Journal of Software Engineering Research and Development, 2019, 7:10, doi: 10.5753/jserd.2019.470

The RoCS Framework to Support the Development of Autonomous Robots

Leonardo Ramos [**Universidade Estadual de Campinas** | ***leo.o.rms@gmail.com***]
Gabriel L. Guimarães Divino [**Universidade Estadual de Campinas** | ***gabriel.lg.divino@gmail.com***]
Guilherme Cano Lopes [**Universidade Estadual de Campinas** | ***gui.c.lopes@gmail.com***]
Breno Bernard Nicolau de França [**Universidade Estadual de Campinas** | ***breno@ic.unicamp.br***]
Leonardo Montecchi [**Universidade Estadual de Campinas** | ***leonardo@ic.unicamp.br***]
Esther Luna Colombini [**Universidade Estadual de Campinas** | ***esther@ic.unicamp.br***]

Abstract With the expansion of autonomous robotics and its applications (e.g. medical, competition, military), the biggest hurdle in developing mobile robots lies in endowing them with the ability to interact with the environment and to make correct decisions so that their tasks can be executed successfully. However, as the complexity of robotic systems grows, the need to organize and modularize software for their correct functioning also becomes a challenge, making the development of software for controlling robots a complex and intricate task. In the robotics domain, there is a lack of reference software architectures and, although most robot architectures available in the literature facilitate the creation process with their modularity, existing solutions do not provide development guidance on reusing existing modules. Based on the well-known IBM Autonomic Computing reference architecture (known as MAPE-K), this work defines a refined architecture following the Robotics perspective. To explore the capabilities of the proposed refinement, we implemented the RoCS (Robotics and Cognitive Systems) framework for autonomous robots. We successfully tested the framework under simulated robotics scenarios that mimic typical robotics tasks and highlight the framework reuse capability. Finally, we understand the proposed framework needs further experimental evaluation, particularly, assessments on real-world scenarios.

Keywords: *Robotics, Software Architecture, Autonomous Computing.*

1 Introduction

With the expansion of autonomous robotics and its applications (e.g. medical, competition, military), the biggest hurdle in developing mobile robots lies in endowing them with the ability to interact with the environment and to make correct decisions so that their tasks can be executed successfully.

Typically operating in the real world — of continuous, unknown, and often unpredictable nature — it is expected that robots act through their perception, reasoning, planning, and the decision-making process to accomplish their goals. With the expansion of cooperative, distributed, and assistive robotics and the widespread utilization of bipedal, aerial, and aquatic robots, other challenges were incorporated, such as multiple robot coordination, human-robot interaction, and three-dimensional control and navigation. These new scenarios demand more complex algorithms and the interaction of various AI (Artificial Intelligence) techniques.

As the complexity of robotic systems grows, the need to organize and modularize software for their correct functioning also becomes a challenge, as information to be processed becomes distributed in space and time. The most desirable qualities for robotics software are modularity, portability, robustness, and reusability for different kinds of robotics applications.

Several architectures and frameworks oriented to robotic systems are available in the literature, e.g., see (Simmons and Mitchell, 1989; Rauch et al., 2012; Jeong and Kim, 2008; Albus et al., 1989; Makarenko et al., 2007; Choulsoo et al., 2010; Malek et al., 2010; Kim et al., 2006; Collett and Macdonald, 2005; Quigley et al., 2009). Nevertheless, when considering heterogeneous robot architectures and applications, it is still an arduous and costly job to reuse existing software, either in partial or complete form, as most with ad-hoc solutions. Furthermore, it is almost impossible to perform fair comparisons of specialized algorithms (e.g., navigation, vision) in a real scenario, when a modular architecture is not available.

In this work, we define RoCS, a development framework to support the current state of autonomous robotics, targeting easier reuse and portability of modules. The framework's architecture instantiates the Autonomic Computing Architecture defined by IBM (2005), known as MAPE-K, under a robotics perspective.

This paper is a revised, extended version of Ramos et al. (2019b). The content has been modified in different ways with respect to the initial version. In particular, we provide the following additional contributions: i) we expand the analysis of the state of the art, discussing the limitations of related work, and justifying our choice of the MAPE-K architecture; ii) we improve the description of the Knowledge component, which was only briefly introduced in the conference version. iii) we clarify the relationship of our proposal to the very popular ROS (Robotics Operating System) middleware, and iv) we improve the evaluation by reporting on two new experiments, involving robots with different physical structures and applications with different tasks.

The rest of the document is organized as follows. Section 2 presents the background and relevant concepts from the robotics domain. Section 3 discusses related work. Section 4 presents our instantiation of the MAPE-K reference architecture. Section 5 presents the actual RoCS Framework for autonomous robots. Section 6 reports on the evaluation of the framework, by applying it to three typical robotics scenarios. Section 7 discusses the relationship of our framework with ROS. Finally, Section 8 presents the final remarks and discusses possible future work.

2 Background

2.1 Service robotics

Robotics has migrated from industrial applications to service robots, where robots help or replace humans in services IFR (2018). In this new scenario, robots are usually autonomous or semi-autonomous, and they have to interact with each other and with humans in dynamic environments efficiently and safely.

The increased complexity of these new applications requires developing new robot platforms and coordinating several modules to accomplish the target tasks, as well as measuring the degree of success. Moreover, robots are often very expensive, and their batteries have short autonomy: two factors that limit the feasibility of extensive physical testing. Therefore, a high-fidelity simulation is commonly applied, and approaches supporting a smooth transition from the simulated environment to the real robot are mandatory.

To foster advancements in service robots, the RoboCup Federation (RoboCup, 2018) has proposed a set of challenges for evaluating the success of developments in different domains. From playing soccer to assisting in typical tasks at home, these autonomous robots need to coordinate a variety of elements to succeed.

RoboCup competitions standardize the tasks that will be addressed, how they will be evaluated, and, in a few cases, which robot platforms are allowed. Still, the job of defining the software and hardware components of the robots is left open.

A quick look at different domains (RoboCup, 2018), such as the *playing soccer task*, where robots can vary from humanoids and wheeled to those with a standard platform, or the assistive robots in *home tasks*, show the variety of solutions that are applied in the software domain to solve the problems.

Because teams that engage in RoboCup challenges participate in various editions, and because the code is necessarily shared among groups after the competition, solutions from some teams become widely used, such as the B-Human (B-Human, 2018) framework. However, the reuse of this framework mostly happens due to the quality of specific algorithms that it implements for solving certain problems, rather than the flexibility and organization of the code itself. We aim, instead, at defining a framework that can guide the user in structuring and reusing its code.

2.2 Robotic Programming Paradigms

The development of control paradigms for robots in dynamic environments has been the subject of research in the field of robotics since the main challenge in the use of robots is how to operate these complex machines and how to coordinate the various elements involved in the operation. The approaches proposed in the literature are usually divided into three paradigms: **deliberative** (Albus et al., 1989), **reactive** (Brooks, 1991; Ranganathan and Koenig, 2003), and **hybrid** (Bayouth et al., 1998; Chan and Yow, 2006).

In the *deliberative paradigm*, the robot uses the available sensory information and its knowledge of the world to reason about it and create a plan. A search is conducted on possible scenarios, to find the one that best fulfills the task. This requires the robot to look ahead and to think about the consequences of each action, which can take a long time. When enough time is available, this approach allows the robot to act accordingly. However, it may not be practical if the robot has to react quickly to environmental changes.

The *reactive paradigm* tightly couples sensory inputs to actuation. It allows the robots to react almost instantaneously to environmental changes and it expects that intelligence emerges from the collective conjunction of very simple behaviors. Typically, the information acquired by sensors is directly used for actuation and it is not retained as internal memory. For this reason, the internal representation of the environment is limited, which prevents long-term planning.

In the *hybrid paradigm*, which is what most of the current architectures classify as, there is a combination of the responsiveness, robustness, and flexibility of reactive systems with more traditional deliberative approaches where reasoning is mandatory. The challenge in this kind of paradigm is solving conflicts between the two different natures, and defining a proper organization of components.

3 Related Work

Several works suggest a structured approach to the control of robots, including middleware (Magyar et al., 2015), frameworks, and architectures. In this section, we discuss those that are most closely related to our proposal.

The Robot Operating System (ROS) (Quigley et al., 2009) is a set of software libraries and tools for robot development, which provides the functionality typical of an operating system for a heterogeneous cluster of robots. ROS has gained popularity because it abstracts the hardware devices, also providing a low-level implementation of commonly used features, message-passing between processes, and package management. This way, it is compatible with multiple simulators and robot models (i.e., physical architectures including its dimensions, sensors, and actuators). However, it provides the basic software components of a robot without necessarily prescribing an architecture, and it is limited to Unix-compatible platforms. Other frameworks that adopt a similar approach are Player, (Collett and Macdonald, 2005), ORCA (Makarenko et al., 2007) and OPRos (Choulsoo et al., 2010). ROS is *de-facto* one of the most adopted frameworks for robot development. We provide a deeper discussion of the relationships between ROS and our proposal later in Section 7.

