.org/dds/`

for local and networked communications, termed local service bus and global service bus. The reference architecture itself is not committed to any specific communication standard, but the implementation uses DDS middleware and the communication mechanisms provided by the real-time operating system (RTOS) Xenomai[2].

Local connection of services as components and the use of DDS as a data-bus for distributed communications combine the strengths of component and service approaches, and provides optimal real-time performance in both cases. DDS can be used on low-end embedded systems to read and send sensor information, whereas XML-based solutions would be too heavy, and would necessitate a separate solution.

- Global service bus: DDS was chosen since it implements asynchronous data-centric publish/subscribe model and provides QoS management, making it suitable for cyber-physical systems, which place a heavy emphasis on sending and receiving data.
- Local service bus: services can use RTOS message queues (an asynchronous "mailbox") or shared memory for local real-time communication between two services. The queue-based local communication is similar to the component wiring approach used in component-based software engineering.

2.5 Composition

In complex systems, the number of internal components can easily grow to the range of hundreds or even thousands. Management of this many components or services can be complex and laborious if the framework-implementation of the architecture does not provide tools for this. Engineering of new applications from reusable components is supported by a repository of available components, configuration services to select and combine components, and run-time mechanisms that allow components to be dynamically changed.

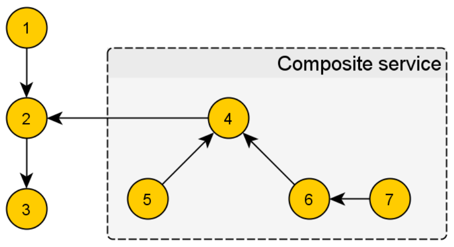


Fig. 3. Composite service (Key: circle denotes a service, arrow shows direction of data flow)

In the service-oriented architecture, higher level functionality can be implemented by creating composite services of the existing services, as shown in Fig. 3. Different means of implementing composition include programmatic, publish/subscribe, events

[2] Real time Linux kernel extension and development framework, `http://www.xenomai.org/`

and orchestration engine. Since our reference architecture is based on the publish/subscribe model, this is a natural match for the composition mechanism, and enables flexible implementation of composite services. Services can be chained locally and globally to form new composite services. A single service can be part of multiple composite services and used by multiple other services, which can reduce the level of unnecessary redundancy in the system.

A repository provides a way to document and list available services or components. For SOAs this can be done by writing an interface description and saving it in the repository. Service registries, on the other hand, provide runtime information for finding and binding services. In our proposed data-centric approach, based on the use of a data bus, the middleware can handle registration of new publishers, and match subscribers to the provided data topics.

Service composition and management at runtime is handled dynamically through a local service manager, which controls spawning of new services. This makes it possible to modify a service and restart it on-the-fly, enabling faster deployment process by updating only related services, instead of having to recompile the whole system after every reconfiguration or update.

2.6 Fault Tolerance

Fault tolerance is a key requirement for the architecture, as many cyber-physical systems perform safety-critical tasks. A fault in the control system may endanger human lives (either directly or indirectly), cause operational downtime or damage the environment or equipment. Service-orientation can support error confinement with the modular architecture, based on the decoupling provided by the service model, although the system still needs to implement error detection and recovery.

Because of the decoupled design, developers cannot make the presumption that other services are always available, and must take the situation into account in their application code so that the service will react if a dependency goes down, e.g. because of failure or manual shutdown. The error handling approach based on decoupling is similar to the one used in the Erlang programming language, which can be summarized as “let it crash” [7]. In the event of an error, the process is terminated, presuming it is not an exception that can be handled. This forces other services to react and do error recovery, including entering their safe state. The architecture can still be prone to error propagation, so the services should be made fail-silent if possible, making it easier to detect faults.

In order to implement error detection, the system can use a service manager to detect crashed services based on heartbeat signals or monitoring the use of resources like CPU and memory. Unresponsive behaviour or unexpected increase in CPU usage for a service can indicate a fault in the service, and may endanger real-time performance of other services and cause unexpected and potentially dangerous behaviour. The service manager restarts the unresponsive service, which will put the system temporarily into a safe state by forcing other services to do error handling, according to the “let it fail” approach. Key principle is writing loosely coupled services, by forcing the developer to consider situations where the dependency services are not available or timing constraints are violated.

3 Implementation for a Remote Handling System

In order to test the proposed data-centric real-time approach to service-orientation, we implemented a remote handling control system (RHCS) for automated teleoperation of an industrial robot Comau SMART NM45-2.0, based on the reference architecture described in the previous section. A basic remote handling scenario consists of an operator using the web-server based Operation Management System (OMS) to send movement commands to the equipment controller. Virtual reality software (IHA3D) is used to visualize the position and movements of the robot.

Services deployed on the equipment controller for the remote handling system implementation are shown in Fig. 4. Service descriptions, real-time task priorities and execution periods are listed in Table 1.

Table 1. List of services used in the remote handling control system

Service name	Service description	Priority [0 .. 99]	Period [ms]
`Trajectory-Generator`	Generate a trajectory profile that the manipulator can follow from one point to another.	50	2
`C4G`	Interact with the low level control system of the manipulator.	91	2
`C4GJoint-DataPub`	Publish manipulator joint position data.	45	10
`OmsCom`	Read OMS commands and manipulator joint data; send commands to the trajectory generator to create new trajectories.	40	50
`Measuring`	Measure task execution time and jitter.	20	0.1

4 Evaluation of the Experiment

This section presents an evaluation of the problems and benefits of the proposed approach that could be observed with the implemented experimental system. The system is evaluated with the following criteria: reusability, interoperability, evolvability, real-time performance, fault tolerance, and ease of development. Dynamic composition performance depends greatly on the algorithm design [2] and it is not evaluated in this paper. Instead, a static composition is used.

Reused software includes `TrajectoryGenerator` service, `C4G` service and two subsystems (OMS and virtual reality). A service-based implementation avoids stovepipe system antipattern[3] as services are loosely coupled (no direct references to other services) and do not interfere with each other's namespaces etc., simplifying future reuse of services.

[3] `http://sourcemaking.com/antipatterns/stovepipe-system`

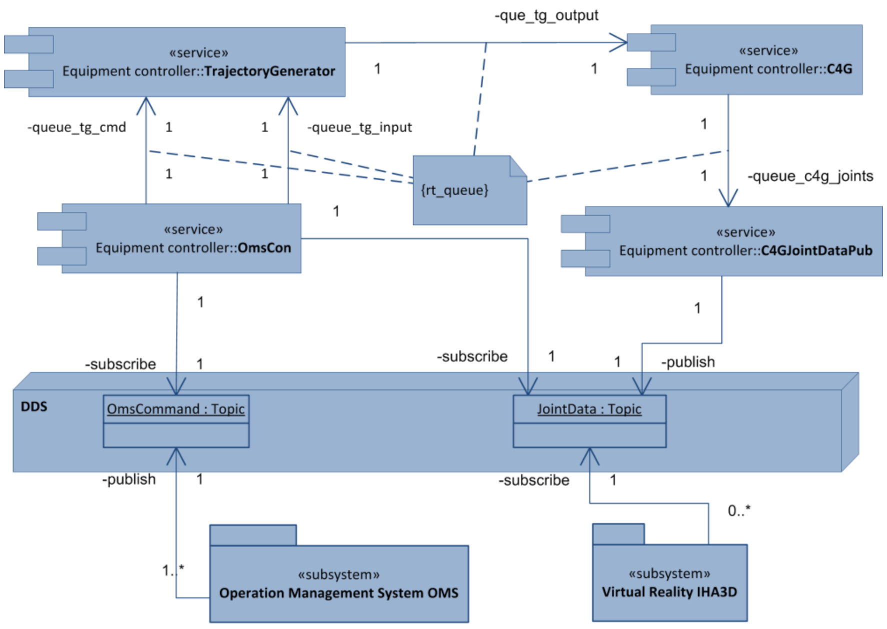


Fig. 4. Service deployment view for the system (Key: UML)

Interoperability of heterogeneous systems (machines and higher-level enterprise systems) is supported on any platform that has a compatible DDS implementation. DDS is available on several programming languages, therefore good programming language independence is provided. Interfaces to Web services, REST-based services and other communication platforms can be implemented with adapters.

Evolvability – the software must be able to accommodate new and changing requirements, including connections to unforeseen external sources. Ability to do this in the long term is especially important for industrial automation systems, because they have long expected lifetimes. This can be measured with evolvability, which describes the ability of software to accommodate future changes [8]. Performing a complete evolvability analysis is not reasonable in this context, so we focus on the changeability, extensibility and portability sub-characteristics:

- Changeability: Data typically has better consistency in the long run when compared to interfaces. However, if the data topics or queue configurations are changed or added, corresponding modifications must be implemented to both publishers and subscribers, but it is possible to provide extensions topics that provide the new or changed data, thus retaining compatibility with old implementations.
- Extensibility: New topics or functionality in the form of services can be added on-the-fly, without shutting down and recompiling the whole system. The run-time composition can be managed with the service manager, which can also be used to lazily launch necessary services (service chains).
- Portability is limited if RTOS-specific features like real-time queues are used.

Real-time performance – we analysed system performance by measuring cycle durations for a real-time task first unloaded and then running a full remote handling system with a script generating artificial CPU, network & disk loads. The real-time measuring task was executed 10000 times with 100 µs period on a 3.4 GHz Pentium 4 CPU. The measured latencies are shown in Fig. 5. Although standard deviation of cycle duration has increased from 172 ns to 410 ns in the heavily loaded system, graphs show highly deterministic behaviour in both cases. Performance of the DDS middleware in embedded real-time systems has been evaluated e.g. by Xiong et al. in [11].

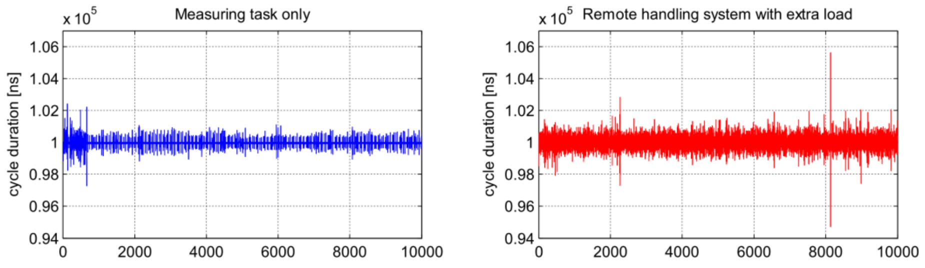


Fig. 5. Cycle durations for single task vs. remote handling system with extra load

Fault tolerance – the service manager can detect if services use more system resources than reserved at start-up, and force a restart. Other services need to react according to the “let it crash” error handling approach. After the services have been restarted, normal operation can be resumed if the fault was transient. A leaky bucket counter or an escalating retry timer can be used to distinguish transient faults from permanent ones.

An example case of error handling: the `TrajectoryGenerator` service is killed in the middle of running a trajectory to the `C4G` service, which controls the robot. `C4G` service detects that there is no new data available, and stops the movement of the robot, by ramping down the power in a controlled and safe fashion. Normal operation can be resumed when the `TrajectoryGenerator` is restarted.

Cost-efficiency & ease of development: the service-model is an intuitive approach for developers, as services can be interfaces to devices or related to tasks that must be accomplished. Linux-based development offers a variety of tools & drivers, reducing need for self-developed or proprietary choices. Communication configurations (for local queues) are currently hardcoded, so managing a large number of local communications becomes cumbersome, although the service manager can be used to start services. The local service communications should be standardized and details moved to external configuration files that could also be managed with tools to simplify management and reduce local coupling between services.

5 Conclusions

A dynamic module system based on services or components is necessary to manage complexity of embedded and distributed control systems. The module system should abstract the communications between modules, and provide tools for managing and deploying the configurations in order to improve software reusability and simplify development process, maintenance, and integration of new devices to the system.

In this paper we have presented our design concept for a service-based software architecture. Our proposed approach adapts the SOA paradigm with data-centric design, based on topic-based publish/subscribe middleware and RTOS. The experimental implementation of the architecture demonstrates integration of heterogeneous subsystems with the service-based control system through a scalable middleware-based data bus. The control system is based on an open RTOS and has deterministic real-time capabilities. Although all composition features in the prototype are not fully implemented, it provides contribution by testing the data-centric approach to implementing RTSOA.

References

1. Calisi, D., Censi, A., Iocchi, L., Nardi, D.: Design choices for modular and flexible robotic software development: the OpenRDK viewpoint. Journal of Software Engineering for Robotics 3(1), 13–27 (2012)
2. Tsai, W., Lee, Y.-H., Cao, Z., Chen, Y., Xiao, B.: RTSOA: Real-Time Service-Oriented Architecture. In: Proceedings of the 2nd IEEE International Symposium on Service-Oriented System Engineering (SOSE 2006), pp. 49–56. IEEE (2006)
3. Cucinotta, T., Mancina, A., Anastasi, G., Lipari, G., Mangeruca, L., Checcozzo, R., et al.: A Real-Time Service-Oriented Architecture for Industrial Automation. IEEE Transactions on Industrial Informatics 5(3), 267–277 (2009)
4. Moussa, H., Gao, T., Yen, I.-L., Bastani, F., Jeng, J.-J.: Toward effective service composition for real-time SOA-based systems. Service Oriented Computing and Applications 4(Special Issue: RTSOAA), 17–31 (2010)
5. Hahto, A., Rasi, T., Mattila, J., Koskimies, K.: Service-oriented architecture for embedded machine control. In: International Conference on Service-Oriented Computing and Applications. IEEE (2011)
6. Machado, A., Ferraz, C.: Guidelines for performance evaluation of web services. In: Proc. of the 11th Brazilian Symp. on Multimedia and the Web, WebMedia 2005, pp. 1–10. ACM (2005)
7. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Dissertation. Royal Institute of Technology, Stockholm (2003)
8. Pei-Breivold, H., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. Inf. and Software Technol. 54(1), 16–40 (2012)
9. Panahi, M., Nie, W., Lin, K.-J.: A Framework for real-time service-oriented architecture. In: 2009 IEEE Conf. on Commerce and Enterp. Comput., pp. 460–467. IEEE (2009)
10. Garces-Erice, L.: Building an Enterprise Service Bus for Real-Time SOA: A Messaging Middleware Stack. In: 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, pp. 79–84. IEEE (2009)
11. Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., Schmidt, D.C.: Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems (2011)

Publication 4

P. Alho and J. Mattila, Software fault detection and recovery in critical real-time systems: an approach based on loose coupling, Fusion Engineering and Design, vol. 89, no. 9-10, pp. 2272-2277, 2014

DOI: 10.1016/j.fusengdes.2014.04.050

© Elsevier. Reprinted with permission.

Contents lists available at ScienceDirect

Fusion Engineering and Design

journal homepage: www.elsevier.com/locate/fusengdes

Software fault detection and recovery in critical real-time systems: An approach based on loose coupling

Pekka Alho*, Jouni Mattila

Department of Intelligent Hydraulics and Automation, Tampere University of Technology, Finland

HIGHLIGHTS

- We analyze fault tolerance in mission-critical real-time systems.
- Decoupled architectural model can be used to implement fault tolerance.
- Prototype implementation for remote handling control system and service manager.
- Recovery from transient faults by restarting services.

ARTICLE INFO

Article history:
Received 30 August 2013
Received in revised form 7 April 2014
Accepted 22 April 2014
Available online 14 May 2014

Keywords:
ITER
Remote handling
Software
Fault tolerance
Dependability
Real-time

ABSTRACT

Remote handling (RH) systems are used to inspect, make changes to, and maintain components in the ITER machine and as such are an example of mission-critical system. Failure in a critical system may cause damage, significant financial losses and loss of experiment runtime, making dependability one of their most important properties. However, even if the software for RH control systems has been developed using best practices, the system might still fail due to undetected faults (bugs), hardware failures, etc. Critical systems therefore need capability to tolerate faults and resume operation after their occurrence. However, design of effective fault detection and recovery mechanisms poses a challenge due to timeliness requirements, growth in scale, and complex interactions. In this paper we evaluate effectiveness of service-oriented architectural approach to fault tolerance in mission-critical real-time systems. We use a prototype implementation for service management with an experimental RH control system and industrial manipulator. The fault tolerance is based on using the high level of decoupling between services to recover from transient faults by service restarts. In case the recovery process is not successful, the system can still be used if the fault was not in a critical software module.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Remote handling (RH) systems are used to inspect, make changes to, and maintain components in the ITER machine. Failure in a mission-critical system like RH may cause damage and, perhaps even more significantly, loss of experiment runtime, therefore making dependability one of its most important properties. However, even if the software for the RH system has been developed using valid development processes, the system might still fail due to undetected faults, hardware failures, etc. Critical systems therefore need to be able to resume operation after faults have occurred, but design of effective fault detection and recovery mechanisms poses a challenge. This is due to timeliness requirements combined with growth in scale and complex dynamic interactions in RH systems and embedded systems in general.

Several programming languages and frameworks, e.g. Erlang or OSGi for Java, support use of decoupled architectural models that can be used to implement fault tolerance solutions and dynamic loading of software modules, but these approaches are typically used in non-critical applications that do not have requirements for deterministic response times. In this paper we evaluate effectiveness of the decoupled architectural approach in mission-critical real-time systems using an experimental RH control system for an industrial manipulator. The control system is based on a real-time service oriented architecture (RTSOA) that we have introduced and evaluated in [1]. Services (i.e. the applications that participate in the control of the manipulator) are managed by a prototype

* Corresponding author. Tel.: +358 505375726.
E-mail address: pekka.alho@tut.fi (P. Alho).

http://dx.doi.org/10.1016/j.fusengdes.2014.04.050
0920-3796/© 2014 Elsevier B.V. All rights reserved.

service manager that is used to detect faults and initiate recovery processes.

The RH control system consists of several heterogeneous subsystems, including equipment controller (EC), virtual reality (VR) and operations management system (OMS), specified in the ITER RH control system handbook [2]. This kind of cooperation of several networked computational units is typical for the field of cyber-physical systems (CPS), featuring a tight coordination between computational and physical elements of the system. CPS research aims to improve interoperability and openness between networked controllers to produce more intelligent applications.

2. Background

Fault tolerance means avoiding service failures in the presence of faults, and consists of error detection and recovery [3]. However, recovery can introduce unpredictable delays that might be in conflict with the predictability requirements of real-time systems. A typical approach for real-time fault tolerance is to use two or more diverse versions of software. Such an approach is suitable, e.g. in aviation, where the scope of critical systems is limited, and the cost of creating multi-version software is distributed over a large number of aircraft [4]. However, for large and complex one-off software systems, such as RH, use of multi-version techniques is difficult to justify.

Key challenges for fault tolerance in RH systems include recovery of state data, reliable detection of faults, fault recovery that supports real-time requirements, and ensuring reliability of end-to-end service chains. Safety of the system relies heavily on reasoning about consequences of faults, which is an important open research area due to the complex and stochastic nature characteristic for CPS.

Previous research on real-time fault tolerance has focused largely on redundancy-based solutions and reconfiguration. Gonzales et al. use adaptive management of redundancy to assure reliability of critical modules by allocating as much redundancy to less critical modules as could be afforded, thus gracefully reducing their resource requirements [5]. Assured reconfiguration in case of failures is used in [6]. This allows the primary function to fail and then reconfigure to some simpler function – reconfiguration of the system is a critical part, and it is formally verified. Simplex architecture by Sha et al. uses high assurance and high performance control subsystems [7]. The high assurance subsystem is used to keep the system within the safety envelope. ORTEGA architecture improves the Simplex architecture by adding on-demand detection and recovery of faulty tasks [8]. An anomaly based approach for detecting and identifying software and hardware faults in pervasive computing systems is proposed in [9]. The methodology uses an array of features to capture spatial and temporal variability to be used by an anomaly analysis engine.

Our work differs from these approaches by focusing on transient faults in a real-time system by using highly modular approach instead of redundancy. Similar solution based on modularity and fault isolation has been successfully used, e.g. in the non-real-time MINIX operating system (OS) for driver management [10].

3. Fault detection and recovery in real-time systems

3.1. System definition

Our hypothesis is that mission critical real-time systems can use service management to recover from transient faults (discussed in Section 5) in a loosely coupled software architecture. In this context, we define a loosely coupled real-time system as follows:

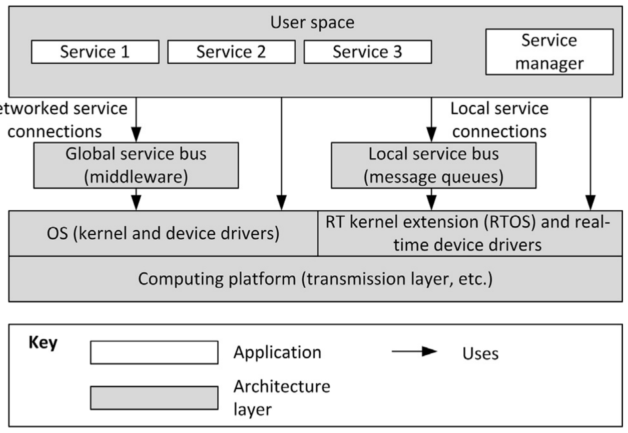

User space
Service 1
Service 2
Service 3
Service manager
Networked service connections
Local service connections
Global service bus (middleware)
Local service bus (message queues)
OS (kernel and device drivers)
RT kernel extension (RTOS) and real-time device drivers
Computing platform (transmission layer, etc.)
Key
Application
Architecture layer
Uses

Fig. 1. Logical system architecture.

- The system is made up of a set of periodic processes, i.e. services.
- Services are loosely coupled, having no direct interdependencies or references to each other.
- Services can be distributed over network or located on a single computer.
- Services communicate with a communication buses that facilitate monitoring of communication deadlines.

Advent of modern, powerful processors to RH systems provides a chance to mitigate the delays caused by the recovery process if the fault is detected before deadlines. In an optimal case, a fault can be detected and recovered before it causes service failures. If fault recovery causes exceeding of a deadline, other services can detect this and react accordingly by moving the system to a safe state while simultaneously isolating the fault. Since this recovery strategy does not rely on redundant versions, there is no need to maintain consistency between replicas, which is a major challenge for redundant systems [11]. We also leave formal methods out of the scope of our solution because of architectural limitations and costs associated with these methods are likely to be prohibitive for ITER.

3.2. Architecture

The system architecture in our implementation is based on RTSOA using data-centric middleware and an open source real-time operating system (RTOS). It provides decoupled connections for the services via local and global service buses (LSB and GSB) and includes a service manager to monitor and manage services (Fig. 1) [1].

The LSB is based on real-time queues, a communication mode provided by the RTOS. A message queue can be created by one service and used by multiple services that send and/or receive messages to the queue. GSB is a wrapper that uses Data Distribution Service for Real-Time Systems (DDS) middleware for networked connections; DDS is a standard for decentralized and data-centric middleware based on the publish/subscribe model and aimed at mission-critical and embedded systems. The standard is maintained by the Object Management Group (http://portals.omg.org/dds/).

3.3. Service manager and service configuration

Service manager is a local component used to start services and detect faults. Service manager spawns the services as child processes, according to a configuration file – more advanced configuration methods could be supported, e.g. GUIs, web interface or GSB-based, but have not been included in the prototype implementation at this phase. Services update their status to the service manager through either LSB or GSB. Possible states are *running, stopped, (re)starting*, and *error*. The local communication bus is also used by the service manager to command services to switch states (*start*, *stop* and *restart*). More extensive set of states and commands would give a more fine-grained control over services, but also add additional complexity to service implementation and service management logic, potentially introducing new faults.

