software

MDPI

Article

A Framework for Rapid Robotic Application Development for Citizen Developers

Konstantinos Panayiotou †, Emmanouil Tsardoulias †, Christoforos Zolotas †, Andreas L. Symeonidis *,† and Loukas Petrou †

School of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece; klpanagi@issel.ee.auth.gr (K.P.); etsardou@ece.auth.gr (E.T.); christopherzolotas@issel.ee.auth.gr (C.Z.); loukas@eng.auth.gr (L.P.)

* Correspondence: symeonid@ece.auth.gr

† These authors contributed equally to this work.

Abstract: It is common knowledge among computer scientists and software engineers that "building robotics systems is hard": it includes applied and specialized knowledge from various scientific fields, such as mechanical, electrical and computer engineering, computer science and physics, among others. To expedite the development of robots, a significant number of robotics-oriented middleware solutions and frameworks exist that provide high-level functionality for the implementation of the in-robot software stack, such as ready-to-use algorithms and sensor/actuator drivers. While the aforementioned focus is on the implementation of the core functionalities and control layer of robots, these specialized tools still require extensive training, while not providing the envisaged freedom in design choices. In this paper, we discuss most of the robotics software development methodologies and frameworks, analyze the way robotics applications are built and propose a new resource-oriented architecture towards the rapid development of robot-agnostic applications. The contribution of our work is a methodology and a model-based middleware that can be used to provide remote robot-agnostic interfaces. Such interfaces may support robotics application development from citizen developers by reducing hand-coding and technical knowledge requirements. This way, non-robotics experts will be able to integrate and use robotics in a wide range of application domains, such as healthcare, home assistance, home automation and cyber–physical systems in general.

Keywords: robotics; system modeling; cyber–physical systems; service-oriented architectures; robotic applications; citizen developers

Citation: Panayiotou, K.; Tsardoulias, E.; Zolotas, C.; Symeonidis, A.L.; Petrou, L. A Framework for Rapid Robotic Application Development for Citizen Developers. *Software* **2022**, *1*, 53–79. https://doi.org/10.3390/software1010004

Academic Editor: Manuel Mazzara

Received: 19 January 2022
Accepted: 25 February 2022
Published: 3 March 2022

1. Introduction

Should one pose a question on the amount of skills needed to create, control or use a robot, the answer is quite straightforward: many! Robotics can be seen as an amalgam of mechanical engineering, electrical engineering and computer science, joining forces towards the construction and control (manually or autonomously) of machines that interact with their environments. In this context, at least four skill sets are needed to create a robot from scratch.

The first concerns the definition of the physical properties of the robot and includes mechanical engineering knowledge on building the robot's body, i.e., knowing which materials are suitable for each application field and how to assemble the sensors and effectors, as well as how to integrate the motors into the body. The second skill set includes electrical and electronic engineering knowledge for hardware design and implementation. This includes knowledge of various sensors, effectors and microcontrollers types, specifications and operation modes, ways to interconnect hardware devices (e.g., microcontrollers) with a variety of communication protocols, such as TTL, UART, SPI, and I2C [1], and electrical power supply and consumption basics. In addition, one should have the ability to decide between the appropriate sensors and effectors that the robot must be equipped with according to the problem at hand. The third skill set involves low-level hardware programming

and Operating Systems (OS). Some examples include programming microcontrollers and embedded systems for sensor data acquisition, knowledge of communication protocols, developing drivers or OS kernel modules for supporting specific hardware devices and configuring the OS layer. Finally, the fourth skill set includes software and algorithmic programming capabilities. As far as algorithms are concerned, a large number of disciplines are involved [2], with localization and simultaneous localization and mapping (SLAM), vision-related detection and classification, control theory, path planning, full exploration and coverage of known or unknown environments, strategy and decision making being some of the most common. Software development capabilities involve knowledge of high-level programming languages, such as C++, Python, Java and server-side JavaScript (e.g., NodeJs).

Obviously, the roboticists do not have to master all of the aforementioned skill sets. The mechanical construction, as well as the interconnection of sensors and effectors, must be performed only in cases of custom robots, since there is a large variety of ready-to-deploy robots in the market (although their prices are still high). Buying a commercial robot also removes the burden of data acquisition and command passing to the various sensors and effectors via OS-specific drivers and kernel modules, as the manufacturing companies offer these abilities in a seamless way via robot-specific software libraries and frameworks. Finally, there exists a number of open source and commercial frameworks [3] that enable a roboticist to easily implement the core functionalities of a robot, as well as an abundance of algorithms offered by the robotics community that can be easily plugged in and operated efficiently.

Returning to the original question, it is a fact that robotics is a wide field, but it is also a field where a number or ready-to-deploy solutions exist, both proprietary and open-source. In this work, we argue that, apart from experienced developers, a new category of users are nowadays interested in building applications with affordable, open hardware/software robots. Citizen developers [4], i.e., users with experience in logic and/or computer programming, and with domain expertise, are, however, not familiar with the specifics of robotics per se. For this apparently wide category of users, a large obstacle between them and their first robot application lies within the development process itself. Even if the developer is aware of the various algorithmic types and knows when to use them, the process of configuring a new robot in order to implement an application is quite tedious: it involves selecting an OS supported by the robot, identifying the necessary algorithms and the components that implement them, compiling them or creating bindings between them and finally creating the software that will be OS, middleware and robot-specific. The latter means that porting an application from one robot to a new one will require quite a large effort, unless robot specifications are the same. Practically, this is one of the main reasons behind the minimal re-usability of robot applications.

In this context of "robotic applications by citizen developers", the current work proposes a robot-agnostic approach, among with a standardization of the robotic application development process, in order to simplify the configuration, reusability and control of robots. It is argued that this standardization will also enable the reusability of applications among heterogeneous robots, as well as solutions for executing heavy duty software on low-resource robots. The potential application developer can boost his/her productivity by using off-the-shelf robotics software components available on the cloud, or in-robot, without being an expert in the various robotics fields. Our approach utilizes model-driven engineering principles and technologies to model and automate aspects of the development process of robotic applications. The purpose of this study is to introduce an approach, based on models and model transformations, that enables access to the robotics application development domain for a wider range of end-users by hiding low-level technological and technical knowledge, such as firmware development, networking, operating systems and communication protocols. Furthermore, our approach promotes the remote control and monitoring of robots via auto-generated robot API gateways, which give remote access to the robots' resources via web and IoT communication protocols [5], such as REST, Websockets and MQTT.

The paper is structured as follows. Section 2 discusses the norm in the robotic application development, describing existing architectures and middleware, followed by how these tools are utilized in the overall procedure of creating a robotic application from scratch, presenting the inherent disadvantages and concluding with the novelty of the Robotic4All (R4A) approach. In Section 3, the proposed architecture requirements are defined, followed by a discussion on the abstraction of robotic resources towards the robot-agnostic production of applications. Additionally, the core components of the R4A architecture are described. Section 4 provides demonstrations of an indicative application creation for two different robots using the traditional method and the R4A approach, whereas Section 5 discusses findings and Section 6 probes on conclusions and future work.

2. The Beaten Path in Robotic Software Development

This section focuses on analysing the traditional way of creating robotic applications, and is split into three parts. The first briefly goes over the state-of-the-art concerning existing robotic middleware and architectures, since these are the core of almost every robotic development endeavor. The second part investigates the big picture, i.e., the steps a robotic developer must usually take, from procuring the robot to the distribution of the final application, where the aforementioned robotic middleware plays a crucial part. Next, the drawbacks and bottlenecks of this procedure are highlighted and the way the proposed R4A approach tackles them is described in the context of citizen development.

2.1. State of the Art Robotic Middleware and Frameworks

The state of practice dictates that robotic software development is performed using robotics middleware. Nevertheless, the first attempt to formalize robotic software development was introduced in the nineties, when several robotic architectures were proposed, which served the task of facilitating the development of robotic systems by imposing beneficial constraints on their design and implementation without being overly restrictive [6].

These first approaches followed the classical artificial intelligence paradigm, where the world representation should be as complete and valid as possible, regardless of the robotic actions or tasks, one example of which being the NASREM architecture [7]. These types of architectures can be described as hierarchical [8], whose successors were *behavior-based (BB)* and reactive architectures [9,10]. Of course, hybrid hierarchical/reactive architectures existed as well, and benefited from both approaches, such as AuRA [11].

As software architectures became more formal with the use of object-oriented programming (OOP) and software patterns, similar concepts were transferred to the robot development domain. The traditional architectures gave their place to modern robotics middleware, which, first of all, offered several ready-to-use tools and, secondly, promoted the software re-usability concept by referring to behaviors (or code chunks in general) as components [12,13]. Some examples of early modular middleworks are Nexus [14], GenoM [15] and PredN [16]. Following the reusable components concept, Nesnas et al. proposed the CLARAty architecture in [17,18], which was based on the hybrid reactive and deliberative systems, followed by the well-known robotic middleware called Player, along with its 2D and 3D simulators, Stage and Gazebo [19]. Of course, middleware with real-time capabilities existed, including Open Robot Control Software (OROCOS) [20], suitable for motion control, and ASEBA [21], targeted for microcontrollers incorporation.

Modern robotics frameworks and middleware follow abstract, component-based approaches. Some examples are MARIE [22], Modular Controller Architecture V2 (MCA2) [23] OpenRDK [24], OpenRTM [25] and SmartSoft, described in [26]. One of the major breakthroughs in the robotic middleware stage was the release of the Robot Operating System (ROS) in 2009 [27]. ROS is described as a meta-operating system and was designed to offer hardware abstraction and a standardized module-based development over a structured communication layer. Moving on to more modern concepts, Cognitive Robot Abstract Machine (CRAM) is a software toolbox for the design, implementation and deployment of cognition-enabled autonomous robots, oriented for everyday use [28], utilizing the KnowRob knowledge processing system [29]. If modularization is a concern, BRICS is

one of the most known meta-frameworks, providing a set of guidelines, meta-models and tools for structuring robotics software via components that were developed using the most known component-based middleware (OROCOS, OpenRTM, ROS, SmartSoft) [30], adopting the benefits of model-driven engineering approaches, as described in [31,32].

Lately, the appearance of the term "robotic applications" emerged, not meaning the conventional collection of code units, but an actual packaged software that can be downloaded and executed seamlessly in a robot, thus promoting the reusability of such code. The need for robotic applications comes as a result of the inherent complexity of the robotic software development procedure, as explained in the introduction. In [33,34], BABEL is proposed, which tackles the multi-domain skillset that robotics require. On the other hand, the RAPP project [35–37], developed a new robotic architecture where robotic applications could be created just by installing a robot-specific middleware in each robot and using a robot and cloud API.

Even though it is of great importance to select the proper middleware, it describes only a part of the equation needed to be solved by the developer, especially when he/she is assigned the whole process of setting up the robot, developing the application, testing it and deploying it. Next, an overview of the necessary steps for the developer to take will be presented, where, undoubtedly, an incorrect middleware selection generates ripples of problems in the overall procedure.

2.2. Robotic Applications: The Conventional Way

Robotic system engineering varies greatly. However, it remains an engineering task and, as such, it can be decomposed to distinct, closely related phases. One can identify at least five phases: (a) the setup phase, which includes the procurement and the setup/-configuration of the robot; (b) the design phase, where the architecture of the application is designed; (c) the development phase, where the application is developed, usually employing simulators to run component tests; (d) the testing phase, where the code is tested and evaluated on the real hardware; and, finally, (e) the distribution phase, where the application is packaged and distributed for use.

