Thesis for The Degree of Licentiate of Engineering

Programming Robots for Activities of Everyday Life

Swaib Dragule

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2021

Programming Robots for Activities of Everyday Life

Swaib Dragule

Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

"Smiling to someone is an act of charity."
- Prophet Muhammad (P.B.U.H)

Abstract

Text-based programming remains a challenge to novice programmers in all programming domains, including robotics. The use of robots is gaining considerable traction in several domains since robots are capable of assisting humans in repetitive and hazardous tasks. Soon robots will commonly be used in tasks-of-everyday-life in homes, hotels, airports, and museums. However, robotic missions have been either predefined or programmed using low-level APIs, making mission specification task-specific and error-prone. To harness the full potential of robots, it must be possible to define missions for specific application domains as needed. The specification of missions of robotic applications should be performed via easy-to-use, accessible ways, and at the same time, be accurate and unambiguous. Simplicity and flexibility in programming such robots are important since end-users come from diverse domains, not necessarily with sufficient programming knowledge.

The main objective of this licentiate thesis is to empirically understand the state-of-the-art in languages and tools used for specifying robot missions by end-users. The findings will form the basis for interventions in developing future languages for end-user robot programming.

During the empirical study, DSLs for robot mission specification were analyzed through published literature, their websites, user manuals, sample missions, and using the languages to specify missions for supported robots.

After extracting data from 30 environments, 133 features were identified. A feature matrix mapping the features to the environments was developed with a feature model for robotic mission specification DSLs.

Our results show that most end-user facing environments exist in the education domain for teaching novice programmers and STEM subjects. Most of the visual languages are developed using Blockly and Scratch libraries. The end-user domain abstraction needs more work since most of the visual environments abstract robotic and programming language concepts but not end-user concepts. In future works, it is important to focus on the development of reusable libraries for end-user concepts; and further, explore how end-user facing environments can be adapted for novice programmers to learn general programming skills and robot programming in low resource settings in developing countries, like Uganda.

Keywords

Mobile robots, Mission specifications, Robot missions, Robotic IDEs, Domain specific languages, End-user programming.

Acknowledgment

My special appreciation goes to my supervisors, Patrizio Pelliccione, Thorsten Berger, and Engineer Bayinomugisha for nurturing my research potential from scratch. Your encouragements, close follow-ups, and patience made it easy for me to continue with this research work amidst several social and technical storms during the study.

I want to thank the SIDA-Bright 317 family, both the staff and the students. By sharing each other's ups and downs, I picked the confidence that it is possible to accomplish this task. I also want to acknowledge the support from my co-authors B. Meyers, S.G. Gonzalo, C. Menghi R. Ghzouli, A. Wasowski, and E. B. Johnsen. I learned a lot from you. My Kiswahili teachers Lucy and Salome, thanks for the social support.

To my family, thank you so much for the patience, and the psycho-social support. You paid a huge price during the study by missing me in your lives. Amidst all the social storms, you stood strong with me. May Allah reward you abundantly and bless the family with more blessings.

The authors acknowledge financial support from the SIDA Bright 317 project. The Centre of Excellence on Connected, Geo-Localized and Cyber-secure Vehicle (EX-Emerge), funded by the Italian Government under CIPE resolution n.70/2017 (Aug. 7, 2017). The work is also supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme GA No. 694277 and GA No. 731869 (Co4Robots).

List of Publications

Appended publications

This thesis is based on the following publications:

[A] S. Dragule, S. Garcia, T. Berger, and P. Pelliccione. "Languages for Specifying Missions of Robotic Applications." *Software Engineering for Robotics. Springer (2020)*

[B] S. Dragule, T. Berger, C. Menghi, and P. Pelliccione. "A Survey on the Design Space of End-User Oriented Languages for Specifying Robotic Missions." *In submission to International Journal of Software and Systems Modeling (SoSyM).*

[C] S. Dragule, B. Meyers, and P. Pelliccione. "A generated property specification language for resilient multirobot missions." *International Workshop on Software Engineering for Resilient Systems, pp. 45–61, Springer, 2017.*

Other publications

The following publication was published during my PhD studies.However, they are not appended to this thesis, due to contents overlapping that of appended publications.

[a] R. Ghzouli, S. Dragule, A. Wasowski, T. Berger, and E. B. Johnsen. "Behavior Trees in Action: A Study of Robotics Applications." *The 13th edition of ACM SIGPLAN International Conference on Software Language Engineering (SLE) 2020.*

Research Contribution

Paper A: overview of robot programming languages, IDEs, and DSLs. From this study, it was observed that robot software consists of control software and robot mission software. The control software is programmed once and embedded into the robot hardware. Mission software helps in programming robot missions based on the user needs, which change from time to time. Robots can be programmed using open source IDEs, which support generic programming languages such as C/C++ and python. Programming missions are usually done using DSLs with abstraction support for robotic and end-user domains. Robotic IDEs offer facilities such as libraries and code editors.

Paper B: State-of-the-art—this paper extracts features from selected environments. The features are presented as a feature model to depict what a typical mission specification environment should have. The paper further demonstrates how these features can be used by end-users such as teachers, robot manufacturers, and language engineers.

Paper C: The paper extends the FLYAQ environment for use in the agriculture domain to spray and monitor crops on the farm. This demonstrates how the environment can be extended to various user domains. However, the extensions require qualified and skilled software and robotic engineers.

Contents

Chapter 1

Introduction

This chapter presents an overview of the licentiate thesis, covering the general introduction to the research area, background of the study, research questions, and the methodology.

1.1 Introduction

Autonomous robots are increasingly replacing humans in repetitive, laborious, or dangerous tasks and doing so often by interacting with humans, the environment, or other robots. According to H2020 Robotics Multi-Annual Roadmap (MAR),[1] inexpensive robots are becoming widely available, including ground robots, multicopters and robotic arms. Also, according to a 2019 press release[2] at the International Federation of Robotics, personal service robots are expected to exceed 61.1 million units in 2022, and sales for agricultural robots are projected to grow by 50% each year. To use robots in activities-of-everyday-life, they must be of general-purpose—can execute several tasks. This will allow robots to be deployed in a large variety of contexts, leading to the presence of robots in everyday life activities in many domains, including manufacturing, healthcare, agriculture, civil, and logistics.

Typically robots are programmed by manufacturers, robotic engineers, or software engineers. This involves programming a control system that defines how components of a robot system work as well as specifies missions to be executed by the robot. A robotic mission is a high-level behavior description of what the robot must perform [1]. As such, a mission coordinates the skills of robots which are translated to lower-level behaviors. Several approaches have been proposed for mission specifications. These can be grouped into two categories: formal and computation tree logic [1–7] and robotics-specific DSLs [8–13]. Formal and computation tree logic, such as Linear-Temporal Logic, require low-level and step-by-step descriptions of missions. Robotic specific domain-specific languages (DSLs) require more high-level mission specifications. The use of formal logic approaches is not only tedious to

[1] https://eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf

[2] https://ifr.org/ifr-press-releases/news/service-robots-global-sales-value-reaches-12.9-billion-usd

software developers with no background in formal logic but prohibitive to non-programmers [14, 15].

For robots to make better sense in activities-of-everyday-life, it is apparent that the end-users specify such missions. The end-users are experts in their domains, such as domestic workers, healthcare professions, teachers, and farmers. Involving end-users in mission specification will eliminate the need to reprogram a robot whenever a new mission to be executed is determined. It will also give flexibility in using general-purpose robots, with several actuators for executing various tasks. The end-users know how the tasks are executed and the outcomes of tasks. However, they may not necessarily know how to program a robot to execute such tasks on their behalf. Weintrop et al. [16] define end-user robot programming as writing programs for immediate and specific tasks, which the robot(s) can execute as opposed to writing general-purpose programs for others to modify.

Researchers and roboticists have invested substantial effort into achieving end-user-oriented programming environments for robots [13, 16–19]. The end-user programming systems are often either manual or automated. Manual end-user programming systems provide a textual or visual notation for which the end-user has total control over the programming instruction. The automated programming systems hide the programming language from the end-user, for instance, learning systems, gesture following robots, and programming by demonstration. This research focuses on manual programming systems, with a particular interest in visual programming systems, with DSLs for end-users. With full access to programming instructions in the DSLs, the end-users can achieve flexibility in specifying robot tasks for activities-of-everyday-life.

1.2 Background

This section provides a background to the main topics related to programming robots, such as programming languages and robot programming (Sect. 1.2.1), integrated development environments (Sect. 1.2.2) and DSLs (Sect. 1.2.3). The section ends with motivation for studying DSLs as the flexible alternative for specifying robot missions by end-users.

1.2.1 Robot programming

To ease developers in the development task, many middleware frameworks have been proposed. These middleware frameworks provide an abstraction layer between the hardware and application layers, largely hiding the inherent complexity and heterogeneity of robotics hardware. Furthermore, middleware frameworks typically support inter-robot communication. Some companies release middleware and software development kits (SDKs) (e.g., Choreographe[3]), but they tend to be robot-specific and therefore their functionality and flexibility is limited. Nevertheless, there exists a variety of middleware frameworks whose goal is to be robot agnostic. Examples of those middleware frameworks are the Robot Operating System (ROS) [20], ROS2 [21], the Open Robot Control

[3] http://doc.aldebaran.com/2-4/software/choregraphe/choregraphe_overview.html

Software (OROCOS) [22], Yet Another Robot Platform (Yarp) [23], and SmartSoft [24]. Among the mentioned robot-agnostic middleware frameworks, ROS is considered the de facto standard for robot application development, officially supporting more than 140 robots (including ground mobile robots, drones, cars, and humanoids) [25]. Examples of repositories from robotics companies that support the integration of ROS are the one from Kuka[4], Aldebaran and Softbank Robotics[5].

Using ROS, robotic applications are modeled as networks of nodes. Each node represents processes needed to perform specific functionalities, like controlling an actuator, parsing and publishing sensor data, or running a planning algorithm. ROS supports mainly Python and C++ programming languages as languages to write these nodes, but there are also existing libraries in Lisp, Java, and Lua.

Programming robots is broadly divided into two: the control software, written during robot manufacturing, and embedded into the robot, and end-user software for specifying missions which the robot executes. As the control software is written once, the end-user software may keep changing depending on the type of mission the robot should execute. Control software are usually written using hardware-description languages such as Verilog or VHDL to program the low-level electronics of robots [26, 27]. A commonly used microcontroller for robots is Arduino[6] [28]. While end-user software are written using DSLs as elaborated in Sect. 1.2.3.

1.2.2 Integrated Development Environments (IDEs)

To develop software applications, developers usually rely on integrated development environments (IDE), and roboticists are not an exception. However, there is no standardized IDE to develop robotic applications, but a variety of them. For instance, the official website of ROS lists all the IDEs that can be configured to work with ROS[7], allowing the building of ROS projects and code debugging. There exist in fact a development environment called ROS Development Studio[8]which integrates several ready-to-use tools as simulators and artificial intelligence (AI) based libraries. However, the environment is web-based and its use is not free. Working with general IDEs like Eclipse or Qt Creator seems to be the most popular option among roboticists, which brings to light the current lack of a free, robotics-centered IDE. The closest approach is the integration of plugins that exist for some of the most popular IDEs. They allow the integration with ROS, as is the case of Qt Creator[9].

Software for robotic applications is developed and configured through several binding times. Firstly, robotic functionalities (e.g., self-localization, planning, collision avoidance) are programmed at the development stage. Robotic applications comprise a network of those functionalities and their interfaces. Robotic functionalities realized as software modules (i.e., software

[4]https://wiki.ros.org/kuka
[5]https://wiki.ros.org/Aldebaran
[6]`https://www.arduino.cc`
[7]https://wiki.ros.org/IDEs
[8]https://www.theconstructsim.com/rds-ros-development-studio/
[9]https://ros-qtc-plugin.readthedocs.io/en/latest/

components) are then integrated with other modules and deployed into or associated with specific robotic entities. If a robotic application comprises several robots working as a team, the robotic entities will need to communicate among them. Nevertheless, interfacing and inter-robot communication are typically supported by middleware frameworks like ROS or OROCOS, as previously explained. Once integrated, the robotic application typically goes through several iterations of testing and validation, which is usually accomplished via simulation as a first step before executing on real a robot. Developing applications for Arduino microcontrollers can be done using IDEs10, which support general purpose languages (GPLs) such as C/C++, Java and Python.

1.2.3 DSLs

In robotics, there exist robot engineers and the end-users. The robot engineering domain deals with manufacturing and developing tools for programming the robots. The end-user domain categorizes the users with the kind of tasks robots can execute. Specification of a robot mission can be influenced by controls from the end-user and the programming languages. Missions specified using dedicated DSLs are easy to comprehend by end-users. Such DSLs abstract the underlying mechatronics and logic involved in the formal description of the missions.

Many internal and external DSLs have been developed to facilitate robot programming by end-users. Internal DSLs are extensions of a general-purpose language—often called host language, which are embedded into a host language, hence, using the host language's syntax and infrastructure. Examples of internal DSLs include: BehaviorTree.CPP [29], ROS Behavior Tree [30] and SMACH [31]. An external DSL on the other hand is a language with independent syntax, semantics, and other related language resources designed with notation and abstractions suitable to the user domain. Examples of external DSLs for robot programming include: NaoText [10], The Unreal Engine 4 (UE4) Behavior Tree [32] and FLYAQ [33–36].

Little is known about the features such DSLs provide and the user domains in which they are applicable. It is also not clear what the expectations of the end-users on such DSLs. It is necessary to understand areas of improvement to meet the expectations of end-users and robot engineers.

1.3 The Research Focus

Programming is a challenge to most computing scientist, including roboticists [37]. The challenge is daunting for novice programmers. The novice programmers can be computing students or end-user domain experts who use programmable systems. An example of an end-user interested in programming a robot can be a farmer who wishes to use robots to plant, weed, spray crops, or harvest farm produces. Graphical programming has demonstrated some potential in mitigating the challenge of programming among novice programmers.

Robots have become increasingly present in activities of everyday life.

For robots to meet the needs of everyday life, they must be flexible both physiologically and software to program execute the tasks. physiological—mechatronic flexibility includes the ability to physically reconfigure the robot to do a diverse category of tasks by plugging in various types of actuators on the same robot.

The software—programs for operating such robots called robot missions must be flexible. Traditionally robot missions are hard programmed by software engineers and roboticists. This is not feasible for activities of everyday life since the missions keep changing. It is important that end-users specify these missions, based on the current need without necessarily involving software engineers.

This research established state-of-the-art in end-user robot programming by surveying visual mission specification environments and DSLs for specifying robot missions. In the second phase, the research looks forward to investigating: (a) introduction to programming for early programmers in universities in Uganda (b) possibility of teaching robot programming to novice programmers in low resource universities in Uganda. Low resource in this context refers to inadequate laboratory facilities including robots.

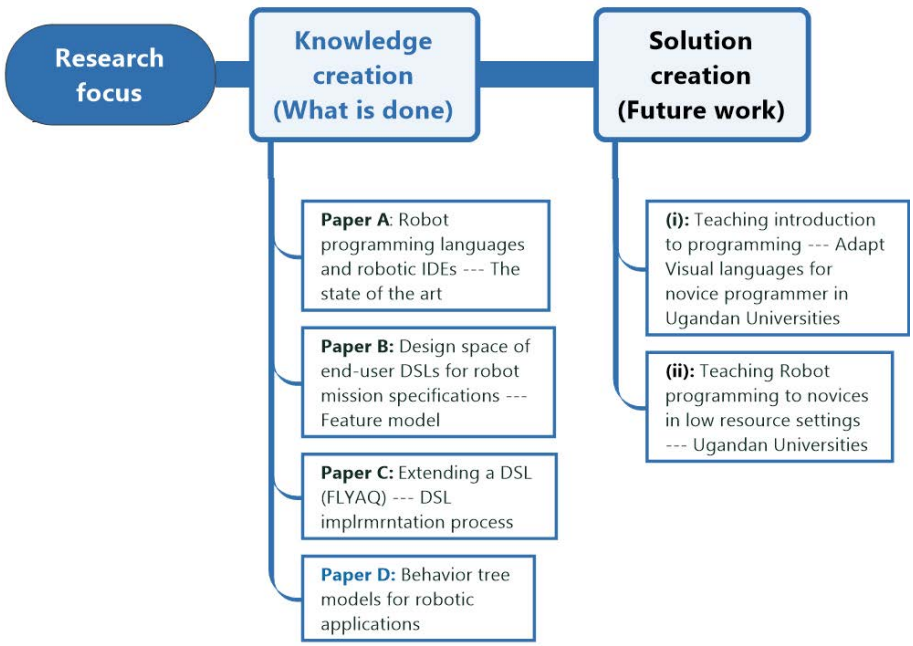


Figure 1.1: The research focus comprises of knowledge creation — what is already done and the solution creation — future works

The papers A, B, and C have shown the key features in DSLs for end-user programming. The results show that most DSLs for end-user programming are used in education for teaching early programmers. There is also a need to develop libraries to abstract mission primitives in order to facilitate end-user programming in the service domains such as hospitality, agriculture, and

healthcare. In Uganda, most students joining universities have never had any programming experience, let alone robot programming. Introducing programming to such students using end-user programming environments will aid in easy comprehension of programming. In future works, it is apparent to understand what is required to adapt the end-user environments to teach novice programmers in universities in Uganda, where laboratory resources are scarce.

1.3.1 Research Questions

The qualitative research methodology was used to investigate robot programming languages—mission specification DSLs. Then design science methodology helped in prototyping the recommendations for DSL design and implementation. During qualitative methodology, we used observation and content analysis methods to extract features of selected environments. The design science methodology was used to extend an existing tool.

This work answers the following research question.

R.Q 1. What languages and tools exist for end-user robot programming? The question explores the existing languages and tools being used for programming robots by end-users. This question is motivated by the fact that robots are typically pre-programmed. However, currently, there is an enormous effort to supporting end-users in programming robot missions. There is a need to understand the kind of languages and tools used for programming robots by end-users. This question was answered by paper A by writing a book chapter on languages for specifying missions of robotic applications, which elaborates on existing programming languages for programming robots and DSLs for mission specifications. Paper B critically explored the existing mission specification environments and their futures by answering the following sub-questions. ***R.Q 1.1*** *what visual, end-user-oriented mission specification environments have been presented for mobile robots?* These environments were selected from three data sources—environments based on authors' experience, google search, and snowballing.

R.Q 1.2 *what is the design space for in terms of common and variable characteristics (features) that distinguish the environments?* Extraction of the features was done by installing and running sample missions in the environments, reading publications on the environments, studying user manuals, and visiting the environments' websites. The environments were distributed among the authors, and after initial extraction of features, authors verified what the colleagues extracted to harmonize extracted data.

The following research questions are intended for full PhD work.

R.Q 2. How effective are the identified languages and tools for end-user programming of robots?

We plan to conduct a user study to get feedback from language engineers, researchers, and end-users to get feedback on the features extracted in RQ.1 and generate requirements from all the user segments. This study will inform the recommendations for the design and implementation of future mission specification environments.

R.Q 3. How can the languages and tools be improved to offer better support to end-users while programming robots for activities of everyday life?

This question will be answered using the design science method to review and extend a visual DSL for novice programming. We shall first review the introduction to programming course in Ugandan universities and match the concepts in the course with the concepts presented by existing Visual DSLs. Then Identify an appropriate DSL, which can be extended to capture all the concepts required for teaching introduction to programming and robot programming. This will involve reviewing the DSL to identify the missing concepts, in order to extend it for robot programming.

1.3.2 Methodology

In this research, we used empirical research methods to create knowledge and we plan to use the design science method to create the solution, a popular approach in software engineering research [38]. During the empirical studies, data were collected using techniques such as snowballing, web search, experimentation, literature review, and author experience.

Paper A extracted data both from related literature and experimentation from software —PROMISE DSLto analyze the state-of-the-art in robot programming languages. For paper B, the data from each data source was studies, and relevant data extracted in a feature matrix table, which is used to design a feature model. Paper C used the design science approach to evaluate an existing tool FLYAQ and extend with new features, which helped in understanding the design process of DSL.

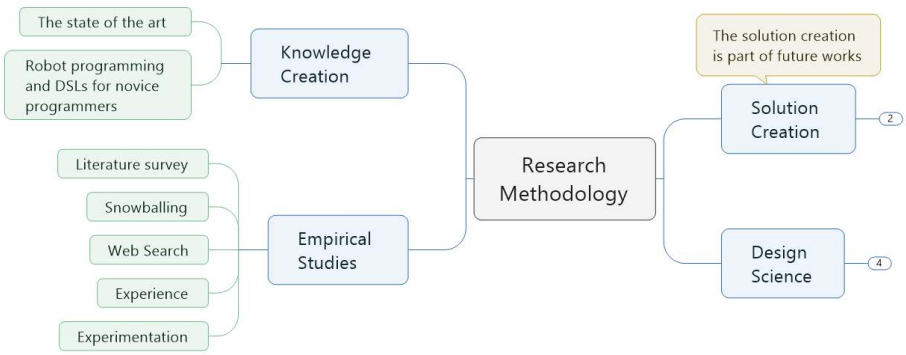


Figure 1.2: The research methodology used for knowledge creation. The solution creation is part of the future works.

1.4 Results

1.4.1 Summary of Papers

The section presents the summaries of the three papers, which form the basis of this licentiate thesis.

1.4.1.1 Paper A

Overview of robot programming languages, IDEs and DSLs In this work, we present integrated development environments for developing robotic applications and DSLs. The DSLs have been profiled as internal and external DSLs. We further demonstrated how the PROMISE DSL works. PROMISE uses mission patterns as mission primitives to specify and execute multi-robot missions.

Robots are complex cyber-physical and safety-critical systems whose programming requires developers with diverse backgrounds, ranging from logic, mechatronics to end-users. The usage of robots can be as a single robot or team of homogeneous or heterogeneous robots in natural environments such as hotels and farms. It becomes necessary that the specification of missions for such robots should be done by end-users, with tools that are easy to use yet accurate.

From the DSLs identified for mission specification, it is apparent that more needs to be done to facilitate mission specifications by end-users. There is a need for tools to support mission reuse and managing the emerging situation in natural environments during mission execution. The paper suggests future research direction in mission re-usability, managing variability in the real world, automatic fleet mission specification for multi-robot missions, and human-robot collaboration.

1.4.1.2 Paper B

Design space of end-user oriented languages for specifying robotic missions. Mobile robots are becoming increasingly important, and they are promising instruments for fulfilling missions that support our everyday life. Simplicity and flexibility in programming such robots are necessary since end-users come from diverse domains, not necessarily with sufficient programming knowledge.

This paper presents an empirical understanding of the current state-of-the-art of end-user-facing languages and their environments used for programming robots. 30 environments were selected and analyzed through a rigorous process, from which a feature model was generated. An evaluation of how the environments can be used by end-users based on their respective features was done. Most of the environments are Blockly based and mostly used in education. There is a need to do more so that visual programming can be used in other technical domains such as agriculture, hospitality, and public order management.

1.4.1.3 Paper C

AGenerated Property Specification Language for Resilient Multirobot Missions As robots become more present in everyday life, robotic missions must be domain-specific, resilient, and collaborative. This paper presents a high-level mission specification language in which users declaratively specify domain constraints as properties of the mission. The paper demonstrates how the DSL can be generated automatically using the ProMoBox approach from the mission language of FLYAQ.

FLYAQ is an extensible platform designed for non-expert end-user for the specification of robotic missions for a team of robots. The platform consists of monitoring mission language, behavior language, and robot language. In this work, the monitoring mission language has been extended to automatically generate the mission specification language (MSL) for specifying missions in commercial, emergency, and agriculture domains. During specification, the user defines the mission goals and constraints at run time. With MSL, a mission is specified by plotting an area on the map and defining a *DetectPest* and *Spray* task in this area using the MSL mission layer, which is extended, with language concepts from agriculture. As future works, there is a need to map the MSL to behavior language, which describes the individual robot movements and actions.

1.4.2 Discussion

The main contribution of this research is knowledge creation on the state-of-the-art in DSLs for mission specifications. The empirical knowledge provides decision support on the use of existing DSLs and the design of future DSLs for end-user domains in robotics.

In order to specify a robot mission, it is important to consider: the range of tasks to be executed, the end-users of the robots, number of robots involved, the physical environment in which the mission will be executed, and the programming language for specifying the mission. The robotics IDEs provide programming environments with a number of programming languages including visual programming languages, hardware abstractions and end-user domain abstractions. Programming robots for activities-of-everyday-life by end-user domains remains a challenge, especially in technical domains such as healthcare, commerce, and agriculture. In these domains, robot missions are preprogrammed, in most cases using textual notation. The end-user is only provided with execution instructions.

Visual DSLs for robot programming use underlying formalisms such as state machines, flowcharts, and behaviour trees [30, 84, 85]. Most of the visual programming environments focus on education—supporting novice programmers to learn how to program and boost critical thinking through STEM support. The existing DSLs need to be abstracted further to fit specific user domains such as healthcare and agriculture if experts from such domains are to use the languages. Software engineers and user domain experts need to collaboratively develop such DSLs for specifying missions used in activities-of-everyday-life. This will ensure that the mission primitives are actual user domain terms, which are easily understood by the domain experts. The research community needs to develop library of robot skills as modules in order to facilitate robot function abstractions at higher levels. This will makes mission primitives easy to understand by end-users. More research needs to be done to profile the user domains in order to facilitate user domain modeling by the language engineers.

Most vendors of the visual DSLs implement the languages' syntax using Blockly and Scratch. The DSLs implement language concepts, robotic domain concepts, and end-user domain concepts. The core language concepts commonly implemented are control flows, logic, and variable. Concepts such

as movement, sensing, and communication are implemented in the robotics domain. While in the end-user domains, concepts such as take picture, recognize humans and objects, pick and place objects, video streaming, and reception services are commonly implemented. However, the potential of Blockly and Scratch libraries to implement advanced language capabilities such as language compositions is lacking.

Some core aspects in robotic missions are mission composition mechanism and intelligence. Robotic mission compositions are classified into vertical and horizontal decomposition. Horizontal (de-)composition refers to putting lower-level functionality (e.g., tasks) into a respective execution order (e.g., sequence or loop), while vertical (de-)composition refers to refining functionality needed to realize that functionality. Intelligence is the ability of robots to act automatically without human intervention. Intelligence can be triggered by events from the environment as captured by sensors, timed executions, or learning from past experience.

Working with general IDEs, such as Eclipse or Qt Creator, appears to be the most popular option among roboticists, despite the existence of a few free robotic-centered IDEs. For many IDEs, there are instructions for configuring towards robotics. For instance, the ROS community provides configurations for several IDEs including Eclipse, Netbeans, KDevelop, Emacs, and RoboWare studio, a variant of Microsoft Visual Studio. Similar to general-purpose IDEs, Robotics IDEs offer facilities for robotics software engineering, including code editors, robotics libraries, build tools, and quality-assurance tools (i.e., debuggers, test environments, and simulators).

1.5 Conclusion and Future Works

Conclusion. There is an increasing presence of robots in activities of everyday life. It is infeasible to have robots pre-programmed by manufacturers, since the robots are capable of doing several tasks that should be flexibly programmed. The research community has recently given remarkable attention to robot programming, with a particular interest in domain-specific languages for mission specification, description of robot missions in natural languages, and end-user-facing visual languages for robot mission specifications. These efforts have led to more novice programmers participating in robot programming.

However, more effort is required to improve end-user-facing DSLs for mission specification in activities-of-everyday-life. Some of the promising domains to use robots in everyday life include healthcare, hospitality, education, and agriculture. In order to specify complex missions by end-users, researchers and practitioners need to develop domain specific abstractions and libraries for expressing mission primitives. More DSLs are required to facilitate mission specifications by end-user with detailed domain primitives. From the survey on end-user-oriented languages for mission specifications, we observed that most of the visual DSLs are used in the education domain to teach (a) introduction to programming languages, (b) introduction to robot programming and (c)STEM subjects.

Future Works. This research intends to contribute in the education domain by further examining what is required to use visual environments for teaching

programming. This can be achieved by answering the following questions: (a) How visual programming can be used to introduce robot programming to domain experts? In order to answer the question, we intend to explore the following questions: What teaching concepts are covered in introductory courses to robot programming? Which of these concepts are not covered by existing visual programming DSLs. How can one of the visual DSLs be enhanced, using the extension mechanism to incorporate the missing concepts. This will make the DSL suitable for teaching introduction to programming languages.

Once programming concepts are understood, robot programming can then be introduced by exploring the following questions. What are the robot specific concepts that must be taught? Does the DSL cover all of them? If not, then extend the DSL to make it suitable for the introductory course.

At the end of the research, concrete steps, concepts and recommendations must be derived to teach introduction to programming and robot programming for novice programmers.

Chapter 2

Paper A

Languages for specifying missions of robotic applications.

S. Dragule, S. G. Gonzalo, T. Berger, and P. Pelliccione.

In submission to Robosoft.

Abstract

Robot-application development is gaining increasing attention both from research and industry. Robots are complex cyber-physical and safety-critical systems with various dimensions of heterogeneity and variability; they typically integrate modules conceived by developers with different backgrounds. Programming robotic applications typically requires programming, mathematical or robotic expertise from end-users. In the near future, multipurpose robots will be used in the tasks of everyday life in environments such as our houses, hotels, airports, museums, etc. It would be then necessary to democratize the specification of missions that robots should accomplish. In other words, the specification of missions of robotic applications should be performed via easy-to-use and accessible ways, and, at the same time, it should be accurate, unambiguous, and precise. This chapter presents domain-Specific Languages (DSLs) for robot-mission specification, by profiling them as internal or external, and by giving also an overview of their tooling support. The types of robots supported by the respective languages and tools are mostly service mobile robots, including ground and flying types.

2.1 Introduction

Inexpensive and reliable robot hardware—including ground robots, multicopters, and robotic arms—is becoming widely available, according to the H2020 Robotics Multi-Annual Roadmap (MAR).[1] As such, robots will soon be deployed in a large variety of contexts, leading to the presence of robots in everyday life activities in many domains, including manufacturing, healthcare, agriculture, civil, and logistics.

Robots are complex cyber-physical and safety-critical systems, which challenges engineering their software [39, 40]. In addition, the robotics domain is divided into a large variety of sub-domains, including vertical ones (e.g. drivers, planning, navigation) and horizontal ones (e.g. defence, healthcare, logistics), with a vast amount of variability [41, 42], further complicating robotics software engineering. Due to this heterogeneity, a robot typically integrates modules conceived by developers with different backgrounds. For instance, electrical engineers design the robot's hardware, control engineers develop planning and control algorithms, and software engineers architect and quality-assure the software system. Coordinating the integration of all these modules from developers with different backgrounds is one of the major challenges that characterize the domain of robotics [39, 40]. Further challenges comprise identifying stable requirements, defining abstract models to cope with hardware and software heterogeneity, seamlessly transitioning from prototype testing and debugging to real systems, and deploying robotic applications in real-world environments.

A core activity when engineering robotics software is defining and implementing the robot's behaviour. Specifically, in addition to building and integrating modules that define the lower-level behaviour, the overall behaviour of robots needs to be defined. This behaviour, often called a *mission*, coordinates the lower-level behaviours that are typically defined in modules realizing the different skills. While this coordination has traditionally been implemented in plain code [43], this will not be feasible in the near future, when multipurpose robots will be used in our houses, hotels, hospitals, and so on, to accomplish tasks of the everyday life. For these reasons, the use of dedicated (domain-specific) languages is becoming increasingly popular [44–46]. These languages target end-users without robotic, ICT or mathematical expertise, and allow them to conveniently command and control robots. This trend is also expressed by the MAR roadmap, given the increasing involvement of robots in our society, especially service robots (i.e., robots that perform useful tasks for humans excluding industrial automation applications[2]). In fact, the MAR roadmap describes DSLs [46–48], together with model-driven engineering [49–53], as core technologies required to achieve a separation of roles in the robotics domain while also improving, among others, modularity and system integration.