Fault detection utilizes several methods: service status updates, monitoring user-set resource limits for CPU and memory usage, heartbeat monitoring, and OS features (parent process can check the state of its child processes). Service manager can be used to dynamically update services by terminating them and replacing with new version, enabled by the decoupled bus-based architecture.

Unresponsive behavior or unexpected increase in resource usage for a service can indicate a fault in the service, potentially endangering real-time performance of other services or causing dangerous movements – this is a typical challenge for design of CPS, needing interdisciplinary co-operation of designers and researches from industry and academia. After a fault has been detected, the service manager either terminates or restarts the faulty service, according to the configuration. These actions will either recover the fault without cascading service failures or put the system to a safe state, according to “let it fail” approach. This method is mainly suited to handling of transient faults. If the fault recovery fails, system simply stays in the safe state.

The fault recovery strategy enables operator to continue operation or recover equipment by avoiding the conditions that trigger the fault, postponing software update to later time and thus retaining higher system availability. Alternatively, operations can be suspended until the fault has been removed.

Effectively the described method of fault tolerance is a form of fault masking. The system therefore needs an error manager component to track number of errors so that they do not go unnoticed. This functionality is logical to implement in the service manager. Error data provided by the service manager can be used to give, e.g. graphical warnings to the user about encountered faults or send notifications via email or another message channel. Finally, if restarting a service does not solve the fault, service manager can utilize escalation and move the system to limp-home or recovery mode.

Configuration of services to be started and managed by the service manager may include the following parameters – new parameters can be easily added on per-need basis (parameters in *italics* are not implemented in the prototype):

- Start command, including necessary command line parameters.
- CPU and memory usage limits in percentage (*or bytes*).
- Heartbeat timeout.
- Actions to be taken on failure: restart, terminate, *execute* {*program*}, *alert* {*email address*}.
- Limit for the number of restarts.
- List of dependencies if they must be also restarted.

3.4. Service design

In the case of a failure, service stops execution or is terminated by the service manager and its state data will be lost. State data can be divided to temporary, static and dynamic: temporary data is related to current computations and is not relevant after a failure, static data is typically configuration data that can be reread, and dynamic data contains results of calculations, user input, etc. Some of the dynamic state data can also be recalculated or reread (e.g. sensor measurements), but commonly it needs to be protected. This means that any state data that needs to be recovered after restart must be stored in a stable storage. However, since it is possible that the saved state data has been corrupted, sanity-checking (typically checksums or valid range for data) is needed. Another issue regards how much of the state should be recovered. For example, an interrupted trajectory of a remotely operated manipulator should not be continued because of potential risks. The software may also hit the exactly same bug again if the full state from the time of failure is recovered.

Another major consideration for service design is that services should be fail-stop or fail-silent so that a failure in one application does not cause unwanted behavior in others. Compared to heartbeat timeouts and increased resource use, detection of erroneous outputs is more difficult to implement and has to be done using more “traditional” methods including contract programming, asserts, exceptions, etc., according to the needs of the specific application. If the service detects an internal error (e.g. an exception), and is still in otherwise sane state, it can report the error to the service manager by publishing status change to *error*. In our implementation, services publish the status in the heartbeat signal periodically sent to the service manager.

Service dependencies may be unavailable at times due to faults or even normal use scenarios. Service developer therefore needs to implement monitoring for communication deadlines and decide how to recover services from faults. Recovery should be taken into account already in the specification [12]. This is a direct consequence of the dynamic nature of decoupled architectures and forces the developer to take into account situations where the dependencies of a service are offline or unavailable. Although the dynamic service connections necessitate some extra code, such as reinitialization of communications after faults, the final result is more resilient to error situations. RH specific failure mode analysis is covered in Section 5.

Another benefit of loose coupling between services is capability to have multiple copies running on more than one computer at once, providing redundant processing or hot backup capabilities. This can provide cost-efficient redundancy against computer hardware faults, especially for critical services. Services can also be hot swapped to enable better maintainability and non-stop reliability.

4. Remote handling control system

We have implemented an experimental RH control system to evaluate the proposed approach to fault detection and recovery. The RH control system operates an industrial manipulator (Fig. 2) manufactured by Comau in ITER relevant RH task scenarios. The

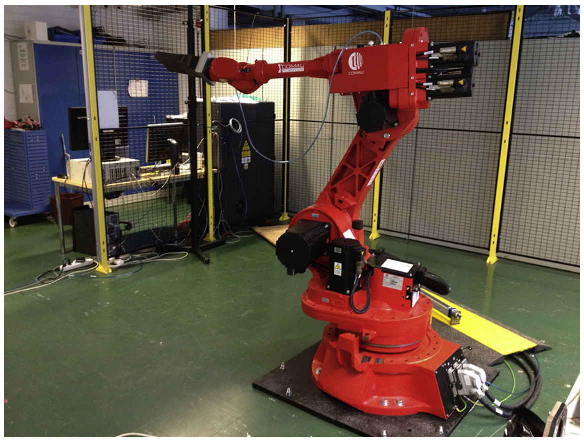


Fig. 2. Industrial manipulator used in experiments, equipped with a pneumatic gripper tool.

Table 1
RTSOA services on EC (service names in `Courier`).

Service	Service task description	Period T_i [ms]
`TrajectoryGenerator (TG)`	Generate a trajectory profile that the manipulator can follow from one point to another.	2
`C4G`	Send position and velocity reference to the manipulator control system at 500 Hz.	2
`C4GJointDataPub`	Publish manipulator joint position data to GSB.	10
`OmsCom`	Read OMS commands from GSB and pass them to the trajectory generator.	50

system architectural model from Section 3.2 is applied to implement safe management of services in the RH control system to detect failures based on observing abnormal behavior. The RH control system is a non-trivial application that is used to test the fault detection and recovery in critical real-time systems. Communication frequency between master and slave controllers in similar teleoperation applications is typically 500–1000 Hz. The prototype uses 500 Hz communication loop between EC and Comau low-level servo controller over real-time Ethernet. Missing a communication deadline causes the low-level controller to engage an emergency stop and drop the communication link, necessitating a system restart.

Test setup includes EC, VR and OMS subsystems that are specified in the ITER RH control system architecture [2], in addition to the Comau's own low-level controller. VR is used to visualize robot position, although the operator also has direct visual contact with the manipulator in the test setup.

EC is a real-time system for operating RH equipment, i.e. the manipulator in this case. Our EC implementation is running real-time Linux with the RTSOA services and service manager. Services and loop timings are listed in Table 1.

VR capabilities are provided by IHA3D, Windows-based software developed at the Department of Intelligent Hydraulics and Automation (IHA) [13]. IHA3D provides a simulated virtual environment for the operator and can be used for virtual force generation in bilateral teleoperation.

OMS is a task planner subsystem used to support operation by planning, helping and instructing execution of RH procedures [13]. Procedures are complete sequences of manual actions and movements required to perform maintenance or testing operations. We used a web-server based OMS implementation which provides a browser-based GUI for the operator. Operator can use the OMS to send movement commands to the manipulator.

5. Fault recovery for RH system

5.1. Fault types

One way to categorize faults is permanent and transient faults [3]. Transient faults include aging-related faults (e.g. memory leaks), interaction faults, race conditions, attempts to exploit security vulnerabilities, resource leaks, bit flips, temporary device failures, etc. [3,10]. Transient faults can be difficult to find with testing because their activation may depend on complex timing, state and runtime environment conditions. This means that even critical systems developed with best practices can encounter them, making error confinement and recovery capabilities important. These faults may be temporarily solved by rejuvenation [14], i.e. shutting down the software item and restarting it. Although the root cause may not necessarily be removed by creating a fresh item, system would typically be usable after restart. Transient faults are detected by service manager with resource limits, service status updates, heartbeat timeouts and monitoring of child process status.

Table 2
RH control system service criticality & recovery. Stop = "stop manipulator movement", restart = "restart service".

Service name	Critical	Restore state	Failure action
`TG`	Y	N	Stop, restart.
`C4G`	Y	Y	Terminate TG, restart.
`C4GJointDataPub`	Y	Y	Stop, restart.
`OmsCom`	N	Y	Restart

Our implementation uses real-time message queues to send heartbeat signals from services to the service manager. Heartbeat signal is combined with the status update for the service. This functionality can be extended to the GSB, providing remote nodes awareness of service health. Resource limit monitoring is currently based on the proc file system, a feature in UNIX-like systems providing a method to access process data through a file-like structure. Alternatively system calls, such as getrusage, could also be utilized.

Permanent faults, caused, e.g. by faults in algorithms or control flow, persist after service restart. Determining if the fault is permanent or transient is based on error counting implemented by the service manager, i.e. limiting the number of restarts per service to prevent infinite restart loops that would otherwise be caused by former.

Severe permanent faults are typically detected in testing for commonly used features. If new permanent faults are encountered during operations, operator can avoid triggering conditions (if known) and recover RH equipment using reduced or alternative functionality for maintenance. For example, remote handling systems typically offer alternative control modes in the form of controllers and input modes (e.g. manual vs. OMS). If a permanent fault prevents use of OMS, operator can recover the equipment using manual mode. Loosely coupled service-based design supports this kind of robustness, as faults are isolated to services (e.g. `OmsCom`) and system can use alternative service configuration. Basically, a service failure is not necessarily a system failure.

5.2. Recovery from service failures

Next we describe services failures and recovery in the prototype RH system (see Table 2). Fault detection and recovery are implemented as previously described. Tests with the prototype system have shown that the service manager implementation is capable of detecting crashed services with heartbeat timeouts and process status. Crashed service is restarted or terminated according to the used configuration file. To be able to safely recover services, we need to understand roles of different services, as stated in [12], including identifying safety-critical services. For example, even if a non-critical service such as `OmsCom` experiences a permanent fault, system can retain partial operationality. VR and OMS are standalone applications that have separate software architectures and are presumed to have relevant fault tolerance mechanisms.

`Trajectory Generator (TG)` failures: steps for recovery process are outlined in Fig. 3. If a failure occurs during movement, robot needs to be stopped with a ramp because of large masses for remotely operated equipment, but this feature is typically implemented either in mechanical design with brakes or in the service responsible for robot movements (`C4G` service) as part of standard fault handling. Recovery of interrupted trajectory would be possible, but is not desirable in order to avoid sudden unwanted movements.

`C4G` failures: service failure stops all manipulator movements and is recovered to a safe state. Service manager must terminate

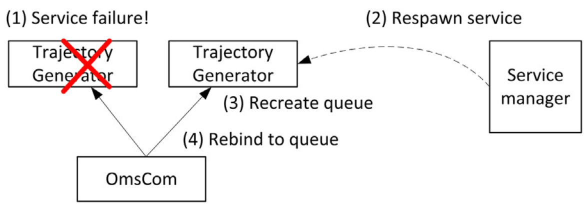


Fig. 3. Service recovering from failure; illustration simplified by showing only a single connected service. Solid arrows indicate the direction of data flow in queues.

also the TG service to avoid unintentional movements after restart. Service needs to recover state data for manipulator position, control mode and equipped tools. Position data can be recovered from sensors, other data can be recovered from GSB.

C4GJointDataPub failures: service can be used for VR visualization and haptic feedback, including virtual constraints and walls. System operation is possible without this service if operator has direct or video-based vision of the manipulator. However, since manipulator position data is used by support systems such as VR and collision detection, manipulator movement is stopped. Operation can resume safely after restart. State data restoration consists of reading joint status data from LSB queue.

OmsCom failures: service is non-critical in the sense that it is only used to initiate movement commands. In the case of permanent faults, manual operations with input device can be used for recovery. Command data is restored from GSB.

Compared to a non-critical application, such as a general-purpose OS in [10], first priority for a critical system is to guarantee safety of all operations. Fault in a critical service may compromise this goal so the system must move to a safe (fail-stop) state. In the case of a critical service, such as C4G, a fail-stop failure is preferred to undetected fault recovery. Therefore, critical services could be restarted to a *stopped* state. In the current implementation, services resume execution after restart (*start* command issued by service manager). In any case, services must be designed, implemented and tested to initialize to a safe state, even after recovery.

5.3. Recovery timing analysis

We analyze a situation where a failure occurs in the TG service. Failure is detected by the service manager and the service is restarted. Faulty service is presumed to report *error* status to the service manager (not shown in the figure) immediately after failure through heartbeat status updates that are sent during every service cycle. Cycle period T_i is 2 ms for C4G and TG services. Service manager uses $T_i = 1$ ms in the prototype. Worst-case jitter for periodic scheduling was tested on our previous study to have been around 6 μs, less than 1% of the RH service cycles [1].

To maintain correct operation, the C4G service must be able to read the updated trajectory values from TG during each period T_i. Therefore, to retain availability while recovering from service failure, combined time to detect failure (T_{detect}), time to restart ($T_{restart}$), and time to send and receive message (T_{com}) must be less than T_i plus possible spare time left from the previous cycle (T_{spare}).

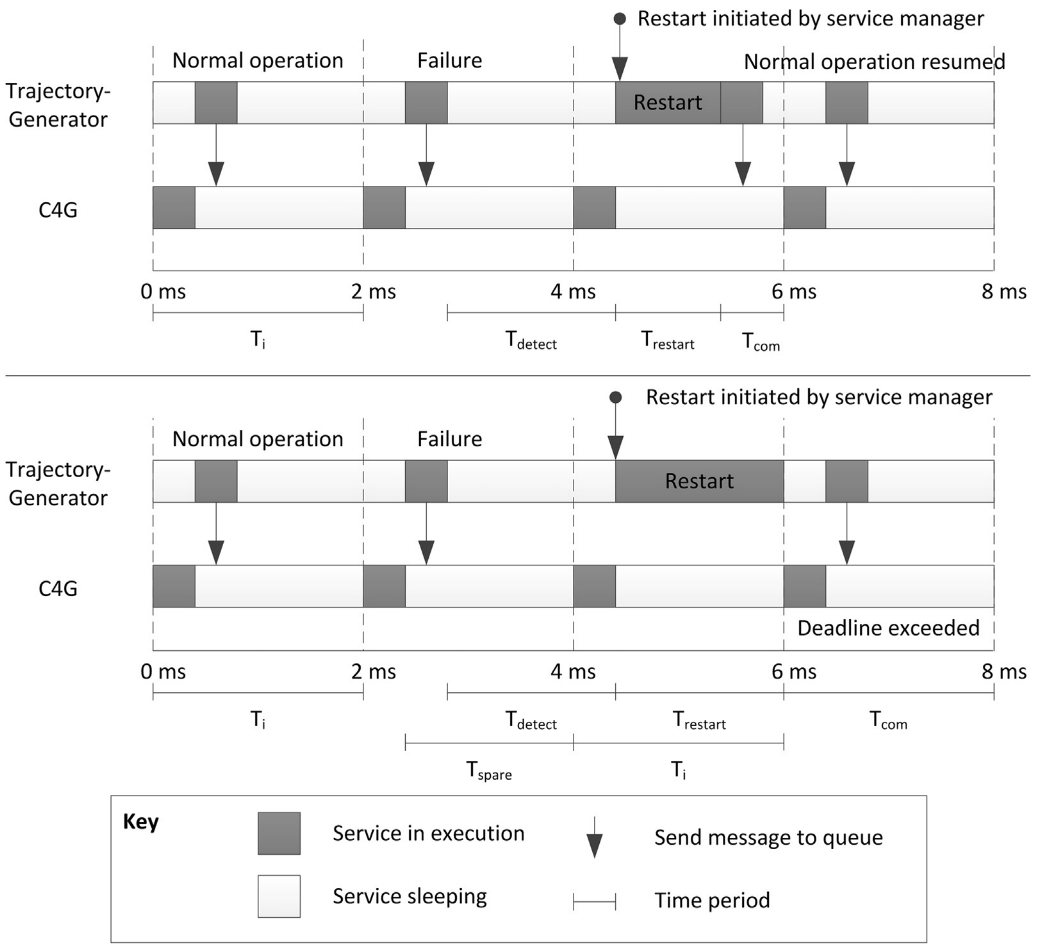


Fig. 4. Timing chart for service failure and restart. Upper scenario shows restart without exceeding communication deadlines and lower scenario restart with a missed deadline.

T_{spare} is included if the service has sent a correct message to C4G's LSB queue during the previous period, otherwise $T_{\text{spare}} = 0$:

$$T_{\text{detect}} + T_{\text{restart}} + T_{\text{com}} < T_{\text{i}} + T_{\text{spare}}$$

If true, operation can be resumed successfully. Otherwise C4G detects an exceeded communication deadline and the failure propagates. Fig. 4 illustrates both an optimal recovery situation and a "deadline exceeded" situation. C4G service has higher priority and is therefore executed first.

Successful recovery within time limits is largely dependent on fault detection time T_{detect} and restart time T_{restart}. Service manager cycle time is a trade-off point: if the service manager uses a fast cycle time ($<T_{\text{i}}$ of monitored services), it can react faster, but uses more CPU time. Alternatively, choosing a slow cycle time means that any service failure would automatically cause deadline miss ($T_{\text{detect}} > T_{\text{i}}$), forcing the system to a fail-safe state and causing a system failure.

5.4. Performance costs

Evaluation of performance costs of fault detection and recovery is challenging, since there is no comparable non-service based implementation available, making meaningful performance comparison problematic. However, other studies using similar approach with operating systems have estimated performance costs to be around 5–10% [10]. Taking into account the use of the service manager and communication queues to services, we estimate that the overall performance cost is around this order of magnitude. However, the exact number depends directly from the number of services, how often CPU and memory usage is checked and service manager task period T_{i}.

6. Conclusions

The decoupled architecture model – in this case the RTSOA – is one approach to managing challenges of complexity and scale. Based on the evaluation of our prototype implementation, it can support handling of transient faults and implementation of fault tolerance design patterns in critical real-time systems such as RH control systems. Although any architectural model cannot make the system automatically fault tolerant, it can provide tools for handling and mitigation of errors. A system based on a loosely coupled architecture can be in a safe state even after a service fault, without losing all system functionalities. The system is more robust to failures, providing mid-ground between binary "works" vs. "broken" options.

A large monolithic and tightly coupled application is difficult to verify, validate and maintain, whereas a service can be managed and verified individually, as long as the deployment environment is specified. Therefore, an approach based on loose coupling (e.g. service or component based) is recommended for ITER. Moreover, the RTSOA allows capabilities to be dynamically introduced to the system that were not initially planned. This is a major concern for a long-term research project such as ITER where the systems are likely to evolve to meet changing scientific and technical needs.

References

[1] P. Alho, J. Mattila, Real-time service-oriented architectures: a data-centric implementation for distributed and heterogeneous robotic system, in: 4th IFIP TC 10 International Embedded Systems Symposium, 2013, pp. 262–271.
[2] D. Hamilton, ITER Remote Handling Control System Design Handbook 2EGPEC v2. (2011) 3.
[3] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, Trans. Dependable Secure Comput. 1 (1) (2004) 11–33.
[4] T. Anderson, J. Knight, A framework for software fault tolerance in real-time systems, IEEE Trans. Softw. Eng. 3 (SE-9) (1983) 355–364.
[5] O. Gonzalez, H. Shrikumar, J.A. Stankovic, K. Ramamritham, Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling, in: Proceedings of the 1st IEEE Real-Time Systems Symposium, 1997, pp. 79–89.
[6] E. Strunk, J. Knight, Dependability through assured reconfiguration in embedded system software, IEEE Trans. Dependable Secure Comput. 3 (3) (2006) 172–187.
[7] L. Sha, Using simplicity to control complexity, IEEE Softw. 4 (18) (2001) 20–28.
[8] X. Liu, Q. Wang, S. Gopalakrishnan, W. He, L. Sha, H. Ding, et al., ORTEGA: an efficient and flexible online fault tolerance architecture for real-time control systems, IEEE Trans. Ind. Inform. 4 (4) (2008) 213–224.
[9] B.U. Kim, Y. Al-Nashif, S. Fayssal, S. Hariri, M. Yousif, Anomaly-based fault detection in pervasive computing system, in: Proceedings of the 5th International Conference on Pervasive Services, 2008, pp. 147–156.
[10] J. Herder, Building a Dependable Operating System: Fault Tolerance in MINIX 3, Vrije Universiteit, Netherlands, 2010.
[11] P.M. Melliar-Smith, L.E. Moser, Progress in real-time fault tolerance, in: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004, pp. 109–111.
[12] P. Alho, J. Mattila, Breaking down the requirements: reliability in remote handling software, Fusion Eng. Des. 88 (9–10) (2013) 1912–1915.
[13] L. Aha, K. Salminen, A. Hahto, H. Saarinen, J. Mattila, M. Siuko, et al., DTP2 control room operator and remote handing operation designer responsibilities and information available to them, Fusion Eng. Des. 86 (9–11) (2011) 2078–2081.
[14] R. Hanmer, Software rejuvenation, in: 17th Conference on Pattern Languages of Programs, 2010.

Publication 5

P. Alho and J. Rauhamäki, Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems, [In press], LNCS Transactions on Pattern Languages of Programming

© Springer. Printed with permission.

Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems

Pekka Alho[1], Jari Rauhamäki[2]

[1] Tampere University of Technology, Dept. of Intelligent Hydraulics and Automation, Finland
pekka.alho@tut.fi
[2] Tampere University of Technology, Dept. of Automation Science and Engineering, Finland
jari.rauhamaki@tut.fi

1 Introduction

Distributed control systems are continuously gaining importance, as more and more devices and machines are equipped with embedded systems that control their operation. Computers in these control systems are increasingly more powerful and networked, providing intelligence and interoperability. Examples of such systems range from large mobile machines to groups of robots and intelligent sensor networks. These cyber-physical systems (CPS) interact with environment and physical processes, influencing many parts of our lives either directly or indirectly. Therefore they need to be *dependable*, which can be measured with the attributes of availability, reliability, safety, integrity and maintainability [1]. However, with the increased functionality and intelligence, the complexity of these systems is also increased, meaning that the development process becomes more demanding and dependability becomes more costly to achieve and verify. Another significant feature of CPS is that they often have strict timing constraints, which may put limitations on the architecture.

Many critical systems that have failed catastrophically are well-known – examples such as Therac-25 radiation therapy machine and the explosion of Ariane 5 rocket are infamous, whereas highly reliable systems receive little recognition, even though their study might give valuable ideas for the design and architecture of new software. One example of such systems can be found in telephony applications, namely Ericsson AXD301 Asynchronous Transfer Mode (ATM) switches that achieved nine nines (99.9999999%) service availability, running software written in Erlang [2]. Erlang's highly decoupled actor model and fault handling based on supervisors have inspired especially LET IT CRASH and SERVICE MANAGER patterns found in this paper.

This paper presents three software patterns that can be used to improve control system dependability – the third pattern is called DATA-CENTRIC ARCHITECTURE – and shows how they fit in the existing literature by addressing the specific needs of CPS. The approach promoted by these three patterns is based on implementing a decoupled architectural design with supporting fault mitigation and handling. The decoupled architecture can also be used to gradually introduce additional fault tolerance solutions such as checkpointing and rejuvenation to the system, until a sufficient level of reliability has been achieved [3]. Our patterns were originally encountered in the research of remote handling control systems for robotic manipulators, but all patterns have examples of other known uses as well. These examples are presented in the corresponding sections of the patterns.

One reason why development of CPS is difficult is because the systems typically consist of dynamic service chains that operate on wide range of platforms, which complicates management of end-to-end deadlines. Moreover, modern middleware provide capabilities to flexibly change service deployment on these subsystems, but some configurations may be inefficient or even unusable if communication links become overloaded. While adaptability has benefits, these uncertainties nevertheless complicate assurance of reliability and predictability of the system. Therefore, CPS benefit from a design that makes the overall system more robust, whereas more traditional fault tolerance solutions, such as hardware redundancy, are arguably better suited for static safety-critical subsystems.