The robotics software development phase usually follows a hybrid process, where functionality is built in an iterative mode, but must be evaluated in a real world stochastic system comprising physical agents (robots) and unexpected situations (environment). It is literally unfeasible to develop software that is correct by design and will be correctly deployed in a robot, regardless of the environmental conditions that appear in the deployment area. As Figure 1 indicates, one of the strategic decisions that the development team must make concerns the testing phase. Some methodologies are proactive and enforce the development of functional software tests before the actual development (Case A), whereas others propose initial development to precede testing (Case C) and some prefer concurrent testing and development processes, such as the agile software development paradigm (Case B).

In this section, a description of the choices and alternative paths a robotics developer is faced with, for all five phases, is presented. The inner structure of these phases is analytically defined by the activity diagrams in Figures 2–4.

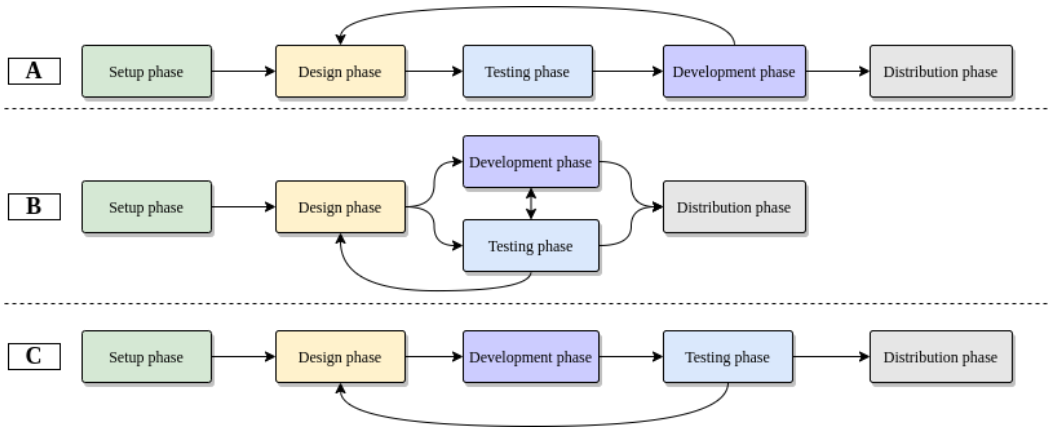

Figure 1. Common end-to-end development processes in robotics: Case A indicates a proactive method, where tests are performed before the actual implementation; Case B adopts the agile development model; Case C proposes initial development to precede testing.

2.2.1. Setup Phase

Naturally, the journey of the robotic applications developer begins with *procuring the robot*. The type of robot procurement varies depending on the final application, the path of research or the budget/available funding. The first step to carry out after acquiring the robot is to check if this robot is *equipped with a processing unit* and if it is *modifiable regarding the hardware parts*, i.e., if you can attach more sensors or effectors. Similarly, one has to check if in-robot software development and deployment is allowed. There exist cases where even the operating system is proprietary and locked to factory settings, and thus modifications and direct extensions are not allowed.

Concerning the HW part, if the robot does not contain a processing unit, the developer has to attach one and connect the existent sensors and effectors to it. Either way, if the robot allows for HW modifications, the roboticist must *choose the suitable sensors and effectors* for the application at hand, *attach* them to the robot body and *connect* them with the processing unit.

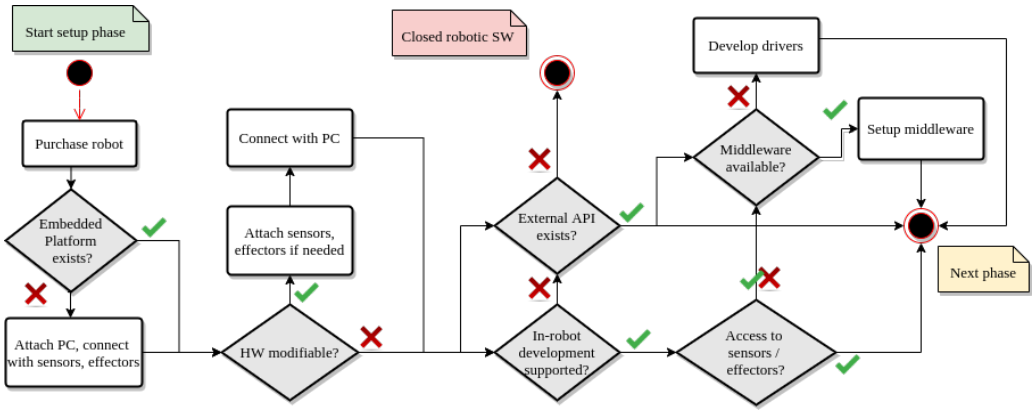

Figure 2. "Conventional" way of creating applications: setup phase.

Regarding software development, if no in-robot development is allowed (i.e., the robot is HW-closed), the developer must check if the robot manufacturer provides an *external API* for controlling the robot. If not, this means that the robot software is closed-source and that its operations are fixed, and thus no application programming is supported. In the case where in-robot or remote (via APIs) development is supported, the roboticist must research if the manufacturer *provides access to the sensors and effectors*, or if any *open-source or proprietary middleware packages exist* that would allow him/her to affect the robot behavior (e.g., ROS). If no such tool exists, then unfortunately the developer must build

the necessary *drivers* to access the robotic devices/parts before the development phase, whereas, if middleware packages exist, they must be *downloaded and set up* in the robot.

As clearly stated, the setup phase in robotics development requires a large amount of effort and multi-domain skills, such as low-level hardware knowledge for constructing the physical part of the robot (base, sensors, actuators, power distribution, etc.) and expertise in software engineering and development. The "conventional way" of the setup phase, regarding robotics application development, is evident in Figure 2.

2.2.2. Design Phase

The design phase assumes that the setup phase has been finished, and thus the respective tools (hardware and software) are in place. One important step that all developers must make before the actual development of their application is to design the individual robot software and the overall software system architecture according to the use cases, functional and non-functional requirements. This is an important process, since the designed architecture will allow for the developer to specify required algorithms, multi-threading or asynchronous executors support.

The next task that burdens the developer is the *programming language selection*. The language decision must be taken under several restrictions and assumptions. Some of the factors affecting the programming language selection are the OS, the selected middleware or the available drivers/APIs, the time criticality of the final application, performance and memory issues and threaded execution or not, as well as the availability of ready-to-use algorithm implementations. For example, if *performance/real time issues* exist, the selected language must be at a low(er) level in order to capitalize on the low-level calls to the OS. Furthermore, if the selected middleware (or the existing API) provides bindings to, e.g., Python, the developer must make a decision regarding *whether this language can be directly utilized or whether wrappers should be created*.

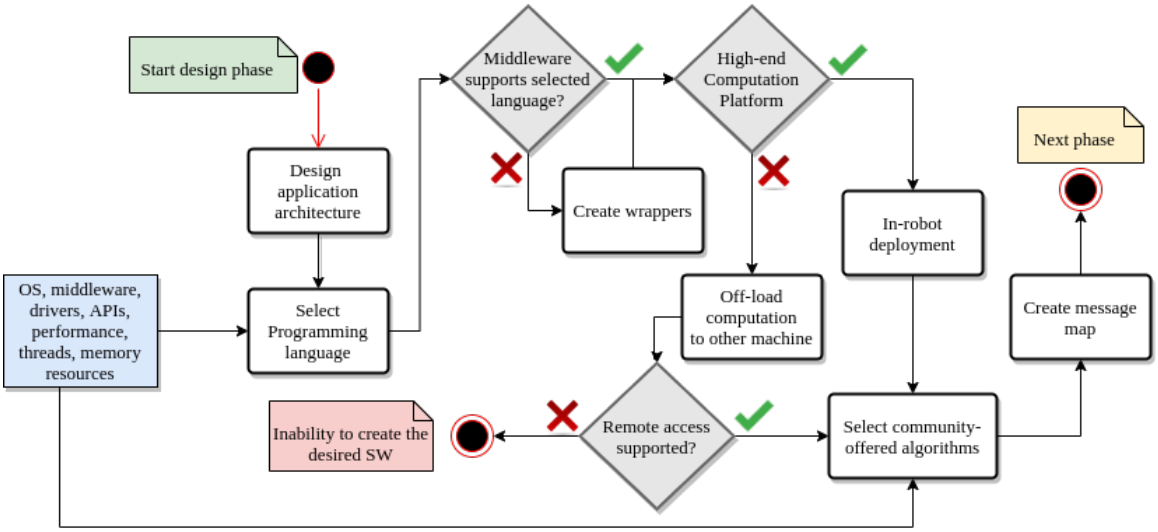

Figure 3. "Conventional" way of creating applications: design phase.

After (or during) the language selection process, the developer must pay attention to the selected middleware. If (a) the envisioned architecture requires multiple threads (or processes), (b) the robot CPU supports these requirements and (c) the middleware allows for distributed/modular application creation, then developers must first *decide on the deployment strategy*, i.e., *which operations will exist in which modules*. In the case where the CPU is low-end or the application requires more memory than available, the developer should look into the remote access capabilities of the robot, i.e., *if remote access is natively supported* via being provided by the manufacturer API or the selected middleware. If remote access is indeed supported, the developer may need to *offload some components that require high computational or memory resources and distribute them to one or more physical devices/nodes*.

Nevertheless, this process is quite demanding, as the roboticist should have knowledge of distributed application execution techniques and protocols between heterogeneous devices. Furthermore, communication and message passing between the main controller and the distributed components is almost always dictated by the selected middleware or is custom made, making the remote modules not reusable between robots.

The state of practice assumes that, since the robotics community offers a vast collection of robotic algorithms and modules, the developer should *take a look in some of the available repositories* in order to not "reinvent the wheel". After identification of the required modules (if any), *the connectivity to the rest of the developed code must be explored*. Finally, *a message map of the module collaboration must be created*. After this step, the developer proceeds with the actual application development.

2.2.3. Development Phase

Since robotics deal with multi-level uncertainty due to the software deployment in a real-life system, it is common to follow an iterative development process, i.e., to create a part of the software, test it, correct it and so on. Apart from bugs in software, the introduced uncertainty may be due to robot motion failures and sensory malfunctions or sensors' deviation from the nominal measures, a fact that enforces the employment of probabilistic theory in almost all algorithmic domains of robotics (SLAM, navigation, path planning, etc.)

After each development iteration, the roboticist must check and evaluate the functional behavior of each software component. When a simulator is available and the code can be deployed on it, the developer must install and setup this tool, perform tests and then reiterate the development procedure. Runtime tests must also be performed on the real robot and in the context of a real environment. If remote execution is supported by the underlying middleware, *the connectivity to the robot must be established* prior to the deployment of the application. If not, the relevant binaries must be *manually transferred* to the physical robot and deployed. If the software performance is satisfactory, the developer proceeds to the next phase, which is testing or distribution (according to the favorite development strategy).

2.2.4. Testing Phase

Initially, the developer must *choose appropriate testing tools* based on (a) the under testing functionality or component and (b) the selected programming language. *The selection of the proper testing tools must be made early*, especially if an agile software development methodology is applied and the testing and development processes run concurrently. Concerning the actual testing, the most common types of tests are unit, functional, integration and system tests. Unit tests *can usually be deployed without a real robot or even a simulator*, since they test specific parts of the software; nevertheless, one can deploy them in a simulator or on the real robot as well. As far as functional and integration tests are concerned, *the robot must be present*, either in its physical form by employing a simulator, or by creating mock endpoints for sensor/effector API calls. System tests are performed on a complete, integrated system in order to evaluate compliance with the initial requirements, and thus these must be deployed in the real robot under realistic environmental conditions. One common practice in software development is *continuous integration (CI)*, where a remote framework is set up to monitor code changes and execute all provided tests in order to offer a continuous status report on whether the software behaves as it should. If a CI is not present, the tests deployment must be manually performed.