The specification of mission ranges from (i) very intricate and difficult-to-use [2, 3] logical languages, such as Linear-Temporal Logic or Computation Tree Logic [1, 4–7], whose instances are directly fed into planners; via (ii) common notations for specifying behaviour, such as Petri nets [54, 55] and state machines [31, 56, 57], which require low-level and step-by-step descriptions

[1] https://eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf

[2] https://www.iso.org/standard/55890.html

of missions; to (iii) robotics-specific DSLs tailored to the robot at hand [8–13], which often allow a more high-level mission specification.

This chapter contributes to the state of the art in mission specifications for robots. We present an overview of programming languages for robotic applications and respective IDEs (integrated development environments) in Sect. 2.2. Thereafter, we present DSLs for mission specification in Sect. 2.3, including internal and external DSLs, together with their tooling. In Sect. 2.4, we discuss how robots are usable in everyday life, with specific reference to the PROMISE tool for specifying missions for multi-robot applications. We put PROMISE into practice by describing a real mission with PROMISE we realized, together with the rest of the robotic software, including a multi-layer architecture. We conclude and discuss areas for future work in Sect. 2.6.

2.2 Programming Languages and IDEs for Robotic Applications

The software of a robotic application can be conceptually organized into two main parts: (i) the software controlling the various modules (written once and embedded into the robot) and (ii) the software that permits the specification and execution of the mission (potentially changing from mission to mission, especially for multipurpose robots). Traditionally, these two parts are mixed for robots capable of doing specific tasks, where the mission specification only involves setting some parameters that are specific for the environment in which the mission will be executed. In this section, we briefly describe programming languages (Sect. 2.2.1) and IDEs (Sect. 2.2.2) used in robotics.

2.2.1 Programming Languages for Robotic Applications

Many different languages are used for the development of mobile robotic applications. Starting from the lowest level of abstraction, hardware-description languages (e.g. Verilog or VHDL) are mainly used by electronic engineers to "program" the low-level electronics of robots [26]. Hardware-description languages are commonly used to program field-programmable gate arrays [27], which are devices that make it possible to develop electronic hardware without having to produce a silicon chip.
At the level of microcontrollers, a widely used option is Arduino[3] [28]. It is an open-source electronics platform that consists of a board with assembled sensors (and potentially actuators) that can be controlled using specific software. Software for Arduino-based applications may be developed using an open-source IDE[4], which supports the languages C and C++, applying a wrapper around programs written in these languages and using special rules of code structuring. The hardware manufacturers typically also provide proprietary software, such as RAPID[5] technical reference manual from ABB

[3] https://www.arduino.cc

[4] https://arduino.en.softonic.com

[5] https://library.e.abb.com/public/688894b98123f87bc1257cc50044e809/Technical\%20reference\%20manual_RAPID_3HAC16581-1_revJ_en.pdf

and KRL[6] reference guide from Kuka.

More powerful machines in terms of computation—including single-board computer solutions such as Raspberry Pi—support Ubuntu distributions and, therefore, the Robot Operating System (ROS) [58]. ROS [20] is an open-source middleware offering a framework for structured communication among various robotic components using a peer-to-peer connection. ROS currently runs on Unix-based platforms, and the software for ROS is primarily tested on Ubuntu. Therefore, a typical setup for a roboticist includes a certain version of Ubuntu[7] with a certain distribution of ROS.[8]

Most packages and libraries of ROS are developed using either C++ or Python so those languages are the most commonly used. However, ROS's communication system is language-agnostic, which enables several languages such as C++, Python, Octave, Java, and LISP to be used depending on the user's proficiency. ROS also offers modularized tool-based microkernel design to aggregate various tools performing specific tasks such as navigating source code tree, get and set configuration parameters, and visualize the peer-to-peer connection topology, among others [59, 60].
ROS has evolved with a number of distributions, supporting more than 20 robotic systems[9], including drones, arm robots, humanoids, and wheeled mobile-base robots. Among the robot-agnostic middleware, ROS is considered the de facto standard for robot application development [39], officially supporting more than 140 robots (including ground mobile robots, drones, cars, and humanoids) [25]. Examples of repositories from robotics companies that support the integration of ROS are the one from Kuka[10] or from Aldebaran and Softbank Robotics.[11]

MATLAB (and its open-source relatives, such as Octave) is a popular option among engineers for analysing data and developing control systems. It has also been used for robotics software development [61], and there even exists a robotics-dedicated toolbox.[12] The toolbox contains tools that support functionalities ranging from producing advanced graphs to implementing control systems.

Machine learning is another technique applied in the context of robotics, as is being used in decision making and image recognition. Machine-learning models are first trained using platforms such as TensorFlow or PyTorch and then implemented as ROS nodes [62]. These training platforms provide dedicated APIs, and they are commonly Python or C++-based. Finally, image processing is a key functionality in robotics, and the most used library in this domain is OpenCV[13] [63], written in C++. Its primary interface is written in and uses C++, but there are bindings for Python, Java, and MATLAB.

[6] http://robot.zaab.org/wp-content/uploads/2014/04/KRL-Reference-Guide-v4_1.pdf
[7] https://wiki.ubuntu.com/Releases
[8] https://wiki.ros.org/Distributions
[9] http://wiki.ros.org/Distributions
[10] https://wiki.ros.org/kuka
[11] https://wiki.ros.org/Aldebaran
[12] https://www.mathworks.com/products/robotics.html
[13] https://opencv.org

2.2.2 IDEs for Developing Robotic Applications

IDEs aid software engineering by providing editing, compilation, interpretation, debugging, and related automation facilities. They often come with version-control, refactoring, visual-programming, and multi-language support. The usage of IDEs improves efficiency in software development and makes it less error-prone.
Working with general IDEs, such as Eclipse or Qt Creator, appears to be the most popular option among roboticists, despite the existence of a few free robotic-centred IDEs. For many IDEs, there are instructions for configuring towards robotics. For instance, the ROS community provides configurations for several IDEs, including Eclipse, Netbeans, KDevelop, Emacs, and RoboWare studio, a variant of Microsoft Visual Studio.

Eclipse, in particular with its tooling for model-driven software engineering (e.g. Eclipse Modeling Framework), has been used to realize DSLs and respective environments for building robotics applications in a model-driven way. For instance, Arias et al. [64] offer a complete robotics toolset upon Eclipse to support the engineering from design to code generation, called the ORCCAD model.

Similar to general-purpose IDEs, Robotics IDEs offer facilities for robotics software engineering, including code editors, robotics libraries, build tools, and quality-assurance tools (i.e. debuggers, test environments, and simulators). As opposed to general IDEs, robotics IDEs primarily target building robotic applications, without support for other domains.

Table 2.1 summarizes IDEs for developing robotic applications with details on target users, languages supported, and features that go beyond a general IDE. To illustrate one of the IDEs, Fig. 2.1 shows a screenshot of the Robot Mesh Studio. The user interface is separated into three main panes. Pane A shows a description of the current mission—richtext entered by the developer to describe and illustrate the mission (here, the visual recognition and lifting of an object by the robot). Pane B shows the actual mission expressed in an external DSL (with Blockly syntax) provided by the IDE, or alternatively the generated textual code. Pane C shows help text, or alternatively the interactive debugger or an overview on the current robot configuration.

2.3 Robot Mission Specification

As robots become an integral part of the everyday life, we need better ways to instruct robots on the tasks they should accomplish. Mission specification is a process that relies on a strategy and mechanism that determines the steps a robot takes when performing a given task [8, 33, 35, 83].

The specification of a robot mission is influenced by the range of tasks the robot can execute, the end-user of the robot, the number of robots involved, the physical environment in which the mission will be executed, and the programming languages provided by the robot manufacturer. Robots performing a specific task are normally pre-programmed by manufacturers, while those with the ability to do a number of tasks require frequent change of what they do depending on the need at a given time—calling for flexible ways of specifying missions.

Table 2.1: List of Dedicated Robotic IDEs.

Name	IDE details
RobotC [65,66]	C-based educational environment providing two notations, RobotC Graphical and RobotC Natural Language e.g. Listing 2.4
Robot Mesh Studio [67]	IDE for programming educational robots from Arduino, PICAXE, Parallax, and Raspberry Pi microcontrollers. It offers two graphical DSLs: Flowol, a flowchart-based language, and a Blockly-based language. Textual languages: C++, and Python
VEX Coding Studio [68, 69]	A robot vendor's environment for programming educational robot kits. The IDE offers Scratch-based syntax (VEXcode Blocks) and a text-based syntax (VEXcode Text)
PICAXE [70,71]	For programming educational PICAXE microcontroller-based robots. It offers the PICAXE language in three syntaxes: PICAXE BASIC–textual, PICAXE Blockly–graphical, and PICAXE Flowchart syntax
ROS Development Studio [72]	An online IDE with ready-to-use tools, such as simulators and AI-based libraries. The ROS Development Studio supports all robots compatible with ROS and a variety of languages, such as C++, Python, Java, MATLAB, and Lisp
Microsoft Robotic Developer Studio (MRDS) [73]	Microsoft product for hobbyist, academic, and commercial robot application developers. The IDE supports programming robot applications in Microsoft's Visual Programming Language (MVPL) and C#
MATLAB and Simulink [74]	IDE offers hardware-agnostic robot control for Arduino and Raspberry Pi microcontrollers, that can be connected to ROS and ROS2. Code from a variety of embedded hardware, such as Field Programmable Gate Arrays (FPGAs), Programmable Logic Controllers (PLCs), and Graphics Processing Units (GPUs), can be generated to various target languages including C/C++, VHDL/Verilog, Structured Text, the PLC language, and Compute Unified Device Architecture (CUDA) language
Webots [75]	An open-source, online IDE simulator that supports a number of robots and a range of languages such as C, C++, Python, Java, MATLAB, and ROS-supported languages C, C++, Python, Java, and MATLAB
Robot Task Commander (RTC) [76]	The IDE is meant for automated task planning for robot(s) using one or more computing devices over a network. It supports humanoid robots programmed using Python scripting language and RTC visual programming language
The SmartMDSD Toolchain [77]	IDE for developing robot systems by providing building blocks that can be used for composing new systems from existing components. The IDE applies modelling techniques using tools such as Xtext, Xtend, and Sirius from Eclipse
BRICS Integrated Development Environment (BRIDE) [78]	IDE for developing editors in robotics based on model-driven engineering principles. BRIDE incorporates the OROCOS and ROS frameworks. The ROS version offers features such as graphical modelling of ROS nodes, code generation in C++ or Python, and generation of launch files
Universal Robotic Body Interface (URBI) [79]	Open-source IDE for programming robot controls, using client-server architecture. The server manages low-level hardware controls for sensors, camera, and speakers, and the client sends high-level behaviour commands like "walk" to the server. Languages supported include C++, Urbiscript scripting language, MATLAB, Java, and Lisp
TeamBots [60,80]	A Java-based environment for developing and executing control systems on teams of robots and on simulation using the application TBSim. The IDE provides a set of applications and packages for multi-agent mobile robots.
Pyro [81]	An educational IDE that abstracts low-level details, making it suitable for students learning to program robots using the C++, Java and Python. Pyro wraps Player/Stage and ARIA, for easy access to its users.
CopeliaSim (VREP) [82]	A multi-robot IDE, which uses distributed control architecture to model objects through: embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution. The IDE supports programming using C/C++, Python, Java, Lua, Matlab or Octave

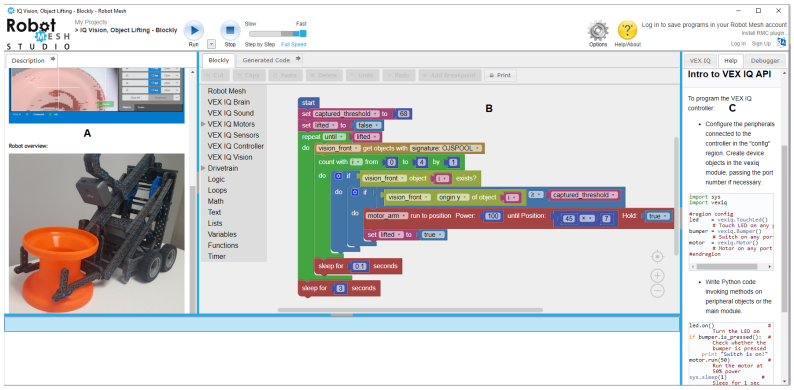


Figure 2.1: Screenshot of the Robot Mesh Studio IDE [67] with three panes. Pane A provides a rich-text description of the mission, Pane B showsthe actual mission expressed in a DSL with Blockly syntax, and Pane C shows help text or alternatively the debugger or an overview on the current robot configuration.

Any mission specified using a DSL should be easily understood by experts in that domain—e.g. logistics, commerce, and health. DSLs are recognized for their ability to abstract low-level details of robotic implementations and allowing users to specify their concerns from higher levels by using common terms in the domain. This abstraction further enhances effective communication of concepts with the domain experts. Due to these reasons, DSLs have been studied and proposed for mission specification by the community [8–13].

DSLs typically work based on the underlying formalisms such as state machines, flowcharts, and behaviour trees [30, 84, 85]. We assume that the reader has already some knowledge on state machines and flowcharts. Before presenting the selected DSLs, we give a brief introduction to behaviour trees, which are less widely known.

A behaviour tree is a hierarchical model in which the nodes of the tree are tasks to be executed [30,84,85]. Behaviour trees emphasize modularity, coupled with two-way control transfer using function calls, unlike one-way (transitions) in finite-state machines. The modular character in behaviour trees makes the reuse of behaviour primitives feasible. Behaviour trees have been applied in computer science, robotics, control systems, and video games.[14] Behaviour trees consist of control-flow nodes (namely *Parallel*, *Fallback*, *Decorator*, and *Sequence*) and executor nodes (i.e., *Action* and *Condition*). An action node executes a task and returns success or failure, while the condition node tests if a certain condition is met.

In the following subsections, we describe a selection of internal and external DSLs for mission specification together with examples. Internal DSLs are extensions of a general-purpose (i.e. programming) language—often called host language. An external DSL is a language with independent syntax, semantics, and other related language resources and designed with notation and abstractions suitable to the user domain.Table 2.2 shows an overview of

[14] `http://wiki.ros.org/behavior_tree`

Table 2.2: List of DSLs, their notation used and their styles (internal or external DSL)

Name of DSL	Notation	Style
Choregraphe	Visual	External
NaoText	Textual	External
Microsoft Visual Programming Language	Visual	External
EasyC	Textual, Visual	External
SMACH	Textual	Internal
Open Roberta	Visual	External
FLYAQ	Visual	External
Aseba	Textual, Visual	External
LEGO Mindstorms EV3	Visual	External
MissionLab	Visual	External
CABSL	Textual	Internal
BehaviorTree.CPP	Textual	External
ROS Behavior Tree	Textual	Internal
Unreal Engine 4 Behavior Trees	Textual	External
PROMISE	Textual, Visual	External

these DSLs with the notations supported (visual or textual) and style (internal or external DSL).

2.3.1 Internal DSLs

Internal DSLs follow the host language's syntax, and their execution is limited to the host language's infrastructure. They provide features specific to given end-user domains, such as robotics engineering, which simplify specification of domain user's concerns. In each of the following internal DSLs, we look into the host language, the developing organization (company), its semantics (compiled/interpreted), features specific to the internal DSL, and the end-user domain the language is targeting.

2.3.1.1 ROS Behavior Tree

ROS Behavior Tree [30] is an open-source C++ library for creating behaviour trees. The DSL's aim is to be used by expert robot developers, who are conversant with the ROS framework and C++ or Python languages. Listing 2.1 shows sample code for creating a behaviour tree.[15] It consists of header files and demonstrates how the action node and the condition node are executed in the behaviour tree.

[15] `https://github.com/miccol/ROS-Behavior-Tree`

```
#include<actions/actiontestnode.h>
#include<conditions/conditiontestnode.h>
#include<behaviortree.h>
#include<iostream>
int main(int argc, char **argv){
  ros::init(argc, argv, "BehaviorTree ");
  try{
    int Tick Period_millisecond s = 1000;
    BT::ActionTestNode *action1 = new BT::ActionTestNode ("Action1");
    BT::ConditionTestNode *condition1 = new BT::ConditionTestNode ("Condition1");
    action1->set_time(5);
    BT::SequenceNodeWithMemory* sequence1= new BT::SequenceNodeWithMemory("seq1"
      );

    condition1->set_boolean_value(true);
    sequence1->AddChild(condition1);
    sequence1->AddChild(action1);
    Execute(sequence1, Tick Period_milliseconds);
  } catch (BT::BehaviorTreeException& Exception){
    std::cout<<Exception.what()<<std::endl;
  }
  return 0;
}

```

Listing 2.1: Creation of a new behavior tree using the ROS Behavior Tree DSL.

Selector nodes are used to find and execute the first child that does not fail. A selector node immediately returns success or running when one of its children returns success or running. Sequence nodes are used to find and execute the first child that has not yet succeeded. A sequence node returns failure or running when one of its children returns failure or running. The parallel node ticks its children in parallel and returns success if M ≤ N children return success, it returns failure if N - M + 1 children return failure, and it returns running otherwise. The decorator node manipulates the return status of its child according to the policy defined by the user. Decorator Retry retries the execution of a node if this fails; and Decorator Negation inverts the Success/Failure outcome.

2.3.1.2 SMACH

SMACH [31] is a non-commercial application programming interface written in Python, based on hierarchical concurrent state machines. It allows executions to be controlled by a higher-level task-planning system.

The library enables a quick way to create robust robot missions with maintainable and modular code. The DSL provides integration with ROS for developing robot applications using state machines. The actionlib library in SMACH provides an interface for tasks such as moving the base to a target location, performing a laser scan and returning the resulting point cloud, and detecting the handle of a door. SMACH Viewer is a graphical interface that shows a hierarchy of state machines, transitions between states, active states, and data passed between states. Once a state machine for a given mission is created, it is executed in the ROS environment.

Listing 2.2 demonstrates how to create a state machine, adding states to the state machine. In the state execution (line 10), "event" depicts the condition to

execute outcome1 if true, outcome2 otherwise.

```
#!/usr/bin/env python
import rospy
import smach
# creating a state
class Foo(smach.State):
def __init__(self, outcomes=['outcome1', 'outcome2']):
# Your state initialization goes here
def execute(self, userdata):
# Your state execution goes here
if event:
return 'outcome1'
else:
return 'outcome2'
# Adding states
sm = smach.StateMachine(outcomes=['outcome4','outcome5'])
with sm:
smach.StateMachine.add('FOO', Foo(),
transitions={'outcome1':'BAR',
 'outcome2':'outcome4'})
smach.StateMachine.add('BAR', Bar(),
transitions={'outcome2':'FOO'})
```

Listing 2.2: Creation of a state and adding the state to a state machine in SMACH

2.3.1.3 C-Based Agent Behavior Specification Language

The C-based agent behavior specification language (CABSL) [86] enables the description of robot behaviours as a hierarchy of finite state machines. The control program executes behaviours based on the acquired sensor data, which maps the sensor data to actions the robot executes. In a state, when an action is taken, either the state generates an output or calls another state machine. Otherwise, there is a transition from one state to another state.

An active graph in CABSL is a tree consisting of a set of state machines being executed. Each state machine can call any other state machine. The language is implemented and compiled in C++.[16] CABSL does not provide the functionality of replacing the behaviour on the fly in case the acquired sensor data requires a change in behaviour.

The DSL's textual notation makes it suitable for developers with experience in using the C language. It has reportedly been used for the NAO robot.

2.3.2 External DSLs

External DSLs have no dependence on the resources of another language. In profiling the external DSLs, the following information is considered: the developing organization (company), its semantics (compiled/interpreted), notation, features specific to the external DSL, type of robots the DSL supports, and the domain the language is targeting.

[16] https://github.com/bhuman/CABSL

2.3.2.1 NaoText

NaoText [10] is an external DSL developed by the research group QualiTune. The DSL is a role-based language for specifying collaborative missions for NAO robots using a textual notation. Nao Text uses CPSTextInterpreter, which runs on the Java runtime environment using Maven to manage dependencies.[17]

The code below shows the declaration of a pass action in a soccer game between NAO robots.[18] Some of the domain terms used in specifying the mission in Listing 2.3 include striker, ballpossesor, and ballseeker.

```
1  activate for {     // (1) player selection
2    BallPossessor p;
3    BallSeeker s;
4  } when {          // (2) condition
5    ((p.robotInVision(s)) and
6    !(p as Striker).isGoalShotPossible());
7  }
8  with bindings {    // (3) role binding
9    p + Sender;    //    bind Sender role
10   s + Receiver;  //    bind Receiver role
11   s − BallSeeker; //   unbind BallSeeker role
12 } with settings {  // (4) evaluation time settings
13   interval 500;  // check every 500ms
14   after 1000;    // start after 1s
15   continuously true;
16 }
```

Listing 2.3: A snippet of a mission for Pass Action in a soccer game between Nao robots.

2.3.2.2 EasyC

EasyC is a commercial product of Intelitek that provides a graphical notation for programming VEX robots. The DSL auto-generates C code from missions specified using the drag and drop graphical editor. Experienced C programmers can seamlessly switch to a fully text-based development environment. This DSL has been enriched with robotic abstractions such as robot driving—Drive, Turn, Stop or Drive for Time.

EasyC uses a graphical interface on top of Intelitek's own C library[19], which was custom made for the VEX Cortex and IQ robot controllers.

2.3.2.3 BehaviorTree.CPP

BehaviorTree.CPP [29] is a C++ library for creating behaviour trees. It is developed by a research group at the Eurecat Technology Centre. The library provides a flexible framework to easily specify robot mission as behaviour trees that can be loaded at runtime for execution. The nodes of the tree are either actions the robot can execute or conditions to be fulfilled before an action is taken.

The BehaviorTree.CPP DSL provides mechanisms to monitor, log, and debug the execution of a tree. The behaviour trees for robot missions are

[17] https://github.com/max-leuthaeuser/CPSTextInterpreter

[18] http://www.qualitune.org/?page_id=453

[19] https://www.slideserve.com/tova/april-27-2006-programming-with-easyc-and-wpilib

executed using the C++ language runtime environment. Groot[20] provides a graphical editor for the C++ library to create and edit behaviour trees. The primitives in Groot, built-in nodes, or custom nodes can be dragged and dropped to build a required behaviour tree. Domain terms and expressions such as DetectObject, Grasp, GetMapLocation, and MoveTo have been used in the DSL for mobile robots with the ability to move, recognize, and grasp objects.

2.3.2.4 Unreal Engine 4 Behavior Trees

The Unreal Engine 4 (UE4) Behavior Tree [32] is a commercial DSL developed and maintained by Epic Games, Inc. Behaviour trees define the Unreal AI agent's processor, which makes decisions and executes various branches based on the outcome of those decisions. The Unreal Engine implements behaviour trees using the Blackboard tool which acts as the "brain" of the AI character and stores key values that the behaviour tree uses to make its decisions. A behaviour tree task is an action the AI character can perform, for instance, move to a location or rotate to face an object.[21] Some examples of domain expressions are SetMovementSpeed, LookStraightAhead, and RapidMoveTo. The DSL has been used for simulation characters in video games, representing humans, helicopters, and vehicles. The Unreal Engine uses the Unreal scripting languages with a graphical editor for creating UE4 behaviour trees, related blackboards for the behaviour trees, and tasks—i.e. actions. The scripting languages are compiled using the UnrealScript Compiler.[22]

2.3.2.5 Choregraphe

Choregraphe [87, 88] is a commercial DSL produced and maintained by SoftBank Robotics for programming Aldebaran robots such as NAO. The language aids users to create animations, behaviours, and dialogues for the NAO humanoid robot—meant for experimentation and research. Choregraphe also offers simulation support for the NAO robot. The graphical DSL provides a flowchart-like interface in which end-users specify missions by connecting boxes to construct a behaviour for the robot.[23] Boxes are pre-programmed libraries, which abstract mission primitives. Some of the mission primitives include Play Sound, Set Speech Lang, and Speech Reco.

2.3.2.6 Microsoft Visual Programming Language

The Microsoft Visual Programming Language (MVPL) [73] is a DSL in Microsoft Robot Development Studio used for programming robotic applications based on the idea of boxes and arrows. The language concepts (activities) are represented by boxes while the arrows connect the boxes to build a program.[24] The MVPL data flow diagram consists of a connected sequence of activities

[20] https://github.com/BehaviorTree/Groot

[21] https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/index.html

[22] https://docs.unrealengine.com/udk/Three/UnrealScriptReference.html

[23] http://doc.aldebaran.com/1-14/software/choregraphe/interface.html

[24] https://acodez.in/microsoft-robotics-developer-studio/

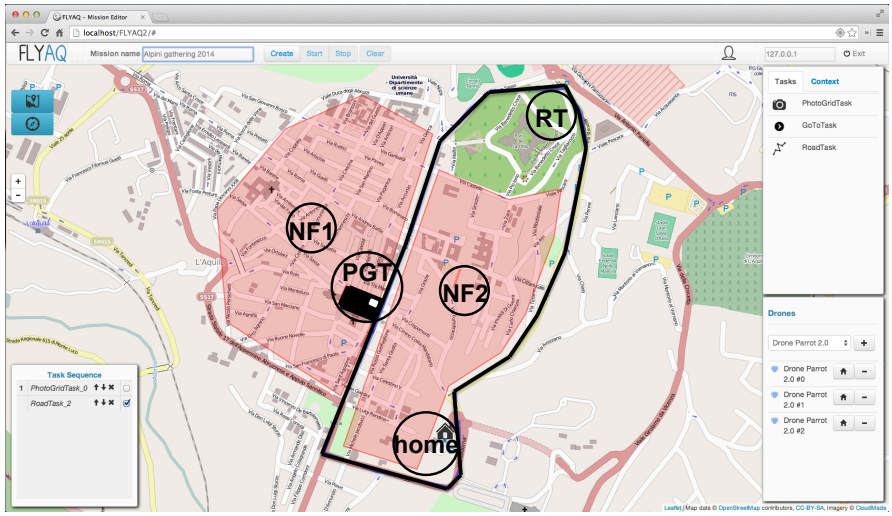


Figure 2.2: Specifying a mission for a drone to hover in given space while avoiding no-fly zones [11].

represented as blocks with inputs and outputs that can be connected to other activity blocks.

2.3.2.7 Open Roberta

Open Roberta [89–91] is a web-based educational DSL developed by the Fraunhofer Institute, which offers free use for individuals, but commercial use for institutional use. The Blockly-based DSL can be used to program a variety of robots: Lego Mindstorms EV3 and NXT, Calliope mini, micro:bit, Bot'n Roll, NAO, and BOB3. This DSL provides a rich set of behaviour abstractions and primitives, which are mainly categorized into actions (drive, turn, steer, show, play, say), sensors (touch, ultrasonic, colour, infrared, temperature, gyro, timer), control (program control flows), logic (comparisons, AND, OR, Boolean), math (constants and arithmetic operators), text, colours, and variables. The specifics of these abstractions vary according to the robot for which the mission is specified. This DSL has a considerable potential in harnessing end-user programming, since it is a drag and drop graphical language with syntactic and semantic editor services. The DSL can either be run on the cloud or installed on a local server. Open Roberta generates code in Python, Java, JavaScript, and C/C++ depending on the target robot.

2.3.2.8 FLYAQ

FLYAQ [33–36] is an open-source research prototype developed and maintained by a team of researchers that provides an extensible DSL for specifying missions for a variety of robots, including quadrotors. The monitoring mission language (MML) allows specification of mission context such as obstacles, flight path (i.e., starting point, action points, ending point), and no-fly zones on a live map.

The executable code is automatically generated to be executed by a robot or a swarm of robots as shown in Fig. 3.6. The DSL is suitable for missions such as surveillance, public order management, and agriculture. The concrete syntax (i.e., the map) used for specifying the mission context makes the language reachable for end-users. The FLYAQ virtual machine provides a ready-to-run version for the end-user.

2.3.2.9 Aseba

Aseba [92, 93] is a DSL with variants of visual and text syntaxes, created by Mobsya under Creative Commons Attribution-ShareAlike 3.0 License. The language syntax variants are the visual programming language (VPL), a Blockly-based language, a Scratch-based language, and the Aseba textual language. VPL provides events and action-based programming in which, for a given event, there is a corresponding action. These events are triggered by data from sensor readings. Examples of events are press button, obstacle detector, ground detector, robot tapped, and hand clap. In turn, examples of actions are set motor speed, set top or bottom colour, and play music. The same language concepts can be programmed using the other DSLs of Aseba. Some common behaviours[25] associated with the Thymio robot are: friendly (follow hand and react to another Thymio robot), explore (avoid obstacles and stop when the ground is dark), fearful (goes away when approached and scream when cornered), attentive (changes colour and moves depending on the number of claps detected), investigator (follows a black track), and obedience (reacts to button and remote control).

2.3.2.10 LEGO Mindstorms EV3

The LEGO Mindstorms EV3 builder [94, 95] makes it possible to create robots that can do a number of things such as walk, talk, or drive. The graphical DSL provides a rich set of language constructs categorized into action, flow, sensor, data, and advanced blocks. For instance, the action blocks include move steering block, display block, and sound block, which can be used for specifying a mission by kids learning how to program. The DSL is a visual language with blocks connected to form missions. Figure 2.3 shows a mission specification in which the robot says "Hello" once, then "Go" six times, and then "Bravo" once. The sound blocks are used for creating the respective sounds while the flow block—the loop is used for repeating the "Go" sound. Each block is an icon of the function it executes.

2.3.2.11 MissionLab

MissionLab [83,96] was created by the Mobile Robot Laboratory at Georgia Tech and is a research prototype DSL that facilitates mission specification through a state-machine-based visual language. The DSL uses assemblage and temporal sequencing constructs to create a temporal chain of behaviours as a mission. The assemblage construct defines behaviour primitives and coordination mechanisms. During mission specification, the assemblage is instantiated.

[25] https://www.thymio.org/basic-behaviours/

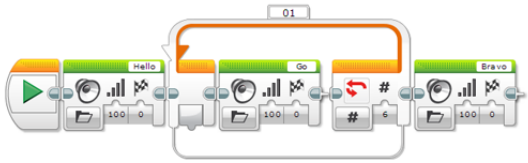


Figure 2.3: Specifying a loop in LEGO Mindstorms EV3.

The temporal sequencing creates states with perceptual triggers to enable transitions between states. MissionLab provides a graphical editor-based configuration description language (CDL) to specify multi-agent missions. Missions can be executed on a simulator or on the following wheeled robots used for smaller commercial applications: ATRV-Jr, Urban Robot, AmigoBot, Pioneer AT, and Nomad 150 & 200.

2.3.2.12 RobotC

The language primitives are in the form of blocks, which users drag and drop to build a program, making it easy for novice programmers to write robot missions. Listing 2.4 illustrates the same program in textual version of RobotC Natural language.[26] The expressions preserve the C language syntax, while the natural language makes it easy for novice programmers to comprehend the programs.

```
1  task main(){
2    int counter = 0; //set the value of variable "counter" to zero
3    while(true) { //loop forever
4      //count the number of times the sensor is pressed
5      if(getTouchValuetouchSensor) == true){
6        counter = counter +1;
7        waitUntil(getTouchValue(touchSensor) == false); // wait for the sensor to no longer be
         pressed
8      }
9      displayVariableValue(line1, counter); //display the value of "counter" on line 1
10   }
11 }
```

Listing 2.4: Snippet of a program in RobotC Natural Language

2.4 Making Robots Usable in the Everyday Life

Mobile robots are increasingly used in everyday life to autonomously realize missions such as exploring rooms, delivering goods, or following certain paths for surveillance. The current robotic market is asking for a radical shift in the development of robotic applications where mission specification is performed by end-users that are not highly qualified and specialized in robotics or ICT. To this end, in the context of the Co4Robots EU H2020 project,[27] we developed

[26] http://www.robotc.net/NaturalLanguage/
[27] http://www.co4robots.eu

our contribution in two steps.