Data-centric approach is one way to increase decoupling between communicating units. However, data-centric design as a central communication paradigm, as well as the concept of CPS, is still fairly novel in the domain of distributed control systems. Although control systems are by nature data-centric (read sensor data and desired output, send actuator command, etc.), this has usually been from point A to point B. The patterns in this paper capture some of the ways that reliability-related challenges faced in developing more intelligent and adaptable distributed control systems have been solved. Next chapter shows how our patterns fit the gaps in the existing pattern literature, by addressing needs specific to CPS.

2 Context of the Patterns

Fault tolerance cannot be implemented without redundancy of some kind. To have fault tolerance for e.g. computer failures, we would need at least two computers – if one fails the other one can detect the error and try to correct it. Software faults on the other hand are typically development faults, which are harder to detect and correct than hardware faults. To have good coverage for software faults, diverse redundancy (e.g. N-version programming) is needed, but it has been criticized of being susceptible to common mode failures [4]. Moreover, development costs for design diversity are often seen as prohibitive.

Patterns in this paper present an alternative approach to fault tolerance, based on dividing the system into highly decoupled modules and implementing lightweight form of fault tolerance. We present an architectural pattern called DATA-CENTRIC ARCHITECTURE as one way to achieve a high level of decoupling. One of the key points of decoupling is that it should by itself improve reliability by limiting fault propagation and improving modularity and understandability of the system. In a way, modular approach can be seen similar to compartmentalization of ships – without compartments, every leak can sink the ship. An example of a software system that uses modularity to successfully implement fault isolation and resilience is the MINIX 3 operating system released in 2005 [5]. Driver management of MINIX 3 is presented as one of the known uses of SERVICE MANAGER.

Modular and decoupled architecture can also be used to implement other reliability-improving patterns like SERVICE MANAGER and LET IT CRASH documented in this paper or other well-known patterns like LEAKY BUCKET COUNTER [6], WATCHDOG [6] [7], etc. The short descriptions of the patterns presented in this paper are listed in the Table 1. List of all referenced patterns with descriptions can be found in an appendix.

Table 1. Pattern descriptions

Pattern	**Description**
DATA-CENTRIC ARCHITECTURE	How to implement reliable and scalable distributed control system? Build the system from autonomous modules that communicate by sharing data that is based on a well-designed and consistent data model.
SERVICE MANAGER	How to detect faults and restart modules or processes after a failure? Implement a service manager that can monitor, start and stop modules.
LET IT CRASH	How to react to failures without crashing the whole system? Flush the corrupted state by "crashing" the process instead of writing extensive error handling code. Let some other process like service manager do the error recovery e.g. by restarting the crashed process.

DATA-CENTRIC ARCHITECTURE provides the decoupled architectural model needed to use LET IT CRASH for fault handling. The SERVICE MANAGER pattern provides a way for trying recovery after failures, in addition to providing error detection and monitoring. The idea of crashing a process suggested by LET IT CRASH may sound like a risky action to take. However, the idea is to offer recovery from transient physical and interaction faults (sometimes called Heisenbugs), ability to keep the system as a whole functioning, even if some internal process would crash, and possibility to hot-swap code and bug-fixes. The downside of this approach is of course that it is not suited for fail-operate systems such as flight controllers that must be operational all the time – this type of systems would be the right domain to apply design diversity.

In order to show how these patterns fit the existing literature, we have built a pattern language for fault tolerance in CPS that references related patterns and pattern languages, shown in Fig. 1. Entry point to the language is the need for introducing fault tolerance to the system in order to improve its dependability. The three main starting points are MINIMIZE HUMAN INTERVENTION [6], REDUNDANCY [6] and UNITS OF MITIGATION [6], but the REDUNDANCY branch has been not been explored in-depth since it presents somewhat different approach from the three patterns found in this paper. Recovery types have also been condensed to a single concept. Some of the connections presented in the original sources have been reorganized in order to better fit in this context, and the figure shows only one of the possible combinations of the patterns. Connections to other patterns and pattern languages can be checked from the references in Table 2 found in the appendix.

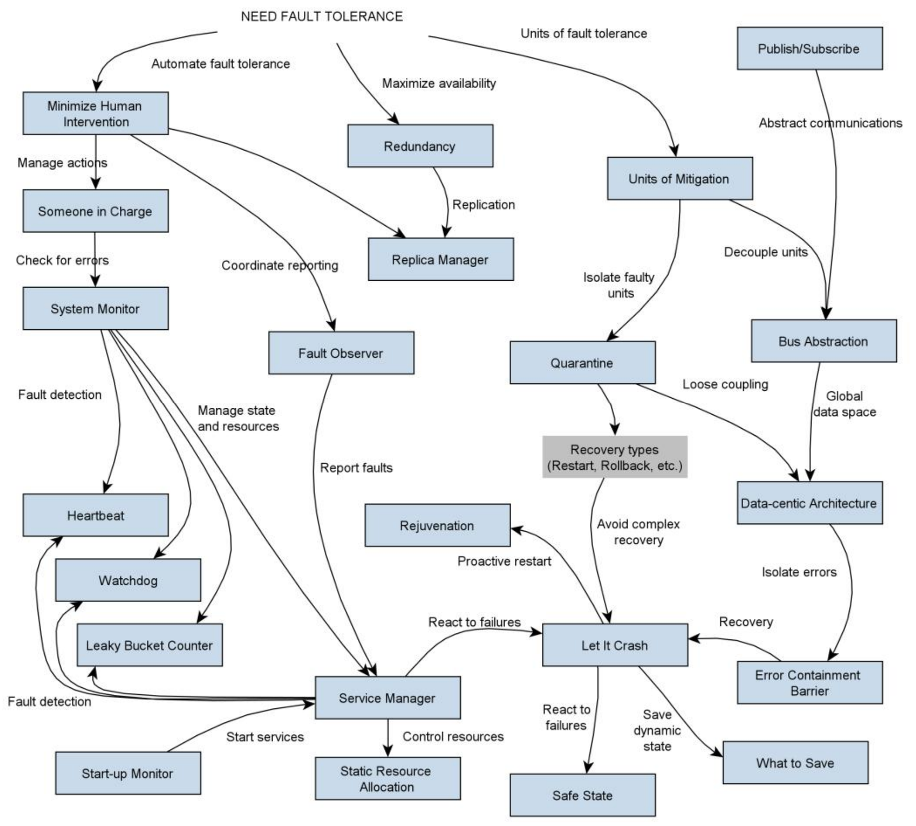


Fig. 1. Pattern language for fault tolerance in cyber-physical systems

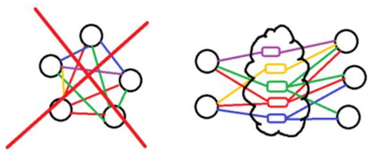
The pattern language shows how the patterns presented in this paper build on top of existing patterns and support implementing fault recovery and SAFE STATE [7] in CPS. Gaps identified in the pattern language are related to CPS being networked systems with real-time requirements and safety concerns. Fault handling needs extra attention since control system cannot try complex fault recovery routines that could have unforeseen consequences. Instead, a better approach is to QUARANTINE [6] the faults locally and stop their propagation, even if that would mean losing some functionality either temporarily or permanently.

There are several existing patterns that have similar purposes as SERVICE MANAGER, such as FAULT OBSERVER [6], REPLICA MANAGER [15], SERVICE CONFIGURATOR [16] and SYSTEM MONITOR [6]. However, CPS benefit from more active management component that can try to react to the failures within system specifications – because they typically have timing-critical control loops and state machines – to mitigate faults and stop their propagation in the system.

Finally, to implement the fault handling, units need a loosely coupled architecture that is robust to failures and supports fault detection. The patterns in the pattern language work together by building on the top of features provided by other patterns as shown in Fig. 1 but all of the patterns can also be used in other contexts besides distributed control systems. Other well-known fault tolerance patterns also work well in combination with the presented patterns. Besides the patterns presented here, other typical examples related to reliability of CPS include implementation for fault detection, fault reporting, sending and acknowledgement of commands, etc. but have been left out of this paper.

3 Patterns

3.1 Data-Centric Architecture



Intent. Implement an architecture based on autonomous modules (e.g. services, processes or applications) that communicate by sharing properly modeled data.

Context. You are developing a distributed control system that consists of several subsystems and needs to interact with other heterogeneous systems such as mobile machines or plant systems. The system has CPU and memory resources available to run an operating system – rather than being based on a basic time-triggered scheduler used in resource-constrained embedded systems. Failures in control functions (e.g. boom or manipulator control) may cause damage to the environment and equipment, meaning that some subsystems may be categorized as safety or mission-critical.

Problem. How to implement a reliable and scalable distributed control system?

Forces.

- *Throughput*: Some time-critical data such as sensor measurements may be updated with short period, producing large amounts of communication.
- *Scalability*: New nodes and subsystems can join the system any time; assumptions about interfaces between modules should be minimized.
- *Changeability*: System configuration and functionality might change. Changing interfaces in a tightly coupled system requires code changes at both ends (and at all clients), so assumptions about expected behavior should be minimized. Point-to-point protocol based client-server architectures (e.g. sockets or remote method invocation) are not ideal because of complexity and coupling introduced.
- *Maintainability and long expected life-cycle*: The control system has long expected lifetime and needs to be maintainable and extensible in the future – if subsystems

are added or substituted, changes to existing modules need to be minimized. System should be easy to understand and modify without breaking it.

- *Maintainability*: Implementing custom communication channels and protocols should be avoided.
- *Reusability*: Same modules could be used in other control system implementations.
- *Interoperability*: Distributed control systems consist of and/or need to communicate with heterogeneous platforms.
- *Testability*: Tightly coupled modules are difficult to test because they are more dependent on other modules.
- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures.
- *Reliability*: A single fault in the control system software should not endanger functionality of the whole system (i.e. no single point of failures).
- *Reliability*: Faults should be detected and their propagation prevented.
- *Real-time performance*: Control system interacts with the real world and needs to react in a deterministic manner.
- *Safety*: Need to detect if a module has crashed or is down (not releasing new information) so that the system can enter SAFE STATE in a controlled fashion. Safety-critical and non-safety-critical subsystems cannot be tightly coupled, since errors may propagate.
- *Quality of service*: Different subsystems may have different requirements for quality of service[1] (QoS) policies. There is an impedance mismatch between e.g. real-time control systems that operate on a timescale of milliseconds and enterprise/high level systems that are several orders of magnitude slower.

Solution. Build the system from autonomous modules that communicate by sharing data that is based on a well-designed and consistent data model.

Implement communication between modules as sharing of data, instead of sending point-to-point messages or request-reply service calls. Data-centric approach is based on minimizing dependencies between modules by removing direct inter-module references and hiding module-specific behavior. This can be achieved by delegating data-handling to a middleware solution that supports publishing of data to topics in a distributed data space and making applications tolerate unavailability of dependencies. Asynchronous messaging is a well-known way to reduce coupling of systems, but data-centric approach increases this further by removing the concept of recipient from the publisher.

Modules should be built to be autonomous and not expect that other services are always started in a specific order and available. Service/module composition may change during runtime; there are patterns for managing the configurations (e.g. SERVICE CONFIGURATOR). Developer should avoid assumption about state of the dependencies, i.e. other services. Dependencies may not always be available and this

[1] QoS policies provide the ability to specify various parameters such as rate of publication, rate of subscription, reliability, data lifespan, transport priority, etc. to control end-to-end connection properties. Policies can be matched on a request vs. offered basis.

must be taken into account in the application code so that the service will react accordingly if its dependency is down because it is in the process of starting, failed, manually shut down, etc.

Management of the global data space is externalized to the middleware that implements a topic-based PUBLISH/SUBSCRIBE model. Middleware disseminates data to all participating nodes, acting as a single source of up-to-date system-wide state information. It acts as a single source of up-to-date state information in the system, instead of applications managing state separately.

Modules do not need to know recipients of the data when publishing it, which reduces coupling. Instead of sending data directly to a recipient, it is published to a topic. Data can be e.g. sensor measurements, events or commands, but it must follow a shared information model which is represented as topics in the actual system implementation. Publishers register as data writers to a topic and interested subscribers can join the topic as data readers. Single topic can have multiple instances, which are identified by a key value, and can have multiple readers and writers, as shown in Fig. 2.

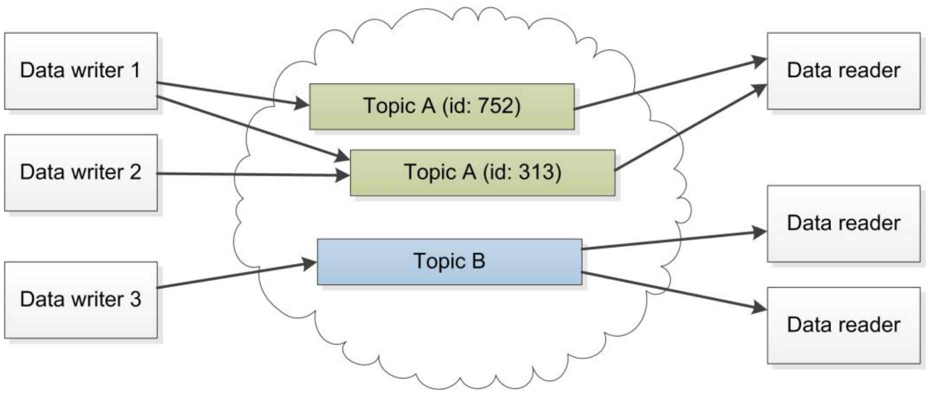


Fig. 2. Data is published to topics that can have multiple data writers and readers. Topic A has two instances, identified by the id number key value.

Since the middleware decouples the modules, publisher might assume that a subscriber is listening when it is not. If a publisher needs to know that data has been received, it should monitor status of the subscriber (published to another topic). This might be true, for example, with commands sequences where commands must be completed before sending the next one.

Instead of designing callable methods for components, you must design how to represent the state of the system and the external or internal events that can affect it. This is captured in a common data model, which contains the essential elements of the physical system and application logic. Conceptually the data model is similar to class diagram in object-oriented programming since it consists of identifying entity types, which have data attributes assigned to them, and associations. The difference is that the data model focuses on data instead of behavior. Data model ensures that communication between modules is unambiguous and interoperable. Appropriate QoS attributes can also be attached to the data model.

Communication and application logic are separated since network communications are delegated to a “data bus” formed by the publish/subscribe middleware (Fig. 3), so that the application logic can focus on the core functionality. Middleware takes care of maintaining the data up-to-date, automatically updating new nodes that join. If the middleware uses a central server as a message BROKER [8], it becomes a single-point-of-failure and possibly a bottleneck. Therefore, choose a decentralized middleware solution, if possible, to avoid this problem

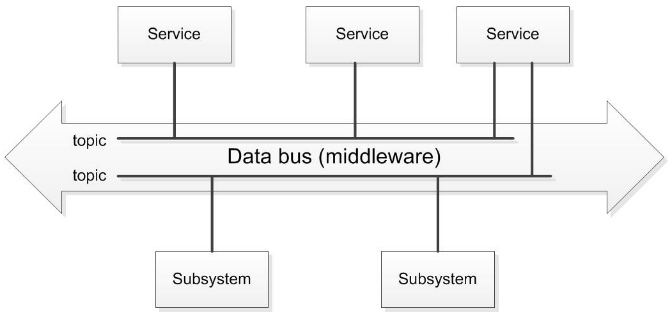


Fig. 3. Middleware implementation as a virtual data bus that has no central components or brokers. Services and subsystems can join topics as publishers and/or subscribers.

Granularity of modules and interactions are important design decisions that affect failure consequences, performance and reusability of modules. Fine-grained autonomous modules (large number of smaller modules) are easier to reuse and make it easier to isolate faults, but limiting the number of modules and interactions helps to avoid potential performance issues. Modules communicating only locally can be more fine-grained than ones communicating remotely, although the data model should not include location dependencies. Fine-grained interactions give more flexibility, as it will be possible to treat data items separately. Coarse-grained interactions are usually preferred between remote modules in order to avoid overhead, but data that is updated rapidly should be separated from data with slow update rates in order to avoid unnecessary use of bandwidth. Further control over system granularity can be achieved by dividing it to domains.

Compared to message-centric publish/subscribe, one of the differences in data-centric model is that data samples published to topics are transparent to the middleware. In message-centric model, middleware does not know or care about message contents and communication is point-to-point by nature which introduces coupling between modules, although some message-centric middleware also support publishing of messages to topics. Data-centric communication is based on a data model that expresses the state of the system. Since data is interpreted through the model, it is platform-independent and middleware can prioritize, filter and manage the data based on its contents and QoS policies, replacing part of the application logic. Although developing a data model adds to upfront planning efforts, systems with long-term lifecycles benefit in terms of maintainability and evolvability.

Consequences.
+ Publishers do not need to know about subscribers.
+ Interoperability between heterogeneous platforms since data is interpreted through the data model.
+ Decoupled design provides error confinement and other benefits such as improved maintainability.
+ Modules can be changed dynamically because late joiners receive new data automatically; ability to hot-swap code can be easily implemented.
+ An application or subsystem can be shut down without impacting the overall operation of the system.
+ Network transport layer is abstracted as communications are externalized to middleware, which reduces communication related code and simplifies implementation.
+ Gives developers control of data delivery with QoS management; QoS can be used e.g. to guarantee reliable delivery (eventually) or that available data is kept up-to-date with best effort. Former would be useful for sending status changes or commands, whereas latter could be used for sensor measurement for which guaranteeing delivery of outdated samples makes no sense.
+ Reusability is improved since modules are using shared memory and have their own namespaces, etc.
+ Publish/subscribe based middleware scales effectively since recipients for data are not explicitly defined.
+ Performance gains can be achieved on multi-core machines since modules can be easily parallelized and they communicate asynchronously.
+/- Needs good and consistent data models that must be managed and maintained, but a well-thought-out data model improves maintainability and makes reuse of the code easier.
- A publisher might assume that a subscriber is listening when it is not.
- Sending of commands is not as straightforward as in client-server architectures since commands need to be parsed from the data. However, interactions can be modeled as operation codes sent between two modules.
- Parsing of data complicates debugging because it adds another potential source for faults. If data is parsed incorrectly, origin of fault may not be self-evident.
- Extra code needed when compared to more monolithic applications since modules cannot presume that all dependencies are started in specific order and available all the time.
- Serialization and deserialization of the data structures for transmission may add overhead.
- Faults in the middleware itself complicate testing and are hard to detect.
- Middleware solutions add some overhead to message size and use system resources.
- Possible vendor lock-in to the middleware provider.

Known uses. Data Distribution Service for Real-Time Systems (DDS) is decentralized and data-centric middleware based on the publish/subscribe model. DDS is aimed at mission-critical and embedded systems that have strict performance and reliability requirements. Therefore, its implementations have typically been optimized and tested to suit the needs of these systems. DDS is used as the information backbone in the Thales TACTICOS naval combat management system that integrates various subsystems such as weapons, sensors, counter measures, communication, navigation, etc. to a "system of systems". Applications are distributed dynamically over a pool of computers in order to provide combat survivability and avoid single-point-of-failures. System configuration can be adapted for use in various mission configurations, on-board & simulator training, and different ship types.

Related Patterns. BUS ABSTRACTION [7], and PUBLISHER-SUBSCRIBER.

MEDIATOR [9] increases decoupling in a similar fashion, but is designed to decrease connections between objects locally.

Decoupled modules in DATA-CENTRIC ARCHITECTURE act as UNITS OF MITIGATION, parts that contain errors and error recovery.

3.2 Service Manager

Also Known as. SUPERVISOR.

Intent. Service manager starts, stops, and monitors processes locally and takes care of resource allocation for systems that need high availability and real-time performance.

Context. You are developing a system with highly decoupled architecture (e.g. using DATA-CENTRIC ARCHITECTURE) that consists of large number of processes or tasks (services). These processes have dependencies and therefore need to be started in specific order. Process composition may change dynamically during runtime because your system will have intelligent functionality, it needs to adapt to new situations, or different functionalities need to be tested without stopping/restarting the whole system.

You know rough upper-limit estimates for how much system resources such as memory and CPU time the processes will use.

The system has long expected life-cycle. It is likely to be deployed on a remote location, for example a forest or a control cubicle, making direct physical interaction with the system a bothersome task.

If you have a real-time operating system and a task gets stuck in a while loop or some other control structure, it freezes the whole system as other lower priority processes (including input devices and network connections) cannot get CPU time. In this case, the only option is usually to restart the whole computer manually.

Problem. How to ensure that all dynamic modules in your control system are running correctly and you have enough system resources to achieve deterministic real-time performance?

Forces.

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, in order to able to use other parts of the system that are not connected to the failed subsystem. The system must detect faults and try to mitigate them automatically. If a failure needs immediate reaction from a human operator, the system will not scale cost-efficiently and reliably.
- *Data logging/testability*: If a process fails, the failure should be detected and logged.
- *Real-time performance*: The control system needs to respond in a deterministic and predicable manner. Predictability includes system behavior when a fault is triggered.
- *System resources*: Control systems are typically deployed on embedded devices that have limited memory and CPU resources available. They may need to be monitored in order to guarantee the real-time performance of the system.

Solution. Implement a service manager that can monitor, start and stop local modules.

Create a local parent process (the service manager) that is responsible for starting, stopping and monitoring its child processes. The basic idea of the service manager is to keep its child processes alive by restarting them when necessary. Location of the service manager is on the same computer as the child processes in order to keep implementation simple. Therefore, all computers in the system need their own, independently functioning, service managers. The service manager is given the highest process priority in the system or put in the kernel so that a faulty real-time process cannot prevent it from functioning by consuming all available CPU time.

Start the child processes based on a fixed order or a dependency table read from a configuration file, similar to START-UP MONITOR [7], and/or implement a user interface that can be used to start and stop processes.

Use the service manager to allocate resources like CPU time and memory for the child processes and monitor their use. Expected maximum resource consumption can be specified in the same configuration file that is used for starting services. New processes are not started if there are not enough resources available. If a process consumes more resources than expected, it can be restarted, triggering error handling according to the LET IT CRASH pattern. Resource use can be followed e.g. with proc filesystem or *getrusage* call in Unix-like systems.

Since one of the key functionalities of service manager is to monitor processes for failures, error detection can be based on additional or alternative techniques besides resource monitoring. This can be done with e.g. operating system features, HEARTBEAT [6] [7] or WATCHDOG.

If fault recovery fails, service manager should mitigate the fault by QUARANTINING the faulty module. If the fault is persistent, LEAKY BUCKET COUNTER can be used to limit the number of restarts.

If the service manager is deployed on a system that uses DATA-CENTRIC ARCHITECTURE, service startup interfaces can be implemented through the middleware. Since the middleware abstracts the location of the data, it can be used to remotely start dependencies. For example, service manager SM_A must start a service called S1. However, it has a dependency called S2 which cannot be found locally, so the service manager publishes a start request for S2. A second service manager SM_B on another computer notices the request, starts S2 and publishes information about the successful startup. SM_A receives information that S2 is available and starts S1.

The implementation for service manager needs to be kept fairly simple, since it acts as a single point of failure locally. This conflicts with the need to use of configuration files, making resource checks, and providing user interface, so they should be based on external components or libraries that have been proven in use.

Consequences.

+ Detects and initializes recovery from transient faults that cause a process to consume too much system resources or become unresponsive.

+ Ensures other processes stay alive and have sufficient resources.

+ Simplifies starting procedure of complex system that consists of large number of processes, making possible to start and stop a large number of processes automatically and in a specific order.

+ Cost-efficiency: the same service manager implementation can be reused on several systems.

+ Supports logging and reporting of errors so that they do not go undetected.

- Cannot detect faults that cause erroneous output for monitored components.

- Cannot recover persistent faults such as development and physical faults, e.g. computer failures.