2.2.5. Distribution Phase

When the developer is satisfied with the quality of the code, the created software matches the requirements of the setup phase and the produced code solves the problem at hand, the distribution phase follows, *provided the developer wants to share*.

Distribution methods highly depend on the runtime platform, both at the hardware and operating system layers, and the software license governing the use and redistribution

of the software. In the case of closed-source distribution, the following (mainstream) methods can be used:

- Make the pre-compiled binaries available to the web via a public link;
- Package and upload to OS-specific upstream repositories (e.g., Debian package);
- Package and upload to specialized repositories of robotics software, according to the selected middleware (for example, ROS upstream repositories).

On the other hand, open-source software can also be hosted by version control systems, such as Git, Subversion, etc. In all the above options, the developer must *create the necessary documentation and tutorials* that explain, in detail, how to download, setup, install and deploy the software. If the software is a generic algorithm that can be applied to a number of robots (e.g., a SLAM algorithm provided as a ROS package), it is helpful to create tutorials for different robots, aiming at assisting a beginner roboticist.

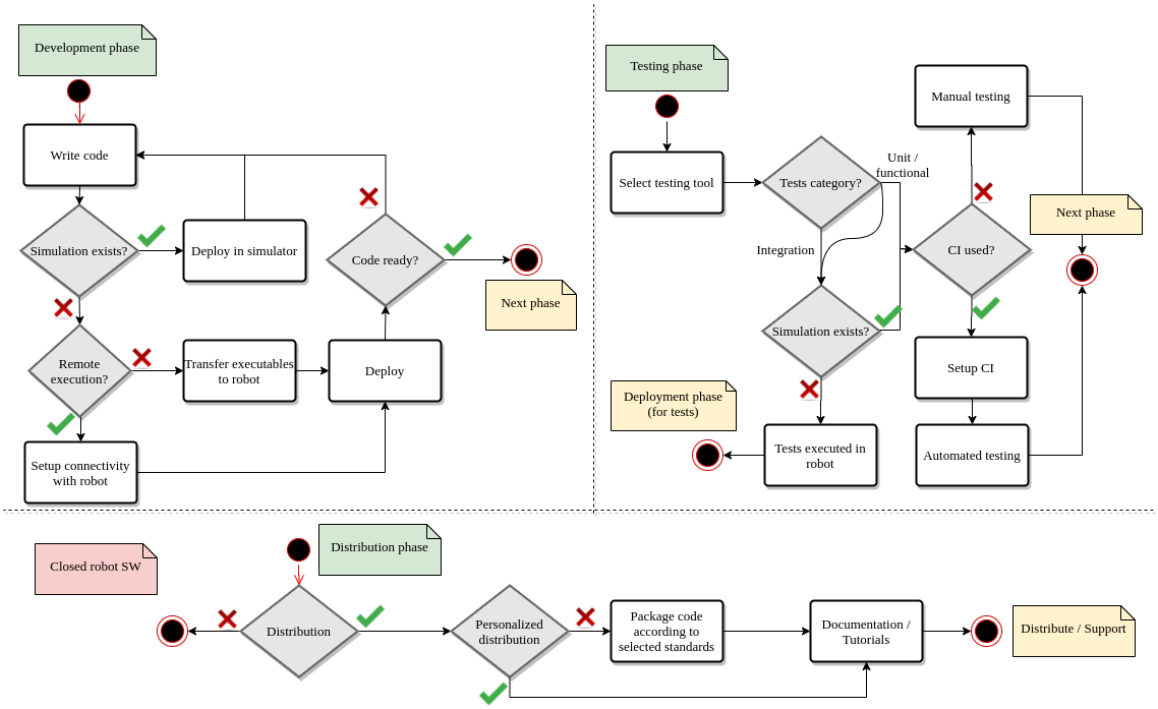

Figure 4. "Conventional" way of creating applications: deployment, testing and distribution phases.

2.3. Identifying the Limitations for Citizen-Developers

As evident from the above description, the development process includes a number of choices to be made, where most of which are only meaningful to an experienced developer. Wrong choices can lead to several issues, which may be critical enough to force the re-design and re-implementation of the whole system.

2.3.1. Resource-Related Inability to Meet the Requirements

In the first step (setup/design), it is important to be perfectly clear what the robot offers for control and data acquisition purposes, as well as the way it provides it. In case the developer has not performed thorough market research, the selected robot may lack functionalities or resources crucial to the final application.

2.3.2. Middleware-Related Inability to Meet the Requirements

During the software development phase, the developer must choose one of the decades of available robotics middleware and possess the required knowledge of the robot's OS in order to correctly perform the configuration. As understandable, problems may arise if the selected middleware does not inherently offer requirements-related functionality (e.g., ROS employment for controlling a real-time system is not a suitable selection).

2.3.3. Portability Problems to Other Robots

Even if the previous choices were wisely made, it is quite certain that the produced software will not be able to be deployed on another robot unless the developer has taken into account cross-robot execution support.

2.3.4. Complexity Due to Distributed Application Deployment

If the robotic embedded hardware platform is not powerful enough (resource-wise), it is quite hard to manually develop the infrastructure and to support a distributed or even remote execution of the code.

2.3.5. Physical Damage Due to Insufficient Testing

During the testing and deployment phases, if the developer does not follow the standard testing procedures and these are performed on the physical robot platform and in real environments, it can even lead to physical damage of the robot (e.g., during the development of an obstacle avoidance algorithm).

2.3.6. Erroneous 3rd Party Setup/Execution

Finally, the distribution phase is quite demanding, since a lot of time must be spent writing error-free instructions for the code retrieval, installation, configuration and deployment.

2.4. Leveraging the Robotic Software Development Problems

This work directly aims towards establishing a rapid application development methodology in robotics for end-users to be able to integrate robotics in their development process and products. Currently, in robotics, in order to reach the application development phase, excessive effort and multi-domain knowledge are required in order to develop the control and monitoring software stack of a robot. By performing high-level abstractions, the undesired effect of restricting experienced developers arises. For this reason, R4A supports the seamless integration of ROS components (since ROS has become the standard as far as robotics middleware is concerned). This way, anyone can include low-level implementations in robot-agnostic applications, or can even utilize algorithms from the ROS ecosystem. Through the R4A architectural approach, one may ensure standardizing the robotic applications development in the context of "robotics for citizen developers".

As far as the standardization of the robotic software development is concerned, the robotic development process includes numerous different decisions to be made based on the work-flows used in the past, the strategy of the involved team or even knowledge retrieved from the Internet. Using the proposed architecture, a new process is introduced in the development cycle, since the steps from the robot acquisition to the application distribution are clearly stated.

For citizen developers, although most of the traditional robotics middleware offer tools, ready-to-use libraries and formalisms to support the development of robotic functionality for complex application domains, things are far from easy. Such frameworks aspire to be flexible towards the user needs; this practically means that they intend to solve the problem optimally, rather than easily. Additionally, they do not support reusability, i.e., to easily execute applications (supported by the core functionalities) in different robotic platforms. The correct configuration sometimes lies within the authority of the build-in software stack and cannot be easily changed or even changed at all. The current work aspires to tackle some of the aforementioned mainstream problems.

2.4.1. Installation and Configuration

Concerning the first phase, during which, the developer must investigate the ways to control the robot and fetch data from its sensors, the creation of a robot-aware module is proposed, which will be easily installed and deployed in a robot and will uptake the role of the robotics middleware. Since this will be a per-robot software module, the burden of complex installations and configurations is eliminated.

2.4.2. Remote Data Acquisition and Control of On-Robot Effectors

The R4A approach provides a robot-agnostic API, meaning that the developer will be able to create applications without having perfect knowledge of the robot sensors and effectors, or even the exact morphology of the robot. The creation of such an API is possible, since the information available and the control of different robots has several common elements. The server-side robot API provides web services using a resource-oriented architecture and can enable both the local and remote execution of robotic applications and behaviors. This way, developers will be able to create complex applications, even for low-end robots.

2.4.3. Simplified Testing Procedures

The middleware functionalities should be tested beforehand by robotic experts, and thus only functional testing is required to be performed by the developer. Integration testing is not required due to the utilization of the abstract robotics middleware.

2.4.4. Seamless Portability/Distribution

Since the applications will be robot-agnostic, the procedure to be executed in a different (but similar) robot will simply require the installation of the specialized middleware and the deployment of the application.

3. R4A Approach

R4A is envisioned as a well-defined methodology with a set of tools towards the rapid prototyping of robot-agnostic applications. This chapter is initiated by stating the requirements that the proposed architecture must fulfil (Section 3.1) and continues with the classification and abstraction of robotic resources (Section 3.2). Then, the R4A architecture is explained in depth in Section 3.3 and, finally, the enhanced development procedure by using the R4A is presented in Section 3.5.

3.1. Specifications

By identifying the common problems of current trends in robotics software development processes, the proposed resource-oriented architectural approach has been designed to meet a number of critical requirements towards the rapid development of applications. The core specifications, exported from these requirements, are further described.

3.1.1. Robot Resource Abstraction

An abstraction of the robotic resources, according to their functionalities and not based on their vendor, allowing for the existence of a robot/vendor-agnostic set of services via which a developer can control a robot, must exist. This means that the same application can be seamlessly executed in robots that have similar HW/SW resources, or at least those that require minimum alterations.

3.1.2. Rapid Development

Several simplifications in all of the development procedure phases (middleware setup, design, development, testing and distribution) must be introduced, since the inherent difficulty of all of these phases lies in their heterogeneity when different tools are concerned. This means a standard way to develop applications independently of the underlying middleware must be proposed.

3.1.3. Robotic Middleware Incorporation

The importance of all other robotic middleware must not be nullified, but rather enhanced. Specifically, middleware, such as Player, ROS and OROCOS, among others, must be utilized in the R4A architecture, since they provide HW abstraction, whereas R4A aims to provide robotic abstraction. Conclusively, R4A should not reject other middleware, but should incorporate them to expose their strengths in a unified way.

3.1.4. Remote Execution

One of the most important specifications is the capability of the remote execution of the final applications, i.e., the capability of deploying the application/behavior outside of the physical computational units of a robot. This means that, provided over-network communications to the robot are available, developers should be able to execute their application from any computation unit/node connected to that network (either directly, or via virtual networks, proxies and tunnels). This will have a profound effect on the ability to create complex or heavy-computational applications that can be applied to low-computational-power consumer robots.

3.1.5. Multi-Robot Support

The fact that the development will be performed only using language-specific APIs means that the applications should handle multiple robots in the same code if the aforementioned APIs are initialized with the proper network configuration for each robot. Combined with the remote execution property, this means that the actual application controlling a multi-robot system should be able to be deployed anywhere in the network, either in one of the robots or in another PC.

3.1.6. Enhanced Connectivity

Several transport layers via which the robotic resources are exposed should exist, allowing for the robotic direct and seamless connectivity to non-traditional tools and frameworks. For example, an MQTT or an AMQP layer can connect a robot to an IoT system, whereas the REST or WebSockets layers allow for the connection of robotic applications to the Internet. The latter enables the deployment of robotic applications, even in-browser.

3.1.7. Robot Runtime and Persistent Memory

The concept of a robotic database is missing from most of the frameworks, even though it is a quite useful tool for tasks, such as the retrieval of past information, performing machine learning algorithms on datasets or even applying data fusion in an automatic fashion in some of the resources.

3.1.8. Extendability

Even though the hierarchical categorization of robotic resources is a specification, this should not prohibit the existence of robot-specific calls. Specifically, the introduction of a new resource (e.g., an algorithm) should be quite easy, without altering other resources.

