DISSERTATIONES
TECHNOLOGIAE
UNIVERSITATIS
TARTUENSIS
90

ROBERT VALNER

Design of TeMoto, a software framework for dependable, adaptive, and collaborative autonomous robots

DISSERTATIONES TECHNOLOGIAE UNIVERSITATIS TARTUENSIS
90

DISSERTATIONES TECHNOLOGIAE UNIVERSITATIS TARTUENSIS
90

ROBERT VALNER

Design of TeMoto, a software framework for dependable, adaptive, and collaborative autonomous robots

UNIVERSITY OF TARTU
Press

Institute of Technology, Faculty of Science and Technology, University of Tartu, Estonia

The dissertation was accepted for the commencement of the degree of Doctor of Philosophy in Engineering of Physical Engineering on 07.11.2024, by the Joint Council of the Doctoral Program of Engineering and Technology of the University of Tartu.

Supervisors: Karl Kruusamäe, PhD
Associate Professor of Robotics Engineering
Institute of Technology, Faculty of Science and Technology
University of Tartu, Estonia

Alvo Aabloo, PhD
Professor of Polymeric Materials, Materials Science
Institute of Technology, Faculty of Science and Technology,
University of Tartu, Estonia,

Reviewer: Huber Raul Flores Macario, PhD
Associate Professor of Pervasive Computing
Institute of Computer Science, Faculty of Science and Technology
University of Tartu, Estonia

Opponent: Markus Vincze, PhD
Automation and Control Institute
Vienna University of Technology, Austria

Commencement: Auditorium 121, Nooruse 1, Tartu, Estonia, at 13.15
on Detsember 4th, 2024

The publication of this dissertation was financed by the Institute of Technology, University of Tartu, and in part supported by the project „Increasing the knowledge intensity of Ida-Viru entrepreneurship“ co-funded by the European Union.

ISSN 2228-0855 (print)
ISBN 978-9916-27-741-6 (print)
ISSN 2806-2620 (pdf)
ISBN 978-9916-27-742-3 (pdf)

Copyright: Robert Valner, 2024

University of Tartu Press
www.tyk.ee

ABSTRACT

Reducing human involvement in hazardous, stressful, and tedious tasks has been among the main driving reasons for reaching self-sufficient, autonomous robots. However, achieving reliable autonomy in unpredictable and dangerous application domains is a challenge that combines two borderline contradictory aspects – system reliability and complexity. Thus, teleoperated robotic systems are often preferred. Nevertheless, when a wired connection is not an option and a wireless connection is intermittent, semi- or fully autonomous capabilities become paramount.

The goal of this work is to a) analyze the software architecture design principles that lead to the increase of LoA of robots deployed for high-risk and high-complexity tasks; and b) develop a software architecture that implements the concluded design principles. This work contributes by developing an adaptive, scalable, multi-agent, and human-robot collaboration-oriented software architecture, TeMoto, derived from analyzing a variety of high-risk task domains and common design principles for a robotic software stack.

TeMoto is based on a decentralized multi-robot variant of a three-layer architecture, i.e., syndicate architecture, and enables dynamic task (executive layer) and resource (functional layer) management. Tasks are outlined in Unified Meaning Representation Format (UMRF), a novel domain-specific language that allows the description of complex hierarchical multi-robot tasks in JSON format. TeMoto Action Engine is a C++ based library that implements the semantics of UMRF. Individual behaviors in a task, i.e., actions, are defined as dynamically loadable modular plugins that can be concurrently executed. The resource management layer provides dynamic control over the lifecycle of resources, such as sensors and actuators. It provides reference counting and hierarchical dependency management, allowing for correct resource allocation, deallocation, and error propagation along the dependency chain. TeMoto is completely open-source, and is designed to work with ROS and ROS2, while the core tools can be used outside ROS.

The work is evaluated via five technical demonstrators, covering the fundamental principles of resource management, task management, and use cases involving human-robot interaction and multi-robot systems. The development of TeMoto is an ongoing process, and this work captures the current state, outlining the core design principles and set of implemented tools.

CONTENTS

List of original publications 15

1. Introduction 17
1.1. Research Questions 19
1.2. Contributions 19
1.2.1. Case Study on Previous Work 20
1.2.2. Technical Summary 20
1.3. Thesis Outline 21

2. Preliminaries 22
2.1. Application Domain Review 22
2.2. Desired Autonomous Capabilities 23
2.2.1. Adaptability and Dependability 24
2.2.2. Scalability 25
2.2.3. Multi-Agent Collaboration Oriented Design 25
2.2.4. Human-Robot Interaction-Oriented Design 26
2.3. Architectures for Autonomous Robots 26
2.3.1. Deliberative Layer 28
2.3.2. Executive Layer 30
2.3.3. Functional Layer 31
2.4. Summary 32

3. Requirements Development 34
3.1. Common Requirements 35
3.2. Executive Layer Requirements 36
3.3. Functional Layer Requirements 37
3.4. Summary 37

4. Related Work 38
4.1. Layered Architecture Implementations 38
4.2. Executive layer 39
4.3. Functional Layer 42
4.4. Summary 43

5. TeMoto – Architecture Overview 45
5.1. Requirements Conformance 48

6. TeMoto – Task Management 50
6.1. Model 50
6.1.1. Actions 52
6.1.2. Graph 52
6.1.3. Conditions 53

6.1.4. State transitions 53
6.2. Text-Based Format 55
6.3. Implementation 57
6.3.1. Hierarchical Graph Resolution 58
6.3.2. N-to-N Handshake 61
6.3.3. Action Evolution 61
6.3.4. Action Execution 61
6.4. Requirements Conformance 64
6.5. Summary 65

7. TeMoto – Resource Management **66**
7.1. Definitions and Semantics 68
7.1.1. Resource Request 69
7.1.2. Resource Release 69
7.1.3. Resource Status Update 70
7.2. Implementation 70
7.3. Requirements Conformance 75
7.4. Summary 75

8. Evaluation **77**
8.1. Demo 1 – Fault Tolerant Sensor Redundancy 77
8.2. Demo 2 – Multi-Robot Escort 77
8.3. Demo 3 – Mission Adaptiveness 78
8.4. Demo 4 – LLM Driven Task Planning 80
8.5. Demo 5 – Multi-Robot Environment Knowledge Sharing 82
8.6. Summary 86

9. Discussion **87**
9.1. Task Management 87
9.2. Resource Management 87
9.3. Evaluation 88
9.4. Lessons Learned 89

10. Conclusion **90**

Bibliography **92**

Acknowledgements **98**

Sisukokkuvõte (Summary in Estonian) **99**

Publications **101**
TeMoto: Intuitive Multi-Range Telerobotic System with Natural Gestural and Verbal Instruction Interface 103
TeMoto: A Software Framework for Adaptive and Dependable Robotic Autonomy With Dynamic Resource Management 127

Unified Meaning Representation Format (UMRF) - A Task Description and Execution Formalism for HRI 149

Curriculum Vitae **176**

Elulookirjeldus (Curriculum Vitae in Estonian) **177**

LIST OF FIGURES

1. State-of-the-art real-world deployment of robots compared in the scale of risk, LoA, and environment complexity. This work addresses the technological gap in improving robot autonomy in high-risk, complex environments. 18
2. Design challenges for an adaptive, modular, and scalable software architecture for complex autonomous robots. 27
3. Different architectures and architectural components depicted via *sense-plan-act* model. The original sequentially executed SPA model (a). For increased reactivity, the planning step is omitted (b), leading to reactive architectures (c). While reactive architectures responded well to sudden environmental changes, the lack of planning limited their use in lengthy and complex tasks, leading to layered architectures (d) where each concurrently operating layer controls the one below. Often the *behavioral* and *executive* layers are combined into a single *executive* layer. 29
4. Syndicate architecture scales the layered architecture into a decentralized multi-robot system. 30
5. Example of a reactive implementation of the task “Find the object and put it on the table” represented as flowchart (a) with interrupts, Finite State Machine (b), Petri-Net Plan (c), Behavior Tree (d). . . 33
6. The architecture of TeMoto consists of task management and resource management layers. 46
7. Extension of TeMoto architecture to a decentralized and heterogeneous multi-agent system. 47
8. Structure of the Task Management layer in the TeMoto framework. Tasks can be issued via different input sources (a), where each source is parsed to a common UMRF graph (b). The UMRF graph maps to respective actions on the robot (c), thus UMRF formalism makes the robot independent from different input modalities. 50
9. Main semantic properties of UMRF graph notation, including sequences (a), concurrency (b), cycles (c), hierarchical graphs (d), multi-agent graphs (e), conditionals, and error management (f). Example UMRF graph with aforementioned properties combined, where “S” denotes graph entry and “E” denotes graph exit (g). 51
10. Example UMRF graph where actions are denoted via "A" and conditions for running actions via "C". 53
11. Potential use-case of a multi-modal HRI system that benefits from having an intermediate task description format for fusing the modalities and commanding the robot. 55
12. Structure of the TeMoto Action Engine. 58

13. Action graph of the autonomous inspection task (a), which demonstrates the cases of dynamic initialization (b), error propagation (c), and un-initialization (d) of shared and hierarchical resources, all of which are dependent on resource management's reference and hierarchical dependency accounting functionalities. 67
14. Abstract structure of the provider-resource-consumer relation in resource management (a), and an example of hierarchical resources, and multi-resource consumer use-cases (b). Including Robot Manager and Process Manager illustrate this specific example and are not conceptual elements in resource management. 68
15. Implementation of resource consumers and providers in TeMoto framework. Hierarchical resource management is not depicted. . . 72
16. Integration of resource consumers in user's custom application. Each resource manager can be accessed via the corresponding interface, which provides a simplified API. 72
17. Working principle of RIS. When a resource manager is initialized, it advertises the available resources, e.g., R_0, through RIS (a). The advertised message is captured by the RIS of another robot's resource manager. Now if the other robot wants to load R_0, the request is automatically redirected to the actual provider of R_0 (b). 73
18. Overview of demo 1, i.e., fault-tolerant sensor redundancy experiment. Visual feedback from the 3D LIDAR (a). Once compromised, feedback is dynamically regained via a depth camera (b), and then a 2D LIDAR (c). 78
19. Overview of demo 2, i.e., multi-robot escort via resource redundancy experiment. The operator teleoperates the Worker via OCS and sees the visual feedback from the Worker's camera (a). Worker's camera fails (b). The Escort robot is instructed to follow the Worker and provide feedback to the operator allowing them to retrieve the Worker which would otherwise be left in the field (c). 79
20. Overview of Demo 3, i.e., mission adaptiveness experiment. The deployed robot (Jackal) performs a surveillance task, passing through locations L_1 to L_3 (a). Then the Jackal is asynchronously re-tasked to deliver objects from location L_p to L_d. 81
21. LLM driven task planning scenario in the operator's AR headset perspective. The operator issues commands via speech (a) and virtual marker (b) based interface. The command is then sent to OpenAI, which returns a UMRF graph that is visualized on the AR interface and executed on the robot (c).[88] 83

22. Prompt that is designed to make GPT-3 extract UMRF graphs from a combined input containing voice commands and virtual marker coordinates. The beginning of the prompt contains the description of the request. Each prompt also embeds five operator command + UMRF graph pair examples. Finally, the voice commands and location of the virtual marker are concatenated as a string and added to the prompt. The prompt is then passed to GPT-3, which returns a UMRF JSON string.[88] 84
23. Multi-robot environment knowledge sharing scenario, where three robots are used to locate and manipulate objects of interest (a). First the two scout robots dynamically locate the objects (trash and a trash can) and share the information to all agents (b). Then the worker robot, knowing the location of the objects, picks up the trash (c) and places it in the trash can (d). 85

LIST OF TABLES

1. Evaluation of TRL and LoA of robots deployed for tasks with high-risk and high-complexity. 23
2. TRL and ATRA LoA level descriptions. Acronyms: Real Time (RT), External System Independence (ESI). A detailed description of the ATRA LoA scale can be found in [20]. 24
3. Common requirements 35
4. Specialization of common requirements to the executive layer ($E_{C\text{-}1}$ to $E_{C\text{-}4}$) and executive layer specific requirements (*E-1* to *E-4*). . . 36
5. Specialization of common requirements to the functional layer (F_{C1} to F_{C3}) and functional layer specific requirements (*F-1* to *F-3*). . . 37
6. Summary of the capabilities of layered architecture reported in the literature. Comparison based on common requirements defined in Table 3. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×" and "*unkn.*" (unknown) indicates unavailability of information. 40
7. Summarized capabilities of common robotic task management tools. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×". . 42
8. Coverage of resource management capabilities by the existing work. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×" and "*unkn.*" (unknown) indicates unavailability of information. 43
9. TeMoto architecture's conformance to common requirements (Table 3). 48
10. State transition table. The events that can trigger the transition are denoted as "O", "L", and "E", or Outer, Local, and Evolution respectively. 54
11. Description of UMRF JSON notation attributes. 56
12. TeMoto Task Management conformance to executive layer requirements (Table 4) 64
13. Overview of the resource managers implemented in the TeMoto framework. 74
14. TeMoto Resource Management conformance to functional layer requirements(Table 5). 75
15. Overview of the fault-tolerant sensor redundancy demo. 78
16. Overview of the multi-robot escort via resource redundancy demo. 79
17. Overview of the fault-tolerant sensor redundancy demo. 80
18. Overview of the Large Language Models (LLM)-driven task planning demo. 82
19. Overview of the multi-robot knowledge sharing demo. 86

LIST OF ABBREVIATIONS

Acronyms

API Application Programming Interface. 43, 70
ATRA Autonomy and Technology Readiness Assessment. 22

BiP Behavior-Interaction-Priority. 32
BT Behavior Tree. 30, 31

CFG Control Flow Graph. 30

D&D Decontamination and Decommissioning. 17, 22
DDS Data Distribution Service. 31
DSL Domain Specific Language. 19, 36, 40, 41, 55, 64, 65

EVA Extravehicular Activity. 22

FSM Finite State Machine. 30

GOAP Goal-Oriented Action Planning. 28
GPT Generative Pre-trained Transformer. 28
GUI Graphical User Interface. 40, 41

HMI Human-Machine Interaction. 38
HRC Human-Robot Collaboration. 26
HRI Human-Robot Interaction. 19, 24, 26, 32, 38, 42, 43, 55, 65, 86, 88, 90

IPC Inter-Process Communication. 31

JSON JavaScript Object Notation. 19, 40, 41, 55, 57, 64, 65, 80, 87

LIDAR Light Detection And Ranging. 20, 66, 77, 79
LLM Large Language Models. 12, 28, 80, 82, 86, 87
LoA Level of Autonomy. 17, 22, 24, 26, 38

MAC Multi-Agent Collaboration. 24–26, 32, 38, 42
MBE Model-Based Engineering. 32
MQTT Message Queuing Telemetry Transport. 31
MRC Multi-Robot Collaboration. 26, 55, 65

OCS Operator Control Station. 77–80
OS Operating System. 31, 42, 75, 76

PDDL Planning Domain Definition Language. 28
PNP Petri Net Plans. 30
Pub-Sub publisher-subscriber. 31, 32

RAII Resource Acquisition Is Initialization. 32
RIS Resource Info Synchronizer. 70, 71, 75
ROS Robot Operating System. 19–21, 31, 32, 39, 41, 42, 70, 74, 80, 87, 90
RPC Remote Procedure Call. 31
RR Resource Registrar. 70, 75, 76, 87, 88

SAR Search and Rescue. 25
SLAM Simultaneous Localization and Mapping. 79
SPA Sense-Plan-Act. 27
STRIPS Stanford Research Institute Problem Solver. 28

TAE TeMoto Action Engine. 57
TRL Technology Readiness Level. 22–24

UML Unified Modeling Language. 32
UMRF Unified Meaning Representation Format. 19, 50, 51, 55, 57, 58, 61, 64, 65, 78–80, 82, 86–88, 90
UX User Experience. 19, 20, 90

XML Extensible Markup Language. 39, 40

LIST OF ORIGINAL PUBLICATIONS

Publications included in the thesis

I **R. Valner**, K. Kruusamäe, M. Pryor (2018). TeMoto: Intuitive Multi-Range Telerobotic System with Natural Gestural and Verbal Instruction Interface. Robotics, 7 (1), 9. DOI: 10.3390/robotics7010009.

II **R. Valner**, V. Vunder, A. Aabloo, M. Pryor, K. Kruusamäe (2022). TeMoto: A Software Framework for Adaptive and Dependable Robotic Autonomy With Dynamic Resource Management. IEEE Access, 10, 5188951907. DOI: 10.1109/ACCESS.2022.3173647.

III **R. Valner**, S. Wanna, K. Kruusamäe, M. Pryor (2022). Unified Meaning Representation Format (UMRF) - A Task Description and Execution Formalism for HRI. ACM Transactions on Human-Robot Interaction, 11 (4), 38. DOI: 10.1145/3522580.

Author's Contribution

In Publications II and III, the author was responsible for the majority of research in all phases. Co-authors contributed to the software design and development, technical discussion, and manuscript writing. The author's contributions per publication were the following:

I Contributed to the developed work by outlining and implementing an algorithm for navigating mobile manipulators. Lead writer of the manuscript.

II Design and implementation of the TeMoto framework in C++. Conducted all demonstrations and data analysis. Lead writer of the manuscript.

III Design and implementation of TeMoto Action Engine and Action Assistant in C++. Conducted all demonstrations and data analysis. Lead writer of the manuscript.

Publications not included in the thesis

1. S. Wanna, F. Parra, **R. Valner**, K. Kruusamäe, M. Pryor (2024). Unlocking underrepresented use-cases for large language model-driven human-robot task planning. Advanced Robotics, 38(18), 1335–1348. DOI: 10.1080/01691864.2024.2366974
2. **R. Valner**, S. Wanna, K. Kruusamäe, M. Pryor (2021). "TeMoto: A Software Framework Supporting Mixed-Modality Command Inputs Necessary for Integration of nonverbal-HRI," workshop on Exploring Applications for Autonomous Non-Verbal Human-Robot Interactions, in ACM/IEEE International Conference on Human-Robot Interaction 2021, Virtual Conference, March 8th.

3. **R. Valner**, V. Vunder, A. Zelenak, M. Pryor, A. Aabloo, K. Kruusamäe (2018). "Intuitive ‘human-on-the-loop’interface for tele-operating remote mobile manipulator robots," International symposium on artificial intelligence, robotics, and automation in space (i-SAIRAS).
4. **R. Valner**, V. Vunder, A. Zelenak, K. Kruusamäe, M. Pryor (2018). "TeMoto 2.0: Source Agnostic Command-to-Task Architecture Enabling Increased Autonomy in Remote Systems," Proceedings of Waste Management Symposia 2018 (WM2018).

Other published work of the author

1. **R. Valner**, H. Masnavi, I. Rybalskii, R. Põlluäär, E. Kõiv, A. Aabloo, K. Kruusamäe, A. Singh (2022). Scalable and heterogenous mobile robot fleet-based task automation in crowded hospital environments—a field test. Frontiers in Robotics and AI, 9, 922835. DOI: 10.3389/frobt.2022.922835.
2. **R. Valner**, J. Dydynski, S. Cho, K. Kruusamäe (2021). Communication of Hazards in Mixed-Reality Telerobotic Systems: The Usage of Naturalistic Avoidance Cues in Driving Tasks. Human Factors: The Journal of the Human Factors and Ergonomics Society, 63 (4), 619634. DOI: 10.1177/0018720820902293.
3. V. Vunder, **R. Valner**, C. McMahon, K. Kruusamäe, M. Pryor (2018). Improved Situational Awareness in ROS Using Panospheric Vision and Virtual Reality. 2018 11th International Conference on Human System Interaction (HSI): 11th International Conference on Human System Interaction (HSI), Gdansk, Poland, 4-6 July 2018. IEEE, 471477. DOI: 10.1109/HSI.2018.8431062.

1. INTRODUCTION

Reducing human involvement in hazardous, stressful, and tedious tasks has been among the main driving reasons for reaching self-sufficient, autonomous robots. Today, robotics and automation are practically irreplaceable in high-volume mass production, and widespread for simple household tasks. But when comparing the most prevalent real-world commercial use cases for robots, there is a clear tradeoff between risk (to humans or expensive equipment), Level of Autonomy (LoA), and environmental complexity (Fig. 1). In tasks, such as warehouse logistics, the environment is fully controlled and predictable (often secluded from human workers), thus the robots can be fully autonomous, despite the potentially high monetary value of the carried payload, i.e., high risk. However, with a combined increase of risk and environmental complexity, evident in applications such as bomb disposal or nuclear Decontamination and Decommissioning (D&D), the control of the robot is often trusted only to the hands of a human operator. Teleoperation, however, is not always an option, evidenced by robots abandoned in Daiichi nuclear power plant due to communication loss [1], or NASA's Mars rover missions, where communication latency can reach up to twenty minutes [2].

The increase of LoA of robots deployed for high-risk and high-complexity tasks is challenged by various reasons originating from, e.g., perception, control, hardware design, communication, and decision-making. Among such design challenges, system integration via software is a significant contributing factor, since it has the unique property of being the backbone that ties all the aforementioned domains together. Limitations in this backbone ultimately translate to the limitations of the whole system [3]. For example, if the system were designed for specific tasks, e.g., autonomous food delivery, it would be difficult to scale the system for a different domain of tasks, regardless of the efforts in hardware, perception, or communication. Even within a specific task domain, the tasks assigned to the robot can be diverse and thus require the software architecture to scale. This motivates the study of robotic software architectures [4].

Application domains, such as disaster response, space exploration, or even flexible manufacturing and healthcare have many aspects in common, that is, a fully autonomous robot deployed in these domains has to be:

- **Adaptive** - While deployed, the robot can adapt to dynamic changes in the environment, mission requirements, and in itself (hardware/software failures or upgrades).
- **Scalable** - The robot's base software architecture can be reused, extended, and maintained.
- **Multi-Agent configurable** - The robot's base software architecture can be applied for heterogeneous multi-agent topologies, including Human-Robot Collaboration (HRC), Multi-Robot Collaboration (MRC), and combinations of both.

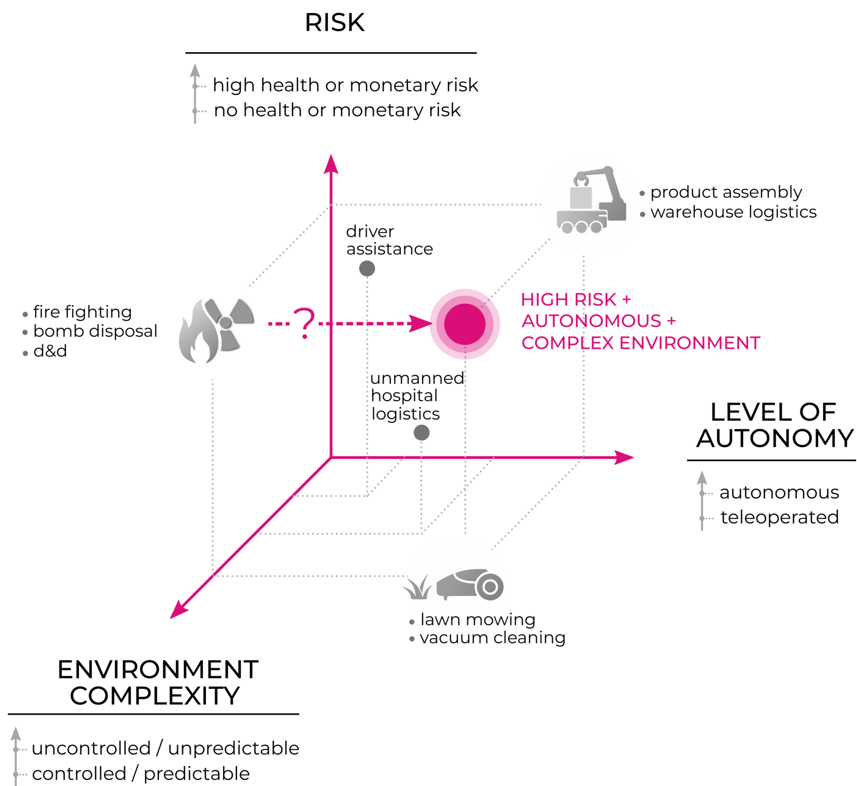


Figure 1. State-of-the-art real-world deployment of robots compared in the scale of risk, LoA, and environment complexity. This work addresses the technological gap in improving robot autonomy in high-risk, complex environments.

The effort undergone in researching robotic autonomy architectures is indicated by the emergence of task management tools, such as SMACH[5], BehaviorTree.Cpp[6], RAFCON[7], TaskForce[8]; component-based system modeling and runtime reconfiguration tools, such as Rorg[9], MROS[10], Dyknow[11], or Dr-Bip[12]; multi-robot task management frameworks, e.g., Open-RMF[13]; User Experience (UX) and Human-Robot Interaction (HRI) guidelines[14], [15]; and robotic application development frameworks, such as Robot Operating System (ROS)[16] or YARP[17].

Despite the numerous advances and tools available, developing a universally adaptive, multi-agent configurable, and scalable software architecture for robotic autonomy remains an ongoing challenge in the robotics community [3], [18].

1.1. Research Questions

This work explores the following research questions:

RQ1 : What are the key software requirements for developing autonomous robots deployed in high-risk and high-complexity task domains?

RQ2 : Given the diversity of robotic tasks, hardware and software components, and team configuration, which software architecture design adheres to **RQ1**?

1.2. Contributions

Addressing **RQ1** and **RQ2**, this thesis covers the research and development of TeMoto, a software architecture for adaptive autonomous robots, and a ROS-based framework of openly available software tools that implement the TeMoto architecture. TeMoto aims to accelerate the development of reliable and flexible robotic applications by providing publicly available software tools for robotic application developers. The main contributions of this work include:

- Analysis of desired autonomous capabilities in high-risk application domains and identification of potential pain points in practical implementations (Section 2.1, 2.2 and I).
- Analysis of robotic architectures (Section 2.3) and related work including code review (Section 4 and II).
- Development of requirements for a scalable and dynamically adaptive layered software architecture (Section 3).
- Design of TeMoto, a layered task and resource management based architecture (Section 5, II, III).
- Design of Unified Meaning Representation Format (UMRF), a task domain independent and JavaScript Object Notation (JSON) based Domain Spe-

cific Language (DSL) for defining complex robotic tasks (Section 6, THRI paper).
- Open source implementation of the TeMoto architecture, which facilitates the design of adaptive single and multi robot applications II.

1.2.1. Case Study on Previous Work

Project TeMoto[I] was initially designed to validate the feasibility of using gestures and speech, and carefully designed visual feedback via a computer screen, to intuitively communicate commands to mobile robots and manipulators in hazardous environments. The findings supported the benefits of our UX design approach, but the underlying implementation, which combined commonly available ROS packages, posed restrictions on:

- Scalability of interaction modalities beyond gestures and speech,
- Scalability of semi-autonomously performed operations, i.e., tasks,
- Scalability to multiple collaborating robots,
- Ability to recover from component failures,
- Ability to control the elements of visual feedback during runtime,

The implementation of TeMoto at the time was tightly coupled to the specific interaction modalities, where gestures and voice commands were directly mapped to navigation, manipulation, or interface control instructions. Also, the operator's feedback assumed the presence of a specific set of sensors, e.g., a 2D Light Detection And Ranging (LIDAR) and 3D camera, while the operator's gestures were captured via the Leap Motion Controller. Thus, adding new features and maintaining the system became increasingly complex — a complete redesign was necessary.