- Potential single point of failure that may stop the entire system from working if services are incorrectly terminated.

- Restarting a service may cause the system to behave in non-deterministic way and miss deadlines, which is a failure for a hard real-time system. However, it should be noted that the failure would have likely caused the system to miss the deadlines or exhibit some other unwanted behavior even without service restart.

- Resource utilization needs to be estimated for the processes in order to set limits.

- Service manager uses system resources and may reduce performance.

Known uses. Node State Manager (NSM) for in-vehicle infotainment systems: GENIVI Alliance (http://genivi.org/) is a non-profit consortium promoting open-source platform for the automotive in-vehicle infotainment industry. Reference implementation of the platform includes NSM that is responsible for information regarding the current running state of the embedded system. NSM component collates information from multiple sources and uses this to determine the current state of the node. It is the highest level of escalation on the node and will therefore command the reset and supply control logic. It is notified of errors and other status signals from components that are responsible for monitoring system health in different ways. NSM also provides shutdown management by signaling applications to shut down.

MINIX 3.0 driver manager: MINIX is a POSIX conformant operating system, based on a microkernel that has minimal amount of software executing in the kernel mode. Most of the operating system runs in user mode as independent processes, including processes for the file system, process manager, and device drivers. The system uses a special component known as the driver manager to monitor and control all services and drivers in the system [5]. Driver manager is the parent process for all components, so it can detect their crashes (based on POSIX signals). Additionally the driver manager can check the status of selected drivers periodically using HEARTBEAT messages. When a failure is detected, the driver manager automatically replaces the malfunctioning component with a fresh copy without needing to reboot the computer. The driver manager can also be explicitly instructed to replace a malfunctioning component with a new one.

Monit (http://mmonit.com/monit/) is an open source tool that can function as a service manager in non-real time systems. Following code listing shows an example configuration for Spamassassin daemon that restarts the daemon if its memory or CPU usage exceeds 50% for 5 monitoring cycles:

```
check process spamd with pidfile /var/run/spamd.pid
   start program = "/etc/init.d/spamd start"
   stop  program = "/etc/init.d/spamd stop"
   if 5 restarts within 5 cycles then timeout
   if cpu usage > 50% for 5 cycles then restart
   if mem usage > 50% for 5 cycles then restart
   depends on spamd_bin
   depends on spamd_rc
```

Related Patterns. FAULT OBSERVER [6], HEARTBEAT, SAFE STATE, SOMEONE IN CHARGE [6], START-UP MONITOR, STATIC RESOURCE ALLOCATION [7], and WATCHDOG.

To see how to design an application in a way that it can be easily restarted at any time, see LET IT CRASH.

MANAGER design pattern [10] can be used to manage multiple objects of same type – the idea is similar to SERVICE MANAGER (keep track of entities and provide unified interface for them) but the MANAGER focuses on different scope, i.e. managing entities (objects) of the same type and does not include resource monitoring or fault detection.

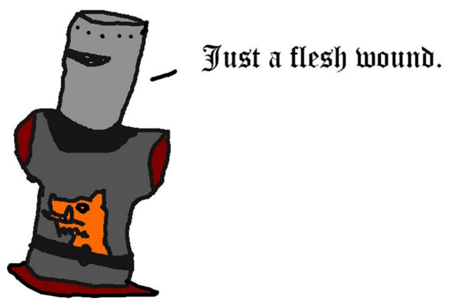
SERVICE CONFIGURATOR is very similar to SERVICE MANAGER in many regards. However, the main use cases for SERVICE CONFIGURATOR are, as the name implies, related to reconfiguration of the system, whereas SERVICE MANAGER aims to improve fault tolerance of the system by managing (monitoring & restarting) services. In CPS, dynamic reconfiguration of the system can often be undesirable due to possible safety implications. An example of SERVICE CONFIGURATOR is the device driver system in modern OSs. A comparable implementation of the to SERVICE MANAGER is the driver manager in MINIX, which adds the management (fault detection & restart) aspect to device drivers.

SERVICE MANAGER can QUARANTINE a module by stopping it if a fault is detected. and recovery does not work.

SYSTEM MONITOR [6] can be used to study behavior of system or specific tasks and make sure they operate correctly, e.g. by using HEARTBEAT or WATCHDOG. If a monitored task stops, SYSTEM MONITOR reports the error. Compared to it, SERVICE MANAGER has a more active role in managing the tasks.

REPLICA MANAGER [15] provides the necessary mechanisms for the replica management in systems that use active node replication, i.e. REDUNDANCY, whereas SERVICE MANAGER does not make presumptions about the use of redundancy.

3.3 Let It Crash



Also Known as. CRASH-ONLY [11], FAIL-FAST, LET IT FAIL or OFFENSIVE PROGRAMMING.

Intent. Avoid complex error handling for unspecified errors. Instead, crash the process and leave error handling for other processes in order to build a robust system that handles errors internally and does not go down as a whole.

Context. You are developing a distributed control system that consists of several processes and subsystems that need to cooperate to complete tasks.

DATA-CENTRIC ARCHITECTURE or some other asynchronous decoupled architectural design has been utilized so that processes are not using shared memory.

Some subsystems might have safety-critical functionality, but it is possible to move the system to SAFE STATE (i.e. the system is fail-safe type, not fail-operate). The system has dynamic state information from the user inputs and working environment in the process memory, e.g. tool tracking data in the case of a robot manipulator. This state data needs to be recovered after a failure.

The system has a mechanism to supervise and restart the processes. This can be implemented at operating system, programming language or framework level, e.g. with the SERVICE MANAGER.

Problem. How to implement lightweight form of error handling that improves reliability and predictability?

Forces.

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, since degraded functionality is better than no functionality. In case of a fault, only minimal part of the system should be affected. Recovery from failures should happen without human intervention and with minimal downtime.
- *Reliability:* Generation of incorrect outputs should be prevented, otherwise errors may propagate and the system could cause damage to the environment.
- *Safety*: If an error is detected, any functionality using the affected process should be stopped and taken to a safe state in order to prevent and minimize damages.
- *Cost-efficiency*: Design diverse fault tolerance techniques are oversized or impractical for the application, but the system needs to be able to recover from errors.
- *Real-time performance*: Control system needs to react within a certain time-limit; exceeding the time-limit causes a failure.
- *Predictability*: The system should behave in a consistent manner. If the process tries to repair its corrupted state, behavior of the system cannot be predicted, which complicates debugging and verification of reliability. Predictability includes system behavior when a fault is triggered.
- *Recovery*: Because it is impossible to foresee all possible faults, specifications do not cover all possible error situations. Various error situations occur seldom, are difficult to handle and non-trivial to simulate in testing [11]. If the programmers try to implement recovery, they will make ad hoc decisions not based on the specifications (i.e. they cannot know how the error should be handled), possibly causing unwanted and undocumented behavior.

Solution. Make processes crash-safe and fast to recover; flush corrupted state by "crashing" the process instead of writing extensive error handling code.

Commodore 64, DOS machines and other old computers were designed to be shut down by simply turning the power off, essentially crashing the system. On the other hand, if an operating system caches disk data in memory, workstation crash may corrupt the file system, which is inconvenient and slow to repair. Control system processes and subsystems should also be designed to be easily terminated and recoverable with a simple recovery path if an error is detected, instead of guessing how error recovery should be attempted, possibly corrupting program state further and causing unpredictable behavior.

Therefore, implement error handling by terminating the process that has encountered the error. Only program extended error recovery routines if they are based on the specification or it is self-evident how the error should be handled – otherwise crash the process. However, only the module or process where the error is should be crashed, not the whole system.

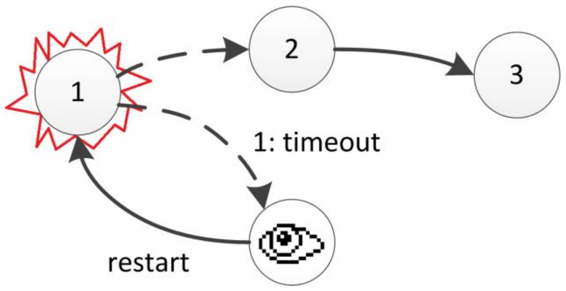


Fig. 4. Process 1 encounters an error and dies, after which it is restarted by the service manager, represented as an eye. If the process 2 detects a deadline overrun, it needs to stop, potentially interrupting process 3, and wait until process 1 is active again before resuming work. Alternatively the process 2 does not notice any deadline overruns and continues working normally.

Processes that have been designed with LET IT CRASH can 1) help to find faults, by making them more visible ("offensive programming"), 2) prevent software degradation with REJUVENATION [11][14], and 3) be used to implement fault tolerance (recovery from faults). In the final case it is possible to perform recovery without affecting service availability if the recovery process is fast enough. Recovery (and rejuvenation) needs an external entity to initiate the procedure, since the process itself has crashed (see Fig. 4). This pattern focuses mostly on the final case since it is more problematic to implement correctly.

You have a monitoring layer that can supervise and recover processes e.g. by restarting. To have the monitoring layer detect a failure, you may need to implement timeouts or the faulty process must terminate upon encountering an error in order to send a signal for the monitoring layer (parent process knows the liveliness state of its child processes). How the error is detected in the first place is not part of LET IT CRASH, but contract programming or error checks could be used. Abnormal program termination can be forced e.g. by using *abort()* or *raise(SIGSEGV).* If the monitoring layer has implemented failure detection – based on watchdog, heartbeat, etc. – it can also hard-fail the service using e.g. *kill(pid, SIGTERM)*. This might be necessary if the process is incapable of detecting its own fault.

Error recovery is performed by restarting the process. Therefore, make processes fast and easy to restart in order to minimize service failures and downtime. To keep recovery path simple, use the single responsibility principle, thereby minimizing responsibilities of a single process. If the process encounters an error and crashes, it might be possible to recover from the error without causing deadline misses for other processes and tripping the system to a SAFE STATE. However, if a control loop has a period of e.g. 1 ms and restarting of a process that provides information for the loop takes several milliseconds, control loop execution will be interrupted.

LET IT CRASH does not mean that error handling or exception handling should not be implemented at all. Indeed, sanity checks and error handling are essential for control systems and should be implemented to prepare for exceptional (but expected) circumstances, such as write operation failures or unavailable dependencies. LET IT CRASH, on the other hand, is applicable in situations where the program experiences an unexpected failure and cannot reliably perform its function. This can happen due to programmer errors, complex interaction faults, intermittent faults, etc.

Recovery paths can be tested extensively by terminating the system forcibly every time it needs to be shut down or restarted, instead of letting it run through a normal shutdown process. This forces the system to do a recovery during the startup

Make processes crash-safe. Processes typically handle three types of state data: dynamic, static, and internal. Internal state is related to current computations and is usually discarded after use. If a process crashes, you must think if you want to recycle its internal state. If you recycle everything you risk hitting the exact same fault again and crashing, so it might be reasonable to recycle only parts of this state. Static state is configuration data that can be easily recovered or read from other processes. Finally, the dynamic state data is generated as the program is executed by reading user inputs, interacting with other processes and environment, etc. Some of it can be computed from other data or read directly from sensors, but rest cannot be reconstructed. This data must be protected by using checkpointing, journaling or some other form of dedicated state store, for example databases and distributed data structures. To implement this, you must know WHAT TO SAVE [6].

Implement a reporting functionality that reports failures so that they do not go unnoticed. Failure information can be forwarded e.g. by using a service manager or supervisors to send NOTIFICATION messages [12].

The corollary to the LET IT CRASH approach is that you must design your software to be ready for processes failing. There is now a possibility that a dependency is not available because it has been crashed and is being restarted. To detect this situation, add timeouts or appropriate QoS policies to interactions between components. If a timeout is triggered, move the system to a SAFE STATE. Normal operation can be resumed when dependencies are back online. A missing dependency is therefore not considered to be an error that would necessitate a crash.

Consequences.
+ Enables simple error handling & recovery; avoids complex error handling constructs in code, therefore improving predictability of the system.

+ Cost-effective (lightweight) form of fault tolerance that does not require use of redundancy.
+ Allows error handling to be implemented separately (externally) from the business logic, e.g. with supervisors.
+ Supports recovery from transient faults since a restart is usually enough to handle them.
+ Possible to achieve high availability (for the system as a whole, not necessary for all services provided by the system).
+ Complements other fault tolerant designs such as REDUNDANCY and REJUVENATION.
+ Processes can be updated to new versions on-the-fly, since the old process can be killed and replaced using the normal recovery path.
+ Limits error propagation to other parts of the system (babbling idiot failure) by acting as an ERROR CONTAINMENT BARRIER [6].
+ Errors are less likely to cause the system to perform unpredictable and potentially dangerous or irreversible operations.
+ Finding faults should be easier, since they are made more visible by crashing and reporting.
- Availability of some services provided by the system is lower (when compared to redundant fault tolerance solutions) – on the other hand availability of other unrelated services provided by the system should be unaffected.
- Cannot mitigate persistent faults.
- Processes need additional code to react to missing dependencies (i.e. other services, when waiting for them to come back online).
- Possible performance cost if state needs to be saved to enable recovery.
- Recovery speed is non-deterministic since it depends on how fast the processes can be restarted, loading of saved state, loading of dependencies, system load level, etc.

Known uses. Erlang actor model and supervisors (Erlang is used e.g. in Ericsson AXD301 ATM switches) [2]: supervisors are processes that are responsible for starting, stopping and monitoring their child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary [13].

Control system of Curiosity: Mars rovers are highly autonomous vehicles that operate in high-radiation environment, relying on a low-bandwidth, high-latency communication link. A warm reset can be executed by control system when it identifies a problem with one of its operations. On November 7, 2013 Curiosity rover performed a reset of its control software upon encountering an unexpected event (an error in a catalog file) [17]. After the reset, rover entered safe mode, but was able to perform operations and communications as expected and successfully resumed nominal operations mode after the fault had been analyzed.

Related Patterns. ERROR CONTAINMENT BARRIER, NOTIFICATIONS, SAFE STATE, SERVICE MANAGER, REDUNDANCY, WHAT TO SAVE.

MINIMIZE HUMAN INTERVENTION (MHI) is about how the system can process and resolve errors automatically before they become failures [6]. LET IT FAIL could be implemented as part of MHI as a final resort or in case there is no specification for error handling.

Software REJUVENATION is a proactive technique where the system has been designed to be booted periodically. Microrebooting [11] refers to a technique where suspect components are restarted before they fail.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. Transactions on Dependable and Secure Computing, 1(1).
2. Armstrong, J. (2003). Making Reliable Distributed Systems in the Presence of Software Errors. Stockholm, Sweden: Royal Institute of Technology.
3. Dunn, W. (2002). Practical Design of Safety-Critical Computer Systems. Reliability Press.
4. Knight, J., & Leveson, N. (1986). An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. Transactions on Software Engineering, 12, 96-109.
5. Herder, J. (2010). Building a Dependable Operating System: Fault Tolerance in MINIX 3. Netherlands: Vrije Universiteit. USENIX Association.
6. Hanmer, R. (2007). Patterns for Fault Tolerant Software. John Wiley & Sons.
7. Eloranta, V.-P., Koskinen, J., Leppänen, M., & Reijonen, V. (2010). A Pattern Language for Distributed Machine Control Systems. Tampere University of Technology, Department of Software Systems.
8. Buschmann, F., Henney, K., & Schmidt, D. (2007). Pattern Oriented Software Architecture: A Pattern Language for Distributed Computing. John Wiley & Sons.
9. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
10. EventHelix.com Inc. Manager Design Pattern. Retrieved January 2, 2013, from: http://www.eventhelix.com/realtimemantra/ManagerDesignPattern.htm#.UOQm6kUbR8E
11. Candea, G. & Fox, A. (2003). Crash-Only Software. Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems.
12. Eloranta, V.-P. (2012). Event Notification Patterns for Distributed Machine Control Systems. Proceedings of VikingPLoP 2012 Conference. Tampere University of Technology, Department of Software Systems.
13. Erlang/OTP R16A documentation. Retrieved February 13, 2013, from: http://www.erlang.org/doc/
14. Hanmer, R. (2010). Software Rejuvenation. Proceedings of 17th Conference on Pattern Languages of Programs. ACM.
15. Pinho, L. & Vasques, F. (1999). Replica management in real-time Ada 95 applications. Proceedings of the 9th international workshop on Real-time Ada. ACM.
16. Jain, P., & Schmidt, D. (1997). Dynamically Configuring Communication Services with the Service Configurator Pattern. C++ Report, June issue.
17. NASA Jet Propulsion Laboratory. Curiosity Out of Safe Mode. Retrieved December 19, 2013, from: http://www.jpl.nasa.gov/news/news.php?release=2013-330

Acknowledgements. Authors would like to thank reviewers and VikingPLoP 2013 participants for the feedback and providing valuable comments, Robert Hanmer for shepherding the paper and VikingPLoP organizers for a great pattern conference. This work was carried out under the EFDA Goal Oriented Training Programme (WP10-GOT-GOTRH) and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

Appendix: List of Referenced Patterns

Table 2. Short descriptions of referenced patterns.

Pattern	**Pattern intent**
BUS ABSTRACTION [7]	Nodes communicate via a message bus. The bus is abstracted so it can be changed easily.
ERROR CONTAINMENT BARRIER [6]	System should stop the flow of errors from one part to another by isolating them to a unit of mitigation and initiating error recovery.
FAULT OBSERVER [6]	Coordinate reporting to all observers that a fault is present, reported, and recovery actions escalated.
HEARTBEAT [6] [7]	Send a status report at regular intervals to let other parts of the system know their status.
LEAKY BUCKET COUNTER [6]	Implement a method to ride over transients by keeping a counter that is automatically decremented and incremented by errors.
MINIMIZE HUMAN INTERVENTION [6]	System should take care of itself without human intervention.
MONITOR [10]	Support many entities of same or similar type. The MANAGER object is designed to keep track of all the entities. In many cases, the MANAGER will also route messages to individual entities.
MEDIATOR [9]	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
NOTIFICATIONS [12]	Communicate noteworthy or alarming events and state changes in the system using a dedicated message type.
PUBLISH/SUBSCRIBE [8]	Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that convey information that may be of interest to others. Notify subscribers interested in those events whenever such information is published.
QUARANTINE [6]	Take steps to isolate faulty unit in order to stop fault propagation.

REDUNDANCY [6]	Maximize availability by having alternate hardware or software that can perform the same function.
REJUVENATION [11][14]	Periodically rejuvenate a software item by shutting it down and restarting it.
REPLICA MANAGER [15]	Hide communication algorithms from the applications to cope with the possible non-deterministic behaviour of replicas by delaying requests until all replicated nodes make the same request.
SAFE STATE [7]	If something potentially harmful occurs, all nodes should enter a predetermined safe state.
SERVICE CONFIGURATOR [16]	Decouple the behaviour of services from the point in time at which service implementations are configured into an application or system.
SOMEONE IN CHARGE [6]	Every fault tolerance action undertaken by the system should have a clearly identified entity controlling and monitoring the action.
START-UP MONITOR [7]	During start-up all devices are started in certain order and with correct delays. Additionally, care is taken that there are no malfunctions.
STATIC RESOURCE ALLOCATION [7]	Critical services are always available when all resources are allocated when the system starts.
SYSTEM MONITOR [6]	Some errors will only manifest themselves at a system level. Check for them at this level.
UNITS OF MITIGATION [6]	Decide what the unit of fault tolerance is.
WATCHDOG [6] [7]	Build a special entity to watch over another to make sure that it is still operating well.
WHAT TO SAVE [6]	Use checkpoints to save information of global interest to a shared, globally accessibly data storage.

Publication 6

P. Alho and J. Mattila, Service-oriented approach to fault tolerance in cyber-physical systems, Journal of Systems & Software, vol. 105 (July), pp. 1-17, 2015

DOI: 10.1016/j.jss.2015.03.041

© Elsevier. Reprinted with permission.

The Journal of Systems and Software 105 (2015) 1–17

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Service-oriented approach to fault tolerance in CPSs

Pekka Alho[*], Jouni Mattila

Department of Intelligent Hydraulics and Automation, Tampere University of Technology,P.O. Box 589, FI-33101 Tampere, Finland

ARTICLE INFO

Article history:
Received 14 July 2014
Revised 9 March 2015
Accepted 14 March 2015
Available online xxx

Keywords:
Dependability
Resilience
RTSOA

ABSTRACT

Cyber-physical systems (CPSs) are open and interconnected embedded systems that control or interact with physical processes. Failures in CPSs can lead to loss of production time, damage to the equipment and environment, or loss of life, meaning that dependability and resilience are key properties for their design. However, existing fault tolerance and safety approaches are inadequate for complex, networked and dynamic CPSs. Service-orientation, on the other hand, is generally considered to be a robust architectural style, but there is a limited amount of research on fault tolerance of service-oriented architecture (SOA), especially on distributed real-time systems. We propose an approach that utilizes the loosely coupled nature of services to implement fault tolerance using a middleware-based real-time SOA (RTSOA) for CPSs. The approach, based on the concepts of fault isolation and recovery at the service level, is empirically evaluated using a demanding bilateral teleoperation (remote handling) application. The empirical evaluation demonstrates that RTSOA supports real-time fault detection and recovery, use of services as a unit of fault isolation, and it provides capability to implement fault tolerance patterns flexibly and without significant overhead.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

Cyber-physical systems (CPSs)combine networked embedded systems controlling physical processes or devices with communication network infrastructure. Such systems have great economical, environmental and societal potential (Kim and Kumar, 2012; Lee, 2008)by aiding in the development of intelligent devices, services, customizable products and big science projects. For example, future trends include networked heterogeneous vehicles and machines, equipped with configurable tools and smart sensing capabilities, collaborating to carry out tasks while being remotely monitored by various supervisory systems. Interaction with environment means that these systems typically have requirements for deterministic response times and that failures can lead to loss of valuable production time, damage to the equipment and environment, or even loss of life. However, due to the complex and cross-disciplinary nature of CPSs, their development will likely involve several contractors and teams, and the resulting systems are composed of heterogeneous computing nodes. These nodes interact with each otherand the physical entities over a stochastic communication network, which may cause delays and packet loss (Kim and Kumar, 2012). Moreover, the computing environment is dominated by uncertainty as participating nodes and subsystems are changed based on the current needs and available resources.

Ability to verify and validate (V&V) system reliability and safety is one of the key research challenges for CPSs because of the characteristics described above. We have processes and tools that support achieving these qualities, but they are proving to be insufficient to cope with the increasingly complex systems Jackson (2009)—a single undetected human error during the development process can create a fault with potential to cause unexpected consequences in the physical world. Software faults are facts to be coped with and should be contained at the application level (Patterson et al., 2002) and the software architecture design can support localization of critical properties to components through decoupling and simplicity (Jackson, 2009). For example, modular architecture can limit fault propagation and support verification of critical components. Fault tolerance techniques based on utilizing modularity of the architecture, such as fine-grained partitioning and microreboots (Candea et al., 2004; Patterson et al., 2002), have been researched as complementary approaches to redundancy-based fault tolerance, but have not been applied in distributed real-time systems.

Service-orientationis an architectural approach originally developed to tackle the problem of complexity in enterprise systems, but it also has potential to provide a basis for building resilient CPSs. Possible benefits of SOA for real-time systems have been recognized, and work has been done to apply SOA approach to distributed real-time and embedded (DRE) systems, including Cândido et al. (2009), Cucinotta et al. (2009), Panahi et al. (2009) and Tiderko et al. (2008). Although service-orientation is associated with the robustness property (Erl, 2008), there has been little research on how this could be utilized to improve fault tolerance capabilities. Some potential

[*] Corresponding author. Tel.: +358505375726.
E-mail addresses: pekka.alho@tut.fi (P. Alho), jouni.mattila@tut.fi (J. Mattila).

http://dx.doi.org/10.1016/j.jss.2015.03.041
0164-1212/© 2015 Elsevier Inc. All rights reserved.