- First (Sect. 2.4.1), with the aim of understanding the missions that are currently expressed in practice, we surveyed the state of the art and formulated and formalized a catalogue of 22 mission specification patterns for mobile robots. We also provide tooling for instantiating, composing, and compiling the patterns to create mission specifications [1, 97].
- Second (Sect. 2.4.2), using specification patterns as main building blocks, we proposed a DSL that enables non-technical users to specify missions for a team of autonomous robots in a user-friendly and effective way [8, 98].[28]

2.4.1 Mission Specification Patterns

The proposed patterns provide solutions for recurrent missionspecification problems for service robots and they focus on robot movement and on how robots perform actions within their environment. The first step for creating the catalogue of patterns was the collection and analysis of 245 natural language mission requirements systematically retrieved from the robotics literature. From these requirements, we identified recurrent mission specification problems to which we provided solutions and organized them as patterns. The patterns provide a formally defined vocabulary that supports robotics developers in defining mission requirements in an unambiguous way.

The patterns provide a formal and precise description of what robots should do in terms of movements and actions, and therefore, relying on the usage of the pattern catalogue as a common vocabulary makes it possible to mitigate ambiguity in natural language formulations. Moreover, the patterns also provide validated mission specifications for recurrent mission requirements, facilitating the creation of correct mission specifications.

A pattern is described in terms of a structured English formulation, its usage intent, known uses, relationships to other patterns, and, most importantly, a template mission specification in temporal logics. Since the patterns do not contain an explicit time or probability, the temporal logics used are LTL and CTL. This catalogue might be extended in many directions, e.g. by considering explicit time, probability, cost, utility, and other aspects. Patterns, while keeping their roots in a formal language, can be used by non-experts as well.

To further support developers in designing missions, we have implemented the tool PsALM (Pattern bAsed Mission specifier). PsALM allows the user (i) to specify a mission requirement through a structured English grammar, which uses patterns as basic building blocks and operators that enable composition of the patterns into complex missions, and (ii) automatically generate specifications from mission requirements. PsALM also enables the composition of patterns towards the specification of complex missions by the conjunction or disjunction of the patterns [97].

We thoroughly validated the patterns [1]. We evaluated the benefits of using our patterns for designing missions by collecting 441 mission requirements in natural language: 436 obtained from robotics development environments used

[28] PROMISE webpage: `https://sites.google.com/view/promise-dsl/home`

by practitioners, and 5 defined in collaboration with 2 well-known robotics companies. Further information about the theoretical aspects might be found in [1], and about the tool in [97], while details about the specification of each pattern might be found in the website[29].

2.4.2 PROMISE

In order to support the specification of more complex missions with respect to those that can be specified using the specification patterns, and in order to enable the specification of missions for multiple robots, we proposed a domain-Specific Language called PROMISE. PROMISE considers the mission specification patterns as atomic tasks that can be executed by robots and proposes sophisticated composition operators for describing complex and multi-robot missions. These operators are inspired by behaviour trees [99, 100] in their style and notation. The DSL is integrated into a framework,[30] which allows the seamless specification and execution of a mission. The framework contains:

[a] The realization of the language using Eclipse and two plugins for language workbench, namely, Xtext[31] and Sirius.[32] In this way, mission specification can be performed through textual and graphical interfaces, which are synchronized.

[b] A compiler implemented using Xtend[33] for mission code generation.

[c] An interpreter, which parses the mission code and gives the low-level commands to each robot accordingly.

While the DSL support is provided by a standalone tool and can be integrated within a variety of frameworks, the current implementation has been integrated with a software platform [101] that provides a set of functionalities, including motion control, collision avoidance, image recognition, SLAM, and planning. This software platform has been implemented in ROS.

Our DSL has been successfully validated through experimentation with both simulation and real robots. Footage of the validation through experimentation we have conducted can be found on the dedicated website. The experimentation led to a demonstration of several missions to the Co4Robots consortium, which triggered important feedback. For instance, an industrial partner from the Bosch Center for Artificial Intelligence suggested that practitioners from their logistics facilities would appreciate a response from the tool stating a natural English description of the mission that had been specified. An example of such a description is provided in Section 2.5. We targeted specific robots during the experimentation, however, PROMISE is intended to be robot-agnostic, so it could be integrated with any robot by modifying the interpreter with the interfaces required for the new robot. The experimentation enabled us to validate PROMISE from the point of view of expressiveness by

[29] Specification Patterns for Robotic Missions webpage: `http://roboticpatterns.com`

[30] `https://github.com/SergioGarG/PROMISE_implementation`

[31] `https://www.eclipse.org/Xtext/`

[32] `https://www.eclipse.org/sirius/`

[33] `https://www.eclipse.org/xtend/`

measuring the ability to write missions defined by practitioners, as we will detail in Section 2.5.

Our language and its framework implementation have been also validated in terms of usability, by measuring the ability of potential end-users in using the DSL for specifying missions. To this end, we conducted two user studies, where participants were instructed before the study and then received a set of tasks to be fulfilled within a given time frame. After the tasks' completion, the participants were asked to submit their results and to fill in a questionnaire. The first of the studies was conducted at the University of L'Aquila as an exploratory validation, which triggered important refinements in PROMISE, especially in its implementation. Examples of refinements are the inclusion of a wizard to help the users in the first steps of mission specification—e.g. defining the number of robots and locations.

The second user study was designed to understand the elements of PROMISE that could be perceived as error-prone by the participants and to measure how confident the participants were of their provided solutions. During this study, the participants had to specify missions using PROMISE from textual descriptions within a time frame of 30 minutes. Furthermore, the participants were requested to validate their solutions through experimentation using a ROS and Gazebo-based setup in a provided laptop. All the participants were able to correctly specify their missions within the given time frame and to validate the results of two thirds of their missions through simulation. Based on the responses to the questionnaire, the perception from the users was positive towards the language and its implementation, not considered error-prone. We collected qualitative data from the questionnaire using open-ended questions, which also triggered refinements in the language and its implementation. Some of the responses to those open-ended questions remain as future lines of work, as, for example, enhancing the feedback offered to the user during mission specification.

Further information regarding PROMISE and the validation procedure we followed during its development might be found in our previous study [8], in a tool paper [98], and in the DSL's dedicated webpage.

2.5 Putting PROMISE into Practice

In the previous section, we introduced the methods and mechanisms we developed to make robots usable in the everyday life in a descriptive way. In the following, we present an example of a mission and its specification using PROMISE together with a comprehensive discussion of the context in which it has been defined. This example originates from our work in the Co4Robots project,[34] which aimed at developing a full functioning robot that integrates several robotic skills that we have developed, including navigation, self-localization, and planning. Its focus is on robotic applications realized on top of robotic platforms provided by our industrial partners, including a

[34] http://www.co4robots.eu

TIAGo robot[35] and an ITA robot[36], both in real life and simulation. To test our developments, when we could not directly access any of these robots, we used an economic and easy-to-use robot, the Turtlebot 2.[37] It does not provide a wide range of functionalities, but allows easy prototyping, while testing recognition and navigation skills before deployment to the production-level robots TIAGo or ITA.

Our example scenario is inspired by a mission proposed for the 2018 edition of the well-known robotics competition RoboCup@Home. We replicated and made available in PROMISE's repository several missions proposed in the rules of this RoboCup@Home'18 [102]. Concretely, we use here the restaurant simulation scenario as an example. In this scenario, two robots collaborate to help clients in a simulated restaurant at the same time. The robots are required to ask the customers for their order and deliver drinks or snacks provided by a barman (i.e. the human *operator*), while people walk around. Both robots must work in parallel.

For our project, we have used Python as the development programming language since it is one of the most common languages used in robotics together with C++ [39], as discussed in Sect. 2.2. It is also well-supported by the Robot Operating System (ROS) [20] middleware. Many libraries, such as for testing or developing dedicated skills, are also written in Python. As anticipated above, we make use of ROS since it is the most widespread middleware and it is used by the Turtlebot 2 and TIAGo.

Next, also influenced by our middleware choice, we have designed a three-layered software architecture for the software, because it supports the separation of concerns among processes with different layers of abstraction [101]. We have also opted to adhere to a component-based approach, mostly because ROS enforces the component-based software development with its clustering of software modules into packages and nodes. If properly performed, the step of designing and adhering to a software architecture simplifies the later integration of robotic skills while promoting their documentation. As a mainstream IDE, we used Eclipse. In Section 2.2.2 we present and discuss popular IDEs that support users in developing robotics software, distinguishing between mainstream IDEs, such as Eclipse, which are extensible via plugins for various robotics aspects, and dedicated robotics IDEs.

Figure 2.4 shows the representation of the restaurant scenario using the graphical syntax of PROMISE. The image has been edited with circled numbers to label nodes and ease the explanation of the mission. In turn, Fig. 2.5 shows the textual representation of the same mission. This figure also contains circled numbers, which label the same nodes, and therefore supports the reader while linking the graphical-textual mapping.

Running example: mission defined using PROMISE.
The root of the mission specification, i.e. the operator *parallel*, is identified by the node ① and specifies that *robot1* and *robot2* must perform their missions in parallel. A robot is assigned to each branch associated with this operator, as indicated with labels in the edges between ① and ② in Fig. 2.4, and with the

[35] http://pal-robotics.com/robots/tiago/

[36] https://www.bosch-presse.de/pressportal/de/en/current-examples-of-robotics-research-102528.html

[37] https://www.turtlebot.com/turtlebot2

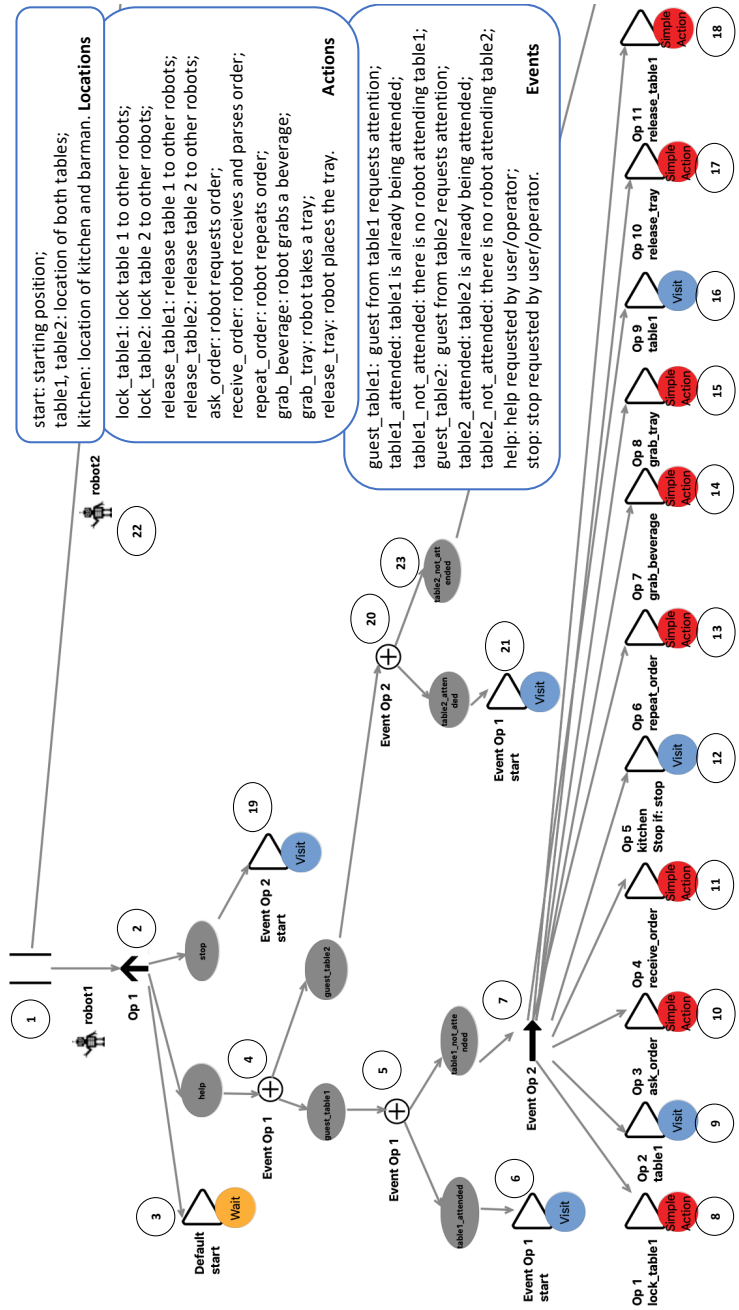


Figure 2.4: Running example specified with the graphical syntax of PROMISE.

name of the assigned robot (i.e., *robot1* and *robot2*) in Fig. 2.5. Since the mission for *robot2* (㉒) is a replica of the one for *robot1*, we only show the latter for the sake of conciseness.

The operator labelled with ② is the *eventHandler*—more information

```
operators{ parallel{ 1
    robot1 (
        eventHandler ( 2
            default ( delegate ( Wait locations start ) ) 3
            except help (
                condition ( 4
                    if guest_table1 (
                        condition ( 5
                            if table1_attended ( delegate ( Visit locations start ) ) 6
                            if table1_not_attended (
                                sequence ( 7
                                    delegate ( SimpleAction actions lock_table1 ) , 8
                                    delegate ( Visit locations table1 ) , 9
                                    delegate ( SimpleAction actions ask_order ) , 10
                                    delegate ( SimpleAction actions receive_order ) , 11
                                    delegate ( Visit locations kitchen stoppingEvents stop), 12
                                    delegate ( SimpleAction actions repeat_order ) , 13
                                    delegate ( SimpleAction actions grab_beverage ) , 14
                                    delegate ( SimpleAction actions grab_tray ) , 15
                                    delegate ( Visit locations table1 ) , 16
                                    delegate ( SimpleAction actions release_tray ) , 17
                                    delegate ( SimpleAction actions release_table1 ) 18
                                )
                            )
                        )
                    ) if guest_table2 (
                        condition ( 20
                            if table2_attended ( delegate ( Visit locations start ) ) 21
                            if table2_not_attended ( 23
                                sequence (
                                    delegate ( SimpleAction actions lock_table2 ) ,
                                    delegate ( Visit locations table2 ) ,
                                    delegate ( SimpleAction actions ask_order ) ,
                                    delegate ( SimpleAction actions receive_order ) ,
                                    delegate ( Visit locations kitchen stoppingEvents stop),
                                    delegate ( SimpleAction actions repeat_order ) ,
                                    delegate ( SimpleAction actions grab_beverage ) ,
                                    delegate ( SimpleAction actions grab_tray ) ,
                                    delegate ( Visit locations table2 ) ,
                                    delegate ( SimpleAction actions release_tray ) ,
                                    delegate ( SimpleAction actions release_table2 )
                                )
                            )
                        )
                    )
                )
            )
            except stop ( delegate ( Visit locations start )))) , 19
    robot2 ( 22
```

Figure 2.5: Running example specified with the graphical syntax of PROMISE.

regarding PROMISE's operators is available in [8]. It has a default behaviour; in our example, it forces the robot to wait in location *start* (③). This behaviour is paused when one of the events that are assigned to the *eventHandler* is detected by the robot. The default robot's behaviour (③) is resumed whenever any of the behaviours triggered by an event is finished (either succeeding or failing).

Each event is assigned to a child of the *eventHandler* (as represented in Fig. 2.4) as grey circles and invoked in Fig. 2.5 by the keyword `except`. If the event "help" is detected, the first operator *condition* (④) is executed. This operator evaluates its associated events in order, and if they hold, it triggers the behaviours associated with them. In this case, the operator *condition* evaluates whether the request of help comes from *table*1 or *table*2.

In case "guest_table1" holds (i.e. the request of help comes from *table*1), another operator *condition* (⑤) is executed. This operator evaluates whether this table is already being attended to by another robot ("table1_attended") and, in this case, makes the robot return to the starting position *start*. This behavior is encoded by the instantiation of an operator *delegate* with a task

Visit (⑥).

The next operator *condition* (⑤) evaluates "table1_not_attended," and, if it holds, the execution of an operator *sequence* (⑦) is triggered. This operator executes in sequence a set of operators. Concretely, the sequence of operators starts with ⑧, which "locks" *table*1 from the rest of the robotic team by forwarding a message (in this case, other robots will recognize it with the event "table1_attended'). The robot will then move to *table*1 (⑨), ask the order (⑩), and receive and parse it (⑪). The robot will then move to *kitchen* (⑫) to interact with the barman (i.e. the human operator). Note that this specific task can be stopped by the user or human operator by means of the event "stop" (see Fig. 2.4). Once the robot has reached location *kitchen*, it will repeat the order to the barman (⑬), after which the robot will grab beverages (⑭) and a tray with the ordered snacks (⑮). The robot will then return to *table*1 (⑯) with the order, where it will place the tray (⑰). The sequence of tasks finishes with the robot "releasing" the table for other robots, in a similar way as to how it locked it.

The operator *condition* ⑳ is a replica of ⑤—see the conditions in the textual representation in Fig. 2.5 "if guest_table1" and "if guest_table2"—and, therefore, we do not show its whole graphical representation for the sake of conciseness.

As suggested by an industrial partner after a demonstration to the Co4Robots consortium, PROMISE prompts a natural English description of the mission once specified and saved. An excerpt from the description of the example introduced in this section is as follows.

> *Robot robot1 does by default wait in location start, and if event help occurs, it will, if event guest_table1 holds, and if event table1_attended holds, visit (without any specific order) location(s) start. If event table1_not_attended holds, it will perform action lock_table1 and then visit (without any specific order) location(s) table1 and then perform action ask_order, and then perform action receive_order, and then visit (without any specific order) location(s) kitchen, and then perform action repeat_order, and then perform action gra_beverage, and then perform action grab_tray, and then visit (without any specific order) location(s) table1 and then perform action release_tray and then perform action release_table1.*

The mission of the example was modelled through mission specification from a natural English description, in this case, from the rules of the RoboCup@Home'18 [102]. Once the mission was modelled, we proceeded to validate it through experimentation in an iterative way. The first step we took was simulation,[38] for which we used simulated models of the facilities and robotic models provided by the industrial partners of Co4Robots for Gazebo [103]. Once the simulation was performed and the mission specification validated, we proceeded with validation in real life. As explained above, we purchased a Turtlebot2 for experimentation. We validated the same restaurant scenario with the Turtlebot in the facilities of the University of

[38] https://www.youtube.com/watch?v=F3BnIEPB8Sk

Gothenburg.[39] The last step was to conduct a demonstration in the presence of the project consortium at the facilities of PAL Robotics, for which we used a TIAGo robot.[40] Through this process, we demonstrated the ability of PROMISE to specify complex missions from textual descriptions. We also demonstrated its capability to operate with different robots by accordingly modifying the interpreter of its framework—see Section 2.4.2.

We invite the interested reader to learn more about the validation procedures we followed during the development of PROMISE in our published studies [**?**,8] and on its dedicated website.

2.6 Discussion and Perspectives for Future Research

As discussed in this chapter, in the last years there have been many contributions from the research community to propose domain-specific languages for mission specification [8, 36], the description of missions in natural language [104], and visual and end-user-oriented mission environments [13, 16, 17, 105].

The approaches surveyed here greatly contribute to the field; however, the mission specification-problem still requires solutions able to make robots usable in everyday life for accomplishing complex missions. In the following, we highlight the limitations of current approaches and we devise perspectives for future research. As stated also in the Multi-Annual Roadmap (MAR) For Robotics in Europe [106], in order to reduce costs and establish a vibrant component market, there is a need for instruments for supporting mission *reuse* and diversification, as well as coping with the *variability* of conditions of application scenarios occurring in real environments. This is also testified by our findings during our collaboration with practitioners in the robotic domain: the complexity does not reside in commanding a robot with a set of tasks but in making the robotic application robust enough to be able to cope with the variability that characterizes the real environments in which the robots are required to operate, especially those that involve humans [39].

To the best of our knowledge, few approaches try to address the reusability and variability envisioned by the MAR. PROMISE and the specification patterns are greatly contributing; however, there are some aspects that should be investigated in the future. In the following, we devise important research directions, which we identified based on our collaboration with companies in the Co4Robots project and additional collaborations in the healthcare domain. Specifically, we believe that the main research directions go in the following directions:

- *Reusability*: the DSLs we will develop for enabling end-users to specify missions will make use of libraries of tasks and skills. They will also integrate with libraries produced by other projects and initiatives, like RobMoSys.[41]

- *Variability of the real world*: the DSL will be conceived to enable the specification of the variability of conditions of complex real-world scenarios.

[39] `https://www.youtube.com/watch?v=Qr9FqzSrZuk`

[40] `https://www.youtube.com/watch?v=zP1PjGX84Qk`

[41] `https://robmosys.eu`

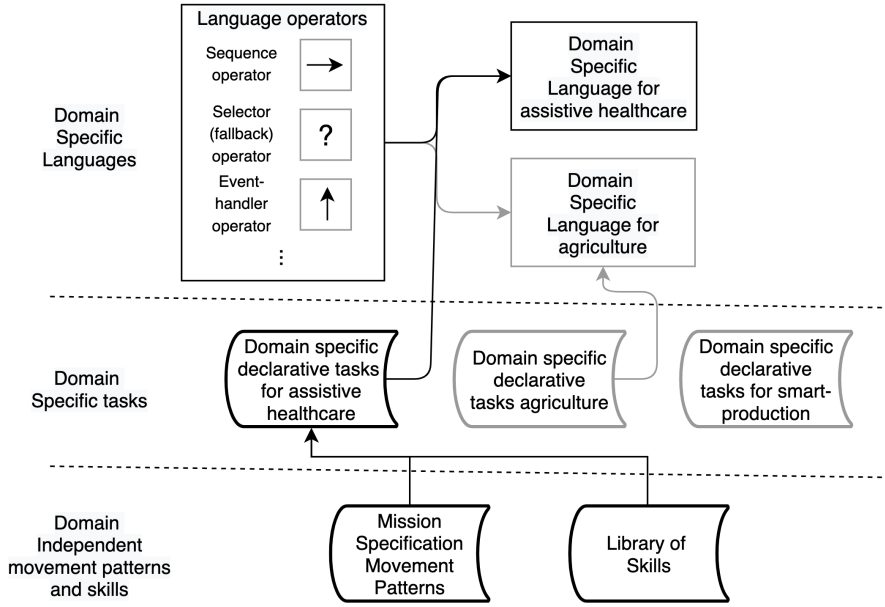


Figure 2.6: Mission specification.

- *Fleet specification of a mission*: the end-user that will specify the mission does not need to assign tasks to specific robots, but the mission specification will represent the "needs" of the end-user and robots will be automatically assigned and potentially re-assigned during the mission execution, according to the capabilities of robots and various quality parameters.

- *Human-robot collaboration*: the mission specification will include also humans, with two different roles, namely, *operators*, able to perform actions needed to successfully accomplish the mission, and *patients*, which will require actions from robots.

In order to support what we believe we might need, various libraries of pre-defined solution schemes that can be reused, instantiated, and composed by means of properly defined operators need to be implemented. As shown in Fig. 2.6[42] we envision three different types of libraries organized on two levels, one being application-domain-independent (specific for service robots), and the other one being domain-specific, e.g. assistive healthcare, agriculture or smart production.

- *Mission specification movement patterns* are pre-defined solutions concerning movements of robots and provide the bridge between a mission requirement expressed in structured English (a subset of English with a well-defined semantics) and a formulation in temporal logic. An initial

[42]We use the same terminology in "Architectural Pattern for Task-Plot Coordination" of the EU H2020 RobMoSys project: `https://robmosys.eu/wiki/general_principles:architectural_patterns:robotic_behavior`

result in this direction consists in the specification patterns described in Section 2.4.1.

- *Library of skills* contains the implementation of the modules for enabling the robot to do specific actions, like grasp object with constraint, low dexterity, soft grasping, image recognition, gesture recognition, and so on, that are compliant with the RobMoSys platform.

- *Domain-specific declarative tasks* are recurrent combinations of mission specifications patterns and skills used to define declarative tasks for domain-specific operations. For instance, in the assistive healthcare domain, a declarative task can be "welcome," and would require patterns for movements and various skills such as human recognition, speech recognition, etc. The tasks are declarative since they specify only what the robot is able to do without saying how the robot will do that. Then, planners will compute how the task will be solved in the specific environment according to the capabilities of the robot that will be allocated to this task.

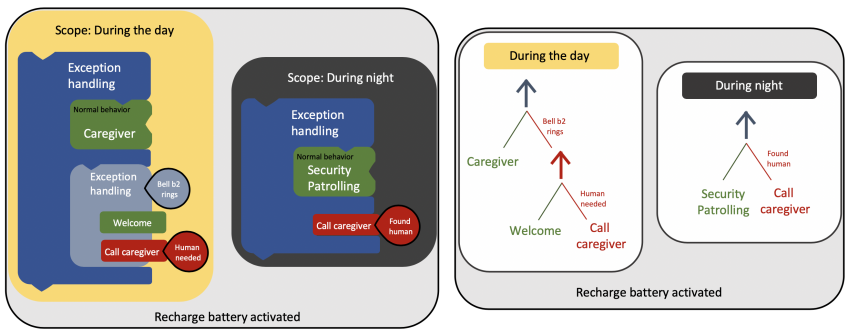
- *Domain-specific Languages*, as for instance PROMISE (Section 2.4.2), enable operators who are not required to have expertise in programming nor robotics, to specify in an easy and correct way the mission they would like the robots to safely accomplish. Each domain-specific language will make use of the language operators that we will define. There will also be specific "dialects" for specializing the language to the various domains. In this way, healthcare operators will find in the domain-specific language for assistive healthcare concepts that are specific of the domain, expressed in terms of domain-specific declarative tasks for assistive healthcare. The language enables the description of complex and sophisticated missions, which will also take into account non-functional properties, such as timing constraints. These properties are captured by composition operators, like *sequence*, *selector* (fallback), or *event-handler*, which are inspired by Behaviour trees [99,100] or by PROMISE.[43] The DSL will help healthcare operators (with a sort of wizard or recommendation system) to deal with the variability that characterizes the environments in which missions are executed. This includes "exceptional" behaviours, such as a robot running out of battery, an unforeseen obstacle hampering the mission satisfaction, an object falling down from the hand of the robot, and so on. As testified by MAR [106] and also highlighted in a recent study [39], one of the most difficult aspects in mission specification is to deal with the variability of real-world scenarios.

Example of Mission specification. During the day, "robot" welcomes newcomers when the bell "s2" of the door rings. According to the needs of the guests, "robot" will provide the needed information or ask them to enter the dining room, and if a human intervention is needed, "robot" informs a caregiver. When "robot" is in the dining room, it acts as a caregiver and interacts with people by calling them to drink and offering water that "tray"

[43]The PROMISE DSL has been developed in the context of the EU H2020 project Co4Robots [8]. PROMISE webpage: `https://github.com/SergioGarG/PROMISE_implementation`

carries. During night, "robot" patrols for security and if it finds humans in the environment it calls an operator. Robots recharge autonomously while guaranteeing the welcoming and caregiving service. Notice that the example does not include quality aspects, such as timing constraints, since patterns including these aspects are not yet available, but they will be developed during the project execution.

Mission specification: A healthcare operator will specify the mission by means of the following domain-specific macros: *Welcome*, *Security Patrolling*, *Caregiver*, and *Call caregiver*. The following figure shows the mission specified foreseeing two different graphical languages, one (a) based on the *blockly*[44] approach and the other one (b) using PROMISE's style [8]. This is just to explain what we mean by graphical and easy-to-use language for mission specification.



a) Blockly style b) Promise style

Domain-specific macros, mission specification patterns, and library of tasks: Behind the scene, i.e. invisible to the end-users, the macros will be built by using the mission specification patterns and the tasks stored in the library. For instance, *welcoming* might be realized by composing the *sequenced visit specification pattern*[45] to reach from the current location of the robot the door (LTL formula: $\Diamond$ (door_location)), with a *delayed action*[46] when the robot reaches the door to welcome and activate the speech recognition—LTL formula: $\Box$(door_location $\Rightarrow \Diamond$(welcome)), where "welcome" is a task in the library of tasks.

[44] https://developers.google.com/blockly
[45] http://roboticpatterns.com/pattern/sequencedvisit/
[46] http://roboticpatterns.com/pattern/delayedreaction/

Chapter 3

Paper B

A Survey on the Design Space of End-User Oriented Languages for Specifying Robotic Missions.

S. Dragule, T. Berger, C. Menghi, and P. Pelliccione.

In submission to International Journal of Software and Systems Modeling (SoSyM).

Abstract

Mobile robots are becoming increasingly important in society. Fulfilling complex missions in different contexts and environments, robots are promising instruments to support our everyday live. As such, the task of defining the robot's mission is moving from professional developers and roboticists to the end-users. However, with the current state-of-the-art, defining missions is non-trivial and typically requires dedicated programming skills. Since end-users usually lack such skills, many commercial robots are nowadays equipped with environments and domain-specific languages tailored for end-users. As such, the software support for defining missions is becoming an increasingly relevant criterion when buying or choosing robots. Improving these environments and languages for specifying missions toward simplicity and flexibility is crucial. To this end, we need to improve our empirical understanding of the current state-of-the-art of such languages and their environments. In this paper, we contribute in this direction. We present a survey of 30 mission specification environments for mobile robots that come with a visual and end-user-oriented language. We explore the design space of these languages and their environments, identify their concepts, and organize them as features in a feature model. We believe that our results are valuable to practitioners and researchers designing the next generation of mission specification languages in the vibrant domain of mobile robots.

Keywords

specification environments, language concepts, visual languages, robotic missions, empirical study

3.1 Introduction

Over the last decades, robots became increasingly present in our everyday life. Autonomous service robots replace humans in repetitive, laborious or dangerous activities, often by interacting with humans or other robots. According to a 2019 press release[1] at the International Federation of Robotics, the sales of robots for professional use, such as autonomous guided vehicles, inspection, and maintenance robots increased by 32%. Personal service robots are expected to exceed 22.1 million units in 2019 and 61.1 million units in 2022, while the sales for agricultural robots is projected to grow by 50% per year.

Different techniques have been proposed for engineering the various aspects of robotic behavior [35,83,107–109], such as interoperability at the human-robot (or human-swarm) level [110, 111] and at the software-component level in middlewares [112], or multi-robot target detection and tracking [109].

Engineering robotics control software is challenging. Specifying the behavior of a robot, typically called the robot's mission, is far from trivial. Sepcifically, a *mission* is a description of the high-level behavior a robot must perform [1]. As such, a mission coordinates the so-called skills of robots, which represent lower-level behaviors. Developing missions requires substantial expertise [113, 114]. For instance, using logical languages such as LTL, CTL or other intricate formalisms to specify missions is complex for users with low expertise in formal and logical languages [14, 15].

Nowadays, the task of defining missions is moving from the robotic manufacturer to the end users, who are far from being experts in robotics. Robots are also evolving from single-purpose machines to general, multi-purpose, and configurable devices. As such, the software support provided for defining missions is becoming a more important ingredient for the selection of a robot by end-users. For example, before buying a robot, in addition to the actuation and sensing abilities of the mobile robot, end-users and developers may want to understand which types of missions can be delegated to the robot and which software support is provided for mission specification.

Over the last two decades, a range of more end-user-oriented programming environments appeared. They allow specifying robot missions in a more user-friendly way, alleviating the need for intricate programming skills, which end-users are usually lacking [15, 16, 18]. Researchers and practitioners have invested substantial effort into achieving end-user-oriented programming environments for robots [13, 16–19]. In fact, almost every commercial mobile robot nowadays comes with an environment, called *mission-specification environment* for programming the behavior. Most of these environments rely on dedicated Domain-Specific Languages (DSLs) that end-users can utilize to specify missions.