1.2.2. Technical Summary

The key design features that TeMoto emphasizes the most include:

- **Dynamic task management**: TeMoto separates mission strategy-related code from functional resources, such as sensors and actuators. A task consists of modular actions that implement a specific behavior, e.g., a sensing or a navigation action. Actions can contain any arbitrary user-defined code and can start, stop, and access resources. Actions can be connected into sequential, concurrent, and cyclical graphs, thus allowing users to implement arbitrarily complex run-time behaviors.
- **Dynamic resource management**: Existing components (ROS nodes or other executables), i.e., resources such as a camera, lidar, network, or CPU-intensive algorithm can be programmatically started, stopped, and monitored for failures, which are reported to all resource consumers. TeMoto provides an accounting mechanism for resources (the Resource Registrar or RR), embedded into all TeMoto subsystems, which mediates resource

queries. Thus, the RR knows how many consumers a resource has and what are its sub-resource dependencies, which is important when propagating resource failure messages to all consumers or allocating (mitigating multiple allocations) and de-allocating (making sure that there are no consumers left) resources.

- **Minimal development overhead**: TeMoto does not require any resource customization, i.e., existing ROS packages can be used via TeMoto without modification. Similar frameworks assume specific behavior from resources to be used within the framework. Having minimal development overhead is important for smooth adoption and maintenance because the users do not need to fully commit their project to a specific framework and its invasive requirements.
- **Modular design**: While TeMoto contains several subsystems (ROS nodes), each subsystem (maintained in separate repositories) has minimal dependencies. This allows the robotics community to adopt only the subsystems that matter for their project and keeps code bloat at a minimum.

The feasibility of TeMoto is demonstrated both via technical demonstrators and qualitatively by asserting related work against the requirements derived from the most prevalent application domains for autonomous robots. Thus in addition to the implemented and tested tools, a significant contribution of this work is establishing the requirements and conceptual architecture for pursuing a robotic autonomy framework.

1.3. Thesis Outline

The rest of this work is segregated into seven main sections. Section 2, addressing **RQ1**, analyzes the desired autonomous capabilities of robots deployed in challenging environments and covers common approaches for a robotic autonomy software stack. Section 3, addressing **RQ1**, establishes the requirements which set the baseline for comparing related work, and for deriving and implementing an architecture. Next, Section 4, addressing **RQ2**, reviews the relevant literature to contextualize this work against existing methodologies. Section 5, addressing **RQ2**, proposes and outlines the TeMoto architecture, where Section 6 covers task management and Section 7 covers the resource management aspect of TeMoto. The proposed architecture is evaluated in Section 8, followed by concluding remarks in Section 10.

2. PRELIMINARIES

2.1. Application Domain Review

This section overviews real-world applications of state-of-the-art robots deployed for high-risk and high-complexity tasks. Table 1 shows a non-exhaustive list of such task domains and tasks, where robots are seen as a necessity, rather than a convenience. Each task is evaluated based on reviewed literature and assesses how mature current solutions are, i.e., Technology Readiness Level (TRL), and how autonomous are the current solutions, i.e., LoA. The TRL ranges from 1 (basic principles observed and reported) to 9 (actual system "flight proven" through successful mission operations) with respect to ESA's TRL definitions [19] (Table 2). The LoA ranges from 0 (remote control) to 10 (fully autonomous) with respect to the Autonomy and Technology Readiness Assessment (ATRA)[20] scale (Table 2). Note: while the ATRA scale has been designed for unmanned aircraft systems, its non-human-centric approach (compared to Sheridan's LoA scale[21]) is preferable for assessing a broader spectrum of autonomous capabilities.

Most robots applied for D&D and emergency response have been teleoperated (LoA $\leq$ 2), while the TRL is high (TRL $\geq$ 7) since high-risk/high-complexity tasks require dependable and proven technologies, further indicating the technical immaturity of autonomous systems in this domain. Healthcare on average has a higher LoA (LoA $\leq$ 4) than the previous domain while the TRL is varied. The higher LoA could be related to a more deterministic environment, where the robots traverse along well-defined but potentially crowded buildings. The manufacturing domain likely has the highest commercial use of robots compared to other domains, indirectly indicated by a very high TRL. However, as mentioned in previous sections, the environment is also very structured and predictable. Thus, tasks, such as warehouse logistics, have a relatively high LoA (LoA $\leq$ 4) despite the potentially high monetary risk. Finally, space exploration is undoubtedly a high-risk task domain concerning health hazards and the loss of expensive equipment. While robots deployed for Extravehicular Activity (EVA) are mostly teleoperated (LoA $\leq$ 1), robots, such as NASA's Mars exploration rovers[22], possess LoA up to 5 with TRL up to 9. Yet, the rover missions are carefully and redundantly planned [23], which may not be feasible or even possible in rapidly changing environments, e.g., fire fighting. Conclusively, increasing the autonomy of robots deployed for high-risk tasks in complicated environments is an open challenge.

Table 1. Evaluation of TRL and LoA of robots deployed for tasks with high-risk and high-complexity.

Domain	Main tasks	Estimated LOA (max 10)	Estimated TRL (max 9)	References
D&D and Emergency Response	*Decontamination*	0-1	7-8	[24], [25]
	Debris removal	0-1	7-8	[24], [25]
	Surveillance	0-2	7-9	[25]–[27]
	Firefighting	0-1	7-8	[26]–[28]
Healthcare	*Delivery*	4	9	[29], [30]
	Delivery & manipulation	4	4	[29]–[31]
	Social interaction	4	6-7	[29]–[31]
Manufacturing	*Warehouse logistics*	4	9	[32], [33]
	Product assembly	1	9	[32]–[36]
Space Exploration	*Remote exploration*	5	8-9	[37]–[39]
	Extravehicular activities (EVA)	0-1	7-8	[37], [38]
	Intra Vehicular activities (IVA)	7	3	[37], [38]

2.2. Desired Autonomous Capabilities

With the technological gap, introduced in Section 2.1, in mind, this section elaborates on the autonomous capabilities desired for reliable and flexible autonomous operations in high-risk/high-complexity task domains. While the specific list of capabilities may depend on task-specific requirements, the literature commonly suggests the following as desired:

- **Adaptability** [24]–[26], [29], [30], [32]–[34], [37], [38] – Task specifications and requirements may evolve both in the long term and during deployment. Furthermore, as software and hardware technologies advance, the system should facilitate modifications or adaptations without necessitating a complete redesign.
- **Dependability** [24], [26], [29], [30], [32], [34], [37], [38] – Given that both hardware and software components are prone to malfunctions, the architecture should inherently include reactive behaviors that enable recovery from failures.
- **Scalability** [4], [25], [30], [33], [37], [38], [40], [41] – It is essential to design the system in a manner that allows for the expansion of task capabilities

Table 2. TRL and ATRA LoA level descriptions. Acronyms: Real Time (RT), External System Independence (ESI). A detailed description of the ATRA LoA scale can be found in [20].

	Level descriptor	
Level	**LoA**	**TRL**
10	Fully autonomous	—
9	Swarm cognizance and group decision-making	Flight proven
8	Situational awareness and cognizance	Flight qualified
7	RT collaborative mission planning	Performance in operational environment
6	Dynamic mission planning	System critical functions in relevant environment
5	RT cooperative navigation and path planning	Component critical functions in relevant environment
4	RT obstacle/event detection and path planning	Component functional verification
3	Fault/event adaptive autonomous system	Experimental proof of function
2	ESI navigation (e.g., non-GPS)	Preliminary concept of application
1	Automatic flight control	Basic principles observed
0	Remote control	—

and integration of additional software or hardware components, as well as adaptability to various application domains without fundamental changes.

- **HRI oriented design** [26]–[28], [31], [32], [34], [35], [37], [38] – Considering many use cases involve supervision or teleoperation by humans, the system should accommodate varying LoA and interaction modalities.
- **Multi-Agent Collaboration (MAC) oriented design** [27], [28], [32], [36]–[39] – In environments where robots operate alongside other robots and humans, the design should support synchronized task execution and the development of a shared world representation.

The following subsections provide a brief insight into each capability and establish modular, component-independent, and dynamically reconfigurable design as necessary design principles that help narrow down the potential implementation.

2.2.1. Adaptability and Dependability

Adaptability and autonomy are often used interchangeably [42]. Still, for this work, an adaptive system, or self-adaptive software is regarded as a system that modifies its behavior in response to environmental or internal changes [42]. This also includes changes in mission requirements [43], or adaptation to new hardware/software configuration in response to faults or functional upgrades [26]. Adaptability is, therefore, an abstract system property, which is often associated

with modularity [24], [26], [30], [43], dynamic reconfigurability [26], [37], [44], component independent design, as well as scalability [24], [30], [43] — thus a set of properties necessary for implementing an adaptive robotic system.

Dependability can be viewed as a subset of adaptive behaviors [42], defined as a system's ability to avoid service failures [45]. Dependability comprises four categories: a fault's prevention, removal, forecasting, and tolerance [46], [47]. Fault prevention, removal, and forecasting involve static analysis of a system's code, heavily related to model-based engineering, and redundant hardware design. However, fault tolerance characterizes the system's ability to cope with errors while operating. For example, an alternative sensor, such as a depth camera, can be used during autonomous navigation when a primary sensor, e.g., 2D lidar, stops working. Hence, fault tolerance requires the system to be operationally adaptive and more precisely — dynamically reconfigurable [4], [48].

2.2.2. Scalability

Scalability is commonly defined as the system's ability to perform a service with increased demand [49], such as increased production in assembly lines, increased queries to a web server, or increased amount of processable data. However, in the domain of robotics, scalability is often associated with reusability (scaling for different application domains) [37], [40], extensibility (scaling for different system components and behaviors) [37], [40], and maintainability (system's ability to remain maintainable with increased scale) [4], [37]. With that in mind, modular [40], component independent [40], and dynamically reconfigurable [4], [33] designs are often referred to as critical architectural aspects that lead to scalable robotic software systems. In this work, scalability refers to the aforementioned extended definition.

2.2.3. Multi-Agent Collaboration Oriented Design

Tasks, such as product assembly and Search and Rescue (SAR), including transportation, remote manipulation, and information gathering, can potentially benefit from human-robot or multi-robot teams [27], [50]. The configuration of the team performing the task can be [36]:

- **Separate** - team members perform tasks individually and independently.
- **Sequential** - team members perform individual tasks in a sequential process flow, e.g., product assembly.
- **Shared** - team members perform individual tasks in a shared workspace.
- **Supportive** - team members depend on each other and supportively work on the same task.

Thus, given the shared nature of tasks, both sequential and concurrent, MAC systems require techniques for defining [7], [51] and sharing [50], [52] the tasks and knowledge (objects of interest, maps, etc.) [39], [53] related to the underlying tasks. However, how such information is communicated between collaborat-

ing team members is fundamentally different. Thus, Human-Robot Collaboration (HRC) systems often cover aspects such as interaction modalities [37] (speech, gestures, etc.) and LoA (teleoperation, supervision, etc.) [26]. In contrast, Multi-Robot Collaboration (MRC) systems cover inter-robot communication [13], [52] and knowledge synchronization [39] techniques. Decentralized implementation of MAC systems is further encouraged to reduce single points of failure and facilitate the addition, substitution, and removal of robots [52], [54].

Task management is therefore a fundamental necessity for autonomous robots in MAC teams. However, the set of potential tasks can be diverse [29], [31], [36], evolve during the mission [26], [36], and be applied to different hardware platforms [36], [38]. Thus, a task management system must be modular and component-independent to support different types of tasks, and dynamically reconfigurable for starting, stopping, and modifying the tasks during deployment.

2.2.4. Human-Robot Interaction-Oriented Design

Most tasks outlined in Table 1 currently assume either supervisory or direct control by the human operator. Thus, HRI remains an essential factor when designing autonomous systems. HRI research is a diverse domain, involving psychology, ergonomics, knowledge representation, kinematics, etc. [55]. However, the choice of specific interaction modalities and LoA are aspects of every HRI system, and both can be considered dynamic throughout the task. From the perspective of dynamic LoA, Wang et al. argue: "*Work instructions need to be adaptive to not only the changing competence level of individual workers but also to declining focus and concentration during the day or within the week*" [56]. Fusing different interaction modalities is a potential method for improved human command recognition rate and programming-free robot control [56]. Task frame formalism, where frames describe primitive behaviors, can be utilized to ground multimodal input to robotic actions [56], [57].

Conclusively, to facilitate the continuous development of HRI systems, it is beneficial to have tools that enable quick adjustments and flexibility for the human interfaces when system requirements change due to updates in the state-of-the-art, application domain, etc. Thus, modularity is a crucial design principle that promotes software reuse, scalability, and reduces development effort. Hence, a robot's autonomous capabilities should not depend on the command interface and should be decoupled via a standard format with descriptive capabilities for outlining tasks and a sensible syntax for HRI [II].

2.3. Architectures for Autonomous Robots

Combining the desired autonomous capabilities requires careful thought and consideration (Figure 2), as early architectural decisions often persist for years [58]. Every software application has an architecture, whether deliberately designed or not [59]. This architecture determines the set of properties and features that are

realizable without a significant redesign of the application [59]. Thus, software architecture should be among the top priorities when designing an autonomous system, as architectural changes can delay progress due to significant code implementations [58].

Robotic architectures can be coarsely described via the Sense-Plan-Act (SPA) [58], [60] model (Figure 3a), where sensing represents feedback from the environment and robot's internal state, planning represents the process of finding the optimal sequence of actions for fulfilling a task, and acting represents the capability of manipulating the environment.

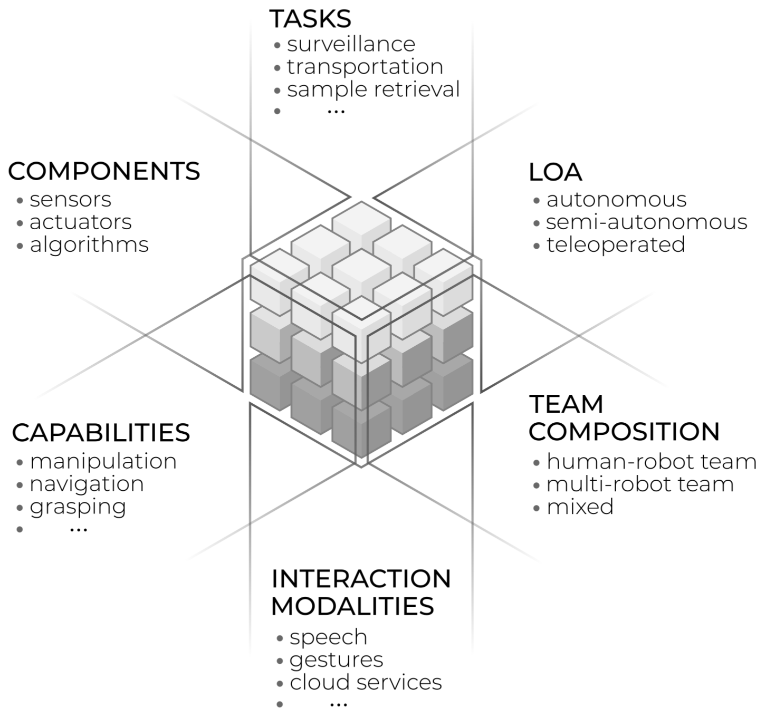


Figure 2. Design challenges for an adaptive, modular, and scalable software architecture for complex autonomous robots.

SPA as an architecture was proposed in the late 60's [61] and followed a linear sequence of sensing, then planning, and finally acting in a continuous loop. While planning gave the SPA model deliberative properties, its sequential execution order made it unreactive to local deviations from the plan, such as dynamic obstacles [58]. The solution was to leave out the planning step and combine sense and act (Figure 3b), leading to reactive architectures (Figure 3c) [58]. A specific controller implementation that utilizes sensing and acting is called a behavior, such as following a human or end-effector servoing. A notable example of a reactive architecture is the subsumption architecture [62], where a collection of such behaviors are invoked on specific environmental conditions and can override

each other based on pre-assigned priority. Yet, reactive architectures were not optimal for complex tasks requiring more advanced behavior arbitration schemes, i.e., planning [58]. This led to the development of layered architectures (Figure 3d), which combine planning (deliberation layer) and reactive (behavioral layer) components, with an executive layer in between, mapping the intended behavior sequences from the deliberation layer to specific behavior implementations. However, many layered architecture implementations [63], [64] combine the executive and behavioral layers into a single executive layer. Additionally, a functional layer is added, which implements the communication with sensing, actuation, and algorithmic (e.g., localization) components.

Layered architectures have become increasingly popular due to their flexibility and ability to simultaneously operate at multiple levels of abstraction [58]. Likewise, layered architectures can be scaled into decentralized multi-robot architectures, e.g., syndicate architecture (Figure 4), by adding coordination and synchronization between the layers [58], [65]. Thus tasks can be collaboratively planned and synchronously executed. This work uses the syndicate architecture as a model to derive the TeMoto architecture. The following subsections provide an overview of common techniques used to implement the deliberative, executive, and functional layers.

2.3.1. Deliberative Layer

Automated planning and scheduling, i.e., deliberation, is the process of synthesizing (planning) a sequence of actions to fulfill a particular task based on the current understanding of the environment [66]. Predicate logic is often applied to formally describe the task domain, the state of the environment, and actions that transform the states [67]. For example a simple environment domain containing a “switch” and a “light” can be manipulated with an action that changes the “switch” from “off” to “on” state. Hence, planning is a process of finding an optimal sequence of actions that transform the environment from starting to the desired state. Planning domain modeling languages, such as Planning Domain Definition Language (PDDL), Stanford Research Institute Problem Solver (STRIPS), and Goal-Oriented Action Planning (GOAP), describe planning domains and specific planning problems.

On the other hand, LLM, such as BERT, T5, and the Generative Pre-trained Transformer (GPT) family of LLMs, offer an appealing alternative to classical planners. Instead of modeling the task domain in minute detail (e.g., via PDDL), the common sense knowledge embedded into glsllms can be leveraged [68]. Replanning a failed task can be achieved by reconditioning the initial prompt with precondition errors [69]. LLM’s context awareness can be increased by fusing multiple modalities of information, such as speech and camera data [70].

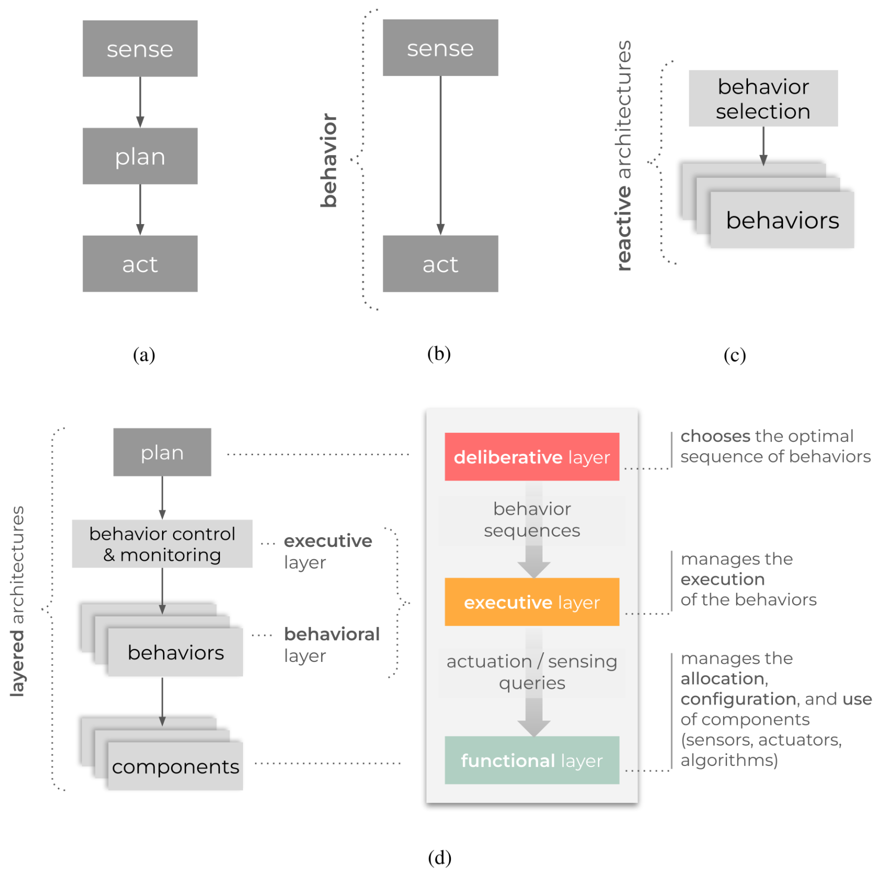


Figure 3. Different architectures and architectural components depicted via *sense-plan-act* model. The original sequentially executed SPA model (a). For increased reactivity, the planning step is omitted (b), leading to reactive architectures (c). While reactive architectures responded well to sudden environmental changes, the lack of planning limited their use in lengthy and complex tasks, leading to layered architectures (d) where each concurrently operating layer controls the one below. Often the *behavioral* and *executive* layers are combined into a single *executive* layer.

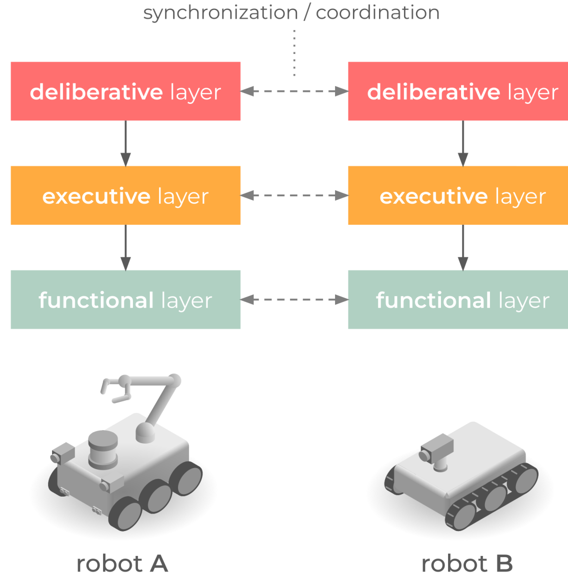


Figure 4. Syndicate architecture scales the layered architecture into a decentralized multi-robot system.

2.3.2. Executive Layer

Once the deliberative layer has provided a valid sequence of actions, the executive layer is responsible for grounding the sequence into executable behaviors. The most common task model in robotics is a Finite State Machine (FSM), Behavior Tree (BT), and s domain-specific Control Flow Graph (CFG) such as Petri Net Plans (PNP). FSMs outline the task as a graph of states or behaviors and possible transitions between them. Note that many popular FSM implementations [5], [7], [71], [72] in robotics refer to *behavioral* FSM's [73] (FSM nodes represent behaviors of the system), as opposed to more traditional *protocol* FSM's[73] (FSM nodes represent the state of the system). BT's are similar to FSM's but the whole BT is periodically evaluated via ticking semantics, leading to a more intuitive graphical representation of tasks compared to FSM's [74]. CFGs represent the flow of data and the sequence of procedures, often used in generic parallel programming frameworks such as Intel TBB[75] or Taskflow[76].

Figure 5 shows graphical representations of the instruction “Find the object and put it on the table” in the flowchart (Figure 5a), FSM (Figure 5b), PNP (Figure 5c), and BT (Figure 5d) notations. The graphs are semantically equivalent and designed to be as reactive as possible, i.e., conditions, such as “object lost?”, are continuously re-evaluated to adapt to unexpected changes, such as accidentally dropping the object. The FSM notation subjectively may be the easiest to read but without standard notation for conditional statements or interrupts, it is assumed such reactive logic is implemented inside each state. This makes the states inside the FSM very coupled and not modular. PNP models the task via places and tran-

sitions. Each place that implements an action (e.g., “Find,” “Move”) is preceded with an initialization place and a transition “s” that starts the action. Similarly, an action is followed by an end transition “e”. Interrupts are modeled via special transitions “i”, that can preempt an action and redirect the execution flow. The actions are decoupled from the task’s logic and can be implemented modularly. Finally, BT also offers a modular and reactive representation. BT differs fundamentally from other representations, as the graph does not follow the typical execution flow pattern. The graph is modeled via fallback (“?” in Figure 5d), sequence (“→” in Figure 5d), and other nodes (not shown in Figure 5d, more details in [74]). BT is periodically re-evaluated, allowing conditions, such as “Found?” or “Reached?”, to preempt any action that came later in a sequence automatically. Thus, if the object is lost from the gripper, the “Found?” condition evaluates to “false” making the BT execute the “Find object” action.

2.3.3. Functional Layer

The functional layer provides access to specific hardware and software components through a standardized communication interface, i.e., a component is modularly encapsulated with only an abstract interface exposed to the user [58]. Thus, the user does not need to know about the component-specific data format, hardware communication protocol (USB, Ethernet, CAN, etc.), or if it is hierarchically composed of other components. This makes the higher architectural layers independent from any particular component implementation, leading to a reusable and scalable system [40], [77], [78]. Distributed software design, where modules are separate Operating System (OS) processes, is often used to further increase the modularity and scalability of the application. As distributed modules can be located on separate physical devices (e.g., a robot and a teleoperation interface), networking protocols, such as TCP/IP, are used for the underlying Inter-Process Communication (IPC). Program design patterns, such as publisher-subscriber (Pub-Sub) or Remote Procedure Call (RPC) are often utilized as standard data exchange semantics, allowing the developer to focus less on low-level intricacies of IPC, such as state of the communication channel, peer discovery, and data serialization. Widely adopted developer tools (e.g., ROS, YARP[17]) and standards for Pub-Sub implementation (e.g., Data Distribution Service (DDS), Message Queuing Telemetry Transport (MQTT)) further facilitate the development of a modular functional layer.

While the Pub-Sub and RPC patterns help decoupling the interface from the implementation, the underlying resource that provides a service or a data stream (i.e., sensor, actuator, algorithm) has a lifecycle. At the very least, this lifecycle includes initialization and un-initialization through which the resource is available. Assuring availability for resource consumers requires the resource to be managed. The naive approach binds a resource’s lifecycle to the whole application’s lifecycle, i.e., all resources are started and stopped simultaneously via a

script (e.g., ROS launch files). However, the drawback of the naive approach is its inability to adapt or change during deployment in case of resource failures, upgrades, or simply energy conservation. On the other hand, if the lifecycle of a resource is dynamically managed, the resource must be available until there are no consumers. A resource should be initialized when first acquired and automatically un-initialized when no longer used to prevent resource leaks. This is also known as the Resource Acquisition Is Initialization (RAII) design pattern. Reference counting techniques, e.g., smart pointers in C++, become paramount in multi-consumer use cases that the Pub-Sub pattern inherently leads to. Yet, reference counting techniques are rarely mentioned in the works relating to functional layer design.