3.1.9. Deployment Flexibility

All inner modules should be able to be deployed in a distributed fashion and communicate via network protocols, either in localhost, a LAN or even over the Internet. This flexibility can be quite beneficial in the deployment phase. For example, a robot may consist of a portable computation node (e.g., a NUC) and two Raspberry PIs 3, on which, web cameras are placed. A possible topology should allow for the camera drivers deployed on the RPI3s, the rest of the robot drivers in the NUC, the memory and the transport layers on a third local PC (or even the cloud) and the application executed in a browser.

3.2. Resource Abstraction

In order to provide a robot (vendor) agnostic set of services, robotic resources must be initially classified in semantic groups, which will then be refined and implemented according to their in-class available operations. Semantic classification is based on the provided functionality of each resource. For example, physical resources are classified as either sensors or effectors due to the distinct separation of roles: sensor resources can be perceived as information providers, whereas effectors are consumers of information and producers of actions within their environment (e.g., motion commands). Sensor resources comprise the following categories: (a) acoustic, (b) vision, (c) distance, (d) chemical, (e) electric, (f) navigation, (g) position, (h) speed, (i) pressure and (j) temperature. On the other hand, R4A abstractions of effector resources comprise five categories: (a) acoustic, (b) optical, (c) main motors, (d) peripheral motors and (e) postures. Table 1 presents the aforementioned classification of a robot's physical resources, along with application examples for each sensor and effector class.

Table 1. Robot resource classification and categorization.

Model Class	Category	Examples
Sensors	Acoustic	Microphones
	Vision	RGB/Depth camera
	Distance	LiDAR, Sonar, IR, point cloud
	Chemical	CO_2, humidity
	Electric	Battery voltage, power consumption
	Navigation	Accelerometers, odometry
	Position	GPS, compasses
	Speed	Tachometers, IMU
	Pressure	Tactiles, bumbers
	Temperature	Temperature sensors
Effectors	Optical	LEDs, lights
	Motion	Motors, base, posture
	Acoustic	Speakers, beepers

Virtual (software-defined) resources are also identified, which refer to in-robot software components implementing various robotics functionalities and resource composition operations. These categories of resources mainly refer to robotic algorithms, such as object detection, SLAM, navigation, path planning, etc.

The overall model of a robot resource is evident in Figure 5. Beyond sensor, actuator and algorithmic resources, composite resources, both physical and virtual, can be defined. For example, a composite physical sensor can be a hardware module that provides both temperature and air quality measurements. Of course, these can also be defined as two separate resources linking to the same hardware component.

3.3. Resource-Oriented Architectural Approach

The in-robot R4A layer can be perceived as a meta-middleware, taking advantage of already established robotics middleware, such as Player, OROCOS, OpenRTM, ROS, etc. Whereas most of them offer hardware abstraction [27], R4A proposes *robot abstraction* by means of a resource-oriented approach. As far as robot-agnostic applications are concerned, the R4A approach allows for exposing robotic resources in a unified and agnostic way. These robot-agnostic resources can be utilized by robotic applications to implement information sources, manipulation commands or algorithmic and behavioral processes.

R4A comprises a resource-oriented architecture and the respective toolset towards the rapid development of vendor-neutral, robot-agnostic software out of software components. The R4A robot architecture is presented in Figure 6. The different architectural components are described below, following a right-to-left fashion.

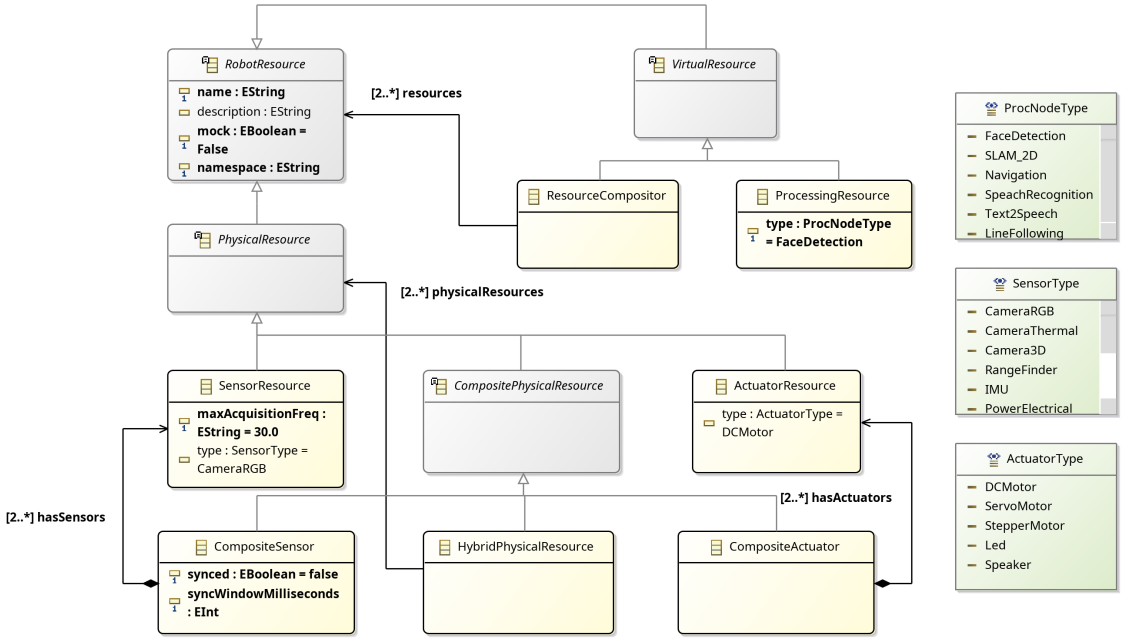

Figure 5. The robot resource entity-relation model. The asterisk (*) in relations refers to "any number". For example, [2..*] indicates a two-to-any (or more-than-one) relation.

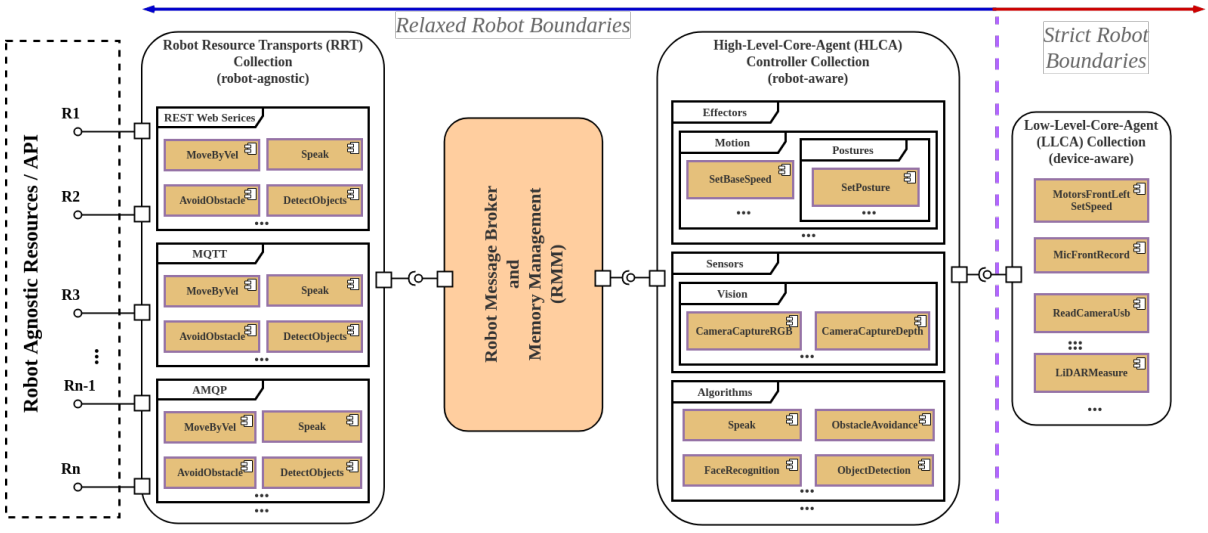

Figure 6. The model of R4A resource-oriented robot architecture.

3.3.1. LLCA—Low Level Core Agent

The most fundamental part is the Low Level Core Agent (LLCA). LLCA is practically the interface/wrapper to the robot. When robots are concerned, they interact with the environment through sensors (the passive or semi-passive devices that offer or produce measurements) and effectors (the devices that act on the environment). Thus, as far as hardware is concerned, the LLCA contains all of the necessary drivers, or higher level modules that directly interact with the drivers of the robotic sensors/effectors. Usually, the modules built in LLCA are either low-level drivers or processes offered by robotics

middleware. For example, if a robot has a depth camera, the most common approach for LLCA to offer the depth data would be utilizing the OpenNI drivers or the ROS-OpenNI package (http://wiki.ros.org/openni accessed on 15 January 2022).

Of course, apart from the hardware, each robot offers software functionalities as well, according to the installed middleware. These may include a variety of algorithms interacting with the hardware, but are at a higher level, such as SLAM, navigation, path planning, etc. Each of these are assumed to pre-exist in the robot, since selecting which (e.g., SLAM) algorithm is suitable for a specific robot cannot be automatically derived from the hardware description of the robotic platform. This is a fact, since the performance and successful operation of a robotic algorithm heavily depends on the robotic morphology, its computational capabilities and the task at hand, as well as the type of environment that the robot operates in. Conclusively, higher-level algorithms that exist in the LLCA can be seen as "software resources", as they either provide "measurements" (i.e., a map) or offer on-demand services (i.e., navigation or path planning).

As evident, the LLCA is *robot-aware*, since it contains hardware-specific code (drivers); therefore, this layer must exist in-robot, this being the reason for denoting its limits as *strict robot boundaries*. Finally, in terms of LLCA connectivity, the resources offer data and services via provided interfaces, so as to be separated from the rest of the architecture.

3.3.2. HLCA—High Level Core Agent

HLCA stands for High Level Core Agent and uptakes the task of connecting to LLCA interfaces for either command execution or data acquisition. One of the most important tasks of an HLCA is to make the robotic services and sources of information robot-agnostic by exposing them via LLCA to the robot memory module (RMM). Thus, abstractions of hardware and software resources of the robot are performed at this layer.

Since an HLCA directly interacts with an LLCA, it is robot-aware as well. However an HLCA can exist out of the physical robot in cases where remote interfaces are provided by the LLCA. This is extremely useful in cases of commercial robots, which are delivered with restricted access to software and hardware layers, usually through an accessible API or SDK. In this case, the HLCA can be deployed on a different (to LLCA) physical machine, and communication with the LLCA (which is practically offered by the manufacturer) is ensured via the provided API/SDK.

3.3.3. RMM—Robot Message Broker and Memory Management

The RMM subsystem uptakes two crucial operations: (a) to aggregate all information produced by the HLCAs, essentially serving as a message broker for the robot-specific temporal sensory and algorithmic information, and (b) to support concurrent streams of varying frequencies. When a streaming connection is established, the corresponding HLCA is notified to adjust the publishing rate based on the highest requested frequency and starts writing data to specific entries of RMM. Each robotic resource holds at least one unique entry. RMM entries are hierarchically structured based on the resource type (e.g., sensor, effector, algorithm), subtype (distance sensor, chemical sensor, etc.) and name, followed by an operation name. For example, urg-04lx LiDAR sensor data are stored and can be retrieved from the "*robot/sensor/distance/urg04lx/front*" entry key (topic) of RMM.

