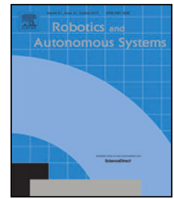
Robotics and Autonomous Systems 183 (2025) 104795

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

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



A microservice based control architecture for mobile robots in safety-critical applications

Manuel Schrick *, Johannes Hinckeldeyn, Marko Thiel, Jochen Kreutzfeldt

Institute for Technical Logistics, Hamburg University of Technology, Theodor-Yorck-Straße 8, Hamburg, 21079, Germany

ARTICLE INFO

Keywords:
Mobile robots
Safety
Software architecture
State machine
Microservices

ABSTRACT

Mobile robots have become more and more common in public space. This increases the importance of meeting safety requirements of autonomous robots. Simple mechanisms, such as emergency braking, alone do not suffice in these highly dynamic situations. Moreover, actual robotic control approaches in literature and practice do not take safety particularly into account. A more sophisticated situational approach for assessment and planning is needed as part of the high-level process control. This paper presents the concept of a safety-critical Robot Control Architecture for mobile robots based on microservices and a Hierarchical Finite State Machine. It expands already existing architectures by drastically reducing the amount of centralized logic and thus increasing the overall system's level of concurrency, interruptibility and fail-safety. Furthermore, it introduces new potential for code reuse that allows for straightforward implementation of safety mechanisms such as internal diagnostics systems. In doing so, this concept presents the template of a new type of state machine implementation. It is demonstrated with the application of a delivery robot, which was implemented and operated in real public during a broader research project.

1. Introduction

The operation of mobile robots in public spaces has become more common in recent years. An instance is the utilization of mobile robots for the automated last-mile delivery of goods. Prominent examples are the delivery robots from Starship Technologies[1] or Yandex[2].

When operating robots in public spaces, the aspect of safety is particularly important [1]. Here, robots are being confronted with more unforeseeable situations, dynamically changing environments and people inexperienced in the interaction with such machines [2]. To navigate these situations without posing a risk to itself or its surroundings, the robot has to meet strict safety requirements. This is especially relevant when an official permit is required for the operation. The necessity of such a permission can originate from local traffic rules, regulations, norms and cyber security guidelines [3]. They demand special considerations in the design and development of robotic hardware and software.

The overall level of safety of a robot can be derived from its *functional safety* and its *safety of the intended functionality* [3]. Functional safety refers to the systems proneness to internal errors as well as its capabilities to react and resolve these. On the software side, functional safety can be addressed by increasing the systems stability by testing thoroughly and reusing its code base as well as internal monitoring systems at runtime.

Safety of the intended functionality on the other hand refers to the system's ability to react safely to unforeseen situations. The most fundamental requirement for this is to allow the robot to interrupt its current behaviour at any point in time. This demands a certain degree of concurrency to run safety checks while progressing the main task. Furthermore, simple mechanisms in lower control layers such as instantly stopping the robot do not suffice. The robot needs to be able to assess the situation, select a series of next steps and act accordingly. It follows that the robot's behaviour in such situations must be part of the high-level process control, or Robot Control Architecture (RCA) [4].

An RCA orchestrates a robot's set of base functionalities, for example autonomous driving or obstacle detection, such that the robot executes a predefined high-level task. It usually also suggests a software architecture pattern to be used in the development process. RCAs differ from robotic middlewares such as ROS [5] and Nvidia Isaac [6] in that they implement the logic that controls the robot whereas middlewares provide an infrastructure that allows RCAs and other software to run and communicate with another.

As shown in the following, current RCAs adopt a centralized, monolithic approach that does not fully make use of the distributed,

* Corresponding author.
E-mail address: manuel.schrick@tuhh.de (M. Schrick).

[1] https://www.starship.xyz

[2] https://sdg.yandex.com/

https://doi.org/10.1016/j.robot.2024.104795

Available online 24 August 2024

0921-8890/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

microservice-based architecture that current middlewares employ and promote. They do not sufficiently consider aspects such as fail-safety, concurrency, interruptibility and code reusability to support the implementation of safety mechanisms. We arrive at this conclusion in the following Section 2 after presenting and discussing the most popular RCA implementations from literature and practice. To address the shortcomings of the current implementations a new concept of RCA for mobile robots is introduced, which allows for a straightforward integration of safety features. This approach improves on current implementations by moving logic that is required to execute each step of the robotic process into a microservice architecture, thus reducing the amount of centralized logic and making better use of current robotic middlewares. The objective of this paper is to present this novel concept and its main mechanisms and discuss its benefits and drawbacks (Section 3). For functional validation, the concept has been implemented in a case study in a real-world public space delivery robot within the scope of a German research project. The project and its main findings are presented in Section 4. Finally, Section 5 concludes this paper and presents the potential for further developments.

This paper extends our previous publication[3] "A Novel Control Architecture for Mobile Robots in Safety-Critical Applications" [7]. We substantially extend that paper by providing an in-depth discussion of requirements for RCAs in safety-critical applications and a detailed assessment of the state of the art regarding these requirements. In addition, we extend the chapter on our case study with an in-depth real-world application example that details the functionality and benefits of our approach.

2. Related work

This section presents an overview of the currently most adopted approaches of RCAs and show to what extent these are suited for safety-critical applications. The underlying concepts that are employed the most in RCA implementations are Hierarchical Finite State Machines (HFSM) and Behaviour Trees (BT) [8]. According to [9], the adoption of HFSMs outweighs BTs by a factor of roughly four, although the number of robotic projects using BTs has been increasing sharply within recent years.

In the following, we will first briefly introduce these two RCA concepts. Secondly, we will give an overview of microservices and show that, even though rarely mentioned by that name, this concept is already currently used in robotics development. Finally, we identify key requirements for mobile robots in safety-critical applications based on our previous research and evaluate the currently most prominent RCA implementations with regards to these requirements.

2.1. Hierarchical finite state machines

Historically speaking, Finite State Machines (FSM) describe a finite set of configurations or *states* that a system may assume during operation as well as a set of *transitions* initiated by an *input* that cause the system to move from one state to another. This concept was initially meant and is still in use for syntax checking and compiling of formal languages where the input is a string of symbols to be checked and the state represents solely a description of the current checking progress indicating whether the input's syntax is correct [10]. The overall logic and structure of the system is defined by the transitions. This concept is often also referred to as *protocol state machine* [11].

In 1987, David Harel introduced *statecharts* as a tool to visually describe a systems functionality in a state-transition-like fashion [12]. He introduced hierarchy to address the most common critique to FSMs, namely, their proneness to high complexity with a growing number of states and transitions. For this reason, statecharts are nowadays mostly referred to as Hierarchical Finite State Machines (HFSM). In contrast to FSMs, HFSMs allow for nested states or states that are made up of a number of child states. Depending on the details of the respective implementation, child states can inherit characteristics such as functionality or transitions. Nested states typically require an initial child state to be defined that the system assumes when transitioning to its parent.