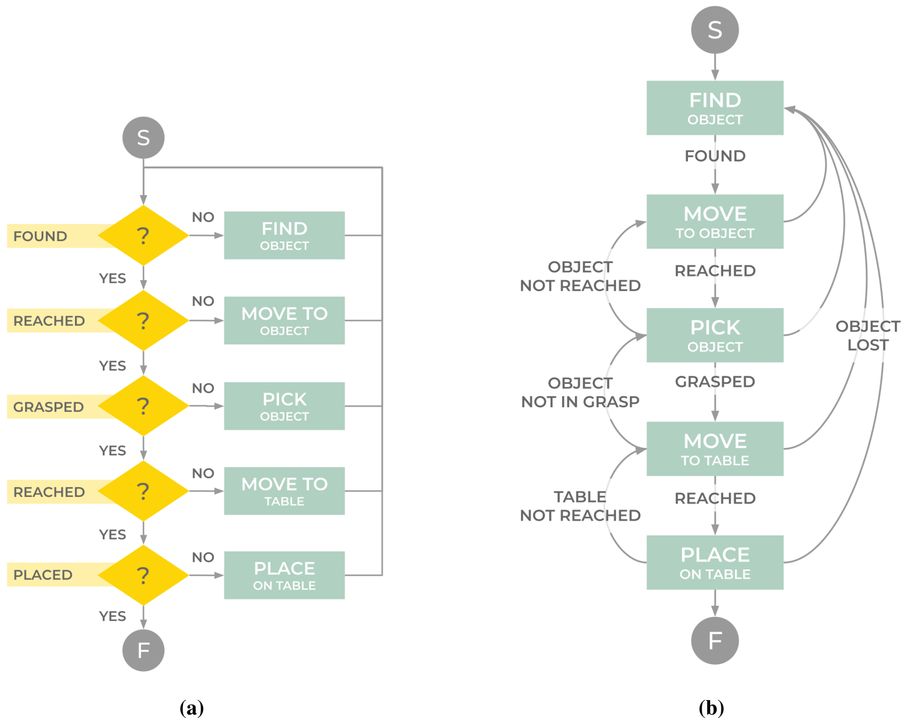
Finally, to minimize the risk of unexpected behavior during dynamic resource lifecycle transitions, Model-Based Engineering (MBE) techniques are used. Resources and their properties are modeled, utilizing languages such as Unified Modeling Language (UML) or Behavior-Interaction-Priority (BiP), and applied to dynamic resource transition management frameworks, such as DR-BiP[12] or Metacontrol[10], allowing the developers to assess the system's behavior before deployment.

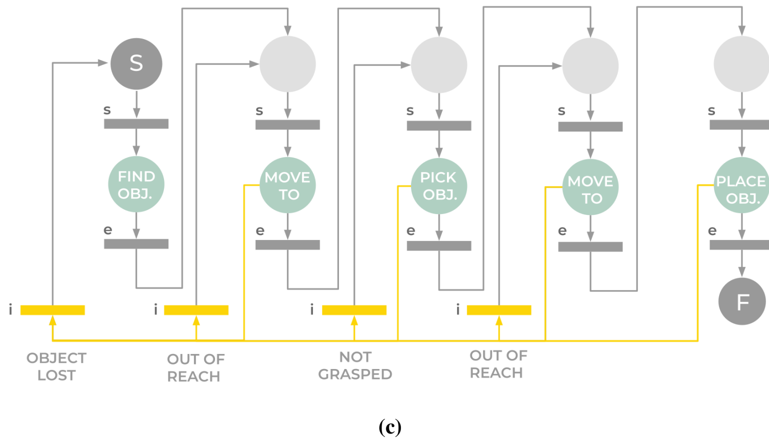
2.4. Summary

This section reviewed the real-world application domains of robots deployed for high-risk and high-complexity tasks, as well as desired autonomous capabilities and software architectures of autonomous robots. To conclude:

- The robots deployed in high risk and high complexity environments are mostly teleoperated, thus requiring stable connectivity with the robot and a trained operator or a team of operators to control it. Yet this may not be feasible in areas that are simultaneously remote, hazardous and hard to access, which necessitates increased autonomy.
- Adapatability, dependability, scalability, HRI-oriented and MAC-oriented design are seen as attributes that further facilitate robotic autonomy.
- Three layer software architectures streamline the design of autonomous behaviors by combining deliberation (deliberation layer), task management (executive layer) and robotic resource management (functional layer).

The next section establishes the requirements for the implementation of a three layer architecture and Section 4 reviews the related work with respect to the established requirements.





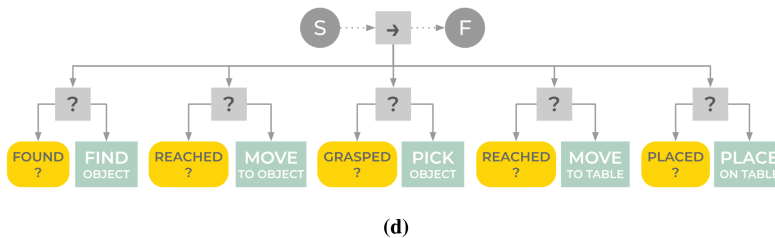


Figure 5. Example of a reactive implementation of the task “Find the object and put it on the table” represented as flowchart (a) with interrupts, Finite State Machine (b), Petri-Net Plan (c), Behavior Tree (d).

3. REQUIREMENTS DEVELOPMENT

The requirements form the guidelines for the design of this work and allow for the comparison of existing work on common criteria. This section maps the desired capabilities for an autonomous robot (Section 2.2) to requirements specific to the multi-robot layered architectures, i.e., syndicate architectures (Section 2.3). The development of requirements is segregated into common, executive layer, and functional layer sections.

This work aims to define a backbone of deterministic software functionalities that allow for modular, scalable, reusable, and dynamic composition of the robot's software stack. Analysis and development of these core functionalities is the main focus of this work (**RQ1**), and the policies for system composition (both during design and deployment) are up for the user to decide. Thus, while this work is structurally based on layered design, more precisely on 3L design, deliberation, and policies for dynamic component substitution are not discussed as these are domain and task-specific topics, which leads to the following base assumptions of this work:

- A1 **Prioritization on scalability**. The developed architecture must remain task and domain-independent, facilitating the execution, not the design of dynamic behaviors. Policies for dynamic system composition (task definition, i.e., deliberative layer, and component composition) are defined by the user who utilizes the architecture to carry out the tasks.
- A2 **Communication hardware traits are not modeled**: Networking topologies and respective challenges, e.g., limited bandwidth or communication drops, are not explicitly handled by the developed architecture, as these are middleware-specific issues that can be modeled as component failures.
- A3 **Von Neumann architecture as a computational platform**. While hardware devices such as Field Programmable Gate Arrays (FPGA) and microcontrollers are heavily utilized in robotics, this work only covers applications designed for computing hardware based on Von Neumann architecture, e.g., modern Personal Computers (PC).

3.1. Common Requirements

The common requirements (Table 3) are derived from the desired features outlined in Section 2.2. and thus apply to the whole architecture.

Table 3. Common requirements

Req. nr.	**Requirement**
ADAPTIVE	***The software stack must be able to adapt to:***
C-1.1	*Dynamic changes in the mission requirements* [40]
C-1.2	*Dynamic changes in the robot, i.e., failures, updates* [26]
C-1.3	*Dynamic changes in the environment* [25]
SCALABLE	***The software stack must be scalable, meaning it is:***
C-2.1	*Reusable for different task domains* [4]
C-2.2	*Extendible with tasks and system components* [4]
C-2.3	*Maintainable so that the system can gradually evolve without requiring complete refactorization of the whole architecture* [4]
C-2.4	*Middleware-independent implementation, i.e., middleware-specific properties cannot leak into the main codebase* [4]
MAC	***The software stack can be applied for tasks involving:***
C-3.1	*Decentralized and mixed multi-agent topologies* [26], [37]
C-3.2	*Heterogeneous hardware platforms* [37]
HRI	***The software stack facilitates designing HRI applications that may contain:***
C-4.1	*Multiple interaction modalities* [37]
C-4.2	*Multiple levels of autonomy* [26], [28]

3.2. Executive Layer Requirements

This subsection derives the requirements for the executive layer (Table 4), which manages the execution of tasks (Section 2.3.2). The derivation is based on common requirements (Table 3) and additional attributes deemed necessary for any implementation of an executive layer.

Table 4. Specialization of common requirements to the executive layer (E_{C}-1 to E_{C}-4) and executive layer specific requirements (*E-1* to *E-4*).

Req. nr.	Requirement
E_{C}-1 ADAPTIVE	*Actions and Tasks can be dynamically invoked, reconfigured, and stopped*
E_{C}-2 SCALABLE	*C-2.1: Reusable across multiple task domains* *C-2.2: Extendible with new actions and tasks* *C-2.3: Maintainable, i.e., implementation of actions, tasks, and executive layer is decoupled* *C-2.4: Middleware independent*
E_{C}-3 MAC	*C-3.1 Individual actions in a shared task can be attributed to specific decentralized actors*
E_{C}-4 HRI	*C-4.1: Interaction modalities are decoupled from the implementation of the executive layer via a DSL*
	Executive layer-specific requirements
E-1	*Expressive task composition, including sequences, concurrency, cycles, and statements* [74]
E-2	*Input and output parametrization of granular tasks* [79]
E-3	*Continuous/reactive behaviors* [74]
E-4	*Hierarchical task composition* [74]

3.3. Functional Layer Requirements

This subsection derives the requirements for the functional layer (Table 5), which manages the allocation, configuration, and use of components (sensors, actuators, algorithms) or resources in general (Section 2.3.3). The derivation is based on common requirements (Table 3) and additional attributes deemed necessary for any implementation of a functional layer.

Table 5. Specialization of common requirements to the functional layer (F_{C1} to F_{C3}) and functional layer specific requirements (*F-1* to *F-3*).

Req. nr.	Requirements derived from common requirements
$F_{C\text{-}1}$ ADAPTIVE	*Resources can be dynamically invoked, reconfigured, and stopped*
$F_{C\text{-}2}$ SCALABLE	*C-2.1: Reusable across multiple task domains* *C-2.2: Extendible with new resources* *C-2.3: Maintainable, i.e., implementation of resources and the functional layer is decoupled. Changes in a resource do not affect the implementation of the resource management functionalities and vice versa.* *C-2.4: Middleware independent*
$F_{C\text{-}3}$ MAC	*C-3.1: Information about a resource can be shared*
	Functional layer-specific requirements
F-1	*Supports multiple consumers – A resource can be simultaneously allocated and used by multiple clients*
F-2	*Supports hierarchies – A resource can use other resources as dependencies* [9]
F-3	*Resource accounting (consumers and dependencies) information must be recoverable* [64]

3.4. Summary

This work is structurally based on layered architectures, but deliberation, and policies for dynamic component substitution are not covered as these are domain and task-specific topics. Common requirements (Table 3) were established, with Executive and Functional layer requirements further specializing them for their respective layers.

4. RELATED WORK

This section evaluates the related work with respect to the requirements defined in Section 3. The analysis is segregated into three subsections, where Section 4.1 covers available implementations of layered architectures, and the other two cover the existing solutions in the executive (Section 4.2) and functional (Section 4.3) layers respectively.

4.1. Layered Architecture Implementations

This section covers the related work where executive and functional layers have been integrated into a uniform architecture. Based on the requirements defined for layered architectures (Table 6), the existing work can be compared in terms of how both layers exhibit adaptive, scalable, MAC, and HRI properties.

Brunner et al.[80] combine RAFCON[7] based task management with Links-and-Nodes based dynamic resource management. The RAFCON tasks are augmented with a resource dependency system, where each task can define data, world state information, or a computational resource dependency prior to the execution. However, little is disclosed about how the Links-and-Nodes-based resource management works and its limitations. The only publicly available implementation is for RAFCON[1], which covers task management (excluding resource dependency augmentations). Also, resources are globally managed, meaning that resources cannot dynamically depend on sub-resources (not hierarchical), e.g., a navigation component depending on sensing and localization components.

TritonBot[81] addresses several key challenges in long-term deployment, such as backward and forward compatibility for data logging, secure communication, and dynamic resource management via containerization. The project utilizes SMACH[5] for task management and Rorg[9] for managing resources. While the work provides valuable lessons learned, the implemented software stack is designed for the hardware layout described in [81] and lacks a methodology for defining tasks other than its original mission[2].

Osmosis[64] is an architecture for fault-tolerant navigation applications. Effectively, Osmosis has a task management layer that allows users to define and switch between navigation missions and a fault-tree-based fault management layer where the user can implement custom fault detection and recovery routines. Also the architecture includes a Human-Machine Interaction (HMI) component, which enables variable LoA (autonomous navigation, teleoperation). The publicly available implementation of Osmosis[3] promotes reuse with well-structured software packaging, but it is limited to navigation tasks. Also, the authors mention the

[1] github.com/DLR-RM/RAFCON
[2] github.com/CogRob/TritonBot
[3] gitlab.com/osmosis

dynamic recovery of failed components [64], yet the implementation details are undisclosed and do not show in the publicly available source code.

STRANDS[82] project focuses on the long-term deployment of an autonomous service robot for navigation-related tasks. The architecture of STRANDS is capable of task and resource management and the publicly available implementation[4] is well organized and promotes reuse. However, task management is limited to sequential behaviors; tasks are implemented as ROS action servers and run constantly in the background. The framework is designed for the navigation tasks and the resource management is limited to restarting ROS nodes.

SOTER[83] architecture combines state machine-based task management with a run-time monitoring system that can switch the behavior of predefined component modules between advanced and safe control modes. While SOTER uses a state machine programming language and code executor called P[84] the whole framework is implicitly designed for navigation tasks. The openly available implementation[5] is demo-specific and unusable as a generic framework. Also, SOTER does not manage resources, i.e., all robotic components are started up separately which the custom runtime assurance modules send commands to. Thus, a critical failure in the components layer or Robot SDK layer (as described in [83]) renders the robot unrecoverable. Finally, the tasks are not dynamically invokable, i.e., tasks are precompiled and executed as a single application.

Conclusively, a considerable amount of effort has been put into the research of adaptive robotic systems. Still, the developed frameworks are often not versatile when assessed on the basis of requirements defined in Table 3. They either lack dynamic task or resource management, are not usable for tasks other than initially designed for (not task agnostic), or require customization of managed resources (lack of minimal overhead). While some customization is technically plausible depending on the system requirements, it increases the maintenance and development effort, especially in keeping the customized resources up-to-date.

4.2. Executive layer

The core of an executive layer is a software module that can manage tasks, as covered in Section 2.3.2. This section evaluates task management tools commonly applied in the robotics domain based on requirements defined in Table 4. Table 7 summarizes the capabilities of common robotic task management tools.

BehaviorTree.CPP[6] is a C++ behavior tree implementation (see Section 2.3.2), supporting hierarchical, reactive, parametrized, and expressive tasks. The tasks can be described in an Extensible Markup Language (XML) based format, which helps to decouple task descriptions from the library's implementation. The core implementation of BehaviorTree.CPP supports adding new tasks and behavior implementations during runtime via plugins, thus making the system scalable

[4] github.com/strands-project
[5] github.com/Drona-Org/SOTERonROS

Table 6. Summary of the capabilities of layered architecture reported in the literature. Comparison based on common requirements defined in Table 3. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×" and "*unkn.*" (unknown) indicates unavailability of information.

	Executive Layer		Functional Layer			
Framework	**Adaptive**	**Scalable**	**Adaptive**	**Scalable**	**MAC**	**HRI**
Brunner et al.	✓	✓	*lim.*	*unkn.*	×	×
TritonBot	*lim.*	×	✓	✓	×	✓
Osmosis	✓	×	✓	*unkn.*	×	✓
STRANDS	×	×	*lim.*	×	×	✓
SOTER	×	×	*lim.*	*lim.*	×	×
TeMoto	✓	✓	✓	✓	✓	✓

in terms of maintainability and extensibility. The tasks can be started and stopped during run-time, but the underlying tree cannot be modified while running. While starting, stopping, and adding tasks during run-time is supported, it is up to the user to programmatically register the new behaviors instead of having a built-in automatic indexing/search. Concurrent (asynchronous) behavior nodes are expected to return immediately, i.e., behaviors are not multi-threaded and it is up to the user to implement a mechanism for long-running tasks. Hierarchical behavior trees are supported as well, but the XML notation requires subtrees to be explicitly denoted. This means if one robot can perform a specific task via a single atomic behavior, and another robot can perform a similar task via a combination of multiple behaviors (described as a sub-tree) with the same outcome, then the XML notation of the parent tree has to be different for either robot. This may become a problem if the XML descriptions are generated by a system that knows only the generic capabilities of the robots. Finally, BehaviorTree.CPP does not scale to shared multi-robot tasks since it has no distributed implementation for the ticking mechanism and the blackboard.

RAFCON[7] is a Python-based Graphical User Interface (GUI) and executive that implements tasks as hierarchical state machines. States can: be connected to sequential, concurrent, and cyclical graphs; have input and output parameters; have multiple outcomes; and can be reactive. Thus, RAFCON supports hierarchical, reactive, parametrized, and expressive tasks. Tasks can be invoked, paused, modified, and stopped during runtime, which facilitates adaptability. States are implemented as modular Python scripts, which can be defined and used during runtime and reused across different tasks, thus making the system scalable in terms of maintainability and extensibility. RAFCON has a JSON-based DSL, which describes the graphical structure of the tasks via a list of transitions and data flows between the states. While the states can be executed concurrently, RAFCON does not support concurrent state machines or shared and decentralized multi-robot tasks.

SMACH[5] is a Python-based executive that implements tasks as hierarchical state machines. States can: be connected to sequential, concurrent, and cyclical graphs; have input and output parameters; and have multiple outcomes. Thus, SMACH supports hierarchical, parametrized, and expressive tasks, but reactive states are not supported. The tasks can be started and stopped during run-time, but cannot be modified while running. SMACH is reusable across multiple task domains and can be extended with new states. However, it is up to the user to programmatically import and register the new states, as opposed to having a built-in automatic indexing/search. Also, SMACH lacks a DSL for describing and executing tasks, thus making it difficult to use in a system where tasks are generated programmatically during run-time. Finally, SMACH does not support shared and decentralized multi-robot tasks.

SkiROS2[85] is a Python-based general-purpose task management framework that combines behavior trees with knowledge representation and reasoning capabilities. It supports hierarchical, reactive, parametrized, and expressive tasks while lacking support for cyclical behaviors. Primitive skills, i.e., behaviors, are outlined by a list of input/output parameters and pre-, hold-, and post-conditions, which set the required world model conditions for valid execution of the behavior. Thus a built-in reasoner can choose a set of skills most appropriate for a given task and conditions. SkiROS2 is reusable across multiple task domains and extendible with new behaviors. The framework is implemented in ROS2 and user-defined behaviors are bundled up via ROS2 packages, which makes SkiROS2 dependent on a specific middleware. SkiROS2 lacks a DSL for describing and executing tasks, thus making it difficult to use in a system where tasks are generated programmatically during run-time. Finally, SkiROS2 does not support shared and decentralized multi-robot tasks.

TaskForce[8] is a Python-based GUI and executive that implements tasks as hierarchical state machines. Similar to RAFCON and SMACH, tasks are formed by connecting a specific event/outcome of a state to a callback of another state. Thus, sequential and cyclical graphs can be formed, and the state outcome semantics can be leveraged to build conditional structures. However it is unclear from the documentation whether TaskForce supports concurrency. New tasks can be modularly added and reused across different projects. Also, TaskForce supports parametrization, but there is no concept of output parameters. Thus, the output of a task can be passed to the input of another task only via a custom blackboard. TaskForce has a JSON-based DSL, which outlines tasks as a list of subtasks and event-callback connection pairs. While DSL supports describing hierarchical tasks (task groups), it is similar to BehaviorTree.CPP where the notation requires task groups to be explicitly denoted. This may become a problem if the JSON descriptions are generated by a system that only knows the generic capabilities of the robots. TaskForce does not scale to decentralized and shared multi-robot tasks.

SMACC / SMACC2[71] is a C++ based behavioral hierarchical state machine implementation developed specifically for ROS/ROS2 middleware. Compared to SMACH or RAFCON, SMACC introduces the concept of state orthogonals, which are asynchronous events that the state can react to when active. The orthogonals can be used to trigger callback procedures and state transitions. Also, the states can be connected to sequential, cyclical, and concurrent graphs, and the state outcome semantics can be leveraged for building conditional structures. The state machines or tasks in SMACC are implemented as executables, which means that tasks cannot be composed during run-time from modular state nodes. The source code for states is maintained along with the parent state machine, which inhibits code reuse across multiple state machines. Input parameters can be passed to states via the ROS parameter server, but states do not support output parametrization. Finally, SMACC does not support shared and decentralized multi-robot tasks.

Table 7. Summarized capabilities of common robotic task management tools. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×".

	Language	Task Model	ADAPTIVE ($E_{A\text{-}1}$)	SCALABLE ($E_{A\text{-}2}$)	MAC ($E_{A\text{-}3}$)	HRI ($E_{A\text{-}4}$)	Expressive (E_1)	Parameterized (E_2)	Reactive (E_3)	Hierarchical (E_4)
BehaviorTree.CPP	C++	BT	*lim.*	✓	×	✓	✓	✓	✓	✓
RAFCON	Python	FSM	✓	✓	×	✓	✓	✓	✓	✓
SkiROS2	Python	BT	*lim.*	*lim.*	×	×	✓	✓	✓	✓
SMACH	Python	FSM	*lim.*	*lim.*	×	×	✓	✓	×	✓
TaskForce	Python	FSM	✓	✓	×	×	*lim.*	*lim.*	×	✓
SMACC2	C++	FSM	×	×	×	×	✓	*lim.*	✓	✓
TeMoto	C++	CFG	✓	✓	✓	✓	✓	✓	*lim*	✓

4.3. Functional Layer

Table 8 summarizes the existing work's support for requirements outlined in Table 5. Most of the reviewed work supports dynamic resource control with the exception of SOTER[83], which can only switch between two resources, and Osmosis[64] and STRANDS[82], which can only restart the resources. Rorg[9] and MROS[10] stand out as having the most supported capabilities. Yet Rorg has no resource status information propagation implementation, meaning developers cannot program recovery behaviors for resource failures. Also in Rorg a resource is a containerized OS process, limiting the versatility of what a resource could be, e.g., a visualization plugin in RViz (see Section 8.1). MROS, on the other hand,

does not allow users to define new configurations for a deployed/running system; scale to concurrent configurations and resource usage overlaps, where independent tasks can simultaneously access the same resource; work with third-party resources without prior modification that fulfills the System Modes[86] design criteria. Finally, all the reviewed contributions in Table 8 implicitly assume a single robot layout, limiting the multi-robot scalability.

Table 8. Coverage of resource management capabilities by the existing work. Support for a specific capability is indicated as "✓" (full support) or as "*lim.*" (limited support). Lack of support is indicated via "×" and "*unkn.*" (unknown) indicates unavailability of information.

	ADAPTIVE ($F_{A\text{-}1}$)	SCALABLE ($F_{A\text{-}2}$)	MAC ($F_{A\text{-}3}$)	Multi-Consumer (F_1)	Hierarchical (F_2)	Recoverable (F_3)
Rorg	✓	✓	×	✓	✓	✓
MROS	✓	✓	×	×	✓	✓
DyKnow	✓	*lim.*	×	×	×	×
Osmosis	✓	*unkn.*	×	×	×	×
STRANDS	*lim.*	×	×	×	×	×
SOTER	*lim.*	*lim.*	×	×	×	×
Wirkus et al.	✓	✓	×	×	×	*unkn.*
Brunner et al.	✓	*unkn.*	*unkn.*	*unkn.*	×	*unkn.*
TeMoto	✓	✓	✓	✓	✓	✓

4.4. Summary

This section analyzed the available implementations of layered architectures with respect to the requirements established in Section 3. To conclude:

- The available full-stack implementations of layered architectures (Section 4.1) often have task domain-specific implementation for the executive layer, making them hard to scale. Also multi-agent collaboration is rarely considered in the design of the available work.
- The software libraries commonly used for designing executive layer-specific (task management) functionalities (Section 4.2) are often designed for tasks that do not change throughout deployment, i.e., are not adaptive. Also the possibility to outline tasks by methods other than using the library's Application Programming Interface (API), opposed to using a domain-specific language, is not common and thus hinders scalability towards HRI applications.

- The software libraries commonly used for designing functional layer-specific (resource management) functionalities (Section 4.3) are rarely considering the life-cycle and consumer count of a resource. Thus dynamic hierarchical resource allocation/de-allocation is not achievable, which limits the adaptiveness of the system.

5. TEMOTO – ARCHITECTURE OVERVIEW

Following the **RQ2**, this section covers the architecture and implementation of TeMoto, the software framework designed to comply with the challenges in robots deployed in high-risk and high-complexity tasks.

The design of TeMoto is driven by the requirements (Section 3) using three-layer syndicate architecture as the backbone, which fundamentally promotes scalable design (Requirement C-2) via separating the structure into loosely coupled and highly cohesive layers that can be deployed on a multi-robot system (Requirement C-3). Having established the base structure, the rest of the common requirements (Table 3) are more specific to the **RQ1** and cannot be directly adapted from the syndicate architecture, which leads to this work and the design of TeMoto architecture. The next paragraph first introduces the overall architecture, followed by a detailed description of how such design helps to address the requirements defined in Section 3, i.e., facilitate adaptive, scalable, multi-agent oriented, and HRI-driven design.

Figure 6 shows the structure of TeMoto segregated into task management and resource management layers. Figure 7 shows the decentralized extension of TeMoto to a multi-agent system. The task management layer, corresponding to the executive layer in the three-tier architecture (Section 2.3.2), performs run-time task execution based on high-level task descriptions. Each task is described as a combination of modular sub-tasks grounded to corresponding executable robotic actions by the task management layer. The robotic actions contain arbitrary user-defined code that may require resources, such as actuators and sensors. Since the actions are managed dynamically (asynchronously invoked, modified, stopped) during run-time, the availability of a resource must also be checked, and resource allocation must be performed during run-time. Thus, the resource management layer, which corresponds to the functional layer in the three-tier architecture (Section 2.3.3), provides dynamic access to resources while accounting for multiple resource allocations, resource dependencies/hierarchies, etc. Specific to the implementation of TeMoto, the resource management layer contains multiple individual resource managers, each with a specific set and scope of resources they are managing (e.g., Visualization Manager or Component Manager).

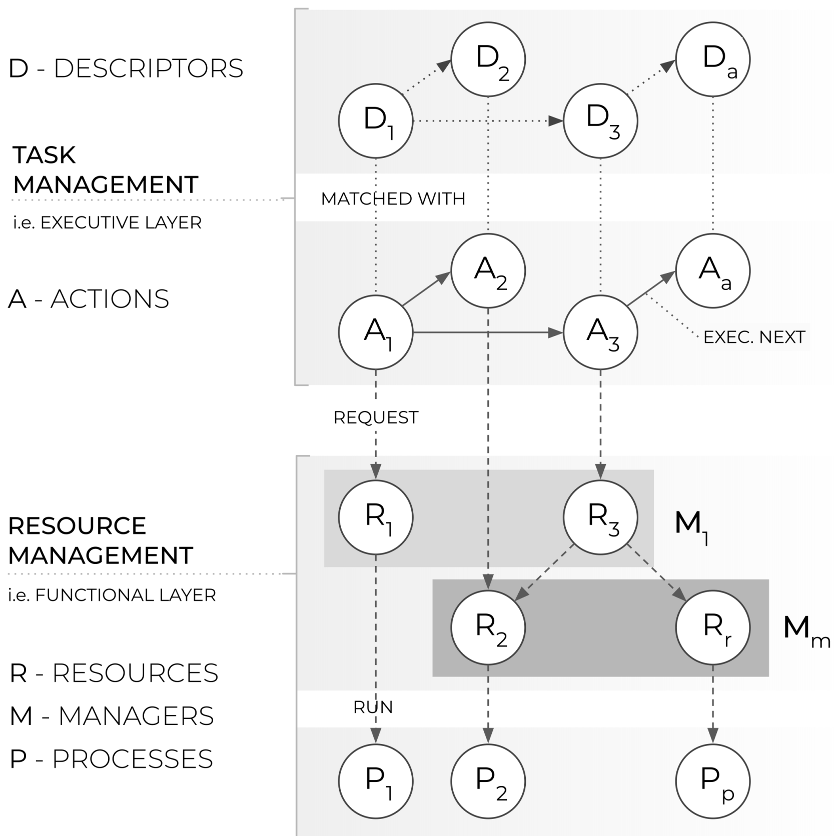


Figure 6. The architecture of TeMoto consists of task management and resource management layers.