The Task Control Architecture (TCA) (Simmons and Mitchell, 1989) consists of *task*-specific modules connected to a central *control* module. The task modules perform all the required processing and communicate with the control module via messages. The control module routes these messages to their destination and it maintains task control information. The architecture defines control constructs to support both deliberative and reactive behaviors. TCA provides a set of commonly needed mechanisms, such as task decomposition, resource management, execution monitoring, and error/failure recovery. Although the TCA facilitates the modular and incremental design of complex robot systems, the centralized control makes this component a single point of failure, compromising the robot reliability. Additionally, it turns into a potential performance bottleneck as the robot complexity and functions scale, not being able to handle all the incoming messages at the desired rate. Therefore, a more decentralized architecture is needed.

To dilute the centralization problem, *layered architectures* were proposed (Rauch et al., 2012; Jeong and Kim, 2008). Their main goal is to further modularize and increase flexibility by organizing the system into layers with related functionality. Each layer should address only specific subsystems, hardware platforms, environment, or the robot's end goals, maintaining its contents non-accessible to the other layers, except through message passing. In this sense, lower layers provide core services to the higher tiers, while upper layers perform global analysis and determine the actions to be performed by lower layers. Three to four layers are typically used, depending on the implementation. In practice, layers are organized differently, based on the kind of robot and tasks, which leads to a large variety in architectures and prevents reuse. Also, crosscutting concerns like the robot and world models may be scattered throughout the layers, hampering their maintenance and evolution.

The *4D/RCS* reference architecture provides a theoretical foundation for engineering software for unmanned vehicle systems (Albus et al., 1989). The architecture consists of a multi-layered hierarchy of computational nodes, each having the capability of observing the world, self-orientation, decision-making, and autonomous action. These capabilities are organized in a decision cycle known as the OODA-loop: observe, orient, decide and act. It is realized by five elements for each node:

- **Sensory Processing**: a set of processes by which sensory data interacts with *a priori* knowledge to detect or recognize useful information about the world;
- **World Modeling**: builds, maintains, and uses a world model to support behavior generation and sensory processing;
- **Value Judgment**: computes cost, risk, and benefit of actions and plans, estimates the importance and value of objects, events, and situations, assesses the reliability of the information, calculates the rewarding or punishing effects of perceived states and events;
- **Behavior Generation**: plans and controls actions designed to achieve behavioral goals;
- **Knowledge Database**: data structures and the information content that collectively form the intelligent system's world model.

The behavior generation module of a node is connected to those of the adjacent nodes, creating a command tree. Also, the knowledge database is shared between all the world modeling elements within the same sub-tree and layer. The lower levels of the hierarchy, responsible for fast planning, generate goal-seeking reactive behavior. At higher levels, responsible for the long-term plan, nodes enable the goal-defining deliberative behavior.

Overall, The *4D/RCS* architecture focuses on the system-level conceptual organization of cooperative robots, rather than the software architecture itself. Besides, although highly modular and flexible, it suffers from excessive fragmentation. Thus, its complex hierarchy requires synchronization among several nodes realizing the OODA-loops that, ideally, should be implemented as an independent thread or process. Such a complex hierarchy imposes performance issues due to the synchronization overhead.

Similarly to our proposal, the authors of (Klös et al., 2015) also propose an extension of the MAPE-K architecture; the focus is however different. The work in (Klös et al., 2015) focuses on self-adaptive systems, like the original MAPE-K proposal, and proposes extensions to supporting the adaptation of the adaptation algorithm itself. A proof of concept based on the Communicating Sequential Processes (CSP) process calculus is provided. In this work, we focus instead on the domain of autonomous robots (which are not necessarily *self*-adaptive), and we propose a general framework to support the development of software for such platforms. Specific algorithms for (self-)adaptation are out of the scope of this paper.

The Object Management Group (OMG) is a consortium that develops standards to promote interoperability, portability, and reuse in complex systems. Among others, the OMG is responsible for maintaining the specification of the well-known Unified Modeling Language (UML). The OMG has recently established a "Robotics Domain Task Force", whose objective is to develop such kind of standards for the robotics domain.

Two of the standards released by OMG are closely related to our proposal. The Robotic Interaction Service Framework (RoIS) (Object Management Group, 2018) defines a framework for the development of service robotics applications, focusing on tasks related to the interactions with humans (e.g., person detection, gesture recognition, etc.). It includes a platform-independent model, specified with UML diagrams, and a platform-specific realization specified in C++. The RoIS proposal aims to standardize the data exchanged between robot components, for example, the kind and content of messages. Conversely, our proposal addresses the software architecture, and it is not application-specific.

The Hardware Abstraction Layer for Robotic Technology (HAL4RT) Object Management Group (2016) defines an Application Programming Interface (API) for the development of robotics applications and control software systems. At the time of writing, HAL4RT is still in version 1.0/Beta (Object Management Group, 2016). HAL4RT has some similarities with our work; for example, it also defines an example interface for sensors and actuators, and it provides examples of recommended flows of execution. However, it does not address the analyze and plan aspects, and it does not provide guidelines for the implementation of deliberative and reactive behaviors. In this perspective, HAL4RT is closer to the

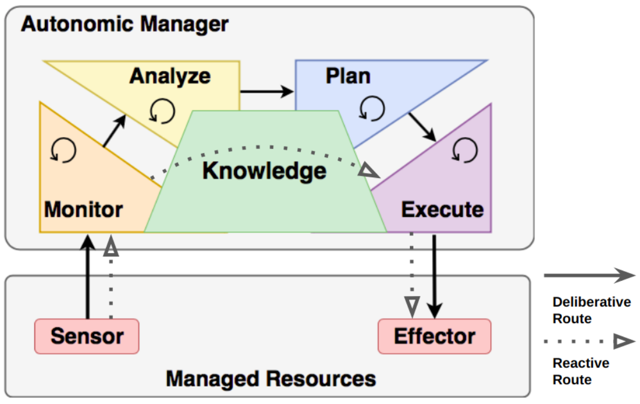


Figure 1. Autonomic Manager in the MAPE-K Architecture [adapted from (IBM, 2005)].

purpose of ROS, i.e., to act as middleware and abstract from the hardware devices.

4 Framework Architecture

The lack of reference software architectures for autonomous robots led us to base our work on the more general Autonomic Computing reference architecture, known as MAPE-K (IBM, 2005), which is summarized in Section 4.1. Then, Section 4.2 discusses the details of our architecture and the choices that we made in its definition. Finally, Section 4.3 discusses how the proposed architecture supports the hybrid robotics paradigm.

4.1 The MAPE-K Reference Architecture

Autonomic, or *self-adaptive*, systems are intended to continuously adjust its operation in response to changes perceived in themselves or in the environment, with minimal outside intervention. To achieve this, in the MAPE-K architecture, systems are composed of autonomic managers and the associated managed resources (Figure 1).

According to the MAPE-K architecture (IBM, 2005), the autonomic manager should be composed of five basic building blocks (Figure 1): *Monitor*, *Analyze*, *Plan*, *Execute*, and *Knowledge*. Besides them, the *Sensor* and *Actuator* (Effector) touch-points work as supporting components for sensing (data collection) and acting upon the managed resources, respectively.

This reference architecture was designed to deal mostly with IT systems, like business information systems, distributed services, and web applications. Later, MAPE-K extensions for adaptive systems such as FORMS (Weyns et al., 2010) and ActivFORMS (De La Iglesia and Weyns, 2015; Qasim and Kazmi, 2016) have been proposed. However, although the authors of these works use robotics scenarios, their proposal advocates an extension that allows the agent to adapt itself to a changing environment, rather than addressing the typical problems of autonomous service robotics presented in Section 1. Accordingly, there is a gap when applying MAPE-K concepts to robotic systems.

4.2 Detailed RoCS Architecture

In this section, we present the instantiation from the MAPE-K reference architecture (Section 4.1) building blocks considering the robotic perspective.

The **Monitor** block (Figure 2c) gathers and interprets raw data incoming from Sensors. One or more `SensorDriver` interact directly with physical sensing devices through the `monitor` port. The information and configuration of the required sensor(s) are obtained from a model (i.e., description) of the physical robot, which is stored in the knowledge source and it is accessed through the `acessKnowledge` port. The actual reference to the needed implementation of the driver is also retrieved in the same way. The `RawDataInterpreter` translates raw data into final values, e.g., voltage readings from a temperature sensor into the actual temperature value. Finally, the `MonitorPublisher` module is the one responsible for publishing the interpreted data through the `publish` port.

Observed data from Sensors can be of diverse types, structures, and dynamics. The sensors can be either real or simulated, and their communication protocol and data nature (e.g. analogical vs. digital) may vary enormously. Regardless of sensors' characteristics, a Monitor block must rapidly organize, interpret, and forward the data for being analyzed.

The **Analyze** block (Figure 2a) filters, processes, and aggregates the data received from the Monitor block to determine if a change in the world has happened. It is where complex algorithms like time-series forecasting, queuing models, and advanced filtering can be executed to analyze the monitored data.