This paper aims at improving our empirical understanding of the current state-of-the-art in mission specification. Our focus is on end-user-oriented languages providing a visual syntax. In our survey, we identify open-source and commercial environments that allow end-user-oriented robotic mission specification. While robot programming environments consider all programmable aspects of the robot system, we focus on environments in which robot missions

[1] https://ifr.org/ifr-press-releases/news/service-robots-global-sales-value-reaches-12.9-billion-usd

are created, designed or particularized. We consider a mission specification environment as a collection of tools that facilitate definition and stipulation of robot tasks that form a mission. We study the environments' and their languages' main characteristics and capabilities, which we model as features in a feature model [115, 116], a common method to analyse and document a particular domain.

We formulated two main research questions:

RQ1: *What visual, end-user-oriented mission specification environments have been presented for mobile robots?* We systematically and extensively identify such environments from various sources, including the Google search engine and research literature.

RQ2: *What is the design space in terms of common and variable characteristics (features) that distinguish the environments?* Our focus is on understanding the concepts that these environments and their languages offer, which end-users utilize to specify the missions of mobile robots. We conduct a feature-based analysis, resulting in a feature model detailing our results in terms of features organized in a hierarchy.

We identified a total of 30 environments and created a feature model with 133 features, both reflecting mandatory features (those found in all environments) and optional ones (those found in only some environments). These features illustrate the design space covering those environments' capabilities, general language characteristics, and, most importantly, language concepts. Table A.1 and Table A.2 show the details of the features supported by each of the considered environments. We show how our survey is useful for end-users, robot manufacturers and language engineers by reporting a set of use case scenarios and explaining how the results of this survey can be used within these scenarios. We believe that our work is valuable to end-users, practitioners, researchers, and tool builders for developing the next generation of mission specification languages and environments and to support users in the selection of the most appropriate robot(s) based on their mission specification needs.

3.2 Background and Motivation

To convey a first understanding of mission specification, we now introduce some key terminology as well as we provide a small example of a mission defined in a dedicated DSL of one of our subject environments, illustrating its advantage over writing the mission in a general-purpose (off-the-shelf) programming language.

A *mission* is a description of the high-level behavior a robot must perform. A mission represents the logic that coordinates the lower-level functionalities of robots, also known as *tasks* or skills. While this coordination logic can be written in any programming language, expressing it in a DSL avoids writing boilerplate code, focusing on the language concepts relevant for defining the mission, as well as comprehending the mission for later maintenance and evolution. Expressing a mission in a dedicated model also gives rise to specific analyses, since a dedicated DSL captures more specific semantics that are not obvious from code written in a general-purpose programming language.

Effectively using a mission-specification DSL requires a *mission specification*

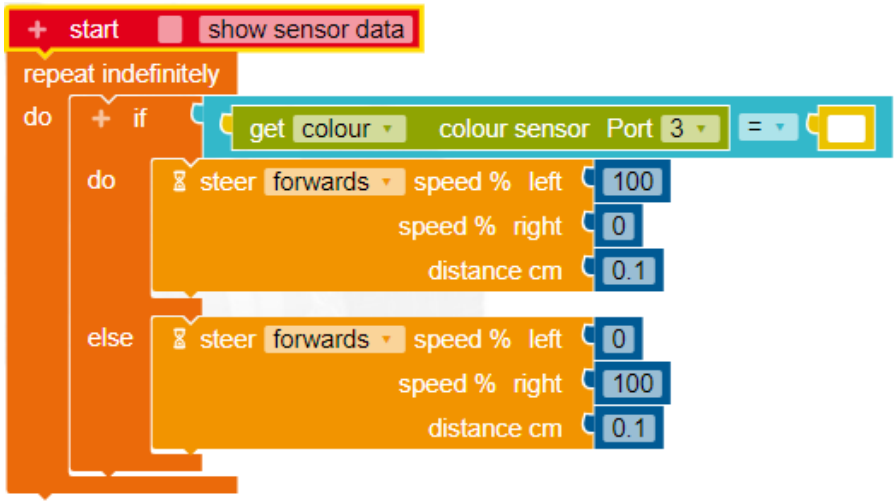


Figure 3.1: Block-based mission of a robot patrolling a perimeter wall, expressed in Open Roberta

environment. We consider such environments as collections of tools centered around one or more DSLs that provide dedicated concepts for defining robotic missions. The tools provide an infrastructure for using the languages and supporting the execution of their instances (i.e., the mission), for instance, by compiling them into programs in general-purpose languages and deploying them to the robots.

In this work, we consider *end-user facing environments*, which target end-users who are technically skilled, but not experts in robotics or in programming.
Example: A Patrolling Mission.. In a patrolling mission, a robot is expected to move along a predefined path repeatedly. During the patrolling, various data-capturing events shall be triggered, including object recognition, live video streaming, (periodically) taking photos, or recognizing noise. For each of the events, corresponding actions are expected, e.g., raise alarm or send message (text, voice, video) to a human or an agent (another robot or central system). The patrolling can continue up to a given period of time, repeated given number of times or until when a given event takes place, e.g., run out of battery.

Various mission requirements might need to be specified, such as: (i) mobile robot(s) that can move on land, air or water, (ii) patrolling data, e.g., sensing data required to keep the robot on course (color detection, obstacle detection or GPS coordinates), actions during patrolling (video screening, photography, or special object recognition), make alarm (communication with human or agent), end mission (number of rounds, time, event).

This mission has been specified using the Open Roberta environment, as shown in Fig. 3.1 with a corresponding textual code in Listing 3.1. The block-based mission can be more appealing for end-user domain experts, like farmers, than text-based mission specifications. Figure 3.1 shows an example mission for a Lego EV3 robot in Open Roberta: patrolling a perimeter wall. Listing 3.1 shows the target C code generated from this mission by Open Roberta. This code, while at the same level of abstraction as Open Roberta's

Table 3.1: Identified environments by data sources. The environments highlighted in bold were discovered first from that data source

Data Source	Identified Environments (after applying inclusion/exclusion criteria)
Environments from experience (26 candidates, selected 14, unique 14)	**MissionLab, Choregraphe, LEGO Mindstorms EV3, Sphero, TiViPE, Aseba, Robot Mesh Studio, Edison software, Makeblock 5, TRIK Studio, Ardublockly, MiniBloq, PROMISE,** and **FLYAQ.**
List of mobile robots (59 candidates, selected 20, unique 8)	MissionLab, Choregraphe, LEGO Mindstorms EV3, Sphero, TiViPE, Aseba, Robot Mesh Studio, Edison software, Makeblock 5, TRIK Studio, Ardublockly, FLYAQ, **PICAXE, Open Roberta, Arcbotics' SparkiDuino, VEX Coding Studio, Metabot, Marty software, Tello Edu App**, and **Code Lab.**
Google search (373 candidates, selected 23, unique 2)	MissionLab, Choregraphe, LEGO Mindstorms EV3, Sphero, TiViPE, Aseba, Robot Mesh Studio, Edison software, Makeblock 5, TRIK Studio, Ardublockly, MiniBloq, FLYAQ, PICAXE, Open Roberta, Arcbotics' SparkiDuino, VEX Coding Studio, Metabot, Marty software, Tello Edu App, Code Lab, **BlocklyPro**, and **Ozoblockly.**
Snowballing (80 candidates, selected 15, unique 3)	LEGO Mindstorms EV3, MissionLab, Aseba, VEX Coding Studio, Choregraphe, MiniBloq, Ozoblockly, Sphero, TiViPE, Open Roberta, TRIK Studio, Robot Mesh Studio, **Enchanting, EasyC**, and **RobotC**
Further alternative environments (3)[1]	**Turtlebot3-blockly, Makecode**, and **Scratch Ev3**

[1] Found by seeking alternative environments for robots supported by the identified environments above

mission specification (Fig. 3.1), contains intricate boilerplate code hidden by the latter's language.

3.3 Method

We now explain our methodology for identifying end-user-oriented mission specification environments (Sect. 3.3.1) and for classifying and analyzing their features (Sect. 3.3.2).

3.3.1 Identification of Environments (RQ1)

We focus on environments that support end-user programming of mobile robots, providing domain-specific languages for specifying robotic missions.
Data Sources.. We used three different data sources: (i) input provided by the authors based on experience and knowledge in the field, (ii) the Google search engine, and (iii) forward and backward snowballing upon a set of related

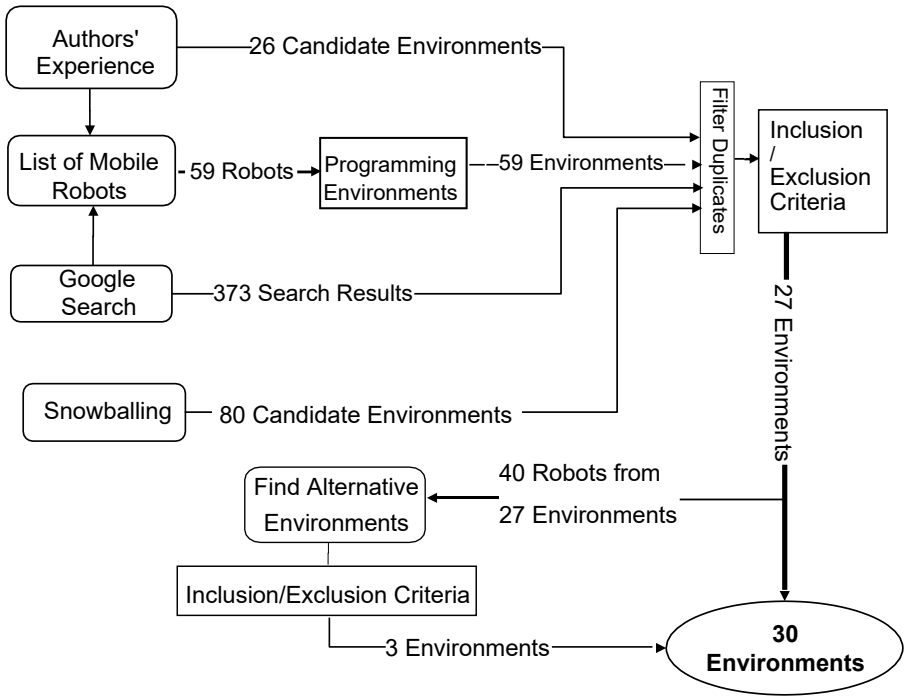


Figure 3.2: Identification of environments. More details in Table 3.1

survey papers. The survey did not use libraries, such as IEEE, Scopus, and Web of Science, since they only list publications. Yet, there are emerging tools that do not necessarily have publications.

Identification of Candidate Environments.. We identified our subject environments with the following steps:

(a) identify robot programming software in general as candidates for mission specification environments, i.e. experience, Google search, and snowballing;

(b) identify the mobile robots, being programmed by such software in (a). The mobile robots were used to search for alternative environments for programming them.

(c) apply inclusion and exclusion criteria to filter environments to be selected as candidate environments.

Figure 3.2, shows the selection strategy used, the three steps are explained in detail below:

Identify robot programming software in general as candidates for mission specification environments.. *Environment candidates from experience.* From our past experience in robotics software engineering, we assembled a list of 26 candidate environments from which 14 were selected. A list of 59 commercial mobile robots was created based on the past experience of the authors (e.g., the authors were aware of many educational robots, such as Thymio, Sphero or NAO). From the robots' web pages, we identified the software that was

offered for programming the robot missions. This step gave us exactly 59 environment candidates, from which 20 environments were selected and 8 were unique from the previously selected.

Google search. We conducted a search with the string *("programmable robots" OR ("robot programming" OR "mission specification") environment) "mobile robot"*, which yielded 373 results (note that Google reported 774,000 results, which collapsed to 373 when scrolling to the last page, which is a common phenomena. Out of the 373, 23 environments were selected but only 2 environments were unique.

Snowballing. Based on a list of six survey paperswe were aware of from our experience, we conducted snowballing. Specifically, we identified environment candidates from reading these survey papers and then from reading all papers being cited in each (backwards snowballing) and all papers citing it (forward snowballing), also ignoring duplicates. We identified 12 from [17], 44 from [117], 14 from [90], 0 from [14], 6 from [13], and 7 from [59], totaling 80 candidates. Out of the 80 candidates, 15 were selected and 3 were unique. Table 3.1 shows environments identified from particular data sources. The filter duplicate stage helped in eliminating duplicates before applying the inclusion/exclusion criteria.

In summary, we collected 537 candidate environments, from which 27 environments were initially selected. After searching for alternative environments for robot found in the 26 environments, we discovered 3 more unique environments, which makes the total of 30 environments selected.

Identify the mobile robots.. In the next step, we used the robots programmed using the environments identified from the above data sources to identify alternative environments for programming them. This was done through a Google search with the robot name as the search string. In summary, we identified 40 robots from the 27 environments selected, which yielded three more environments. As such, our final number of environments identified is 30. Table 3.1 summarizes our results.

Application of Inclusion and Exclusion Criteria.. We sieved the 537 candidates according to the following inclusion and exclusion criteria.

Inclusion Criteria. We included a candidate when it fulfilled all of the following conditions. It must:

- allow the specification of missions for mobile robots;
- offer a domain-specific language with a visual notation targeting end users;
- come with documentation about the environment and its language;
- be available to users in the sense that it is either sold or can be downloaded freely.

Exclusion Criteria. An environment is excluded if any of the following conditions holds. It must not:

- be an environment that focuses on programming system aspects of a robot, such as the Robotics Operating System (ROS), instead of specifying missions;

- target non-mobile robots, such as stationary industrial robots, 3D printers or Arduino boards;
- be a mission control application with pre-programmed missions;
- be a remote-control application for mobile robots.

In summary, we selected 27 environments using the inclusion/exclusion criteria from the initial collection of 537.

3.3.2 Analysis of Identified Environments (RQ2)

Our goal is to identify the characteristics that distinguish our subjects in the form of features [118] and to organize them in a feature model [115, 116]. Performing such a feature-based analysis is a common method for comparing the design space of technologies, such as model transformations [119], language workbenches [120] or variation control systems [121].

The data sources for analyzing the identified environments are scientific papers about them, their websites, related documentation (e.g., user manuals or programming guides), and examples of missions expressed in the respective languages.

Our strategy is as follows. First, in a brainstorming meeting, after an initial screening of the subjects, we identified key features of the environments—mainly representing the top-level and intermediate features in the feature model. Second, we consulted the websites to further identify key features and organize them in a hierarchy. This first skeleton provided the basis for an iterative refinement of the feature model by systematically, for each environment: (i) reading the scientific publications and other technical documentations about the environment; (ii) when possible, downloading and installing the environment to specify example missions, or alternatively for web-based environments, using the online tooling; and (iii) reading through the help menu for better understanding of how the environments are used to specify missions.

In this process, we iteratively refined the feature model and maintained notes about the realization of individual features in each environment. The features were discussed among all authors to reach consensus.

3.4 The Environments (RQ1)

We now summarize the identified environments. We classify them by the kind(s) of syntax they offer: block-, flowchart-, graph-, text- or map-based syntaxes. Table 3.2 lists all environments together with (i) the language syntax(es) supported; (ii) whether the environment is designed for desktop computers, mobile devices or is web-based; and (iii) the mobile robot that is supported, and its manufacturer. Our online appendix[2] provides further details about each environment.

[2] https://drive.google.com/file/d/1sh5qNS7Oo3itioNijez6ryekZ4FRlXtx/view?usp=sharing

3.4.1 Environments with Block-Based Languages

Block-based languages use visual blocks to represent the language syntax. Such blocks have various shapes and colors for the various language constructs. Typically, the block shapes visualize constraints, e.g., where in the mission specification the language concept represented by the block can be used. Block colors often depict a particular kind of functionality, such as yellow for actions and green for sensor usages, as seen in the environment Open Roberta [89].

The majority, that is, 23 out of our 30 environments offer a block-based syntax. Most of these environments are used for teaching, as shown in Table 3.2. There is some attempt to use these languages for industrial use.[3]

The syntaxes of these block-based languages are typically implemented using the popular open-source libraries Blockly [155, 156] and Scratch [157]. Specifically, Blockly is developed by Google for creating visual notations, where each block represents a programming concept. The library can be extended to define new blocks, support functions, and procedures. Blockly allows access to the parse tree offers a code-generation framework to generate code in the target (general-purpose) language [137]. Scratch is similar to blockly, but developed by the MIT media laboratory [157]. The library can be extended by adding custom, end-user-oriented blocks.

3.4.2 Environments with Flowchart-Based Languages

A flowchart is a diagram representing a step-by-step process of executing tasks. Flowchart-based languages make use of flowcharts to define the behavior and to organize the various blocks, which include: start/stop, process block, decision block, and input/output block. Each of these blocks is connected by a flow line (arrow) indicating the order of executing the mission. The syntax supports language constructs such as if-then-else, loops, and assignments.

Only 3 of our 30 environments offer a flowchart-based syntax for mission specification, namely EasyC, PICAXE, and Robot Mesh Studio. As an example, Fig. 3.3 shows a flowchart for managing a flashlight that switches on and off with a time interval of 0.25 time units. The program consists of a main program and a subroutine (FLASH). After creating the mission in flowchart editor, a separate text file is generated when the mission is compiled.

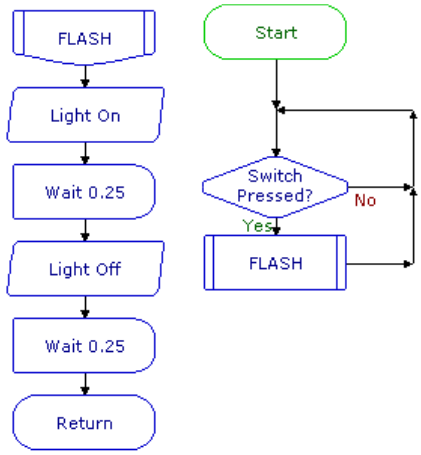
3.4.3 Environments with Graph-Based Languages

Environments offering languages with graph-based syntax represent mission components, such as tasks and mission primitives, as graph nodes. These nodes are connected in a directed graph, where the edges indicate control flow.

Only 4 out of our 30 environments exhibit languages coming with a graph-based syntax, namely Choregraphe, TiViPE, MissionLab, and TRIK Studio.

Figure 3.4 shows a graph-based mission specified using MissionLab. MissionLab offers finite state automata (FSA) to model the behavior of robots, where each node represents a high-level behavior. The mission is specified using the graphical configuration editor to create the FSA. The FSA in Fig. 3.4

[3] https://new.abb.com/news/detail/59950/abb-makes-robot-programming-more-intuitive-with-wizard-easy-programm




```
1    'BASIC converted from flowchart:
2
3  ⊞ ;Symbols
40
41   main:
42
43   Cell_7_4:
44         if b1 <> b1 then
45
46               goto Cell_7_5
47         end if
48         goto Cell_7_4
49
50   Cell_7_5:
51         gosub prc_FLASH
52         goto Cell_7_4
53
54   prc_FLASH:
55         high A.0
56         pause 250
57         low A.0
58         pause 250
59         return
```

Figure 3.3: Left: A mission specified in PICAXE's flowchart-based language. Right: generated target code in PICAXE's BASIC language (from [70])

describes a scouting mission of multiple robots operating in different formations. Each state (illustrated by circle arrows) represents a formation, with transitions (arrows with labels in rectangle boxes) representing conditions in which to advance to a new state. In our example, the robots start in the line formation, then column formation, then wedge formation, and finally the diamond formation.

3.4.4 Environments with Text-Based Languages

Most of the textual syntaxes offered by our mission specification environments are abstracted with domain-specific terms and expressions, either in the robotics domain or the end-user domain. In total, 13 of our 30 environments support mission specification in textual syntax—in almost all cases when the environment supports using a GPL in addition to its main DSL for mission specification. Notable exceptions are Aseba and PROMISE, whose DSLs also offers a textual syntax. In the other environments, the GPLs with textual syntax used include, for instance, Python, C/C^{++}, Java, Java-script, and BASIC. Figure 3.5 shows a text-based mission specified for a robot to follow a line using Edison software.

3.4.5 Environments with Map-Based Languages

Finally, one environment, FLYAQ, provides a syntax that does not fit into the types of syntaxes reported above. FLYAQ provides the DSL Monitoring Mission Language (MML) to specify missions. By interacting with a map, end-users indicate points of interest, as well as no-fly-zones. The environment automatically generates the mission in an intermediate language, which is

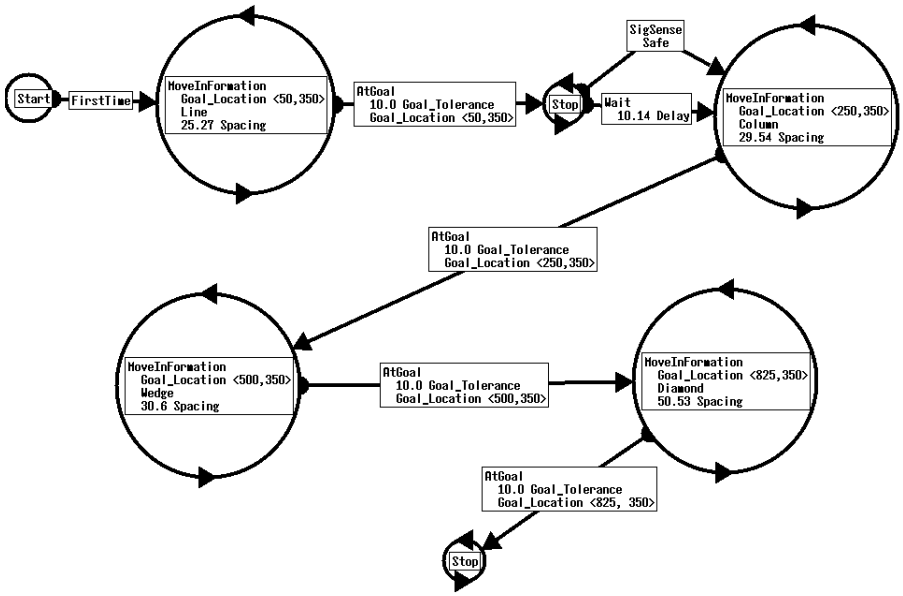


Figure 3.4: Graph-based state transition diagram for a multi-robot scouting mission in MissionLab (from [158])

conceptually close to a flowchart diagram, with a swim lane for each robot. Finally, the mission is executed on real robots or on a simulated environment.

Figure 3.6 shows an example mission specified in MML, where a drone patrols a street to monitor a public event, while taking photos at specified distances, avoiding no-fly zones.

3.5 The Environments' Features (RQ2)

We now present the design space of our subject environments, focusing on their DSLs for specifying robotic missions as well as the environments' capabilities to use these languages. We identified 133 features that distinguish our environments and that we organized in a feature model. In the following, Sect. 3.5.1 presents the *high-level* features extracted from our subject environments; Sect. 3.5.2 presents the *language-specific* features we identified; and Sect. 3.5.3 presents the features related to the *constructs* of the considered languages.

The detailed mapping between each environment and its supported features (a.k.a., feature matrix) is contained in Appendix **??**, in Table A.1 (high-level environment features and language-specific features) and in Table A.2 (language concepts offered by the DSLs).

3.5.1 Specification Environments

Figure 3.7 shows the top-level features characterizing our subject environments: Language, MultiLanguageSupport, Editor, Simulator, Debugging, Specification-

Time, MissionDeployment. The support of these features by each environment is detailed in Table A.1 (upper half) in Appendix **??**.

MultiLanguageSupport.. As a defining characteristic, all our subjects are built around a DSL for mission specification. While we will discuss the languages and their concepts shortly (Sections 3.5.2 and 3.5.3), we observe that all these languages are domain-specific, tailored to the robotics domain. As many as eight environments offer more than one language—either another mission-specification DSL or an off-the-shelf programming language (e.g, Python, C, Javascript) that can be used within the environment. This excludes any library APIs for client applications, as sometimes offered by SDKs associated with the respective environment (e.g., Choregraphe offers APIs for eight different programming languages). When multiple languages were available, the environments typically offered a separate editor for each; there were no facilities for language composition. A notable environment here is PICAXE, which offers a block-based language, a flowchart-based language, and a language in the style of the programming language BASIC with a textual syntax—all of which are individual languages (as opposed to being language language with different syntaxes). However, some environments such as Aseba offer a unique language with different syntaxes.

Editor.. As the main interface to use the respective languages, the editor tooling in our environments offers typical editor capabilities (e.g., copy, paste, or undo). We classify the editing support into EditingMode, SemanticServices, and SyntacticServices features.

Not surprisingly, given the mostly visual syntaxes of our environments, the underlying editing technology (feature EditingMode) is primarily projectional editing (a.k.a., structured or syntax-directed editing) [159, 160]. As opposed to parser-based editing, where the user edits textual code character-by-character, which is then parsed and translated into an abstract syntax tree (AST), in projectional editing, the user's editing gestures modify the AST directly. The AST is projected using projection rules into a user-observable syntax, which can resemble textual and visual syntax, or a combination of both. All of our environments offer projectional editing. 12 of them also come with a parser-based editor to handle the languages with textual syntax—the latter is either an alternative syntax for the visual language or the main syntax of another language offered by our environments. While being the default for editing visual syntax, only once it is also used for textual syntax—in the environment EasyC, displaying visual and textual syntax side-by-side, as shown in Figure 3.8. The typical continuous enforcement of a correct AST in projectional editing guides users towards correct mission specifications, which can also be seen as a semantic service. For instance in Open Roberta, while specifying a mission, the next block can not fit if it is not syntactically correct. In text notations like Edpy in Edison software, projections of next possible text to type are suggested while specifying a mission.

The majority of our environments (26) provide so-called syntactic services (feature SyntacticServices) [120]. These support developers in creating syntactically correct missions, according to the language's syntax. We identified three syntactic services:

- *Visual Highlighting* consists in language-specific syntax coloring of text, or shapes of notation primitives to guide syntax. 25 of the 30 environments

provide syntax-highlighting services.

- *Syntactic completion* suggests a template of concrete syntax primitives (e.g., a code snippet) to the user. Such syntactic completion templates are offered by six environments.
- *Automated formatting* helps in restructuring and layout of the mission being specified. Five of our environments offer this support.

For convenience, seven of our environments offer so-called semantic services (feature SemanticServices) [120]. These support developers in creating semantically correct missions by offering information about mission primitives and how they are used. Semantic services guide the user by providing editing support using:

- *Error marking* highlights mission elements with errors by showing the error message. For instance, a pop-up help displays errors in Edison software, MissionLab, and Choregraphe.
- *Quick-fixes* are proposed solutions, when selected can fix the problem, e.g., interactive tooltips in PICAXE, pop up help and autocomplete in Edpy of Edison software;
- *Reference resolution* links declarations to the usage of the variable, e.g., invalid variable names are pointed out in MiniBloq;
- *Live translation* is the immediate generation code from mission as it is specified, which is displayed side-by-side to the graphical notation e.g., in EasyC Fig. 3.8.

As seen in Table A.1, five of the environments offer semantic services to the end-user.

Simulator.. As many as ten of our environments provide a simulator to test missions in a virtual environment before deployment. Eight of these are limited to simulating single robots, while two, namely FLYAQ and PROMISE, even support multi-robot simulation using off-the-shelf simulators (Mavproxy[4] and Gazebo,[5] respectively).

Debugging.. We identified debugging support in 8 of our environments. Specifically, we found a variety of debugging tools, including the live monitoring of sensor data, of actuator states, and of mission variables. In addition to typical debuggers with stepwise execution, breakpoint support, and stack-trace monitoring. A very typical debugger is contained, for instance, in Robot Mesh Studio. Interestingly, Makecode communicates execution traces via sound and by printing text between the execution of program blocks. Furthermore, Open Roberta provides a 'check box' in the start block that, when checked, displays current values of the connected sensor data during program execution.

SpecificationTime.. Missions are specified either at design time or run-time. Design-time specification provides all the details about the mission before the execution starts. All environments support design-time specification. Three of our environments (namely Turtlebot3-blockly, Sphero, and Choregraphe),

[4]https://ardupilot.org/mavproxy
[5]http://gazebosim.org

however, also offer some remote-control functionality to intercept the mission execution at runtime.

MissionDeployment.. Missions, once specified, are deployed to the robots for execution. We identified three features related to mission deployment:

- *Over the air.* Supported by 10 environments, we identified the WiFi and Bluetooth connections as wireless options used for deploying missions to robots.
- *Via cable.* Supported by 23 environments, the cable options for mission deployment observed are USB cable, Ethernet cable, and custom cables specific to some robots.
- *Runtime redeployment.* Two of our environments support re-deploying a modified mission at runtime, i.e., when the previously specified mission was already started, without restarting the robot.

3.5.2 General Language Characteristics

As illustrated in the feature model in Fig. 3.9 and the feature matrix in Table A.1 (Appendix **??**), we identified the following general characteristics in which the languages differ, represented by the features Notation, SemanticsRealization, LanguageParadigm, and Extensibility. The actual concepts offered by the languages will be discussed in Sect. 3.5.3. Below, we discuss their core characteristics.

Notation.. As already discussed above in Sect. 3.4, all our environments offer languages with a domain-specific visual notation, which forms the concrete syntax for their end-users. The textual and visual notations offered by the respective environments are summarized in Table 3.3. According to definitions provided in Sect. 3.4, we classified the considered languages as block-based (in 24 environments), flowchart-based (in 3 environments), graph-based (in 14 environments), map-based (in one environment), and text-based (in 13 environments).

Almost every syntax is customized with robotics-domain-specific visual symbols. For instance, a block *Motor forward* in TRIK Studio has a gear icon with a forward arrow depicting a forward-running motor. The user only specifies the motor power and the port in which the motor connects. As many as 13 of our environments additionally offer a textual syntax, often obtained by allowing the use of a general-purpose programming language to be used as an alternative to the main DSL. Some environments use a mix of textual and visual notations, like EasyC, as shown in Fig. 3.8.

SemanticsRealization.. The semantics of our languages are realized by either interpretation (in two environments) or compilation, i.e., generation of code in a target language (in 28 of our environments). The mission is either semantically translated (compiled) as shown in Table 3.4 or executed by an interpreter. LEGO Mindstorms EV3 and Code Lab interpret the visual mission directly during execution. Metabot directly generates assembler code, while the rest compiles generated code. TRIK Studio supports multiple robots (Lego EV3, Lego NXT, Pioneer Kit, and the TRIK robot), while it does not cross-compile,

since missions are robot specific, it generates code in various target languages, including C, JavaScript, Pascal, Python, and F#.

LanguageParadigm.. While all our environments come with a DSL for mission specification, nine of them also support a general-purpose programming languages (GPLs) usable directly in the environment. Examples of the latter are C/C++, Java, and Python, as shown in Table 3.3. The DSLs all provide language concepts related to the robotics domain.

Extensibility.. Some environments provide features for extending the language with new concepts, which we classified into ScriptingSupport (12) and AddLanguageConcepts (16). ScriptingSupport allows the creation and launching of new language constructs to extend the existing language. For instance, Choregraphe allows users to write new scripts for defining action boxes for the NAO robot. AddLanguageConcepts allows users to edit and create new blocks. For example, LEGO Mindstorms EV3 allows importing custom blocks from vendors that manufacture sensing blocks compatible with the Lego Mindstorms EV3 robot.

3.5.3 Language Concepts

We found a range of different concepts offered by the languages for specifying missions. We consider a concept as a distinct element of the abstract syntax of the language. We focus on concepts that are recognizable via the notation (concrete syntax), since many of our environments are not open-source, a look at the exact implementation of the language's abstract syntaxes is not possible. End-users observe these concepts via the language's notation and utilize them via the respective projectional editor, or in a parser-based editor for the the textual languages available in some environments. As shown in Fig. 3.10 and Table A.2, we classified the concepts into the following features: MissionSpecificationParadigm, ControlFlow, Modularity, DataTypes, EventSupport, ReadSensor, Actions, ExceptionHandling, FileAccess, FunctionLibrary, Multithreading, and MultiRobotHardwareSupport. Below we show the details on the respective language concepts.