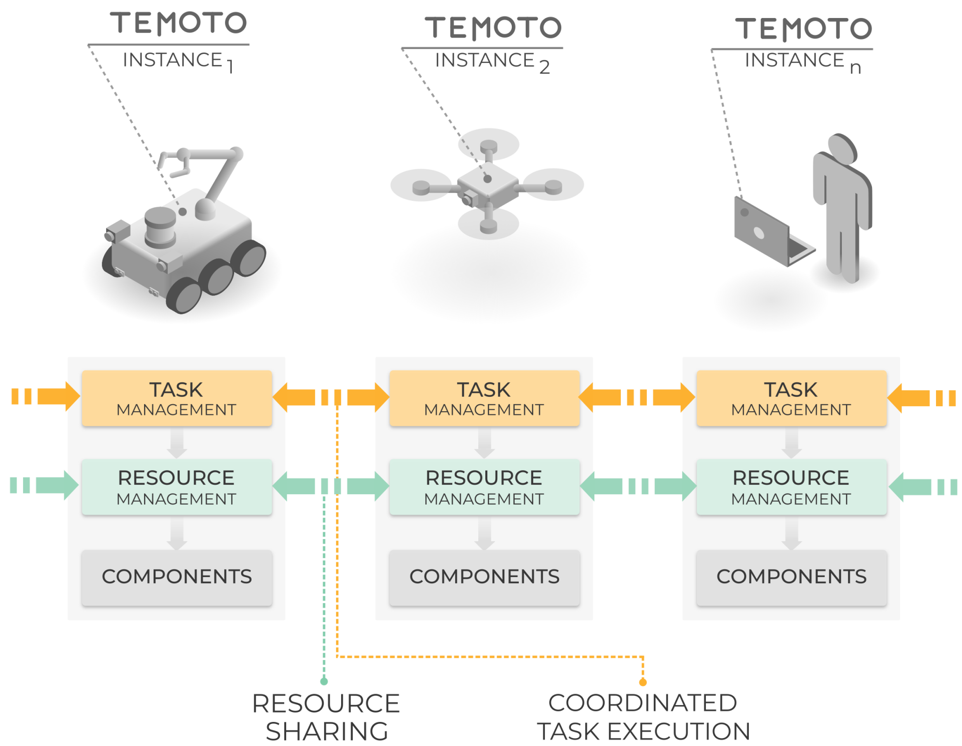


Figure 7. Extension of TeMoto architecture to a decentralized and heterogeneous multi-agent system.

5.1. Requirements Conformance

With a brief introduction to TeMoto architecture, this section describes (Table 9) how each requirement in Table 3 can be addressed via TeMoto architecture.

Table 9. TeMoto architecture's conformance to common requirements (Table 3).

Requirement	TeMoto design feature
Adaptive – The software stack must adapt to:	
C-1.1: *Dynamic changes in the mission requirements.*	Missions can be described as a combination of actions that can be invoked, modified, and stopped during run-time.
C-1.2: *Dynamic changes in the robot, i.e., failures, and updates.*	Resource managers can dynamically start and stop resources and provide status updates about, e.g., failures or changes, to resource consumers both for single or hierarchical resources.
C-1.3: *Dynamic changes in the environment.*	Combined with dynamic task and resource management, the user can create routines to trigger a reconfiguration process for specific events in the environment.
Scalable – The software stack must be scalable, meaning it is:	
C-2.1: *Reusable for different task domains.*	The task and resource management layers have no underlying assumptions about the deployment-specific details or domains.
C-2.2: *Extendible with tasks and system components.*	New self-contained/independent tasks and components can be added to the existing setup.
C-2.3: *Maintainable so the system can gradually evolve without requiring complete refactorization of the whole architecture.*	Architectural layers are independent, i.e., the interfaces between task descriptions, task management, and resource management are standardized, hence changes in one layer do not propagate to the other layers.
C-2.4: *Middleware independent implementation, i.e., middleware-specific properties cannot bleed into the main codebase.*	The core components of TeMoto are designed in C++ utilizing libraries that are available for, e.g., ROS1 and ROS2. These components are then adapted for specific middleware with thin wrappers.
C-3.1: *Decentralized and mixed multi-agent topologies.*	TeMoto is based on a three-layer syndicate architecture (Figure 4), where each agent is assumed to be autonomous but able to collaborate with other agents. The task management layer allows developers to distribute tasks between multiple robots with no centralized coordination while executing the tasks. Similarly, the resource management layer allows different agents to access each other's resources, where the information about the resources is distributed non-centrally.

Continued on next page

Table 9 – continued from previous page

Requirement	TeMoto design feature
C-3.2: *Heterogeneous hardware platforms.*	The decentralized interaction within the agents does not require homogeneous software or hardware layout.
HRI – The software stack facilitates designing HRI applications that may contain:	
C-4.1: *Multiple interaction modalities.*	The task management layer accepts task descriptions in a format independent from any certain interaction modality, thus allowing the developers to design interfaces with any desired modality.
C-4.2: *Multiple levels of autonomy.*	TeMoto utilizes publisher-subscriber-based middleware implementations, such as ROS. Thus, lower autonomy levels, such as direct teleoperation, and higher autonomy levels, such as supervision via task descriptions, are both attainable.

6. TEMOTO – TASK MANAGEMENT

The Task Management layer in the TeMoto framework is based on the Executive layer outlined in Section 2.3.2. The Task Management layer is responsible for grounding task descriptions, which originate from layers above, into executable behaviors (Figure 8). This section covers the details of the implementation of the Task Management layer. First, the model of a task is introduced (Section 6.1), followed by defining the text-based format, i.e., the Unified Meaning Representation Format (UMRF)III, for describing the tasks (Section 6.2). Next, the implementation details of TeMoto's task management system, the Action Engine, are covered (Section 6.3) and limitations and the scope of future work are discussed (Section 9.1).

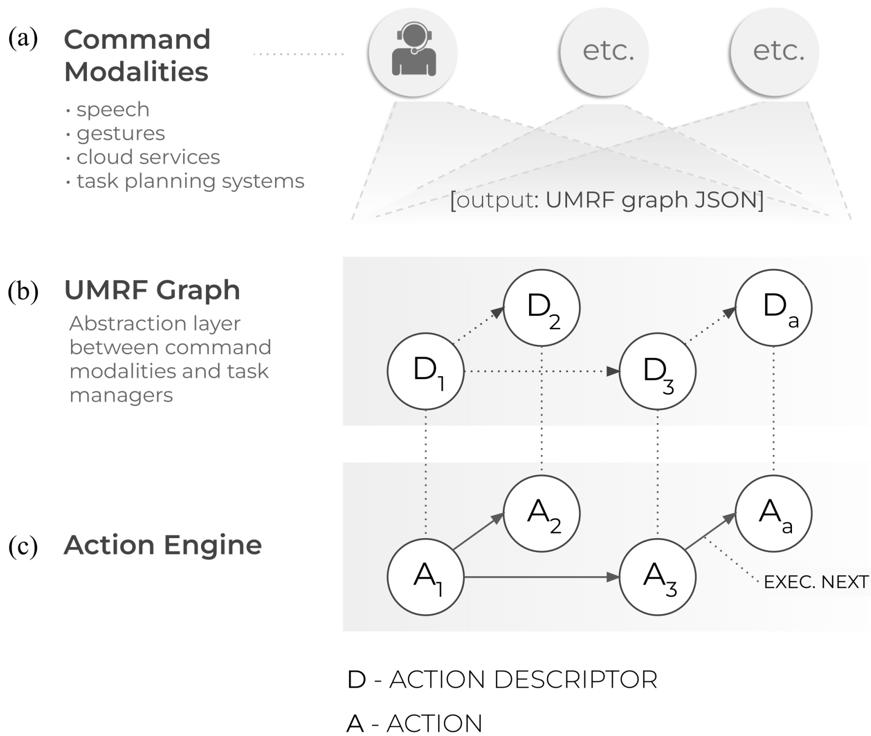


Figure 8. Structure of the Task Management layer in the TeMoto framework. Tasks can be issued via different input sources (a), where each source is parsed to a common UMRF graph (b). The UMRF graph maps to respective actions on the robot (c), thus UMRF formalism makes the robot independent from different input modalities.

6.1. Model

Tasks are outlined as a combination of modular behaviors called actions. Such a combination of actions is also called a UMRF graph, a directed graph that indi-

cates the execution flow. The task model in TeMoto resembles a loose combination of Petri Nets, Hierarchical State Machines, and Behavior Trees. Semantically UMRF graph supports (Figure 9):

- **Sequential**, **concurrent**, and **cyclical** combination of actions,
- **Hierarchical** graphs, where an action can be a sub-graph,
- **Conditionals** and **error management**,
- **Parametrization** – Actions can be reused with different set of input parameters,
- **Reactive** actions, where an action can execute without being explicitly invoked,
- **Shared multi-agent** graphs.

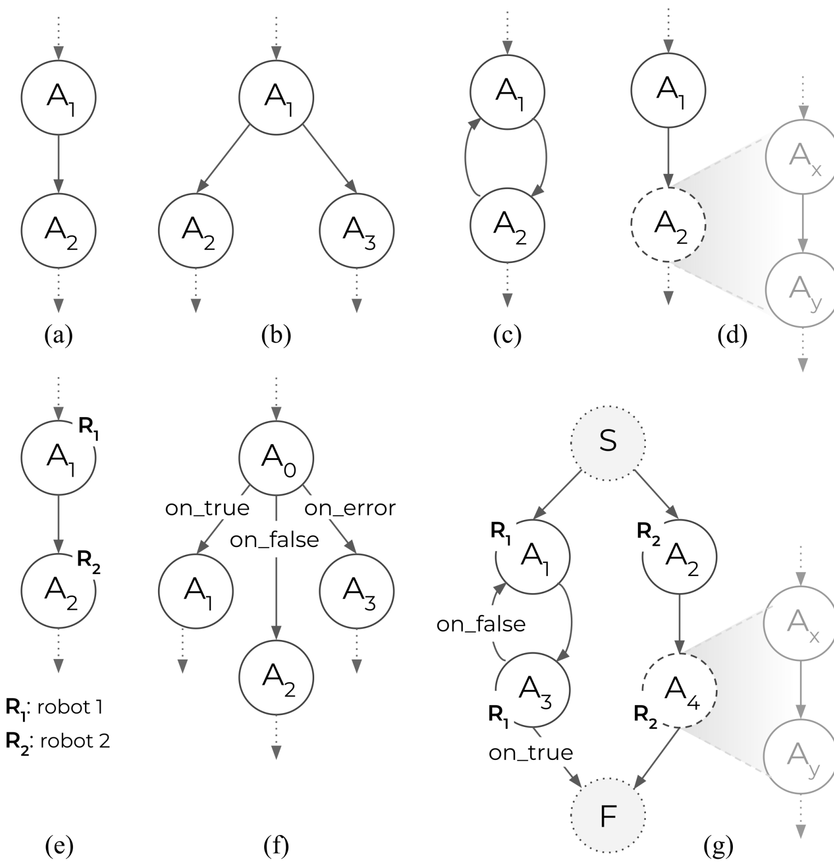


Figure 9. Main semantic properties of UMRF graph notation, including sequences (a), concurrency (b), cycles (c), hierarchical graphs (d), multi-agent graphs (e), conditionals, and error management (f). Example UMRF graph with aforementioned properties combined, where "S" denotes graph entry and "E" denotes graph exit (g).

6.1.1. Actions

Action a is defined as a 3-tuple:

$$a = \langle s, P_{in}, P_{out} \rangle$$

where

- s is the state of the action,
- P_{in} is a set of input parameters,
- P_{out} is a set of output parameters.

The state s can be any of the following:

$$s \in \mathbf{S} = \{s_r, s_p, s_s, s_e, s_{id_0}, s_{id_T}, s_{id_F}\}$$

where

symbol	description	*one-hot* representation
s_r	*running*	$[1,0,0,0,0,0,0]$
s_p	*paused*	$[0,1,0,0,0,0,0]$
s_s	*stopping*	$[0,0,1,0,0,0,0]$
s_e	*error*	$[0,0,0,1,0,0,0]$
s_{id_0}	*idle*	$[0,0,0,0,1,0,0]$
s_{id_T}	*idle after outcome T*	$[0,0,0,0,0,1,0]$
s_{id_F}	*idle after outcome F*	$[0,0,0,0,0,0,1]$

6.1.2. Graph

UMRF graph g is defined as a 3-tuple (example shown in Fig. 10):

$$g = \langle A, E, C \rangle$$

where

- A is a set of actions, where A_0 is the root action of the graph and A_n is the finishing action of the graph.
- E is a set of edges between actions, so that:
$$\forall a \in A, \Big((a,a) \notin E\Big) \wedge \Big((a,A_0) \notin E\Big) \wedge \Big((A_n,a) \notin E\Big) ,$$
i.e., an action cannot be connected to itself, root action A_0 has no parents and finishing action A_n has no children.
- C is a set of conditions which describe the state transition criteria per each action.

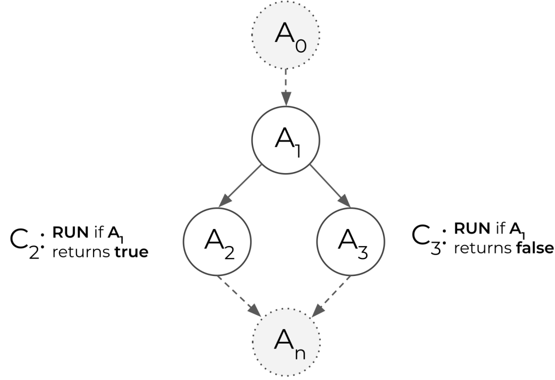


Figure 10. Example UMRF graph where actions are denoted via "A" and conditions for running actions via "C".

6.1.3. Conditions

The state transition conditions C_i of an action A_i indicate which state the action should transition into, given the current state of the parent actions (i.e. state *evolution*, covered in Section 6.1.4). Conditions C_i is defined as a 4-tuple:

$$C_i = \langle C_{R_i}, C_{P_i}, C_{S_i}, C_{E_i} \rangle$$

where

- $C_{R_i} = \{c_{R_0}, c_{R_1}, \ldots, c_{R_j}\}$ is a set of *run* conditions, where each condition $c_R \subset (A \times \mathbf{S})$ is a set of *action-state* pairs
- $C_{P_i} = \{c_{P_0}, c_{P_1}, \ldots, c_{P_j}\}$ is a set of *pause* conditions, where each condition $c_P \subset (A \times \mathbf{S})$ is a set of *action-state* pairs
- $C_{S_i} = \{c_{S_0}, c_{S_1}, \ldots, c_{S_j}\}$ is a set of *stop* conditions, where each condition $c_S \subset (A \times \mathbf{S})$ is a set of *action-state* pairs
- $C_{E_i} = \{c_{E_0}, c_{E_1}, \ldots, c_{E_j}\}$ is a set of *error* conditions, where each condition $c_E \subset (A \times \mathbf{S})$ is a set of *action-state* pairs
- $C_{R_i} \neq C_{P_i} \neq C_{S_i} \neq C_{E_i}$, i.e., conditions are unique

6.1.4. State transitions

The states can change as a result of one of three events:

- **Outer** event, where the state is changed by an external force, i.e., user starting, pausing, or stopping the graph.
- **Local** event, where the state changes spontaneously, e.g., an action finishes execution and goes from s_r to s_{id_T} or s_{id_F} or s_{id_E}.
- **Evolution** event, where the state of an action is changed as a reaction to other events, e.g., child action is started after the parent action has finished.

The possible state transitions are indicated in Table 1, where row indicates the starting state, column indicates the state after the transition. The events that

can trigger the transition are denoted as "O", "L", and "E", or Outer, Local, and Evolution respectively.

Table 10. State transition table. The events that can trigger the transition are denoted as "O", "L", and "E", or Outer, Local, and Evolution respectively.

		To state					
		s_r	s_p	s_s	s_e	s_{id_0}	s_{id_x}
From state	s_r	-	E/O	E/O	E/L	-	L
	s_p	E/O	-	E/O	E/L	-	-
	s_s	-	-	-	E/L	-	L
	s_e	-	-	-	-	O	-
	s_{id_0}	E/O	-	-	E	-	-
	s_{id_x}	E	-	-	E	-	-

When a graph g is started, the states of the actions A are initialized/transitioned by setting:

$$s_i = \begin{cases} s_{id_T} \; or \; s_{id_F} \; or \; s_e, & i = 0 \\ s_{id_0}, & \text{otherwise} \end{cases}$$

If the graph is started separately, then $s_0 = s_{id_T}$. If the graph is started as a hierarchical sub-graph, then the specific transition depends on the state transition in the parent graph, i.e., s_{id_T} or s_{id_F} or s_e.

As indicated by Table 10, an action can *evolve* into states s_r, s_p, s_s or s_e. The outcome of the evolution depends on the conditions defined by the user (see Section 6.1.3) As the potential combinations for different conditions grows exponentially with the number of parents, then each action is given a set of *default* conditions in addition to *user-defined* conditions. Thus the conditions are evaluated accordingly:

1. evaluate *user-defined* conditions.
2. transition to s_e (error) if: no mach in step 1 and any parent is in state s_e.
3. transition to s_r (run) if: no mach in step 2 and any parent is in *idle true* or *idle false* state $s_{id_x} = s_{id_T} + s_{id_F}$.
4. transition to s_s (stopping) or (paused): *not defined*.

The default conditions of transitioning the state s of an action a given the state of parents $\mathbf{S}$ can be formally expressed as:

$$s_{evolved} = {s_r}^T [max(sign(\mathbf{1S}s_{id_x}) - sign(\mathbf{1S}s_e), 0)] + {s_e}^T sign(\mathbf{1S}s_e) \tag{6.1}$$

where:

- s_r and s_e are *running*, *error* states in one-hot encoded format.
- $\mathbf{S}$ is a matrix, where each row is a one-hot encoded state of a parent of A_i.
- $\mathbf{1}$ is a row vector of size $|\mathbf{S}_i|$ with all elements equal to one.

6.2. Text-Based Format

Task management systems are tied to a specific programming language (Python, C++, etc. See Section 4.2) used to implement the tasks. Yet HRI systems can have a variety of command input modalities, meaning if the developer desires to control a specific task management system, then each command modality has to be appropriated (hard-coded) for this. Therefore, the task management of an HRI-enabled robotic system must be decoupled from command modalities. Task description or DSLs are leveraged for this purpose, acting as an abstraction layer between input sources and task implementations. Figure 11 shows an example of a multi-modal HRI setup (speech, gaze, gestures), where the instruction "Go to the workshop, pick up that object, and then go over there" is segregated into three parametrized instructions. The instructions are combined into a sequence outlined in a task description format. Thus, the robot only needs to be able to read this format and execute the task.

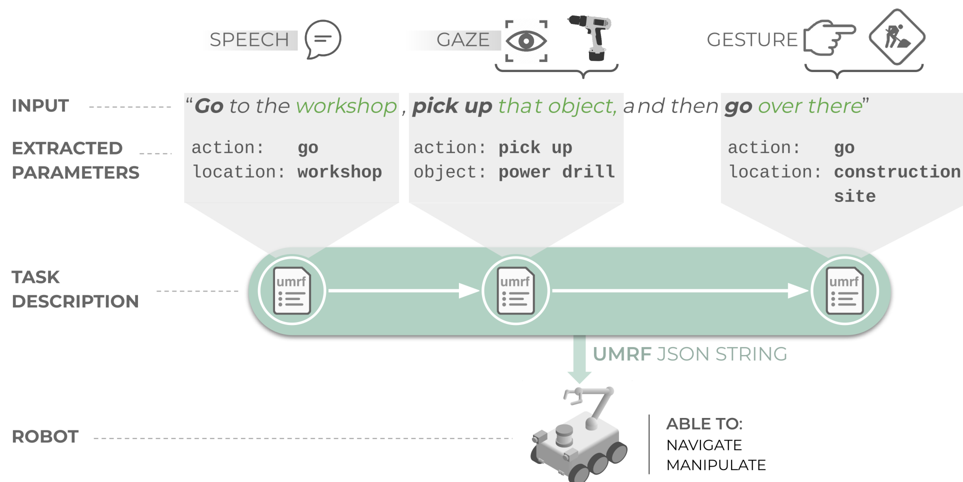


Figure 11. Potential use-case of a multi-modal HRI system that benefits from having an intermediate task description format for fusing the modalities and commanding the robot.

UMRF is a DSL designed for HRI- and MRC-oriented use cases. The UMRF notation allows for describing complex tasks independent of the task domain while having a clearly defined and easily adoptable serialized form via JSON, facilitating prototyping. The semantics of UMRF are inspired mainly by Petri Nets, Hierarchical State Machines, and Behavior Trees. The syntax closely resembles Semantic frames, while the serialized form is inspired by the JSON-based data exchange format of Google Actions and Amazon Skills. The attributes that describe a UMRF graph are outlined in Table 11:

Table 11. Description of UMRF JSON notation attributes.

Attribute	Type	Description
MAIN ATTRIBUTES		
Name	*[string]*	Unique name of the graph
Actions	*[list of actions]*	List of actions in the graph
Graph entry	*[list of child relations]*	List of actions that mark the start of the graph
Graph exit	*[list of parent relations]*	List of actions that mark the start of the graph
ACTION ATTRIBUTES		
Name	*[string]*	Name of the action
Instance ID	*[unsigned integer]*	Instance ID, i.e., a graph can contain multiple instances of the same action
Actor	*[string]*	Name of the agent that is supposed to perform the action
Type	*[string]*	Type of the action (reactive or non-reactive)
Input parameters	*[list of parameters]*	Data the action is expecting
Output parameters	*[list of parameters]*	Data the action is producing
Parents	*[list of parent relations]*	Actions that are run right before the child
Children	*[list of child relations]*	Actions that are run after the child has finished running
Conditions	*[list of conditions]*	Additional execution conditions on top of default conditions
PARENT RELATION ATTRIBUTES		
Name	*[string]*	Name of the parent action
Instance ID	*[unsigned integer]*	Instance ID of the parent action
Remappings	*[list of remappings]*	List of parameter name remappings
CHILD RELATION ATTRIBUTES		
Name	*[string]*	Name of the child action
Instance ID	*[unsigned integer]*	Instance ID of the child action
CONDITIONS ATTRIBUTES		
Operation	*[string]*	Behavior of the child action. Any of: '*run*', '*pause*', '*stop*', '*error*'
Parent States	*[list of parent-state pairs]*	State of parents required for the condition to be valid
REMAP ATTRIBUTES		
From	*[string]*	Name of the parameter to be remapped
To	*[string]*	New remapped name
PARAMETER ATTRIBUTES		

Continued on next page

Table 11 – continued from previous page

Attribute	Type	Description
Name	*[string]*	Name of the parameter
Type	*[string]*	Type of the parameter. No restrictions but non-native JSON data can only be generated and passed during runtime
Default	*[string or double or boolean or strings or doubles or booleans]*	Default value of the parameter. Restricted to native JSON types
Allowed values	*[strings or doubles or booleans]*	List of values that can be passed to the action, otherwise the data is ignored. Restricted to native JSON types
Required	*[boolean]*	Denotes if the parameter must have a value for the action to run

6.3. Implementation

TeMoto Action Engine (TAE) is a C++ implementation of the model described in Section 6.1. Figure 12 shows the structure of the TAE. UMRF graphs and actions are implemented as objects where actions are contained within the graphs. Each action is implemented as a plugin dynamically loaded to the TAE during run-time. The description of an action is contained within a UMRF JSON file (see section 6.2), which is used by the action indexing module to find the actions from the filesystem. Each action is executed in a separate thread, thus enabling concurrent execution. A synchronization module indicates the Action Engine when an action or a graph has finished its execution. The synchronization is also used to send and receive notifications among other actors for synchronized multi-robot task execution.

UMRF graphs are executed by first receiving either the name of the graph to be executed or a complete description of the graph as a JSON string. Algorithm 1 provides an overview of the procedures when a graph is invoked. First, all the actions within the graph are resolved, i.e., a matching plugin or a sub-graph that has the same signature (combination of name, input, and output parameters) is found. If the graph has multiple actors, then a handshake is performed among all actors, ensuring that every actor is ready and executes the graph simultaneously. After a successful handshake, the child actions of the root node are executed, i.e., evolved (see section 6.1.4). This process is iterative, meaning that once the child actions finish execution, the evolution procedure is invoked on their children until the graph exit is reached. Graph modification is done by first receiving a list of modifiers. A modifier outlines which action it is modifying and what the operation is. The modifier can add/remove actions or add/remove edges (child-parent connections). Finally, graph stopping involves invoking the stopping routine for all actions, each in a separate thread. Thus all actions can concurrently de-initialize.

The following subsections cover the steps shown in Algorithm 1, including hierarchical graph resolution (Section 6.3.1), N-to-N handshake (Section 6.3.2), action evolution (Section 6.3.3), and finally the procedures undertaken when executing an action (Section 6.3.4).

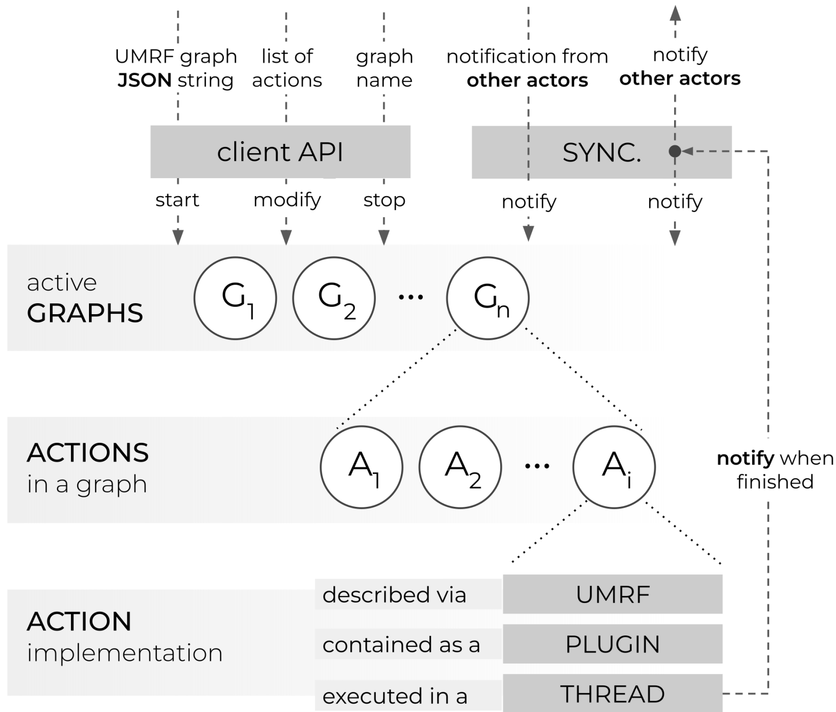


Figure 12. Structure of the TeMoto Action Engine.