3.3.4. RRT—Robot Resource Transport

RRT instances serve as the intermediary components and expose the underlying robot resources to the application/behavior level (e.g., the mission, task planning and skill levels). These components implement the most common web-based and IoT communication protocols, such as HTTP, MQTT and AMQP, in order to provide maximum platform independence and flexibility for higher-level functionalities. In addition, RRT instances provide proper robot-agnostic application interfaces (APIs), such as '*RobotMove*' or '*CaptureImage*', thus concealing lower level complexity and facilitating the robotic application development process. In Figure 6, only REST web services, websockets and UDP streaming are mentioned; nevertheless, any transport protocol or data schema can be implemented.

3.3.5. RAPI—Robot API

By providing resource-oriented access to the robot sensors and effectors, robotic services are easily consumable by auto-generated clients (REST, Websockets, etc.). Moreover, by adopting software automation techniques (automatic code generation using template engines), for each RRT instance, a robot API (RAPI) call is generated and delivered to application developers in the form of a language-specific library. This library can then be embedded in applications and employed to consume robot resources, acquire sensor measurements and control effectors. Partially, this implies that any number of RAPIs can be embedded in robotic applications depending on the requirements and specifications of the applications themselves. Furthermore, the fact that RAPI calls are common among robots enables the integration of more than just one robot API in a single application, allowing for single-point multi-robot applications while dealing with synchronization problems in-code.

3.4. R4A Robotic Resources as Components

As explained in the start of Section 3, the proposed architecture is heavily influenced by resource-oriented system descriptions, where each resource is defined by the composition of at least one low-level controller instance (LLCA) accompanied by a high-levelabstract controller instance (HLCA), a transport service (RRT) and an entrytopic in the robotic memory RMM (Figure 7). Thus, the overall architecture can be perceived either as different layers that perform different functionalities (LLCA, HLCA, RMM, RRTs) or as a stack of individual resource components.

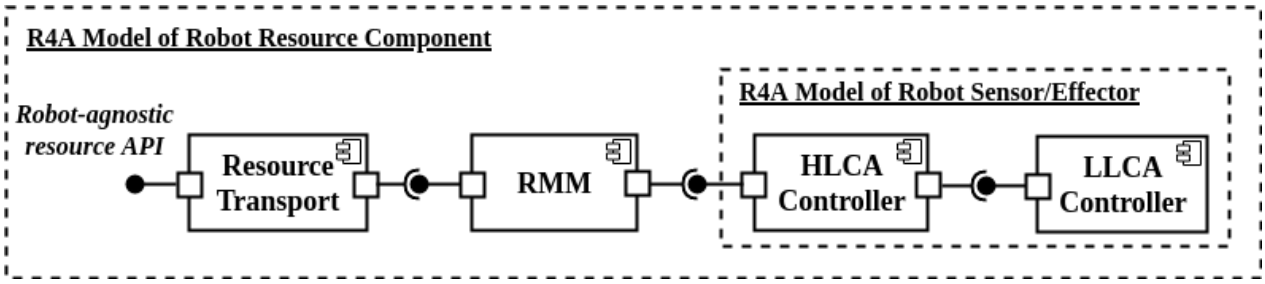

Figure 7. R4A model of robot resource component.

3.5. Rapid Development of Robotic Applications

It is understandable that several of the aforementioned decisions the roboticist have to make are eliminated or simplified. Once more, the five development phases of creating robotic software will be presented, but, this time, the differences between the classic development procedure and the supported one will be stressed out. For comparison reasons, the FSAs were maintained as similar as possible to the original ones; nevertheless, elements that are not needed when the R4A approach is employed are deleted and new elements are denoted with a red dotted line (Figure 8).

3.5.1. Setup Phase

The main difference introduced in the setup phase is the check about the robot being "R4A-compatible" after (or before) its purchasing. The first step is to check if the robot is R4A-compatible. If the developer has the necessary expertise to manually add extra sensors of effectors to the robot's physical body, they must be mechanically attached and connected to the robotic HW controller. Furthermore, if, for these devices, R4A LLCA/HLCA controllers exist, the developer must enhance the robot's R4A resources by integrating the necessary ones in order to support the new HW. Finally, the developer must "install" the appropriate modules in-robot (if they have access) or in a remote PC (since R4A guarantees that remote connectivity to the specific robot exists). Comparing this procedure to the classical one, the developer is relieved of the burden of selecting the necessary middleware to investigate its resources and to set it up in the robot.

3.5.2. Design Phase

The design phase is initiated as long as the setup phase is successfully completed. Here, the developer can search for if the envisioned application is available at the R4A Application Store in order to download it and seamlessly execute it in- or out-of-robot. The aggregation of R4A-enabled applications in a store is valid, since the applications (1) can be robot-agnostic by design and (2) can be seamlessly executed on a robot.

In case the developer decides to create his/her own software, they must choose the programming language. Since the R4A resources are exposed via web protocols, the developer can choose from a large pool of languages, provided they support data transmission over standard protocols (REST, Web-Sockets, IoT, etc.). For example, this means that the application can easily be developed in JS so that it is deployed in a modern browser, which is something that deviates from the classic understanding of what robotics software has been.

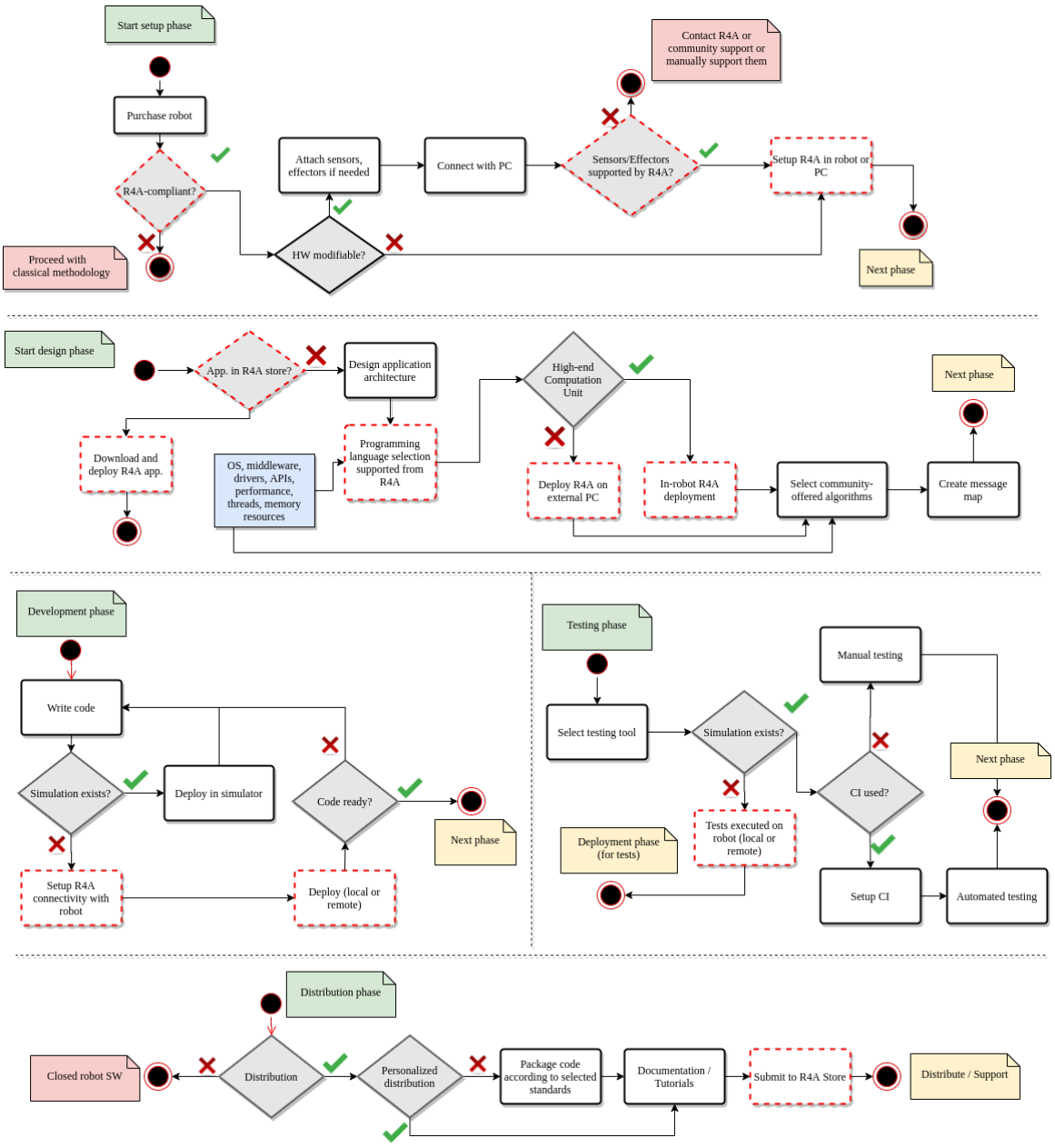

Figure 8. R4A perspective of robotic applications production: setup, design, application development, testing and distribution phases.

3.5.3. Development and Testing Phases

The development phase is also simplified, since the application can be seamlessly executed locally or remotely. The same applies for the testing phase as well. Specifically, the application can be seamlessly executed in a simulator or the real robot. Finally, the fact that the robot's HLCAs and LLCAs have been developed by domain experts and have been extensively tested means that the developer is relieved from the burden of creating multiple unit and functional tests to check the low-level software functionality, leaving him/her only with the task of validating the behavioral (application) part.

3.5.4. Distribution Phase

The distribution phase is essentially the same now that the developer has the ability to easily share applications via cloud repositories/stores. Once again, this is possible due to the lack of effort required to acquire an application and deploy it to another robot.

4. Robot-Agnostic Application Example—A Case Study

In order to demonstrate the effectiveness, usability and rapid prototyping abilities of the R4A approach, a simple robotic application will be described and demonstrated, capable of being executed in two different robots, as evident in Figure 9. Specifically, the naive problem of random exploration with concurrent obstacle avoidance is assumed, followed by image acquisition and object detection capabilities; this is a scenario that can be programmed in any robot with motion capabilities, and is a way to detect obstacles and a camera. The selected robots are a custom build of Turtlebot2 robot (http://www.turtlebot.com/turtlebot2/ accessed on 15 January 2022) and the social humanoid robot NAO (http://doc.aldebaran.com/2-1/home_nao.html accessed on 15 January 2022).

Utilization of the robot API provided by the R4A architecture allows for the rapid development of robot-agnostic applications. As explained earlier in Section 3, our approach hides low-level knowledge and allows developers to program their applications using high-level operations via a single API. Finally, this section discusses the differences between the beaten path and the R4A approach, which lead to increased productivity in terms of the development time, easiness and errors, while developing high-level robotic application scenarios. In Sections 4.1 and 4.2, the conventional way of developing the naive "random exploration with concurrent obstacle avoidance" application for NAO and Turtlebot robots is presented, followed by Section 4.3, where we discuss the robot-agnostic approach introduced in the context of the current study.

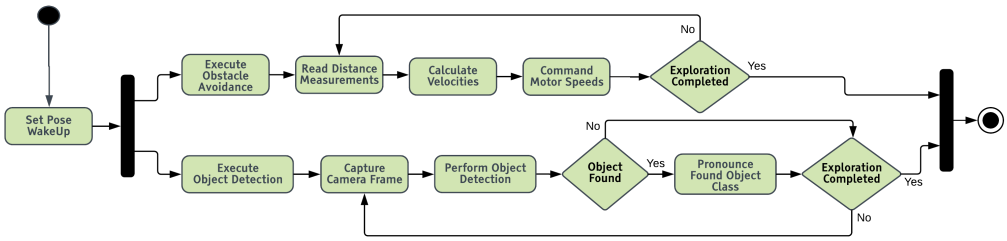

Figure 9. Activity diagram of "random exploration with concurrent obstacle avoidance" robotic application.