The `MonitorSubscriber` block receives data from the `subscribe` port. The `DataProcessor` module runs data processing algorithms to search for alterations with respect to the known world state; afterward, the `DataAggregator` module translates the results to application-level concepts, possibly fusing the results of multiple data processing algorithms. Both these blocks can access and store information into the knowledge, through the `accessKnowledge` port. Finally, if a relevant change in the world is perceived, the `AnalyzePublisher` module notifies this information to the next block, which can elaborate a response accordingly.

The **Plan** block (Figure 2b) creates and selects the behaviors to be executed, and forwards them to the `Execute` block. Specific changes trigger the generation of new behavior. The realization of this block may range from simple Finite State Automata to complex Cognitive Systems (software architectures inspired in the human brain that mimics, in some ways, human intelligence capabilities) fine-tuned for specific tasks.

First, the block receives the changes detected by the `Analyze` block, via its `subscribe` port, which is delegated to the `AnalyzeSubscriber` module. The received change is sent to the `Planner` module, which is the module that decides which behaviors the robot should execute based on the tasks at hand and occurred changes. The definition of the new plan may also involve accessing the knowledge (`accessKnowledge` port), for example, to assess whether a known plan that has successfully solved the same problem in the past exists. The new planned behavior is specified as a sequence of one or more *actions*, e.g., head movement or team

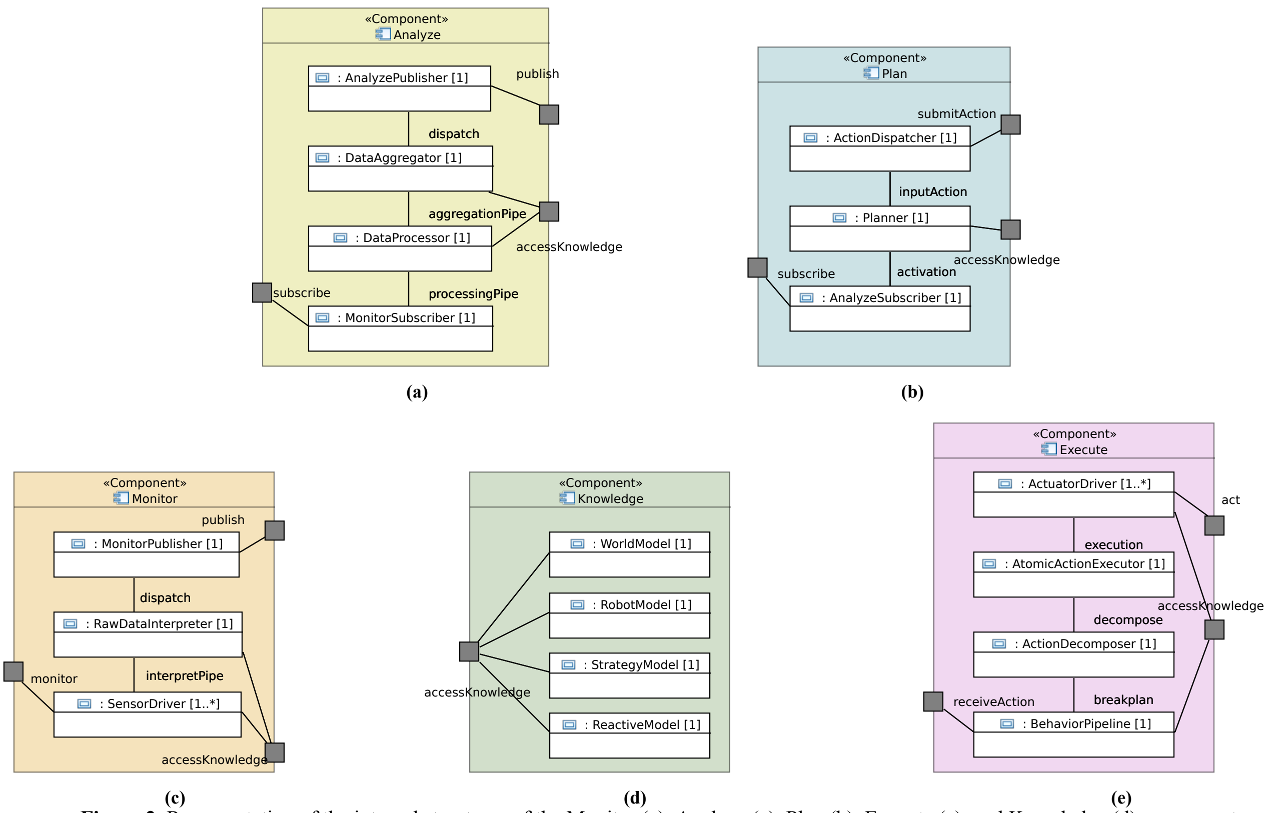


Figure 2. Representation of the internal structures of the Monitor (c), Analyze (a), Plan (b), Execute (e), and Knowledge (d) components.

communication. These actions are enqueued to a pipeline maintained in the `Execute` block (`submitAction` port).

In the **Execute** block (Figure 2e) the planned actions are retrieved from the pipeline actually executed and, if needed, they are transformed into lower-level actions. In fact, the execution of a high-level action may require the execution of multiple, smaller, actions to concretely change the world state and thus realize the intended behavior.

The actions received from the `Plan` block are maintained in the `BehaviorPipeline` module. The `ActionDecomposer` retrieves the next action and, if needed, decomposes it into a sequence of atomic actions. Then, each of these actions is executed by the `AtomicActionExecutor`, which is in charge of sending the correct message to the specific actuator drivers. One or more `ActuatorDriver` modules then directly communicate with the physical actuators. This is repeated until no atomic actions are left, after which the next behavior is extracted from the pipeline. This block is also responsible for any outgoing communication to other robots, via TCP, UDP, or other protocols. Similarly to the `SensorDriver`, the information, and configuration of the actuator, as well the actual reference to the needed implementation of the driver, are retrieved from the knowledge, through the `accessKnowledge` port.

The **Knowledge** (Figure 2d) source is a grouping of data structures designed with the sole purpose of providing access to the world information to all the building blocks. These data structures can be registries, dictionaries, databases, or any other type with designed syntax and semantics. As mentioned above, all the other four MAPE components have access to it, through the `accessKnowledge` port. Information in the `Knowledge` block is organized into four structures.

The `WorldModel` represents any data that the robot can perceive from the world, such as the temperature, a map of the robot's surroundings, or the position of other robots. The `RobotModel` deals with information about the robot structure and its internal configuration, such as the parameters of its actuators, kinematics, and inverse kinematics Mathworks (2018). The `StrategyModel` is a mapping of pre-defined strategies from which the robot can choose for executing its tasks and behaviors if they exist. For example, the "defender" or "attacker" roles in case of football competition. Finally, the `ReactiveModel` is responsible for storing the rules that should trigger a reactive action of the `Execute` block.

In robot swarms or other cooperative scenarios, agents would share this `Knowledge` source, or part of it, among all of them. Each robot would be responsible for sending any newly acquired world information to the other robots it knows until all robots share a common database.

4.3 Design for the Hybrid Paradigm

For the architecture to be consistent with robotics practices, it has to obey one of the Robotics Programming Paradigms (Section 2.2). We designed our architecture to support the *hybrid* paradigm, as it represents the current state of the art, and it is the most flexible solution. In the following, we discuss how our architecture supports both the *deliberative* and the *reactive* routes.

Deliberative Route. To implement a deliberative system, an agent has to go through the following phases: sensory input acquisition, task generation and behavior filtering, action selection, and action execution De Silva and Ekanayake (2008). In the proposed architecture, the first step is performed by the `Monitor` and the `Analyze` blocks. Meanwhile, the `Plan` block is responsible for generating the task plan, filtering the robot's behavior, and selecting the actions to be performed. Finally, the `Execute` block executes them. Therefore, the full MAPE-K classical cycle (solid arrows in Figure 1) works as a Deliberative System.

Reactive Route. A reactive robot system tightly couples perception to action without the use of intervening abstract representations or history (Arkin, 1998). The selection of the reaction to the sensory input is done in an inhibitory way, in which a hierarchy of behaviors is defined and then enacted by actuators, bypassing the planned actions from deliberative behaviors.

Such a mechanism can be defined in our architecture, although a different abstraction is needed. There are two ways of implementing a reactive system within the MAPE-K architecture: i) implementing a high-priority reactive layer in the `Plan` block or ii) capturing a low-level view of the world in the `Knowledge`, based on which a reactive action can be triggered immediately.

Biologically, a reactive action, like the reflex arc, is processed by a neural pathway that can act on an impulse without the assistance of the brain. That is, the response to specific stimuli does not need conscious thought. From the computational point of view, the deliberative planning and the reactive behavior will run independently, at different frequencies, and based on different data.

Therefore, the first abstraction (high-priority module in the `Plan`) could not be interpreted as biologically correct, as the sensory input would reach the `Plan` block, which can be thought as the robot's conscious brain. The second abstraction, instead, considers the `Knowledge` Source as that neural pathway, connecting the receptors (the `Monitor` block), to the effectors (the `Execute` block). The `Execute` block thus triggers pre-defined reflexes in case specific changes in the `Knowledge` Source occur (dashed arrows in Figure 1).