6.3.1. Hierarchical Graph Resolution

Graph resolution (Algorithm 2) is the process of finding a matching implementation to each action outlined in a UMRF graph. An action is considered local if the assigned actor in the graph and the name of the Action Engine match. Similarly, the action is considered remote if the assigned actor has a different name than the local Action Engine. If a suitable local action is not found but a graph with the same signature is present, then the particular action is resolved to a hierarchical sub-graph. For hierarchical sub-graphs, the graph resolution is invoked recursively.

Algorithm 1: Overview of the flow of executing, modifying, and stopping a graph.

```
Data:
  • G – data structure embedding the UMRF graph
  • Δ – set of graph modifiers, each modifying an action
1  Function start_umrf_graph(G):
2      resove_actions(G)
3      if G has multiple actors then
4          n_to_n_handshake(get actors in G)
5      end
6      start_children_of(get root of G)
7  Function modify_umrf_graph(G_name, Δ):
8      G ← look up graph G_name
9      pause G
10     for each graph modifier δ in Δ do
11         G = G + δ // add/remove action or add/remove child
12     end
13     continue G
14 Function stop_umrf_graph(G_name):
15     G ← look up graph G_name
16     for each action a in G do
17         stop a                                   ▷ non-blocking via a thread
18     end
19     wait until all actions in G are finished
```

Algorithm 2: Hierarchical graph resolution algorithm, which resolves actions to local, remote, or hierarchical implementations.

```
Data:
  • g – data structure embedding the UMRF graph
  • G – set of pre-indexed/known UMRF graphs

1  for each action a in graph g do
2      if a is remote or has a local implementation then
3          continue
4      end
       // check if the action can be substituted by a
          hierarchical sub-graph
5      for each known graph g' in G do
6          if signature of g' ≠ signature of a then
7              continue
8          end
9          if resove_actions(g') = true then                ▷ recursive call
10             mark a as a hierarchical sub-graph g'
11             break
12         end
13     end
14     if a is unresolved then
15         return false
16     end
17 end
18 return true                          ▷ all actions a in graph G were resolved
```

6.3.2. N-to-N Handshake

The N-to-N handshake is a process where:

- the local actor acknowledges the presence of the other actors,
- the local actor acknowledges that the other actors have acknowledged each other.

This procedure ensures every actor is ready and can simultaneously invoke the UMRF graph.

Algorithm 3 shows the algorithm of the N-to-N handshake. The details of handshake timeout, i.e., ensuring the handshake happens within a set timeframe, are not covered. First, the variables utilized throughout the algorithm are initialized, where, most notably, a set of all possible actor name pair combinations are generated (denoted as Ω). Next, three concurrent procedures are started:

1. continuous broadcasting of handshake messages;
2. continuous checking if the handshake has been reached;
3. a callback function, which processes messages broadcasted by the other actors.

The broadcasted messages contain the local actor's name and a set of remote actor names that the local actor has heard from. When receiving messages from the other actors, entries of local-remote and all remote-remote actor name pairs are added or updated in the buffer (denoted as B). The handshake is considered successful once the sets B and Ω are equal, i.e., B contains the same name pairs as Ω.

6.3.3. Action Evolution

Action evolution (more details in Section 6.1.4) is a procedure that tests if an action should be started, paused, stopped, or set to error given the state of its parent actions and predefined execution conditions. Thus, the action evolution is invoked when an action finishes execution (or when the graph is started). Algorithm 4 shows the main details of what is done before and after the evolution step. First, as the input is an action a_p that has finished its execution, all children of a_p are evolved. If the newly proposed state of the child action remains unchanged after the evolution, the function returns. If the evolution changes the state, the proposed state change is carried out in a separate thread.

6.3.4. Action Execution

As established in Section 6.1.4, an action can be local, remote, or a sub-graph, which defines how an action is executed (Algorithm 5).

Local actions are implemented as plugins, which are loaded and instantiated when the action is executed, i.e., lazy initialization. Then, a "run" routine is invoked on the plugin which returns an outcome of "true" or "false" when the routine is finished. The outcome depends on the plugin's context and is up to the

Algorithm 3: N-to-N handshake algorithm.

Data:

- α – name of self
- Γ – set of remote actor names
- Ω – set of all possible actor-actor name pairs
- B – an initially empty set/buffer of actor-actor name pairs

```
1  Function n_to_n_handshake(Γ):
2      handshake_reached ← false
3      B ← ∅
4      Ω ← generate all possible pairs of ⟨α,Γ⟩, ⟨Γ,α⟩, ⟨Γ,Γ⟩ |∀γ(¬Ω(γ,γ))
5      listen_callback ← register
       // broadcast handshake messages
6      concurrent: while (handshake_reached = false):
7          msg ← initialize                          ▷ message to be broadcasted
8          msg.actor ← α
9          for each γ in Γ do
10             if ⟨α,γ⟩ ∈ B then
11                 msg.handshakes ← append γ
12             end
13         end
14         broadcast msg to all actors Γ;
15     end
       // check if every actor has heard from everyone
16     while handshake_reached = false do
17         if B = Ω then
18             handshake_reached ← true
19         end
20     end
21     return handshake_reached
   // register incoming handshake messages
22 Callback listen_callback(msg):
23     if msg.actor = α then
24         return                                    ▷ ignore own messages
25     end
26     γ ← msg.actor
27     B ← insert ⟨α,γ⟩                              ▷ α has heard from γ
28     for each actor γ′ in msg.handshakes do
29         B ← insert ⟨γ,γ′⟩                         ▷ γ has heard from γ′
30     end
```

Algorithm 4: Algorithm of evolving child actions of a_p after its execution.

Data: a_p – parent action

```
1  for each child action a of a_p do
2      A_p ← get all parents of a
3      s ← a.get_state()                              ▷ get the current state
4      s′ ← a.evolve(A_p)                               ▷ get the new state
5      if s = s′ then
6          return
7      end
8      switch s′ do
9          case s′ = s_r do: a.run()                   ▷ non-blocking via a thread
10         case s′ = s_p do: a.pause()                 ▷ non-blocking via a thread
11         case s′ = s_s do: a.stop()                  ▷ non-blocking via a thread
12         case s′ = s_e do: a.set_to_error()
13     end
14 end
```

developer to utilize this information when designing graphs. The outcome is then passed to the synchronization system shown in Figure 12.

Remote actions are executed by a remote actor, thus the local actor waits until remote execution is finished. The waiting procedure is implemented by putting the thread to sleep and waking it up (via condition variables) when a notification is received.

Hierarchical sub-graphs are executed by invoking the graph execution routine shown in Algorithm 1. Then, the thread sleeps until it is woken up by receiving a notification, similar to waiting for remote actions to finish.

Finally, after the action has finished, the action evolution event is triggered on the child actions, as described in Section 6.3.3. Thus, the action evolution and action execution form a loop, which iterates through the graph until the graph exit is reached.

Algorithm 5: Running an action depending on whether it's local, remote, or a hierarchical sub-graph.

```
Data: a – action to be run
1  if a is a local action then
2  |   a.load_plugin()
3  |   a.state ← a.run()                           ▷ blocking call
4  |   send outcome to other actors
5  end
6  if a is a remote action then
7  |   a.state ← wait for remote outcome           ▷ blocking call
8  end
9  if a is a graph g_a then
10 |   invoke graph g_a
11 |   a.state ← wait for outcome of g_a           ▷ blocking call
12 |   send outcome to other actors
13 end
14 start_children_of(a)
```

6.4. Requirements Conformance

Table 12 provides a qualitative analysis of how the design of the executive layer, i.e., TeMoto Task Management, conforms to the requirements defined in Table 4.

Table 12. TeMoto Task Management conformance to executive layer requirements (Table 4)

Requirement	TeMoto design feature
$\mathbf{E}_{\mathbf{C\text{-}1}}$: *Actions and tasks can be dynamically invoked, reconfigured, and stopped.*	TeMoto Action Engine enables invoking, modifying, and stopping actions during run-time.
$\mathbf{E}_{\mathbf{C\text{-}2.1}}$: *Reusable across multiple task domains.*	UMRF notation and TeMoto Action Engine are not constrained to any specific domain, such as manipulation- or navigation-only tasks.
$\mathbf{E}_{\mathbf{C\text{-}2.2}}$: *Extendible with new actions and tasks.*	Actions are implemented as dynamically invokable plugins isolated from the main codebase. Tasks can be outlined in a JSON-based DSL, i.e., UMRF.
$\mathbf{E}_{\mathbf{C\text{-}2.3}}$: *Maintainable, i.e., implementation of actions, tasks, and executive layer is decoupled.*	Actions are implemented as dynamically invokable plugins isolated from the main codebase. Tasks can be outlined in a JSON-based DSL, i.e., UMRF.
$\mathbf{E}_{\mathbf{C\text{-}2.4}}$: *Middleware independent.*	TeMoto Action Engine is implemented as a middleware-independent C++ library, which can be wrapped with a middleware layer of choice, such as glsros or glsros2.

Continued on next page

Table 12 – continued from previous page

Requirement	TeMoto design feature
E$_{C\text{-}3}$ :: *Individual actions in a shared task can be attributed to specific decentralized actors*	UMRF notation enables attribution of actions to specific actors. TeMoto Action Engine provides full support for decentralized multi-actor task execution, including parameter sharing.
E$_{C\text{-}4}$: *Interaction modalities are decoupled from the implementation of the executive layer via a Domain Specific Language.*	Tasks are outlined in UMRF format, a JSON-based DSL that decouples the implementation-specific details from command sources.
E-1 : *Expressive task composition, including sequences, concurrency, cycles, and statements.*	The UMRF notation enables the design of tasks with sequential, concurrent, and cyclical elements, as well as design statements, by utilizing the condition mechanism outlined in Section 6.1.3. TeMoto Action Engine provides full support for expressive tasks.
E-2 : *Input and output parametrization of granular tasks.*	The UMRF notation enables the definition of input and output parameters for actions, as well as the passing of parameters between actions. TeMoto Action Engine provides full support action parameterization.
E-3 : *Continuous/reactive behaviors.*	Reactive behaviors,. i.e., spontaneous actions, are supported in UMRF. Spontaneous actions are restarted automatically after finishing. When such action finishes, it invokes the evolution procedure (see Section 6.3.3) in child actions.
E-4 : *Hierarchical task composition.*	The UMRF notation is implicitly hierarchical, meaning that if an action within a graph has no implementation as a plugin but is defined as a graph with the same signature, then the hierarchical graph is executed (see Section 6.3.1)

6.5. Summary

This section covered the fundamental design and implementation of the Task Management layer in the TeMoto framework. To conclude:

- Task Management layer is responsible for grounding task descriptions, which originate from layers above, into executable behaviors.
- Tasks are outlined as a combination of modular behaviors called actions. Such a combination of actions is also called a UMRF graph.
- UMRF is a DSL designed for HRI- and MRC-oriented use cases. The UMRF notation allows for describing complex tasks independent of the task domain while having a clearly defined and easily adoptable serialized form via JSON.
- TeMoto Action Engine is a C++ implementation of the UMRF semantics, supporting the design of sequential, concurrent, cyclical, parametrized, reactive, hierarchical, and multi-agent tasks.

7. TEMOTO – RESOURCE MANAGEMENT

A multifunctional robot, deployed to perform various tasks during the same mission, may require different resources per task (e.g., sensors, actuators, data processing algorithms). Furthermore, the set of resources can change even within the same task, due to unexpected failures that require the resource to be reconfigured or replaced with an alternative resource(s). Thus, the lifecycle of a resource is dynamic and depends on the mission's strategy, which may include energy conservation (minimizing the number of active resources) and adaptiveness (reconfigure or substitute resources). In a publisher-subscriber-based data distribution architecture the resources are potentially shared by multiple clients/consumers, and a resource may depend on sub-resources, making the resource itself a client. Therefore, given the dynamic, shared, and hierarchical nature of resources, an accounting mechanism is required for:

- **Reference counting** to assure initialization when a resource is first acquired and deinitialization when the last client releases the resource.
- **Hierarchical dynamic dependency management**, to register all sub-resource dependencies on initialization, release all sub-resources on deinitialization, and provide a lookup table for propagating resource state updates (e.g., failures) along the hierarchy.

To better illustrate the use cases of resource management, imagine an autonomous inspection task where the robot is required to navigate and visually inspect the environment until specific conditions are met, e.g., a hazard is detected. The robot is equipped with a sensor (e.g., 3D LIDAR sensor) to perform inspection and collision-free navigation simultaneously. Figure 13a shows the action graph of such a mission, where the mission starts by running the navigation (A_N) and inspection (A_I) tasks in parallel. The navigation action runs indefinitely until externally stopped by the inspection action. Figure 13b shows the sequence diagram of how both actions acquire resources. While the order of resource acquisition is non-deterministic, let us say the inspection action A_I is first to request the LIDAR. Next, the navigation action A_N requests the whole robot as a single resource, which hierarchically requests the LIDAR and the mobile base. Now, the LIDAR is not re-initialized, as it has already been done earlier, only the user count is increased.

Figure 13c shows how, in case of a potential resource failure, e.g., failure of the aforementioned LIDAR, the information about the event is distributed to all consumers that are either directly or hierarchically related to the failed resource. Figure 13d shows the resource unload sequence, where the LIDAR is uninitialized only after the navigation action A_N and inspection action A_I have both released it, as well as how hierarchical resources are released.

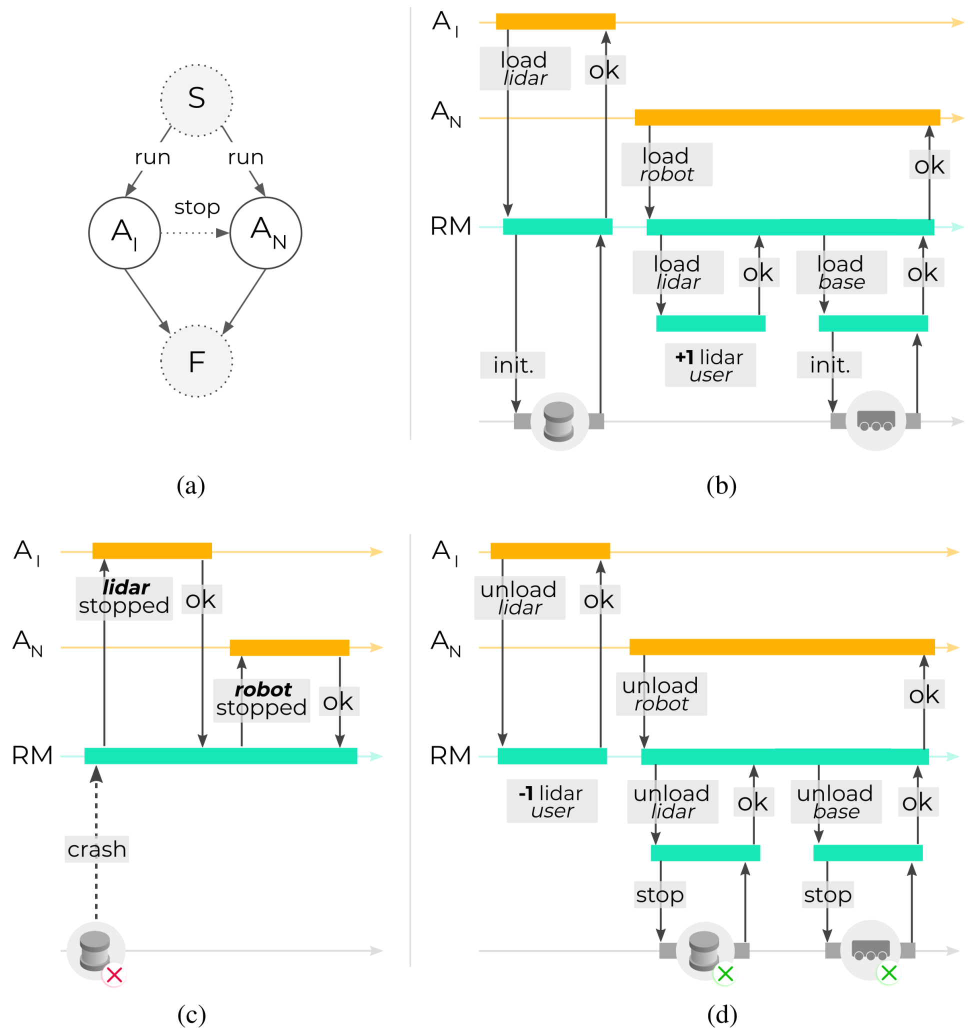


Figure 13. Action graph of the autonomous inspection task (a), which demonstrates the cases of dynamic initialization (b), error propagation (c), and un-initialization (d) of shared and hierarchical resources, all of which are dependent on resource management's reference and hierarchical dependency accounting functionalities.

7.1. Definitions and Semantics

A *resource*, e.g., a sensor, actuator, or a data processing algorithm, is something that is loaded upon a request by a resource *provider* to a resource *consumer* (Figure 14a). The resource *query* is described by a combination of *request* and *response* data structures, where the request outlines the details of the required resource, and the response contains the attributes of the loaded resource (ID and any other resource-specific information). Resources can have multiple consumers (e.g., multiple clients subscribing to the same sensor stream) and the resource can hierarchically consume multiple sub-resources (Figure 14b). Each consumer, provider, and loaded resource has a unique ID that is used to handle resource queries and relations. The following subsections cover the details of a resource request, release, and updates.

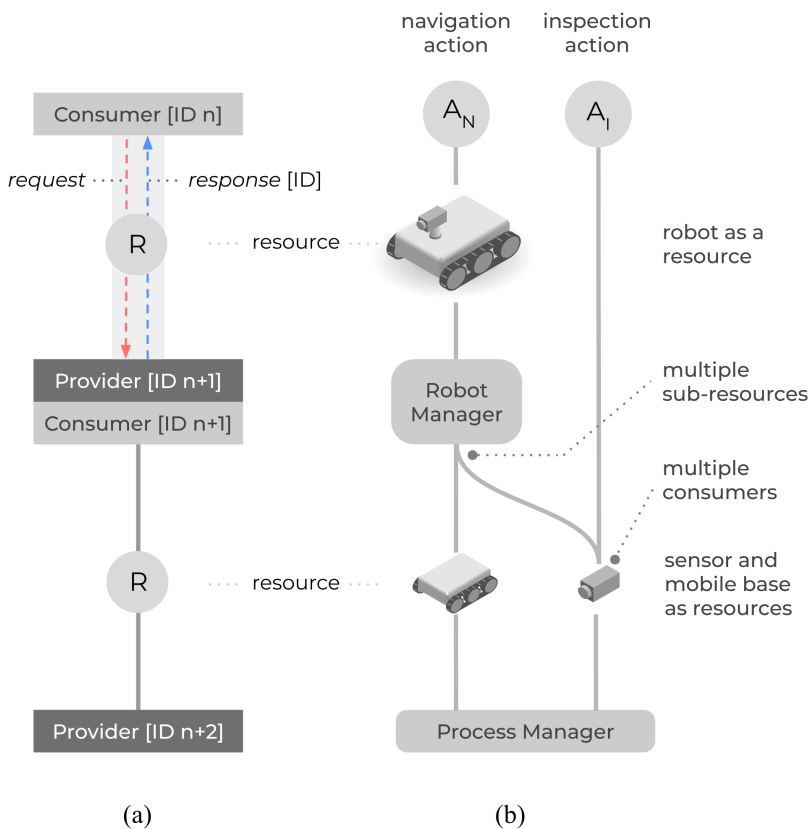


Figure 14. Abstract structure of the provider-resource-consumer relation in resource management (a), and an example of hierarchical resources, and multi-resource consumer use-cases (b). Including Robot Manager and Process Manager illustrate this specific example and are not conceptual elements in resource management.

7.1.1. Resource Request

Requesting a resource involves two principal procedures: consumer counting and resource loading (Algorithm 6). The resource is loaded when it is requested for the first time, with subsequent requests only increasing the consumer count. The specific loading procedure depends on the resource, and it may hierarchically perform sub-resource requests from other resource providers. In this case, the sub-resources are associated with the initial request. A response data structure is returned to the consumer when the loading is complete. If the same resource is requested again while already loaded, the previous response is immediately returned.

Algorithm 6: Resource request procedure.

Data:
- id_C – ID of the consumer
- *req* – Data structure describing the desired resource

Result:
- id_{req} – ID of the initialized resource
- *res* – Data structure describing the details of the loaded resource

```
1  Function request_resource(id_C, req):
2      if req already loaded then
3          Append id_C to the list of consumers of req;
4          < id_req, res > ← Fetch the resource ID and response of req;
5      else
6          id_req ← Generate unique resource ID;
7          res ← Invoke the loading routine specific to req;
8          for each requested sub-resource id_sub when loading req do
9              Append id_sub as a dependency of id_req;
10         end
11         Register a new resource entry < id_req, req, res > with id_C as a
             consumer;
12     end
13     return id_req, res
```

7.1.2. Resource Release

Resource release is when the consumer no longer requires the resource and notifies the provider, as shown in Algorithm 7. Similar to resource requests, as the same resource can be shared with multiple consumers, the unloading procedure is invoked when all consumers have released the resource. In that case, a custom/subsystem-specific resource unloading routine is invoked. If the resource has any sub-resources, then these are released as well.

Algorithm 7: Resource release procedure.

Data:

- id_C – ID of the consumer
- id_R – ID of the loaded resource

```
1 Function release_resource(id_C, id_R):
2     Remove id_C from the list of consumers of the resource with id_R;
3     if resource associated with id_R has no consumers then
4         req ← Fetch the request associated with id_R;
5         Invoke the unloading routine specific to req;
6         for each sub-resource dependency id_sub of req do
7             release_resource(id_R, id_sub);
8         end
9     end
```

7.1.3. Resource Status Update

Resource status update is a procedure invoked by the resource provider whenever the status of the resource has changed, including resource failures. After receiving the update message (Algorithm 8), the consumer invokes a custom status update callback (if registered). If the consumer is also a provider (see Figure 14), then the update is passed to the other consumers of the super-resource.

7.2. Implementation

Figure 15 shows how resource consumers and providers are implemented (hierarchical resource management is not depicted) in the TeMoto framework. Consumers and providers are segregated into separate operating system processes, and data is exchanged via inter-process communication, relying on middleware, such as ROS. The procedures discussed in Section 7.1, i.e., resource allocation, release, and status updates, are embedded into a module named Resource Registrar (RR) (more details about RR in II and [87]). Thus, the consumer utilizes RR to mediate the request, release, and status update procedures, while the provider utilizes RR to trigger the respective resource loading and unloading procedures via callback functions. The internal database of RR is backed up during each request thus the respective subsystem that hosts RR can be restored after a critical failure. In most cases, the developer, who designs the consumer application, does not need to be exposed to the API of RR. Thus a simplified interface is provided that wraps the RR and middleware-specific details. The interface is specific to each provider, shown in figure 16, where the implementation of a provider is also referred to as a *resource manager*. Table 13 gives an overview of the resource managers implemented in the TeMoto framework.

As depicted in Figure 7, each robot, i.e., a TeMoto instance, can have an individual set of resource managers deployed. The Resource Info Synchronizer (RIS)

Algorithm 8: Resource status update procedure.

Data:

- id_R – ID of the allocated resource
- msg – Status message

`// Initiated by a resource provider`

1 **Function** `send_status_update(`id_R, msg`)`**:**
2   **for** *each consumer id_C of resource id_R* **do**
3   Send msg to consumer id_C;
4   **end**

`// Received by a resource consumer`

5 **Function** `receive_status_update(`id_R, msg`)`**:**
6   **if** *custom status update callback is assigned* **then**
7         $< req, res > \leftarrow$ Fetch the query associated with id_R;
8   Invoke the custom status update callback with arguments $< req, res, msg >$
9   **end**
10   **if** *resource id_R is a sub-resource of id'_R* **then**
11   `send_status_update(`id'_R, msg`)`
12   **end**

provides tools for sharing resource-related information between robots. RIS allows the sharing of resource information between different instances of the same manager. This allows one robot to use another robot's resources, giving an extra level of dependability and flexible use of resources. The collaborative utilization of robotic resources via RIS provides the system developer additional flexibility when designing robotic applications, which in similar scenarios would need to rely on custom and non-standard solutions. Figure 17 shows the working principle of RIS. When a resource manager is initialized, it advertises the available resources, e.g., R_0 in Figure 17a, through RIS. The advertised message is captured by the RIS of the other robot's resource manager. Now if the other robot wants to load R_0 (Figure 17b), the request is automatically redirected to the actual provider of R_0.

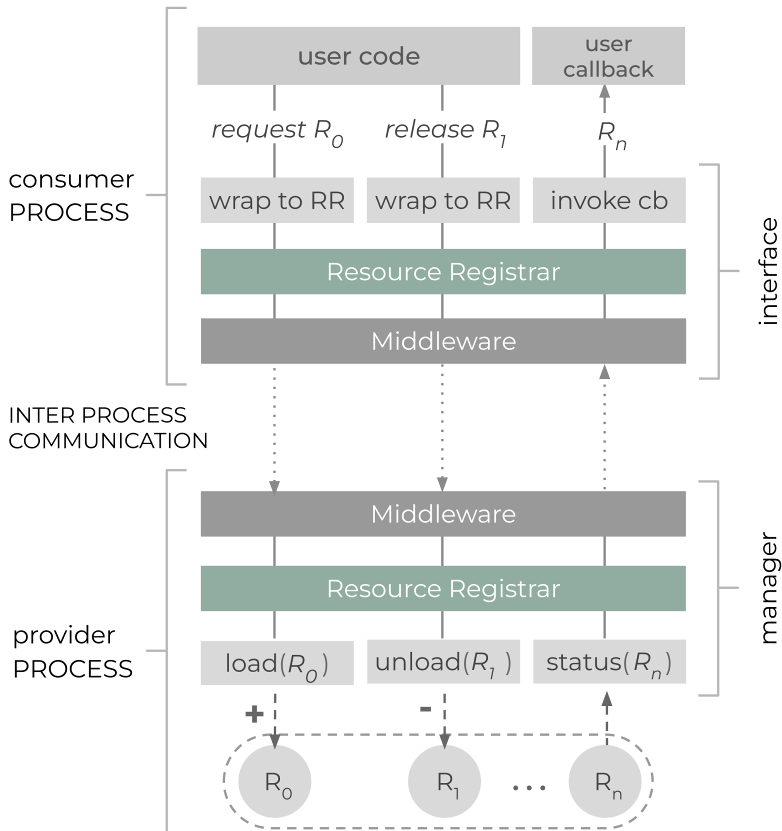


Figure 15. Implementation of resource consumers and providers in TeMoto framework. Hierarchical resource management is not depicted.

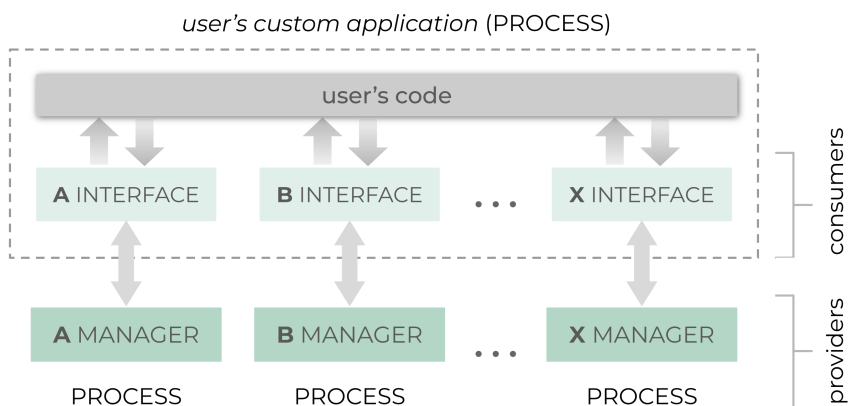


Figure 16. Integration of resource consumers in user's custom application. Each resource manager can be accessed via the corresponding interface, which provides a simplified API.

