The RoCS Framework to Support the Development of Autonomous Robots

Leonardo Ramos, Gabriel Divino, Breno Bernard Nicolau de França, Leonardo Montecchi, and Esther Colombini

Universidade Estadual de Campinas
Campinas, SP, Brazil
leo.o.rms@gmail.com, gabriel.lg.divino@gmail.com,
{breno,leonardo,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. Finally, we understand the proposed framework needs experimental evaluation as well as 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 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 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 [5, 10, 11, 13–16, 18, 21, 23]. 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 are 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 [1], known as MAPE-K, under a robotics perspective.

The remaining sections are 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 to the MAPE-K reference architecture. Section 5 presents the RoCS Framework for autonomous robots. Section 6 presents the framework evaluation for an usual robotics scenario. Finally, Section 7 presents the final remarks and 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 [3]. 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 securely.

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

To foster advancements in service robots, the RoboCup Federation [4] has proposed a set of challenges for evaluating the success of developments in a variety of domains. From playing soccer to assisting in typical tasks at home, these autonomous robots need to coordinate a variety of elements to succeed.

Although RoboCup standardizes the tasks that will be addressed, how they will be evaluated, and, in few cases, which robot platforms are allowed, the job of defining the software and hardware components of the robots is left open. A quick look at different domains [4] 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 work 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 [2] framework. However, 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 reuse 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. The approaches proposed in the literature are usually divided into three paradigms: **deliberative** [5], **reactive** [8, 20], and **hybrid** [7, 9].

In the *deliberative paradigm*, the robot uses the available sensory information and its knowledge of the world to reason about 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 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, 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 the organization of components.

3 Related Work

Several works suggest a structured approach to the control of robots.

The Robot Operating System (ROS) [18] 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, thus being compatible with multiple simulators and robot models. However, it provides the basic software components of a robot without necessarily prescribing an architecture, also limited to Unix-compatible platforms. Other framework that adopt a similar approach are frameworks such as Player, [11], ORCA [15] and OPRos [10].

The Task Control Architecture (TCA) [23] consists of *task*-specific modules connected to a central *control* module. The task modules perform all the required processing and communicate with the control via messages. The control routes these messages to their destination and 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 recovery. Although the TCA facilitates the modular and incremental design of complex robot systems, the centralized control can easily become the bottleneck.

To dilute the centralization problem, *layered architectures* were proposed [13,21]. Three to four layers are typically used, depending on the implementation. In practice, layers are organized differently, based on the kind of robot and kind of task, which leads to large variety in architectures and prevents reuse.

The *4D/RCS* reference architecture provides a theoretical foundation for engineering software for unmanned vehicle systems [5]. The architecture consists of a multi-layered hierarchy of computational nodes, each having the capability of world observing, self-orientation, decision-making, and autonomous action. This decision cycle is mostly known as the OODA-loop: observe, orient, decide and act. It is realized by five elements for each node: *sensory processing*, *world modeling*, *value judgment*, *behavior generation*, and *knowledge database*. The behavior generation module of a node is connected to those of the adjacent nodes, creating a command tree. Furthermore, each robot is partitioned into *subsystems*, that in turn are partitioned in *primitives* and then in *servo* computational nodes. This granularity, while feasible in specific scenarios, poses a problem when the robot's components are not unambiguously separable.

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 [1], 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 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 the environment, with minimal outside intervention. To do this, in MAPE-K, systems are composed by autonomic managers and the associated managed resources (Figure 1).

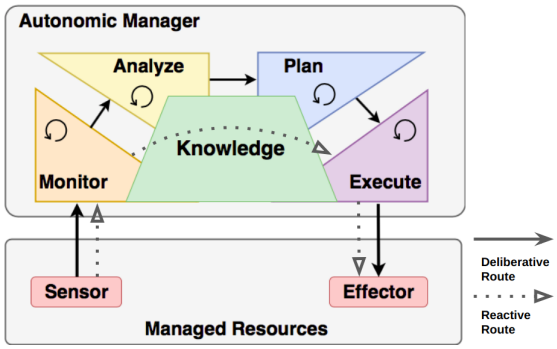


Fig. 1: Autonomic Manager in the MAPE-K Architecture (adapted from [1]).

According to the MAPE-K architecture [1], 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) touchpoints 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 [24] and ActivFORMS [12, 17] 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.