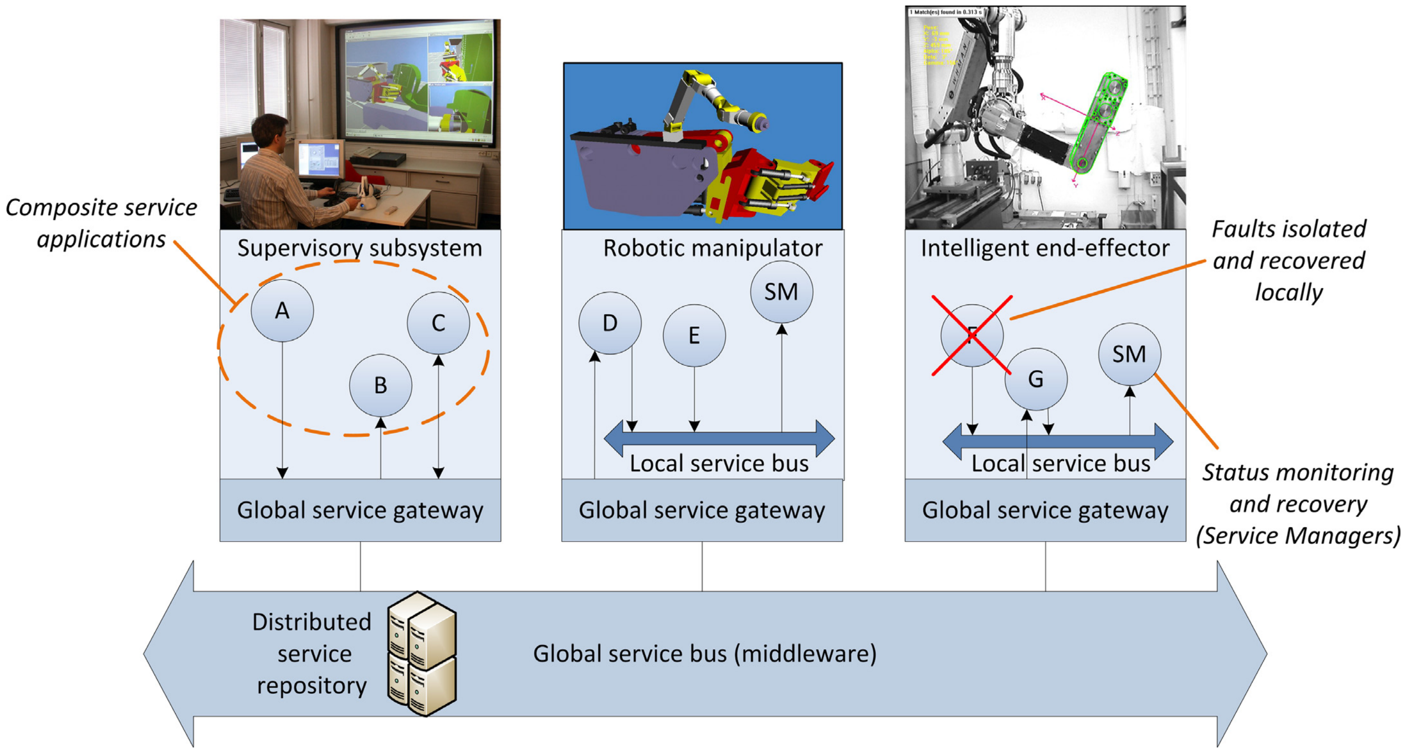

Composite service applications
Supervisory subsystem
A
B
C
Global service gateway
Robotic manipulator
D
E
SM
Local service bus
Global service gateway
Intelligent end-effector
F
G
SM
Local service bus
Global service gateway
Faults isolated and recovered locally
Status monitoring and recovery (Service Managers)
Distributed service repository
Global service bus (middleware)

Fig. 1. An example CPS scenario, showing services deployed on several nodes and connected by local and global service buses.

approaches include recovery of Web services (Mansour and Dillon, 2011) and reconfiguration of service compositions for distributed real-time systems (Garcia Valls et al., 2013).

Time-bounded reconfiguration has been successfully used in iLAND middleware (Garcia Valls et al., 2013) and research by Moussa et al. (2010). However, CPSs need resilience to maintain correct operation in the context of improperly coordinated control of cyber and physical resources (Rajkumar et al., 2010), for which we provide contribution by evaluating how the system detects and recovers service faults while being actively used in a mission-critical operation. Our research applies to the fault recovery process, for which timing guarantees cannot be given. Due to an activated fault, timing properties of the affected parts of the system are now unknown because it operates in an erroneous state, outside of specification. Other service composition changes are handled in the same manner as service failures, in order to avoid introducing coupling between services. Further, by using the same execution paths for start-up/shutdown and fault recovery, we can confirm that the fault handling mechanisms work correctly during normal system operation. For normal operation, real-time targets are met through the use of a real-time operating system (RTOS) and real-time capable middleware.

This paper continues our earlier work on the subject of RTSOAs, evaluating applicability of service-orientation as basis for building dependable and resilient CPSs. Service-orientation by its nature is suited for systems that face uncertainty and changes, enabling more agile approaches (Cândido et al., 2009), and having potential to support fault tolerance capabilities. Our approach uses and extends a prototype implementation of Sulava platform RTSOA, developed for distributed machine control and evaluated earlier with applications including control of mobile working machines (Hahto et al., 2011) and robotic maintenance systems (Alho and Mattila, 2013; 2014). Compared to our previous work in Alho and Mattila (2014), we have improved the architecture fault tolerance capabilities, illustrated the service-based approach in more detail and performed extensive measurements on service fault recovery.

In this paper we explore the possibility of using fault tolerance mechanisms that utilize key features of the Sulava RTSOA, namely loose coupling and service autonomy, to improve fault tolerance and robustness of systems with Quality of Service (QoS) needs. The architecture supports connecting intelligent devices, machine control units and supervisory system elements and integrating controlled elements to supervisory and plant systems, as shown in Fig. 1. In order to evaluate effectiveness of the service-oriented approach to fault tolerance, we perform empirical evaluation of the architecture, testing fault isolation and recoverability of services, while functioning as part of a distributed system with QoS needs. The test case is based on bilateral teleoperation of a remote maintenance robot, known as remote handling (RH). RH is a demanding and safety-critical application, where the remotely operated robot must be able to perform precisely in confined and pitch-black maintenance tunnels that have no human access. The system is controlled in a bilateral control loop that provides haptic feedback for the operator. Instead of aiming to carry out recovery inside bound time limits, recovery is performed with best effort, while rest of the system keeps operating normally, using QoS and deadline parameters to detect possible failures. For the evaluation of fault tolerance capability, our criteria include fault detection and recovery, capability to change services on the fly and estimation of overhead caused by management of services. Other possible benefits of using the service-oriented approach, such as service deployment, configuration flexibility or maintainability, are not evaluated. Interoperability and real-time performance have been evaluated earlier in Alho and Mattila (2013, 2014).

The rest of this paper is organized as follows. Section 2 presents the technical background, including the concepts of fault tolerance, service-orientation, and CPSs. Section 3 describes the service-based fault-tolerant architecture concept and its design goals. Empirical evaluation of the service-oriented fault tolerance approach in CPSs is presented in Section 4. The results of the evaluation and discussion about significance of the results of the work is in Section 5. Comparison with related work and limitations are presented in

Sections 6 and 7 respectively. Finally, the conclusions are drawn in Section 8.

2. Background

2.1. Fault tolerance, dependability and resilience

Dependability is defined as the ability of the system to deliver service that can be justifiably trusted, encompassing the attributes of availability, reliability, safety, integrity and maintainability (Avizienis et al., 2004). CPSs need dependability especially because they interact with the physical world, which means potential for damage. Although achieving dependability requires a systems engineering approach, utilizing fault prevention, fault removal, fault tolerance and fault forecasting (Avizienis et al., 2004), on an architectural level it is implemented mainly through fault tolerance techniques and patterns that provide error detection and recovery. A key factor of fault tolerance is modularity, which can be used to implement fault containment (Gray, 1986). This is especially interesting for service-based systems because they are highly modular by their nature, possibly providing a basis for cost-efficient fault tolerance implementation.

However, dependability does not completely describe the needs of complex, networked and evolving CPSs. Such systems need to be able to deliver the service when *facing changes* (Laprie, 2008). Capability to do this is defined as resilience. Laprie lists evolvability, assessability, usability and diversity as technologies for achieving resilience (Laprie, 2008).

Traditionally in machine control systems, communication between units is facilitated hierarchically by central control (Hahto et al., 2011). This central control is a single point of failure. Autonomous, intelligent units, on the other hand, would be capable of operating even without the guidance of high-level systems. Also, the availability of the machines has a great impact on utilization rate of expensive machines and equipment, affecting return on investment. Besides minimizing Mean Time Between Failures (MTBF), higher availability can also be achieved by improving Mean Time To Repair (MTTR), which can be supported through fine-grained recovery (Patterson et al., 2002).

2.2. Service-orientation

Service-oriented computing focuses on building architecture from autonomous and heterogeneous components (services) (Huhns and Singh, 2005), enabling reorganization of previously siloed or monolithic software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols (Papazoglou, 2003). Adoption of SOA could therefore provide a solution for the problem of increasing scale and complexity in distributed control systems. In the context of CPSs, services can abstract devices and functions of the machines, providing, for example, motor control capabilities or access to Controller Area Network (CAN) bus.

RTSOA implementations typically either extend the standardized Web service approach (Moussa et al., 2010) or use middleware, such as Data Distribution Service for Real-Time Systems (DDS) or Java Messaging Service (JMS) (Garces-Erice, 2009) as an enterprise serial bus (ESB). However, Web service technologies face challenges when applied to CPSs, especially because of timeliness requirements and verbosity of XML-based protocols (Alho and Mattila, 2013). Middleware-based approach provides a compromise between performance of proprietary messaging and well-defined but verbose Web service standards. Therefore, middleware is a key technology for enabling collaboration between control system nodes, abstracting the low-level communication infrastructure for developers and simplifying service development. Although middleware cannot automatically guarantee deterministic latencies for use in real-time control systems (Tuominen et al., 2014), it can offer efficient distribution of data with minimal overhead and the ability to control QoS properties that affect predictability and used resources.

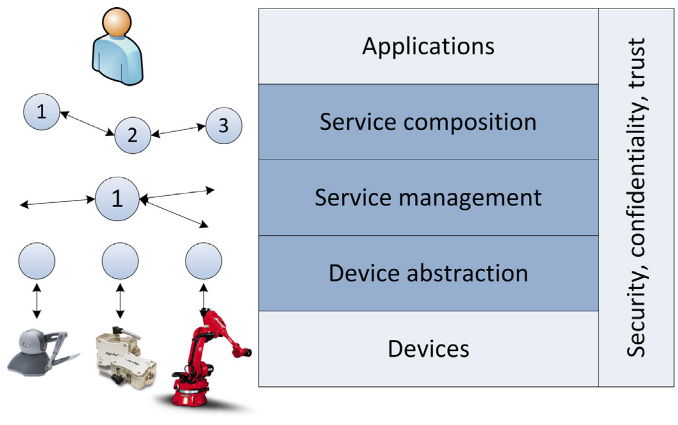


Fig. 2. SOA for CPSs, adapted from Atzori et al. (2010).

Common interfaces provided by middleware are essential technology for the service infrastructure. Layers of service infrastructure (Fig. 2), provided by middleware, lifecycle services and other architecture components, include composition, service management and object abstraction layers (Atzori et al., 2010). Service composition provides capabilities to build applications out of single services and includes inventory of currently connected services. Service management provides the basic set of functions to manage devices and services, such as dynamic discovery, status monitoring and service configuration. This layer also includes management of QoS and fault tolerance parameters and functions. Device (object in Atzori et al., 2010) abstraction provides wrapping layer to the device, consisting of the messaging interface and implementation logic sub-layers.

Challenges of using service-orientation include that service composition reliability may suffer if one service experiences a failure (Pan, 2009), which may degrade reliability of the whole system, and isolation of faults. Fault isolation is one aspect of the service-based fault tolerance that we study in this paper. Another significant challenge is guaranteeing real-time performance of services for the dynamic and changing environments that are typical for CPSs, because all possible combinations of environmental and computational factors cannot be foreseen during development. Service specification should include description of the intended use environment, but overly strict specification limits reusability of the services.

The following well-known principles for service design, presented in Erl (2008), are also used as a basis for designing real-time services:

- Standardized service contract—Consists of general information about the service, capabilities, QoS, etc. For middleware-based SOA, this is mapped to an information model.
- Reusable—Services need to be generic and configurable to be used (composed) in different applications.
- Autonomous—Services need to have control over their execution environment and resources. Removing unpredictable outside influences will improve reliability.
- Stateless—Services should minimize the time they spend being stateful, i.e. storing or processing task-specific data.
- Discoverable—Meta-information about service purpose and capabilities, published over middleware. Service contract must be registered in the service repository in order to be discoverable (Dai et al., 2014).
- Loosely coupled—No direct references to other services.
- Abstracted—Hide non-essential information about service.
- Composable—Service is a potential composition member, and application is a specific service composition. Real-time service

composition of services is an open challenge (García Valls and Basanta Val, 2013).

2.3. Cyber-physical systems

CPSs combine computing provided by embedded systems and IT infrastructure to monitor and control the dynamics of physical and engineered systems. They share many concepts with DRE systems, such as heterogeneity and transient behavior, but the CPSs inherently also include the element of interaction with physical processes and environment. Essentially, DRE systems provide the flexible and open infrastructure for the implementation of a CPS.

Innovation in CPSs is powered by trends of increased computing power and connectivity of embedded systems, and availability of highly capable low-cost sensors, often based on microelectromechanical systems (MEMS) technology. CPSs hold promise to transform how we interact with and control the physical world around us, by providing us with building blocks to construct self-correcting systems, infrastructure that calls automatically for preventive maintenance, energy-efficiency-aware buildings, vehicles capable of assisting drivers, etc. (Rajkumar et al., 2010).

The technical challenge of CPSs is that understanding the computational and control elements separately is not sufficient, as the dynamics, time and concurrency must be managed in the combined system. More specifically, the challenges include interfacing precision of the computing with the uncertainty and noise in the physical environment, lack of synchrony across time and space, dealing with failures of components in both cyber and physical domains, security and privacy issues, managing system dynamics across multiple time-scales, etc. (Rajkumar et al., 2010).

Typically both computational and physical parts of the system are highly likely to change during the system life-cycle. Compared to the traditional embedded systems approach, nodes and components in a CPS must be more resilient in order to properly function as part of the dynamic and complex computing environment. However, the increased complexity due to the dynamic nature of the system also means that V&V becomes more challenging. This is in conflict with the typical applications of the CPSs, which include interaction with the physical environment (sometimes including humans). Therefore, traditional certification processes for safety-critical systems quickly become too burdensome and expensive. One possible approach to building critical CPSs is to develop a dependability case (Jackson, 2009), which includes evidence such as models, architectural solutions, testing data, etc., to build confidence in the correct functionality of the system. To support development of dependability case, services would ideally be units that can be verified and validated individually. If criticality-related properties of the system can be localized to a single service, resources can be focused on that service alone (Jackson, 2009).

Applications for CPSs range from assistive devices to electric power grid infrastructure and large scientific experimental systems such as the Large Hadron Collider and ITER machine (the international nuclear fusion research and engineering project). In this paper, we will use the ITER remote handling system as a target to provide a real-world demanding test case scenario and requirements for the empirical evaluation. The choice of the target application is due to the fact that the fault-tolerant prototype system used in the empirical evaluation has been developed as part of the research efforts of ITER RH systems (see Acknowledgments). ITER RH systems, described briefly in Section 4, consist of several subsystems that are used for remote maintenance of the machine. These systems need to be highly modular and integrate into the plant IT infrastructure. In a typical RH scenario, the operator uses a supervisory system to operate maintenance robots in an inaccessible environment using either automatic operations or man-in-the-loop teleoperation, as shown in Fig. 1.

Requirements and characteristics for CPSs may include highly deterministic (local) communication deadlines and distributed communication being typically either reliable, e.g. commands and status updates, or best effort with varying deadlines. For example, sampling rates for bilateral (force feedback) teleoperation control loops are in the range of 500–1000 Hz, which sets a requirement for high data packet rate for the transmission of the sampled command and sensor data between the human interface and the telerobot.

The RH scenario has a “99% real-time” requirement for sending force feedback-related teleoperation signals; high packet rate is hard to maintain constantly, but latency and overdue messages increase the risk of instability and decrease transparency of the system(the “feel” of the remote site;Lawrence, 1993). ITER RH systems also have a long expected deployment life-cycle, made more challenging by the highly dynamic technical environment as systems are likely to receive upgrades to handle new experiments. Regardless of the upgrades, systems must behave correctly on newer and more powerful hardware, which has typically been a problem for certified real-time systems, such as fly-by-wire aircraft control systems, making it necessary for manufacturers to stockpile hardware components.

3. Service-based fault-tolerant architecture

In this section we describe the RTSOA, Sulava platform, designed to meet the requirements for CPS-applications and used to implement the remote handling system for the empirical evaluation of fault tolerance described in Section 4.

3.1. Requirements for CPS SOA

Dependability and resilience are critical for CPSs, as noted in the previous section. Service-based design seems to be a promising approach to achieve these goals, but the architecture needs to support communication deadlines and QoS parameters, which limits the possibility of using existing SOA technologies. To build an RTSOA platform that fulfills CPS-specific needs, we focused on the following requirements during the architecture development:

- *Decoupled and autonomous services* act as units of mitigation, stopping propagation of faults, and are used as units for fault detection and recovery (Hanmer, 2007).
- *Diversity*—Alternative compositions and services can help to avoid single points of failure and also provide workaround solutions. Diverse operation modes and alternative versions of services increase resilience of the system.
- *Evolvability* and *assessability* are key properties for a CPS, in order to successfully create a resilient system (Laprie, 2008).
- Fault tolerance for *transient faults*, which are a significant cause of failures.
- *Let it crash* and *fine-grained recovery* approaches to fault recovery (Alho and Rauhamäki, 2014), based on partial restarts (microreboots) to recover the system (Candea and Fox, 2003; Patterson et al., 2002), where the services will exit upon failures or return an error—the error will be detected on the next heartbeat check.
- *Service composition* is one of research topics for service-based industrial devices, enabling services to work together to create added value (Cândido et al., 2009). However, in order to focus on fault tolerance capabilities of the architecture, service composition management will be implemented in a lean manner through a service manager providing lifecycle management capabilities.
- *Decentralized design* to avoid single points of failure. Autonomous and intelligent services can be used to support choreography-based service composition (Cândido et al., 2009).
- Off-the-shelf solutions for communications and platform technologies to enable rapid development of systems. Platforms should be standards based to support interoperability between different vendors of equipment and software (Cândido et al., 2009) and provide QoS and real-time capabilities. Proprietary (in-house) solutions are seen to lead to fragmented ecosystems,

limiting possibilities for collaborative multi-machine control and software reuse.

An evolvable system can facilitate changes, including service upgrades, addition of new functionalities, etc. Assessability means that verification and evaluation of the system can be carried out on a running (deployed) system to provide justified confidence for the resilience of the system (Laprie, 2008). Also, if services can be verified individually, i.e. they are capable of acting as units of fault isolation and mitigation, fault forecasting and removal techniques can be used more efficiently, further supporting assessability.

Interaction and timing faults are often transient faults, which are common in distributed systems. For example, Grottke et al. (2010) classified failures found in software anomality reports of 18 robotic exploration space missions and found that 36.5% of failures were caused by non-deterministic and aging-related faults (so-called Mandelbugs). Mandelbugs are difficult to isolate and reproduce, making them difficult to remove in the testing phase. On the other hand, it is possible that operation will work on re-execution, making techniques such as retrying and restarting the application effective (Alonso et al., 2013).

This idea is applied also in the let-it-crash approach, which is similar to reset and supervisor-integrated circuits, used in embedded system design to force the microprocessor back into a safe state and then restart the processor if its parameters are not within the nominal operating range. Typically, bugs will cause the software to crash, deadlock, leak memory, etc., for which restarts are an effective, although temporary, form of recovery. Because the recovery mechanisms are kept simple, the possibility of having faults in the recovery code is reduced, and the recovery paths get executed and tested often, as they are based on the standard startup paths. According to our knowledge, the microreboot approach is not commonly used in realtime systems, but the decoupled design of services seems like a good match for it. This approach also emphasizes MTTR instead of MTTF—high MTTF cannot guarantee fault-free operation, but low MTTR can help to resume operation after faults (Patterson et al., 2002). For a CPS there are safety considerations regarding the use of fault recovery; these are covered in Section 3.4.

3.2. Sulava RTSOA platform

The Sulava platform is based on the use of data-centric middleware and an RTOS to implement communications and provide real-time scheduling for loosely coupled, collaborating applications, i.e. services. The platform is intendedly very lean, providing capabilities to rapidly test and develop machine control systems.

Key concepts of the architecture are global service bus (GSB), local real-time service bus (LSB) and a local service manager. GSB is an abstraction for middleware-based communications and LSB for local realtime queues. The communication buses (middleware) and service managers effectively implement the layers for composition and service management from Fig. 2. Service status information is published to a GSB topic, where it is available for other nodes (usable by remote service managers) and provides a distributed repository of deployed services.

The architecture includes a number of fault tolerance patterns, such as heartbeat, directly implemented as part of the services, and supports implementation of additional detection and recovery patterns. Although the architecture cannot *guarantee* fault isolation, services are intended to contain faults and act as units for fault detection and handling; testing the services as units of fault tolerance is one of the research objectives of this paper.

Additional functionality, such as employment of semantic techniques, business process management or service configuration, can be implemented by introducing additional software tools, similar to the service manager, which is basically an RTOS application responsible for managing the services. These tools can utilize existing service information available from GSB, published by services or service managers.

3.2.1. Communication buses

Inter-service communication is divided into two buses to better facilitate the wide range of requirements for deadlines (Fig. 3). The buses, GSB and LSB, are abstractions for the underlying communication technologies that extend the more commonly used concept of ESB. Abstraction functions to further shield developers from the infrastructure details and to make sure that the services conform to the same information model and QoS parameters. The buses can also be used to connect services to heterogeneous subsystems, plant/worksite systems and external applications.

GSB is used as a wrapper for the DDS middleware standard maintained by the Object Management Group (OMG), providing distributed connectivity with QoS capabilities for the services. DDS has non-deterministic communication delays over networks, due to using standard networking protocols, communication deadlines and QoS

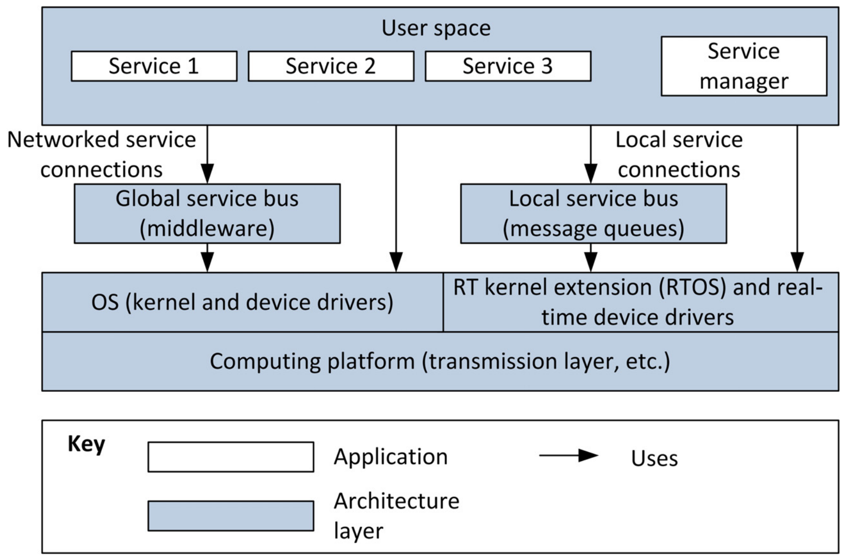

User space
Service 1
Service 2
Service 3
Service manager
Networked service connections
Local service connections
Global service bus (middleware)
Local service bus (message queues)
OS (kernel and device drivers)
RT kernel extension (RTOS) and real-time device drivers
Computing platform (transmission layer, etc.)
Key
Application
Uses
Architecture layer

Fig. 3. Communication buses in the logical system architecture.