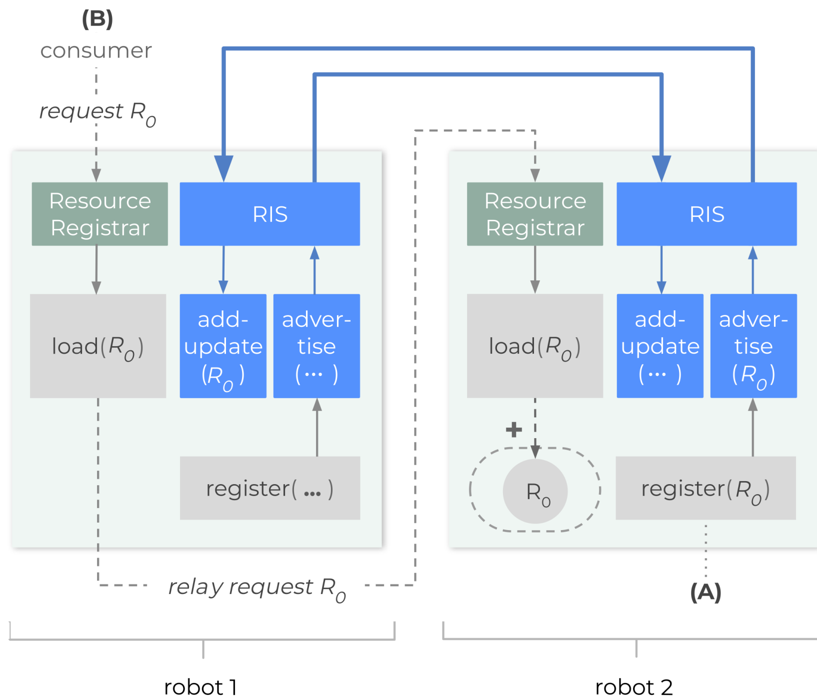


Figure 17. Working principle of RIS. When a resource manager is initialized, it advertises the available resources, e.g., R_0, through RIS (a). The advertised message is captured by the RIS of another robot's resource manager. Now if the other robot wants to load R_0, the request is automatically redirected to the actual provider of R_0 (b).

Table 13. Overview of the resource managers implemented in the TeMoto framework.

TeMoto Resource Manager	Description
Process Manager[1]	Can dynamically invoke programs, including ROS executables, ROS launch files and regular executables. Running programs are then monitored for process failures (via segmentation faults, unexpected shutdowns, etc), which are reported to the consumers of the failed resource.
Visualization Manager[2]	Uses RViz as the main platform for visualization, where each displayable data type is displayed via dynamically loadable RViz plugins. Thus, data can be visualized programmatically on demand.
Component Manager[3]	Maintains information about components (including published/subscribed topics, and package names) and can dynamically compose them based on component templates. Components are ROS-based programs (nodes/launch files) which provide sensing or data processing functionalities. A component template outlines the structure of a composed component, e.g., a combination of a manipulator arm and a force-torque sensor with a compliance controller. Templates can then be used to dynamically combine individual components and replace faulty ones.
Context Manager[4]	Context Manager maintains a model of the world as hierarchical relations between maps and world objects in a tree structure. The world model is shared with other robots (other instances of Context Manager allow to build and utilize a common world model in collaborative tasks.
Robot Manager[5]	Robot Manager maintains information about robotic manipulators, grippers, and mobile bases and can dynamically provide access to all drivers comprising a single robot through a unified interface, i.e., Robot Manager Interface. A single device or an assembly of robotic devices composes a robot resource.

[1] github.com/temoto-framework/temoto_process_manager
[2] github.com/temoto-framework/temoto_visualization_manager
[3] github.com/temoto-framework/temoto_component_manager
[4] github.com/temoto-framework/temoto_context_manager
[5] github.com/temoto-framework/temoto_robot_manager

7.3. Requirements Conformance

Table 14 provides a qualitative analysis of how the design of the functional layer, i.e., TeMoto Resource Management, conforms to the requirements defined in Table 5.

Table 14. TeMoto Resource Management conformance to functional layer requirements(Table 5).

Requirement	TeMoto design feature
$\mathbf{F_{C\text{-}1}}$: *Resources can be dynamically invoked, reconfigured, and stopped.*	A service-based resource acquisition and release module, Resource Registrar, along with resource managers supporting a variety of resource types.
$\mathbf{F_{C\text{-}2.1}}$: *Reusable across multiple task domains.*	The core features of resource management, RR and RIS, have no underlying assumptions of the deployment-specific details or domains.
$\mathbf{F_{C\text{-}2.2}}$: *Extendible with new resources.*	The core features of resource management, RR and RIS, allow developers to register arbitrary resource types.
$\mathbf{F_{C\text{-}2.3}}$: *Maintainable, i.e., implementation of resources and the functional layer is decoupled.*	At the lowest level, a resource is an OS process, meaning that changes in the implementation of the resource do not affect the resource management. Also, the dynamic resource accounting, i.e., the Resource Registrar, is segregated from resource allocation, i.e., specific resource managers (see Table 13), which facilitates better maintenance of the code base.
$\mathbf{F_{C\text{-}2.4}}$: *Middleware independent.*	Resource management follows modularized design, where commonly utilized modules are implemented independent of any middleware, which in turn are wrapped with middleware specific wrapper code.
$\mathbf{F_{C\text{-}3}}$: *Information about a resource can be shared.*	The RIS module (Section 7.2) allows resource managers on different robots to share information about resources.
F-1, **F-2**, **F-3** : *Supports multiple consumers; Supports hierarchies; Resource accounting (consumers and dependencies) information must be recoverable.*	RR supports resource reference counting, dependencies among resources, and the state of RR is recoverable from a backup file.

7.4. Summary

This section covered the fundamental design and implementation of the Resource Management layer in the TeMoto framework. To conclude:

- The resource management layer, which corresponds to the functional layer in the three-tier architecture (Section 2.3.3), provides dynamic access (start, stop, reconfigure) to resources while accounting for multiple resource allocations (reference counting) and resource dependencies/hierarchies.

- Specific to the implementation of TeMoto, the resource management layer contains multiple individual resource managers (OS processes), each with a specific set and scope of resources they are managing.
- The Resource Registrar (RR) is a C++ based module, embedded into each resource manager, which supports resource reference counting, dependencies among resources, and the state of RR is recoverable from a backup file.
- The Resource Information Synchronization module (Section 7.2) allows resource managers on different robots to share information about resources.

8. EVALUATION

Following the qualitative assessment of this work (Table 9, 12, and 14) this section covers a set of practical demonstrators that show TeMoto's eligibility as a software backbone for adaptive robotic applications. Each demonstration provides a description of the configuration of TeMoto, as well as the list of fulfilled requirements (Section 3).

8.1. Demo 1 – Fault Tolerant Sensor Redundancy

This demo (Table 15) evaluates the viability of dynamically loading resources to enable redundant and fault-tolerant robots. The underlying scenario depicts a remote inspection mission where the operator uses visual sensor feedback. The operator's situational awareness is challenged by incidents that compromise the robot's sensors. As a countermeasure, each resource failure is addressed by utilizing an alternative onboard sensor.

The robot, equipped with a set of sensors (3D LIDAR, 2D LIDAR, depth camera), is initialized with 3D LIDAR being the primary active sensor, while the other sensors remain inactive to minimize power consumption (Figure 18a). The robot is teleoperated until a critical failure is introduced to the primary sensor. The failure is then registered by TeMoto, which automatically switches over to a new primary sensor (Figure 18b), i.e., depth camera, based on a user-defined priority list. The new primary sensor is automatically visualized in the Operator Control Station (OCS) via Visualization Manager. Similarly, after continuing a fault is introduced to the 2D LIDAR, making TeMoto reconfigure for a 2D LIDAR (Figure 18c).

In addition to demonstrating the functional aspects of dynamic resource management, this demo also exemplifies the aspect of energy conservation enabled by dynamic resource management. If the same experiment was repeated without dynamic resource management, meaning that all sensors (and respective visualization setup in the OCS) were initialized during startup (as opposed to being initialized only when actually needed), then the OCS and the robot consumed 12

8.2. Demo 2 – Multi-Robot Escort

This demo (Table 16) evaluates the resource synchronization features of TeMoto in a remote inspection mission where an operator has visual feedback from the robot's camera. Visual feedback is severed but automatically compensated by a nearby robot which escorts the compromised robot back to the operator.

The teleoperated robot, i.e., the Worker, is equipped with a 2D camera, which provides visual feedback to the operator (Figure19a). The robot is teleoperated until a critical failure is introduced to the camera (Figure19b). The failure is then registered by TeMoto, which automatically looks for other robots (the Escort) that

Table 15. Overview of the fault-tolerant sensor redundancy demo.

Objective:	Demonstrate dynamic resource management
Utilized tools:	Action Engine, Component Manager, Robot Manager, Visualization Manager, Process Manager
Agents:	Robot and the OCS
Evaluates requirements:	$F_{C\text{-}1}$
Code:	`github.com/temoto-framework-demos/temoto_architecture_demos`
Video:	`youtube.com/watch?v=4AxX6-BUUlw`
Original publication:	II

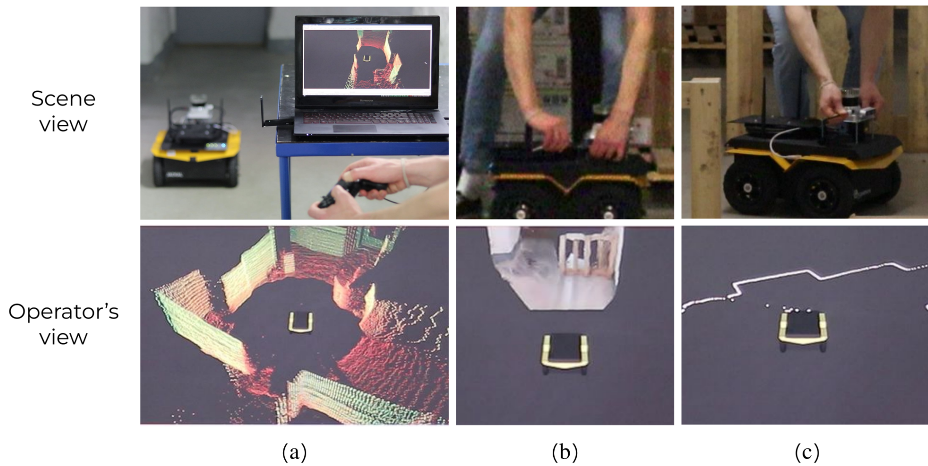


Figure 18. Overview of demo 1, i.e., fault-tolerant sensor redundancy experiment. Visual feedback from the 3D LIDAR (a). Once compromised, feedback is dynamically regained via a depth camera (b), and then a 2D LIDAR (c).

could provide substitute visual feedback and follow the Worker via AR-tag (Figure19c). This is achieved by the OCS issuing a query to the Component Manager, requesting information about other registered robots that have a 2D camera. With a positive response, the OCS asks the Escort to initialize its camera and follow the Worker. The operator still controls the Worker, but visual feedback is received from the Escort, effectively yielding a third-person view of the Worker.

8.3. Demo 3 – Mission Adaptiveness

This demo (Table 17) evaluates how robots can be adapted for new tasks described by a UMRF graph. The scenario depicts a mobile robot autonomously surveying an area as its “day job” until the robot is retasked to deliver goods, and transferred from another robot. After completing the delivery, the robot resumes the surveillance task.

Table 16. Overview of the multi-robot escort via resource redundancy demo.

Objective:	Demonstrate resource synchronization features
Utilized tools:	Action Engine, Component Manager, Robot Manager, Visualization Manager, Process Manager
Agents:	Worker (Clearpath Jackal), Escort (Robotont), OCS
Evaluates requirements:	$F_{C\text{-}1}$, F-1, F-2, F-3
Code:	`github.com/temoto-framework-demos/temoto_architecture_demos`
Video:	`youtube.com/watch?v=DgODvDsWBcY`
Original publication:	II

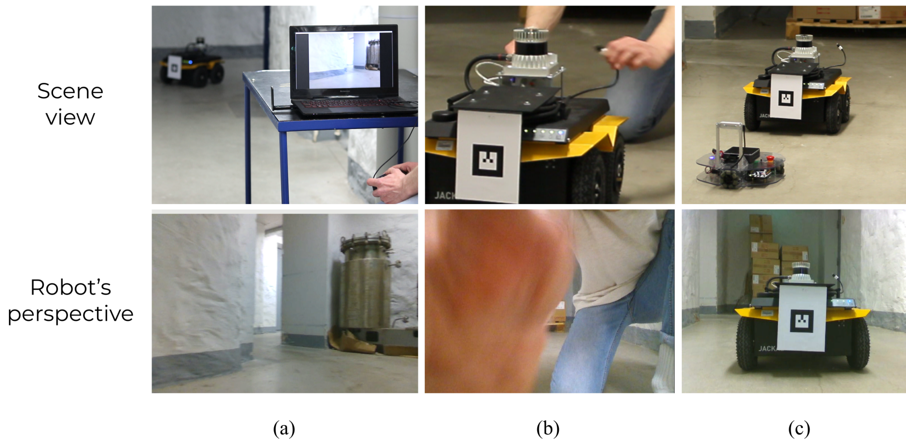


Figure 19. Overview of demo 2, i.e., multi-robot escort via resource redundancy experiment. The operator teleoperates the Worker via OCS and sees the visual feedback from the Worker's camera (a). Worker's camera fails (b). The Escort robot is instructed to follow the Worker and provide feedback to the operator allowing them to retrieve the Worker which would otherwise be left in the field (c).

Demo 3 involves three actors: Jackal, Ufactory xArm7, and the OCS. The Jackal utilizes 2D LIDAR and Gmapping Simultaneous Localization and Mapping (SLAM) to localize in its environment. TeMoto's Robot Manager controls Jackal's navigation and xArm7's manipulation functionalities. The OCS stores the mission plans as UMRF graphs where the first graph (Figure 20a) instructs the Jackal to cyclically navigate through three different locations and the second graph (Figure 20b) coordinates the Jackal and xArm7 to deliver two objects.

Task 1 starts when the operator sends the surveillance UMRF graph to the OCS's Action Engine. The graph contains three navigation actions, each preconfigured for a specific location via UMRF input parameters. The navigation action internally instructs the Robot Manager to move the Jackal. Similar to Demo 2 the OCS's Robot Manager knows the mobile platform is managed in Jackal's TeMoto instance, thus relaying the navigation command to Jackal. After a few cy-

cles of surveillance, the operator invokes the delivery UMRF graph. This UMRF graph instructs the Jackal to navigate to the pick-up location while initializing xArm7's ROS driver and controller via Robot Manager. After the parallel actions are finished, the xArm7 is instructed to transfer two objects to the Jackal. The Jackal is then instructed to navigate to a predefined delivery point. After the delivery task, the robot continues the surveillance task.

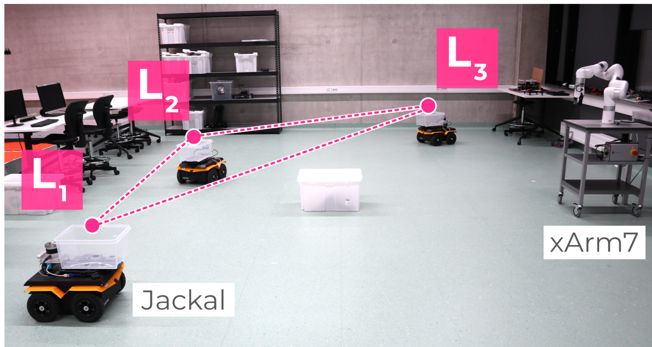
Table 17. Overview of the fault-tolerant sensor redundancy demo.

Objective:	Demonstrate dynamic task management
Utilized tools:	Action Engine, Component Manager, Robot Manager, Process Manager
Agents:	Clearpath Jackal, xArm7, OCS
Evaluates requirements:	$E_{C\text{-}1}$, E-1
Code:	`github.com/temoto-framework-demos/temoto_architecture_demos`
Video:	`youtube.com/watch?v=Lze2jsyeN74`
Original publication:	II

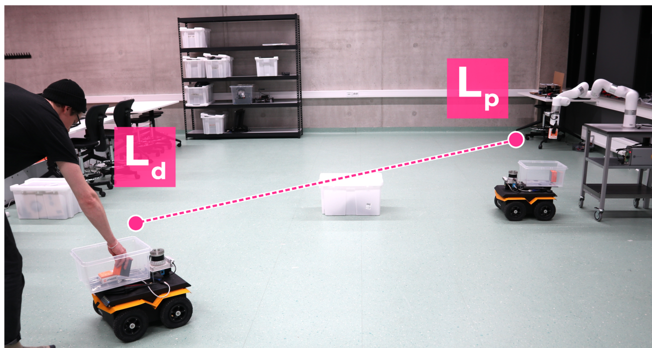
8.4. Demo 4 – LLM Driven Task Planning

This demonstration (Table 18) exemplifies the source agnosticism of UMRF-based task management. A LLM is utilized to generate UMRF graphs from multimodal input that contains natural language with metadata from an augmented reality headset. The generated graph is forwarded to the robot, which performs an inspection task, combining navigation, manipulation, and image acquisition actions.

The demonstration depicts a remote inspection scenario, where a mobile manipulator robot (Clearpath Husky + two Universal Robots UR5s), equipped with a camera (Intel RealSense D435) attached to the end-effector, has to navigate to and inspect specific areas defined by the operator. The operator is equipped with an Augmented Reality (AR) headset (Microsoft HoloLens 2) that is able to capture voice commands (Figure 21a) and allows defining goal locations via gesture-operated virtual markers (Figure 21b). Inspection and task execution feedback is overlaid to the operator's field of view in real-time (Figure 21c). The Azure Spatial Anchors plugin was used on HoloLens to allow the robot and AR devices to co-localize and share the same reference frame. Target poses are generated by spawning a coordinate frame in the world, and dragging it to the desired pose. The voice commands and location of the virtual marker are concatenated as a string, which is passed to the UMRF parser that constructs a full prompt (Figure 22). Each prompt embeds five operator command + UMRF graph pair examples, which then is sent to OpenAI via openai v.0.25.0 Python API. The UMRF JSON string, returned by OpenAI, then is sent to the Action Engine hosted on the robot via a ROS message.



(a)



(b)

Figure 20. Overview of Demo 3, i.e., mission adaptiveness experiment. The deployed robot (Jackal) performs a surveillance task, passing through locations L_1 to L_3 (a). Then the Jackal is asynchronously re-tasked to deliver objects from location L_p to L_d.

Figure 21 shows an example where the operator places the virtual marker near a valve and says "Robot inspect the lab". The combined input along with few-shot examples are sent to OpenAI, which returns a UMRF graph that first navigates the robot close to the valve; moves the end-effector with a mounted camera to the target pose; and finally captures an image.

Table 18. Overview of the LLM-driven task planning demo.

Objective:	Demonstrate UMRF source agnosticism
Utilized tools:	Action Engine, Component Manager, Robot Manager, Process Manager
Agents:	Dual arm (Universal Robots UR5) Clearpath Husky
Evaluates requirements:	$E_{C\text{-}1}$, $E_{C\text{-}4}$
Code:	`github.com/temoto-framework-demos/gpt_temoto_demo`
Video:	`youtube.com/watch?v=E3RwfcX1KpM`
Original publication:	[88]

8.5. Demo 5 – Multi-Robot Environment Knowledge Sharing

This demo (Table 19) shows how a decentralized knowledge representation framework (TeMoto Context Manager) can be used in heterogeneous multi-robot tasks. The scenario depicts a clean-up task, where two robots (scouts) are assigned to find objects of interest and the worker is assigned to manipulate the found objects.

The underlying framework shown in this demo could be adapted to functionally similar real-world scenarios in search-and-rescue or disaster response domains. For example, a fleet of drones can cover a wide area in search of people, pockets of fire, etc, and heavy machinery can then be optimally deployed for immediate action.

The scenario starts with all robots located in a common area on a predefined map where each robot is able to localize (Figure 23a). Then the scout robots (Robotont platform with a Realsense D435 camera) are tasked to look for objects of interest, each deployed to a separate location in the environment (Figure 23b). The object detection was implemented via AR-Tag detection, where each tag was a priori associated with a certain type of object and then attached to it (trash, bin). When an object is detected, the respective scout adds an entry about the object to the Context Manager, outlining information about its location with respect to the map. The information about the new entry is automatically distributed among all TeMoto instances (robots). Next, the worker is teleoperated to the location of the trash (Figure 23c), picks it up (adjusts the entity relation from "map $\rightarrow$ trash" to "map $\rightarrow$ worker $\rightarrow$ trash"), navigates to the bin (Figure 23d), and drops the trash to the bin (adjusts the entity relation from "map $\rightarrow$ worker $\rightarrow$ trash" to "map $\rightarrow$ bin $\rightarrow$ trash").

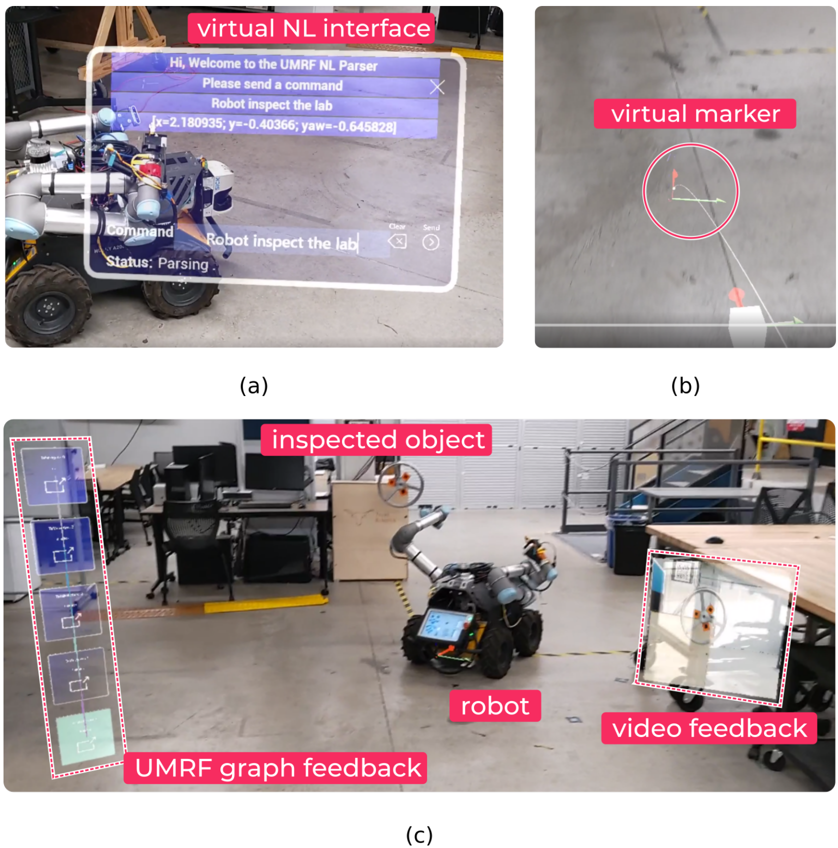


Figure 21. LLM driven task planning scenario in the operator's AR headset perspective. The operator issues commands via speech (a) and virtual marker (b) based interface. The command is then sent to OpenAI, which returns a UMRF graph that is visualized on the AR interface and executed on the robot (c).[88]

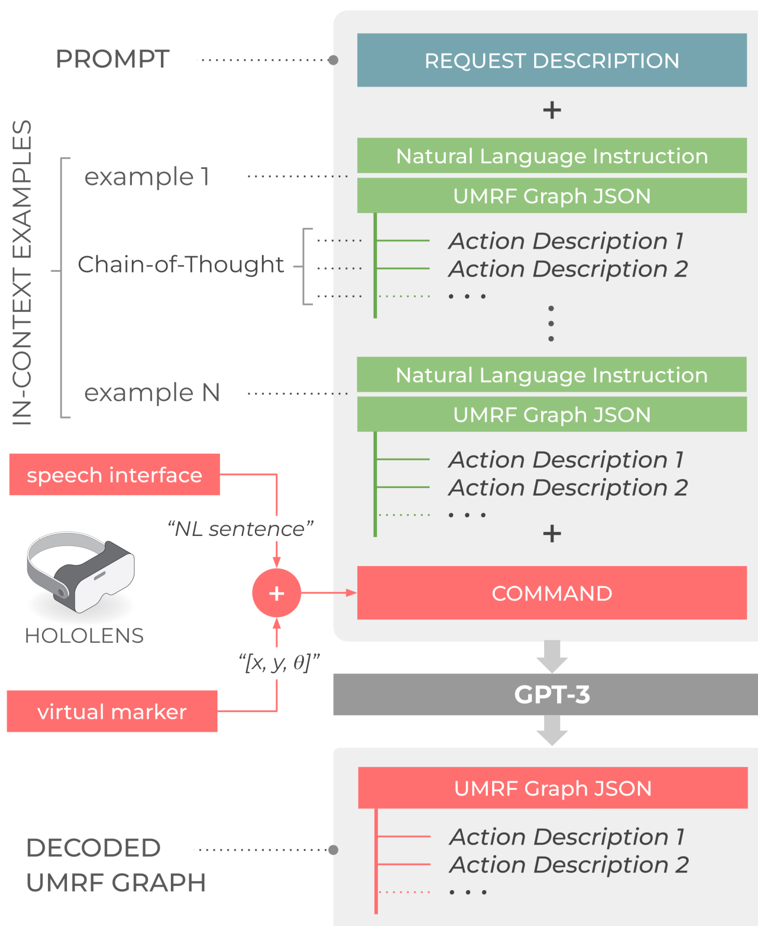


Figure 22. Prompt that is designed to make GPT-3 extract UMRF graphs from a combined input containing voice commands and virtual marker coordinates. The beginning of the prompt contains the description of the request. Each prompt also embeds five operator command + UMRF graph pair examples. Finally, the voice commands and location of the virtual marker are concatenated as a string and added to the prompt. The prompt is then passed to GPT-3, which returns a UMRF JSON string.[88]

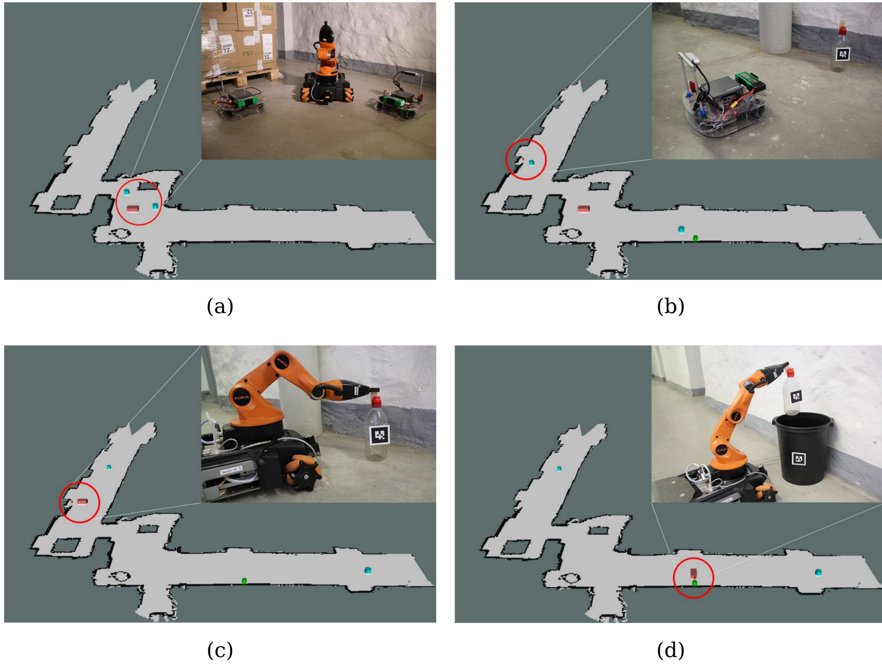


Figure 23. Multi-robot environment knowledge sharing scenario, where three robots are used to locate and manipulate objects of interest (a). First the two scout robots dynamically locate the objects (trash and a trash can) and share the information to all agents (b). Then the worker robot, knowing the location of the objects, picks up the trash (c) and places it in the trash can (d).