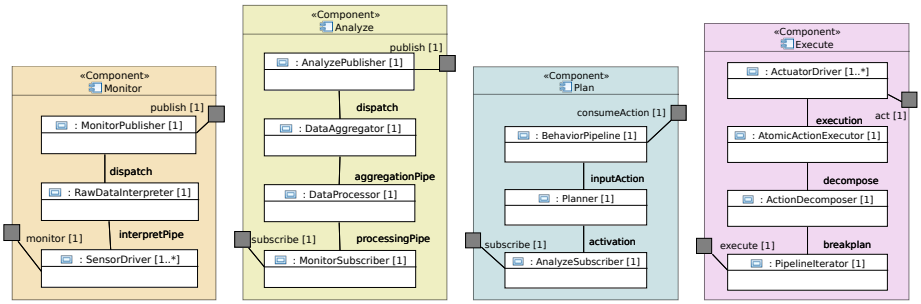
4.2 Detailed RoCS Architecture

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 2a) gathers and interprets raw data incoming from Sensors. One or more `SensorDriver` interact directly with physical sensing devices through the `monitor` port. The `RawDataInterpreter` translates raw data into final values, e.g., voltage 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 2b) 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.



(a) (b) (c) (d)

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

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. The `DataAggregator` module translates the results in application-level concepts, possibly fusing the results of multiple data processing algorithms. Finally, if a relevant change in the world is perceived, the `AnalyzePublisher` sends this information to the next block, which can elaborate a response accordingly.

The **Plan** block (Figure 2c) creates, selects, and forwards behaviors to be executed by the `Execute` block. Specific changes trigger the generation of new behavior. The realization of this block may range from simple Finite State Automatas to complex Cognitive Systems fine-tuned for specific tasks.

First, the block receives changes detected by the `Analyze` block via its `subscribe` port, which is delegated to the `AnalyzeSubscriber` module. Their 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. Lastly, this new behavior is sent to a pipeline (`BehaviorPipeline`) that will be read by the Execute block.

In the **Execute** block (Figure 2d), a planned behavior gets transformed into real robot actions like head movement or team communication. Once the next behavior has been decided, multiple actions might be needed to actually change the world state and realize the intended behavior.

The `Plan` block sends the desired behavior in the pipeline, which the `Execute` block has to iterate over. This is done by the `PipelineIterator` module, which is directly connected to the `execute` port. Whenever a behavior is received, it has to be decomposed in a sequence of atomic actions (`ActionDecomposer`). 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.

A **Knowledge** source is a grouping of data structures designed with the sole purpose of providing access to the world information to all the building blocks. Information in the `Knowledge` block is organized in 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 internal configuration, such as the parameters of its actuators, kinematics and inverse kinematics. The `StrategyModel` is a mapping of predefined strategies (if any) from which the robot can choose for executing its tasks and behaviors, e.g., the "defender" or "attacker" roles in case of a football competition. Finally, the `ReactiveModel` is responsible for storing the rules that trigger a reactive action by 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 new 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. 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 [6]. 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.

Such mechanism can be defined in our architecture, although a different abstraction is needed. There are two ways of implementing a reactive system within the 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 reaches 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 one of the many pre-defined reflexes once a specific change in the `Knowledge` Source occurs (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 [19]. The structure of the framework is shown in the Class Diagram in Figure 3 and 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 with their respective colors in Figure 3, as in Section 4.

While there may exist, in general, multiple independent instances of the Monitor and Analyze blocks, this is not the case for Plan, Execute, and Knowledge, and for this reason they apply the *Singleton* pattern. It is reasonable to constrain the framework user 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`).

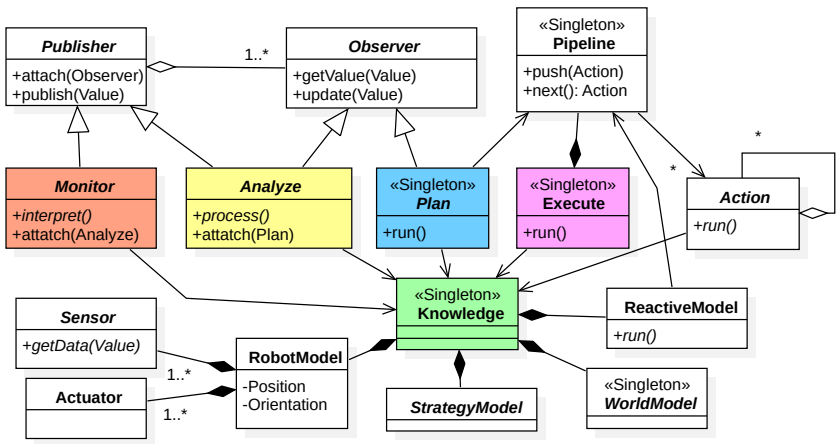


Fig. 3: Class Diagram of the RoCS Framework.