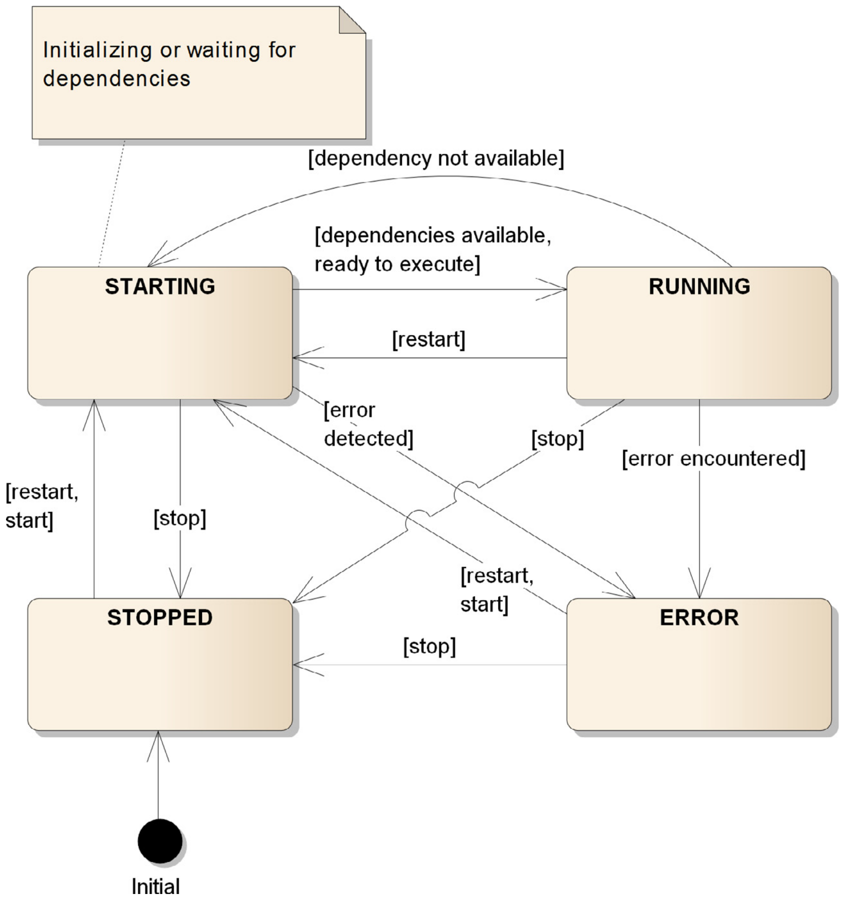


Fig. 4. Service states and commands.

parameters can be monitored (Tuominen et al., 2014), making DDS suitable for use in a "99% real-time" application in which a single deadline overrun is not catastrophic. Real-world performance of DDS naturally depends on latency requirements, network structure and network loads. Services using GSB can be integrated to enterprise SOA through DDS-based adapters or service mediators. From the enterprise systems point of view, the RTSOA platform provides a service subset that has been designed for a specific environment (CPSs) and is available from its own domain inventory.

LSB is a wrapper for real-time queues, which are local point-to-point message channels that can have multiple readers and writers. Real-time queues are used for communication with real-time applications locally, providing sufficient performance and determinism to qualify as real-time (Brown and Martin, 2010). The queues are a significantly simpler communication method when compared to middleware-based GSB, but provide sufficient capabilities to connect services locally in a loosely coupled manner. To support evolvability, implementation technology of LSB can be changed, depending on the execution environment.

3.2.2. *Design of services*

Service is the unit that is used for abstraction of devices and concepts, providing the capability to connect e.g. USB, CAN, FireWire or RTnet based devices as services to the architecture. Service in the Sulava platform is an independent program (process) with common code components and a standardized structure, including connection and identifying properties (Hahto et al., 2011). This differs from Web services, which typically are deployed on an application server.

In order to facilitate management, services are expected to publish status data to be used by the service managers and other possible tools, and listen for management commands. The status data are published to the LSB and includes the state of the service, which can be stopped, (re)starting, running, and error (Fig. 4). Initially, when a service has been started and initialization is completed, it will be in a stopped state. After receiving a start command from the service manager, service moves to the starting state where it checks availability of its dependencies. When dependencies are available, service moves to the running state. Restart command from service manager is an indication for the service to reset its operational parameters and restart computation. Service will move to the stopped state after receiving stop command or after completing its task, if the service is not expected to be running constantly. If the service encounters an error that it cannot be reasonably expected to handle, it will change its status to error, leaving the decision about further actions to fault tolerance mechanisms. For example, the service manager can terminate and restart the service based on the let-it-crash fault recovery approach.

Service startup can potentially cause problems if services have circular dependencies, which is typical in control systems (e.g. for feedback loop participants). Therefore, all services are started in sequential order: services initialize after the start command, as described in Fig. 4. Service waits for dependencies in the starting state and starts executing normally in the running state once dependencies are available.

The internal architecture of a service does not matter from the perspective of the RTSOA platform, as long as it conforms to basic service interface (states and command interfaces). Service is responsible for implementation of commands and should be verified to conform to the state chart in Fig. 4 in order to implement service management and fault tolerance capabilities, although V&V processes are outside the scope of this paper. In addition to real-time services, non-real-time services can also participate in compositions if the connection is done in accordance with the service contracts.

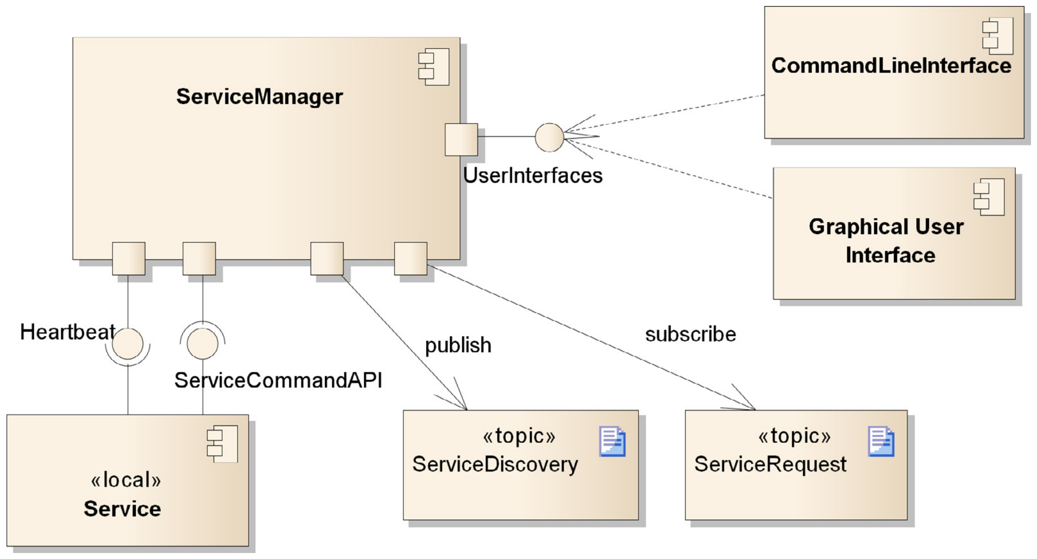


Fig. 5. Service manager component diagram.

For example, with Web services service contract is usually composed of WSDL document and XML schemas. For middleware-based RTSOAs, there are no specific standards for defining the service contract, although the data model is comparable to a contract. In the Sulava platform, the service contract is formed by the data model expressed with OMG specification compliant Interface Definition Language, extended with QoS and deadline parameters.

Typical structure of a real-time service consists of a main loop, which is executed periodically, as opposed to one-shot services. The Sulava platform provides a service template in C++ for construction of services in stepped-loop format, which includes setting up of the service management LSB queues and boilerplate methods for initializing, stepping, starting, resetting, and stopping the service execution. The current Sulava RTSOA implementation is based on Xenomai-Linux and uses C/C++ based real-time libraries, but can be extended to other platforms through the GSB; interoperability is demonstrated in the empirical evaluation.

Statelessness is one of the key design principles for service design, aiming to maximize scalability, and also related to fault recovery as state data lost during service restart might need to be re-retrieved. The Sulava services have internal state (status of the service), with the main states being running and stopped (passive vs. active). In addition to this primary state, an active service may be stateful or stateless, depending on whether it currently has task-related data, e.g. context or business data (Erl, 2008). When a service is active (running), it should aim to minimize the time spent processing state data specific to a task. To achieve this goal, the service needs to defer its state data temporarily, typically to a database. In the Sulava architecture, GSB and LSB can also be used by the service to send the state data to itself, enabling increased statelessness and fault recovery. However, task-centric services deployed as part of an active control system are usually constantly performing processing and remain stateful.

Service autonomy is another critical principle for fault tolerance—a high level of autonomy provides flexibility to use services in compositions and isolate failures to other services. Service autonomy for Sulava is achieved by implementing them as independent processes, but services may still affect each other indirectly. For example, starting a high-priority service can temporarily use all available processor resources due to start-up activities and cause other services to miss deadlines, possibly causing failures. To facilitate varying workloads and resource availability, CPSs would benefit from development of adaptive resource management strategies.

3.2.3. *Service manager—Servicecompositions and fault tolerance*

Service manager is a local lifecycle management component that can be used to start, stop and monitor services, i.e. it provides functionality of the service management layer from Fig. 2. The service manager, component diagram in Fig. 5, has been designed to be easily extendable to include new monitoring methods, optional user interfaces and fault detection. Sulava platform has a prototype service manager GUI, shown in Fig. 6, enabling the user to start and stop an inventory of services. Service composition can be changed on the fly, including updating of services and switching to backup services, e.g. hot standby or a previous version that has been proven in use.

The service manager publishes the status of local services to GSB, available to other service managers. Other service managers can publish requests to start specific services on the basis of needed dependencies of their local services. Although centralized solutions can be detrimental to system dependability, the service manager is a compromise, a “locally centralized” component that is a single point of failure but only locally.

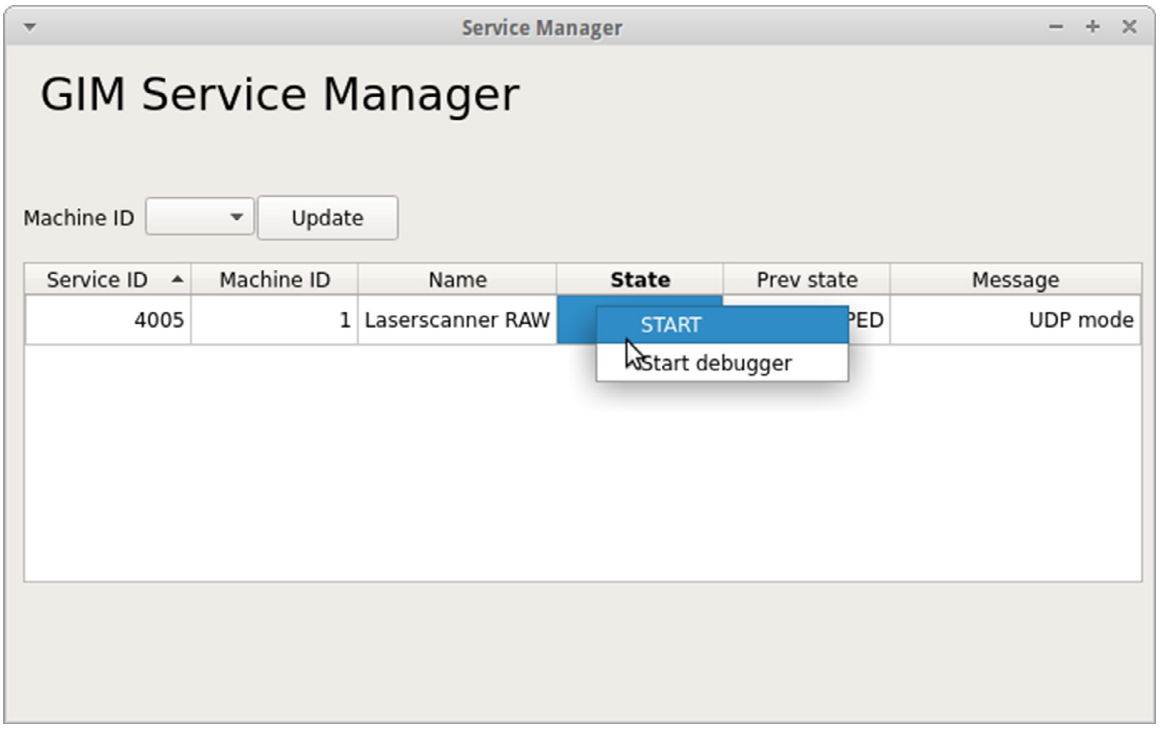
The service manager spawns the services, therefore becoming the parent process for them, enabling it to monitor and affect the life cycle of the services. Although the main methods of interaction with services are listening status updates and sending commands, termination of a service can be used as an escalated action if the service becomes unresponsive.

Service manager detects service faults with:

1. Heartbeat (Hanmer, 2007)—Services send a heartbeat message for the local service manager. Heartbeat deadline can be configured for each service.
2. Service status updates—Services report their state (Fig. 4) with the heartbeat message.
3. Resource monitoring—Services can have a per-service limits for resources, such as CPU, memory, network, etc. In the prototype implementation memory and CPU usage limits are supported.
4. Service exit signals are sent to the service manager, as it is the parent process for services.

Fault handling is based on fault escalation (Hanmer, 2007):

- Escalation level I: soft restart, send restart command.
- Escalation level II: hard restart. Kill service and spawn new.
- Escalation level III: switch to backup. Kill service and spawn backup service (if defined).
- Escalation level IV: terminate service, remove from execution.



Service ID	Machine ID	Name	State	Prev state	Message
4005	1	Laserscanner RAW		PED	UDP mode

Fig. 6. Prototype GUI for distributed service manager.

Algorithm 1 shows key tasks for management of services performed by the service manager. Services are started and stopped from a service inventory, based on requests from users or remote service managers. Fault handling is escalated, if the number of faults detected exceeds the specified maximum number of faults for that service.

Algorithm 1. Logic for service management main loop.

```
start all services
loop
    for all services do
        if start requested then
            start service
        end if
        if stop requested then
            stop service
        end if
        monitor resources
        monitor heartbeat
        monitor exit signal
        if service fault detected then
            if $numberOfFaults \geq maxFaults$ then
                $escalationLevel{+}{+}$
                $numberOfFaults \leftarrow 0$
            end if
            take action for $escalationLevel$ n
            $numberOfFaults{+}{+}$
        end if
    end for
    update service discovery information
    sleep for $taskPeriod$
end loop
```

In order to focus the evaluation efforts on services as units of fault containment and recovery, the architecture lifecycle services, responsiblefor managing the lifecycle of SOA solutions, aim for simplicity in the design. Therefore, both the composition and management layers (Fig. 2) are implemented through the service managers. For composition layer and discoverability, service manager publishes service names, status, and other information relevant to the use of services, functioning as a repository of running (in memory) services. Service startup (and stop) requests can be published to GSB topic, which is monitored by the service manager, enabling initiation of new service compositions (Fig.5). Service manager will launch the requested service if it is available locally. Service management layer is implemented through LSB queues, which includes status monitoring and configuration.

3.3. Fault tolerance and service recovery

For designing the error management of services used in a CPS, we assume that the services are well-tested and used in the specified environment, so failure should be unlikely if and that the cost of failure is high. Therefore, the desired error handling mechanism is the opposite of fail silent, which could cause the errors to go unnoticed and produce unwanted behavior. Terminating the service upon detecting an error (i.e. let-it-crash) is a form of fail-fast response, designed to minimize these unwanted behaviors.

Fault detection and recovery are primarily provided by single-version techniques used by the service manager. Fault tolerance is based on isolating the fault to the faulty service, locating the service and either recovering the fault or at least removing the faulty service from the system. After the recovery attempt, the system can resume operation, change to alternative compositions and applications, or wait for the human operators to decide the correct course of actions, which can include using the system “as is” and avoiding conditions that trigger the fault. Alternative service compositions can include alternative operational modes (e.g. automatic vs. manual) or degraded operation. A set of basic services can be designed to provide a degraded functionality mode, enabling, for example, recovery of equipment. However, changing of service composition dynamically is potentially a critical functionality; in this paper we have focused on fault tolerance at the service level instead of compositions and implemented service composition management by loading statically configured service compositions.

Let-it-crash fault handling and service management are based on well-known patterns, such as fail-fast and escalation. However, their feasibility and effectiveness have not been tested in RTSOA or CPS applications. The goal of using let-it-crash approach is to flush the corrupted state by terminating the process and improve system MTTR, as only the faulty service needs to be restarted, not the whole system. A secondary goal is to avoid writing overly complex error handling code.

An example of handling of service recovery process by the service manager is shown in the sequence diagram (Fig. 7). In the diagram, Service1 encounters an error that causes it to terminate unexpectedly. After the service manager notices a missing heartbeat, it tries to send a restart message, but LSB queue to the service is not available. This causes fault handling to escalate to the next level, where the service manager will try to perform a hard restart by sending a termination signal and spawning a new service. As the service has already exited, the service manager proceeds directly to spawning a new service. After the new service has performed initialization and sent the first heartbeat message, the service manager sends the start command, causing the service to transition directly to the running state (all dependencies available).

As mentioned earlier, recovery of faults without cascading service failures is necessary for achieving sufficient reliability. If a service fails, this can potentially cause the fault to propagate and other services to fail, depending on the failure mechanisms and design of the services. A faulty service can, for example, transmit messages at arbitrary points in time (babbling idiot failure). Another typical scenario is having a failure where service functions otherwise normally, but output is corrupted (but inside expected values). If services have been designed to be autonomous, it is easier to recover the situation without cascading errors. In the case that dependencies are not available (due to failures or otherwise), service transitions to starting state, which is considered to be a non-failed state, even though the system cannot provide full service.

3.4. Service criticality

CPSs interact with environment and physical processes. As a direct result, systems are often mission- or safety-critical. For safety-critical systems, separate safety-related systems are developed and certified according to safety standards (e.g. IEC 61508;International Electrotechnical Commission, 2010) that can monitor the system and retain its safety (Rauhamäki et al., 2012). As certification is expensive, only separate safety-related systems are typically certified, not the whole control system. Especially for CPSs, this would be either prohibitively expensive or even completely impossible, as all necessary information about the intended use and environment may not be available a priori. However, as the CPS interacts with environment and physical processes, the software still needs to be highly reliable and safe; possible safety-related functions are only a last resort.

Decoupled architecture can be designed to have no single points of failure as systems can run multiple copies of services and subsystems on different nodes. However, automatic switching to a “hot standby” for a safety-critical component is not necessarily desirable (e.g. for services designated critical as defined in Alho and Mattila, 2014). Alternative compositions can also be utilized to provide diverse redundancy, thereby improving resilience. However, having multiple compositions capable of sending commands deployed simultaneously can be dangerous if not taken into account in the system design and implementation. For example, ownership QoS parameter can be used to limit the number of writers to a command topic.

By analyzing which system components are critical, V&V efforts can be concentrated on the parts of the system where they have the most impact (Jackson, 2009). Autonomous and decoupled services are easier to verify than a monolithic system, as they act as independent units of fault isolation and deployment. A service is critical if its failure can cause damage or other significant losses. Fault propagation can also be a factor in the criticality analysis of a service, if input sanity checking, assertions and external systems (e.g. collision detection) cannot sufficiently guarantee safety. Moreover, fault handling of critical services needs to be analyzed on a case-by-case basis. If a service is critical only indirectly (e.g. IDCom service in the evaluation, responsible for transmitting position and velocity commands for the manipulator), fault recovery can be performed safely. The critical actions are performed in another service (C4G service in this case) which detects deadline overrun for input, moving to a safe state. Otherwise services are stopped and wait for manual operator intervention.

Validation of service deployments can be further supported with VR models and simulation, enabling assessment of the system with an operational configuration. Models can be used either with the system (as in RH) or in the place of the actual plant (hardware-in-the-loop simulation).

4. Empirical evaluation of the approach

This sectiondescribes the test scenario and experimental setup we have used to evaluate the fault detection and recovery capabilities of the RTSOA described in the previous section. We focus on the detection and recovery of the faults in a control system application, using services as the unit of fault isolation. Our null hypothesis is that the let-it-crash approach cannot support real-time fault detection and recovery, therefore we aim to disprove it through the empirical evaluation. The objective of the empirical evaluation is to validate the feasibility of implementing fault tolerance on a real-time service-oriented platform that interacts with the physical world. This includes testing fault detection and recovery using service manager; services as a unit of fault isolation and recovery (let-it-crash approach); capability to change services on the fly (including switching to a back-up service); and estimating overhead of the service management.

4.1. Test scenario

To evaluate effectiveness and performance of the service-oriented approach to fault tolerance, we use a distributed real-time control system implemented with the previously described software architecture in an ITER-relevant remote handling test scenario. Remote handling systems are used in the ITER machine to inspect, make changes to, and maintain components using automatic or man-in-the-loop teleoperation of robots. The test scenario consists of teleoperating an industrial robotic manipulator with a 6 DOF haptic input device (3 DOF force feedback). The manipulator used in the tests can be used e.g. to vacuum radioactive particles from other maintenance robots that have been working inside the ITER reactor.

The remote handling control system (RHCS) has a 99% real-time requirement for communications (Tuominen et al., 2014); the system tolerates occasional deadline misses, but a failure has potential to cause significant damage to equipment, lost experiment time, etc. The system consists of Virtual Reality (VR) software IHA3D, Input Device Controller (IDC), Equipment Controller (EC), and an Operations Management System (OMS). Nodes participating in the scenario are shown in Fig. 8; service managers and LSBs have been left out to clarify the figure.