5 The RoCS Framework

Based on the software architecture described in Section 4, we designed and implemented the *RoCS Framework*, a concrete framework to guide robot developers in structuring their code. The framework has been implemented in C++, which is one of the most popular languages in the robotics domain, and its source code is available on the GitHub platform (Ramos et al., 2019a). The structure of the framework is shown in the Class Diagram in Figure 3 and it is described in the following.

5.1 Framework Structure

The core of the framework consists of one class for each of the five main blocks of the architecture: `Monitor`, `Analyze`, `Plan`, `Execute`, and `Knowledge`, depicted in Figure 3 with their respective colors as used in Section 4.

While in general there may exist multiple independent instances of the `Monitor` and `Analyze` blocks, this is not the case for `Plan`, `Execute`, and `Knowledge`. For this reason, these latter classes apply the *Singleton* pattern. In fact, it is reasonable to constrain the user of the framework to have i) a single consistent knowledge base (`Knowledge`), ii) a single place where robot reasoning is performed, to avoid the production of conflicting plans (`Plan`), and iii) a single engine responsible for executing the planned actions (`Execute`).

The `Monitor`, `Analyze`, and `Plan` classes communicate using a *publisher-subscriber* messaging pattern. As such, each of them extends the `Publisher` and/or `Observer` abstract classes, which realize communication channel. However, it should be noted that the `attach` operation, which is needed to register an observer in the list of recipients of a publisher, is not present in the `Publisher` class itself. Instead, the more specific `attach(Analyze)` and `attach(Plan)` methods are added to the `Monitor` and `Analyze` classes, respectively. This choice constrains the user of the framework to follow the proposed architecture, preventing him from connecting blocks in an arbitrary way. Otherwise, with a generic `attach(Observer)` it would be possible to make the Plan observe the Analyze, or even an Analyze to observe itself.

Communication between the `Plan` and the `Execute` occurs through a queue, implemented by the `Pipeline` class. This queue contains a list of actions that are meant to be executed, as objects of type `Action`. Because the possible actions depend on the kind of robot and scenario at hand, the `Action` class is abstract and is meant to be extended by users of the framework. The queue is actually a *priority queue*, to allow the reactive behavior to override the decisions taken by the Plan block (deliberative behavior), by submitting actions with a higher priority. The priority is assigned to actions when they are inserted in the queue. Two levels of priority are supported: *high* and *normal*. The action that is selected for execution is thus the first action with *high* priority in the queue or, if no high priority actions exist, the first action with *normal* priority. When the reactive action route is active, the pipeline is cleared and then re-plan. The `Execute` class owns the Pipeline, which is also a *Singleton*.

The `Knowledge` class maintains a knowledge base that is shared among all the other blocks. It is composed of four main parts. The `WorldModel` class contains a representation of the world as known to the robot. It is also a *Singleton*, to avoid inconsistencies. We just provide an abstract class to be extended by the user, since this is application-specific. The `StrategyModel` class is meant to contain pre-defined strategies, if they exist, to be accessed by the `Plan` block. Its usage is optional and it also widely dependent on the target application. As such, we provide an abstract class to be extended by the user.

The `RobotModel` class contains a description of the physical structure and status of the robot, in terms of its position

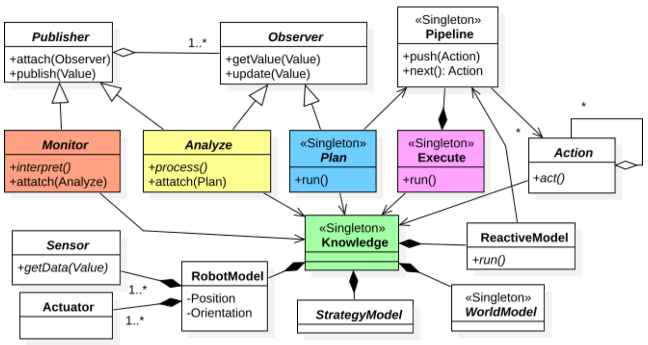


Figure 3. Class Diagram of the RoCS Framework.

(coordinates), orientation, as well as the available sensors and actuators. The information about available sensors and actuators is maintained by aggregation with objects of type `Sensor` or `Actuator`. These two classes are also abstract.

The `Sensor` class has a `getData(Value)` method, which is used to read the data from the sensor. `Sensor` is a C++ *template* class, parameterized with the `Value` type, in order to support sensors returning different kinds of data. Implementation of the `getData` method is however specific for each kind of sensor. A Monitor block may monitor the value of one or more sensors (e.g., a group of sensors of the same type), as represented by the association between the `Monitor` and `Sensor` classes. In general, the opposite is also true: the value of a sensor can be used for multiple different purposes requiring different processing paths.

The `ReactiveModel` class represents a portion of reactive behavior, that is, a known deterministic mapping between input observed at sensor and actions to be executed by the robot. In general, the `Knowledge` may contain multiple instances of `ReactiveModel`, each one addressing a specific aspect of the robot's behavior, e.g., a reactive model to avoid falling, and another one to avoid obstacles. The `ReactiveModel` class accesses sensors via the `Knowledge` class and, when specific thresholds are exceeded, it issues actions directly to the `Pipeline`.

5.2 Control Flow

In the *initialization* phase, the setup of the robot configuration is performed. This involves creating instances of the required blocks, that is, the `Knowledge` (and consequently its parts), the `Execute`, the `Plan`, and a certain number of `Sensor`, `Actuator`, `Analyze`, and `Plan` instances, according to the physical structure of the robot and to the target application.

Then, the required associations are established, which means: i) associating each `Monitor` with the `Sensors` it will monitor, ii) subscribing each `Analyze` to the publishing channel of one or more `Monitor`, and subscribing the `Plan` to the publishing channel of one or more `Analyze`.

The *execution* is multi-threaded, that is, there is a thread running for each instance of the main blocks of the architecture: `Monitor`, `Analyze`, `Plan`, and `Execute`. All of these threads may access the `Knowledge` to read or write information from/to the knowledge base, typically the `WorldModel` or the `RobotModel`. A separate thread is also running for each existing `ReactiveBehavior`.

A `Monitor` thread cyclically runs the `interpret` abstract method, in which it is supposed to read the sensors to which it is connected, and to return the value to be published to its observers. The implementation of this function is, of course, dependent on the application and should be implemented by the user. The publishing part is instead handled by the framework, transparently to the user.

An `Analyze` thread cyclically runs the `process` abstract method. In this method, the data values received from the `Monitor` instances should be merged and processed to return high-level information, which is then published towards the `Analyze`. Also, in this case, the publishing part is handled by the framework.

The `Plan` thread cyclically runs the `run` abstract method, which is meant to be implemented by the user of the framework. In this method, the actual planning is performed, based on data received from the various `Analyze` instances. The results of planning are one or more `Action` objects, which are enqueued to the `Pipeline`. An `Action` can be atomic, when its behavior is entirely defined in its class, or can be composed of a sequence of other simpler actions (macro-action).

The `Execute` thread cyclically runs the `run` method. In this case, the concrete implementation of this method is provided by the framework, and `Execute` is a concrete class. The behavior is simple: at each iteration, it gets the next `Action` in the pipeline and executes it, by calling the `act` method. This method is abstract and its implementation depends on the concrete kind of action. A macro-action would typically call the `act` operation on its children according to a specific sequence, possibly complementing it with additional behavior.

Finally, threads corresponding to `ReactiveModel` instances also run periodically, executing the `run` operation. At each iteration the reactive model reads the value of the sensors of its interest (e.g., a gyroscope to avoid falling) and, in case specific conditions are met, it issues a new high-priority `Action` that is added to the `Pipeline`. Such conditions are again application-dependent.

5.3 Extension Points

The user of the framework may extend the provided classes to implement the behavior of a specific robot. We organize the extension points of our framework in four categories: *mandatory*, *deliberative*, *reactive*, and *optional*.

Mandatory. The user must extend the `Sensor` and `Actuator` abstract classes, implementing concrete classes based on the specific scenario. It must provide an implementation of the `getData` method in `Sensor`, and a method to control the `Actuator`. The user must also extend the `Action` abstract class with concrete classes, based on the kinds of actions that are possible in the considered scenario, and provide an implementation of the `act` method for each of them.

Deliberative. These are extension points that must be used when the robot should implement a deliberative behavior. In this case, the user must extend the `Monitor` and `Analyze` classes and implement the `intepret` and `process` methods, respectively. The user may create a hierarchy of classes, in case different data should be monitored or analyzed separately. To implement deliberative behavior, the user must also extend the `Plan` class and implement the `run` method.

Reactive. These are extension points that must be used when the robot should implement a reactive behavior. In this case, the user must extend the `ReactiveModel` class and implement the `run` method. The user may create a hierarchy of classes extending `ReactiveModel`, in case different kinds of reactive behaviors should be executing concurrently.