To address an increasing level of complexity, the concept of *orthogonals* have been added. These allow states to not only be made up of more than one child state but also a number of child state machines where the system assumes one state in each of these simultaneously. Harel suggested to model these according to existing hardware components. An event may trigger a transition in any number of orthogonals and the overall state is then defined by the set of assumed state in all orthogonals.

Harel also introduced the concepts of *actions* and *activities* as a way to add functionality to states. Actions refer to pieces of functionality that are executed instantaneously when entering or when exiting a state. In contrast, activities denote functionality that is executed over an extended period of time, for example as long as the system assumes a certain state. Nowadays, this concept is referred to as *behaviour state machines* [11]. It is important to note that Harel introduced a visual representation of such systems and more specifically did not elaborate on how this might translate to the systems software architecture or implementation. We will discuss the current implementation approaches in the separate Section 2.5.

The concept of HFSMs is widely adopted in RCAs. A variety of RCA implementations based on HFSMs can be found in the literature including FlexBE [13], SMACC [14], SMACH [15] and RSM [16]. According to a study conducted in [9], a majority of robotic projects use HFSM-based RCA implementation.

2.2. Behaviour trees

Behaviour Trees have been developed by the video game industry for the control of non-player characters [8,17,18] and have recently found their way into robotics applications [8,19]. Much like HFSMs, architectures based on BTs divide the overall system's process into smaller steps. However, in BTs these are arranged in a tree structure in which the leave nodes or *execution nodes* represent the execution of a specific subtask and non-leaf nodes or *control flow nodes* determine the order in which their child nodes are executed [8]. The system is executed in discrete steps using a tick signal that is propagated through the tree. Execution nodes pass their progress to their parent nodes by returning either a *success*, *failure* or *running*.

Behaviour Trees (BT) can be considered the most prominent alternative to HFSMs. One reason for that is that their tree structure is superior to the highly connected graph structure of HFSMs in terms of scalability and modularity. This is achieved by limiting the outcome of a subtask to be either success or failure. However, this limitation can render BTs less suitable for safety-critical applications in which a more fine-grained distinction of outcomes is required. For this reason, BTs are not considered in the approach presented in this paper. See Section 3 for further elaboration. For the sake of correctness and completeness, it should be stated that it has been shown in [8] that any FSM can be translated into a BT using a global state variable that acts as a GOTO statement. Though we would argue that this would sacrifice all its scalability and modularity benefits.

2.3. Microservices

Microservices have been introduced in the discipline of server development to address shortcomings of the then predominant approach of monolithic software architectures. With increasing complexity, existing

[3] Copyright © 2022, IEEE. Reprinted with permission from IEEE Proceedings.

architectures struggled to meet the required levels of scalability, modularity and maintainability [20]. In microservice-based architectures (MSA), the system is divided into a set of subsystems, each with a well-defined and concise functionality. Microservices communicate using publish–subscribe or request–response patterns to realize the overall system's functionality. To form more complex workflows in such architectures, sequences of microservices are formed either through central *orchestration* or decentralized *choreography* [20]. Having a set of small loosely coupled subsystems instead of a monolithic structure increases the overall systems modularity and adaptability. In doing so, it reduces the required effort to keep the subsystems functionality concise and testable. However, it comes with the same challenges of other distributed systems, for example an increased complexity of the system as a whole and possible inconsistencies in distributed storage [21].

The most prominent robotics discipline in which microservices can be found in the literature is cloud robotics. In these applications, the computations required to carry out the robot's process are partially offset to remote computing units to reduce the overall computational load on the robots hardware [22]. In [23], for example, the mapping and parking process of an autonomous robot is planned in the cloud and only executed by the robot. In [24], such an architecture is used for mission planning and routing for unmanned aerial vehicles. Further examples of microservice based cloud robotics include [25–27].

Disagreeing with [28], we would argue that, without considering cloud robotics, the microservices architecture still has made its way into robotics software development even though one does not come across that name in the literature very often. We base that assumption on the fact that most prominent robotics middleware such as the Robot Operating System (ROS) or Nvidia Isaac follow an approach of distributed software nodes that communicate through network interfaces [5,6] and can be orchestrated through a central unit using HFSMs or BTs. However, we agree with [28] that further guidelines or conventions as to how larger projects should be structured are required. Moreover, RCAs employing the single-method approach negate some of the benefits of the decentralized architecture by forcing developers to implement most of the systems functionality within the orchestration node.

2.4. Requirements for safety-critical applications

We have compiled a comprehensive list of general requirements for a delivery robot developed in a previous research project [29] to be granted permit for operation in German public space. We have extracted the following requirements that address safety demands that fall into the scope of RCAs:

Overridability The robot has to provide the functionality for an operator to override the robot's autonomous behaviour.

Error Indications The robot has to monitor the correctness of its functionality continuously and notify the operator if an error has been detected.

Battery Monitoring The robot has to monitor its battery level continuously and raise an error if the battery is critically low.

Reactiveness The robot has to react to detected hardware and software faults by assuming a dedicated error mode.

Gentle Stop The robot has be able to come to a stop safely at any point in time, most notably in case of a major system failure.

The full list of requirements can be found in [29]. Based on these general requirements, we have derived the following RCA-specific requirements for such robots:

Fail-Safety The *reactiveness* and *gentle stop* requirements above both imply that the robot must be capable of reacting to internal hardware and software faults such that it assumes a fault-specific error mode and comes to a stop if needed. This capability is referred to as *fail-safety* [30]. It follows that the system needs to continue functioning to a certain degree even after hardware or software faults have occurred.

Concurrency A system's level of concurrency refers to its ability to do multiple computations in parallel or, on a broader scale, to execute multiple pieces of functionality at the same time. In real-live robotics applications with strict safety requirements, it is often not feasible to separate the robot's task into strictly sequential subtasks. Considering the requirements above, *error indications* and *battery monitoring* both imply background functionality that is executed to some degree independent of the overall process. For example while the battery monitoring is required during the whole runtime of the process, certain error monitoring functionality might only be needed in a subset of steps of the process. It follows that the system is required to execute a certain amount of background functionality to ensure its own safety as well as the safety of its surroundings without blocking the overall task's progression.

Interruptibility This directly follows from the *reactiveness* and *overridability* requirements mentioned above. In order to properly react to malfunctions, a robot has to be able to interrupt its process at any point in time. The same holds for a manual override of the autonomous functionality. Interrupting the current activity is one of the fundamental capabilities required to implement such mechanisms.

In addition to these requirements directed specifically for control architectures in robotic applications, certain well established goals of general software architecture rise in significance, when deploying a robot in safety-critical applications. These are particularly the ones that increase the efficacy of *a priori* tests:

Testablility A very common approach to ensure a system's correctness and stability as well as to cope with its susceptibility to human error during the implementation process is to cover the codebase with unit tests [31]. A higher degree of testability reduces the effort required to test the system and increases their efficacy. In addition, unit test protocols can be used in technical assessments as part of the system's proof of reliability.