MissionSpecificationParadigm.. One of the core distinguishing characteristics is the kind of mission specification paradigm supported by the languages, which determines how missions are executed. We found three major kinds: imperative paradigm (28), and reactive paradigm (3). The former two are relatively low-level; none of our environments supports goal-based paradigm.

In imperative execution, the mission runs in the order of sequence of the tasks specified, which is the common paradigm for most of the environments in our study. In contrast, Aseba features a reactive paradigm in which events, which act as triggers are matched with corresponding actions. See Fig. 3.11 for an example. In some of the imperative environments, such as PICAXE, reactive aspects are also realized, where the robot can be instructed to respond to sensor data during mission execution.

ControlFlow.. 29 languages offer several kinds of control-flow statements. Typical examples of conditionals we found are `if-do`, `if`, `if-else`, and `switch`. Loops are also common (27), represented by statements such as `do- while`, `while`, `forever`, `repeat while`, `repeat count`, and `repeat until`. The latter two are shown for LEGO Mindstorms EV3 in Fig. 3.12 (`repeat count`)

and in RobotC (`repeat until`). Finally, execution interrupts are provided (20), such as for loops with `loop interrupt` in LEGO Mindstorms EV3 and `stop all` in Tello Edu App, and for general execution using `wait (time/event)` in Makeblock 5.

Multi-threading controls are also found in TRIK Studio (`fork`, `join`, and `kill thread`), in LEGO Mindstorms EV3 for running tasks simultaneously (`sequence plug exit`) and in Robot Mesh Studio, for example in blockly, `start block` creates a thread, `sleep for x seconds` forces a thread to yield. See feature Multithreading below for more information.

Modularity.. The majority of the environments (17/30) offer modularization concepts to structure larger missions. We found functions, which are graphically represented using dedicated blocks, in the environments called functions or procedures, or modules. Each function gets input parameters and (often) return values. This represents a relatively basic, but pragmatic modularity mechanism, which the non-technical end-users of the environments can utilize. Environments with modularity features include: Metabot, Ardublockly, Open Roberta, Choregraphe, Sphero, Robot Mesh Studio, Metabot, Makeblock 5, Ozoblockly, and Turtlebot3-blockly, which create mission modules using functions and function calls. Choregraphe implements robot behaviors as boxes, which are connected in a flow chart to form a mission. LEGO Mindstorms EV3 imports blocks from external environments that are compatible with LEGO Mindstorms EV3. TRIK Studio implements subprograms, functions, and module with symbolic icon of what these components do; however, these program modules do not have information on return values and scoping information. PICAXE implements procedures of particular concepts, which then can be invoked and used.

DataTypes.. The environment's main languages offer dedicated data types for variables or functions, comprising primitive (25) and compound (25) types. Only FLYAQ, MissionLab, TiViPE, and TRIK Studio do not have exclusive variable data types. The primitive types we found include: integer, decimal, character, float, number, and Boolean. Compound types include string, arrays, table, and lists. Not surprisingly, we also found domain-specific types, such as sound in Ozoblockly, degrees in Tello Edu App, and color in Sphero. The environment LEGO Mindstorms EV3 calls the Boolean type logic, apparently also for enhancing comprehension by end-users. In Aseba, state is a type, which is essentially an enum (e.g., a state variable temperature can take the values off, low, medium or high).

FunctionLibrary.. Almost all of the languages come with function libraries that offer typical arithmetic and logic (23), and string operations (10) on data, but also complex algorithms (10) used to process data. Essentially, we found full range of functions one would expect, including logical operators (e.g., conjunction, disjunction, negation), mathematics functions for trigonometric calculations, rounding, aggregation, and so on. String operations include create, append, build string, length, substring, while list operations include find, sublist, isEmpty, join, and so on.

Actions.. Every language provides statements representing actions. These are activities that robots execute to achieve a given task. Some are reactions to events, while others are activities that are imperatively specified in the mission.

The first distinguishing characteristic we found is the action type, as shown in Fig. 3.13 and Table A.2. Specifically, actions can be of type:

- Instantaneous action (14), which is executed immediately and only once, such as take photo in FLYAQ.

- Continuous action (14), which executes immediately for an infinite amount of time or a fixed time, for instance, "random eyes duration ms" in Open Roberta changes the NAO robot's eye colors for a specified duration in milliseconds, or initiated and stopped by events, e.g. user interaction. Another example is follow line in LEGO Mindstorms EV3 and Sphero, or record a video in FLYAQ.

- Delayed action (19), which starts after a delay, which can be due to an event or specified time to wait

Most environments support instantaneous actions. The others typically require some notion of time and timer manipulation, where we found different realizations. In LEGO Mindstorms EV3 the block "get value ms timer" is used as a sensor to read the current time for another block, while the block "reset timer" is used to reset the internal timer to zero. Similar time constructs also exist in other environment, such as the blocks "wait time" and "elapsed time" in Ardublockly, the statements "set roll time" in seconds as a variable, "time elapsed", "get current time", "set timeout", "set time interval" in Sphero, and finally the statements "set timer (seconds)", "timer", "wait (seconds)" in VEX Coding Studio.

Then, the environments typically realize concrete actions as dedicated language concepts, which sometimes result in relatively large languages. We further classified the actions into: actuation, communication, and movement actions as explained below.

Actuation (28). This refers to device/actuator-related activities, such as grasp an object, motor movements, or play audio. More examples include get button code, play tones, sound, stop motors, clear encoder, angular servo, turn on LED, detection with video camera, line detector, video streaming enabling, beep note, buzzer, display, status light, object observation, taping, face status (smile, frown), among others. This illustrates that the languages can become relatively large when (as for most languages) no library mechanism is provided.

CommunicationActions. This includes interacting with humans (8) or other agents(13). Communication with humans can be of the form: text, video or audio. Non-human agents can be categorized as tuple space, publish-subscribe or message passing. Tuple space is a shared space where shared data items are kept for access to entities entitled to access them. In publish-subscribe the publishers create messages regardless of receivers, while subscribers receive messages they have subscribed to. Communication examples include infrared messages exchanged among robots in Edison software, LEGO Mindstorms EV3, and Open Roberta, and Bluetooth messages exchanged among robots in LEGO Mindstorms EV3. In MissionLab, robots can share information about target goal position and map of the environment among each other directly or through broadcasts. Sphero, VEX Coding Studio, Makeblock 5, and Tello Edu App broadcast message between robots. FLYAQ supports synchronization and communication messaging among drones at run time. TRIK Studio supports sending messages to other robots. For what concerns human and central system to robot communications, examples are hand-clap and touch

in Aseba, and Enchanting, send text, send number, send command, receive character, send character, send, and receive message in Tello Edu App, speak short phrase in Code Lab, say text in TRIK Studio, TiViPE from the robot to the environment. In Choregraphe, a robot can share text with humans. In Arcbotics' SparkiDuino, humans interact with robot through beep, status led colors, infrared remote code and PICAXE communicates through infrared messages. Eleven of the 30 environments do not offer any communication language constructs.

MovementActions. Languages offer concepts that specify how a robot moves from one location to another. Such concepts are either absolute, e.g. "map coordinates", or relative, e.g. direction, distance or travel time.

Few of the environments support absolute movement actions, such as goto (coordinates) in FLYAQ, MissionLab, and Makeblock 5; moveto (coordinates), movefast (coordinates, duration) in TiViPE; roll (angle, speed, time), spin (angle, time in seconds) in Sphero; drive (distance) in Code Lab. For relative movements we mention go/move/drive/turn/fly (forward/backward/left/right/room) as seen in most of the environments.

EventSupport.. 24 of our languages provide event support, which concerns the languages' abilities to handle events like creating event handlers; the types of events that can be recognized; and the mechanisms of synchronizing events to subsequent actions. For instance, in VEX Coding Studio, the common language constructs for event support identified include: when (event), when (event) do, wait for (event), wait until (event), wait (event), on (event), capture (event), move until (event), broadcast and wait (event). MissionLab has even more domain-specific events, such as AtGoal or AtEndOfHall. The events are sensory data that trigger the next robot action. However, in the environments Arcbotics' SparkiDuino, TiViPE, MiniBloq, Turtlebot3-blockly, and RobotC, we did not recognize event support in their languages.

ReadSensor.. All of the considered environments provide dedicated concepts for reading sensor data. These are used to record data through sensor readings. The sensor concepts identified are shown in Fig. 3.14 with summary of sensor usage in Table 3.5. We classified the sensors into: tactile sensors, wheelMotor sensors, heading sensors, and other measurements. Tactile sensors measure by direct contact e.g., touch sensor, fingerprint sensor, infrared, line detector, brick button, light (on/off detection, intensity), and distance proximity sensors (ultrasonic, sonar, infrared proximity sensor, obstacle/object detector, distance sensor).

Movement sensors measure the actual robot movement via, as we identified, a motor rotation sensor, a gear potentiometer, a magnetometer, an accelerometer or a rotation detector. Heading sensors guide direction and orientation of the mobile robot e.g., gyro, vision/seeing sensors and compass sensor. The gesture sensors measure body gestures such as speech recognizer, face detector, and marks sensor (color). Other sensors which were not readily classified include: GPS module, energy meter (reads battery energy), temperature/thermometer, barometer, sound detector, encoder, timer, power meter[6] (watts), and analog sensor.

ExceptionHandling.. We identified exception handling constructs in Open

[6]e.g., https://www.generationrobots.com/en/401216-mesureur-de-puissance-pour-nxt.html

Roberta, Choregraphe, MissionLab, PROMISE, and Code Lab environments, particularly in their textual languages but not their primary, visual DSLs. More specifically, Open Roberta exploits the Python exception handler. Choregraphe exploits the try/catch block for all errors in its C++ software development kit (SDK), and the try/catch block for face detection error in its Python SDK. Code Lab offers support for exception handling for very specific error (e.g., action error, animations not loaded, cannot place objects on this, connection aborted).

FileAccess.. Eight of our visual languages provide concepts for file access. For example, LEGO Mindstorms EV3 provides blocks for reading and writing data to the local storage, and to close or delete a file. Such files can record, for instance, ambient light measurements taken at given time intervals.

Multithreading.. Eleven of our environments provide support for concurrency. Multithreading allows users to do several activities without waiting for one activity to end, improving the performance of executions. In Robot Mesh Studio, using the blockly editor, the *start* block creates a thread, *sleep* for x seconds forces the thread to *yield*, *start autonomous* creates a thread that runs the autonomous mode of the robot, and *start driver* creates a thread that runs a driver. Recall that Robot Mesh Studio also supports various textual general-purpose languages (cf. Table 3.3), where the typical multi-threading concepts can be used. For instance, in Python concepts like sys.run_in_thread(f), sys.thread_id(), sys.sleep(t) are offered. In C++, thread (void (*callback)(void)), get_id(), join(), interrupt(), yield(), sleep_for(unit32_t time_ms), lock(), try_lock(), unlock() are used. TRIK Studio offers fork, join, kill thread, and send message to thread. Furthermore, LEGO Mindstorms EV3 offers dedicated blocks for creating parallel tasks, as well as Makecode and RobotC. TiViPE offers *splitSCIS* to split and run modules in parallel. MissionLab supports Cthread, which is a lightweight process threads package that operates under Unix-style operating systems. Also PICAXE supports multi-tasking with operations such as restart, resume, and suspend.

MultiRobotHardwareSupport.. Eleven of our environments support more than one robot hardware platform. Missions in these environments are specified for a particular robot hardware, making it impossible for hardware-independent missions. However, in Sphero, there are some missions which are compatible with more than one robot, though we did not observe a single mission that runs in all the Sphero robot varieties. The aspect of robot independent missions therefore remains a dream to be achieved by roboticists and language engineers.

By traversing through the selected environments, it is evident that, most of the environments have common features, other environments present peculiar features, specific to a given aspect of either robotics domain or end-user domains.

3.6 Discussion

In this section we discuss the practical implications of the findings, intelligence, and collaborative multi-robots.

The practical implications.. The practical implications of the findings of our study are:

- As we observed, most of the languages are targeting toy robots used mainly for teaching. If we want to use these languages in domains like agriculture, healthcare, etc., there is the need of extending the visual programming concepts beyond toy robots.

- The research community and language tool developers have a benchmark to better develop future tools by using the feature space explored by this survey.

Intelligence.. Robot system intelligence is characterised by its ability to automatically act without human intervention. This can be triggered by events from the environment as captured by sensors, timed executions or learning from past experience. The feature model presents concepts such as event support in Fig. 3.10, delayed action type in Fig. 3.13 and reading sensor data in Fig. 3.14 and Table 3.5, which can facilitate intelligence in the robot systems. In TiViPE and choregraphe, the NAO robot senses soundata and intelligently determines the direction the sound. Picture frames are captured by NAO robot at intervals to determine a moving object by the NAO robot. NAO robot tracks known people by comparing all people in a video with known people, hence automatically identifying unknown people. In Open Roberta and Choregraphe, NAO robot recognizes predefined words and phrases in different languages. Data from vision sensor in VEX Coding Studio robots can be used to track up to seven individual colours at once, analyse objects for advanced tracking and path planning.

Collaborative multi-robots.. Environment such as FLYAQ, PROMISE and MissionLab support collaborative mission specification. FLYAQ [36] provides for specification of missions for multiple drones. The end-user explicitly sequences the tasks for each robot, with location details, hence avoiding collisions. Some of the mission primitive used include: *Takeoff, Goto(location), DoPhot, Land*. Since distribution of tasks to drones is done manually, there are no language concepts to express multi-robot mission specifications as shown in Fig. 3.6. García in PROMISE [161] proposes visual mission specification environment for multi-robots, however the decomposition of the mission to local missions for each robot is also done manually. The parallel operator — *parallelOp* takes robots as inputs and assigns a robot to each branch (each child) We therefore observe that multi-robot mission specification and task distribution is not trivial.

Mission Composition. This feature explores classification of mission components. We classify the mission composition into two subfeatures, i.e. HorizontalCompositionMechanism and Vertical Composition[7]. HorizontalCompositionMechanism involves task level decomposition. For instance, in Aseba, actions and events in the visual programming language are horizontal mission components. To specify a task, user matches events to related actions in event-action paring e.g., Detect object – set top color red as shown in Fig. 3.11. In Open Roberta, horizontal decomposition can be conceived from how the language concepts are categorized, for instance, actions, sensor, control, logic, math, text, color, and variables. Vertical Composition can be viewed from

[7] https://robmosys.eu/wiki/general_principles:separation_of_levels_and_separation_of_concerns

robotic behavior coordination abstraction levels such as, (i) Patrolling (mission); (ii) steering task (move repeatedly) or sensing task (task); and (iii) drive forward or detect object/line (skill). In the vertical hierarchy, mission is the most abstract while skill is at the lower level of abstraction closer to the robot hardware.

Beyond Language-Based Mission Specification.. Gorostiza et al. [162] propose a natural programming environment in which robot skills are accessed verbally to interact with end-users. The environment uses dialog system to extract actions and conditions to create a sequence function chart. The challenge with this approach is, the end-user cannot add new dialogue constructs for new tasks, making the languages inflexible.

Miguel Campusano et al. [163] in their work on "live robot programming" implement a language that supports live feedback. This language helps end-users in rapid creation and variation of robot behavior at run-time. This approach however does not provide end-user with domain constructs to simplify the programming effort during mission specification.

Finally, Doherty et al [108] propose a framework and architecture for the automated specification, generation, and execution of missions for multiple drones that collaborate with humans. The focus of the study is on how the language can clearly and concisely specify and generate missions, but not on how the language is easy for end-users.

3.7 Practical Implications of our Findings

To evaluate the results of the survey, we define one usage scenario for each of our end-users: a teacher, a robotic manufacturer, and a language engineer.

3.7.1 End-User—Teacher

A teacher has to instruct a robot development course to students with limited background on programming languages. The teacher has to select the robotic mission specification environment to be used in her course. The choice is based on the following requirements:

[a] *Simulation support* — The robotic mission specification environment shall provide simulation support for simulating the mission execution and support for deploying the mission on the actual physical robots;

[b] *Language control flows* —The robotic mission specification environment shall provide support for specifying sequences of tasks repeated until a certain condition holds (loop statements) and executing alternative tasks depending on some conditions (conditional statements);

[c] *Actions* — The robotic mission specification environment shall allow users to specify movement actions and the use of the speakers of a robot to provide audio messages;

[d] *Run-time environment* — The robotic mission specification environment shall be executable both on a web interface (for quick mission prototype) and as a stand-alone application.

Within this scenario, to select the mission specification environment to be used in her course, the teacher uses the results of this survey as follows:

[a] *Simulation support* — The teacher consults the results presented in Table A.1 and checks for the environments that provide simulation support. Among these environments, Aseba, FLYAQ, Makecode, Metabot, PICAXE, Robot Mesh Studio, MissionLab, Open Roberta, RobotC, and TRIK Studio are providing simulation support, and can be used within the course;

[b] *Language control flows* — The teacher consults the results presented in Table A.2 checks for the environments that provide loop and conditional statements. Among these environments, 26 of the 30 environments are providing loop and conditional statements, and can be used within the course;

[c] *Actions* — The teacher consults the results presented in Table 3.13 and checks for the environments that provide support for the specification of movement actions and the use of the speakers of a robot to provide audio messages. All the environments offer movement actions since this survey covered mobile robots. MissionLab, Code Lab, Open Roberta, and TiViPE provide audio messaging feature, which can be used within the course;

[d] *Run-time environment* — The teacher consults the results presented in Table 3.2 and checks for environments that provide both the web interface and can be executed as stand-alone applications. Among these environments, Open Roberta, FLYAQ, Robot Mesh Studio and Sphero provide these features, and can be used within the course.

Based on the results of all these steps, the teacher finally selects Open Roberta as mission specification environment to be used during the course. Indeed, this environment meets all of the teacher's requirements.

3.7.2 End-user — Robot Manufacturers

Robotic engineering has become flexible since most engineers do ensemble of existing parts. It is handy for such engineer to check environments with the features that the robot should have. For instance, a robot manufacturer is interested in creating a new robot, which can move on land, recognize objects and sound within the environment. The purpose of the robot is to aid in learning programming and research.

The robot manufacturer has the following requirements to be fulfilled:

[a] *Robot mobility requirements e.g., motor, steering* — The robot manufacturer needs to know the existing movement features in mobile robots for benchmarking;

[b] *Sensor requirements e.g., proximity, vision, sound* — There is need to know which sensors exist with language constructs;

[c] *Robot action requirements e.g., movement actions, instantaneous actions relative actions and actuations* — There is need to know which robot actions a new robot can execute.

The robot manufacturer can be guided as follows:

[a] *Mobile robots* — Table 3.2 contains list of mobile robot such as VEX robots, PICAXE, LEGO robots. By further profiling the manufacturer specifications of such robots a robot manufacturer can takes informed decision on what mobility features he/she can incorporate in the new robot.

[b] *Sensor requirements* — A rich collection of sensor constructs for capturing data from the environment are shown in Fig. 3.14, and usage of sensors by the environments is summarised in Table 3.5. The environments having particular sensors can be obtained from the appendix – "List of URL for each of the environments" For instance VEX robots use proximity sensors, vision sensor, accelerometer sensor and motor sensor

[c] *robot actions* — Fig. 3.13, and Table A.2 can guide the manufacturer to analyse variety of action, which the robot can execute. Actions such as movement, instantaneous actions, relative actions, delayed actions and actuations can be performed by VEX robots.

The robot manufacturer can use VEX robotics. For instance VEX robots are open and can be programmed by a number of environments such as VEX Coding Studio, EasyC, Robot Mesh Studio and RobotC. It is therefore wise for a robot manufacturer to make a new robot using VEX products since the new robot can be programmed using a number of environments hence attracting more users.

3.7.3 End-user — Language Developer

A language engineer wants to develop a language that supports mission specification. The new language will target children younger than seven years. The language developer wants to understand the features provided by similar languages to determine which languages are providing features that are relevant for this class of users. The language developer has the following requirements:

[a] *Editing mode* — The language should provide projectional editing mode, which guides on next step to take while editing.

[b] *Notation* — The robotic mission specification environment shall provide a visual language based on blocks and connections since users can barely read and write;

[c] *Simulation support* — The robotic mission specification environment shall provide simulation support to allow children to play and simulate the behavior of the robots when executing different missions;

[d] *Language control flows* — Choice of control flow from available options such as loops, conditional and interrupts is also required

[e] *Actions* — The engineer wants to provide children the possibility to specify complex movement actions, such as making the robot dancing. Furthermore, children should be allowed to communicate with the robots.

[f] *Run-time environment* — The language developers aim at developing a mission specification environment that is executable as a stand-alone application.

Within this scenario, the language developer uses the results of this survey as follows:

[a] *Editing mode* — the engineer considers languages with projectional editing modes. Table A.1 shows that all the environments offer projectional editing mode except TiViPE.

[b] *Notation* — Section 3.4 provides various syntax for each or the environments to guide a suitable one e.g. block-based, graph-based, flowchart-based or map-based. Block-based environments such as Arcbotics' SparkiDuino, Ardublockly, Aseba, BlocklyPro, Edison software and LEGO Mindstorms EV3 fit the user requirement. All environments which support blockbased syntax can be found Table 3.2.

[c] *Simulation support* — The language engineer consults the results presented in Table A.1 and checks for the environments that provide simulation support. Among these environments, Aseba, FLYAQ, Makecode, Metabot, PICAXE, Robot Mesh Studio, MissionLab, Open Roberta, RobotC, PROMISE, and TRIK Studio are providing simulation support.

[d] *Language control flows* — Table A.1 can guide on the available control flow concepts that exist and the environments which already have them. All environments offer loop control flows except Choregraphe, FLYAQ, and MissionLab. 21 of the 30 environments also offer conditional control flows such as if, if-else and selection, while 21 of the environments offer interrupt controls. TRIK Studio provides multithreading fork control flow support.

[e] *Actions* — Language concepts summarized in Table A.2 and the read sensors described in Table 3.5 can help the language engineer to identify the language concepts required to develop the actions that need to be incorporated in the new language. All the environments support movement actions, for communication with agents; language engineers can explore environments such as Arcbotics' SparkiDuino, BlocklyPro, Edison software, FLYAQ, LEGO Mindstorms EV3, Makeblock 5, Sphero, and Tello Edu App.

[f] *Run-time environment* — Using Table 3.2, the engineer can determine features of stand-alone environments. Most of the environments offer support for stand-alone installations except Edison software, Makecode, Marty software, Metabot, Ozoblockly, and Scratch Ev3.

3.8 Threats to Validity

Internal Validity.. The manual process of collecting and classifying the features is subject to biases. We mitigated this effect by distributing environments among the authors to collect features and allow one author to verify the features collected by another, followed by discussions to reconcile any differences on views. This made the data collection rigorous and thorough. Secondly, the Google search engine returns different results to different people on the same search due to personalized search behavior customized by Google. Therefore, if anyone else did the same search, the results would not necessarily be the same. This has been resolved by relying on multiple sources of data as well as searching for robots and then try to identify any environment they are shipped with, and by snowballing. Another threat is the fact that the total number of results returned is greater than the actual number. For instance, our search result returned 774,000 results, but when we scanned all the results, the last page only reported 373. However, this is not a limitation since we used different sources of information and, as can be seen in Table 3.1, all the results from Google search were also captured by other data sources, including authors' experience, list of mobile robots, snowballing, and alternative environments for the robots, with exception of BlocklyPro.

External Validity.. Extracting the features using independent data collection based on documentation available in public domain is a thread to external validity. Contacting the developers of the tool would have provided more information and allowed to detect more features. However, this has been countered by the fact that the considered environments are significantly different among each other. As these tools try to cover user needs from different angles, features that are hard to identify in one type of environment are usually key and easily identifiable features in a different environment. Secondly, there is diversity in phrases and terms used to describe mission specification. Since we observed that different authors refer to mission specification by using different terminologies, we constructed a search string comprising of a number of phrases as seen in (cf. Sect. 3.3.1).

3.9 Related Work

Bravo et al. [117] review intuitive robot programming environments for education. They categorize their languages into textual, visual, and tangible. However, they do not discuss individual language features that facilitate end-user programming, as we do.

Biggs et al. [17] survey robot programming systems, which they classify into manual and automatic. The manual systems require users to specify missions, while the automatic ones control robots based on their interactions with the environment, indicating that such missions are specified on a higher level, for instance, by declaring the mission goals instead of the concrete movements. However, the survey did not discuss language features that enhance robot programming by novice programmers.

Ray et al. [164] survey user expectancies from robots. They find that, at the personal level, users expect support with household daily tasks, security,

entertainment, and company (child, animal, or elderly care). More than half of them expect robots providing such services to be in the market soon. These findings imply raising mission specification to higher levels of expressiveness closer to the end-user domains.

Abdelfetah Hentout et al. [59] survey development environments for robotics. They identify frameworks for programming robotic system, but not targeting mission specification.

Jost et al. [90] in their review of 10 visual programming environments for educational robots, discuss advantages of visual over textual environments, in order to present eh Open Roberta project. They do not analyze any of the existing environments to the extent we do.

Luckcuck et al.'s survey [14] identifies challenges, formalisms, and formal methods for specifying and verifying autonomous robot missions. For instance, it covers KLAIM, a formal language used to capture properties about distributed systems. The survey has little to do with the features to support end-user programming or features expected to support visual specification.

Nordmann et al.'s [13,165] survey on DSLs for robotics identifies a large number of languages. Surprisingly, none of the languages supports mission specification, which makes their work distinct from our study. Specifically, the survey covers aspects of environmental features and constraints, which are expressed using formalisms such as LTL, OWL, and (E)BNF. Scenario definitions are made using formalisms such as ANTLR grammars, (E)BNF, UML/MOF, LTL, or Ecore. These formalisms are suitable for robotic and software engineers, but not novice end-users. This gap also motivates our study.

3.10 Conclusion

Mobile robot systems have become general purpose in terms of the number of actuators and tasks which they can execute. As such, it is not realistic having these robots hard-coded at manufacturing time. It is also unrealistic to keep relying on robotic and software engineers to always be the ones to program the robots. With increasing presence of robots in aspects of everyday life, more research is being done to enable end-users to program and specify missions for robots. While evaluating issues for visual programming languages, Menzies [166] argues that visual environments motivate students as they lessen the burden of memorizing a lot of syntax in textual languages such as C++ and Java. However, at the best of our knowledge, there is no organized literature on related features in such environments and languages.

In our survey, we have studied the language design space of 23 specification environments and extracted mandatory and optional features, which these environments offer to support end-user programming of robots in aspects of everyday life. We present this as a feature model and further analyze how the environments differ from each other. In summary, we found many typical constructs (e.g., control-flow statements) from general-purpose languages, provided using visual syntax. In addition to the primary visual DSLs supported by the environments, many – often general-purpose languages – provide alternative textual syntax to complement the visual DSLs when they are not expressive enough. While all these visual languages come with a projectional

editor, we also found environments that they provide a textual notation projected right next to the visual one. The majority of our environments also use the Blockly or Scratch library, both of which have significantly eased the development of visual syntax.

We also identified some challenges to be addressed by the next generation of languages:

- Actions are often very concrete and every action has its own language concept; on one side this is unavoidable, on the other side it would be beneficial to find a way to categorize or organize the various possible actions in groups so to facilitate their definition, management, and treatment.
- The language concepts for actions found in the environments are basic, and, consequently, they cannot sufficiently express what end-users, such as farmers and nurses, can use. It is important that more vibrant library of controllers to specify behaviors with well-defined semantics are built to facilitate real-life mission specifications for end-users.
- In general, the languages we surveyed have a rather low-level of abstraction. In most of the cases the mission specifier is required to model in detail the behavior that the robots should perform to achieve the mission. This has some disadvantages:
  - (i) it is error-prone and the mission specifier should know details about the language concepts used, which are not standard and in most cases biased to the robotics domain;
  - (ii) it is difficult to estimate the partial satisfaction of the mission that is needed when re-planning is required by some changes in the execution environment or in the mission specification itself; and
  - (iii) it requires knowledge and expertise potential end-users will not necessarily have. Goal-based and declarative mission specification languages look more promising and attractive.