4.1. NAO Humanoid Robot—The Conventional Way

Robotic application development for the NAO robot is traditionally performed using the NAOqi SDKs (http://doc.aldebaran.com/2-1/index_dev_guide.html accessed on 15 January 2022), or via official ROS interfaces (http://doc.aldebaran.com/2-5/dev/ros/index_ros.html accessed on 15 January 2022). Softbank Robotics provides and supports implementations of the SDK in four programming languages; Python, C++, Java and JavaScript. On the other hand, utilization of the NAO-specific ROS interfaces can be achieved via ROS client libraries (e.g., Python and C++) that presuppose the installation

of the ROS meta-operating system, NAOqi SDKs (Python and C++) and the official ROS stack to interact with NAOqi. Furthermore, the model-based development of applications (using data-flow graphs) is supported for the NAO robot via the official *Choregraphe Suite* (http://doc.aldebaran.com/2-5/software/choregraphe/index.html accessed on 15 January 2022); however, in the context of this paper, this will not be investigated, since it offers limited functionality for comparison.

The Python implementation of the NAOqi SDK was used due to the rapid prototyping capabilities of the language and minimal required setup and testing effort compared to ROS interfaces. Furthermore, since an approach utilizing ROS is described next, the use of a different framework of added value is considered, so as to showcase different development strands.

Based on the aforementioned, the *setup phase* requires the installation and configuration of the NAOqi SDK on the host machine where the application will be executed. Installation instructions can be obtained from the official documentation for the NAO robot (http://doc.aldebaran.com/2-1/dev/python/install_guide.html accessed on 15 January 2022). To test that everything has been properly installed and configured, one must write a simple demo application that connects to NAOqi proxies and utilizes (for example) the *speak* operation of the *ALTextToSpeech* proxy. A successful execution of this demo application leads towards the actual robotic application development phase, though, before writing the source code of the robotic application, developers must first study the NAO-specific APIs in order to have an overall understanding of the capabilities and provided interfaces of both the robot and the NAOqi framework (http://doc.aldebaran.com/2-1/ref/index.html accessed on 15 January 2022).

Descriptively, the example application that tackles the naive problem of random exploration with concurrent obstacle avoidance, followed by image acquisition and object detection capabilities, utilizes the following (core) NAOqi API proxy endpoints:

- ALMotionProxy::wakeUp: This is the initial command sent to the robot. The robot wakes up, sets the state of the motors to active, commands it to go to its initial position (stand) and finally sets the stiffness of the motor;
- ALMotionProxy::moveToward: This commands the robot to move at given y, x, z, theta velocities. This operation affects the overall body of the robot and not individual motors/joints;
- ALTextToSpeech::say: This utilizes the in-robot speech synthesis engine, transforming plain text to audio data, and performs playback via the NAO embedded speakers. This operation is used to inform (pronounce) about detected objects;
- ALVideoDevice: This module provides image streams obtained from the robot's embedded front camera. It is used to obtain scene image frames to be subsequently sent to an object detector;
- ALSonar and ALMemory: The ALSonar module gives access to the ultrasonic sensor hardware and allows it to start data acquisition. In composition with the ALMemory module, it is possible to subscribe to sonar events and collect data, as long as they are available.

In Figure 10, the obstacle avoidance application is presented in the form of a sequence diagram. It is important to observe the need to develop classes in order to support asynchronous operations. This adds another dependency to the overall application development process, which is the knowledge of threaded or event-loop asynchronous programming in Python, a dependency that is offloaded from the application development process to the middleware layer in case of utilizing the R4A architecture. Namely, the *ObjectDetector* and *Exploration* classes implement asynchronous execution operations, a feature that is required for the concurrent execution of these distinct tasks. Both the Exploration and Object Detection tasks are performed in a loop, until the termination of the application. The first calculates the robot's body velocity for each next time step based on the distance measurements acquired from NAO embedded sonar sensors and random selection. Concurrently, the *ObjectDetector* acquires image streams from the NAO embedded front camera,

sends each one to a remote object detection web service (*ObjDetectorProxy*) and commands NAO to pronounce any found object classes. For this demonstration, the R4A platform (formerly the RAPP platform) (https://github.com/rapp-project/rapp-platform accessed on 15 January 2022) object recognition web service was used in order to remotely perform object detection operations.

4.2. Turtlebot2—The Conventional Way

Concerning the Turtlebot2 platform, there exist community ROS packages that provide ROS services, actions and topics and enable the control of the base of the robot by sending motion commands under a ROS environment. Thus, it is preceded by a hardware setup of the Turtlebot2 platform in order to support the development of the "*random exploration with concurrent obstacle avoidance, followed by object detection*" application, which requires the robot to have a camera and a distance sensor (e.g., LiDAR) installed.

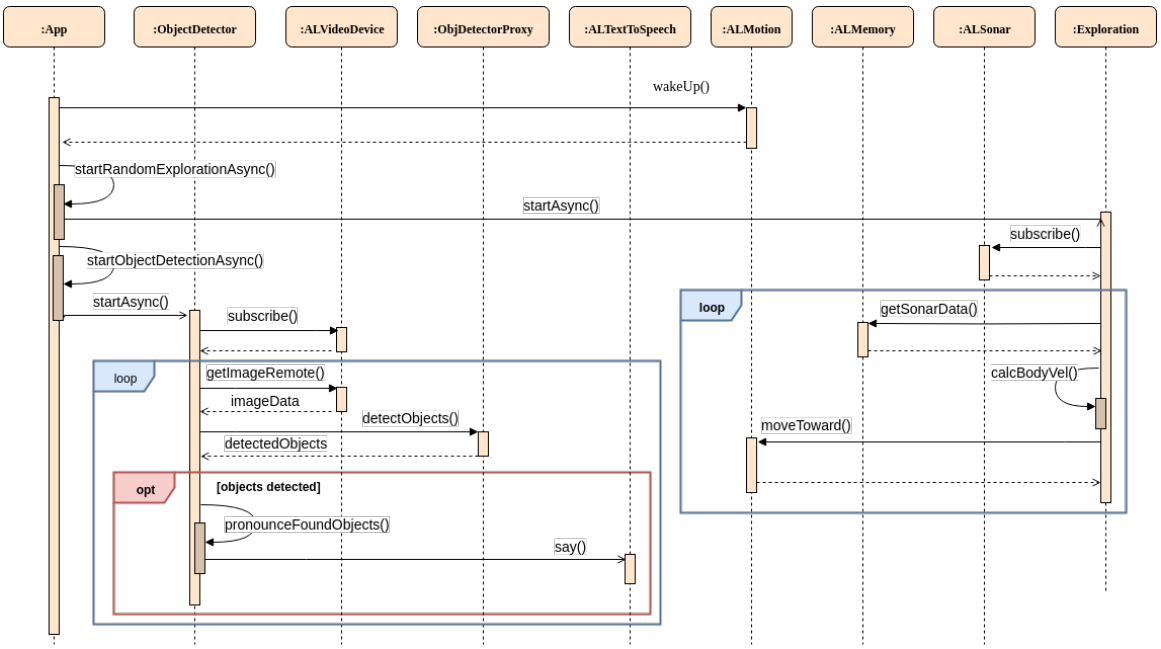

Figure 10. Sequence diagram of "random exploration with concurrent obstacle avoidance" robotic application for NAO robot, using the NAOqi SDK.

In contrast to the NAO robot, a bare Turtlebot2 platform does not include a central processing node, a camera and a distance sensor. It only provides a USB interface for sending motion commands to the base and acquiring feedback information from motor encoders and battery voltage levels. Thus, a mini PC, laptop or embedded board must first be placed on the Turtlebot2 platform and connected to the USB2 port. Furthermore, an RGB camera and a LiDAR are placed and connected to the central node via provided hardware interfaces (e.g., USB or Ethernet). The hardware setup consists of an RPLiDAR (http://www.slamtec.com/en/Lidar/A3 accessed on 15 January 2022) placed on the top level of the Turtlebot2 platform, an RGB web camera placed at the middle layer and an Intel NUC (https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html accessed on 15 January 2022) miniPC at the bottom.

Based on the previously defined hardware setup and beyond the basic ROS environment, the Turtlebot base platform (https://wiki.ros.org/turtlebot_node accessed on 15 January 2022), RPLiDAR (http://wiki.ros.org/rplidar accessed on 15 January 2022) and usb-cam (http://wiki.ros.org/usb_cam accessed on 15 January 2022) ROS drivers must be installed and configured on the Intel NUC miniPC, assuming a Linux operating system that is supported by ROS was previously installed.

Next, developers must identify and design the architecture of the software that implements the desired application, including how many and which ROS nodes will be created, what the message map will be and what type of communication channel will be used (ROS messages, services or actions). In this case, since the application is quite simple, a single ROS node would be developed. In order for the robot to perform the task at hand, the following actions must materialize:

- A ROS subscriber that binds the LiDAR ROS topic to a callback, which will handle the data. This ROS topic is usually named */scan*;
- A ROS publisher to be used for setting velocities. The publisher must be bound to the */cmd_vel* topic and the velocities must be of type *geometry_msgs* :: *Twist*;
- A callback function that will receive *sensor_msgs* :: *LaserScan* messages, compute the velocities and publish them in the */cmd_vel* topic;
- A ROS subscriber that binds the RGB camera topic to a callback (*/camera_name/image*);
- A callback that will receive the published images from the camera topic in the form and implement a call to the ObjectDetection service of the R4A platform.

During the development process, it is a common approach to incrementally test the correctness of the code. In the current case, the developer can either test the code in a standard PC using a simulator or transfer the code to the computational unit attached on Turtlebot and remotely deploy it. The overall architecture of the Turtlebot case is presented in Figure 11.

As evident, even if this is a quite simple application (since no complex messaging using ROS services or ROS actions was required, or more advanced tools, such as *tf*), the steps to create the ROS software are not as straightforward as in the NAO case. Specifically, here, the application is distributed, there is more than one executable and the way to acquire the data via pub/sub and callbacks is intuitive for more experienced programmers. This opinion can be found in several robotics forums, stating that the ROS learning curve is quite steep, but, of course, its employment is much better than trying to carry out everything from scratch.

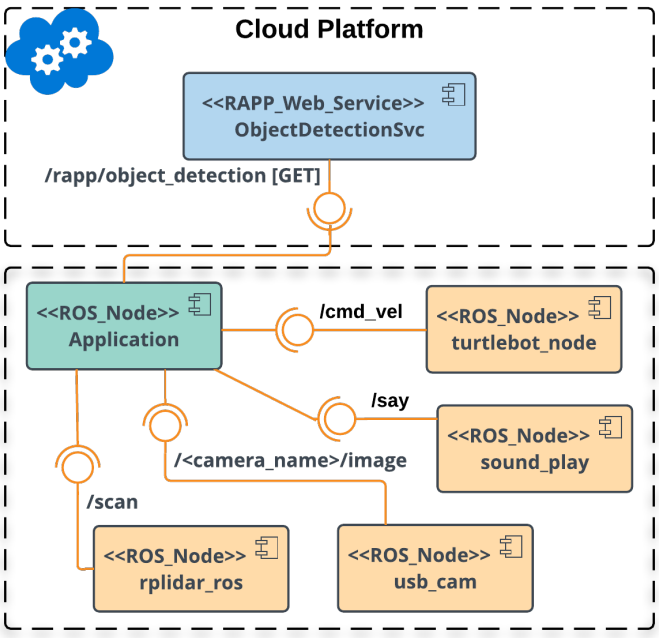

Figure 11. Component diagram of Turtlebot architecture.