Reusability One way to increase a systems testability is to ensure code reusability or to allow developers to reuse parts of their implemented functionality. Reusing thoroughly tested code wherever possible can drastically reduce the overall implementation effort and thus the system's susceptibility to human error in the implementation process. In robotics applications it is common to see a system executing certain pieces of functionality, such as manoeuvring, object detection or interacting with its environment multiple times along the overall process. Additionally, other functions such as internal diagnostics are required to run continuously across the whole span of the process.

Loose coupling A system is considered loosely coupled if its individual components have few direct references to one another. A loosely coupled code base increases the system's testability by allowing tests to be targeted at smaller, less complex functionality. It can also lessen the impact of internal errors on other subsystems and the RCA itself.

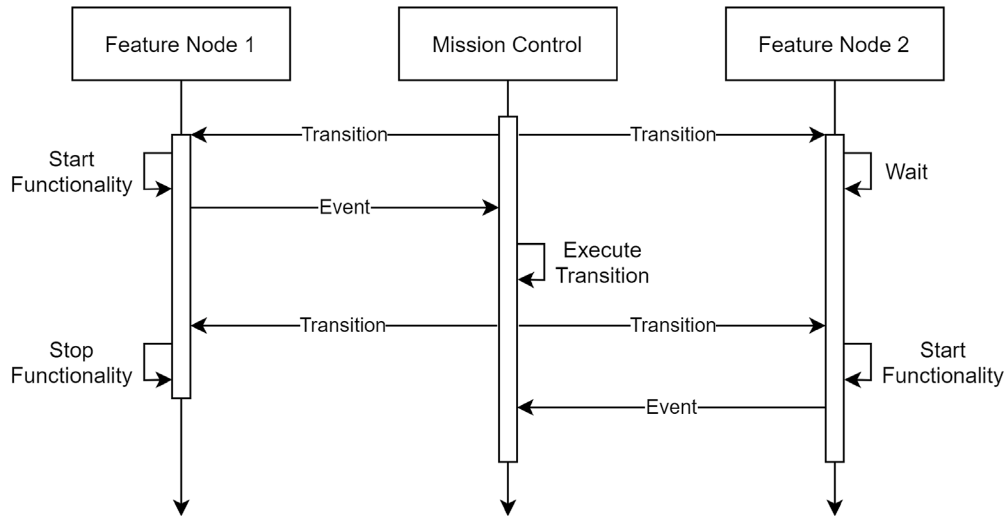


Fig. 1. A flowchart showing how Mission Control and feature nodes communicate through transition and event messages.

2.5. Current RCA implementations

According to the study conducted by Ghzouli at al. in [9], the most adopted RCA implementations are FlexBE [13], SMACC [14], SMACH [15] and RSM [16]. These approaches employ, in essence, a centralized, monolithic and object-oriented approach with one class per state containing a dedicated method in which the functionality of this state is implemented. Based on the return value of this method, the state machine determines which transition to execute next. We will refer to this approach as the *single-method approach*. These implementations do not claim to be safety relevant and promote safety considerations solely on lower control layers such as motor control or navigation. However, our reasoning above shows that RCAs should be part of the safety considerations for mobile robots in more complex environments. In the following, we will therefore evaluate the RCA implementations mentioned above with regards to the requirements compiled in Section 2.4.

When it comes to **fail-safety** with regards to hardware faults, all of these RCA implementations run in the same process and thus on the same computing unit. Without adding an extra architectural layer, a fault in this unit could render the RCA and all state functionality inoperative. The robot would be unable to assume a safe state in a controlled fashion. It follows that none of these implementations allow to ensure fail-safety in case of hardware faults. Looking at software faults, all implementations show a strong coupling of the state machine functionality and the implementation of states. The state machine class maintains a reference to its current state instance and calls the method to execute its behaviour directly. SMACC does provide APIs for exception handling. In the other approaches, exceptions raised in state logic would propagate to the state machine if uncaught in the state itself. It follows that FlexBE, SMACH and RSM do not provide fail-safety with regards to software faults.

With the exception of RSM, all of the considered RCA implementations provide APIs to introduce **concurrency** into the process. SMACC employs orthogonals, whereas FlexBE and SMACH use synchronization states that can be integrated into the process definition. That means that these approaches do provide the infrastructure to execute functionality concurrently, though it requires a certain degree of implementation effort and added complexity to do so.

The degree of **interruptibility** of these systems is directly linked the degree of concurrency in the defined process. Each state that is required to be interruptible, has to be implemented as orthogonal (in the case of SMACC) or contained in a synchronization state (FlexBE, SMACH). It follows that FlexBE, SMACC and SMACH allow for interruptible states, provided the defined process makes use of the respective concurrency API. As mentioned above, this requires extra implementation effort and adds complexity to the overall system.

By employing the single-method approach, all of the considered RCAs encourage developers to join all functionality required in one state into a single method and thus promote a **strong coupling** of inherently independent functionality. This increases the implementation effort required to cover the codebase with unit tests (i.e. its **testability**) as well as the error proneness of the system and the tests themselves. This results in the fact that a majority of robotics code bases seem to be untested [28]. In addition, as mentioned above, all evaluated RCAs show a strong coupling of the states and the state machine itself.

RCAs adopting the single-method approach offer a convenient way to reuse functionality on state level, in that single states can easily be duplicated and inserted at a different part of the process. On a broader scale, they also allow for the reuse of behaviour on a subsystem level, meaning the reuse of a set of connected states, even though this arguably requires some refactoring effort depending on the state machine's interconnectedness [8]. However, their **reusability** is limited in that they do not provide a convenient way to divide states into more atomic functions that run concurrently and to reuse this sub-state-level functionality.

Our conclusion is, that requirements arising in safety-critical applications have not yet been sufficiently met by current RCA implementations. Most notably, fail-safety with regards to hardware faults cannot be ensured in any approach and only SMACC provides the infrastructure to provide fail-safety with regards to software faults. When it comes to concurrency and interruptibility – with the exception of RSM – these implementations do provide APIs to achieve that, although they require extra implementation effort and increase the overall complexity of the system. Finally, the RCAs considered here promote strong coupling of atomic functions that are required in the same state, ultimately decreasing the code base's testability and reusability.