Optional. If needed: i) the `RobotModel` class can be extended to contain further properties that reflect the current status of the robot; ii) the `WorldModel` class can be extended to contain properties that reflect the current status of the world as known to the robot; and iii) the `StrategyModel` abstract class can be extended to contain predefined strategies to be

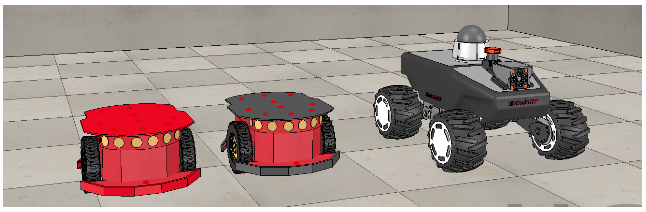


Figure 4. The three robots involved in the experiments, in the V-REP simulated environment. From left to right: R1) the *Pioneer P3DX* running a simple on-board/hard-coded script, R2) the *Pioneer P3DX* running RoCS, and R3) the *Robotnik* robot (running RoCS).

accessed by the Plan. Whether these extensions are needed or not depends on the application.

6 Evaluation

To assess the flexibility of the framework, and evaluate its applicability in typical robotics tasks, we conducted three different experiments involving two different robot models, the *Pioneer P3DX* and the *Robotnik SUMMIT-XL*.

6.1 Robot Configurations

Experiments were run in the V-REP (Rohmer et al., 2013) simulator, which simulates the interaction with the real world (i.e., sensors and actuators). The actual robot code was implemented using the RoCS Framework. Along with the two robots running RoCS, which are the targets of the experiments, the scenarios include a third robot that is used as a distractor, to add more complexity to the system dynamics. The three robots that are used in the experiments are depicted in Figure 4, and their architectures used in the simulations are briefly introduced in the following.

Pioneer P3DX. The robot uses 16 range sensors (sonars), distributed around the robot circumference (yellow dots in the figure). For simplicity, we add 1 position and 1 orientation simulated sensors, which are aware of the real position (x, y, z) and orientation (θ) of the robot. These special kinds of sensors are provided by the simulator; in a real deployment, they would be replaced by a localization algorithm running in the `Analyze`. As actuators, the robot uses two rotation motors that are responsible for driving the differential robot across the scene.

Robotnik. The robot uses three frontal range sensors, one position, and one orientation sensor, one robot sensor (to see other robots). The Robotnik is also a wheeled robot and it is moved by rotation motors. Differently from the Pioneer P3DX however, it uses four motors.

6.2 Experiments Planning

The objective of the experiments is to evaluate the flexibility of the framework according to a different perspective, in particular, we want to answer the following questions:

Q1. Is the framework able to support the hybrid approach (i.e., deliberative and reactive)?

Q2. What is the effort needed to reuse existing code written in RoCS for a different robot model (i.e., different physical architecture)?

Q3. What is the effort needed to reuse existing code written in RoCS for a different application (i.e., different tasks)?

To evaluate the capability of the framework of supporting a hybrid approach (Q1), we defined tasks where both the deliberative and reactive routes get activated. For instance, if the robot task is to "*reach a specific position in the environment while avoiding collisions*", the robot will have to commute between the two routes to successfully accomplish the task.

In fact, in the planned scenario the robot is not aware of the existence of obstacles while it computes its plan to reach the goal (*deliberative* route). Therefore, actions to avoid obstacles will not be included in the plan, and they will be executed in a *reactive* fashion. Similarly, when there are other moving robots in the scene, the robot cannot anticipate their behavior.

To evaluate the effort needed to reuse the code for different robot models (Q2) we run the same experiment first using the *Pioneer P3DX*, and then using the *Robotnik*. This involves adapting the code to different numbers and kinds of sensors and actuators.

To evaluate the effort needed to reuse the code for different applications (Q3) we run experiments where the robots have different objectives. This involves adapting the code to different analysis and planning algorithms, as well as to different actions.

We thus planned three experiments, each one targeting one of the above questions. The configuration of such experiments is reported in the following, and it is summarized in Table 1.

- **EXP1:** Robot R2 (*Pioneer P3DX*) is running the RoCS Framework, with the objective to reach a specific position and avoid collisions. At the same time, R1 (*Pioneer P3DX*) acts as a disturbing element, running a simple script that implements random movement. This experiment aims to answer Q1.
- **EXP2:** Robot R3 (*Robotnik*) is running the RoCS Framework, with the goal to reach a specific position while avoiding collisions. R1 acts as a disturbing element like in EXP1. The process of implementing EXP2 by trying to reuse code from EXP1 aims to answer Q2.
- **EXP3:** Robot R2 (*Pioneer P3DX*) is running the RoCS Framework, with the objective to reach a specific position and avoid collisions. Robot R3 (*Robotnik*) is running the RoCS Framework, with the goal to follow R2 and avoid collisions. As in the other experiments, R1 is acting as a disturbing element. The process of implementing EXP3 by trying to reuse code from EXP1 and EXP2 aims to answer Q3.

The execution of the experiments and their results are discussed in the following sections. Only EXP1 was already present in the conference version of the paper (Ramos et al., 2019b).

Table 1. Summary of the configurations of the three experiments.

Robot	**R1**		**R2**		**R3**	
Hardware	Pioneer P3DX		Pioneer P3DX		Robotnik SUMMIT-XL	
Software	*Code*	*Behavior*	*Code*	*Behavior*	*Code*	*Behavior*
EXP1	Script	Random	RoCS	Go to position Avoid collisions	—	
EXP2	Script	Random	—		RoCS	Go to position Avoid collisions
EXP3	Script	Random	RoCS	Go to position Avoid collisions	RoCS	Follow R2 Avoid collisions

Table 2. Actions included in the experiment and their decomposition.

Action	**Decomposition**
`SetWheelSpeed`	*atomic*
`TurnAngle`	*atomic*
`AvoidCollision`	`TurnAngle`; `SetWheeelSpeed`
`GoToOrigin`	`TurnToPosition`; `SetWheeelSpeed`
`TurnToOrigin`	`TurnAngle`

6.3 EXP1

R2 (Pioneer P3DX) is running the RoCS Framework, with the objective to reach a specific position and avoid collisions. At the same time, R1 (Pioneer P3DX) acts as a disturbing element, running a simple script that implements random movement.

To implement the behavior of R2, we consider two *atomic actions* and three *macro-actions* (see Table 2).

Figure 5a presents the simulated environment. In this scene, the robot starts at the top of the environment, close to the walls and to the box, and it should reach the origin of the system, defined as a point with coordinates $(0, 0)$. Figure 5b depicts the resulting trajectory performed by the robot and the macro-actions enqueued in the `Pipeline`. Each small dot represents an action, while the squares indicate a change in the executing paradigm (deliberative or reactive). Since the pipeline is cleared once the reactive action route is active, the robot has to re-plan based on the information on its new position.

To accomplish the task described in Section 6, several extension points (see Section 5.3) were implemented. The classes constituting the implementation of the experiment are depicted the diagram of Figure 6, and described in the following.

Each sensor in the Pioneer P3DX robot model was implemented through the extension of either `Sensor` or one of its heirs, like `Range`. The same applies to `WheelVREP`, which implements the motion of robot wheels and it is the concrete implementation of an `Actuator` in our scenario. Three `Monitors` were created, one for watching each type of `Sensor`, and three corresponding `Analyzers`: `PassOrientation`, `PassPosition` and `PassRange`. The `GoToOriginPlanner` is responsible for defining which actions should be sent to the queue, whereas `AvoidObstacle`, `GoToOrigin`, `SetWheelSpeed`, `TurnAngle` and `TurnToOrigin` represent the actual actions that will be performed and transformed into actuators commands. Finally, `AvoidObstacleModel` and `PioneerP3DXModel` are the extensions of the `ReactiveModel` and of the `RobotModel`, respectively.

It is important to remember that one of the requirements of the framework is to facilitate the transfer from the simulated robot to the real one. In this case, we would only have to change the concrete `Sensor` and `Actuator` classes, which facilitates enormously such a task.

6.4 EXP2

R3 (Robotnik) is running the RoCS Framework, with the goal to reach a specific position while avoiding collisions. At the same time, R1 (Pioneer P3DX) acts as a disturbing element, running a simple script that implements random movement.

In the second experiment, we aim at showing the effort required when changing the robot model without changing the application task. For this purpose, we replaced the RoCS *Pioneer P3DX* robot of EXP1 (R2) with the *Robotnik* robot (R3). As discussed earlier, the two robots differ mainly in the number and kind of adopted sensors and actuators.

The goal of the robot is the same, so no modifications are required to the planning algorithms. Few elements of the implementation required additions or modifications, as described in the following.

Because the robot contains an additional kind of sensor (a special sensor to detect another robot), a new class that extends `Sensor` was required. The other `Sensors` were just instantiated with different values for their properties, but it was not needed to implement a new class. The same happened for the `Actuator` classes, that is, the same classes used in EXP1 were instantiated with different parameters. An additional `Monitor` was incorporated to deal with the new `Sensor`, as well as a corresponding `Analyzer`, which is implemented by the class `P3dxPositonAnalyze`.