As future work, we plan to study the syntax of these languages in more detail, aiming to understand what are the best ways of presenting the mission-specification concepts our surveyed languages are offering. A possible route is to assess the syntax with respect to Moody's notational design principles [167]. Furthermore, a user study can validate the need for certain features as well as recover needs not realized so far. Especially eliciting user experiences with different kinds of decomposition mechanisms for missions would be valuable to inform the design of future mission-specification languages. We also plan to establish how the mission-specification languages are used and perceived, for instance, what concepts are used frequently, and in what combination. We hope to eventually build the next generation of languages upon these empirical results, also lifting the language to higher levels, perhaps offering different language profiles.

```
#define PROGRAM_NAME "NEPOprog"
#define WHEEL_DIAMETER 5.6
#define TRACK_WIDTH 18.0
#include <ev3.h>
#include <math.h>
#include <list>
#include "NEPODefs.h"
int main () {
  NEPOInitEV3();
  NEPOSetAllSensors(NULL, NULL, EV3Color, NULL);
  while ( true ) {
    if ( ReadEV3ColorSensor(IN_3) == White ) {
      SteerDriveForDistance(OUT_C, OUT_B, Speed(100), Speed(30), 1);
    } else {
      SteerDriveForDistance(OUT_C, OUT_B, Speed(30), Speed(100), 1);
    }
  }
  NEPOFreeEV3();
  return 0;
}
```

Listing 3.1: Target C code generated from the mission in Fig. 3.1

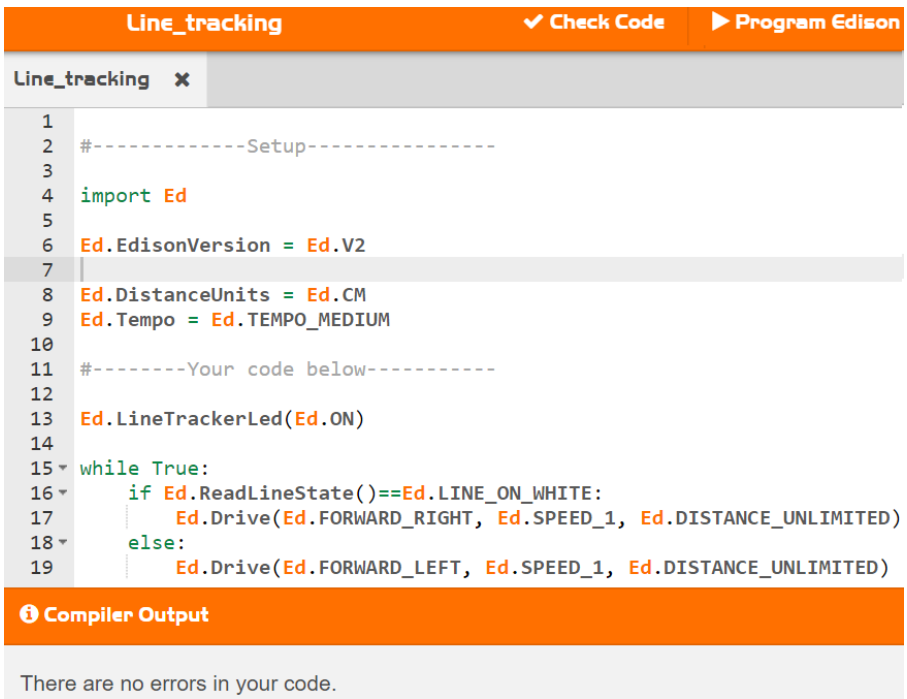


Figure 3.5: A text-based mission for line tracing specified in Python within the environment Edison software (from [129])

Table 3.2: Subject environments and their characteristics

Environment	Syntax	Runtime environment	Mobile robots supported	User domain
Arcbotics' SparkiDuino 1.8.7.1 [122]	Block-based, text-based	Desktop	Sparki	Education
Ardublockly 2.4.220 [123,124]	Block-based	Desktop	Spartan	Education
Aseba 3 [92,93]	Block-based, text-based	Desktop	Thymio II	Education
BlocklyPro 1.1.1.455 [125]	Block-based	Desktop	ActivityBot, Scribbler 3 Robot	Education
Choregraphe 2.1 [87, 88,126]	Graph-based	Desktop	NAO, Romeo	Education
Code Lab [127]	Block-based, text-based	Desktop, mobile-app	COZMO	Education
EasyC 5 [128]	flowchart-based	Desktop	VEX EDR & VEX IQ	Education
Edison software [129, 130]	Block-based, text-based	Web	Edison robot	Education
Enchanting 0.2.4.3 [131,132]	Block-based	Desktop	LEGO Mindstorms NXT	Education
FLYAQ [34–36]	Custom map-based	Desktop, web	Parrot AR drone	Education, research
LEGO Mindstorms EV3 1.3.1 [94,95]	Block-based, text-based	Desktop, mobile-app	LEGO Mindstorms EV3	Education
Makeblock 5 [133, 134]	Block-based, text-based	Desktop, web	Codey rocky, mbot, Airblock	Education
Makecode 1.0.11 [135]	Block-based	Web	LEGO Mindstorms EV3	Education
Marty software 3.0 [136]	Block-based, text-based	Web	Marty	Education
Metabot [137,138]	Block-based	Web	Metabot	Education
Ozoblockly [139,140]	Block-based	Web	Bit, Evo	Education
PICAXE 6 [70,71]	Block-based, flowchart-based, text-based	Desktop, mobile-app	PICAXE 20X2 Microbot	Education
Robot Mesh Studio 2.0.0.6 [67]	Block-based, flowchart-based, text-based	Desktop, web,	VEX IQ, VEX EDR, VEX V5	Education
Scratch Ev3 [141,142]	Block-based	Web	LEGO Mindstorms EV3, WeDo 2.0	Education
Sphero 5.2.0 [143,144]	Block-based, text-based	Desktop, web, mobile-app	Sphero Bolt, Spark+, Sphero Mini	Education
Tello Edu App 1.1.2.23 [145,146]	Block-based	Mobile-app	Tello drone	Education
TiViPE 2.1.3 [147,148]	Graph-based	Desktop	NAO	Education, research
Turtlebot3-blockly [149,150]	Block-based	Desktop	TurtleBot3	Education, research
VEX Coding Studio 18.08.2010.100 [68,69]	Block-based, text-based	Desktop	VEX IQ, VEX EDR	Education, research
MiniBloq 0.83 [151, 152]	Block-based	Desktop	DuinoBot, Sparki	Education
MissionLab 7.0 [83, 96]	Graph-based	Desktop	ATRV-jr, Urban robot, Amigobot, Pioneer AT, Nomad 150,and 200	Education, research
Open Roberta 3.0.3 [89–91]	Block-based	Desktop, web	Micro:bit, LEGO Mindstorms EV3 and NXT, NAO, WeDo, BoB3, Nepo4Aduino, Bot'n Roll, calliope mini	Education
RobotC 4 [65,66]	Block-based, text-based	Desktop	VEX IQ, VEX CORTEX, LEGO Mindstorms EV3 and NXT	Education
TRIK Studio 3.2.0 [153,154]	Graph-based, text-based	Desktop	LEGO Mindstorms EV3 and NXT	Education
PROMISE [8]	Graph-based, text-based	Desktop	TIAGo, ITA, Turtlebot2	Research

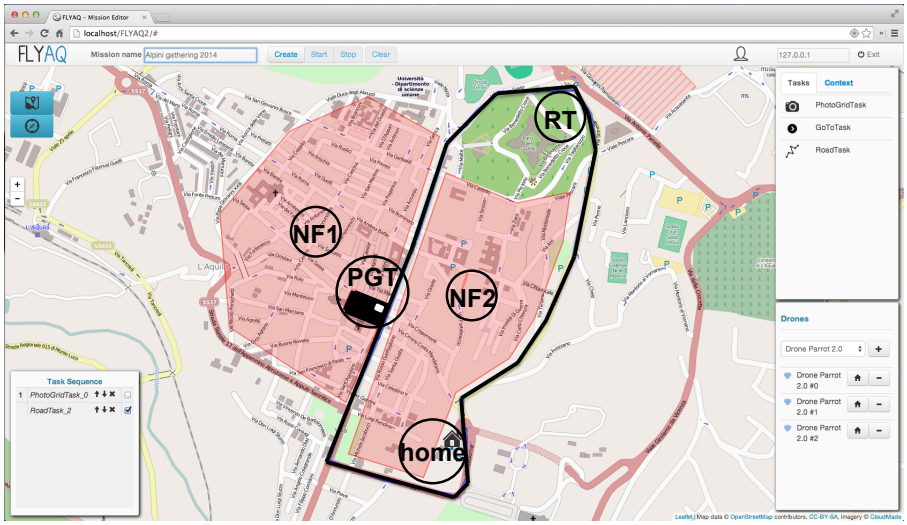


Figure 3.6: A patrolling mission in FLYAQ, where a drone follows a street, repeatedly takes photos (at specified distances), and avoids no-fly zones. NF1 and NF2 are no-fly zones, RT is road task to follow a street while PGT is photo grid task indicating where photos can be taken. (from [11])

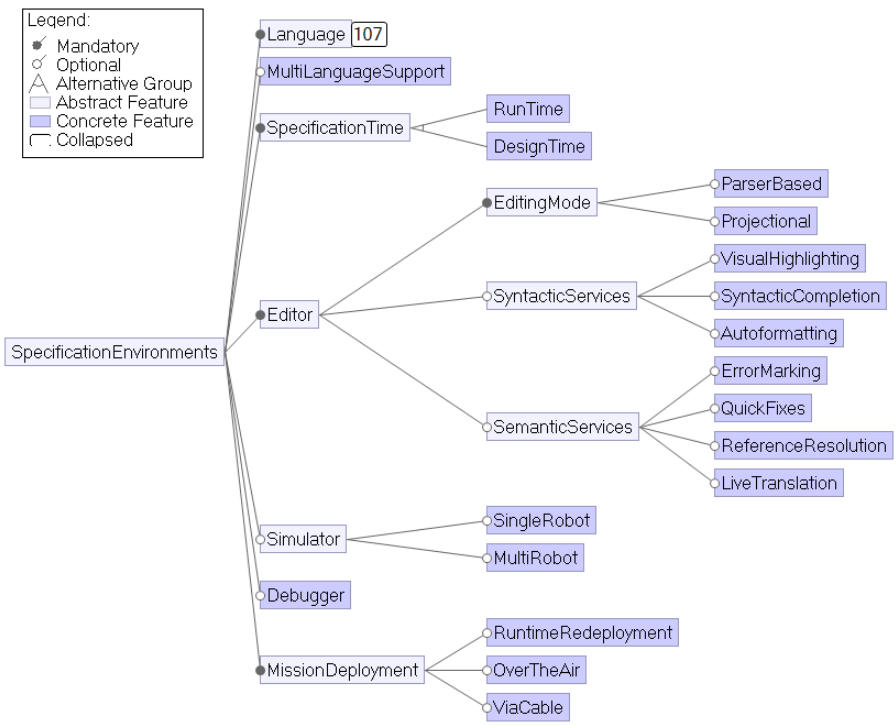


Figure 3.7: Overview of the features (133 features in total)

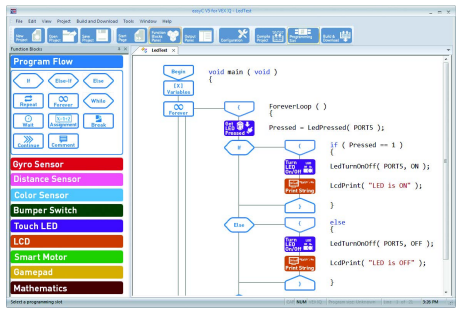


Figure 3.8: Visual and textual syntax side-by-side in EasyC's projectional editor (from [128]).

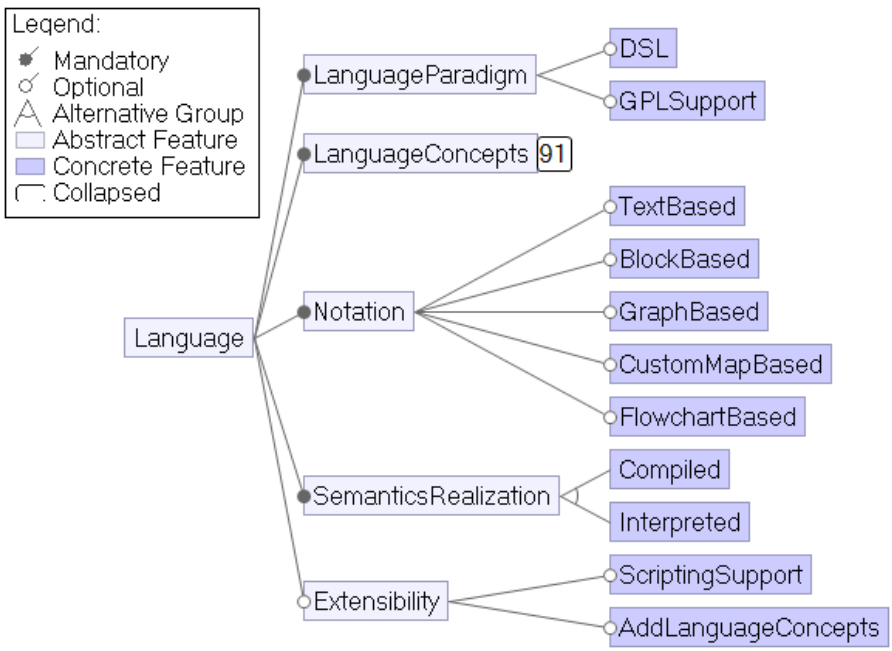


Figure 3.9: General language characteristics identified

Table 3.3: Kinds of notation supported by the environments – the visual notations belong to the primary DSLs of the environments; the textual notations are additional languages supported

notation	environment
Visual	
Block-Based	Arcbotics' SparkiDuino, Ardublockly, Aseba, BlocklyPro, Code Lab, Edison software, Enchanting, LEGO Mindstorms EV3, Makeblock 5, Makecode, Marty software, Metabot, Ozoblockly, PICAXE, Robot Mesh Studio, Scratch Ev3, Sphero, Tello Edu App, Turtlebot3-blockly, VEX Coding Studio, MiniBloq, Open Roberta, RobotC
Flowchart-based	EasyC, PICAXE, Robot Mesh Studio (flowol)
Graph-Based	Choregraphe, MissionLab, TRIK Studio, TiViPE, PROMISE
Map-based	FLYAQ
Textual	
C/C++	Arcbotics' SparkiDuino, Makeblock 5, Robot Mesh Studio, VEX Coding Studio, RobotC, TRIK Studio
Python	Code Lab, Edison software, LEGO Mindstorms EV3, Marty software, Robot Mesh Studio, TRIK Studio
JavaScript	Marty software, PICAXE, Sphero, TRIK Studio
Basic	PICAXE
Textual DSL	Aseba (custom event-based language), PROMISE (textual behavior-tree language)

Table 3.4: Target general-purpose language when code is generated by the environment

target language	environment
C/C++	RobotC, BlocklyPro, Robot Mesh Studio, Arcbotics' SparkiDuino, Open Roberta, TRIK Studio, Choregraphe, EasyC, MiniBloq, TiViPE
Java	Open Roberta, Enchanting, Scratch Ev3, VEX Coding Studio
Java Script	Open Roberta, Makecode, Ozoblockly, Sphero, TRIK Studio, Choregraphe
Python	Open Roberta, Turtlebot3-blockly, Robot Mesh Studio, Tello Edu App, Makeblock 5, Marty software, TRIK Studio, Choregraphe, Edison software
Others	Ardublockly (Arduino code), Metabot (assembly code), TRIK Studio (F#, PascalABC, NXT OSEK C), Choregraphe (Matlab), PICAXE (Basic), FLYAQ (QBL), Aseba (VPL to Aseba event scripting language AESL), PROMISE (PROMISE intermediate language)

Table 3.5: Sensory data abstraction usage by specification environments

Usage instances	No. of sensors	Sensors and frequency of usage
1 — 5	20	Analog (1), gesture detection (1), magnetomenter (2), power-watts (2), encoder (2), vision-seeing (2), gear potentiometer (3), barometer (1), finger print scanner (3), GPS module (3), proximity-sonar (3), rotation detector (3), energy meter (5), brick button (3), motor rotation sensor (3), face detector (1), mark sensor (1), speech recognizer (3), compass (3), timer (5).
6 — 10	10	Line-detector (7), proximity-distance (10), accelerometer (10), sound (10), proximity-ultrasonic (9), color (9), infrared (10), touch (9), light (9), thermometer(8).
11 — 15	1	Gyro-orientation (13).

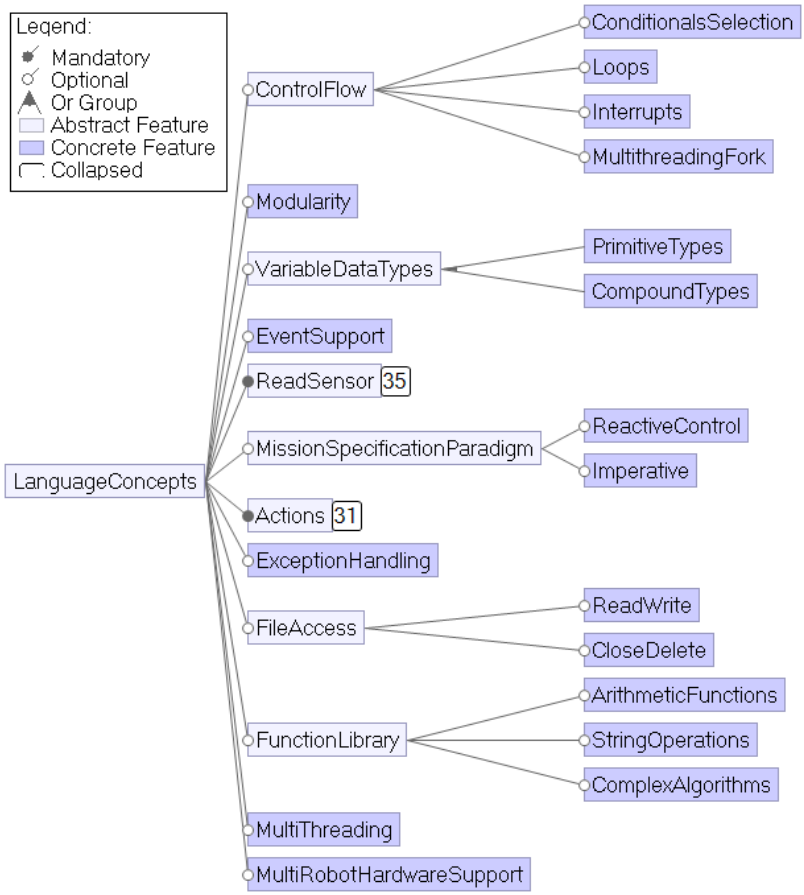


Figure 3.10: Language Concepts

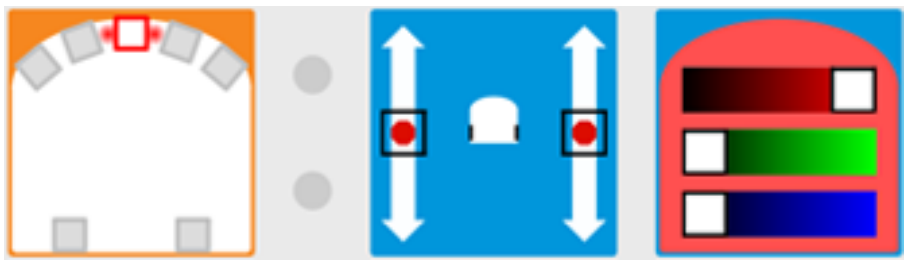


Figure 3.11: An example of Aseba's block-based syntax for its language VPL—an event-based language consisting of event-action pairs. Here, when an object is detected (event), the top color (action) is set to red.

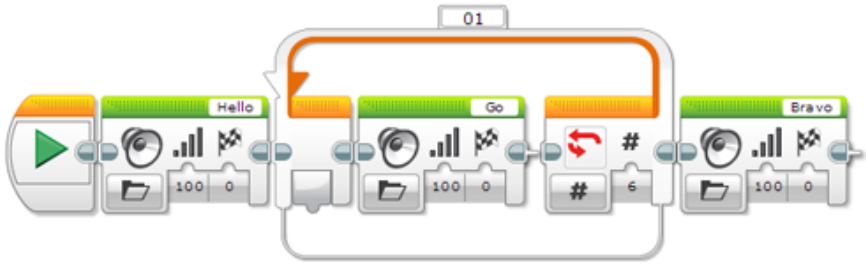


Figure 3.12: Program control flow example in LEGO Mindstorms EV3: The

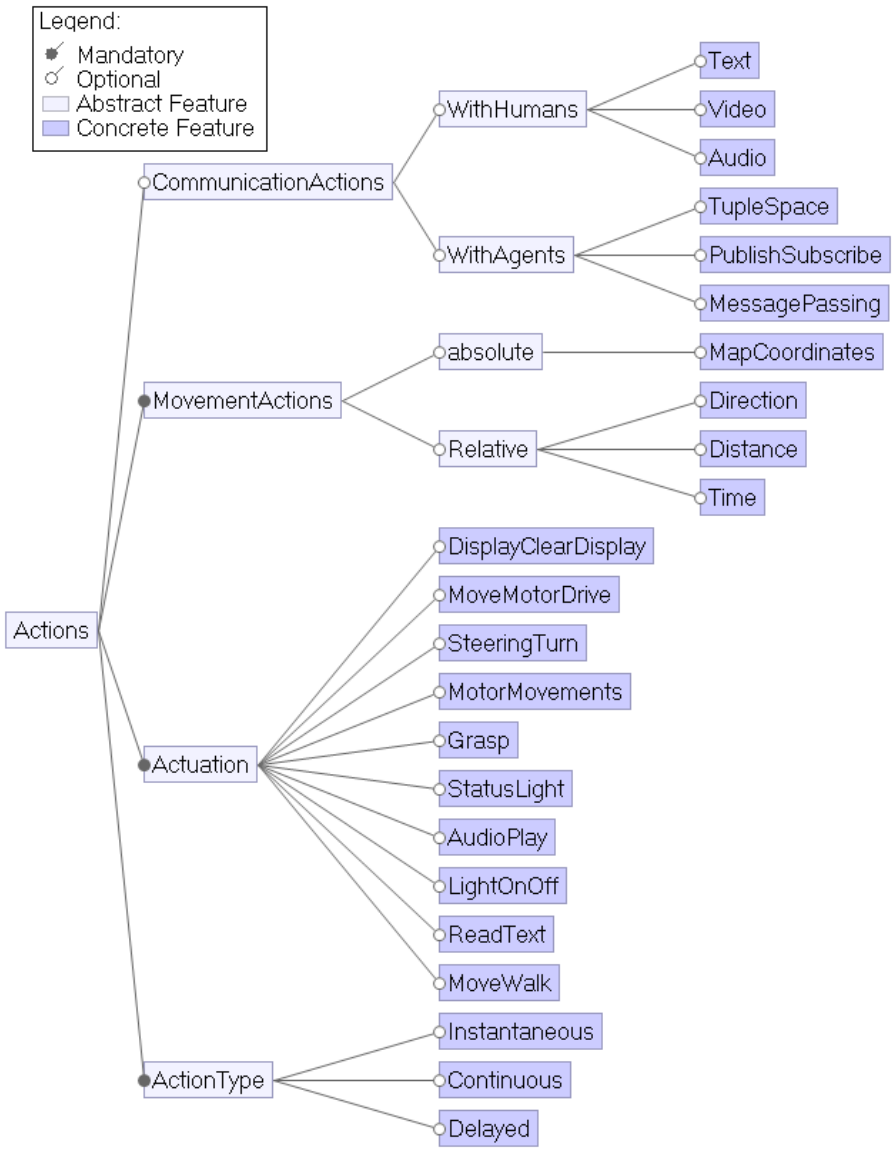


Figure 3.13: Actions

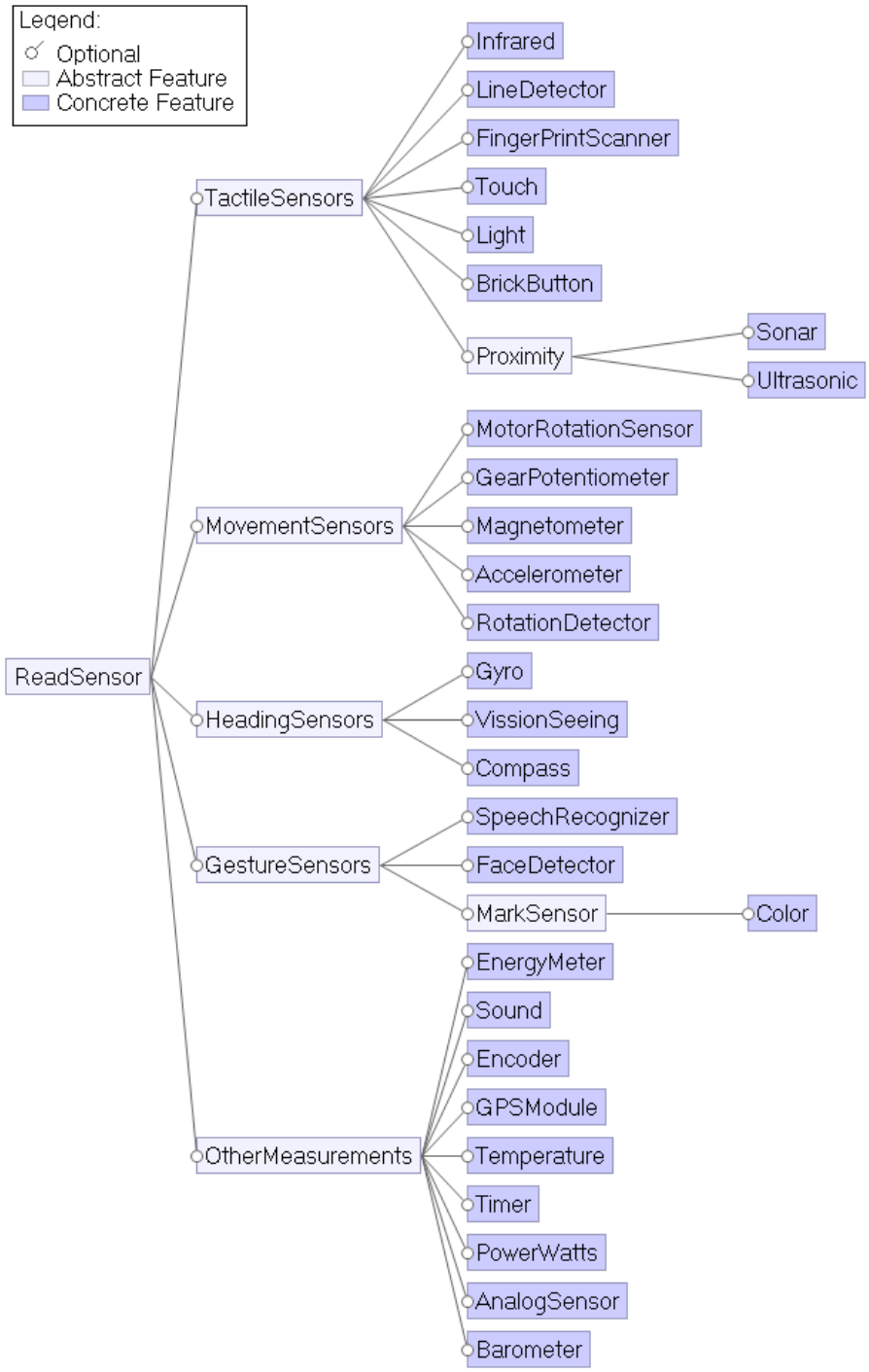


Figure 3.14: Sensors with language constructs for reading data

Chapter 4

Paper C

A generated property specification language for resilient multi-robot missions.

S. Dragule, B. Meyers, and P. Pelliccione.

International Workshop on Software Engineering for Resilient Systems, pp. 45–61, Springer, 2017.

Abstract

The use of robots is gaining considerable traction in several domains, since they are capable of assisting and replacing humans for everyday tasks. To harvest the full potential of robots, it must be possible to define missions for robots that are domain-specific, resilient, and collaborative. Currently, robot vendors provide low-level APIs to program such missions, making mission definition a task-specific and error-prone activity. There is a need for quick definition of new missions, by users that lack programming expertise, such as farmers and emergency workers. In this paper, we extend the existing FLYAQ platform to support the high-level specification of adaptive and highly-resilient missions. We present an extensible specification language that allows users to declaratively specify domain-specific constraints as properties of missions, thus complementing the existing FLYAQ mission language. This permits to move at runtime, the actual generation of low-level operations to satisfy the declaratively specified mission. We show how this specification language can be automatically generated from a domain-specific FLYAQ mission language by using the generative ProMoBox approach. Next, we show how mission goals are achieved taking mission properties into account, and how missions may change due to unexpected circumstances.

Keywords: *Domain-Specific Languages, Robotics, Model-Driven Engineering, Resilient Systems, Cyber-Physical Systems.*

4.1 Introduction

The use of multirobot systems in civilian missions requires high variability due to the diversity of domains [168, 169]. Moreover, robotic systems are defined through a craftsmanship instead of established engineering processes. Programming missions for robots requires high knowledge of robotic programming and robot mechatronics. While domain users are experts in their domains (e.g., emergency, commercial and agriculture) they are not trained to program missions for multirobot execution in their domains using the low-level APIs provided by robot vendors. Not much has been done to enable domain experts to easily use robots to execute missions in the respective domains.

To address this problem, Di Ruscio et al. introduced FLYAQ [11,36]. FLYAQ is a platform designed to enable non-expert domain users to program missions for a team of multicopters. The platform has been then generalized to different types of robots in [35, 168]. The platform is extensible, so that domain-specific robots and missions can be defined. Unfortunately, this platform can only define missions at design time. This is unrealistic since most missions will be faced by unforeseeable and emergent situations during mission execution, and, consequently, robots should be resilient to these unforeseeable and emergent situations. For example, one robot may malfunction calling for re-planning so that another robot can take the roles this robot was executing. This need for run-time adaptation is clearly described in the Robotics Multi-Annual Roadmap 2020 [169]. In this context, the document describes the degree in which models can be used in robotics in three steps ([169] § Section 5.2). Step 1 assumes that models are used to define missions by people at design time. Step 2 requires robots to use models at run-time to interact and explain what they are doing. Step 3 means that robots adapt and improve models to redefine what they are doing based on artificial intelligence.

The FLYAQ platform uses models according to step 1. In this line of research, we intend to improve FLYAQ to support self-adaptive robots at the mission level, thus achieving step 3. This means that robots can change their behaviour to successfully carry out missions under unforeseen circumstances. We achieve this by introducing a declarative language for describing mission goals and constraints. In this research we exclusively focus on the high-level strategic, domain-specific, collaborative aspects of self-adaptation. To this end, we specify mission objectives in a declarative way, as properties, using a language we call the Mission Specification Language (MSL). We present a technique that allows the generation of such a MSL for a specific FLYAQ extension (e.g., emergency, commercial, agriculture). As MSL is declarative, it does not specify *how* the mission is planned for a team of robots, but instead specifies what goals must be achieved and what constraints cannot be violated. This way, missions become fully specified only at run-time and they can be re-planned at run-time.

Paper structure: Section 4.2 discusses the background of this research. Section 4.3 introduces the property specification language. Section 4.4 evaluates the approach by showing an implementation of the property specification language. Section 4.5 discusses related work. Section 4.6 concludes the paper with opportunities for future works.

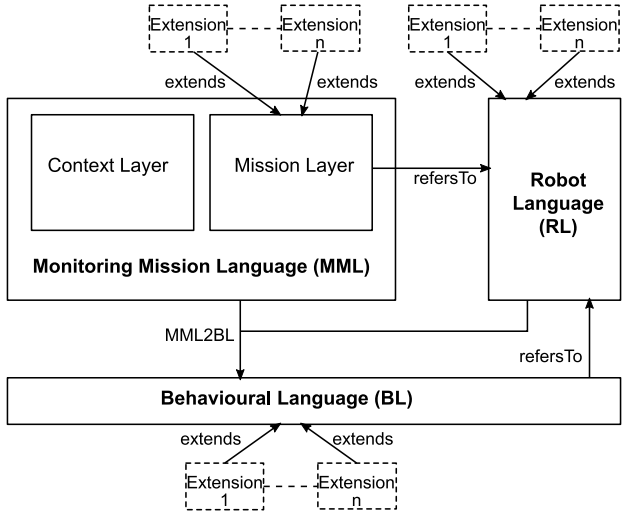


Figure 4.1: The family of FLYAQ DSMLs (adapted from [35]).

4.2 Background

In this section, we briefly explain domain-specific modelling, and the FLYAQ platform, on which we build our research.

4.2.1 Domain Specific Modelling

In Domain-Specific Modelling (DSM) [170], a methodology in model-driven software engineering, the general goal is to provide means for domain users to model systems in their problem domain. Model-driven techniques such as metamodelling and model transformation enable the creation of Domain-Specific Modelling Languages (DSMLs). These DSMLs can be used by domain experts, to specify, for example, missions for a team of robots. Current DSM techniques allow domain users to model at the domain level and simulate, optimise, and transform the model to other formalisms, synthesise code, generate documentation, etc.

4.2.2 FLYAQ platform

The FLYAQ platform [11, 36, 168] employs domain-specific modelling to take care of the various domains involved in mission definition and specification. The approach proposes a family of DSMLs for the specification of missions of multirobot systems (MMRSs), as shown in Figure 4.1:

- Monitoring Mission Language (MML): this DSML consists of the context layer and mission layer. This DSML is meant to be used by domain users, to model missions. Missions are represented in the mission layer as sequences of tasks on a map, as shown in Figure 4.2. The context layer provides additional constrains over the mission area, such as obstacles and no-fly zones;

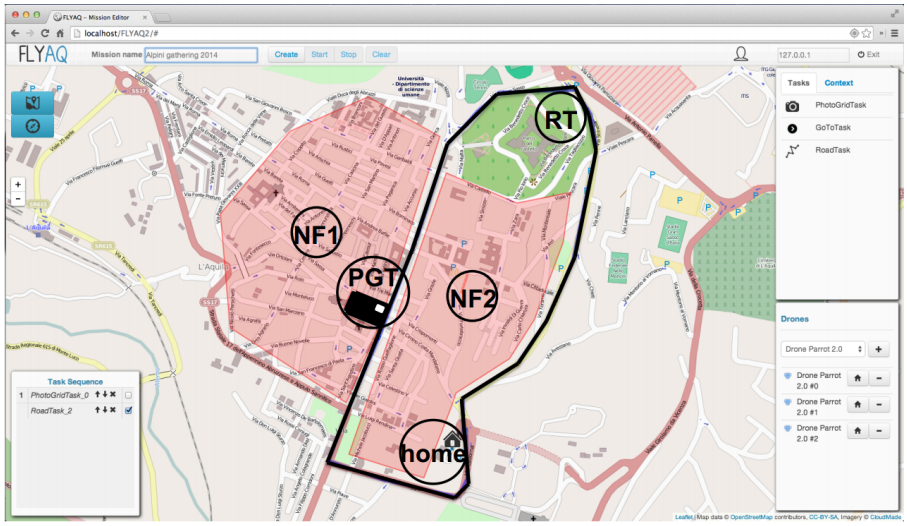


Figure 4.2: A screenshot of the FLYAQ tool (from [11]).