3. The architecture

This section will introduce a new RCA approach, which addresses safety requirements by allowing a higher degree of concurrency, code reusability and testability by combining a HFSM with microservices. In this approach, instead of having a single method for each state,

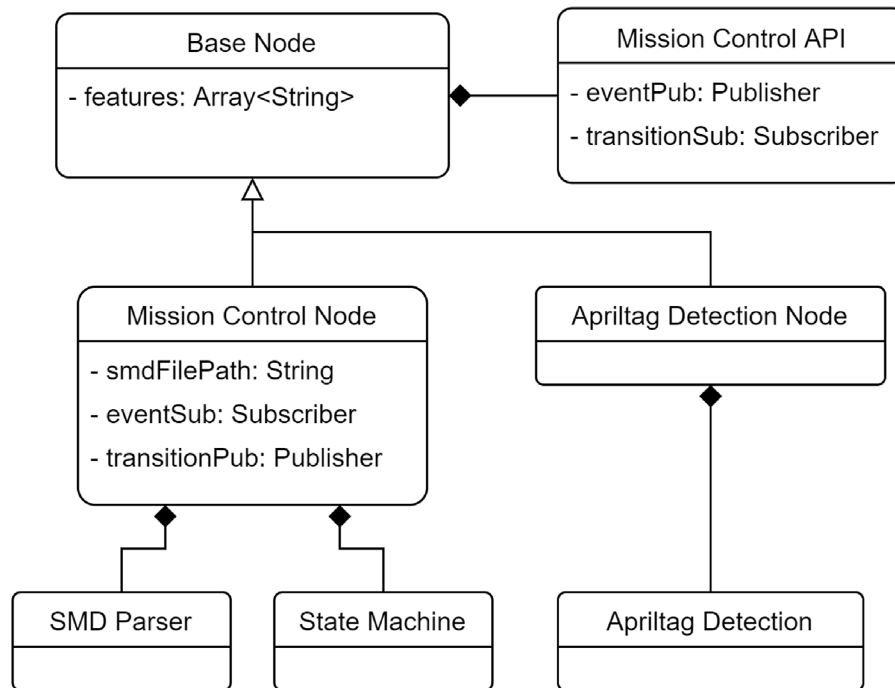


Fig. 2. A reduced UML class diagram of a system with Mission Control and one exemplary feature node for Apriltag detection. There are no dependencies of Mission Control node and feature nodes.

the functionality of one state is defined as a set of smaller functions or *features*. Each feature is implemented by a dedicated microservice or *feature node* and executed in a separate process. Feature nodes are activated and deactivated by a separate node called *Mission Control*, which is responsible for the execution of the state machine functionality itself. Nodes communicate with one another using a one-to-many publisher/subscriber communication pattern. To do so, *messages* are being posted on dedicated *topics* and nodes can subscribe to a topic to receive the respective messages. To carry out the defined process, Mission Control uses one topic to broadcast state transitions, another one to receive events that trigger these transitions. This is shown in Fig. 1.

In addition, this approach further decouples the state machine's functionality and its *State Machine Definition* (SMD). Here, SMD refers the set of states and transitions that make up the representation of one specific process. It is written and stored in a dedicated file. The state machine's functionality, implemented by Mission Control, includes loading the SMD file and based on its contents storing the system's current state and executing transitions according to incoming events. Fig. 2 shows a UML class diagram of a small system with Mission Control and one feature node. The diagram displays the relationships among the different nodes of the system. The exemplary feature node contains some kind of visual sensor functionality to detect Apriltags. This functionality was chosen for a better illustration of the example and a variety of other kinds of nodes would be also possible. One can see that, while Mission Control and the feature node inherit from the same base node class, there are no direct dependencies of Mission Control and the feature node.

The following sections will use the SMD file to discuss the concepts of states and transitions (Section 3.1) and error handling (Section 3.2) in further detail. Mission Control and feature nodes will be introduced in Sections 3.3 and 3.4. Finally, Section 3.5 will discuss the benefits and limitations of this approach.

3.1. States and transitions

The implementation of a state consists of two parts: the list of *features* required in this state and the actual implementation of these features. This section will focus on the first aspect, the latter one will be discussed in Section 3.4.

The list of required or *active* features is part of the state definition in the SMD file. Listing 1 shows an exemplary definition of the state `drive_to_coordinates`. A state definition contains the `active_features` array (lines 5 to 10) in which each feature is referenced by its unique identifier. In this example, these are `autonomous_ride`, `horn`, `localization` and `teleoperation`. Note that no actual implementation is done at this point. As this approach is based on a HFSM, it allows states to be made up of one or more child states. These can be introduced by adding further attributes of the same structure to the parent state JSON object. The name of the attribute then determines the name of the child state. If a state is nested, one child state has to be identified as `initial_state` (line 3) to be assumed on first entry. The `transitions` array (line 4) contains all transitions that originate from one of the children. Child states inherit the `active_features` from their parent state, which allows for a structured and concise definition of each state.

Listing 1: The state definition is made up of the list of active features and optional hierarchy definitions

```
1  {
2    "drive_to_coordinates": {
3      "initial_state": "",
4      "transitions": [],
5      "active_features": [
6        "autonomous_ride",
7        "horn",
8        "localization",
9        "teleoperation"
10     ]
11   }
12 }
```

A transition is defined by its source and its target state as well as the event by which it is initiated. An optional data field allows developers to add static data to the transition that will be passed to the target state. Listing 2 shows a definition going from state `autonomous_ride` (line 4) to state `wait_for_loading` (line 5). When executing this transition, a data field `timeout_in_s` is passed to the target state (lines 6 to 8).

Listing 2: The transition definition contains the source and target state as well as optional static data

```
{
  "transitions": [
    {
      "start": "autonomous_ride",
      "dest": "wait_for_loading",
      "data": {
        "timeout_in_s": 60
      }
    }
  ]
}
```

3.2. Error scenarios

To allow the robot to react instantly and reliably in critical situations, this architecture introduces a global *error state* and *error scenarios*. One can think of an error scenario as an event that requires the robot to react immediately in a certain way, regardless of its current state. Listing 3 shows a definition of an error state with one error scenario in the SMD file.

Listing 3: The definition of the global error state consists of a list of active features as well as the list of error scenarios

```
{
  "active_features": [
    "horn",
    "localization",
    "teleoperation"
  ],
  "scenarios": [
    {
      "name": "controller_connection_lost",
      "trigger": "controller_disconnected",
      "resolve_trigger": "controller_connected",
      "inactive_features": [
        "teleoperation"
      ]
    }
  ]
}
```

Consequently, an error scenario is comparable to a transition in that it is initiated through the same event and `trigger` mechanism as transitions (line 10) and it will force the system to assume a new state, namely the error state. It differs from standard transitions in that its definition does not require a source state. Instead, implicit transitions are created going from each non-error state into the error state, using the error scenario's trigger. This leads to a reduced amount of implementation effort and a more readable SMD file. It further differs from standard transition in that it requires a `resolve_trigger` (line 11). An event containing the `resolve_trigger` of an error scenario indicates that the respective error is resolved and that the robot can resume its previous state.

The global error state is the state the system assumes as soon as one of the defined error scenarios occurred. After assuming this state the system will collect additional error scenarios that occur and will only leave the error state once all error scenarios have been resolved. This allows the robot to automatically run the appropriate error recovery procedures. Much like other states, the definition of the error state contains a list of required features (lines 2 to 6). In the definition of error scenarios, developers can then state features to be removed from that list in case said scenario occurs (lines 12 to 14). This allows the system to deactivate certain functionality once an error has been detected. In the example in Listing 3.2, when a `controller_disconnected` event will cause the system to assume the error state with the features `horn` and `localization` activated and `teleoperation` deactivated due to the error caused by that feature.