The Monitor, Analyze, and Plan classes communicate using a publish/subscribe 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 framework user 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 of a higher priority. 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 also widely dependent from 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 (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 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 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 return the value to be published to its observers. The implementation of this function is of course dependent of 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 a 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 is 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 consider four kinds of extension points: *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 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 accessed by the Plan. Whether these extensions are needed or not depends on the application.

6 Evaluation

To assess the application of the framework in a typical robotics task, we deployed a Pioneer P3DX robot in the V-REP [22] simulator running the framework. The robot has the following configuration. As **sensors**, it uses 16 range sensors (sonars) distributed around the robot circumference, 1 position and 1 orientation sensor that act as entities that are aware of the real position (x, y, z) and orientation (θ) of the robot. This special kind 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**, it uses two rotate motors responsible for driving the differential robot in the scene. In terms of actions, we consider two **atomic actions** and three **macro-actions** (see Table 1).

To evaluate the capability of the framework of supporting a hybrid approach, we defined a task where both deliberative and reactive routes can be activated. The robot task is *to reach a specific position in the environment (origin 0,0) while avoiding collisions*. It is important to mention that the robot is not aware of the

Table 1: Actions included in the experiment and their decomposition.

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

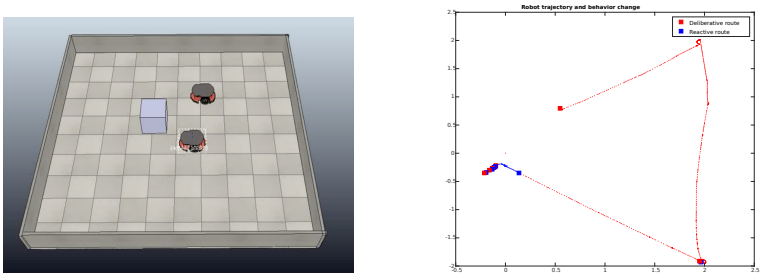
existence of obstacles while it computes its plan to reach the goal. Therefore, the "avoid the obstacle" behavior will be executed in a reactive fashion. At the same time, another robot with a simple 'explore the world' behavior (not implemented using the framework) is running in the environment to add more complexity to the scene.

Figure 4a presents the simulated environment. In this scene, the robot starts in the top of the environment, close to the walls and the box, and it should reach the origin of the system, defined as a point with coordinates (0,0). Figure 4 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). As 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.

6.1 Implementation of Extension Points

To accomplish the task described in Section 6, the following extension points (see Section 5.3) were implemented according to the Diagram presented in Figure 5.

Each sensor described earlier 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 is the concrete implementation of an `Actuator` in our scenario. Three `Monitor`s were created, one for watch-



(a) (b)

Fig. 4: Simulated environment for the experiment (a) with two robots deployed. The robot at the bottom is the secondary robot that is not running the framework. (b) the resulting trajectory with the corresponding active paradigm for the robot that implements the framework. A video with the resulting simulation can be found at: https://youtu.be/fbZCgmZIjqg

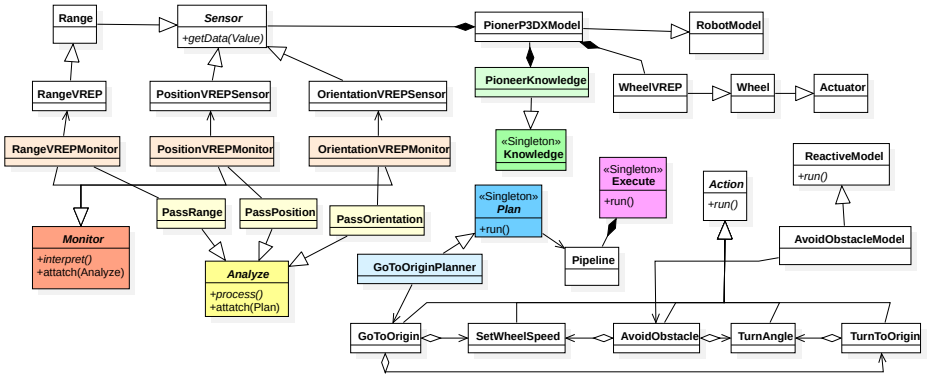


Fig. 5: Implementation using the RoCS framework for the proposed scenario. Associations that can be inferred from Figure 3 have been omitted for simplicity.

ing each type of `Sensor`, and three correspondly `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 requirement 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 work.