4.3. Robot-Agnostic Development of Applications

As the previous sections showcased, the development of simple applications in different robots includes vastly heterogeneous skills and frameworks. In the case of the R4A approach, several of these inter-development procedures become easier or common between different platforms. R4A adds a middleware on top of these robot-specific frameworks and provides an abstract API for robots that is accessible via well-established web and IoT communication protocols, such as REST, Websockets, MQTT and AMQP.

The sequence diagram of the "random exploration with concurrent obstacle avoidance" application, is evident in Figure 12. The application utilizes the robot API to obtain images from the installed camera, read from sonar distance sensors and send motion commands to the motors, whereas the cloud API is used for performing object detection on input images from the robots' camera. Notice that the Websockets transport of the API (RApiWS) is used to read from sensors due to the asynchronous nature of the protocol, which provides means of delivering events and messages to the client via PubSub channels. On the other hand, the REST transport is used for sending commands to effector resources, such as the motors of the robots in this case.

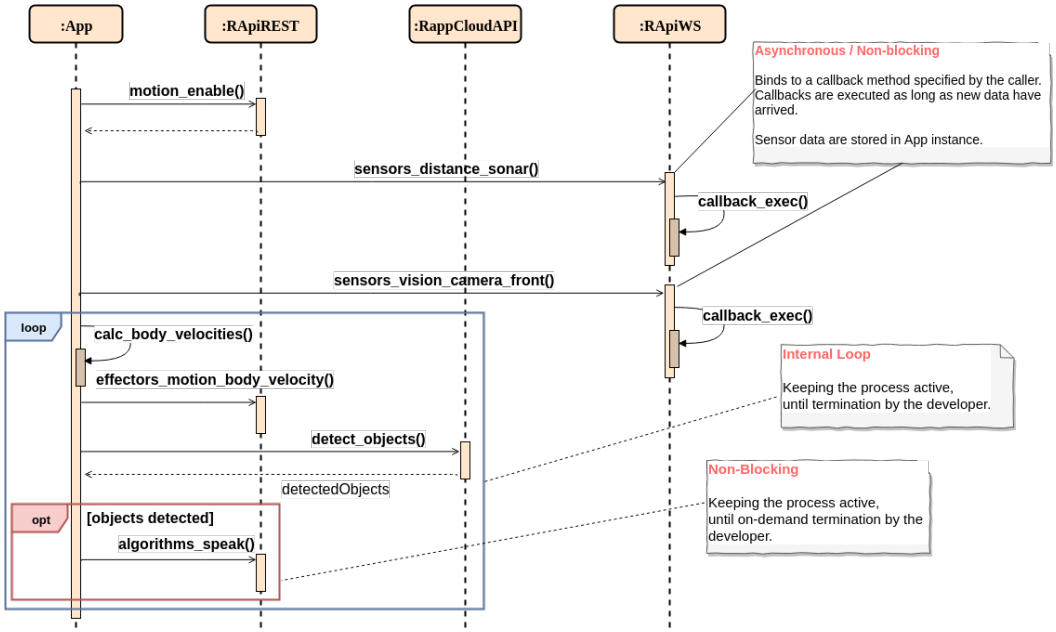

Figure 12. Sequence diagram of "random exploration with concurrent obstacle avoidance" robot-agnostic application, developed using the R4A architecture.

Finally, the application is robot-agnostic, meaning that it can be executed in different robots, provided that they meet its requirements. In this example, due to the fact that both NAO and Turtlebot robots have a camera and sonar distance sensors installed and have navigation capabilities, the application can be executed without any modifications to the source code. This promotes the reusability of whole applications or parts of applications and enables future studies on the distribution and remote execution for different robot types, via dedicated MDE processes, to perform transformations.

5. Discussion

In a nutshell, R4A offers robotic abstraction (in contrast to the rest of the middleware, which offers hardware abstraction), enables the rapid development of robotic applications and supports already existent robotics middleware, the remote execution of applications and development of multi-robot applications. Furthermore, the proposed architecture offers enhanced connectivity between robots via local or web and IoT protocols and robot memory functionalities, such as persistent and runtime (cache) data storage.

The current study proposes a robot-agnostic approach in order to simplify the configuration, reusability and control of robots. It is argued that this standardization will also enable the reusability of applications among heterogeneous robots, as well as solutions for executing heavy duty software on low-resource robots. Our approach utilizes model-driven engineering principles and technologies to model and automate aspects of the development process of robotic applications. The purpose of this study is to enable access to the robotics application development domain for a wider range of end-users by hiding low-level technological and technical knowledge, such as firmware development, networking, operating systems and communication protocols. Furthermore, our approach promotes the remote control and monitoring of robots via the auto-generated robot API gateways, which give remote access to the robots' resources via web and IoT technologies Nevertheless, R4A has certain limitations as well.

5.1. Limitations

One of the greatest strengths of the proposed architecture is the deployment flexibility, which is, at the same time, one of the greatest drawbacks. If the network is absent, or if the connectivity is not stable, the deployed application may not operate as intended. Of course, if the developer/owner of the robot suspects that the robot connectivity is unstable, they can easily deploy everything (including the application) in-robot, ensuring that a possible network disconnection will not affect the application execution. QoS functionalities and rules are not supported by the current implementation and are considered as future work.

Since network layers exist between the data generation and consumption modules, it is expected that no real time operations should be executed, since no guarantees concerning the channel health or bandwidth can exist. Thus, when time-critical operations must take place (e.g., multiple motor synchronization or manipulation), these must be implemented as LLCAs; therefore, the critical operations will be locally executed. The integration of QoS aspects is, again, under research.

Even though resource abstractions allow for robotic application agnosticity, this only applies to robots with similar HW capabilities. For example, in the aforementioned obstacle avoidance application, the behavior will be correct if and only if the distance sensors are placed at the same physical location of the robot (within a specific range). If robots share similar resources but with different configurations, the application cannot be deployed as is, but R4A allows for quick and easy recalibration, since only high-level concepts exist. In this case, a more detailed robot model must exit that will include physical concepts in order to allow for such complex configurations. Finally, in case the robots are vastly heterogeneous, i.e., they do not share similar HW resources, it makes no sense for the same application to be executed in both.

Another drawback of the R4A approach is that it increases the work effort of robotics experts, since each new robot or new alteration of an already existent robot (e.g., the addition of an extra sensor) must be followed by the adaptation of the inner-modules of R4A in order for these changes to leave the application developers unaffected. Nevertheless, it is preferable to burden the experts and not the beginners with such tasks, as robotics experts will accomplish them in a relatively smaller amount of time.

It should be clearly stated that the core functionalities and modules supporting the rapid prototyping of robotic applications offered by the R4A approach can only be maintained by domain experts. Therefore, when a new resource (for a specific robot) is required by the application developers (e.g., the integration of extra sensors/effectors or implementation of core algorithmic functionalities), the model and implementation of that resource must be enhanced by robotics domain experts.

The aforementioned drawbacks and limitations of the R4A approach can be summarized as:

- Network existence necessity;
- Prohibited real-timeness;
- Not complete robotic agnosticity;
- Continuous robotic support effort;
- Extendability of robot resources.

5.2. Threats to Validity

The usage of experiments to evaluate and compare algorithms is a recent trend. The same can be said for software engineering, since major experimental studies in the area are usually no more than twenty years old. Search-based software engineering (SBSE) experiments share the limitations born out of the immaturity in both its source areas. One of these limitations regards the lack of a list of validity threats that may affect SBSE experiments. In our study, we introduce a series of abstractions applied for the robotics application development domain, as proposed by the model-driven engineering paradigm and the model-driven architecture (MDA) [38], so as to lower the complexity and required time to develop robotic applications. Much research adopting the MDA paradigm in domain-specific software engineering [39] and development tasks resulted in more effective and robust software, while also lowering the complexity, time and errors [40], resulting in a productivity improvement of standalone developers and development teams. Our case study (Section 4) was internally evaluated by four robotics developers, and self-reported the results for productivity improvement. However, we consider the self-reported results to be threat-negligible, since the developers established and applied a solid methodology to measure the productivity gain based on the Goal-Question-Metric approach [41]. Furthermore, the developers used their preferred methodology, frameworks and tools for the manual build of the in-robot software stack. Since each developer used technologies that they are highly familiar with, it resulted in a higher productivity during the manual build phase.

Another possible threat is regarding the development of time-critical robotic applications. Our approach of remotely controlling robots via web and IoT transports introduces a network layer, which, in many cases, has unstable transmission rates and a high latency. Such time critical applications may include real-time access to the hardware and synchronization tasks (e.g., grasping via visual feedback). Network delays introduced by the R4A architecture must be considered while developing applications, while network failures must also be taken into account. For example, a disconnection of the robot while an application is running should never lead to faulty states and unwanted behaviours. In these cases, our approach suggests an implementation of the time-critical components at the LLCA and HLCA layers of the R4A architecture.

Even though it is stated that applications are robot-agnostic, this does not necessarily mean that it can be executed on two custom robots (e.g., Turtlebots), because it depends on whether the same sensors and effectors utilized by the application exist at exactly the same place. Many robotics algorithms, such as Navigation, SLAM and Computer Vision, require specific hardware when configured to work properly; thus, in this case too, it is highly recommended to implement such algorithms as LLCA and HLCA components and to provide high-level control interfaces to the applications via the robot API.

Finally, to enable the automated generation of the software-defined middleware (HLCAs, LLCAs, transports, etc.), the robot must be supported by the R4A architecture. For example, in cases of building custom robots with custom controllers, LLCAs and HLCAs must be developed and integrated into R4A before being able to utilize code generation processes. Notice that, in this case, R4A is able to automate the development process of the RMM and RRT components and to enable remote access to the robot resources.

6. Conclusions and Future Work

In the context of the proposed R4A architectural approach , robot resources are exposed in a homogeneous and standardized way, and thus automate the robot system and application design and development processes. We argue that this is expected to lead to a reduction in the development time and will allow for the wider penetration of low-cost robots in the market.

R4A is not aspiring to replace commonly used tools and frameworks (e.g., ROS), but to build on top of them and make them available in an integrated, vendor-agnostic and robot-agnostic context, opening the gates of robotic application production to citizen developers by reducing excessive hand-coding and domain-specific technical knowledge requirements. R4A hides low-level, irrelevant-to-robotics details that often make it hard for practitioners to focus on the actual robotic application domain, requiring disproportionate expertise and effort, especially for SMEs and non-robotics experts. Finally, via R4A, robotic applications can be deployed literally everywhere, promoting the cloud robotics, as well as the IoT concepts, since robots can now be perceived as "Things" [42,43], providing sensing and actuation functionalities.

We intend to adapt the R4A methodology and framework to the model-driven engineering (MDE) baseline, envisioning: (a) being part of the standardization of robotics development practices, (b) enabling a wide range of practitioners to benefit from the advantages of a unified robotics modeling and development approach and (c) the rapid adaptation and integration of community-driven domain expertise. In addition, MDE techniques and processes can be applied to offload domain expertise requirements and can allow for the fast prototyping of robot-agnostic applications. Several modeling tools and frameworks have been proposed that target the robotics domain [26,30,44,45], though none of these focus on the entire application development process and mainly provide metamodels for specific (sub)-domains, such as service robots, home assistance, home automation, etc. Furthermore, beyond the exposition of robotic resources to the outer world via well-established web protocols, several qualitative and control functionalities will be investigated for integration within or next to already existing functionalities, such as QoS, fault-detection and domain specific fallback strategies.