3.3. Mission control

The *Mission Control* node is the central orchestration unit of all feature nodes. It is responsible for loading the SMD file and checking its content for syntactic correctness and semantic integrity. After that, the purpose of Mission Control is to keep track of the system's current state, execute transitions based on incoming events and communicate state changes to the feature nodes. Fig. 3 shows how this interaction of Mission Control and feature nodes is realized through the following topics:

The topic `mission_control/state_event` allows feature nodes (see Section 3.4) to post event messages that are used to initiate state transitions. Examples of such events are the arrival at a certain destination, a detected internal malfunction or user input. These messages contain a `trigger` that is matched with the transition triggers mentioned in Section 3.1 to determine the transition to be executed. These messages may also contain optional data in a key–value structure that allows feature nodes to pass on computational results to upcoming states.

The `mission_control/state_change` topic is used to broadcast state change messages of the system. These contain identifiers of the new and previous states as well as the list of features required in the new state and data that has been attached through previous state events.

3.4. Feature nodes

A *feature node* is a node that is responsible for executing the functionality of one or more features. Feature nodes subscribe to the `mission_control/state_change` topic and use the contents of these messages to determine whether or not their functionality is required. They use the `mission_control/state_event` topic to publish events that represent the results of their computation and internal sanity checks. Additional data that is required to execute a node's functionality will be passed in the respective message received through the `mission_control/state_change` topic.

The functionality of a feature node is typically either a one-time computation, such as a network request, or a continuous process based on one or more data streams such as object detection (see Fig. 4).

Nodes of the former nature use the received state change messages to trigger their functionality once and finish their computation by informing Mission Control about their success or failure by publishing an event message. Nodes of the latter kind simply store the information passed in state change messages and use it with each incoming data message to check whether their feature is currently required or not or unsubscribe and resubscribe the respective data stream altogether. These nodes may publish multiple state events, one for each time their computations yielded a desired result.

3.5. Benefits and limitations

In this section we will evaluate the presented approach with respect to the requirements compiled in Section 2.4 and show its advantages as well as limitations that still need to be addressed.

In contrast to existing RCAs, this approach has no state implementation logic in the central control unit. There are no references in

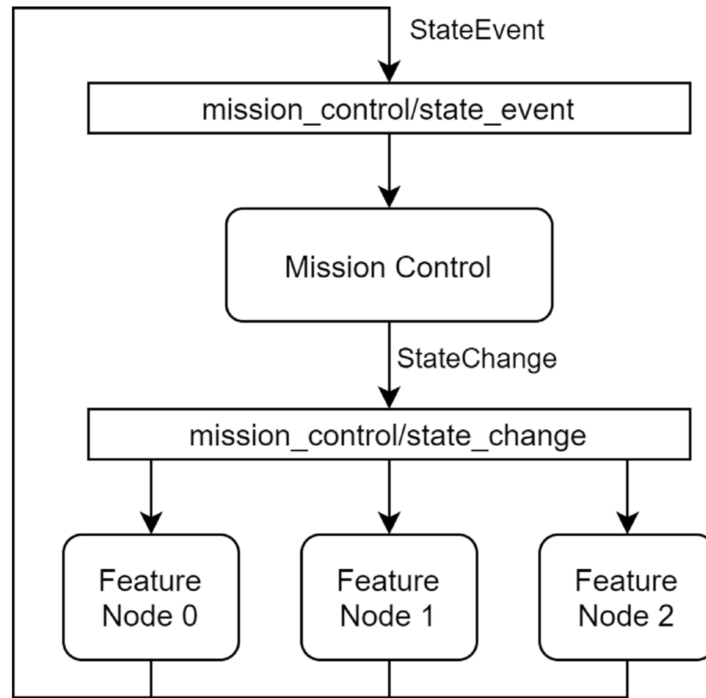


Fig. 3. The Mission Control node and feature nodes communicate through state event and state change messages.

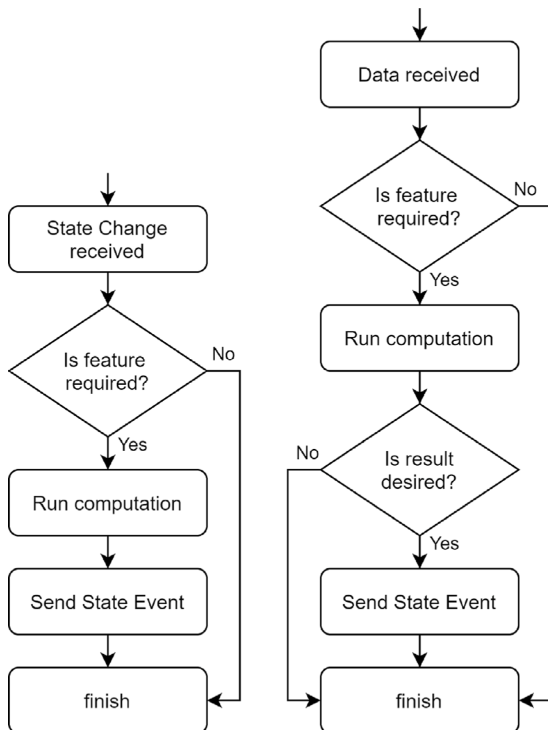


Fig. 4. Feature nodes implement either a one-time computation (left) or a continuous process based on data streams (right).

the presented architecture between state machine and state implementations. The communication of Mission Control and feature nodes is realized solely using network communication. The result of this is that internal errors in state logic (i.e. feature nodes) cannot propagate to Mission Control and thus the RCA provides **fail-safety** with regards to software faults in feature nodes without extra computational effort. In addition, the distributed nature of this architecture allows to execute feature notes on separate computing units and thus to provide fail-safety with respect to hardware faults. However, like other RCA implementations, this approach uses a central control unit that can still act as a single point of failure. Software faults in the RCA as well as hardware faults in the computing unit executing the RCA can render the whole system inoperable. An update of the RCA shown here that is fail-safe with regards to these faults is subject of our current research and will be presented in an upcoming publication.

Our approach introduces **concurrency** at the core of its architecture without requiring extra implementation overhead by implementing each feature in a dedicated microservice and executing it in a separate process. As a result, it provides a fundamental infrastructure for the implementation of runtime fault detection mechanisms. The extent to which this high degree of concurrency introduces additional computational load is still subject of ongoing research.

Since all feature nodes are executed concurrently by default and Mission Control and feature nodes are fully decoupled, state transitions can be initiated at any point in time and by any part of the system. It follows that the process is **interruptible** by default at any time. This property introduces new ways to implement safety features and complex recovery routines into the high-level process.

The explicit separation of feature implementations allows for an easier implementation of unit tests compared to the single-method approach and thus leads to an increased **testability** of the code base. The decoupling of Mission Control and feature nodes allows to conveniently replace feature nodes with mocks for the implementation of integration tests. Additionally, the separation of SMD and its implementation allows for an integration of *a priori* SMD syntax and semantics validation and thus introduces an infrastructure for fault prevention methods on process level. Conveniently, it also drastically reduces the effort to exchange SMDs and thus to switch from one use case to another.