The class `GoToOriginPlanner` (specialization of `Planner`), and the actions `AvoidObstacle` and `GoToOrigin` have been reused as they are, because they employ lower level actions such as `SetWheelSpeed`, `TurnAngle` and `TurnToOrigin` that represent the actual actions that will be transformed into actuators commands. These lower level actions were modified, because the commands to drive a 4-wheeled robot are differenet from those to drive the 2-wheeled robot used in EXP1.

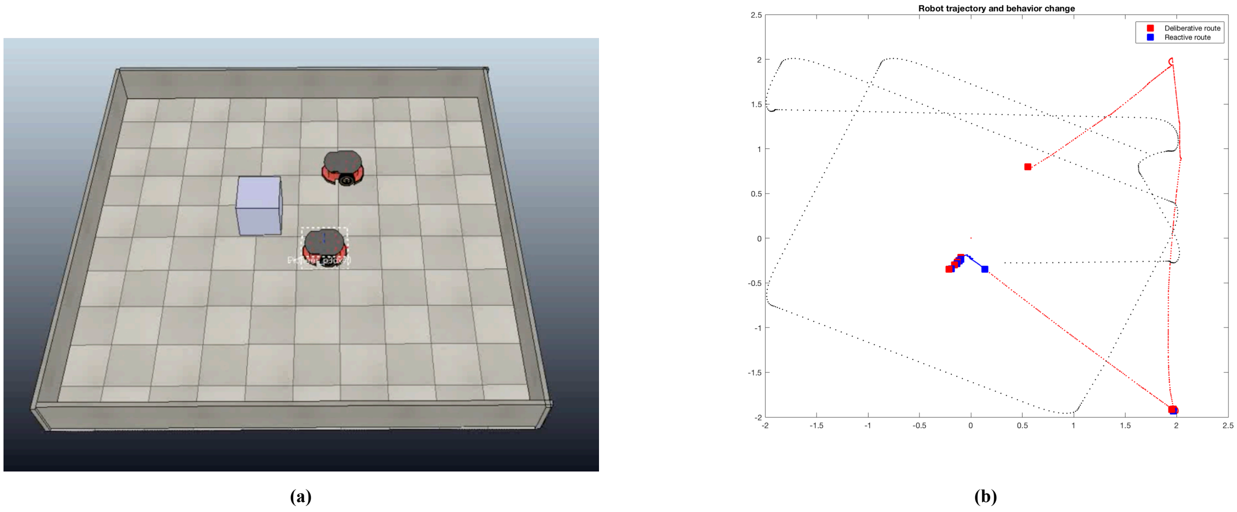


Figure 5. Simulated environment for EXP1 (a) with the two robots R1 and R2 deployed. The robot at the bottom is R1, the disturbing element that is not running the framework. The plot (b) depicts the resulting trajectory and the corresponding active paradigm for the robot that implements the framework. The black dots represent the trajectory for the R1 (disturbing robot). The red/blue trajectory is the R2 trajectory. A video with the resulting simulation can be found at `https://youtu.be/fbZCgmZIjqg`.

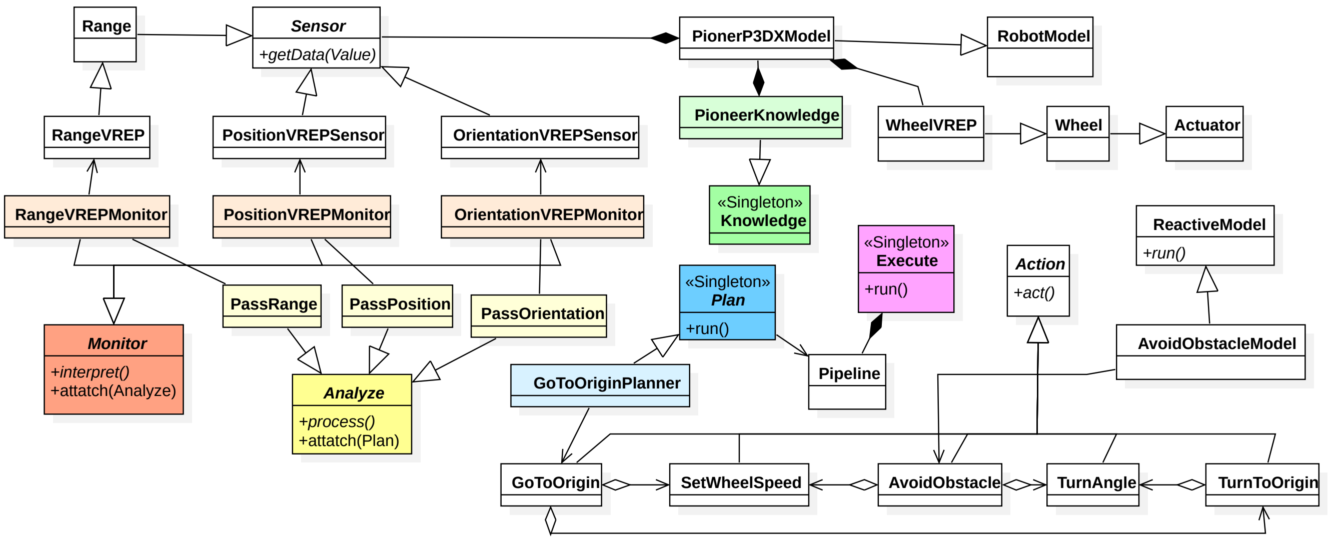


Figure 6. Implementation using the RoCS framework for the Pioneer P3DX. Associations that can be inferred from Figure 3 have been omitted for simplicity.

Finally, `AvoidObstacleModel` and `RobotnikModel` are the extensions of the `ReactiveModel` and of the `RobotModel`, respectively. These models were adapted to support the new configuration of the robot model.

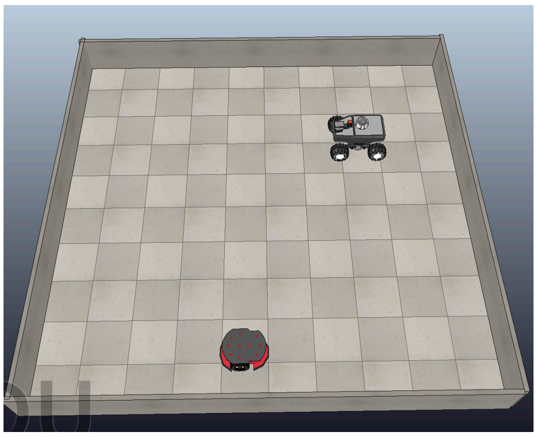
Figure 7 depicts the scene for running EXP2 with the two robots R1 and R3 deployed. The robot at the bottom is R1, the disturbing element that is not running the framework. The plot (b) depicts the resulting trajectory and the corresponding active paradigm for the robot that implements the framework. The red/blue trajectory is the R3 trajectory. It is important to mention that once the R3 robot reaches its goal position (around x(1.2) and y(1.2)), it tries to remain there, only reacting when the disturbing robot R1 gets too close.

6.5 EXP3

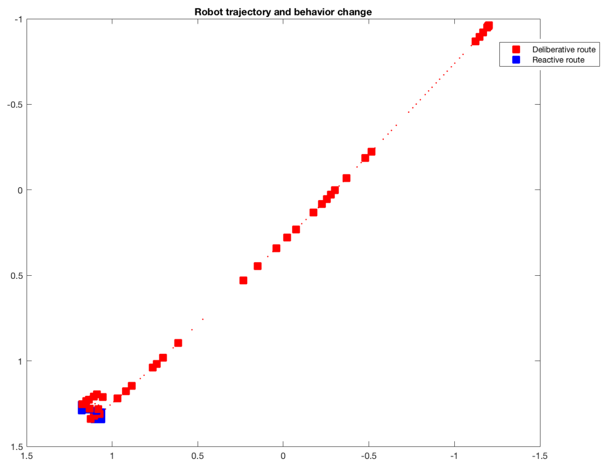
R2 (Pioneer P3DX) is running the RoCS Framework, with the objective to reach a specific position and avoid collisions. Robot R3 (Robotnik) is running the RoCS Framework, with the goal to follow R2 and avoid collisions. At the same time, R1 (Pioneer P3DX) acts as a disturbing element, running a simple script that implements random movement.

The third experiment aims at showing the impact of changing the robot task (possibly while maintaining its physical model), when using RoCS. In this last experiment, R2 maintains the same objective as in EXP1, while R3 (*Robotnik*) is assigned a different task with respect to the previous experiment. In this experiment it should follow the RoCS *Pioneer P3DX* robot (R2), maintaining a safe distance, and at the same time avoiding obstacles.

Compared to EXP2, the only change required in the Robotnik robot implementation was related to the `GoToOriginPlanner`. Now, instead of directing the robot to a fixed position, it uses the `P3dxPositonAnalyze` analyzer to compute how far and how misaligned it is from the robot that it should follow. Everything else in the implementation deployed on R3 remains the same as in EXP2, although the behavior of the robot has changed drastically.