We have used a service composition based on using an OMS earlier in Alho and Mattila (2013, 2014)for automatic control of the telerobot. In this paper, the system has been extended for a more demanding man-in-the-loop scenario. The service composition used is shown in Fig. 9, with two of the four services being reused from the previous composition. The OMS can be used to manage execution of complex event sequences and is used here to provide an

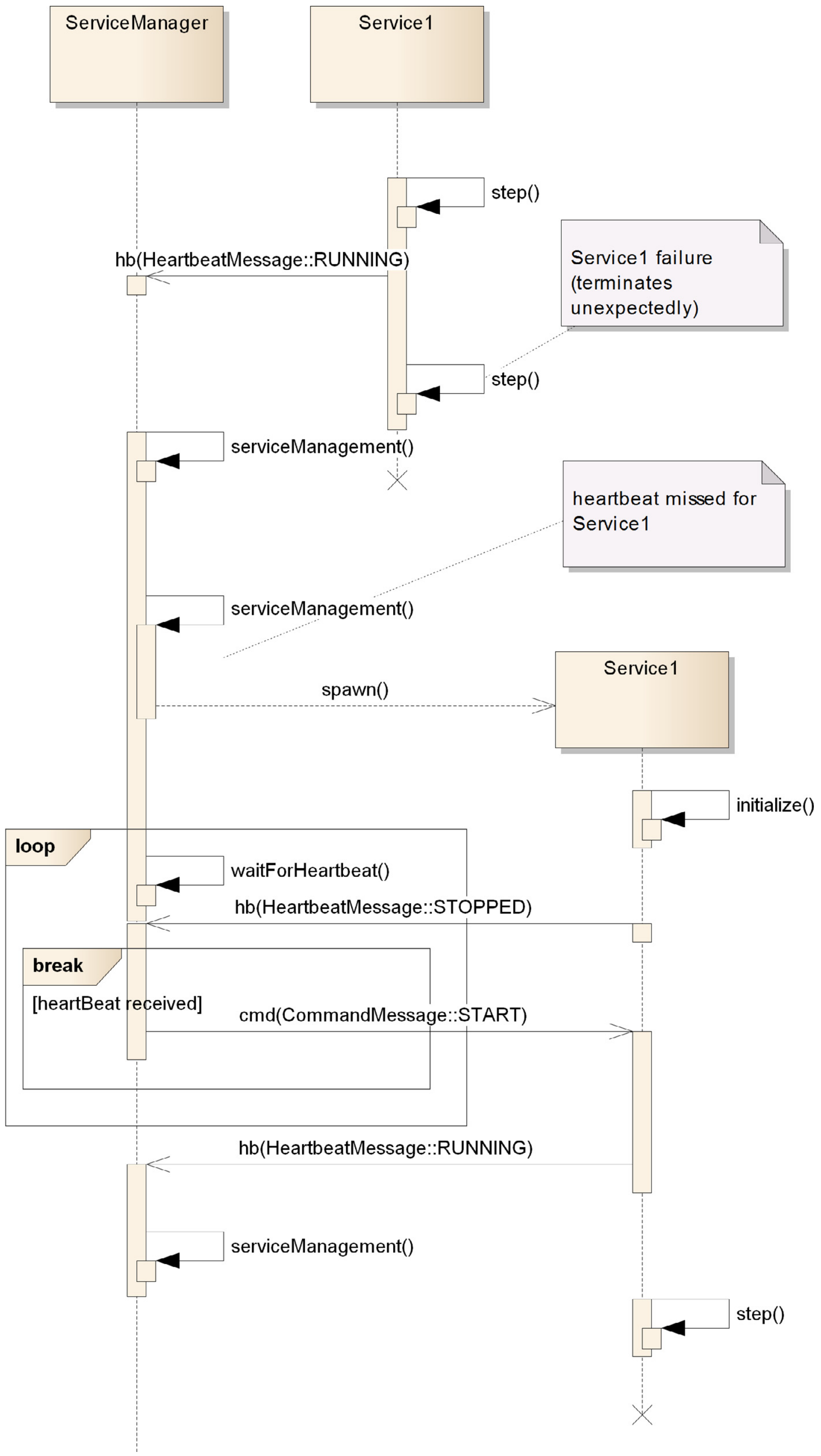

ServiceManager
Service1
step()
hb(HeartbeatMessage::RUNNING)
Service1 failure (terminates unexpectedly)
step()
serviceManagement()
heartbeat missed for Service1
serviceManagement()
Service1
spawn()
initialize()
loop
waitForHeartbeat()
hb(HeartbeatMessage::STOPPED)
break
[heartBeat received]
cmd(CommandMessage::START)
hb(HeartbeatMessage::RUNNING)
serviceManagement()
step()

Fig. 7. Sequence diagram for detecting and recovering service failure.

alternative operation mode. The nodes participating in the scenario include:

- **Equipment Controller** is a real-time system for operating RH equipment, i.e. the robot. Control service for robot (C4G) is responsible for abstracting the communications with the robot, sending position and velocity target values every 2 ms to the servo controllers of the robot using RTnet protocol. Missing a communication deadline with the robot causes an emergency stop (hard real-time requirement).

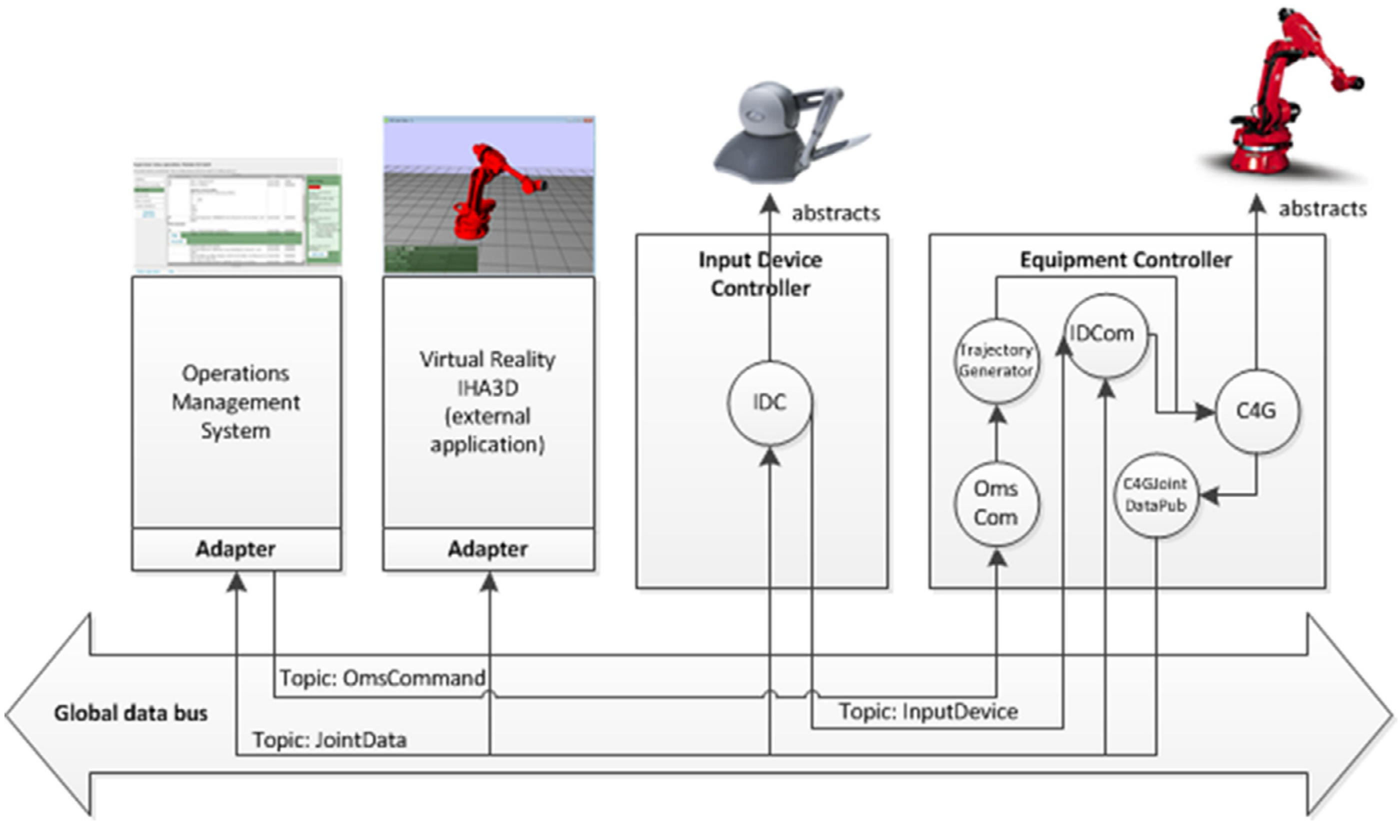


Fig. 8. Participating nodes.

- **Input Device Controller** is a non-real-time system, responsible for abstracting the input device. Missing several deadlines degrades performance (transparency of the system) which increases risk of accidental collisions.
- **Virtual Reality** software is used to visualize robot position, although the operator also has direct visual contact with the manipulator in the test setup. VR capabilities are provided by IHA3D, Windows-based software developed at the Department of Intelligent Hydraulics and Automation(IHA;Aha et al., 2011). The VR subsystem is connected to the remote handling system through a middleware adapter plugin, which allows it to track the position of the manipulator and publish virtual fixtures that can be used to overlay guiding forces to the force feedback signals from the slave manipulator.
- As an alternative mode of operation, **Operations Management System** can also be used to send commands to the robot. OMS is a Web server-based system, requiring different service composition and therefore improving system resilience by providing diversity.

4.2. Control system architecture

Aim of a teleoperation system is to enable the operator to perform precise work in inaccessible or hazardous environments, such as the ITER machine. The system consists of two manipulators, which are referred to as the master and the slave, and other necessary sensors and computers for implementing the control system architecture. The master manipulator is driven by the operator, and the slave manipulator is located in the remote environment, following movement commands given with the master (Aliaga et al., 2004).

The control system was implemented using the bilateral position–position(PP) control architecture (Aliaga et al., 2004), shown in Fig. 10. As demonstrated by the block diagram, system interacts physically with the operator, environment, master manipulator and slave manipulator in real-time. Timely data for the nodes areessential for safe operation of the system. Essentially, the architecture instructs both manipulators to track positions of each other. If the teleoperation system is not able to match the positions of the master and the slave, e.g. because of a physical or virtual obstacle, the difference generates a force F that drives positions of the manipulators to the same value. Joint position X_m from the master manipulator is used to control position of the slave, and vice versa. PP architecture is relatively stable and does not need expensive and delicate force sensors. As a downside, transparency (sensing of the remote environment) is worse when compared to more advanced bilateral teleoperation architectures.

Blocks in Fig. 10 denote different components affecting the control system. C_m and C_s are the position controller nodes IDC and EC, for the master and the slave respectively. Z_m is the impedance (mechanical resistance to motion) of the master manipulator, Z_h the operator's hand, Z_s the slave manipulator and Z_e the working environment of the manipulator.

4.3. Experimental setup

The experimental setup consisted of the following hardware units:

- Equipment controller PC: 3.4 GHz Pentium 4, Debian 5.0.8, Xenomai 2.5.6. Scheduler uses preemptive priority-based FIFO policy.
- Input device controller PC: 3.2 GHz Pentium 4, Lubuntu 4.8.1
- Virtual reality PC: 3.0 GHz Pentium 4, Windows XP SP3
- Phantom Omni 6 DOF haptic input device (Fig. 11)
- Comau Smart NM45-2.0 robotic manipulator (Fig. 11)
- Comau C4G robot controller with C4GOpen option Blomdell et al. (2005)

Services and their parameters (deployment shown in Fig. 9) are listed in Table 1. For use with OMS, IDCom and InputDeviceController

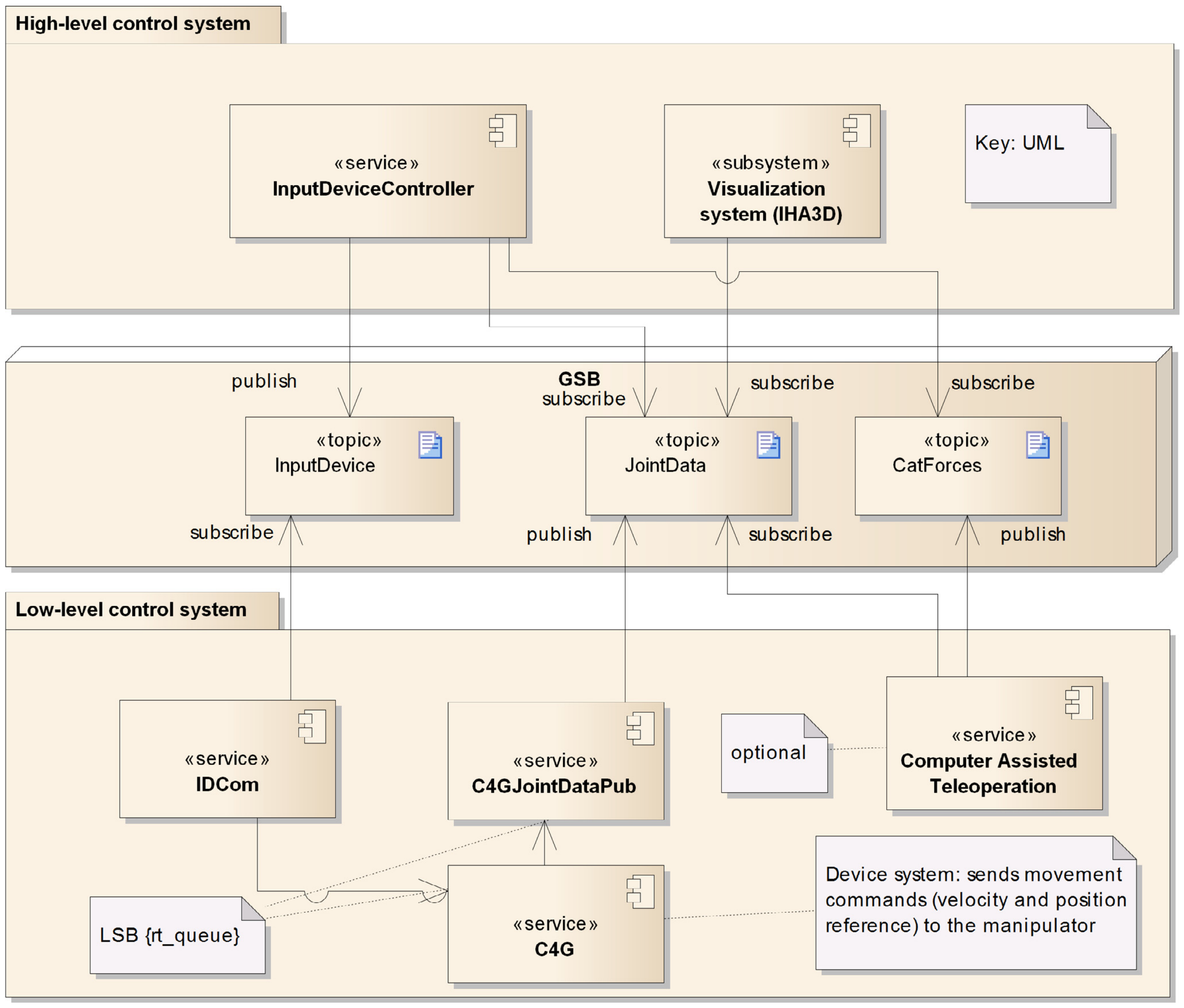


Fig. 9. Service composition used in the experiments (with IDC).

services are replaced with TrajectoryGenerator and OmsCom services in the composition.

The practical implementation of the control architecture is outlined briefly to document the experiment setup and demonstrate the 99% real-time requirement. Due to the applied filtering, a single lost packet has limited effect on the reference outputs, enabling the system to tolerate a missed deadline. IDCom service reads joint positions (i.e. position command) from the Phantom Omni, scales them based on user settings and publishes this data to the GSB. Joint values are scaled down because of the significant size difference between master and slave devices. EC runs the control services for the Comau robot, implementing the PP force feedback, as in Fig. 10: position change of the Phantom's joints is used to calculate position setpoint in IDCom service. Backward difference is used to estimate the derivatives for joint velocities. However, with small increments and high sampling rate, the resulting differentiated signals are often noisy. Therefore, a Geometric Moving Average (GMA) filter (Roberts, 1959) is used to calculate the joint position reference. For joint velocities, a finite difference method (Harrison and Stoten, 1995) is used to produce noise-suppressed differentiated signals.

Position and velocity setpoints are then sent from IDCom service to C4G service, which is integrated to the motor control servo loops of the robot with RTnet network protocol stack; this connection is timing critical and an exceeded deadline will trigger a fail-safe mode. Correspondingly, C4G service sends slave manipulator position to C4GJointDataPub service, which publishes this data to the GSB for IDC, in accordance with the PP control architecture. IDC reads this data to calculate forces for the master manipulator.

4.4. Evaluation criteria

Validation of the idea is based on empirical evaluation of the prototype implementation of the service-based control system architecture. We estimate the viability of the let-it-crash and recovery-oriented fault handling by performing recovery of failed real-time services, using service as the unit of fault isolation and estimating overhead of the active service management. Because the focus of evaluation is in the architectural style, the efficiency or coverage of the used fault detection and recovery mechanisms is not measured.

In the test case scenario (Section 4.1), an operator uses a haptic input device connected to the IDC to control an industrial manipulator. Faults are emulated by manually killing a service or altering service source code.

In brief, we evaluate following for real-time services:

1. Does the SM detect faults with (i) missed heartbeats, (ii) service status updates, and (iii) resource monitoring?

Table 1
Service names and parameters.

Service name	Task period	Priority	Heartbeat timeout	Restart limit	CPU limit
C4G	2 ms	91	-	0	-
IDCom	2 ms	40	5 ms	1	20%
C4GJointDataPub	10 ms	45	50 ms	1	40%
InputDeviceController	2 ms	non-real time	-	0	-

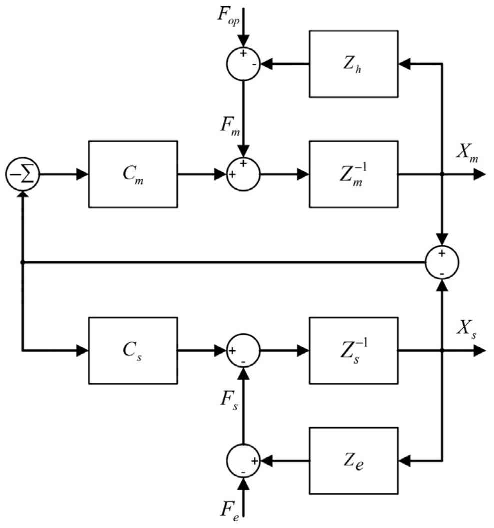


Fig. 10. Position–position control architecture.

2. Compare CPU load with service manager and using no service manager.
3. Fault handling with escalation to levels consisting of (i) restart command, (ii) kill and respawn service, and (iii) switch to backup service.
4. Test recovery of a critical real-time service that participates in the 500 Hz control loop.
5. Evaluate fault isolation: does service restart cause a cascade of failures?

5. Results and discussion

This section describes the results of the experiments we conducted to evaluate fault tolerance of the RTSOA architecture. All fault recovery tests were completed by running the same 120 s test sequence with the manipulator, consisting of starting the services, enabling drives on the manipulator, moving the manipulator, and then shutting the system down. CPU usage was measured by reading the proc filesystem on the EC.

Fault detection was tested to work with heartbeat, service status update, resource monitoring and service exit signals. Fault recovery escalation was tested to work with restart command, forced killing combined with restarting service, and forced killing combined with switching to backup service. The final escalation level is not considered in this test, as it simply terminates the service, necessitating operator interference.

Fig. 12 shows the effects of fault recovery on commanded and measured position and velocity values of the first joint of the manipulator, as recorded by the C4G service. Service IDCom was manually killed, the service manager detected the missing heartbeat, sent the restart command, detected the missing heartbeat again and escalated the fault handling to restarting of service (first fault), similar to sequence diagram of Fig. 7. Lack of valid commanded position and velocity are detected by the C4G service, while the IDCom is not available. Therefore, velocity output (measured velocity in the figure) is set to zero and commanded position (i.e. state data) held until the system is ready to resume normal execution. Second fault causes the service manager to load the backup service. We measured recovery of the first fault to take 1.440 s, and recovery of the second fault 2.314 s. This covers the time from detecting the fault (heartbeat timeout triggered) to C4G receiving the first data message from the new service, including initialization of the new service, recovering state data, sending heartbeat message to the service manager and the service manager sending the “start” command to the service. Most of this delay is caused by initialization of GSB communications middleware for the new service.

Resource usage was evaluated with the following comparisons: CPU usage with and without service management, faulty service with and without escalation, and scalability with different numbers of services.

Overhead of service management: CPU usage with no service manager (using services in standalone mode) and with service manager for Fig. 9 service composition, results shown in Fig. 13. Measured

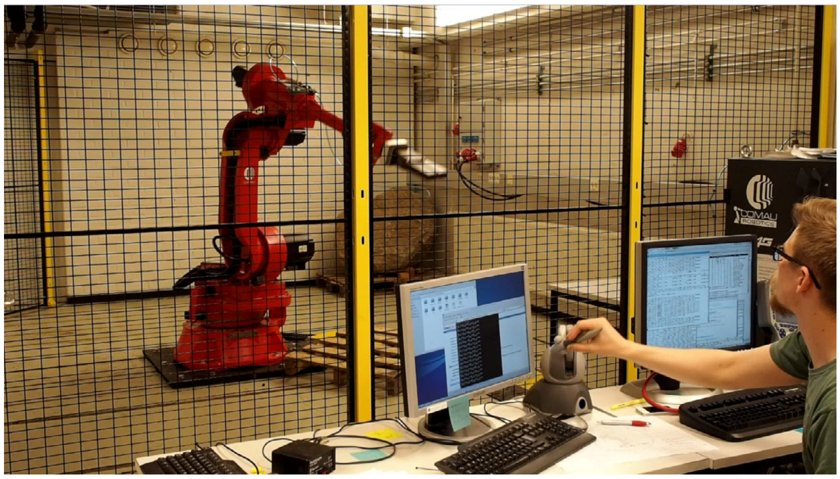


Fig. 11. Operator controlling the Comau robotic manipulator with the Phantom Omni haptic device.

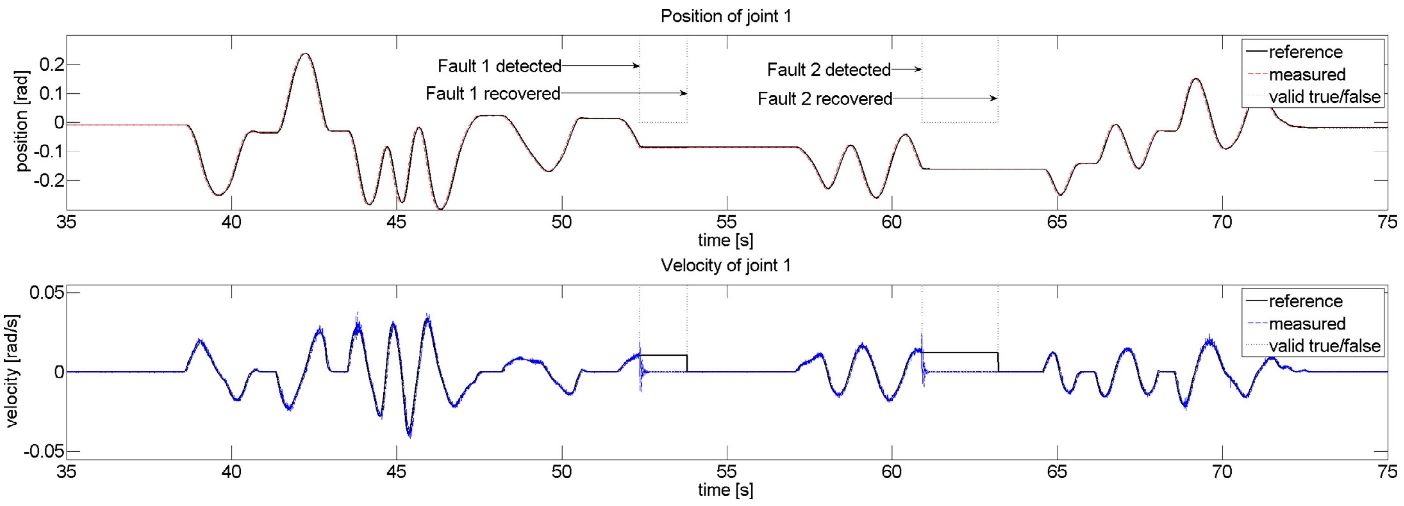


Fig. 12. Recovery from faults, detection based on missing heartbeats.