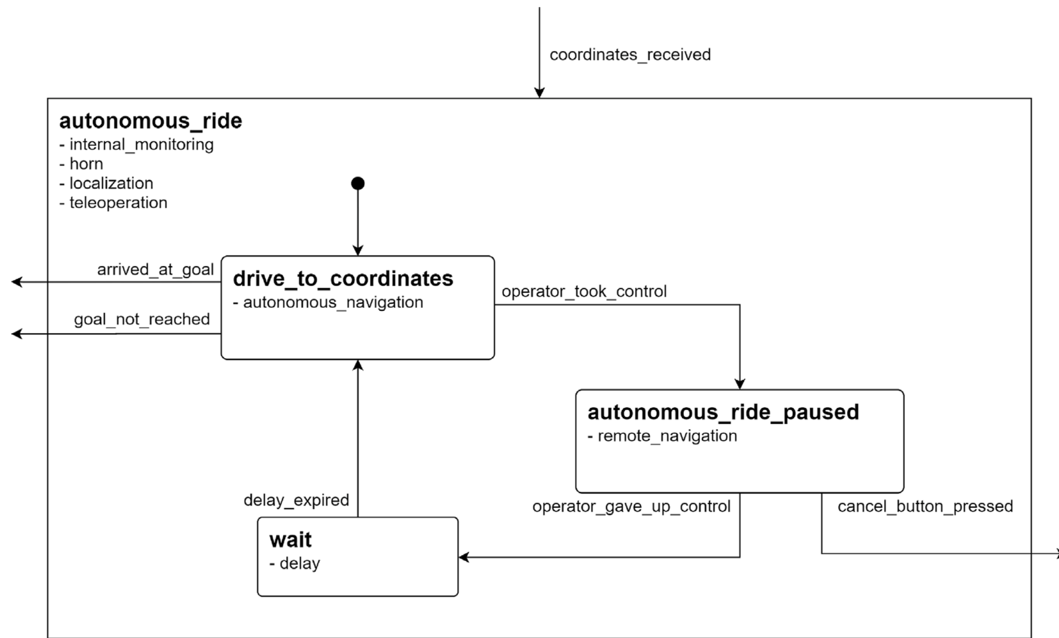


Fig. 5. An excerpt from the SMD used in the TaBuLa-LOG research project showing how the autonomous ride can be interrupted by an operator.

However, the text-base definition can lead to long and complex files. Different approaches such as graphical interfaces or SMD generation should be investigated.

Separating state behaviour into features that are activated and deactivated on demand introduces a new degree of **reusability** by allowing to reuse functionality not only on state level but also on sub-state level. It allows for a full exploitation of the distributed architecture current robotics middlewares such as ROS [5] or ISAAC [6] promote. With the presented approach, functionality can easily be shared across multiple states without requiring any extra implementation effort.

The interpretation of a state as a set of features allows to fully **decouple** the robot's pieces of functionality, even when being used in the same state. As stated above, Mission Control and feature nodes are fully decoupled as well.

In conclusion, the RCA presented above is better suited to meet the increased requirements for mobile robots in safety-critical applications. Most notably when it comes to fail-safety, this is the first approach that allows the development of systems that are fail-safe with regards to software faults as well as hardware failures. Since all robot functionality is executed concurrently this approach provides interruptibility without extra implementation effort. As opposed to the current state of the art, this RCA promotes a loose coupling throughout the whole code base, which increases its overall testability and code reusability.

4. Case study

The RCA concept introduced in this paper has been successfully implemented in the TaBuLa-LOG research project. This section gives an overview of the project and outlines how the RCA was used to meet its safety requirements. In doing so, it demonstrates and validates the functionality of and the advances through the new RCA concept.

In the scope of the TaBuLa-LOG project, an autonomous delivery robot is developed and operated on public roads and sidewalks in the district Duchy of Lauenburg, Germany. The robot delivers internal mail between different branches of the local administration. To cover farther distances, it makes use of an autonomous passenger shuttle that has been implemented in the public transport in a previous project. The results from this actual project are summarized in [29,32].

The implementation of the robot is based on ROS. The presented RCA is implemented in a dedicated ROS node and controls a total of 18 feature ROS nodes providing the functionality of 22 features. The code base was covered by a total of 345 unit tests. The final SMD is made up of 13 states, 21 transitions and 2 error scenarios. In the defined process, 10 features are reused at least once, 4 safety-related features were used in the majority of the process. A teleoperated intervention feature for example is used in 12 states and another internal diagnostics system in all 13 states.

Fig. 5 shows an excerpt of the implemented SMD. It shows the state `autonomous_ride`, in which the robot drives autonomously to a set of given coordinates and how this autonomous ride can be stopped by an operator at any point in time. In the figure, the name of each state is printed in bold and below that the graphic shows the features active in that state. For example in `autonomous_ride` and all of its child states, the features `internal_monitoring`, `horn`, `localization` and `teleoperation` are active. Its initial child state is `drive_to_coordinates`. Fig. 6 shows the corresponding flowchart of the messages that are passed among Mission Control and the most relevant feature nodes *Drive Command Mux*, *Teleop* and *Delay*. The node Drive Command Mux implements the two features `autonomous_ride` and `remote_navigation` by forwarding drive commands from either the autonomous navigation system or the remote control to the motor control unit. The Teleop node generates drive commands and events based on input by the operator and the Delay node allows the system to wait for a given time in order to make certain processes more predictable and controllable. To increase readability, the autonomous navigation system as well as the drive commands that are sent to the Drive Command Mux are omitted. In Fig. 6, the background colours and labels on the left indicate the state of Mission Control at any given time.

When the operator presses the button to take control of the robot, an event with the trigger `operator_took_control` is sent, causing Mission Control to transition to `autonomous_ride_paused` and thus deactivating the feature `autonomous_navigation` and