Table 19. Overview of the multi-robot knowledge sharing demo.

Objective:	Demonstrate multi-robot knowledge sharing
Utilized tools:	Context Manager, Action Engine, Component Manager, Robot Manager, Process Manager
Agents:	Two mobile robots (Robotont) and a mobile manipulator (youBot)
Evaluates requirements:	$F_{C\text{-}1}$, $F_{C\text{-}3}$
Code:	`github.com/temoto-framework-demos/temoto_examples`
Video:	`youtube.com/watch?v=FsC-puN8q9w`
Original publication:	[89]

8.6. Summary

The demonstrations showcased how TeMoto facilitates the design of robotic applications that can:

- dynamically adapt to citical resource failures (8.1),
- utilize multi-robot systems for increased redundancy (8.2) and joint task execution (8.3),
- dynamically adapt to changes in the task (8.3),
- be driven by command sources (LLM-based HRI interface) that are fully decoupled from the implementation of TeMoto via UMRF (8.4),
- and share semantic knowledge of the environment within a robotic fleet (8.5).

9. DISCUSSION

This section discusses the future work and limitations of TeMoto's Task and Resource management layers, as well as the limiting factors and further improvements of the evaluation. Finally, author's experiences and lessons learned are listed.

9.1. Task Management

Formal analysis of graphs: Section 6.1 covers the definitions and semantics of graph execution, but there is currently no mathematical framework for defining and checking the validity of UMRF graphs. This would be a necessary addition when either designing graphs or when graphs are modified during run-time, where the resulting graph can be validated before any operations are applied.

Model for Reactive Actions: Reactive behavior is currently implemented as actions that can re-run after they finish, thus being able to trigger the graph evolution procedure on every subsequent child action. Yet, such implementation for reactiveness can lead to unexpected behavior if the graph is not carefully designed and analyzed.

Design and Introspection Tools: TeMoto Action Designer is a graphical tool that streamlines the development of UMRF graphs and action plugins. However, it is currently not flexible enough to support different middleware layers (no middleware, ROS, ROS2). Also, it does not fully support the condition mechanism outlined in Section 6.1.3. Since UMRF graphs can be converted to JSON notation and back, along with their state and parameters, the graphs can be potentially visualized and controlled through an external introspection tool, which is a direction for future work.

LLM-Based Task Description Generation: Utilizing LLMs to generate UMRF graphs has already been demonstrated in a limited scope (see Section 8.4). LLM which has not been fine-tuned for a specific application domain requires an extensive set of examples to be embedded into each prompt, to yield complex UMRF graphs. Analyzing the scalability and accuracy/trustworthiness of such an approach is a research domain to be explored.

9.2. Resource Management

ROS2 support: The common functionalities of resource management are implemented as a middleware-independent module, i.e., the RR (see Section 7.2). However, the RR needs to be bridged/wrapped with middleware-specific functionalities in order to use it in a respective robotic setup. While ROS is a widely used middleware in the robotics community, the TeMoto framework lacks a feature-complete wrapper for ROS2, thus hindering the transition of all TeMoto resource managers (see Table 13) to ROS2.

Other programming languages: One of the strengths of distributed applications is the potential to utilize different programming languages in the implementation of individual distributed components. As the RR is utilized both on the provider and consumer sides of a distributed application (see Figure 15), the consumer application currently needs to be implemented in C++. Supporting languages, such as Python, thus require either a programming language-specific binding or a complete reimplementation of the RR.

Consumer prioritization: The RR is missing resource consumer prioritization features, which are needed for allowing higher priority consumers to gain control over a resource (or deny a request of a low priority consumer). For example, if a manipulator arm (a resource) is initialized and controlled by a low-priority task (consumer 1), a higher-priority task (consumer 2), e.g., an external collision risk assessment routine, can gain control by acquiring the resource and releasing it shortly after the risk has been mitigated.

Formal representation: Resources are dynamically registered, allocated, monitored, and released, which facilitates the design of adaptive robotic systems. Yet modeling the resources and using the models would allow developers to design and check the validity of specific resource configurations prior to deployment.

9.3. Evaluation

The technical evaluation of this work focused on demonstrating the dynamic or deployment-time capabilities of TeMoto, enabling the robot to autonomously reconfigure its resources and tasks at hand to overcome hardware/software errors, conserve energy and computational resources, or adapt to changes in the mission. Thus the evaluation covered fundamental principles of resource management, task management, and use cases involving human-robot interaction and multi-robot systems. III further evaluates the HRI multi-modality aspects, enabled via UMRF, which are not reiterated in this work.

While the demonstrations provide a functional overview of TeMoto, long-term deployment of a TeMoto-enabled robot or multi-agent fleet in an actual high-risk scenario is to be conducted. Also, in addition to qualitative comparison with related work (see Section 4), it would be valuable to collect quantitative data based on developer feedback (time spent on programming a behavior, etc) in a comparative user study.

The demonstrations did not cover decentralized multi-robot task execution (as opposed to a centralized setup outlined in Sections 8.2, 8.3, and 8.5), but the capability is fully implemented, tested via continuous integration, and available as a part of TeMoto Action Engine[1].

Finally, quantitative evaluation of robotic autonomy architectures is an open question in the robotics community. In most cases (including this work) evalu-

[1] github.com/temoto-framework/temoto_action_engine

ations are predominantly performed qualitatively, demonstrating the features in related context. Yet the results are rarely comparable on quantitative level, as there are no uniformly accepted performance metrics. Benchmarking suites for autonomous robotic tasks in, e.g., rescue[90] and manufacturing[91] scenarios have also been proposed, but are yet to receive wider adoptance by the robotics research community.

9.4. Lessons Learned

Developing and maintaining a research-driven and constantly evolving code-base has come with it's own challenges, which have lead to some useful insights.

Naming in the code - With a growing code-base, the names of classes, variables, methods, etc., should be carefully considered. At first glance such suggestion might sound a bit obsessive and unnecessary. But once a specific name has propagated as a dependency through other code modules, potentially hosted on separate repositories, renaming may become a chore that can potentially even break the code. Thus it might not hurt to sit down for a bit and even ask opinions when coming up with names for the structural elements of the code.

Prototype vs production - Avoid production-driven perfectionism rabbit-holes, but do not start prototyping without giving some thought either. When implementing code, finding a balance between a minimally viable and a feature-complete "production-ready" solution helps to save significant amount of work down the road. One should always ask the question of "what is the domain and scope of the use-case" and narrow down the set of features that are fundamentally necessary, and which are not. The reasoning is that without fully understanding the domain (which can occasionally happen), the code is likely subject to refactoring. And even if the domain is fully known, the path to the final solution is often not. In that light, usually non-critical features are to be changed or removed completely, and thus a minimally viable implementation for non-critical code will suffice. The opposite of it can be considered as a premature optimization. Yet the critical features need more consideration, as these are the foundational building-blocks that are used to structure the code and may necessitate a cascade of unplanned changes if not laid out carefully.

Sleep on it - As unoriginal it may sound, knowing when to take a break may avoid unnecessary stress. Often the solutions for a specific design problem, or a bug in the code are non-trivial. One may arrive at a solution as a result of thorough and systematic analysis, but that is not always guaranteed to happen. Such moments induce the feeling of incompetence and frustration. The more likely reason for not finding a solution is not incompetence, but the fact that there is a lot of information to process. In such cases forcing oneself to work without any significant breaks does not assure success. Sleeping on it might give time to truly absorb the complexity of the subject.

10. CONCLUSION

Robotic autonomy, especially in unpredictable and hazardous application domains, is a non-trivial challenge, which aims to combine two almost contradictory aspects - system reliability and complexity. Even from a purely statistical perspective, the fewer components or points of failure a system has, the more reliable it is. Thus teleoperated robotic systems, often even controlled over a wired communication link, are preferred in, e.g., nuclear decontamination and decommissioning tasks. Yet conditions where wired connection is not an option and wireless connection is intermittent (extravehicular activities during planetary exploration, semi-collapsed caves, etc), the direct teleoperation needs to be gradually replaced with semi or fully autonomous capabilities without compromising system reliability.

The effort undergone in researching robotic autonomy architectures is indicated by the emergence of task management tools, such as SMACH[5], BehaviorTree.Cpp[6], RAFCON[7], TaskForce[8]; component-based system modeling and runtime reconfiguration tools, such as Rorg[9], MROS[10], Dyknow[11], or Dr-Bip[12]; multi-robot task management frameworks, e.g., Open-RMF[13]; UX and HRI guidelines[14], [15]; and robotic application development frameworks, such as ROS[16] or YARP[17]. Despite the numerous advances and tools available, developing a universally adaptive, multi-agent configurable, and scalable software architecture for robotic autonomy remains an ongoing challenge in the robotics community [3], [18].

The goal of this work was to design a software architecture that facilitates the deployment of autonomous robots to high-risk and high-complexity task domains. Thus, the desired architectural features, as well as architectural design principles were analyzed in the given context. Adaptability, scalability, and support for multi-agent and human-robot collaboration were concluded as the desired features for such a robotic system, with the syndicate architecture chosen as a reference for design and development. Based on the qualitative analysis, the requirements were defined which were used both as a baseline for comparing related work and for deriving and implementing a novel software framework — TeMoto.

TeMoto focuses on the task management (executive layer) and resource management (functional layer) aspects of the robotic autonomy stack, emphasizing adaptive and scalable system design. The layers are combined via ROS-based middleware infrastructure, streamlining the integration with the variety of software modules available in the ROS ecosystem. The task management layer constitutes a novel domain-specific language, the Unified Meaning Representation Format (UMRF), along with a modular C++ based executive, the TeMoto Action Engine, which can execute decentralized multi-agent tasks outlined in UMRF notation. The intention of UMRF is to provide a flexible notation for describing tasks, which helps to decouple command sources (user input, etc) from executive layer tools. The resource management layer on the other hand provides dynamic

control over resources, e.g., sensors, actuators, and algorithms. Often resources are assumed to have a fixed configuration throughout the application's lifecycle. Yet during long-term deployment or in unpredictable conditions, unexpected issues may occur which could alleviated by programmatically reconfiguring the system setup, i.e., switch from a damaged sensor setup or unreliable controller configuration to a stable one. TeMoto's resource management provides a set of tools to dynamically start, stop, combine, and monitor resources.

Five technical demonstrators were outlined to evaluate the feasibility of TeMoto. The evaluation focused on demonstrating the dynamic or deployment-time capabilities of TeMoto, enabling the robot to autonomously reconfigure its resources and tasks at hand to overcome hardware/software errors, conserve energy and computational resources, and adapt to changes in the mission. Thus the evaluation covered fundamental principles of resource management, task management, and use cases involving human-robot interaction and multi-robot systems. The demonstrations cover the major functional aspects of TeMoto, indicating the potential of utilizing it as a core autonomy architecture for robots deployed in high-risk and high-complexity task domains. Still, TeMoto is yet to be tested in a full-scale scenario involving long term deployment in challenging environments interlaced with complex user and multi-robot interactions.

The development of TeMoto is an ongoing process, and this work captures the current state, outlining the core design principles and set of implemented tools. TeMoto continues to push the research of robotic autonomy, with the goal of streamlining the development of reliable and adaptive robotic systems.

BIBLIOGRAPHY

[1] T. Yoshida, K. Nagatani, S. Tadokoro, T. Nishimura, and E. Koyanagi, "Improvements to the rescue robot quince toward future indoor surveillance missions in the fukushima daiichi nuclear power plant," in *Field and Service Robotics: Results of the 8th International Conference*, K. Yoshida and S. Tadokoro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 19–32, ISBN: 978-3-642-40686-7. DOI: 10.1007/978-3-642-40686-7_2.

[2] M. Parisi, T. Panontin, S.-C. Wu, K. McTigue, and A. Vera, "Effects of communication delay on human spaceflight missions," in *14th International Conference on Applied Human Factors and Ergonomics (AHFE)*, 2023.

[3] S. Bensalem, F. Ingrand, and J. Sifakis, "Autonomous robot software design challenge," in *Proceedings of Sixth IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, 2008.

[4] A. Ahmad and M. A. Babar, "Software architectures for robotic systems: A systematic mapping study," *Journal of Systems and Software*, vol. 122, pp. 16–39, 2016, ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.08.039.

[5] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[6] D. Faconti and M. Colledanchise, *BehaviorTree.CPP*, version 3.0, 2019. [Online]. Available: https://github.com/BehaviorTree/BehaviorTree.CPP.

[7] S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel, "Rafcon: A graphical tool for engineering complex, robotic tasks," vol. 2016-November, Institute of Electrical and Electronics Engineers Inc., Nov. 2016, pp. 3283–3290, ISBN: 9781509037629. DOI: 10.1109/IROS.2016.7759506.

[8] P. Strawser, *askForce: A Software Task Design and Execution Framework*. [Online]. Available: https://github.com/BehaviorTree/BehaviorTree.CPP.

[9] S. Wang, X. Liu, J. Zhao, and H. I. Christensen, "Rorg: Service robot software management with linux containers," vol. 2019-May, Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 584–590, ISBN: 9781538660263. DOI: 10.1109/ICRA.2019.8793764.

[10] C. H. Corbato, D. Bozhinoski, M. G. Oviedo, G. van der Hoorn, N. H. Garcia, H. Deshpande, J. Tjerngren, and A. Wasowski, "Mros: Runtime adaptation for robot control architectures," Oct. 2020. [Online]. Available: https://arxiv.org/abs/2010.09145v1.

[11] D. D. Leng and F. Heintz, "Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system," Institute of Electrical and Electronics Engineers Inc., Feb. 2017, pp. 55–60, ISBN: 9781509046164. DOI: 10.1109/SIMPAR.2016.7862375.

[12] A. El-Hokayem, S. Bensalem, M. Bozga, and J. Sifakis, "A layered implementation of drbip supporting run-time monitoring and analysis," vol. 12310 LNCS, Springer Science and Business Media Deutschland GmbH, Sep. 2020, pp. 284–302, ISBN: 9783030587673. DOI: 10.1007/978-3-030-58768-0_16.

[13] M. Quigley, *Programming Multiple Robots with ROS 2*. OSRF, 2020, https://osrf.github.io/ros2multirobotbook/intro.html(visited 2024-05-16). [Online]. Available: https://osrf.github.io/ros2multirobotbook/intro.html (visited on 05/16/2024).

[14] M. Giuliani, C. Lenz, T. Müller, M. Rickert, and A. Knoll, "Design principles for safety in human-robot interaction," *International Journal of Social Robotics*, vol. 2, pp. 253–274, 2010. DOI: 10.1007/s12369-010-0052-0.

[15] H. A. Yanco, A. Norton, W. Ober, D. Shane, A. Skinner, and J. Vice, "Analysis of human-robot interaction at the darpa robotics challenge trials," *Journal of Field Robotics*, vol. 32, no. 3, pp. 420–444, 2015. DOI: 10.1002/rob.21568. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21568.

[16] A. Koubâa *et al.*, *Robot Operating System (ROS).* Springer, 2017, vol. 1. DOI: https://doi.org/10.1007/978-3-319-26054-9.

[17] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006. DOI: https://doi.org/10.5772/5761.

[18] X. Mao, H. Huang, and S. Wang, "Software engineering for autonomous robot: Challenges, progresses and opportunities," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2020-December, pp. 100–108, Dec. 2020, ISSN: 15301362. DOI: 10.1109/APSEC51365.2020.00018.

[19] E. S. A. (ESA), *Technology readiness levels (trl)*, 2024. [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/Technology_Readiness_Levels_TRL.

[20] F. Kendoul, "Towards a unified framework for uas autonomy and technology readiness assessment (atra)," in *Autonomous Control Systems and Vehicles: Intelligent Unmanned Systems*, K. Nonami, M. Kartidjo, K.-J. Yoon, and A. Budiyono, Eds. Tokyo: Springer Japan, 2013, pp. 55–71, ISBN: 978-4-431-54276-6. DOI: 10.1007/978-4-431-54276-6_4.

[21] T. B. Sheridan, W. L. Verplank, and T. Brooks, "Human/computer control of undersea teleoperators," in *NASA. Ames Res. Center The 14th Ann. Conf. on Manual Control*, 1978.

[22] A. Rankin, M. Maimone, J. Biesiadecki, N. Patel, D. Levine, and O. Toupet, "Driving curiosity: Mars rover mobility trends during the first seven years," IEEE Computer Society, Mar. 2020, ISBN: 9781728127347. DOI: 10.1109/AERO47225.2020.9172469.

[23] S. M. Milkovich, K. M. Stack, V. Z. Sun, K. Maxwell, R. Kronyak, S. L. Schnadt, K. Steadman, and N. Spanovich, "Balancing predictive and reactive science planning for mars 2020 perseverance," in *2022 IEEE Aerospace Conference (AERO)*, 2022, pp. 1–12. DOI: 10.1109/AERO53065.2022.9843572.

[24] V. Michal, "Remote operation and robotics technologies in nuclear decommissioning projects," in. Elsevier, Jan. 2012, pp. 346–374. DOI: 10.1533/9780857095336.2.346.

[25] Y. Yokokohji, "The use of robots to respond to nuclear accidents: Applying the lessons of the past to the fukushima daiichi nuclear power station," *Annu. Rev. Control Robot. Auton. Syst*, 2021. DOI: 10.1146/annurev-control-071420.

[26] J. Delmerico, S. Mintchev, A. Giusti, B. Gromov, K. Melo, T. Horvat, C. Cadena, M. Hutter, A. Ijspeert, D. Floreano, L. M. Gambardella, R. Siegwart, and D. Scaramuzza, "The current state and future outlook of rescue robotics," *Journal of Field Robotics*, vol. 36, pp. 1171–1191, 7 Oct. 2019, ISSN: 1556-4959. DOI: 10.1002/rob.21887.

[27] G. Kruijff, I. Kruijff-Korbayová, S. Keshavdas, B. Larochelle, M. Janíček, F. Colas, M. Liu, F. Pomerleau, R. Siegwart, M. Neerincx, R. Looije, N. Smets, T. Mioch, J. van Diggelen, F. Pirri, M. Gianni, F. Ferri, M. Menna, R. Worst, T. Linder, V. Tretyakov, H. Surmann, T. Svoboda, M. Reinštein, K. Zimmermann, T. Petříček, and V. Hlaváč, "Designing, developing, and deploying systems to support human–robot teams in disaster response," *Advanced Robotics*, vol. 28, pp. 1547–1570, 23 Dec. 2014, ISSN: 0169-1864. DOI: 10.1080/01691864.2014.985335.

[28] P. Liu, H. Yu, S. Cang, and L. Vladareanu, "Robot-assisted smart firefighting and interdisciplinary perspectives," Institute of Electrical and Electronics Engineers Inc., Oct. 2016, pp. 395–401, ISBN: 9781862181311. DOI: 10.1109/IConAC.2016.7604952.

[29] M. Kyrarini, F. Lygerakis, A. Rajavenkatanarayanan, C. Sevastopoulos, H. R. Nambiappan, K. K. Chaitanya, A. R. Babu, J. Mathew, and F. Makedon, "A survey of robots in healthcare," 2021. DOI: 10.3390/technologies9010008.

[30] A. D. Lallo, R. Murphy, A. Krieger, J. Zhu, R. H. Taylor, and H. Su, "Medical robots for infectious diseases: Lessons and challenges from the covid-19 pandemic," *IEEE Robotics and Automation Magazine*, vol. 28, pp. 18–27, 1 Mar. 2021, ISSN: 1558223X. DOI: 10.1109/MRA.2020.3045671.

[31] L. Rozo, H. B. Amor, S. Calinon, A. Dragan, and D. Lee, *Special issue on learning for human–robot collaboration*, Jun. 2018. DOI: 10.1007/s10514-018-9756-z.

[32] R. Benotsmane, L. Dudás, and G. Kovács, "Survey on new trends of robotic tools in the automotive industry," vol. 22, Springer Science and Business Media Deutschland GmbH, Nov. 2021, pp. 443–457, ISBN: 9789811595288. DOI: 10.1007/978-981-15-9529-5_38.

[33] C. Fries, M. Fechter, G. Nick, Á. Szaller, and T. Bauernhansl, "First results of a survey on manufacturing of the future," vol. 180, Elsevier B.V., 2021, pp. 142–149. DOI: 10.1016/j.procs.2021.01.137.

[34] L. D. Evjemo, T. Gjerstad, E. I. Grøtli, and G. Sziebig, "Trends in smart manufacturing: Role of humans and industrial robots in smart factories," *Current Robotics Reports*, vol. 1, pp. 35–41, 2 Jun. 2020. DOI: 10.1007/s43154-020-00006-5.

[35] E. Matheson, R. Minto, E. G. G. Zampieri, M. Faccio, and G. Rosati, "Human–robot collaboration in manufacturing applications: A review," *Robotics*, vol. 8, p. 100, 4 Dec. 2019, ISSN: 2218-6581. DOI: 10.3390/robotics8040100.

[36] J. A. Marvel, S. Bagchi, M. Zimmerman, and B. Antonishek, "Towards effective interface designs for collaborative hri in manufacturing: Metrics and measures," *Trans. Hum.-Robot Interact*, vol. 9, 2020. DOI: 10.1145/3385009.

[37] *Nasa technology roadmaps - ta 4: Robotics and autonomous systems*, 2015. [Online]. Available: https://www.nasa.gov/offices/oct/home/taxonomy.

[38] D. Miranda, "2020 nasa technology taxonomy," 2020. [Online]. Available: https://ntrs.nasa.gov/search.jsp?R=20200000399.

[39] M. Eich, R. Hartanto, S. Kasperski, S. Natarajan, and J. Wollenberg, "Towards coordinated multirobot missions for lunar sample collection in an unknown environment," *Journal of Field Robotics*, vol. 31, pp. 35–74, 1 Jan. 2014, ISSN: 15564959. DOI: 10.1002/rob.21491.

[40] M. Amoretti and M. Reggiani, "Architectural paradigms for robotics applications," *Advanced Engineering Informatics*, vol. 24, pp. 4–13, 1 Jan. 2010, ISSN: 14740346. DOI: 10.1016/j.aei.2009.08.004.

[41] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019, ISBN: 9781492047841.

[42] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, pp. 1–42, 2 May 2009, ISSN: 1556-4665. DOI: 10.1145/1516533.1516538.

[43] M. Törngren, D. J. Chen, and I. Crnkovic, "Component-based vs. model-based development: A comparison in the context of vehicular embedded systems," vol. 2005, 2005, pp. 432–440, ISBN: 0769524311. DOI: 10.1109/EUROMICRO.2005.18.

[44] M. Wojtynek, H. Oestreich, O. Beyer, and S. Wrede, "Collaborative and robot-based plug produce for rapid reconfiguration of modular production systems," *SII 2017 - 2017 IEEE/SICE International Symposium on System Integration*, vol. 2018-January, pp. 1067–1073, Feb. 2018. DOI: 10.1109/SII.2017.8279364.

[45] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

[46] D. Crestani, K. Godary-Dejean, and L. Lapierre, "Enhancing fault tolerance of autonomous mobile robots," *Robotics and Autonomous Systems*, vol. 68, pp. 140–155, Jun. 2015, ISSN: 09218890. DOI: 10.1016/j.robot.2014.12.015.

[47] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, Aug. 2017, ISSN: 09218890. DOI: 10.1016/j.robot.2017.04.004.

[48] R. E. Ballouli, S. Bensalem, M. Bozga, and J. Sifakis, "Programming dynamic reconfigurable systems," vol. 11222 LNCS, Springer Verlag, Oct. 2018, pp. 118–136, ISBN: 9783030021450. DOI: 10.1007/978-3-030-02146-7_6.

[49] G. Brataas and P. Hughes, "Exploring architectural scalability," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 125–129, Jan. 2004, ISSN: 0163-5948. DOI: 10.1145/974043.974064.

[50] Q. Yang and R. Parasuraman, "Needs-driven heterogeneous multi-robot cooperation in rescue missions," *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2020*, pp. 252–259, Nov. 2020. DOI: 10.1109/SSRR50563.2020.9292570.

[51] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, *Rosplan: Planning in the robot operating system*, 2015. DOI: https://doi.org/10.1609/icaps.v25i1.13699.

[52] S. Garcia, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab, "An architecture for decentralized, collaborative, and autonomous robots," *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, pp. 75–84, Jul. 2018. DOI: 10.1109/ICSA.2018.00017.

[53] M. Beetz, D. Beßler, A. Haidu, M. Pomarlan, A. K. Bozcuoğlu, and G. Bartels, "Know rob 2.0—a 2nd generation knowledge processing framework for cognition-enabled robotic agents," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 512–519. DOI: 10.1109/ICRA.2018.8460964.

[54] A. Koubâa, M. F. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. Shakshuki, "Coros: A multi-agent software architecture for cooperative and autonomous service robots," *Studies in Computational Intelligence*, vol. 604, pp. 3–30, 2015, ISSN: 1860949X. DOI: 10.1007/978-3-319-18299-5_1.

[55] Z. Kemény, J. Váncza, L. Wang, and X. V. Wang, "Human–robot collaboration in manufacturing: A multi-agent view," in *Advanced human-robot collaboration in manufacturing*, Springer, 2021, pp. 3–41. DOI: https://doi.org/10.1007/978-3-030-69178-3_1.

[56] L. Wang, J. Váncza, Z. Kemény, and X. V. Wang, "Future research directions on human–robot collaboration," *Advanced Human-Robot Collaboration in Manufacturing*, pp. 439–448, 2021. DOI: https://doi.org/10.1007/978-3-030-69178-3_18.

[57] S. C. Akkaladevi, M. Propst, M. Hofmann, L. Hiesmair, M. Ikeda, N. C. Chitturi, and A. Pichler, "Programming-free approaches for human–robot collaboration in assembly tasks," in *Advanced Human-Robot Collaboration in Manufacturing*, Springer, 2021, pp. 283–317. DOI: https://doi.org/10.1007/978-3-030-69178-3_12.

[58] D. Kortenkamp, R. Simmons, and D. Brugali, "Robotic systems architectures and programming," in. Springer International Publishing, Jan. 2016, pp. 283–305, ISBN: 9783319325521. DOI: 10.1007/978-3-319-32552-1_12.

[59] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software architecture: a comprehensive framework and guide for practitioners*. Springer Science & Business Media, 2011.