- Robot Language (RL): using this DSML, types of robots or individual robots can be defined by a robot engineer, mapping out their capabilities and characteristics;

- Behaviour Language (BL): this language allows the definition of sequential atomic movements and actions of each robot that are used to instruct the individual robots. The BL serves as the low-level language, to which high-level missions defined in MML can be transformed automatically using the MML2BL transformation. This transformation takes care of low-level planning, such as path finding, covering areas, etc. while achieving the high-level goals. Code can be easily generated from the generated BL models, and then it can be uploaded to the individual robots.

Mission goals, robot characteristics, and actions should be customised to the application domains. Therefore, extensions can be defined on MML, RL and BL, as shown in Figure 4.1. In case of MML, extensions may define a task to "scan an area by taking pictures". Example extensions to RL may include domain-specific notions like "number of propellers", "launch type" (horizontal or vertical takeoff), "maximum altitude", etc. BL may be extended with movements like "take off" and "land", and a "go to strategy" (move first over the horizontal or vertical axis, or move diagonally?), "take a picture", "start recording a video", etc.

For example, it is possible to define extensions in FLYAQ to allow flying robots to take pictures of areas. Using this extension, one can specify missions to e.g., survey an area where a public event is being held. Another example is in the domain of agriculture. One multicopter is able to detect pests by taking pictures and using image recognition techniques. If a pest is detected, another multicopter that is able to spray insecticide must spray the infected plants. It should only spray plants that are infected. We use these examples throughout

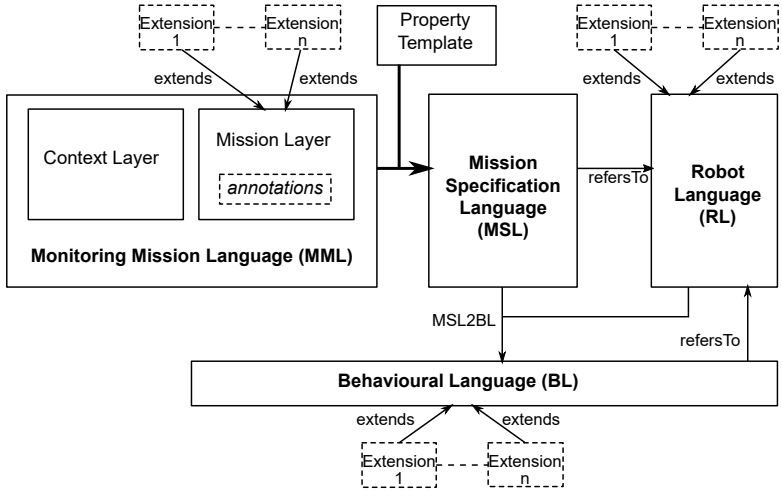


Figure 4.3: Overview of the approach as an extension of the FLYAQ platform.

the paper.

Despite its extension mechanism, FLYAQ does not support (a) advanced temporal constraints (other than order, fork or join) over various tasks or robots in MML, e.g., a certain task can only start if another robot is surveying the task area (for safety reasons), or video recording can only start after clearance (for privacy reasons); and (b) run-time adaptation of a mission due to some information at run-time, e.g., taking pictures of areas where high temperature was detected by another robot, or reacting to a loss of signal of a robot. The research presented in this paper addresses these shortcomings.

4.3 Mission Specification Language

Our approach extends the FLYAQ platform as shown in Figure 4.3. The mission layer of MML is annotated, and as a consequence a *Mission Specification Language* (MSL) can be generated automatically from MML and a *Property Template* to better match the platform extensions of MML. MSL extends MML with language constructs to define temporal properties for robot missions. Our approach ensures that, when an extension is defined as done in FLYAQ, no additional effort is required to generate MSL.

4.3.1 Mission Specification Language

The mission specification language (MSL) is intended to specify properties of a mission that allows users to define temporal mission constraints in a highly declarative way. This complements MML, where areas are selected, and specific tasks, obstacles and no-fly zones are plotted on the map. MSL replaces the order, fork and join of MML, supporting more expressive constraints. We use a number of temporal patterns, taken from Dwyer et al. [171] and Autili et al. [172], as a basis for the Property Template from which MSL is generated. According to this work, properties consist of a *temporal pattern* in a

scope, over some propositions P, Q, R and S (i.e., occurrences of something, e.g., spraying, entering an area, etc.). Temporal patterns can be *absence* (something should never occur), *universality* (something should always occur), *existence* (something should eventually occur), *bounded existence* (something should occur at most n times), *precedence* (an occurrence of P must be preceded by an occurrence of Q), or *response* (an occurrence of P must be followed by an occurrence of Q). Scopes can be *globally*, *after* the occurrence of R, *before* the occurrence of S, *between* occurrences of R and S, or after an occurrence of R until an occurrence of S (*after until*).

The declarative constraint specification shields the user from the actual planning. For example, if pests are detected, the corresponding areas are sprayed. This is an example of a response pattern with global scope. The user may use a precedence pattern to say that a pest needs to be detected at a location before this point is sprayed. This constraint can be met in a number of equally valid ways. A first option would be that one robot first detects all locations, then returns to the base where its data is downloaded and locations of infected plants are uploaded to a second robot, who goes out to spray the infected plants. A second option would be that two robots perform the task in parallel: one robot sends coordinates of detected pests to the other robot, which only sprays infected points. The second robot may follow a preplanned path, or may plan its path at run-time, according to the received coordinates. Collisions may occur, or may be avoided by flying at different altitudes. A third option would be that multiple robots detect pests, and multiple robots spray. If robots can adapt their mission at run-time, this may involve advanced scheduling, employing run-time monitors [173]. This shows that a declarative language can be supported by very simple to very advanced algorithms. The goal of MSL is that the domain user is shielded from such advanced planning algorithms.

To further illustrate MSL, we give some more examples of properties.

- Between entering and exiting an area, a robot can never exceed a given altitude. According to Dwyer et al. [171], this is an absence pattern with between scope. Note that this between scope may be more intuitively expressed as “during” or “while”.
- Between receiving a “stop” message and a “start” message, pictures cannot be taken.
- A robot can only start its activity if another robot is in a given position to monitor this activity.

4.3.2 Run-time Adaptation of Multirobot Missions

In its current state, the MML platform generates robot missions at design time. This means that robot missions cannot be adapted at run-time. We intend to support the run-time recalculation of BL models (i.e., robot commands) from a declarative mission description; this is needed in case information at run-time prompts the robots to change the mission. Our approach is applicable to various implementation techniques: for example, the mission recalculation may be achieved by the robot or by the ground station, and may be specified off-line or at run-time, or a mix of these.

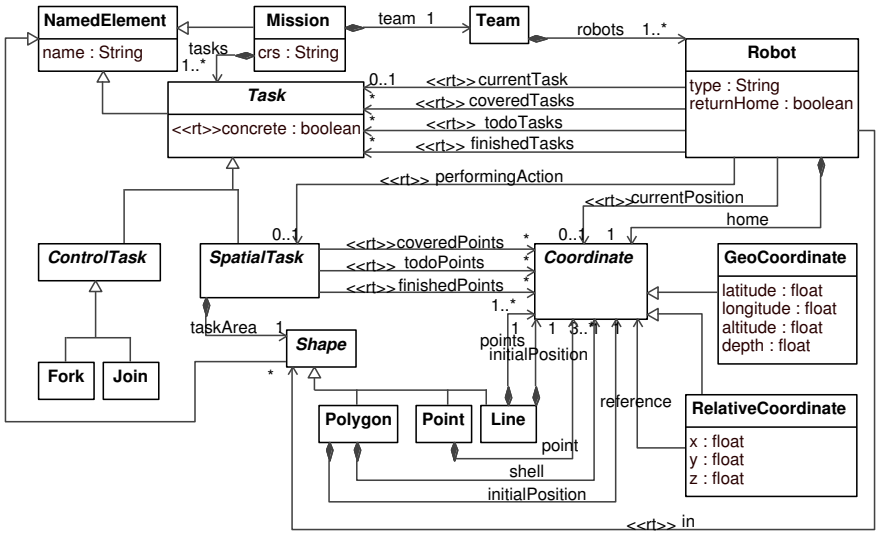


Figure 4.4: The annotated class diagram of the MML mission layer (context layer remains unchanged and is not shown).

In order to allow run-time information in a mission specification in MSL, we altered the existing MML mission layer from [168], as shown in Figure 4.4. We have changed the metamodel in several ways:

- We have extracted a Shape class (and Polygon, Point, Line subclasses) from the original PolygonTask, LineTask, and PointTask. In particular, the new Polygon class serves now as superclass of Area in the context layer of MML in [35]. This new Shape class will allow users to specify new shapes on the map that may trigger rules like: do not record within a specific area.

- The meaning of Task has been extended. At mission specification time, a task may be addressed by multiple robots. After mission generation, tasks are split up into multiple concrete tasks, each for one robot.

- TaskDependency has been removed from MML and its functionality will be subsumed by the specification language.

- We added run-time language constructs (annotated with *rt*), so that specifications can be defined in terms of the current state of the mission in terms of tasks and position. We added the following run-time information in terms of tasks:

  - *currentTask*: the task a robot is currently working on;
  - *coveredTasks*: the concrete tasks that are planned for a robot;
  - *todoTasks*: the concrete tasks that a robot still needs to perform;
  - *finishedTasks*: the concrete tasks that a robot has done;

– *performingAction*: the action (defined in the task) a robot is currently performing. It may be none if e.g., the robot is moving and the action is instantaneous (e.g., taking a picture).

We added the following run-time information in terms of position:

– *currentPosition*: the current position of a robot;
– *coveredPoints*: the points of a concrete task that are defined by the cover function;
– *todoPoints*: the points of a concrete task that still need to be visited;
– *finishedPoints*: the points of a concrete task that have been visited;
– *in*: the shapes the robot is currently in.

As is usual in FLYAQ, extensions can be defined for specific application domains, as shown in Figure 4.3. Note that for brevity, we do not show the MML context layer and RL (which can be extended in its own right).

4.3.3 Generation of the Property Specification Language

As shown in Figure 4.3, a domain-specific MSL can be generated from the annotated MML (as shown in Figure 4.3), with defined extensions (e.g., to enable detection of pests in an area, and spraying certain plants). This means that extensions have to be defined only once, and can be used for specifying missions in the original MML as well as in MSL. The metamodel of MSL, which results from the language generation process without an extension, is shown in Figure 4.5. It consists of three parts:

- Mission layer: the upper part (unshaded) represents our variant to the original MML mission layer, which allows the user to define missions at design-time like in the original MML. For example, "pictures should be taken in an area, with a distance of x from each other". Additionally, shapes can be defined, that can be used in MSL properties. In case of an MML extension, extensions will also appear in this part.

- Temporal pattern layer: the middle part (shaded) represents the temporal patterns, which allow the user to define temporal constraints based on the patterns by Dwyer et al. [171]. For example, "after R happens, P must be followed by Q".

- Proposition layer: the bottom part (unshaded) represents the language fragment to define propositions P, Q, R, and S of temporal patterns. More specifically, it allows the user to specify a condition on the state of a mission (i.e., a structural pattern). For example, "a robot is in a specific area", or "a task is completed". In case of an MML extension, pattern versions of extensions will also appear in this part.

With MSL, a mission can be specified by plotting an area on the map, and defining a DetectPest and Spray task in this area, using the MSL mission layer, which is extended with language concepts from agriculture. With the MSL temporal pattern and MSL proposition layer, a property can be specified that states that detecting a pest at a location must result in spraying that location.

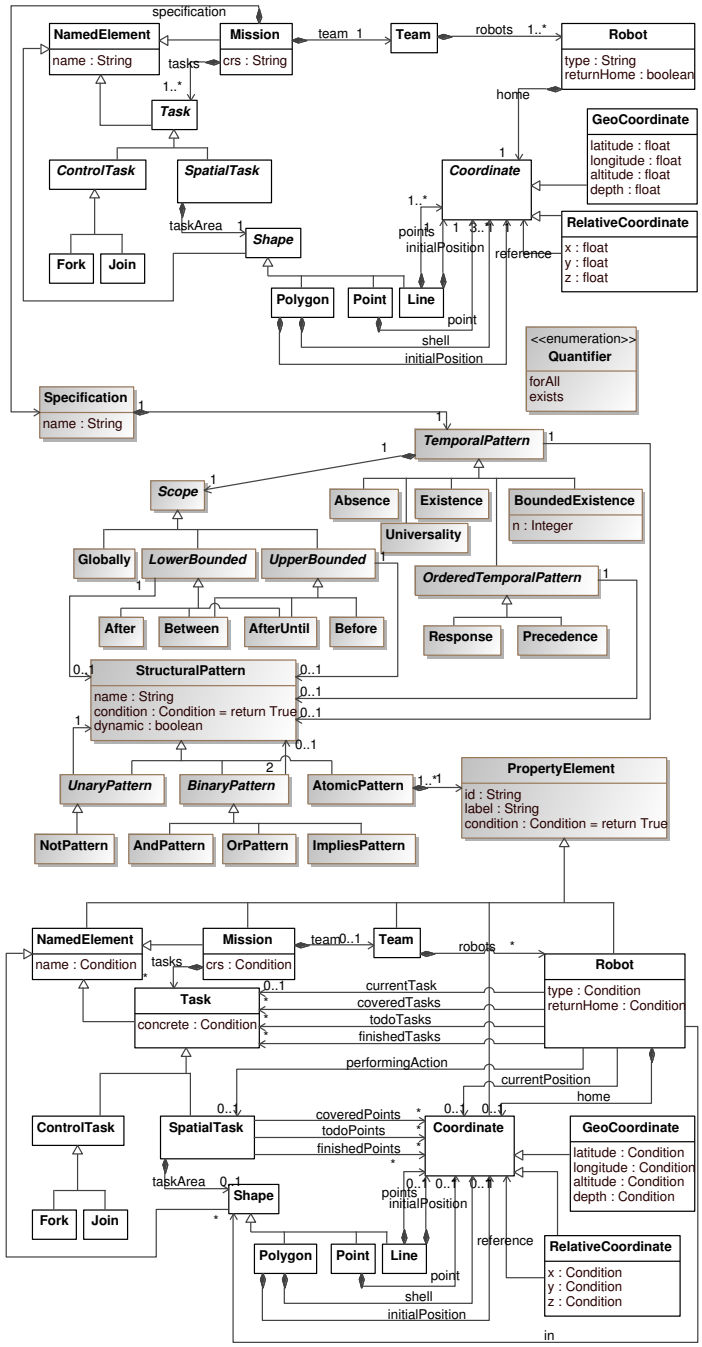


Figure 4.5: The generated metamodel of MSL without extensions.

The MSL metamodel of Figure 4.5 is generated fully automatically from an annotated (and possibly extended) MML metamodel (Figure 4.4) and the generic property template (shaded part of Figure 4.5). Why do we need to automatically generate an MSL metamodel? Please note that, if there was no generative approach, each of the domain-specific language constructs (e.g., spraying a pest, maximum altitude of a robot, etc.) would have had to be modelled a second time in MSL. With our approach, the extension mechanisms of FLYAQ can be reused as described in [168], and a domain-specific MSL is generated without any additional effort.

We use techniques from the ProMoBox framework [174, 175] to achieve this. First, the MSL mission layer is generated by taking the annotated MML metamodel and removing all run-time language constructs, which are annotated with *rt*, thus creating the unshaded upper part of Figure 4.5. Next, the property template is merged into this model by adding an association called *specification* from Mission to Specification. Finally, the annotated MML metamodel is taken for a second time, and the run-time language constructs are kept (by removing the annotation). This time, the metamodel is converted into a structural pattern language by using the RAMification process [176]: relaxing all lower multiplicities, making all abstract classes concrete, and changing all attribute types to Condition, as can be seen in the proposition layer of Figure 4.5. This RAMified metamodel is merged into MSL by generating inheritance links from all top-level classes to PropertyElement.

4.3.4 Transforming MSL to BL

Transforming MSL to BL (see MSL2BL in Figure 4.3) can be done according to several stategies. Given the tight relation between MSL and MML, the transformation algorithms of MML2BL [11] (i.e., path finding, covering areas) can be reused. Moreover, various implementation strategies as mentioned in Section 4.3.1 can be covered by MSL2BL: mission recalculation may be achieved by a mix of the robot or the ground station, off-line or at run-time. These strategies may requiring enhancement of BL to e.g., explicitly support data communication or monitoring. As this paper focuses on the definition and generation of MSL, this is left as future work.

4.4 Evaluation: Implementation of MSL as Textual DSL

In this section, we evaluate the MSL by introducing an implementation as a textual language in Xtext [177], and show how missions can be expressed in this language.

4.4.1 A Concrete Syntax for MSL

According to what described in [172], temporal properties (the shaded part of Figure 4.5) might be profitably described using a structured English grammar. For instance, we can devise a textual syntax for the MSL proposition layer (the bottom part of Figure 4.5), where each of the associations can form a subsentence with the two attached instances. For example, "*a Robot currently on*

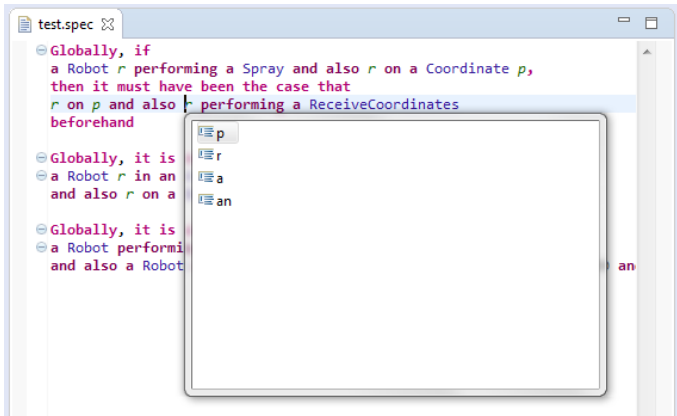


Figure 4.6: Screenshot of the MSL in Xtext.

a GeoCoordinate" denotes the presence of an instance of Robot and an instance of GeoCoordinate, with a currentPosition link in between. More intricate, "*a Robot r currently on a GeoCoordinate with latitude lower than 100*" denotes additional conditions on the robot, etc. A structured English grammar to represent a subsentence for one association is defined as follows (id, Label, Value, Attribute, Class, Association are terminals):

`Proposition ::= Proposition (`**`and also`**` Proposition)+`
`| Proposition (`**`or`**` Proposition)+`
`| Proposition (`**`implies`**` Proposition)+ | AtomicProposition`
`AtomicProposition ::= Expression [Association Expression]`
`Expression ::= Instance [InstanceCondition]`
`InstanceCondition ::=` **`with`** `(ValueCondition | BooleanCondition (`**`and`**` ValueCondition | BooleanCondition)*)`
`ValueCondition ::=` {Attribute} (**as** | **less than** | **greater than**) {Value}
`BooleanCondition ::= [`**`not`**`]` {Attribute}
`Instance ::=` {id} | {Label} | **a** {Class} [{Label}]
`Association ::=` (**that is a task of** | **that is a team of** | **that is in** | [**currently**] **doing** | **that has scheduled** | **that has planned in the future** | **that has finished** | [**currently**] **performing** | **in** | [**currently**] **on** | **with as home** | **with task area** | **which visits** | **which will visit in the future** | **which has visited** | **with points** | **with initial position** | **which references** | {Association})

The above grammar is combined with the grammar for temporal properties presented in [172] so that temporal properties can be described in standard LTL or CTL. This might enable the use of model checking approaches, like UPPAAL[1]. With this grammar, temporal patterns involving multiple links can be expressed with AndPatterns. MML extensions can be used by instantiating classes defined by the extension. This is illustrated below in the examples.

Our current implementation in Xtext includes variable name resolution, parse error visualisation, auto-completion and syntax highlighting[2]. A screenshot of the MSL editor is shown in Figure 4.6. Since both the FLYAQ platform and MSL are implemented on top of the Ecore platform, they can be easily

[1] `http://www.uppaal.org/`
[2] An implementation of this grammar can be found at `http://msdl.cs.mcgill.ca/people/bart/flyaq/flyaq.html`.

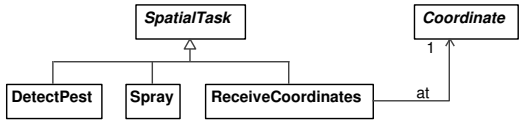


Figure 4.7: The MML extension.

merged at the EMF layer [178].

4.4.2 Examples of MSL

This section presents examples of temporal properties defined in MSL, while illustrating the relation between the grammar presented above and the MSL metamodel presented in Figure 4.5. For these examples, we define a MML extension in the agricultural domain as shown in Figure 4.7, with:

- DetectPest: scanning for a pest in an area and in case of detection, send some coordinates;
- Spray: spraying pesticides at a point;
- ReceiveCoordinates: receiving coordinates where a pest has been detected, with the "at" association referring to the received coordinates.

Note that, after generation of MSL, these additional language constructs will occur twice in MSL, namely in the MSL mission layer and in the MSL proposition layer.

The example of Figure 4.8 (top) shows the abstract syntax of the MSL property "a robot only sprays at a location if it has received these coordinates to spray at that location" as an object diagram. The Specification consists of a Precedence pattern. The left AtomicPattern states the condition Q, saying that a ReceiveCoordinates task is executed at a coordinate **p**. Note how the "at" association is used. The right AtomicPattern P describes a robot **r**, spraying at aforementioned point **p**. Note that the coveredPoints link is superfluous, because if the robot is currently performing an action of a task, it must be inside the task area. In structured English grammar, the temporal specification is as follows (leaving out the superfluous quantification and coveredPoints link): "*Globally, if a SprayRobot **r** performing a Spray and **r** on a Coordinate **p**, then it must have been the case that a ReceiveCoordinates **at p** beforehand*". Note how "at" is automatically resolved to an instance of the "at" association.

The example of Figure 4.8 (bottom left) represents "in a certain area, a robot can never exceed a given altitude". Note that the Area class (now a subclass of Polygon) is part of the MML context layer and is not shown in Figure 4.5. In structured English grammar, the temporal specification is as follows: "*Globally, it is never the case that a Robot **r** in an Area with name as "lowflyzone" and also **r** on a GeoCoordinate with altitude more than 20*".

The example of Figure 4.8 (botton right) represents "a robot can only perform a certain task if another robot is at a certain position". In structured English grammar, the temporal specification is as follows: "*Globally, it is always the case that a Robot performing a Task implies a Robot on a RelativeCoordinate with*

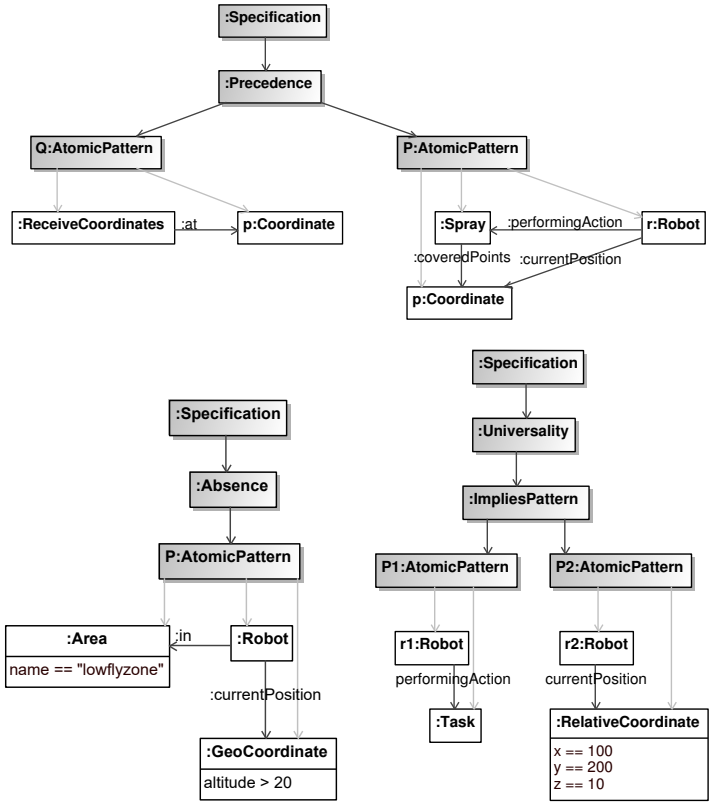


Figure 4.8: Three examples of temporal specifications as instances of MSL.

x as 100 and y as 200 and z as 10". Note that, since spatial constraints are at the very core of FLYAQ, it is interesting to introduce syntactic sugar for a robot being at a coordinate, e.g., by allowing syntax like "a robot on (100, 200, 10)".

4.5 Related Work

In this section, we present related works on run-time adaptation of multirobot missions. There are several works that focus on robotics and self-adaptation, like [179–181]. For the sake of space in this section we focus on related works in run-time adaptation of (robot) missions, with a focus on MDE approaches. While most of the mission specification tools (e.g., [83]) and the FLYAQ platform [35,36] provide for specification of multirobot missions at design time, there is need to have specification and recalculation of missions at run-time for missions executed under uncertain environments.

In an effort to leverage run-time adaptation for UAV based systems, the work in [18] uses an ensemble concept to aggregate teams collaborating in a mission at run-time. This platform focuses on the aggregation of agents but on not the high-level expressiveness of the mission properties. Using run-time models for automatic reorganization of multirobot system, the work

in [182] focuses more on techniques for task distribution based on the goals and organisation of the teams, but not how goals are expressed so that adaptation at run-time is made easy. In [183] robots adapt models at run time, but configurations are made by an expert programmer, not domain experts declaratively. The work in [184] focuses on the behavioural model and how it auto-validates at run-time, yet we employ a generative approach. The work in [185] focuses on design-time to run-time explication of models, however this work does not really deal with adaptation triggered by run-time uncertainties. The work in [186] proposes an approach that uses models at design-time and run-time for collaboration. The proposed approach is specific to a particular domain without a clear path to adapt it for working in other domains.

4.6 Conclusion and Future Work

In this paper, we extended the FLYAQ platform with MSL, a highly declarative language that allows users to describe robot missions with temporal properties as constraints. The declarative nature of MSL allows run-time adaptation of these missions in case of unforeseen circumstances. We showed how MSL can be automatically tailored with domain-specific extensions by a generative approach. Additionally we presented a structured English grammar for MSL.

Future work will mainly focus on the mapping from MSL to BL (a language for describing individual robot movements and actions), allowing run-time adaptation and exploring different execution strategies. We intend to model communication between robots and/or the ground station explicitly in BL to achieve this. Furthermore, we are planning to incorporate real-time constraints in missions. Moreover, since our approach for enabling run-time adaptation of missions is model-driven and relies on code generation, we intend to analyse the feasibility of generating code in real-time.

Bibliography

[1] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *Transactions on Software Engineering*, pp. 1–1, 2019.

[2] G. J. Holzmann, "The logic of bugs," in *Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10, 2002.

[3] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar," *IEEE Trans. Software Eng.*, vol. 41, no. 7, pp. 620–638, 2015.

[4] C. Finucane, G. Jing, and H. Kress-Gazit, "LTLMoP: Experimenting with language, temporal logic and robot control," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2010, pp. 1988–1993.

[5] C. Menghi, S. García, P. Pelliccione, and J. Tumova, "Multi-robot LTL planning under uncertainty," in *International Symposium on Formal Methods*. Springer, 2018, pp. 399–417.

[6] M. Guo, K. H. Johansson, and D. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *International Conference on Robotics and Automation*, 2013.

[7] E. M. Wolff, U. Topcu, and R. M. Murray, "Automaton-guided controller synthesis for nonlinear systems with temporal logic," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 4332–4339.

[8] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2019, 2019.

[9] B. Schwartz, L. Nägele, A. Angerer, and B. A. MacDonald, "Towards a graphical language for quadrotor missions," *arXiv preprint arXiv:1412.1961*, 2014.

[10] S. Götz, M. Leuthäuser, J. Reimann, J. Schroeter, C. Wende, C. Wilke, and U. Aßmann, "A role-based language for collaborative robot applications," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2011, pp. 1–15.

[11] D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Automatic generation of detailed flight plans from high-level mission descriptions," in *International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS. ACM, 2016.

[12] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 1, no. 01, pp. 75–119, 2013.

[13] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, 2016.

[14] M. Luckcuck, M. Farrell, L. Dennis, C. Dixon, and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems: A Survey," *ACM Computing Surveys*, 2019.

[15] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *Transactions on Software Engineering (TSE), accepted for publication*, 2019.

[16] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating coblox: A comparative study of robotics programming environments for adult novices," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 366:1–366:12.

[17] G. Biggs and B. Macdonald, "A survey of robot programming systems," in *Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, 2003, p. 27.

[18] D. Bozhinoski, A. Bucchiarone, I. Malavolta, A. Marconi, and P. Pelliccione, "Leveraging Collective Run-Time Adaptation for UAV-Based Systems," *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 214–221, aug 2016.

[19] R. W. Button, J. Kamp, T. B. Curtin, and J. Dryden, "A survey of missions for unmanned undersea vehicles," RAND NATIONAL DEFENSE RESEARCH INST SANTA MONICA CA, Tech. Rep., 2009.

[20] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," *ICRA workshop on open source software*, vol. 3, no. 3.2, p. 5, 2009.

[21] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[22] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the OROCOS project," in *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, vol. 2. IEEE, 2003, pp. 2766–2771.

[23] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.

[24] C. Schlegel and R. Worz, "The software framework smartsoft for implementing sensorimotor systems," in *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No. 99CH36289)*, vol. 3. IEEE, 1999, pp. 1610–1616.

[25] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry, "The robot operating system: Package reuse and community dynamics," *Journal of Systems and Software*, vol. 151, pp. 226–242, 2019.

[26] T. Ohkawa, D. Uetake, T. Yokota, K. Ootsu, and T. Baba, "Reconfigurable and hardwired orb engine on FPGA by java-to-hdl synthesizer for realtime application," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 77–82, 2014.

[27] F. Piltan, N. Sulaiman, M. Marhaban, A. Nowzary, and M. Tohidian, "Design of FPGA based sliding mode controller for robot manipulator," *International Journal of Robotic and Automation*, vol. 2, no. 3, pp. 183–204, 2011.

[28] H.-S. Juang and K.-Y. Lurrr, "Design and control of a two-wheel self-balancing robot using the arduino microcontroller board," in *2013 10th IEEE International Conference on Control and Automation (ICCA)*. IEEE, 2013, pp. 634–639.

[29] D. Faconti, "Models and Tools to design Robotic Behaviors," Eurecat Tecnológic, Barcelona Spain, Tech. Rep. 732410, 2020. [Online]. Available: https://github.com/BehaviorTree/BehaviorTree.CPP/blob/master/MOOD2Be_final_report.pdf

[30] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, April 2017.

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

[32] H. Båtelsson, "Behavior trees in the unreal engine: Function and application," 2016.

[33] S. Dragule, B. Meyers, and P. Pelliccione, "A generated property specification language for resilient multirobot missions," in *Software Engineering for Resilient Systems*, A. Romanovsky and E. A. Troubitsyna, Eds. Cham: Springer International Publishing, 2017, pp. 45–61.

[34] FLYAQ, "http://www.flyaq.it/," 2019.

[35] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "A family of domain-specific languages for specifying civilian missions of multi-robot systems," *CEUR Workshop Proceedings*, vol. 1319, pp. 16–29, 2014.

[36] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 801–806.

[37] S. S. Skiena and M. A. Revilla, "Programming challenges: The programming contest training manual," *Acm SIGACT News*, vol. 34, no. 3, pp. 68–74, 2003.

[38] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–51, 2018.

[39] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: A perspective from the service robotics domain," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[40] D. Brugali, A. Agah, B. MacDonald, I. A. Nesnas, and W. D. Smart, "Trends in robot software domain engineering," in *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 3–8.

[41] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, 2019, pp. 1–6.

[42] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The state of adoption and the challenges of systematic variability management in industry," *Empirical Software Engineering*, vol. 25, pp. 1755–1797, 2020.