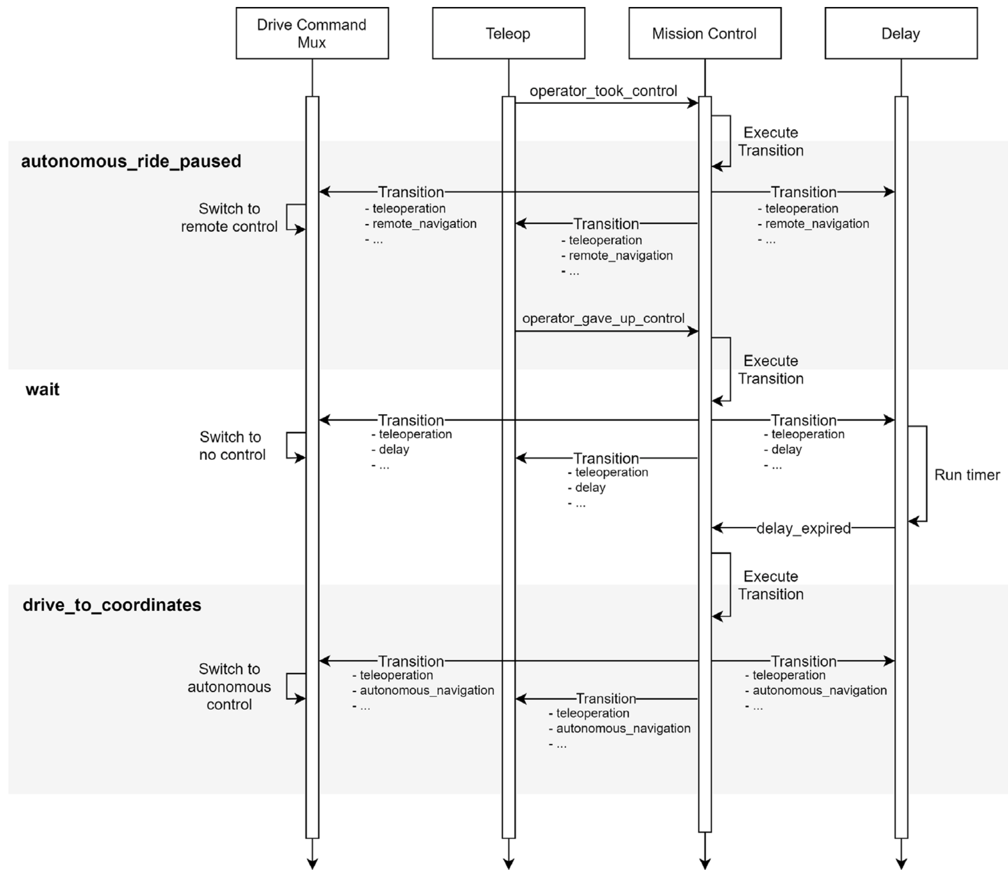


Fig. 6. A flowchart of the execution of the SMD shown in 5 in which the autonomous ride is interrupted by the operator.

activating `remote_navigation`. When receiving the corresponding transition message, the Drive Command Mux node stops forwarding the autonomous drive commands and starts forwarding the ones generated by the remote control. When the operator yields the control of the robot, an event with the trigger `operator_gave_up_control` is published causing Mission Control to transition to the state `wait` and again broadcasting that transition to all feature nodes. In this state neither `autonomous_navigation` nor `remote_navigation` is active so the Drive Command Mux node does not forward any drive commands to the motor control unit. The active `delay` feature causes the Delay node to start a time and send an Event to Mission Control once the timer expires. This event causes Mission Control to transition to `drive_to_coordinates`, activating the `autonomous_navigation` feature and causing the Drive Command Mux to return to forwarding the autonomous drive commands.

For the robot to be allowed to operate legally and safely in public space in Germany it must comply with the respective laws and regulations and be granted permission by the local authorities. Naturally, safety considerations play the main role in the assessment process. During the life span of the TaBuLa-LOG project, German traffic law did not provide a legal framework for fully autonomous driving. Instead, it required an operator to monitor the robot and to be able to take control of it or stop it altogether at any point in time. In addition, a comprehensive safety concept that covers functional safety [33], safety of the intended functionality [34] and cybersecurity [35] is mandatory. A key component of the robot's safety concept is the internal diagnostics system that continuously supervises each of the robot's subsystems. It monitors all components required to carry out the current step of the process and triggers error scenarios and thus safety or recovery protocols in case of failures or malfunctions. The highly concurrent and event-based nature of the proposed RCA allows the system to be interruptible at any step of the process not only by the operator but also by the internal diagnostics system and other safety features.

The final mandatory assessment of the robot and its safety concept was conducted by the German technical inspection agency TÜV Nord. The assessment was carried out in two steps. The first step focused on the robot's hardware design with its range of functions being reduced to teleoperation and internal diagnostics such as monitoring of the connection to the controller. The second step of the assessment covered the automated ride itself and other safety-critical functionality such as object avoidance or right-hand driving. The developed robot passed both steps of the assessment. This demonstrates and validates the fulfilment of the necessary safety requirements of this novel RCA concept.

5. Conclusion

In this paper, a new approach to robot control architectures is presented for addressing safety concerns when operating in public space. These includes fail-safety, interruptibility, concurrency as well as code reusability and testability of the code base. The approach is based on microservices orchestrated by a Hierarchical Finite State Machine building upon the fundamental idea of statecharts from [12]. It is shown that current robot control architecture implementations share

characteristics which we refer to as *single-method-approach* that render these approaches not suited to meet the above mentioned requirements.

In the novel approach presented in this paper, the functionality of states is defined as a set of atomic features that are implemented in separate, concurrent software nodes. The orchestration of these nodes into states is done in a separate State Machine Definition file which is parsed and executed in a dedicated node. This allows for straight forward ways to reuse functionality on a sub-state level as well as an integration of *a priori* validation of the defined process. With the functionality of any state being fully decoupled from the implementation of the robot control architecture and no unnecessary coupling of the state's features, this architecture promotes a concise code base that is easier to test and thus ultimately more stable. Additionally, running all pieces of software in separate processes allows the process to not be blocked by a long running functionality. Instead, it can be interrupted at any point in time and by any part of the system and thus provides a vital infrastructure for implementing safety features and complex recovery routines. With this new approach, this paper presents a template for other robot developers, who are confronted with designing a control architecture for robots in safety-critical environments.

The proposed architecture was implemented, evaluated and demonstrated on a mobile robot in a real-live application in the scope of a German research project. The developed robot successfully underwent and passed the technical assessments required to be granted a permit for operation in public space.

Future work will include an intuitive tool to design State Machine Definitions, as the current textual approach can lead to large and hard to comprehend files. In addition, much like other robot control architecture implementations, this approach uses a central unit responsible for the execution of the State Machine which can act as a single point of failure and thus poses a threat to the overall system's stability. This is subject of ongoing research.

Funding

This work was supported by the German Federal Ministry for Digital and Transport as part of the funding guideline "Ensuring a Viable and Sustainable Mobility System through Automated Driving and Connectivity".

CRediT authorship contribution statement

Manuel Schrick: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Johannes Hinckeldeyn:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Funding acquisition. **Marko Thiel:** Investigation, Writing – original draft, Writing – review & editing. **Jochen Kreutzfeldt:** Writing – original draft, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] P. Salvini, D. Paez-Granados, A. Billard, On the safety of mobile robots serving in public spaces: Identifying gaps in EN ISO 13482: 2014 and calling for a new standard, ACM Trans. Hum-Robot Interact. (THRI) 10 (3) (2021) 1–27.

[2] P. Salvini, D. Paez-Granados, A. Billard, Safety concerns emerging from robots navigating in Crowded Pedestrian Areas, Int. J. Soc. Robotics 14 (2) (2022) 441–462.

[3] M. Thiel, S. Tjaden, M. Schrick, K. Rosenberger, M. Grote, Requirements for robots in combined passenger/freight transport, in: Hamburg International Conference of Logistics (HICL) 2021, epubli, 2021, pp. 195–215.

[4] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, I. Crnkovic, Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective, J. Syst. Softw. 151 (2019) 150–179.