[60] J. Calzado, A. Lindsay, C. Chen, G. Samuels, and J. Olszewska, "Sami: Interactive, multi-sense robot architecture," in *2018 IEEE 22nd international conference on intelligent engineering systems (INES)*, IEEE, 2018, pp. 000 317–000 322. DOI: 10.1109/INES.2018.8523933.

[61] N. Nilson, "A mobile automaton: An application of ai techniques," in *Proceedings of the 1969 International Joint Conference on Artificial Intelligence*, 1969.

[62] R. Brooks, "A robust layered control system for a mobile robot," *IEEE journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986. DOI: 10.1109/JRA.1986.1087032.

[63] P. Muñoz, M. D. R-Moreno, D. F. Barrero, and F. Ropero, "Mobar: A hierarchical action-oriented autonomous control architecture," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 94, pp. 745–760, 3-4 Jun. 2019, ISSN: 15730409. DOI: 10.1007/s10846-018-0810-z.

[64] A. Favier, A. Messioux, J. Guiochet, J.-C. Fabre, and C. Lesire, *A hierarchical fault tolerant architecture for an autonomous robot*, Jun. 2020. [Online]. Available: https://hal.laas.fr/hal-02558604.

[65] B. Sellner, F. W. Heger, L. M. Hiatt, R. Simmons, and S. Singh, "Coordinated multiagent teams and sliding autonomy for large-scale assembly," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1425–1444, 2006. DOI: 10.1109/JPROC.2006.876966.

[66] F. Ingrand and M. Ghallab, *Deliberation for autonomous robots: A survey*, Jun. 2017. DOI: 10.1016/j.artint.2014.11.003.

[67] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003. DOI: https://doi.org/10.1613/jair.1129.

[68] L. Guan, K. Valmeekam, S. Sreedharan, and S. Kambhampati, "Leveraging pre-trained large language models to construct and utilize world models for model-based task planning," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 79 081–79 094.

[69] S. S. Raman, V. Cohen, E. Rosen, I. Idrees, D. Paulius, and S. Tellex, "Planning with large language models via corrective re-prompting," in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=cMDMRBe1TKs.

[70] M. Shridhar, L. Manuelli, and D. Fox, "Perceiver-actor: A multi-task transformer for robotic manipulation," in *Proceedings of The 6th Conference on Robot Learning*, K. Liu, D. Kulic, and J. Ichnowski, Eds., ser. Proceedings of Machine Learning Research, vol. 205, PMLR, 14–18 Dec 2023, pp. 785–799.

[71] D. Carvalho, A. Martins, J. M. Almeida, and E. Silva, "A smacc based mission control system for autonomous underwater vehicles," in *OCEANS 2022, Hampton Roads*, IEEE, 2022, pp. 1–10. DOI: 10.1109/OCEANS47191.2022.9977228.

[72] P. Schillinger, S. Kohlbrecher, and O. V. Stryk, "Human-robot collaborative high-level control with application to rescue robotics," vol. 2016-June, Institute of Electrical and Electronics Engineers Inc., Jun. 2016, pp. 2796–2802, ISBN: 9781467380263. DOI: 10.1109/ICRA.2016.7487442.

[73] O. A. Specification, "Omg unified modeling language (omg uml), superstructure, v2. 1.2," *Object Management Group*, vol. 70, p. 16, 2007.

[74] M. Colledanchise, "Behavior trees in robotics," Ph.D. dissertation, KTH, Robotics, perception and learning, RPL, 2017, p. 63, ISBN: 978-91-7729-283-8.

[75] A. Katranov and A. Kukanov, "Intel® threading building block (intel® tbb) flow graph as a software infrastructure layer for opencl™-based computations," in *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1–3. DOI: https://doi.org/10.1145/2909437.2909446.

[76] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2021. DOI: 10.1109/TPDS.2021.3104255.

[77] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9, Vienna, Austria: Association for Computing Machinery, 2001, pp. 99–108, ISBN: 1581133901. DOI: 10.1145/503209.503224.

[78] P. Gupta, "Modularity enablers: A tool for industry 4.0," *Life Cycle Reliability and Safety Engineering*, vol. 8, pp. 157–163, 2 Jun. 2019, ISSN: 2520-1352. DOI: 10.1007/s41872-018-0067-3.

[79] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler, “Parameterizing behavior trees,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7060 LNCS, pp. 144–155, 2011, ISSN: 03029743. DOI: 10.1007/978-3-642-25090-3_13/COVER.

[80] S. G. Brunner, A. Domel, P. Lehner, M. Beetz, and F. Stulp, “Autonomous parallelization of resource-aware robotic task nodes,” *IEEE Robotics and Automation Letters*, vol. 4, pp. 2599–2606, 3 Jul. 2019, ISSN: 23773766. DOI: 10.1109/LRA.2019.2894463.

[81] S. Wang, X. Liu, J. Zhao, H. Christensen, and H. I. Christensen, “Robotic reliability engineering: Experience from long-term tritonbot development,” EasyChair, Jul. 2019.

[82] N. Hawes, C. Burbridge, F. Jovan, L. Kunze, B. Lacerda, L. Mudrová, J. Young, J. Wyatt, D. Hebesberger, T. Körtner, R. Ambrus, N. Bore, J. Folkesson, P. Jensfelt, L. Beyer, A. Hermans, B. Leibe, A. Aldoma, T. Faülhammer, M. Zillich, M. Vincze, E. Chinellato, M. Al-Omari, P. Duckworth, Y. Gatsoulis, D. C. Hogg, A. G. Cohn, C. Dondrup, J. P. Fentanes, T. Krajník, J. M. Santos, T. Duckett, and M. Hanheide, “The strands project: Long-term autonomy in everyday environments,” *IEEE Robotics and Automation Magazine*, vol. 24, pp. 146–156, 3 Sep. 2017, ISSN: 10709932. DOI: 10.1109/MRA.2016.2636359.

[83] S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia, “Soter on ros: A run-time assurance framework on the robot operating system,” in. Springer, Cham, Oct. 2020, pp. 184–194. DOI: 10.1007/978-3-030-60508-7_10.

[84] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: Safe asynchronous event-driven programming,” *SIGPLAN Not.*, vol. 48, no. 6, pp. 321–332, Jun. 2013, ISSN: 0362-1340. DOI: 10.1145/2499370.2462184.

[85] M. Mayr, F. Rovida, and V. Krueger, “Skiros2: A skill-based robot control platform for ros,” in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 6273–6280. DOI: 10.1109/IROS55552.2023.10342216.

[86] A. Nordmann, R. Lange, and F. M. Rico, “System modes - digestible system (re-)configuration for robotics,” *Proceedings - 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering, RoSE 2021*, pp. 19–24, Jun. 2021. DOI: 10.1109/ROSE52553.2021.00010.

[87] A. Kustavus, “Design and implementation of a generalized resource management architecture in the temoto software framework,” M.S. thesis, University of Tartu, 2021.

[88] S. Wanna, F. Parra, R. Valner, K. Kruusamäe, and M. Pryor, “Unlocking underrepresented use-cases for large language model-driven human-robot task planning,” *Advanced Robotics*, vol. 0, no. 0, pp. 1–14, 2024. DOI: 10.1080/01691864.2024.2366974.

[89] M. Pihlap, “Mitme roboti koostöö funktsionaalsuste väljatöötamine tarkvararaamistikule temoto,” University of Tartu, 2021.

[90] A. Jacoff, E. Messina, H.-M. Huang, A. Virts, A. Downs, R. Norcross, and R. Sheh, “Guide for evaluating, purchasing, and training with response robots using dhs-nist-astm international standard test methods,” *National Institute of Standards and Technology report*, 2014.

[91] T. Niemueller, G. Lakemeyer, S. Reuter, S. Jeschke, and A. Ferrein, “Chapter 13 - benchmarking of cyber-physical systems in industrial robotics: The robocup logistics league as a cps benchmark blueprint,” in *Cyber-Physical Systems*, ser. Intelligent Data-Centric Systems, H. Song, D. B. Rawat, S. Jeschke, and C. Brecher, Eds., Boston: Academic Press, 2017, pp. 193–207, ISBN: 978-0-12-803801-7. DOI: 10.1016/B978-0-12-803801-7.00013-4.

ACKNOWLEDGEMENTS

This work, or rather a milestone, was a lengthy, exhausting, and very rewarding path that would not have been possible without the support of helpful and insightful people around me.

Starting from the supervisors, Karl Kruusamäe and Alvo Aabloo, thanks to whom I got a chance to work on something I have desired to do many years before. In addition to their support, I was provided a productive work environment, i.e., a combination of available resources, great connections, and a lab full of brains to pick at.

I am very thankful to Mitch Pryor who welcomed me to the Nuclear and Applied Robotics Group. I cannot stress enough what an impact it had on my research and life in general. The traces of Mitch's open-mindedness and support can likely be seen in every aspect of this work.

Big thanks to my fellow code ninjas Veiko Vunder, Fabian Parra, Allan Kustavus, and Meelis Pihlap. The fact that they agreed to listen, passionately argue about, and actually realize many of my vague concepts is greatly appreciated and fundamentally necessary for the success of this work. Also special thanks to Bahman Ghandchi, who not only helped me to translate my rough ideas into concise mathematical form but also contributed to it while doing so.

The time I devoted to this work did not come out of thin air, and on many occasions, it came at the cost of mutual time with family and friends. Thus I thank you all for being there for me, especially you Piia, thank you.

This doctoral thesis was in part supported by the project „Increasing the knowledge intensity of Ida-Viru entrepreneurship“ co-funded by the European Union.

SISUKOKKUVÕTE

TeMoto – töökindlate, adaptiivsete ja koostöövõimeliste autonoomsete robotite arendamise tarkvararaamistik

Autonoomsete robotite arendamise üks suurimaid motivatsioone on võtta inimestelt üle tööd, mis on eluohtlikud, stressirohked ja füüsiliselt rasked. Samas on usaldusväärse autonoomia saavutamine ettearvamatutes ja ohtlikes rakendusvaldkondades (tulekahjud, tuumajäätmete ja tuumakriiside haldamine, ehitise rusudes töötamine, kosmosemissioonid, jne) väljakutse, mis eeldab arendatavalt süsteemilt kaht vastuolulist omadust - kõrget usaldusväärsust ja komplekssust. Seetõttu eelistatakse väljakutsuvates keskkondades kaugjuhitavaid robotisüsteeme, kus usaldusväärse soorituse tagab kvalifitseeritud operaator. Kui aga roboti kaugjuhtimine ei ole võimalik (liiga suured vahemaad, keeruline keskkond, jne), peab robot olema paratamatult kas pool- või täielikult autonoomne.

Käesoleva doktoritöö eesmärkideks on: a) analüüsida tarkvara arendamise printsiipe, mis aitavad suurendada robotite autonoomia taset riskantsete ja keerukate ülesannete puhul; ning b) arendada välja tarkvara arhitektuur, mis on kooskõlas nende printsiipidega. Antud doktoritöö peamiseks väljundiks on tarkvara raamistik TeMoto, mis võimaldab arendada adaptiivseid, skaleeruvaid, robot-robot ja inim-roboti koostööle orienteeritud robotite tarkvara.

TeMoto on struktuurselt kolmekihiline arhitektuur (*three layer architecture*), mis on kohandatud detsentraliseeritud ja hajusate mitme-roboti süsteemide jaoks, ja haldab käitusaegselt nii roboti missiooni (täitevkiht, ing. k. *executive layer*) kui ka tarkvara/riistvara ressursse (funktsionaalne kiht, ing. k. *functional layer*). Missioonide kirjeldamiseks on käesoleva töö raames arendatud välja formaat (*Unified Meaning Representation Format*, ehk *UMRF*), mis võimaldab kirjeldada komplekseid, hierarhilisi, ja mitut robotit hõlmavaid missioone JSON-vormingus. UMRF'il baseeruvaid missioonikirjeldusi haldab C++ põhine teek *TeMoto Action Engine*, kus iga missiooni alamkomponent (navigeerimine, objektide manipuleerimine, jne) on dünaamiliselt laetav ja kontrollitav plugin. Ressursside haldamise kiht võimaldab dünaamiliselt kontrollida hierarhiliste ressursside elutsüklit, tagades ressursi korrektse allokeerimise/deallokeerimise ja veahalduse kanali. TeMoto on avatud lähtekoodiga ja mõeldud eeskätt töötamiseks nii ROS-i kui ROS2-ga, kuid põhitööriistu saab kasutada ka väljaspool ROS-i.

Antud töö on valideeritud erinevate stsenaariumite põhjal, mis kätkevad ressursside ja ülesannete haldamist, ning inim-robot ja robot-robot koostööd. TeMoto tarkvararaamistik on pidevas arenduses, ning käesolev töö annab ülevaate TeMoto hetkeseisundist, peamistest disainipõhimõtetest, arendatud tööriistadest ja tulevikusuundadest.

PUBLICATIONS

CURRICULUM VITAE

Personal data

Name:	Robert Valner
Date of birth:	21.01.1991
Contact:	robert.valner@ut.ee
Current Position:	Specialist of Robotics (University of Tartu) and Research Engineer (Clevon)

Education

2015–	Ph.D. Candidate, University of Tartu
2013–2015	MSc. Computer Engineering, University of Tartu
2010–2013	BSc. Physics, University of Tartu

Employment

2022–	Research Engineer, Clevon
2020–	Specialist of Robotics, Institute of Technology, University of Tartu

Scientific work

Main fields of interest:

- Autonomous robotic systems
- Fault tolerant and adaptive robotic architectures
- Multi-robot systems
- Human-robot interaction

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Robert Valner
Sünniaeg:	21.01.1991
Kontaktandmed:	robert.valner@ut.ee
Praegune positsioon:	Robootika spetsialist (Tartu Ülikool) ja teadusarenduse insener (Clevon)

Haridus

2015–	Tartu Ülikool, Loodus- ja täppisteaduste valdkond, tehnoloogiainstituut, doktoriõpe
2013–2015	Tartu Ülikool, Loodus- ja täppisteaduste valdkond, tehnoloogiainstituut, arvutitehnika, magistriõpe
2010–2013	Tartu Ülikool, Loodus- ja täppisteaduste valdkond, füüsika instituut, füüsika, bakalaureuseõpe

Teenistuskäik

2022–	Teadusarenduse insener, Clevon
2020–	Tartu Ülikool, Loodus- ja täppisteaduste valdkond, tehnoloogiainstituut, robootika spetsialist

Teadustegevus

Peamised uurimisvaldkonnad:

- Autonoomsed robotsüsteemid
- Veakindlad ja adaptiivsed robotarhitektuurid
- Mitmerobotisüsteemid
- Inim-robot interaktsioon

DISSERTATIONES TECHNOLOGIAE UNIVERSITATIS TARTUENSIS

1. **Imre Mäger**. Characterization of cell-penetrating peptides: Assessment of cellular internalization kinetics, mechanisms and bioactivity. Tartu 2011, 132 p.
2. **Taavi Lehto**. Delivery of nucleic acids by cell-penetrating peptides: application in modulation of gene expression. Tartu 2011, 155 p.
3. **Hannes Luidalepp**. Studies on the antibiotic susceptibility of *Escherichia coli*. Tartu 2012, 111 p.
4. **Vahur Zadin**. Modelling the 3D-microbattery. Tartu 2012, 149 p.
5. **Janno Torop**. Carbide-derived carbon-based electromechanical actuators. Tartu 2012, 113 p.
6. **Julia Suhorutšenko**. Cell-penetrating peptides: cytotoxicity, immunogenicity and application for tumor targeting. Tartu 2012, 139 p.
7. **Viktoryia Shyp**. G nucleotide regulation of translational GTPases and the stringent response factor RelA. Tartu 2012, 105 p.
8. **Mardo Kõivomägi**. Studies on the substrate specificity and multisite phosphorylation mechanisms of cyclin-dependent kinase Cdk1 in *Saccharomyces cerevisiae*. Tartu, 2013, 157 p.
9. **Liis Karo-Astover**. Studies on the Semliki Forest virus replicase protein nsP1. Tartu, 2013, 113 p.
10. **Piret Arukuusk**. NickFects–novel cell-penetrating peptides. Design and uptake mechanism. Tartu, 2013, 124 p.
11. **Piret Villo**. Synthesis of acetogenin analogues. Asymmetric transfer hydrogenation coupled with dynamic kinetic resolution of α-amido-β-keto esters. Tartu, 2013, 151 p.
12. **Villu Kasari**. Bacterial toxin-antitoxin systems: transcriptional cross-activation and characterization of a novel *mqsRA* system. Tartu, 2013, 108 p.
13. **Margus Varjak**. Functional analysis of viral and host components of alphavirus replicase complexes. Tartu, 2013, 151 p.
14. **Liane Viru**. Development and analysis of novel alphavirus-based multifunctional gene therapy and expression systems. Tartu, 2013, 113 p.
15. **Kent Langel**. Cell-penetrating peptide mechanism studies: from peptides to cargo delivery. Tartu, 2014, 115 p.
16. **Rauno Temmer**. Electrochemistry and novel applications of chemically synthesized conductive polymer electrodes. Tartu, 2014, 206 p.
17. **Indrek Must**. Ionic and capacitive electroactive laminates with carbonaceous electrodes as sensors and energy harvesters. Tartu, 2014, 133 p.
18. **Veiko Voolaid**. Aquatic environment: primary reservoir, link, or sink of antibiotic resistance? Tartu, 2014, 79 p.
19. **Kristiina Laanemets**. The role of SLAC1 anion channel and its upstream regulators in stomatal opening and closure of *Arabidopsis thaliana*. Tartu, 2015, 115 p.

20. **Kalle Pärn**. Studies on inducible alphavirus-based antitumour strategy mediated by site-specific delivery with activatable cell-penetrating peptides. Tartu, 2015, 139 p.
21. **Anastasia Selyutina**. When biologist meets chemist: a search for HIV-1 inhibitors. Tartu, 2015, 172 p.
22. **Sirle Saul**. Towards understanding the neurovirulence of Semliki Forest virus. Tartu, 2015, 136 p.
23. **Marit Orav**. Study of the initial amplification of the human papillomavirus genome. Tartu, 2015, 132 p.
24. **Tormi Reinson**. Studies on the Genome Replication of Human Papillomaviruses. Tartu, 2016, 110 p.
25. **Mart Ustav Jr**. Molecular Studies of HPV-18 Genome Segregation and Stable Replication. Tartu, 2016, 152 p.
26. **Margit Mutso**. Different Approaches to Counteracting Hepatitis C Virus and Chikungunya Virus Infections. Tartu, 2016, 184 p.
27. **Jelizaveta Geimanen**. Study of the Papillomavirus Genome Replication and Segregation. Tartu, 2016, 168 p.
28. **Mart Toots**. Novel Means to Target Human Papillomavirus Infection. Tartu, 2016, 173 p.
29. **Kadi-Liis Veiman**. Development of cell-penetrating peptides for gene delivery: from transfection in cell cultures to induction of gene expression *in vivo*. Tartu, 2016, 136 p.
30. **Ly Pärnaste**. How, why, what and where: Mechanisms behind CPP/cargo nanocomplexes. Tartu, 2016, 147 p.
31. **Age Utt**. Role of alphavirus replicase in viral RNA synthesis, virus-induced cytotoxicity and recognition of viral infections in host cells. Tartu, 2016, 183 p.
32. **Veiko Vunder**. Modeling and characterization of back-relaxation of ionic electroactive polymer actuators. Tartu, 2016, 154 p.
33. **Piia Kivipõld**. Studies on the Role of Papillomavirus E2 Proteins in Virus DNA Replication. Tartu, 2016, 118 p.
34. **Liina Jakobson**. The roles of abscisic acid, CO_2, and the cuticle in the regulation of plant transpiration. Tartu, 2017, 162 p.
35. **Helen Isok-Paas**. Viral-host interactions in the life cycle of human papillomaviruses. Tartu, 2017, 158 p.
36. **Hanna Hõrak**. Identification of key regulators of stomatal CO_2 signalling via O_3-sensitivity. Tartu, 2017, 260 p.
37. **Jekaterina Jevtuševskaja**. Application of isothermal amplification methods for detection of *Chlamydia trachomatis* directly from biological samples. Tartu, 2017, 96 p.
38. **Ülar Allas**. Ribosome-targeting antibiotics and mechanisms of antibiotic resistance. Tartu, 2017, 152 p.
39. **Anton Paier**. Ribosome Degradation in Living Bacteria. Tartu, 2017, 108 p.
40. **Vallo Varik**. Stringent Response in Bacterial Growth and Survival. Tartu, 2017, 101 p.

41. **Pavel Kudrin**. In search for the inhibitors of *Escherichia coli* stringent response factor RelA. Tartu, 2017, 138 p.
42. **Liisi Henno**. Study of the human papillomavirus genome replication and oligomer generation. Tartu, 2017, 144 p.
43. **Katrin Krõlov**. Nucleic acid amplification from crude clinical samples exemplified by *Chlamydia trachomatis* detection in urine. Tartu, 2018, 118 p.
44. **Eve Sankovski**. Studies on papillomavirus transcription and regulatory protein E2. Tartu, 2018, 113 p.
45. **Morteza Daneshmand**. Realistic 3D Virtual Fitting Room. Tartu, 2018, 233 p.
46. **Fatemeh Noroozi**. Multimodal Emotion Recognition Based Human-Robot Interaction Enhancement. Tartu, 2018, 113 p.
47. **Krista Freimann**. Design of peptide-based vector for nucleic acid delivery in vivo. Tartu, 2018, 103 p.
48. **Rainis Venta**. Studies on signal processing by multisite phosphorylation pathways of the *S. cerevisiae* cyclin-dependent kinase inhibitor Sic1. Tartu, 2018, 155 p.
49. **Inga Põldsalu**. Soft actuators with ink-jet printed electrodes. Tartu, 2018, 85 p.
50. **Kadri Künnapuu**. Modification of the cell-penetrating peptide PepFect14 for targeted tumor gene delivery and reduced toxicity. Tartu, 2018, 114 p.
51. **Toomas Mets**. RNA fragmentation by MazF and MqsR toxins of *Escherichia coli.* Tartu, 2019, 119 p.
52. **Kadri Tõldsepp**. The role of mitogen-activated protein kinases MPK4 and MPK12 in CO_2-induced stomatal movements. Tartu, 2019, 259 p.
53. **Pirko Jalakas**. Unravelling signalling pathways contributing to stomatal conductance and responsiveness. Tartu, 2019, 120 p.
54. **S. Sunjai Nakshatharan**. Electromechanical modelling and control of ionic electroactive polymer actuators. Tartu, 2019, 165 p.
55. **Eva-Maria Tombak**. Molecular studies of the initial amplification of the oncogenic human papillomavirus and closely related nonhuman primate papillomavirus genomes. Tartu, 2019, 150 p.
56. **Meeri Visnapuu**. Design and physico-chemical characterization of metal-containing nanoparticles for antimicrobial coatings. Tartu, 2019, 138 p.
57. **Jelena Beljantseva**. Small fine-tuners of the bacterial stringent response – a glimpse into the working principles of Small Alarmone Synthetases. Tartu, 2020, 104 p.
58. **Egon Urgard**. Potential therapeutic approaches for modulation of inflammatory response pathways. Tartu, 2020, 120 p.
59. **Sofia Raquel Alves Oliveira**. HPLC analysis of bacterial alarmone nucleotide (p)ppGpp and its toxic analogue ppApp. Tartu, 2020, 122 p.
60. **Mihkel Örd**. Ordering the phosphorylation of cyclin-dependent kinase Cdk1 substrates in the cell cycle. Tartu, 2021, 228 p.
61. **Fred Elhi**. Biocompatible ionic electromechanically active polymer actuator based on biopolymers and non-toxic ionic liquids. Tartu, 2021, 140 p.

62. **Liisi Talas**. Reconstructing paleo-diversity, dynamics and response of eukaryotes to environmental change over the Late-Glacial and Holocene period in lake Lielais Svētiņu using sedaDNA. Tartu, 2021, 118 p.
63. **Livia Matt**. Novel isosorbide-based polymers. Tartu, 2021, 118 p.
64. **Koit Aasumets**. The dynamics of human mitochondrial nucleoids within the mitochondrial network. Tartu, 2021, 104 p.
65. **Faiza Summer**. Development and optimization of flow electrode capacitor technology. Tartu, 2022, 109 p.
66. **Olavi Reinsalu**. Cancer-testis antigen MAGE-A4 is incorporated into extracellular vesiclesand is exposed to the surface. Tartu, 2022, 130 p.
67. **Tetiana Brodiazhenko**. RelA-SpoT Homolog enzymes as effectors of Toxin-Antitoxin systems. Tartu, 2022, 132 p.
68. **Georg-Marten Lanno**. Development of novel antibacterial drug delivery systems as wound scaffolds using electrospinning technology. Tartu, 2022, 175 p.
69. **Liubov Cherkashchenko**. New insights into alphaviral nsP2 functions. Tartu, 2023, 171 p.
70. **Kristina Kiisholts**. Peptide-based drug carriers and preclinical nanomedicine applications for endometriosis treatment. Tartu, 2023, 138 p.
71. **Kai Rausalu**. Alphaviral nsP2 protease: From requirements for functionality to inhibition. Tartu, 2023, 175 p.
72. **Laura Sandra Lello**. Unraveling the intricate nature of the alphavirus RNA replicase. Tartu, 2023, 219 p.
73. **Houman Masnavi**. Visibility Aware Navigation. Tartu, 2023, 180 p.
74. **Kadir Aktas**. Cosmic Ray Tomography based Object Reconstruction and Recognition. Tartu, 2023, 104 p.
75. **Egils Avots**. Brain abnormality detection using statistical analysis of individual structural connectivity networks and EEG signals. Tartu, 2023, 223 p.
76. **Sainan Wang**. Structure-guided insights into the functions of CHIKV nsP2. Tartu, 2024, 154 p.
77. **Anneli Samel**. Unveiling the characteristics of cancer-testis antigen MAGEA10. Tartu, 2024, 136 p.
78. **Ikechukwu Ofodile**. Fault tolerant attitude control for nanosatellites: ESTCube-2 case. Tartu, 2024, 130 p.
79. **Olena Zamora**. Impacts of plant hormones on controlling stomatal conductance. Tartu, 2024, 166 p.
80. **Mariliis Hinnu**. *In vitro* methods for studying the mechanisms of ribosome-targeting antibiotics. Tartu, 2024, 143 p.
81. **Chung-Yueh Yeh**. Characterization of MPK and HT1 kinases in CO_2-induced stomatal movements. Tartu, 2024, 118 p.
82. **Iman Dadras**. Low power neural network-based control and actuation solutions for insect-scale robots. Tartu, 2024, 149 p.
83. **Fatemeh Rastgar**. Towards reliable real-time trajectory optimization. Tartu, 2024, 158 p.

84. **Maria Maloverjan**. Optimizing cell-penetrating peptide-based nano-particles for delivery of nucleic acid therapeutics. Tartu, 2024, 172 p.
85. **Joonas Merisalu**. Resistive switching in memristor structures with multi-layer dielectrics. Tartu, 2024, 149 p.
86. **Siim Laanesoo**. Novel high-performance biomass-based polymers. Tartu, 2024, 117 p.
87. **Henri Ingelman**. Systems-level characterisation and improvement of *Clostridium autoethanogenum* metabolism. Tartu, 2024, 164 p.
88. **Mailis Laht**. Using the One Health approach for mapping the spread of antibiotic resistant bacteria in Estonia. Tartu, 2024, 188 p.
89. **Ingrid Rebane**. Structure-property relationships of moldable silicone foams. Tartu, 2024, 164 p.