(a)



(b)

Figure 7. Simulated environment for EXP2 (a) with the two robots R1 and R3 deployed. The robot at the bottom is R1, the disturbing element that is not running the framework. The plot (b) depicts the resulting trajectory and the corresponding active paradigm for the robot that implements the framework. The red/blue trajectory is the R3 trajectory. A video with the resulting simulation can be found at `https://youtu.be/2tkBhFGNOJo`.

Figure 8 and Figure 9 depict the scene where the experiment was evaluated, and the trajectory followed by the robots, respectively.

Figure 9a presents the trajectories (x, y) performed by R3 (*Robotnik*) and R2 (*Pioneer P3DX*) with the RoCS implementation. The goal of R2 is to go to a specific plane coordinate in the scene while avoiding collisions. We have specified the four corners of the scene as targets, and the goal changes once the robot reaches a target, requiring it to re-plan. In EXP1 and EXP2, only one point in the scene was defined as the goal. R3, on the other hand, has the objective to follow R2, at a safe distance. However, as the figure shows, sometimes it has to change its course to avoid colliding with either wall, the disturbing element R1, and the R2 itself (e.g., when the latter stops to change direction). Such reactions to unexpected obstacles are handled by the reactive route. Figure 9b and Figure 9c plot the trajectories of the two robots along the x and y axis, respectively, with respect to time. From the plot, it can be observed that the *Robotnik* (R3) is actually following R2 during the whole experiment.

Finally, Figure 9d and Figure 9e show which behavior is activated in R2 and R3, respectively. As in the corresponding figure of EXP1 and EXP2, red points indicate that the deliberative behavior is active, while blue points indicate that the reactive route is active, with squares indicating a change of behavior. It is possible to notice that in this experiment, as the scene is more complex than the previous experiments, the robot changes behavior much more frequently.

6.6 Summary

The experiments conducted demonstrated that by using the RoCS framework it was possible to, with minimum adaptation: a) change from a robot architecture to another one without changing the purpose of the robot (task to be performed) and, b) change the goal of the robot while maintaining its physical architecture. We also demonstrated the application of the RoCS framework in increasingly complex scenarios.

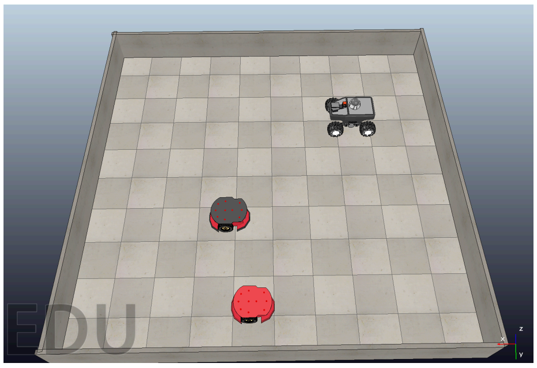


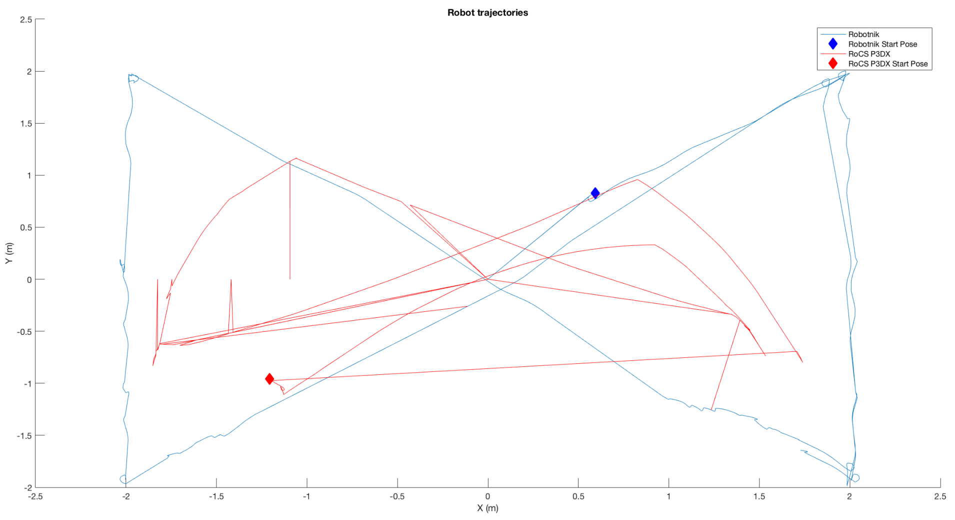
Figure 8. V-REP scene for EXP3, showing the three deployed robots: R1 (red robot), R2 (red/black robot), and R3 (four-wheeled robot). A video with a simulation of this experiment can be found at `https://youtu.be/qIeYfc5b1Yk`.

7 Relationship with ROS

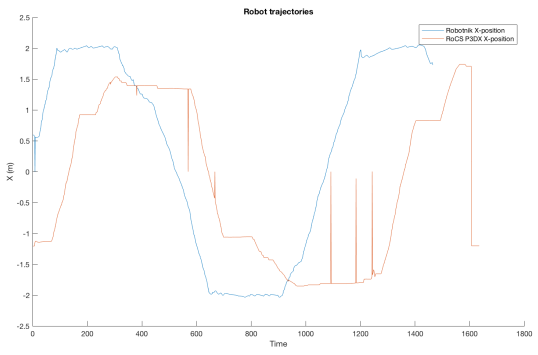
Since 2007, the numerous contributions to ROS led to a vast set of libraries and tools that can be very helpful when developing robot applications. However, this middleware is mostly focused on defining a message interface and conventions that abstract from the hardware details. ROS also contains several task-specific modules (ROS packages), provided by its contributors. RoCS, on the other hand, extends the MAPE-K reference architecture to the context of robotics. Therefore, RoCS can be used to appropriately structure an autonomous robots system, while ROS offers a set of modules (mostly drivers) that can also be exploited.

ROS provides an abstraction layer, conventions and several modules that can easily be integrated into autonomous robotics systems. However, it does not provide a general pipeline for complex robotic systems. On the other hand, RoCS focuses on structuring the software architecture and can thus supply to this limitation, while being compatible with ROS modules.

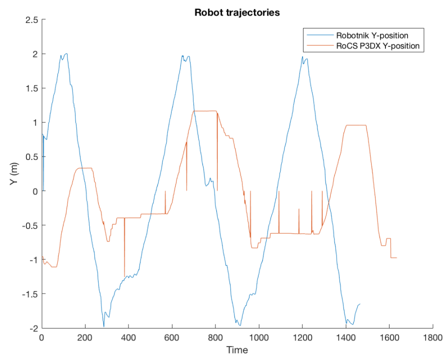
To better understand how the integration of ROS and RoCS can be performed, some of the main ROS concepts are detailed below.



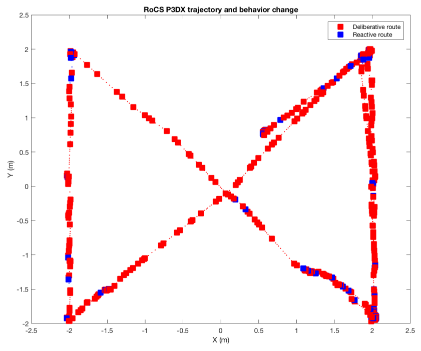
(a) (x, y) trajectories performed by R2 (*Pioneer P3DX*) and R3 (*Robotnik*).



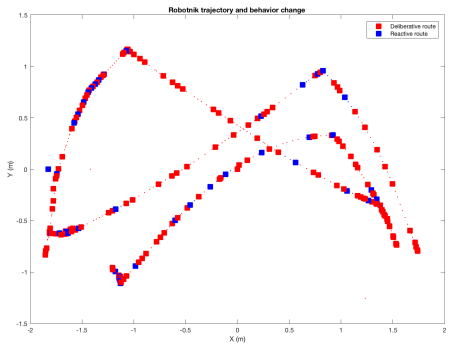
(b) R2 and R3 – x trajectories.



(c) R2 and R3 – y trajectories.



(d) R2 – Activation of deliberative and reactive routes.



(e) R3 – Activation of deliberative and reactive routes.

Figure 9. EXP3 Results. a) Trajectories (x,y) performed by P3DX with RoCS and Robotnik. (b) X-trajectory performed by both robots. (c) Y-trajectory performed by both robots. (d) P3DX with RoCS behavior execution showing deliberative and reactive routes and behavior switch (squares indicate a change of behavior) (e) Robotnik with RoCS behavior execution showing deliberative and reactive routes and behavior switch (squares indicate a change of behavior).