7 Final Remarks

We presented an instantiation to the MAPE-K reference architecture towards the Robotics perspective, mainly for service robots applications with heterogeneous physical platforms. From this, we developed the RoCS framework to support the development of autonomous robots under a known architecture. We understand 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 robot models using particular physical devices (e.g., wheels, legs, propulsion). Initially, the RoCS framework was evaluated in a simple but usual scenario for service robots as proof of concept. For future work, we envision the development of robots using the RoCS framework for competitions such as the RoboCup Humanoid Soccer Teen Size League and RoboCup Flying Robots Competition.

References

1. An architectural blueprint for autonomic computing. Tech. rep., IBM (Jun 2005)

2. B-human team homepage (2018), https://www.b-human.de/index.html
3. International federation of robotics (2018), https://ifr.org/
4. The robocup federation (2018), http://www.robocup.org
5. Albus, J.S., Lumia, R., Fialaa, J., Wavering, A.: NASREM: The NASA/NBS Standard Reference Model for Telerobot Control System Architecture. In: Proceedings of 20th Int. Symposium on Industrial Robots. pp. 1412–1419 (October 1989)
6. Arkin, R.C.: Behavior-Based Robotics. MIT Press (1998)
7. Bayouth, M., Nourbakhsh, I.R., Thorpe, C.E.: A hybrid human-computer autonomous vehicle architecture. In: Third ECPD International Conference on Advanced Robotics, Intelligent Automation and Control (1998)
8. Brooks, R.: Intelligence without representation. Artificial Intelligence **47**, 139–159 (1991)
9. Chan, Y.J., Yow, K.C.: A strategy-driven framework for multi-robot cooperation system. In: Control, Automation, Robotics and Vision, 2006. ICARCV'06. 9th International Conference on. pp. 1–6. IEEE (2006)
10. Choulsoo, J., et al.: OPRoS: A New Component-Based Robot Software Platform. ETRI Journal **32**(5), 646–656 (2010)
11. Collett, T.H.J., Macdonald, B.A.: Player 2.0: Toward a practical robot programming framework. In: in Proc. of the Australasian Conference on Robotics and Automation (ACRA) (2005)
12. De La Iglesia, D.G., Weyns, D.: 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 (2015)
13. Jeong, I.B., Kim, J.H.: Multi-layered architecture of middleware for ubiquitous robot. In: Systems, Man and Cybernetics 2008. pp. 3479–3484 (October 2008)
14. Kim, D., et al.: SHAGE: A Framework for Self-managed Robot Software. In: Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems (SEAMS'06). pp. 79–85 (2006)
15. Makarenko, A., Brooks, A., Kaupp, T.: On the benefits of making robotic software frameworks thin. In: IROS Proceedings (2007)
16. Malek, S., et al.: An Architecture-driven Software Mobility Framework. Journal of Systems and Software **83**(6), 972–989 (2010)
17. Qasim, A., Kazmi, S.A.R.: MAPE-K Interfaces for Formal Modeling of Real-Time Self-Adaptive Multi-Agent Systems. IEEE Access **4**, 4946–4958 (2016)
18. Quigley, M., et al.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software (2009)
19. Ramos, L., Divino, G., de França, B.B.N., Montecchi, L., Colombini, E.: RoCS GitHub Repository (2018), https://github.com/oramleo/RoCS
20. Ranganathan, A., Koenig, S.: A reactive robot architecture with planning on demand. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003). pp. 1462–1468 (2003)
21. Rauch, C., et al.: A concept of a reliable three-layer behaviour control system for cooperative autonomous robots. In: 35th German Conference on Artificial Intelligence. pp. 24–27. Germany (September 2012)
22. Rohmer, E., Singh, S.P.N., Freese, M.: V-REP: a Versatile and Scalable Robot Simulation Framework. In: IROS Proceedings (2013)
23. Simmons, R., Mitchell, T.: A task control architecture for autonomous robots. In: Proc. Third Annual Workshop on Space Oper. Auto. and Robotics (July 1989)
24. Weyns, D., Malek, S., Andersson, J.: FORMS: A Formal Reference Model for Self-adaptation. In: Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10). pp. 205–214. ACM (2010)