[5] Open Source Robotics Foundation, Nodes – ROS wiki, 2024, URL http://wiki.ros.org.

[6] NVIDIA Corporation, Isaac SDK - NVIDIA documentation center, 2024, URL https://docs.nvidia.com/isaac/doc/index.html.

[7] M. Schrick, J. Hinckeldeyn, M. Thiel, A novel control architecture for mobile robots in safety-critical applications, in: 2022 27th International Conference on Automation and Computing, ICAC, 2022, pp. 1–6, http://dx.doi.org/10.1109/ICAC55051.2022.9911084.

[8] M. Colledanchise, P. Ögren, Behavior Trees in Robotics and AI: An Introduction, CRC Press, 2018.

[9] R. Ghzouli, S. Dragule, T. Berger, E.B. Johnsen, A. Wasowski, Behavior trees and state machines in robotics applications, 2022, arXiv preprint arXiv:2208.04211.

[10] E. Roche, Y. Schabes, Finite-State Language Processing, MIT Press, 1997.

[11] O.A. Specification, Omg unified modeling language (omg uml), superstructure, v2. 1.2, Object Manag. Group 70 (2007).

[12] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274.

[13] P. Schillinger, S. Kohlbrecher, O. von Stryk, Human-robot collaborative high-level control with application to rescue robotics, in: IEEE International Conference on Robotics and Automation, Stockholm, Sweden, 2016, pp. 2796–2802.

[14] smacc.dev, SMACC – State machine asynchronous C++, 2022, URL https://smacc.dev/.

[15] J. Bohren, S. Cousins, The smach high-level executive [ros news], IEEE Robot. Autom. Mag. 17 (4) (2010) 18–20.

[16] M. Steinbrink, P. Koch, S. May, B. Jung, M. Schmidpeter, State machine for arbitrary robots for exploration and inspection tasks, in: Proceedings of the 2020 4th International Conference on Vision, Image and Signal Processing, ACM, New York, NY, USA, ISBN: 9781450389532, 2020, pp. 1–6, http://dx.doi.org/10.1145/3448823.3448857, URL https://dl.acm.org/doi/10.1145/3448823.3448857.

[17] R. Marcotte, H.J. Hamilton, Behavior trees for modelling artificial intelligence in games: A tutorial, Comput. Games J. 6 (2017) 171–184.

[18] Y.A. Sekhavat, Behavior trees for computer games, Int. J. Artif. Intell. Tools 26 (02) (2017) 1730001.

[19] M. Colledanchise, L. Natale, On the implementation of behavior trees in robotics, IEEE Robot. Autom. Lett. 6 (3) (2021) 5929–5936.

[20] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, Present Ulterior Softw. Eng. (2017) 195–216.

[21] J. Soldani, D.A. Tamburri, W.-J. Van Den Heuvel, The pains and gains of microservices: A systematic grey literature review, J. Syst. Softw. 146 (2018) 215–232.

[22] N. Torvekar, S.G. Pravin, Microservices and it's applications: An overview, Int. J. Comput. Sci. Eng. 7 (4) (2019) 803–809.

[23] Z. Yin, J. Liu, B. Chen, C. Chen, A delivery robot cloud platform based on microservice, J. Robotics 2021 (2021) 1–10.

[24] L. Matlekovic, F. Juric, P. Schneider-Kamp, Microservices for autonomous UAV inspection with UAV simulation as a service, Simul. Model. Pract. Theory 119 (2022) 102548.

[25] C. Xia, Y. Zhang, L. Wang, S. Coleman, Y. Liu, Microservice-based cloud robotics system for intelligent space, Robot. Auton. Syst. 110 (2018) 139–150.

[26] A.M. Panchea, D. Létourneau, S. Brière, M. Hamel, M.-A. Maheux, C. Godin, M. Tousignant, M. Labbé, F. Ferland, F. Grondin, et al., OpenTera: A microservice architecture solution for rapid prototyping of robotic solutions to COVID-19 challenges in care facilities, Health Technol. 12 (2) (2022) 583–596.

[27] P. Sabol, P. Sincak, Ai bricks: A microservices-based software for a usage in the cloud robotics, in: 2018 World Symposium on Digital Intelligence for Systems and Machines, DISA, IEEE, 2018, pp. 207–212.

[28] M. Ivanou, S. Mikhel, A. Maloletov, ROS-like framework using modern development concepts and microservices, in: 2021 International Conference "Nonlinearity, Information and Robotics", NIR, 2021, pp. 1–6, http://dx.doi.org/10.1109/NIR52917.2021.9666141.

[29] M. Thiel, J. Ziegenbein, N. Blunder, M. Schrick, J. Kreutzfeldt, From concept to reality: Developing sidewalk robots for real-world research and operation in public space, Logist. J. Proc. 2023 (1) (2023).
[30] T. Stolte, S. Ackermann, R. Graubohm, I. Jatzkowski, B. Klamann, H. Winner, M. Maurer, Taxonomy to unify fault tolerance regimes for automotive systems: Defining fail-operational, fail-degraded, and fail-safe, IEEE Trans. Intell. Veh. 7 (2) (2021) 251–262.
[31] E. Daka, G. Fraser, A survey on unit testing practices and problems, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, IEEE, 2014, pp. 201–211.
[32] M. Thiel, M. Grote, M. Schrick, S. Tjaden, Transport robots in automated shuttles, ATZ Worldw. 124 (4) (2022) 46–51.
[33] ISO, ISO 26262-2018, road vehicles - Functional safety, 2018.
[34] ISO/PAS, ISO/PAS 21448:2019, road vehicles - Safety of the intended functionality, 2019.
[35] ISO/SAE, ISO/SAE 21434:2021, road vehicles - Cybersecurity engineering, 2021.

Manuel Schrick studied Computer Science at University of Lübeck and RWTH Aachen University and is currently working as Ph.D. Student and Research Assistant at the Institute for Technical Logistics, Hamburg University of Technology. His main topic of research is high level control of mobile robots with focus on fault tolerance, error recovery and safety.

Dr. Johannes Hinckeldeyn, Senior engineer at the Institute for Technical Logistics, Hamburg University of Technology. After completing his doctorate in Great Britain, Johannes Hinckeldeyn worked as Chief Operating Officer for a manufacturer of measurement and laboratory technology for battery research. Johannes Hinckeldeyn studied industrial engineering, production technology, and management in Hamburg and Münster.

Marko Thiel studied Mechanical Engineering and Theoretical Mechanical Engineering at Hamburg University of Technology and is currently working as a Ph.D. student and Research Assistant at the Institute for Technical Logistics. His research focuses on autonomous mobile robots in public traffic environments.

Dr.-Ing. Jochen Kreutzfeldt, Professor and Head of the Institute for Technical Logistics, Hamburg University of Technology. After studying mechanical engineering, Jochen Kreutzfeldt held various managerial positions at a company group specializing in automotive safety technology. Jochen Kreutzfeldt then took on a professorship for logistics at the Hamburg University of Applied Sciences and became head of the Institute for Product and Production Management.