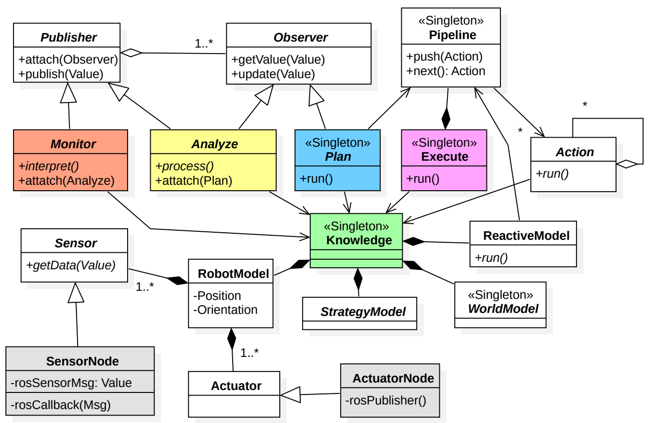


Figure 10. Class Diagram of the RoCS Framework integrated with a ROS environment. The nodes instantiate `SensorNode` and `ActuatorNode` objects, allowing RoCS to receive data from subscribed topics and to publish messages, respectively.

Package. ROS packages are sets of related files, which may include the execution files (nodes or components), and other elements of a program. The organization in package aims to organize related programs into one location.

Message. This is how the middleware performs the communication of different types of data between nodes. Messages are defined by means of files having a *.msg* extension, which consists of simple data structures. A message may contain exclusively a particular primitive type such as an integer or a floating-point, but it can also be defined as a set of primitive types.

Topic. Messages are organized into topics having a name. Therefore, communication happens in a publisher-subscriber fashion: nodes that transmit information must post messages on a given topic, while a node that needs to receive such information should subscribe to the topic in question.

The RoCS Framework can be extended to work on top of the ROS middleware. This is accomplished by accepting ROS messages as inputs and outputs to the autonomous system. In a certain sense, we consider the ROS middleware as part of the world in which the robot must sense and actuate. This solution is depicted in Figure 10 through the addition of the `SensorNode` and `ActuatorNode` classes, which extend the `Sensor` and `Actuator` classes of the RoCS Framework.

In practice, the `SensorNode` class instantiates a general `ROS::Subscriber` class, i.e., an object of type `SensorNode` starts a callback operation. In this context, a callback corresponds to a function that "listens" to a ROS topic and that executes whenever a ROS message is published through it. The received message is accessed inside the callback, which simply updates a class attribute (rosSensorMsg in Figure 10). Therefore, any ROS message can be accessed through the `getData` operation.

To send commands to actuators through ROS, it is necessary to instantiate a slightly different implementation of the `Actuator` class, defined as `ActuatorNode`. This class uses a `ROS::Publisher` object to send messages to any ROS node subscribing to a specific topic where this message is published.

In this way, any external information (i.e., message) that comes from ROS can be treated as sensory information from the RoCS perspective, while commands to actuators that are driven by ROS can be set by publishing data to their respective topics.

8 Final Remarks

We presented an instantiation of the MAPE-K reference architecture towards the Robotics perspective, in particular, focusing on service robot applications with heterogeneous physical platforms. Based on this, we developed the RoCS Framework to support the development of autonomous robots following a precise architecture.

We believe that this approach can assist students or novices in robot development, and help experienced developers focus on their specific problems like machine learning algorithms for computer vision, sensor fusion techniques, and locomotion for robots using particular physical devices (e.g., wheels, legs, propulsion). Actually, various students are using RoCS for different problems in our laboratory, including some without prior knowledge of the proposed framework.

Initially Ramos et al. (2019b), the RoCS Framework was evaluated in a simple but usual scenario for service robots as proof of concept. In this work, we performed two additional experiments exploring the reuse capability of the framework by changing robot models and their application tasks. For future work, we envision the development of physical robots using the RoCS Framework for competitions such as the *RoboCup Humanoid Soccer Teen Size League* and the *RoboCup Flying Robots Competition*. A deeper experimental evaluation of the benefits of using the framework is also part of our future work.

9 Declarations

9.1 Availability of data and material

The RoCS Framework implementation and all the configuration to run the experiments are available in the GitHub repository at `https://github.com/larocs/RoCS`.

9.2 Competing interests

The authors declare that they have no competing interests.

9.3 Funding

This research was partially supported by FAEPEX (Fundo de Apoio ao Ensino, à Pesquisa e à Extensão) of the UNICAMP and PIBIC/CNPq.

9.4 Authors' contributions

LR developed the source code of the RoCS framework and the instances for the experiments. GLGD instantiated the MAPE-K architecture and designed the first version of the RoCS framework. BBNF and LM reviewed the design of the

RoCS architecture and implementation, improved the comparison with related works, as well as defined the experiments. ELC provided the conceptual basis on Robotics Systems, defined the requirements for the RoCS framework, and supported the analysis of the results. GCL analyzed the RoCS framework and its instantiation to compare it with the ROS middleware.

9.5 Acknowledgements

The authors would like to thank the CIbSE 2019 reviewers for their positive contributions to the improvement of this work.

References

Albus, J. S., Lumia, R., Fialaa, J., and Wavering, A. (1989). NASREM: The NASA/NBS Standard Reference Model for Telerobot Control System Architecture. In *Proceedings of 20th Int. Symposium on Industrial Robots*, pages 1412–1419.

Arkin, R. C. (1998). *Behavior-Based Robotics*. MIT Press.

B-Human (2018). B-human team homepage.

Bayouth, M., Nourbakhsh, I. R., and Thorpe, C. E. (1998). A hybrid human-computer autonomous vehicle architecture. In *Third ECPD International Conference on Advanced Robotics, Intelligent Automation and Control*.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

Chan, Y. J. and Yow, K. C. (2006). A strategy-driven framework for multi-robot cooperation system. In *Control, Automation, Robotics and Vision, 2006. ICARCV'06. 9th International Conference on*, pages 1–6. IEEE.

Choulsoo, J. et al. (2010). OPRoS: A New Component-Based Robot Software Platform. *ETRI Journal*, 32(5):646–656.

Collett, T. H. J. and Macdonald, B. A. (2005). Player 2.0: Toward a practical robot programming framework. In *in Proc. of the Australasian Conference on Robotics and Automation (ACRA)*.

De La Iglesia, D. G. and Weyns, D. (2015). MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 10(3):15:1–15:31.

De Silva, L. and Ekanayake, H. (2008). Behavior-based robotics and the reactive paradigm a survey. In *2008 11th International Conference on Computer and Information Technology*, pages 36–43.

IBM (2005). An architectural blueprint for autonomic computing. Technical report, IBM.

IFR (2018). International federation of robotics.

Jeong, I. B. and Kim, J. H. (2008). Multi-layered architecture of middleware for ubiquitous robot. In *Systems, Man and Cybernetics 2008*, pages 3479–3484.

Kim, D. et al. (2006). SHAGE: A Framework for Self-managed Robot Software. In *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems (SEAMS'06)*, pages 79–85.

Klös, V., Göthel, T., and Glesner, S. (2015). Adaptive knowledge bases in self-adaptive system design. In *41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2015)*, pages 472–478, Funchal, Portugal.

Magyar, G., Sinčák, P., and Krizsán, Z. (2015). Comparison study of robotic middleware for robotic applications. In *Emergent Trends in Robotics and Intelligent Systems*, pages 121–128. Springer.

Makarenko, A., Brooks, A., and Kaupp, T. (2007). On the benefits of making robotic software frameworks thin. In *IROS Proceedings*.

Malek, S. et al. (2010). An Architecture-driven Software Mobility Framework. *Journal of Systems and Software*, 83(6):972–989.

Mathworks (2018). What is inverse kinematics.

Object Management Group (2016). Hardware Abstraction Layer for Robotic Technology (HAL4RT). Version 1.0 – Beta 1, dtc/2016-01-01.

Object Management Group (2018). Robotic Interaction Service Framework (RoIS). Version 1.2, formal/2018-05-04.

Qasim, A. and Kazmi, S. A. R. (2016). MAPE-K Interfaces for Formal Modeling of Real-Time Self-Adaptive Multi-Agent Systems. *IEEE Access*, 4:4946–4958.

Quigley, M. et al. (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

Ramos, L., Divino, G., de França, B. B. N., Montecchi, L., and Colombini, E. (2019a). RoCS GitHub Repository. `https://github.com/larocs/RoCS`.

Ramos, L., Divino, G., de França, B. B. N., Montecchi, L., and Colombini, E. (2019b). The RoCS Framework to Support the Development of Autonomous Robots. In *XXII Ibero-American Conference on Software Engineering (CIBSE 2019)*, La Habana, Cuba.

Ranganathan, A. and Koenig, S. (2003). A reactive robot architecture with planning on demand. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, pages 1462–1468.

Rauch, C. et al. (2012). A concept of a reliable three-layer behaviour control system for cooperative autonomous robots. In *35th German Conference on Artificial Intelligence*, pages 24–27, Germany.

RoboCup (2018). The robocup federation.

Rohmer, E., Singh, S. P. N., and Freese, M. (2013). V-REP: a Versatile and Scalable Robot Simulation Framework. In *IROS Proceedings*.

Simmons, R. and Mitchell, T. (1989). A task control architecture for autonomous robots. In *Proc. Third Annual Workshop on Space Oper. Auto. and Robotics*.

Weyns, D., Malek, S., and Andersson, J. (2010). FORMS: A Formal Reference Model for Self-adaptation. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*, pages 205–214. ACM.