Finally, networking, cloud infrastructures and IoT technologies are exponentially evolving, enabling the integration of robots into a wider range of application domains, such as home automation, home assistance and healthcare [46,47]. Cloud technologies can be used for the creation of a repository of robotics-oriented services in order for developers to be able to use off-the-shelve complex algorithms, such as face and object detectors, speech synthesis and recognition, natural language processing, etc. The cloud repository will be open for submissions from robotic developers, essentially establishing an online and ready-to-use database of functionalities. Furthermore, cloud technologies can also be used for the monitoring and remote control of robots.

Author Contributions: All authors contributed equally to the study conception and the methodology (K.P., E.T., C.Z., A.L.S., L.P.). Software development for this study was performed by K.P., E.T. and C.Z. The first draft of the manuscript was written by E.T. and K.P., and all authors contributed to the review of the final manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research is co-financed by Greece and the European Union (European Social Fund—ESF) through the Operational Programme "Human Resources Development, Education and Lifelong Learning" in the context of the project "Strengthening Human Resources Research Potential via Doctorate Research" (MIS-5000432), implemented by the State Scholarships Foundation (IKY).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of Things
R4A	Robotics4All
SW	Software
HW	Hardware
PC	Personal Computer
LLCA	Low-level Core Agent
HLCA	High-level Core Agent
RMM	Robot Message Broker and Memory Management
RTT	Robot Resource Transport
RAPI	Robot Application Interface
SLAM	Simultaneous Localization and Mapping
OS	Operating System
SDK	Software Development Kit
API	Application Interface
CI	Continues Integration
REST	Representation State Transfer
MDE	Model-Driven Engineering
QoS	Quality of Service

References

1. Catsoulis, J. *Designing Embedded Hardware: Create New Computers and Devices*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2005.
2. Thrun, S. Probabilistic robotics. *Commun. ACM* **2002**, *45*, 52–57. [CrossRef]
3. Tsardoulias, E.; Mitkas, P. Robotic frameworks, architectures and middleware comparison. *arXiv* **2017**, arXiv:1711.06842.
4. Waszkowski, R. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine* **2019**, *52*, 376–381. [CrossRef]
5. Elhadi, S.; Marzak, A.; Sael, N.; Merzouk, S. Comparative study of IoT protocols. In *Smart Application and Data Analysis for Smart Cities (SADASC'18)*; Springer: Marrakesh, Morocco, 2018.
6. Coste-Maniere, E.; Simmons, R. Architecture, the backbone of robotic systems. In Proceedings of the 2000 ICRA. Millennium Conference, IEEE International Conference on Robotics and Automation, Symposia Proceedings (Cat. No. 00CH37065), San Francisco, CA, USA, 24–28 April 2000; Volume 1, pp. 67–72.
7. Albus, J.; McCain, H.; Lumia, R. *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*; NBS Tech. Technical report, Note 1235; NIST Research Information Center: Gaithersburg, MD, USA, 1987.
8. Albus, J.S. Outline for a theory of intelligence. *IEEE Trans. Syst. Man Cybern.* **1991**, *21*, 473–509. [CrossRef]
9. Brooks, R. A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **1986**, *2*, 14–23. [CrossRef]
10. Arkin, R.C. Motor schema—Based mobile robot navigation. *Int. J. Robot. Res.* **1989**, *8*, 92–112. [CrossRef]
11. Arkin, R.C. Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-Made Environments. Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1987.
12. Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S.; Oreback, A. Towards component-based robotics. In Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton, AB, Canada, 2–6 August 2005; pp. 163–168.
13. Stewart, D.B.; Khosla, P.K. Rapid development of robotic applications using component-based real-time software. In Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems, Human Robot Interaction and Cooperative Robots, Pittsburgh, PA, USA, 5–9 August 1995; Volume 1, pp. 465–470.
14. Fernandez, J.A.; Gonzalez, J. NEXUS: A flexible, efficient and robust framework for integrating software components of a robotic system. In Proceedings of the 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146), Leuven, Belgium, 20–20 May 1998; Volume 1, pp. 524–529.
15. Fleury, S.; Herrb, M.; Chatila, R. A tool for the specification and the implementation of operating modules in a distributed robot architecture. *Comput. Stand. Interfaces* **1999**, *6*, 429. [CrossRef]
16. Stasse, O.; Kuniyoshi, Y. Predn: Achieving efficiency and code re-usability in a programming system for complex robotic applications. In Proceedings of the 2000 ICRA. Millennium Conference, IEEE International Conference on Robotics and Automation, Symposia Proceedings (Cat. No. 00CH37065), San Francisco, CA, USA, 24–28 April 2000; Volume 1, pp. 81–87.
17. Nesnas, I.A.; Volpe, R.; Estlin, T.; Das, H.; Petras, R.; Mutz, D. Toward developing reusable software components for robotic applications. In Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180), Maui, HI, USA, 29 October–3 November 2001; Volume 4, pp. 2375–2383.

18. Volpe, R.; Nesnas, I.; Estlin, T.; Mutz, D.; Petras, R.; Das, H. The CLARAty architecture for robotic autonomy. In Proceedings of the 2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542), Big Sky, MT, USA, 10–17 March 2001; Volume 1, pp. 1–121.
19. Collett, T.H.; MacDonald, B.A.; Gerkey, B.P. Player 2.0: Toward a practical robot programming framework. In Proceedings of the Australasian conference on robotics and automation (ACRA 2005), Sydney, Australia, 5–7 December 2005; Citeseer: State College, PA, USA, 2005; p. 145.
20. Bruyninckx, H. Open robot control software: The OROCOS project. In Proceedings of the 2001 ICRA, IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164), Seoul, Korea, 21–26 May 2001; Volume 3, pp. 2523–2528.
21. Magnenat, S.; Longchamp, V.; Mondada, F. ASEBA, an event-based middleware for distributed robot control. In Proceedings of the Workshops and Tutorials CD IEEE/RSJ 2007 International Conference on Intelligent Robots and Systems, San Diego, CA, USA, 29 October–2 November 2007; IEEE Press: New York, NY, USA, 2007.
22. Cote, C.; Brosseau, Y.; Letourneau, D.; Raïevsky, C.; Michaud, F. Robotic software integration using MARIE. *Int. J. Adv. Robot. Syst.* **2006**, *3*, 10. [CrossRef]
23. Uhl, K.; Ziegenmeyer, M. MCA2-an extensible modular framework for robot control applications. In *Advances in Climbing and Walking Robots*; World Scientific: Singapore, 2007; pp. 680–689.
24. Calisi, D.; Censi, A.; Iocchi, L.; Nardi, D. OpenRDK: A modular framework for robotic software development. In Proceedings of the 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, Nice, France, 22–26 September 2008; pp. 1872–1877.
25. Ando, N.; Suehiro, T.; Kotoku, T. A software platform for component based rt-system development: Openrtm-aist. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 87–98.
26. Steck, A.; Lotz, A.; Schlegel, C. Model-driven engineering and run-time model-usage in service robotics. *ACM Sigplan Not.* **2011**, *47*, 73–82. [CrossRef]
27. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*; IEEE Press: Kobe, Japan, 2009; Volume 3, p. 5.
28. Beetz, M.; Mösenlechner, L.; Tenorth, M. CRAM—A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, Taiwan, 18–22 October 2010; pp. 1012–1017.
29. Tenorth, M.; Beetz, M. KnowRob—Knowledge processing for autonomous personal robots. In Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, St. Louis, MO, USA, 11–15 October 2009; pp. 4261–4266.
30. Bruyninckx, H.; Klotzbücher, M.; Hochgeschwender, N.; Kraetzschmar, G.; Gherardi, L.; Brugali, D. The BRICS component model: A model-based development paradigm for complex robotics software systems. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1758–1764.
31. Schlegel, C.; Haßler, T.; Lotz, A.; Steck, A. Robotic software systems: From code-driven to model-driven designs. In Proceedings of the 2009 IEEE International Conference on Advanced Robotics, Singapore, 14–17 July 2009; pp. 1–8.
32. Joyeux, S.; Albiez, J. Robot development: from components to systems. In Proceedings of the 6th National Conference on Control Architectures of Robots, Grenoble, France, 8–14 May 2011; pp. 15p.
33. Fernández-Madrigal, J.A.; Galindo, C.; González, J.; Cruz-Martín, E.; Cruz-Martín, A. A software engineering approach for the development of heterogeneous robotic applications. *Robot. Comput.-Integr. Manuf.* **2008**, *24*, 150–166. [CrossRef]
34. Fernández-Madrigal, J.A. *The BABEL Development System for Integrating Heterogeneous Robotic Software*; Technology Report System Engineering and Automation Department, University of Malaga: Malaga, Spain, 2003.
35. Tsardoulias, E.G.; Kintsakis, A.M.; Panayiotou, K.; Thallas, A.G.; Reppou, S.E.; Karagiannis, G.G.; Iturburu, M.; Arampatzis, S.; Zielinski, C.; Prunet, V.; et al. Towards an integrated robotics architecture for social inclusion–The RAPP paradigm. *Cogn. Syst. Res.* **2017**, *43*, 157–173. [CrossRef]
36. Reppou, S.E.; Tsardoulias, E.G.; Kintsakis, A.M.; Symeonidis, A.L.; Mitkas, P.A.; Psomopoulos, F.E.; Karagiannis, G.T.; Zielinski, C.; Prunet, V.; Merlet, J.P.; et al. Rapp: A robotic-oriented ecosystem for delivering smart user empowering applications for older people. *Int. J. Soc. Robot.* **2016**, *8*, 539–552. [CrossRef]
37. Szlenk, M.; Zieliński, C.; Figat, M.; Kornuta, T. Reconfigurable agent architecture for robots utilising cloud computing. In *Progress in Automation, Robotics and Measuring Techniques*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 253–264.
38. Soley, R. Model driven architecture. *OMG White Pap.* **2000**, *308*, 5.
39. Fowler, M. *Domain-Specific Languages*; Pearson Education: Upper Saddle River, NJ, USA, 2010.
40. Pastor, O.; Molina, J.C. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*; Springer: Berlin/Heidelberg, Germany, 2007; Volume 1.
41. Caldiera, V.R.B.G.; Rombach, H.D. The goal question metric approach. *Encycl. Softw. Eng.* **1994**, 528–532.
42. Yan, H.; Hua, Q.; Wang, Y.; Wei, W.; Imran, M. Cloud robotics in smart manufacturing environments: Challenges and countermeasures. *Comput. Electr. Eng.* **2017**, *63*, 56–65. [CrossRef]
43. Simoens, P.; Dragone, M.; Saffiotti, A. The Internet of Robotic Things: A review of concept, added value and applications. *Int. J. Adv. Robot. Syst.* **2018**, *15*, 1729881418759424. [CrossRef]
44. Schlegel, C.; Lotz, A.; Lutz, M.; Stampfer, D.; Inglés-Romero, J.F.; Vicente-Chicote, C. Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot. *it-Inf. Technol.* **2015**, *57*, 85–98. [CrossRef]

45. Bubeck, A.; Weisshardt, F.; Verl, A. BRIDE-A toolchain for framework-independent development of industrial service robot applications. In Proceedings of the ISR/Robotik 2014, 41st International Symposium on Robotics, VDE, Munich, Germany, 2–3 June 2014; pp. 1–6.
46. Jordan, S.; Haidegger, T.; Kovács, L.; Felde, I.; Rudas, I. The rising prospects of cloud robotic applications. In Proceedings of the 2013 IEEE 9th International Conference on Computational Cybernetics (ICCC), Tihany, Hungary, 8–10 July 2013; pp. 327–332.
47. Beigi, N.K.; Partov, B.; Farokhi, S. Real-time cloud robotics in practical smart city applications. In Proceedings of the 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Montreal, QC, Canada, 8–13 October 2017; pp. 1–5.