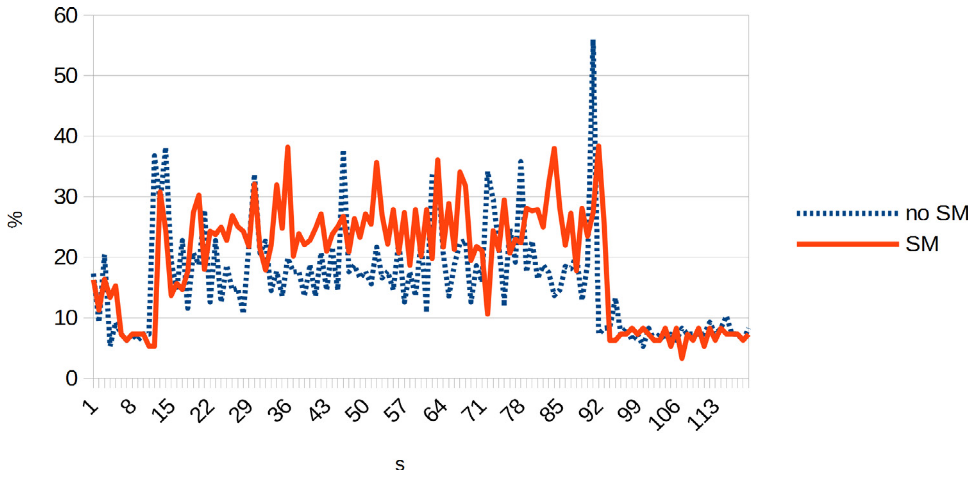


Fig. 13. Overhead evaluation, CPU usage with no service manager (SM) and with the SM using the IDC-based RH service composition.

average management overhead for the period when services were active was measured to be 4.65%.

Fault escalation: Test case has a faulty service (C4GJointDataPub with CPU limit set to 20%) that starts consuming too much CPU after 10 s due to a manually injected fault. Service manager escalates service recovery through levels I–III, finally switching the faulty service to a backup service and successfullyrecovering the fault. Fig. 14 shows measured CPU usage with service manager and comparison to standalone services without fault recovery. Without the service manager, the fault stays active in the system, keeping CPU usage on a higher level.

Scalability of service management for a single service vs. a large number of services is shown in Fig. 15. Comparison is performed with 1, 8 and 64 copies of a same dummy service. Based on the results, service management has a moderate fixed cost and low per-service variable cost—total CPU usage increased by a factor of 6.5 when the number of services was increased from 1 to 64. Service management overhead depends on heartbeat payload (for service status 4 bytes), service update frequencies (1 ms on dummy service), the rate of resource utilization checks, which can cause significant resource usage itself, and the task rate of the service manager. In this paper, we have used 1 ms task rate for the service manager in order to keep up with the monitoring of the managed services.

False positives are possible, e.g. due to overly strict heartbeat deadline combined with a (re)starting high-priority service, which temporarily starves other, lower-priority services. The amount of false positives depends on heartbeat deadline implementation and configuration and was not estimated in this research.

Fault propagation was not observed during experiments with the emulated faults, as services correctly responded to suddenly killed dependencies by moving to the restarting state to wait for resuming of operation. Therefore degradation of service composition reliability due to service failures was not an issue in the experiments.

To summarize, the results of our experimental evaluation validate that the RTSOA supports fault detection and recovery in distributed real-time systems:

- Several fault detection methods were implemented and tested to function correctly.
- Services were successfully used as units of fault isolation.
- Let-it-crash approach (termination and restart of services upon encountering errors) was successfully used to recover faults in a critical real-time application.
- The system was capable of recovering from different types of faults and continuing operation without cascading failures.
- Service compositions can be dynamically changed, including switching to back-up service on the fly. This causes a temporary disruption in the delivery of service by the system, which is resumed after the services are in the running state.
- Overhead for service management in the RH scenario was measured to be fairly low, around 5%, and shown to scale in a manner

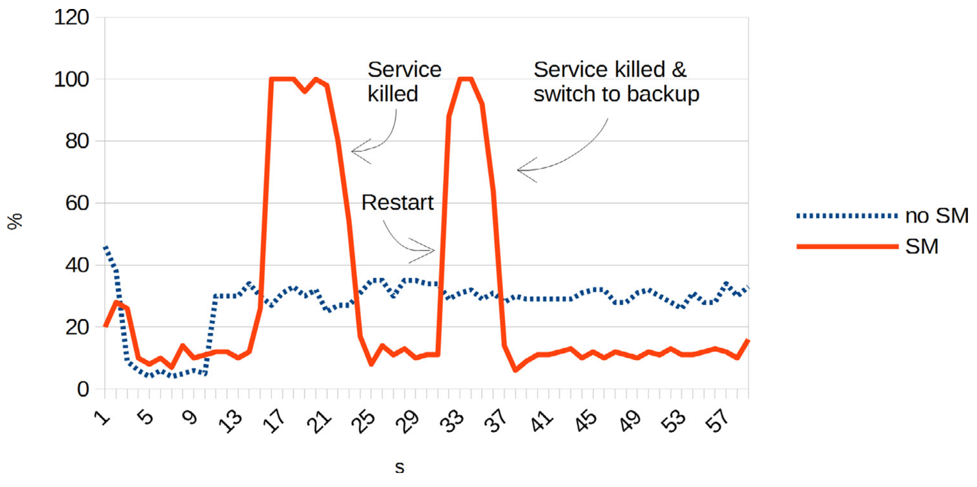


Fig. 14. Recovery from faults with escalation of fault handling, CPU usage.

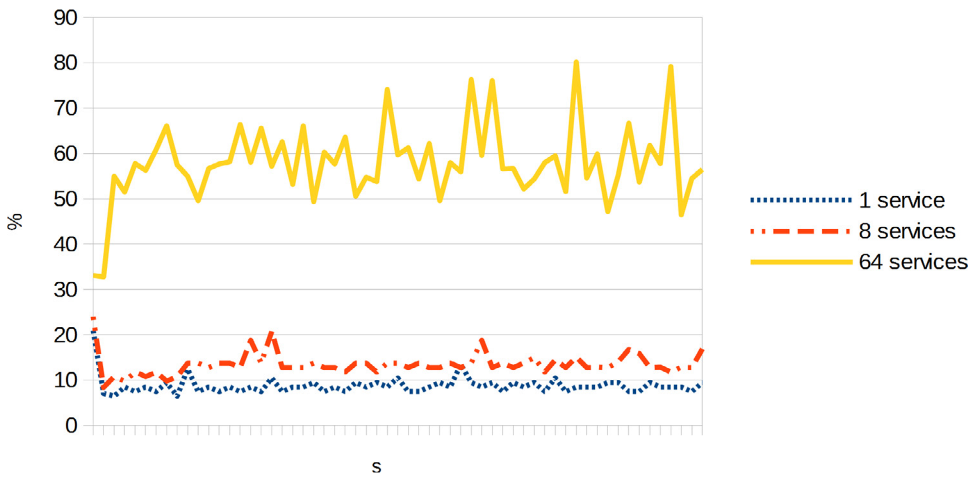


Fig. 15. Scalability of service management, CPU usage with 1, 8 and 64 services.

that enables deployment and monitoring of large numbers of services.

6. Related work

Sulava is a generic RTSOA, validated with distributed working machines using hardware-in-the-loop simulation (Hahto et al., 2011), and used as the reference architecture for the implementation in this paper. Compared to Hahto et al. (2011), our work (Alho and Mattila, 2013) applies the Sulava architecture to a distributed robotic system with more heterogeneous and diverse environment, more demanding real-time requirements and use of physical robot with automatic teleoperation instead of simulation models. Our previous work (Alho and Mattila, 2014) extended the fault-tolerance features of the architecture design and implemented them for the prototype system, with validation based on analysis. This paper presents so far the most complete description of the architecture, further extensions to the architecture fault tolerance mechanisms regarding fault escalation and service state management, and also extends the implementation to use more demanding bilateral teleoperation, presenting an extensive empirical evaluation of the fault tolerance approach for building of dependable and resilient CPSs.

Although fault tolerance and resilience have been recognized as key elements for CPSs (Bonakdarpour, 2008; Rajkumar et al., 2010), relatively little research is available on the topic of fault tolerance of CPS architectures, e.g. service recovery. Most work related to CPS dependability and resilience is on the topics of resilient controllers (based on control theory) and security. Naturally, most of the traditional fault tolerance research is directly applicable to CPSs, but the capability to improve system resilience based on the loosely coupled nature of SOA has been somewhat neglected. Research on this topic includes using SOA to improve system dependability by enhancing the process of exchanging a device in an industrial automation environment in case of device breakdowns and replacements (Cândido et al., 2013). On the service-level, roll-back recovery of Web services (Mansour and Dillon, 2011) and byzantine fault-tolerant Web services (Pallemulle et al., 2008) have also been used successfully. These approaches, based on improving fault tolerance of services themselves, should be seen as complementary to the one presented in this paper.

Dynamic resource management can be used to ensure CPS capability to adapt to dynamically changing conditions. Lardieri et al. have developed a dynamic resource management system for enterprise DRE systems (Lardieri et al., 2007) based on Real-time CORBA Component Model. Their research indicates that statically provisioned DRE systems require a great deal of manual engineering effort to create and validate reconfigurations, whereas dynamic resource management capabilities can be used to extend system operational flexibility without human intervention. They also note that QoS and deployment configuration of components for DRE systems is a complex task that has not yet been solved. The same problem has also been noticed in the development of the Sulava platform.

An approach to cope with the dynamic behavior of distributed real-time systems using reconfiguration of services is presented by García-Valls et al. (2014) to achieve time-bounded real-time reconfigurability for RTSOA systems modeled as graphs, using a real-time prune algorithm to search for schedulable reconfiguration solutions. The reconfiguration approach has been used to develop the iLAND middleware framework for time-bounded operation and reconfiguration in service-oriented soft real-time systems (Garcia Valls et al., 2013; García-Valls et al., 2014). Although fault detection and recovery are not specifically evaluated, the approach could be used to provide fault recovery capabilities. Compared to the Sulava platform, which focuses on evaluating fault tolerance capabilities on service-level and provides best-effort service recovery, iLAND provides time-bounded reconfiguration of service compositions, but requires an a priori temporal analysis of the system (Garcia Valls et al., 2013).

Cloud support for real-time systems is a new development that can provide scalability and efficient use of resources for CPSs. However, use of cloud infrastructure can increase errors, especially because latency of virtual machines is unknown. Malik and Huet (2011) propose a fault tolerance scheme based on replication of software on multiple virtual machines and quantification of node reliability. This approach could potentially be combined with RTSOA to run critical services on cloud infrastructure.

Component-based approach to fault tolerance for service robots (viz. CPS) has been researched by Ahn et al. (2012). Their work focuses on how typical fault tolerance mechanisms can be accommodated to a component-based architecture and evaluates the implemented, fault manager-based, fault detection mechanisms and their performance, achieving fault recovery performance in the range of 20–500 ms. Similar to our empirical results, most of the recovery time is spent while loading components, and as a solution the authors implemented preloading of components, which shortened recovery times to less than 20 ms, although no estimation for the overhead of this form of fault tolerance is given.

7. Limitations

Our prototype implementation demonstrates that it is possible to use a service-oriented approach in real-time systems to improve fault tolerance and robustness, but it has certain limitations:

1. No built-in recovery mechanisms for permanent faults, as restart-based recovery does not remove the root cause of the error. This is handled through resilience, i.e. backup services and diverse service compositions.
2. Safety considerations. Although a faulty service can be restarted, it is not necessarily safe to do so (see Section 3.4).
3. Non-deterministic communication delays over networks, i.e. hard real-time for the service connections is not guaranteed.
4. Cannot guarantee fail-silent behavior, although the service manager can detect some abnormal behavior patterns; it is possible that errors can propagate to another services in some cases.
5. High-priority service start-up can violate autonomy of lower-priority services temporarily.

In this work we have focused on feasibility of using RTSOA capabilities as a basis for fault tolerance and have therefore omitted schedulability analysis of the system. Also, some features that would be expected from a finalized SOA have not been implemented in the prototype. Specifically, we have left composition infrastructure (e.g. for business processes), end-to-end service chain predictability, security aspects and dynamic resource management outside the scope of this research. Service configuration was implemented using static configuration locally, instead of dynamic configuration through GSB. Distributed service management and service management GUI were not used in this paper, although they have been tested in related, on-going work using the Sulava platform.

Experiments were carried out on relatively low-performance computing hardware, and therefore should be applicable to modern embedded systems and industrial PCs, but exact performance and overhead numbers depend on platform, computing hardware and implementation. One benefit of using autonomous services is the capability to utilize benefits of concurrency provided by multi-core processors, which are becoming increasingly available even for embedded systems. The experiments in this paper were carried out on single-core CPUs; using multi-core CPUs would very likely have a positive effect on predictability and performance and enable bin-packing solutions for real-time tasks (services) as proposed in Lakshmanan et al. (2010).

8. Conclusions

This paper has described an approach to fault tolerance for CPSs based on loosely coupled real-time services and presented an empirical evaluation of the approach using a remote handling system as a demanding real-world test case, which included combination of heterogeneous and autonomous subsystems collaborating to provide force-feedback control of a telerobot. Based on the validation, RTSOA supports real-time fault detection and recovery, providing the ability to implement fault tolerance patterns flexibly and without significant overhead. The main contributions of this paper include showing that services can function as a unit of fault isolation and recovery without cascading failures, allowing recovery of transient faults using partial restarts (let-it-crash approach) that should improve system availability compared to monolithic systems. Resilience of the system is improved through assessability with virtual models and diversity from operating modes (compositions) and alternative back-up versions of services. Our work also emphasizes evolvability of the system, supported by use of standardized middleware and open platforms.

Acknowledgements

This work was partly carried out under the EFDA Goal Oriented Training Programme (WP10-GOT-GOTRH) and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

References

Aha, L., Salminen, K., Hahto, A., Saarinen, H., Mattila, J., Siuko, M., Semeraro, L., 2011. Dtp2 control room operator and remote handing operation designer responsibilities and information available to them. Fusion Eng. Des. 86 (9–11), 2078–2081.
Ahn, H., Loh, W.K., Yeo, W.Y., 2012. A framework-based approach for fault-tolerant service robots. Int. J. Adv. Robot. Syst. 9 (200), 1–10.
Alho, P., Mattila, J., 2013. Real-time service-oriented architectures: A data-centric implementation for distributed and heterogeneous robotic system. In: Embedded Systems: Design, Analysis and Verification. Springer, pp. 262–271.
Alho, P., Mattila, J., 2014. Software fault detection and recovery in critical real-time systems: An approach based on loose coupling. Fusion Eng. Des. 89 (9–10), 2272–2277.
Alho, P., Rauhamäki, J., 2014. Patterns for light-weight fault tolerance and decoupled design in distributed control systems. In: LNCS Transactions on Pattern Languages of Programming. Springer in press.
Aliaga, I., Rubio, A., Sanchez, E., 2004. Experimental quantitative comparison of different control architectures for master-slave teleoperation. IEEE Trans. Contr. Syst. Technol. 12 (1), 2–11.
Alonso, J., Grottke, M., Nikora, A.P., Trivedi, K.S., 2013. An empirical investigation of fault repairs and mitigations in space mission system software. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 1–8.
Atzori, L., Iera, A., Morabito, G., 2010. The internet of things: A survey. Comput. Netw. 54 (15), 2787–2805.
Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Depend. Secure Comput. 1 (1), 11–33.
Blomdell, A., Bolmsjö, G., Brogårdh, T., Cederberg, P., Isaksson, M., Johansson, R., Haage, M., Nilsson, K., Olsson, M., Olsson, T., et al., 2005. Extending an industrial robot controller-implementation and applications of a fast open sensor interface. IEEE Robot. Automat. Mag. 12 (3), 85–94.
Bonakdarpour, B., 2008. Challenges in transformation of existing real-time embedded systems to cyber-physical systems. ACM SIGBED Rev. 5 (1), 11.
Brown, J.H., Martin, B., 2010. How fast is fast enough? Choosing between xenomai and linux for real-time applications. In: Proceedings of the 12th Real-Time Linux Workshop (RTLWS'12). OSADL, pp. 1–17.

Candea, G., Brown, A.B., Fox, A., Patterson, D., 2004. Recovery-oriented computing: building multitier dependability. Computer 37 (11), 60–67.
Candea, G., Fox, A., 2003. Crash-only software. In: HotOS, vol. 3. Usenix, pp. 67–72.
Cândido, G., Barata, J., Colombo, A.W., Jammes, F., 2009. Soa in reconfigurable supply chains: A research roadmap. Eng. Appl. Artif. Intell. 22 (6), 939–949.
Cândido, G., Sousa, C., Di Orio, G., Barata, J., Colombo, A.W., 2013. Enhancing device exchange agility in service-oriented industrial automation. In: 2013 IEEE International Symposium on Industrial Electronics (ISIE). IEEE, pp. 1–6.
Cucinotta, T., Mancina, A., Anastasi, G.F., Lipari, G., Mangeruca, L., Checcozzo, R., Rusinà, F., 2009. A real-time service-oriented architecture for industrial automation. IEEE Trans. Indus. Inform. 5 (3), 267–277.
Dai, W., Vyatkin, V., Christensen, J.H., 2014. The application of service-oriented architectures in distributed automation systems. In: 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp. 252–257.
Erl, T., 2008. Soa: Principles of Service Design. vol. 1. Prentice Hall.
Garces-Erice, L., 2009. Building an enterprise service bus for real-time soa: A messaging middleware stack. In: 33rd Annual IEEE International Computer Software and Applications Conference, 2009 (COMPSAC'09), vol. 2. IEEE, pp. 79–84.
García Valls, M., Basanta Val, P., 2013. A real-time perspective of service composition: Key concepts and some contributions. J. Syst. Arch. 59 (10), 1414–1423.
Garcia Valls, M., López, I.R., Villar, L.F., 2013. iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. IEEE Trans. Indus. Inform. 9 (1), 228–236.
García-Valls, M., Uriol-Resuela, P., Ibáñez-Vázquez, F., Basanta-Val, P., 2014. Low complexity reconfiguration for real-time data-intensive service-oriented applications. Futur. Gener. Comput. Syst. 37, 191–200.
Gray, J., 1986. Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems. Los Angeles, CA, USA, pp. 3–12.
Grottke, M., Nikora, A.P., Trivedi, K.S., 2010. An empirical investigation of fault types in space mission system software. In: 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 447–456.
Hahto, A., Rasi, T., Mattila, J., Koskimies, K., 2011. Service-oriented architecture for embedded machine control. In: 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, pp. 1–4.
Hanmer, R., 2007. Patterns for Fault Tolerant Software. John Wiley & Sons.
Harrison, A., Stoten, D., 1995. Generalized finite difference methods for optimal estimation of derivatives in real-time control problems. Proc. Inst. Mech. Eng. I: J. Syst. Contr. Eng. 209 (2), 67–78.
Huhns, M.N., Singh, M.P., 2005. Service-oriented computing: Key concepts and principles. IEEE Inter. Comput. 9 (1), 75–81.
International Electrotechnical Commission, 2010. IEC61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems.
Jackson, D., 2009. A direct path to dependable software. Commun. ACM 52 (4), 78–88.
Kim, K.D., Kumar, P.R., 2012. Cyber–physical systems: A perspective at the centennial. Proc. IEEE 100 (Special Centennial Issue), 1287–1308.
Lakshmanan, K., De Niz, D., Rajkumar, R., Moreno, G., 2010. Resource allocation in distributed mixed-criticality cyber-physical systems. In: 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp. 169–178.
Laprie, J.C., 2008. From dependability to resilience. In: 38th IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, pp. G8–G9.
Lardieri, P., Balasubramanian, J., Schmidt, D.C., Thaker, G., Gokhale, A., Damiano, T., 2007. A multi-layered resource management framework for dynamic resource management in enterprise dre systems. J. Syst. Softw. 80 (7), 984–996.
Lawrence, D.A., 1993. Stability and transparency in bilateral teleoperation. IEEE Trans. Robot. Automat. 9 (5), 624–637.
Lee, E.A., 2008. Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC). IEEE, pp. 363–369.
Malik, S., Huet, F., 2011. Adaptive fault tolerance in real time cloud computing. In: 2011 IEEE World Congress on Services (SERVICES). IEEE, pp. 280–287.
Mansour, H.E., Dillon, T., 2011. Dependability and rollback recovery for composite web services. IEEE Trans. Serv. Comput. 4 (4), 328–339.
Moussa, H., Gao, T., Yen, I.L., Bastani, F., Jeng, J.J., 2010. Toward effective service composition for real-time soa-based systems. Serv. Orient. Comput. Appl. 4 (1), 17–31.
Pallemulle, S.L., Thorvaldsson, H.D., Goldman, K.J., 2008. Byzantine fault-tolerant web services for n-tier and service oriented architectures. In: The 28th International Conference on Distributed Computing Systems, 2008 (ICDCS'08). IEEE, pp. 260–268.
Pan, Y., 2009. Will reliability kill the web service composition. Technical Report. Department of Computer Science, Rutgers University, USA.
Panahi, M., Nie, W., Lin, K.J., 2009. A framework for real-time service-oriented architecture. In: IEEE Conference on Commerce and Enterprise Computing, 2009 (CEC'09). IEEE, pp. 460–467.
Papazoglou, M.P., 2003. Service-oriented computing: Concepts, characteristics and directions. In: Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003 (WISE'03). IEEE, pp. 3–12.
Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., et al., 2002. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175. UC Berkeley Computer Science.
Rajkumar, R.R., Lee, I., Sha, L., Stankovic, J., 2010. Cyber-physical systems: the next computing revolution. In: Proceedings of the 47th Design Automation Conference. ACM, pp. 731–736.
Rauhamäki, J., Vepsäläinen, T., Kuikka, S., 2012. Functional safety system patterns. In: Proceedings of the VikingPLoP 2012 Conference. TUT, pp. 48–68.
Roberts, S., 1959. Control chart tests based on geometric moving averages. Technometrics 1 (3), 239–250.
Tiderko, A., Bachran, T., Hoeller, F., Schulz, D., 2008. Rose—A framework for multicast communication via unreliable networks in multi-robot systems. Robot. Autonom. Syst. 56 (12), 1017–1026.
Tuominen, J., Viinikainen, M., Alho, P., Mattila, J., 2014. Using a data-centric event-driven architecture approach in the integration of real-time systems at dtp2. Fusion Eng. Des. 89 (9–10), 2289–2293. doi:10.1016/j.fusengdes.2014.04.040.

Pekka Alho received M.Sc. (Eng.) in 2009 from Tampere University of Technology (TUT), Finland. He is a research scientist at the Departmentof Intelligent Hydraulics and Automation of TUT, where he participated in ITER Remote Handling maintenance projects that involve research on heavy-duty hydraulic robotic manipulators and control systems. Currently he is finalizing his doctoral dissertation on fault tolerance of cyber-physical systems. Earlier in 2007–2010 he worked as a researcher at Departmentof Automation Science and Engineering. His research interests include distributed robotics, safety-critical systems and software architectures.

Jouni Mattila received M.Sc. (Eng.) in 1995 and Dr. Tech. 2000 from TUT. From 1998 to 2000 he was a visiting researcher at the University of British Columbia. He is currently a professor at Department of Intelligent Hydraulics and Automation. For the past 10years, he has been program manager in ITER Remote Handling maintenance projects. Currently, he coordinates European Marie Curie-project with 15 Ph.D.-students developing mobile service robotics for scientific infrastructures such as Cern. His research interests include machine automation, preventive maintenance, and control system development for robotic manipulators and off-highway machinery. He has over 100 papers to his name.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3642-7
ISSN 1459-2045