[43] L. Hugues and N. Bredeche, "Simbad: an autonomous robot simulation package for education and research," in *International Conference on Simulation of Adaptive Behavior*. Springer, 2006, pp. 831–842.

[44] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[45] M. Fowler and R. Parsons, *Domain-Specific Languages*. Addison-Wesley, 2011.

[46] S. Schauss, R. Lämmel, J. Härtel, M. Heinz, K. Klein, L. Härtel, and T. Berger, "A Chrestomathy of DSL Implementations," in *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2017.

[47] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, G. Wachsmuth *et al.*, *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[48] A. G. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2009.

[49] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *ICSE*, 2011, pp. 471–480, http://doi.acm.org/10.1145/1985793.1985858.

[50] T. Stahl and M. Völter, *Model-Driven Software Development*. Wiley, 2005.

[51] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003, http://csdl.computer.org/comp/mags/so/2003/05/s5019abs.htm.

[52] J. Bézivin, "On the unification power of models," *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[53] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[54] F.-Y. Wang, K. J. Kyriakopoulos, A. Tsolkas, and G. N. Saridis, "A Petri-net coordination model for an intelligent mobile robot," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 4, pp. 777–789, 1991.

[55] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha, "Petri net plans: a formal model for representation and execution of multi-robot plans," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 79–86.

[56] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using UML/P Statecharts," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 461–466.

[57] M. Klotzbücher and H. Bruyninckx, "Coordinating robotic tasks and systems with rfsm statecharts," 2012.

[58] B. S. Krishna, J. Oviya, S. Gowri, and M. Varshini, "Cloud robotics in industry using Raspberry Pi," in *2016 Second International Conference on Science Technology Engineering and Management (ICONSTEM)*. IEEE, 2016, pp. 543–547.

[59] A. Hentout, A. Maoudj, and B. Bouzouia, "A survey of development frameworks for robotics," in *2016 8th International Conference on Modelling, Identification and Control (ICMIC)*, Nov 2016, pp. 67–72.

[60] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, jan 2007.

[61] M. Tamre, R. Hudjakov, D. Shvarts, A. Polder, M. Hiiemaa, and M. Juurma, "Implementation of integrated wireless network and MatLab system to control autonomous mobile robot," *International Journal of Innovative Technology and Interdisciplinary Sciences*, vol. 1, no. 1, pp. 18–25, 2018.

[62] M. Kouzehgar, Y. K. Tamilselvam, M. V. Heredia, and M. R. Elara, "Self-reconfigurable façade-cleaning robot equipped with deep-learning-based crack detection based on convolutional neural networks," *Automation in Construction*, vol. 108, p. 102959, 2019.

[63] J. Cicolani, *Beginning Robotics with Raspberry Pi and Arduino: Using Python and OpenCV*. Apress, 2018.

[64] S. Arias, F. Boudin, R. Pissard-gibollet, D. Simon, S. Arias, F. Boudin, R. Pissard-gibollet, D. S. Orccad, S. Arias, F. Boudin, R. Pissard-gibollet, and D. Simon, "ORCCAD , robot controller model and its support using Eclipse Modeling tools," 2010.

[65] ROBOTC, "ROBOTC's Graphical feature," 2019. [Online]. Available: http://www.robotc.net/graphical/

[66] S. L. Salcedo and A. M. O. Idrobo, "New tools and methodologies for programming languages learning using the SCRIBBLER robot and Alice," *Proceedings - Frontiers in Education Conference, FIE*, pp. 1–6, 2011.

[67] Robot Mesh, "http://docs.robotmesh.com/ide-project-page," 2019.

[68] VEX Robotics, https://www.vexrobotics.com, 2019.

[69] D. Caron, "competitive robotics the best brings out in students, Tech Directions, v69 n6 p21-23 Jan 2010, https://eric.ed.gov/?id=EJ894879," pp. 21–24, 2010.

[70] PICAXE, "http://www.picaxe.com/software," 2019.

[71] E.-M. Jarvinen, A. Karsikas, and J. Hintikka, "Children as Innovators in Action–A Study of Microcontrollers in Finnish Comprehensive Schools," *Journal of Technology Education*, vol. 18, pp. 37–52, 2007.

[72] "Ros development studio," 2020. [Online]. Available: https://www.theconstructsim.com/rds-ros-development-studio

[73] R. P. Y. Ho, "Configuration of robotics solutions in microsoft robotics developer studio, http://aunilo.uum.edu.my/Find/Record/sg-ntu-dr.10356-20828," 2009.

[74] F. Piltan, M. H. Yarmahmoudi, M. Shamsodini, E. Mazlomian, and A. Hosainpour, "PUMA-560 robot manipulator position computed torque control methods using Matlab/Simulink and their integration into graduate nonlinear control and Matlab courses," *International Journal of Robotics and Automation*, vol. 3, no. 3, pp. 167–191, 2012.

[75] O. Michel, "Cyberbotics Ltd. Webots™: professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, 2004.

[76] S. Hart, P. Dinh, J. D. Yamokoski, B. Wightman, and N. Radford, "Robot task commander: A framework and ide for robot application development," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 1547–1554.

[77] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel, "The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, pp. 3–19, 08 2016.

[78] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Hägele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis, "BRICS - best practice in robotics." 01 2010, pp. 1–8.

[79] J.-C. Baillie, "URBI: towards a universal robotic body interface," 12 2004, pp. 33 – 51 Vol. 1.

[80] T. Balch, "Teambots 2.0: https://www.cs.cmu.edu/~trb/TeamBots/," 2000.

[81] D. Blank, D. Kumar, L. Meeden, and H. Yanco, "Pyro: A Python-based versatile programming environment for teaching robotics," *Journal on Educational Resources in Computing (JERIC)*, vol. 3, no. 4, pp. 1–es, 2003.

[82] "Copella simulator," 2020. [Online]. Available: https://www.coppeliarobotics.com/

[83] P. Ulam, Y. Endo, A. Wagner, and R. Arkin, "Integrated mission specification and task allocation for robot teams - design and implementation," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2007, pp. 4428–4435.

[84] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[85] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wasowski, "Behavior trees in action: A study of robotics applications," in *International Conference on Software Language Engineering (SLE)*. ACM, 2020.

[86] T. Röfer, "CABSL – C-Based agent behavior specification language," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11175 LNAI, pp. 135–142, 2018.

[87] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier, "Choregraphe: a graphical tool for humanoid robot programming," in *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, Sep. 2009, pp. 46–51.

[88] M. A. Miskam, S. Shamsuddin, H. Yussof, A. R. Omar, and M. Z. Muda, "Programming platform for NAO robot in cognitive interaction applications," in *2014 IEEE International Symposium on Robotics and Manufacturing Automation (ROMA)*. IEEE, dec 2014, pp. 141–146.

[89] Fraunhofer IAIS, "https://lab.open-roberta.org/," 2019.

[90] B. Jost, M. Ketterl, R. Budde, and T. Leimbach, "Graphical Programming Environments for Educational Robots: Open Roberta - Yet Another One?" in *2014 IEEE International Symposium on Multimedia*, Dec 2014, pp. 381–386.

[91] M. Ketterl, T. Leimbach, and R. Budde, "Open Roberta," no. 14, pp. 1–22, 2015. [Online]. Available: https://lab.open-roberta.org/

[92] Thymio, "https://www.thymio.org/en:start," 2019.

[93] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A modular architecture for event-based control of complex robots," *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 2, pp. 321–329, 2011.

[94] LEGO MINDSTORMS EV3, "https://www.lego.com/en-us/mindstorms/downloads/download-software," 2019.

[95] W. Burnett, "http://www.legoengineering.com/alternative-programming-languages/," 2018.

[96] R. Arkin, "Missionlab: Multiagent robotics meets visual programming," *Working notes of Tutorial on Mobile Robot Programming Paradigms, ICRA*, vol. 15, 2002.

[97] C. Menghi, C. Tsigkanos, T. Berger, and P. Pelliccione, "PsAlM: Specification of dependable robotic missions," in *International Conference on Software Engineering (ICSE): Companion Proceeedings*, 2019.

[98] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "Promise: High-level mission specification for multiple robots," in *42nd International Conference on Software Engineering (ICSE 2020 Demos)*, 2020.

[99] M. Colledanchise, "Behavior trees in robotics," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2017.

[100] D. Isla, "Handling complexity in the halo 2 ai," in *In Game Developers Conference*, 2005.

[101] S. García, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab, "An architecture for decentralized, collaborative, and autonomous robots," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 75–7509.

[102] M. Matamoros, C. Rascon, J. Hart, D. Holz, and L. van Beek, "Robocup@home 2018: Rules and regulations," http://www.robocupathome.org/rules/2018_rulebook.pdf, 2018.

[103] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[104] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Autonomous Robots*, vol. 38, no. 1, pp. 89–105, 2015.

[105] J. D. Robert W. Button, John Kamp, Thomas B. Curtin, *A Survey of Missions for Unmanned Undersea Vehicles*, 2010.

[106] SPARC, "Robotics 2020 Multi-Annual Roadmap," https://eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf, 2016.

[107] E. Fernández-Perdomo, J. Cabrera-Gómez, A. C. Domínguez-Brito, and D. Hernández-Sosa, "Mission specification in underwater robotics," *Journal of Physical Agents*, vol. 4, no. 1, pp. 25–34, 2010.

[108] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 01, no. 01, pp. 75–119, 2013.

[109] C. Robin and S. Lacroix, "Multi-robot target detection and tracking: taxonomy and survey," *Auton Robot*, vol. 40, pp. 729–760, 2016.

[110] A. Kolling, P. Walker, N. Chakraborty, K. Sycara, and M. Lewis, "Human interaction with robot swarms: A survey," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 1, pp. 9–26, feb 2016.

[111] A. Hocraffer and C. S. Nam, "A meta-analysis of human-system interfaces in unmanned aerial vehicle (uav) swarm management," pp. 66–80, 2017.

[112] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *2008 IEEE International Conference on Robotics, Automation and Mechatronics, RAM 2008*, 2008, pp. 736–742.

[113] D. Brugali and E. Prassler, "Software engineering for robotics [From the Guest Editors]," *IEEE Robotics and Automation Magazine*, vol. 16, no. 1, 2009.

[114] M. Aragão, P. Moreno, and A. Bernardino, "Middleware Interoperability for Robotics: A ROS–YARP Framework," *Frontiers in Robotics and AI*, vol. 3, oct 2016.

[115] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Tech. Rep., 1990.

[116] D. Nesic, J. Krueger, S. Stanciulescu, and T. Berger, "Principles of feature modeling," in *27th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[117] F. A. Bravo, A. M. Gonzalez, and E. Gonzalez, "A review of intuitive robot programming environments for educational purposes," in *2017 IEEE 3rd Colombian Conference on Automatic Control, CCAC 2017 - Conference Proceedings*, 2018.

[118] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *19th International Software Product Line Conference (SPLC)*, 2015.

[119] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, pp. 621–645, July 2006.

[120] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "The State of the Art in Language Workbenches," in *SLE*, 2013.

[121] L. Linsbauer, T. Berger, and P. Grünbacher, "A classification of variation control systems," in *16th International Conference on Generative Programming: Concepts & Experience (GPCE)*, 2017.

[122] Arcbotics, "http://arcbotics.com/lessons/sparki/," 2016.

[123] Ardublockly, "https://ardublockly.embeddedlog.com/," 2015.

[124] R. Holwerda and F. Hermans, "A usability analysis of blocks-based programming editors using cognitive dimensions," *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, vol. 2018-Octob, pp. 217–225, 2018.

[125] Parallax, "Getting started with blocklyprop," 2019. [Online]. Available: https://learn.parallax.com/tutorials/language/blocklyprop/getting-started-blocklyprop

[126] SoftBank Robotics, "http://doc.aldebaran.com/1-14/software/choregraphe/interface.html," 2017.

[127] Anki, "https://www.anki.com/en-us/cozmo/code-lab/how-it-works," 2019.

[128] EasyC, "Easyc v5 for cortex and vex iq," 2019. [Online]. Available: http://www.intelitek.com/engineering/easyc/

[129] Microbric, "https://meetedison.com/robot-programming-software/," 2018.

[130] B. Bacca-Cortés, B. Florián-Gaviria, S. García, and S. Rueda, "Development of a platform for teaching basic programming using mobile robots," *Revista Facultad de Ingeniería*, vol. 26, no. 45, pp. 61–70, 2017.

[131] Enchatnting, "http://enchanting.robotclub.ab.ca/tiki-index.php," 2017.

[132] G. F. Lozenko and L. V. D. , Vadim Olegovich Dzhenzher, "Training future teachers in computer skills in extra-curricular activity with schoolchildren," *Life Science Journal 2014*, vol. 11, no. 8s, p. 203, 2014.

[133] Makeblock, "https://www.makeblock.com/software," 2018.

[134] A. Lane, B. Meyer, and J. Mullins, *Robotics with Enchanting*, version 1.1 ed., D. Albrecht, Ed. © 2012 Monash University under a Creative Commons, 2012.

[135] Microsoft, "https://makecode.mindstorms.com," 2018.

[136] Robotical, "http://martytherobot.com/users/using-marty/program/scratch/getting-started-with-scratch/," 2018.

[137] G. Passault, Q. Rouxel, F. Petit, and O. Ly, "Metabot: A low-cost legged robotics platform for education," in *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, May 2016, pp. 283–287.

[138] Metabot, "http://blocks.metabot.fr," 2019.

[139] Ozobot, "https://ozoblockly.com/editor?lang=en&robot=bit&mode=2," 2019.

[140] R. Fojtik, "The Ozobot and education of programming," *New Trends and Issues Proceedings on Humanities and Social Sciences*, vol. 4, no. 5, 2017.

[141] Scratch, "https://scratch.mit.edu/projects/editor/?tutorial=ev3," 2019.

[142] Ö. Korkmaz, "The Effect of Scratch and Lego Mindstorms Ev3 Based Programming Activities on Academic Achievement, Problem Solving Skills and Logical Mathematical Thinking Skills of Students," *Malaysian Online Journal of Educational Sciences*, vol. 4, no. 3, pp. 73–88, 2016.

[143] Sphero, "https://www.sphero.com/education/," 2018.

[144] M. Ioannou and T. Bratitsis, "Teaching the Notion of Speed in Kindergarten Using the Sphero SPRK Robot," *Proceedings - IEEE 17th International Conference on Advanced Learning Technologies, ICALT 2017*, pp. 311–312, 2017.

[145] Wiedu, "http://www.wiedu.com/telloedu/index_en.html,," 2019.

[146] A. Guide, B. Guide, and F. Foundation, "Program Tello drone to do back ips with Scratch !" pp. 1–11, 2019.

[147] T. Lourens and E. Barakova, "User-friendly robot environment for creation of social scenarios," in *Foundations on Natural and Artificial Computation*, J. M. Ferrández, J. R. Álvarez Sánchez, F. de la Paz, and F. J. Toledo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 212–221.

[148] T. Lourens, "Programming robots using tivipe - a step by step approach using aldebaran's nao robots," TiViPE, Netherlands, Tech. Rep., 2011.

[149] TurtleBot3Blockly, "https://turtlebot-3-blockly-wiki.readthedocs.io/en/latest/launchblockly.html," 2017.

[150] A. Krishnan, "TurtleBot3Blockly Documentation," 2017.

[151] Multiplo, "http://blog.minibloq.org/," 2016.

[152] L. A. Junior, O. T. Neto, M. F. Hernandez, P. S. Martins, L. L. Roger, and F. A. Guerra, "A Low-Cost and Simple Arduino-Based Educational Robotics Kit," *Multidisciplinary Journals in Science and Technology, Journal of Selected Areas in Robotics and Control (JSRC)*, vol. 3, no. 12, p. 12, 2013.

[153] Trikset, "http://www.trikset.com/products/trik-studio,," 2019.

[154] D. Mordvinov, Y. Litvinov, and T. Bryksin, "Trik studio: Technical introduction," in *2017 20th Conference of Open Innovations Association (FRUCT)*, April 2017, pp. 296–308.

[155] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with blockly," in *Proceedings - 2017 IEEE Blocks and Beyond Workshop, B and B 2017*, vol. 2017-Novem, 2017, pp. 21–24.

[156] W. Burnett, "https://developers.google.com/blockly/," 2019.

[157] B. Kaučič and T. Asič, "Improving introductory programming with scratch?" in *MIPRO*. IEEE, 2011, pp. 1095–1100.

[158] D. C. MacKenzie, R. C. Arkin, and J. M. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, 1997.

[159] M. Völter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *7th International Conference on Software Language Engineering (SLE)*, 2014.

[160] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[161] J. García-Zubía, I. Angulo, G. Martínez-Pieper, P. Orduña, L. Rodríguez-Gil, and U. Hernandez-Jayo, "Learning to program in k12 using a remote controlled robot: Roboblock," in *Online Engineering & Internet of Things*. Springer, 2018, pp. 344–358.

[162] J. F. Gorostiza and M. A. Salichs, "End-user programming of a social robot by dialog," *Robotics and Autonomous Systems*, vol. 59, no. 12, pp. 1102–1114, dec 2011.

[163] M. Campusano and J. Fabry, "Live robot programming: The language, its implementation, and robot api independence," *Science of Computer Programming*, vol. 133, pp. 1–19, 2017.

[164] C. Ray, F. Mondada, and R. Siegwart, "What do people expect from robots?" in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2008, pp. 3816–3821.

[165] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots - Volume 8810*, ser. SIMPAR 2014. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 195–206.

[166] T. Menzies, "Evaluation issues for visual programming languages," in *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. World Scientific, 2002, pp. 93–101.

[167] D. Moody, "The physics of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, nov 2009.

[168] F. Ciccozzi, D. Di Ruscio, I. Malavolta, and P. Pelliccione, "Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems," *IEEE Access*, vol. 3536, no. c, pp. 1–1, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7576686/

[169] SPARC, "Robotics 2020 Multi-Annual Roadmap," vol. 2016, p. 325, 2015.

[170] J. Gray, S. Neema, J. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-specific modeling," in *Handbook of Dynamic System Modeling.*, 2007. [Online]. Available: http://dx.doi.org/10.1201/9781420010855.pt2

[171] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, 1999, pp. 411–420. [Online]. Available: http://portal.acm.org/citation.cfm?id=302405.302672

[172] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Trans. Software Eng.*, vol. 41, no. 7, pp. 620–638, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSE.2015.2398877

[173] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. Fickas, "Automatic monitoring of software requirements," in *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997.*, 1997, pp. 602–603. [Online]. Available: http://doi.acm.org/10.1145/253228.253493

[174] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer, "ProMoBox: A Framework for Generating Domain-Specific Property Languages," in *International Conference on Software Language Engineering (SLE)*, vol. 8706. Springer, Cham, 2014, pp. 1–20. [Online]. Available: http://link.springer.com/10.1007/978-3-319-11245-9{_}1

[175] B. Meyers, J. Denil, I. Dávid, and H. Vangheluwe, "Automated testing support for reactive domain-specific modelling languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*. New York, New York, USA: ACM Press, 2016, pp. 181–194. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2997364.2997367

[176] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit transformation modeling," in *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, 2009, pp. 240–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12261-3_23

[177] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, 2010, pp. 307–309. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869625

[178] B. Schätz, "Formalization and rule-based transformation of EMF ecore-based models," in *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, 2008, pp. 227–244. [Online]. Available: https://doi.org/10.1007/978-3-642-00434-6_15

[179] J. M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan, "Improving self-adaptation planning through software architecture-based stochastic modeling," *Journal of Systems and Software*, vol. 115, pp. 42 – 60, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216000212

[180] S. Shevtsov and D. Weyns, "Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 229–241. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950301

[181] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time," *IEEE Transactions on Software Engineering*, vol. 42, no. 1, pp. 75–99, 2016.

[182] C. Zhong and S. a. DeLoach, "Runtime Models for Automatic Reorganization of Multi-robot Systems," *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 20–29, 2011. [Online]. Available: http://doi.acm.org/10.1145/1988008.1988012

[183] L. Gherardi and N. Hochgeschwender, "RRA: Models and tools for robotics run-time adaptation," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2015-Decem, pp. 1777–1784, 2015.

[184] Y. Kim, J.-W. Jung, J. C. Gallagher, and E. T. Matson, "An Adaptive Goal-Based Model for Autonomous Multi-Robot Using HARMS and NuSMV," *The International Journal of Fuzzy Logic and Intelligent Systems*, vol. 16, no. 2, pp. 95–103, jun 2016. [Online]. Available: http://www.ijfis.org/journal/view.html?doi=10.5391/IJFIS.2016.16.2.95

[185] A. Steck, A. Lotz, and C. Schlegel, "Model-driven engineering and run-time model-usage in service robotics," *Proceedings of the 10th ACM international conference on Generative programming and component engineering - GPCE '11*, p. 73, 2011.

[186] S. Götz, M. Leuthäuser, J. Reimann, J. Schroeter, C. Wende, C. Wilke, and U. Aßmann, "A role-based language for collaborative robot applications," *Communications in Computer and Information Science*, vol. 336 CCIS, no. 1, pp. 1–15, 2012.

[187] J. Shin, R. Siegwart, and S. Magnenat, "Visual programming language for thymio ii robot," in *Conference on Interaction Design and Children (IDC'14)*. ETH Zürich, 2014.

[188] C. Vandevelde, F. Wyffels, M.-C. Ciocci, B. Vanderborght, and J. Saldien, "Design and evaluation of a diy construction system for educational robot kits," *International Journal of Technology and Design Education*, vol. 26, no. 4, pp. 521–540, 2016.

[189] T. Lourens, "Tivipe - tino's visual programming environment," in *COMPSAC*, 2004.

[190] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*, 2018.

Appendix A

Appendix - Paper B

Table A.1 shows an overview of features in the specification environments.Table A.2 reports the features related to language concepts.

	Ardublockly	Aseba	BlocklyPro	Choregraphe	CodeLab	EasyC	EdisonSoftware	Enchanting	FLYAQ	LegoMindstormsEV3	Makeblock5	Makecode	MartySoftware	Metabot	Ozoblockly	PICAXE	RobotMeshStudio	ScratchEv3	Arcbotics' SparkiDuino Sphero	TelloEduApp
Specification environments																				
MultiLanguageSupport	x	x	x	x	x	x	✓	x	x	x	✓	x	x	x	x	✓	✓	x	✓	✓	x
Editor Modes																				
Projectional	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Parser-based	x	✓	x	x	x	✓	✓	x	x	x	✓	✓	x	x	x	✓	✓	x	✓	✓	x
Semantic services																				
Error Marking	✓	✓	✓	x	x	x	✓	x	x	x	x	x	x	x	x	x	✓	x	x	x	x
Quick fixes	x	x	x	✓	x	x	✓	x	x	x	x	x	x	x	x	✓	x	x	x	✓	x
Reference resolution	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Live translation	x	✓	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Syntactic services																				
Visual highlighting	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Syntactic completion	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x
Auto formatting	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	✓	x	x
Simulation																				
Single Robot	x	✓	x	x	x	x	x	x	✓	x	x	✓	x	✓	x	✓	✓	x	x	x	x
Multi-robot	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x
Runtime specification[1]	x	x	x	✓	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Debugging	x	✓	x	✓	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x
Mission deployment[2]																				
Run time redeployment	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Over the air	x	x	x	✓	✓	x	x	x	x	✓	x	x	x	✓	✓	x	x	✓	x	x	✓
Via cable	✓	✓	✓	✓	x	✓	✓	✓	x	✓	✓	✓	✓	✓	x	✓	✓	x	✓	x	✓
Language characteristics																				
Language concepts[3]																				
Notation																				
Block-based	✓	✓	✓	x	✓	x	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Flowchart-based	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x
Graph-based	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Text-based	x	✓	x	x	✓	x	✓	x	x	✓	✓	x	✓	x	x	✓	✓	x	✓	✓	x
Custom map-based	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x
Semantics																				
Compiled	✓	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Interpreted	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x
Language Paradigm																				
DSL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GPL support	x	x	x	x	x	x	✓	x	x	x	✓	x	✓	x	x	x	✓	x	✓	✓	x
Extensibility																				
Scripting support	✓	✓	✓	✓	x	✓	✓	x	x	x	x	✓	x	x	x	x	x	x	✓	x	x

	Ar	As	Bl	Ch	Co	Ea	Ec	En	Fl	Le	Mi	Mo	Mo	Mo	Op	Pl	Ro	So	Ar	Sp Sp	Ta
General concepts																					
Control flow																					
Conditionals	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Loops	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Interrupts	x	x	✓	✓	x	✓	✓	x	x	✓	✓	✓	✓	✓	✓	x	✓	✓	x	✓	x
Multithreading forks	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Modularity-modules[1]	✓	x	✓	✓	x	x	x	x	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	✓	x
Variable Data types																					
Primitive	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x
Compound	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓
Mission Specification paradigms																					
Reactive Control	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Imperative	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Function Library																					
Arithmetic functions	✓	x	✓	✓	x	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x
String operations	x	x	✓	x	x	x	x	x	x	x	✓	✓	✓	x	✓	✓	✓	✓	x	x	x
Complex algorithms	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Multithreading	✓	x	✓	✓	x	✓	✓	x	x	✓	✓	✓	✓	x	x	✓	✓	x	x	x	x
Multirobot Hardware Support	x	x	✓	✓	x	✓	x	x	x	x	✓	x	x	x	x	x	x	✓	x	✓	x
File access																					
Read/write	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	✓	x	x
Open/close	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	✓	x	x
ReadSensor[3]																					
Event support	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓
Exception Handling	x	x	x	✓	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Actions																					
Communication actions																					
With Human	x	x	x	✓	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x
With Agent	x	x	✓	x	x	x	✓	x	✓	✓	✓	x	x	x	x	x	x	x	✓	✓	✓
Movement																					
Absolute	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	✓	✓	✓	x	x	✓	✓
Relative	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Action type																					
Instantaneous	✓	x	✓	x	x	x	x	✓	✓	✓	x	✓	x	x	✓	x	x	✓	✓	x	x
Continuous	x	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x	x	x	x	x	x	✓	x	✓
Delayed	✓	✓	✓	✓	x	x	✓	✓	x	✓	✓	x	✓	✓	✓	✓	✓	x	x	✓	✓
Actuation[4]	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓

Table A.2: Feature matrix for language concepts

[1] Modules comprise of functions and components. [2] Details of ReadSensor in Table 3.5 [3] Details of actuation in Fig. 3.13

A.1 Subject Environment Descriptions

This section provides a high-level textual description of the environments identified. In Table 3.2, for each environment we report: (i) the version we considered; (ii) whether the environment is designed for desktop computers, mobile devices or is web-based; (iii) the mobile robot that is supported and its manufacturer.

Arcbotics' SparkiDuino. [122] provides a Blockly-based programming environment for a robot called Sparki—a wheeled educational robot kit for teaching programming. The robot relies on a specific Arduino board, and the environment uses Arduino-specific software for uploading the mission to the robot.
Type of language: Block-based and text-based.

Ardublockly. [123,124] is a Blockly-based environment supporting an educational, wheeled robot called Spartan [123,124,161], manufactured by Modern Robotics, Inc. Spartan also relies on an Arduino board, and the environment is described as compatible with multiple other Arduino-based robots.
Type of language: Block-based.

Aseba. [92,93] is a collection of environments with the same languages, but different syntaxes and, therefore, editors: VPL-based (Visual programming language) [187], Blockly-based, Scratch-based, and text based for programming an educational, wheeled robot called Thymio. VPL provides icons of events and corresponding actions as building blocks.
Type of language: Block-based and text-based.

BlocklyPro. is a Blockly- and web-based environment for specifying missions for the wheeled educational robots ActivityBot robot and Scribbler robot [125].
Type of language: Block-based.

Choregraphe. [87,88,126] is a desktop-based environment that allows users to create animations, behaviors and dialogues for the NAO humanoid robot—meant for experimentation and research, as shown in Fig. A.1. Choregraphe allows to test these missions on a simulated NAO robot or directly on a real NAO.
Type of language: Graph-based.

Code Lab. provides two variants of a Scratch-based environment: Sandbox for novice programmers and Constructor for intermediate programmers, both to specify missions for a wheeled educational robot called Cozmo [127].
Type of language: Block-based and text-based.

EasyC. [128] is an environment with a flow-chart-like visual language for programming the educational robot kits (Lego-Mindstorms-like) VEX EDR and VEX IQ, used for building wheeled or stationary robots. For advanced programmers, a C-like textual syntax is also available.
Type of language: Flowchart-based.

Edison software. [129,130] is an environment for the educational wheeled robots Edison V1.0 and V2.0. The environment provides a language with two visual notations—one based on a custom block-based syntax and one based on Scratch. It also offers Python for advanced programmers.
Type of language: Block-based and text-based.

Enchanting. is a Scratch-based environment for programming the educational

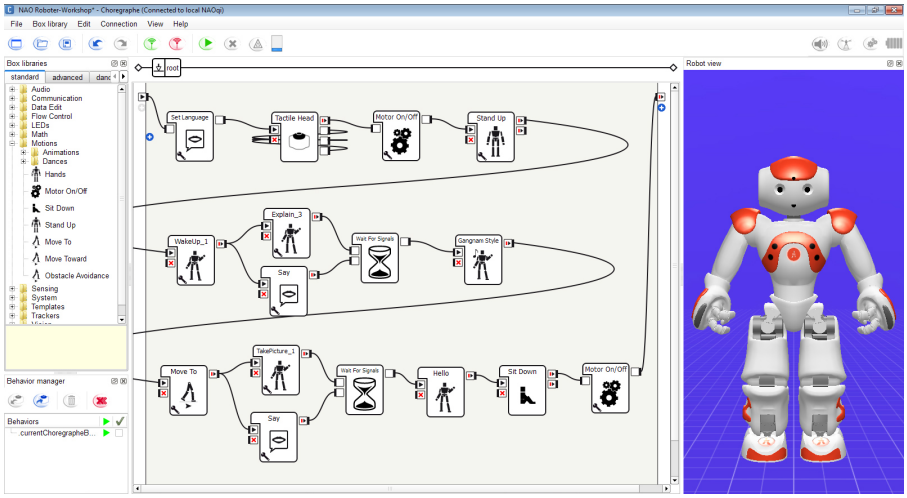


Figure A.1: The environment Choregraphe for the robot NAO

and toy robot Lego Mindstorms NXT [131, 132]—a kit like the VEX robots above (EasyC). Its successor, EV3, is supported by LEGO Mindstorms EV3 and Scratch Ev3, explained shortly.
Type of language: Block-based.

LEGO Mindstorms EV3. [94, 95] is an environment for the educational and toy robot with the same name. It provides a visual language with blocks connected to form a control flow (see also Fig. 3.12 in Sect. 3.5.3).
Type of language: Block-based and text-based.

Makeblock 5. is a Scratch-based environment for programming the educational, wheeled robots micro:bit and makeblock [133, 188]. Beyond Scratch, it also offers Python for advanced programmers.
Type of language: Block-based and text-based.

Makecode. provides an online visual editor for programming the (typically wheeled) Lego EV3 robot [135]. JavaScript code is generated from the visual program, which can be downloaded to the computer to which the EV3 robot is connected. The environment also provides a simulator, and it can also be used for other robots, such as micro:bit.
Type of language: Block-based.

Marty software. [136] is a Scratch-based environment specifically created for the humanoid educational robot marty. A screenshot of the Scratch-based visual language is shown in Fig. A.3. The environment also offers a customized Python language called martypy.
Type of language: Block-based and text-based.

Metabot. is web-based environment relying on Blockly, to create missions for the 4-legged robot Metabot v1 and v2 [137, 138]. Figure A.2 shows a mission demonstrating the use of loop control structure with a corresponding assembler code generated [137, 138].
Type of language: Block-based.

Ozoblockly. [139, 140] is a Blockly-based environment particularly for the educational, wheeled robot ozobot. The language and its visual syntax

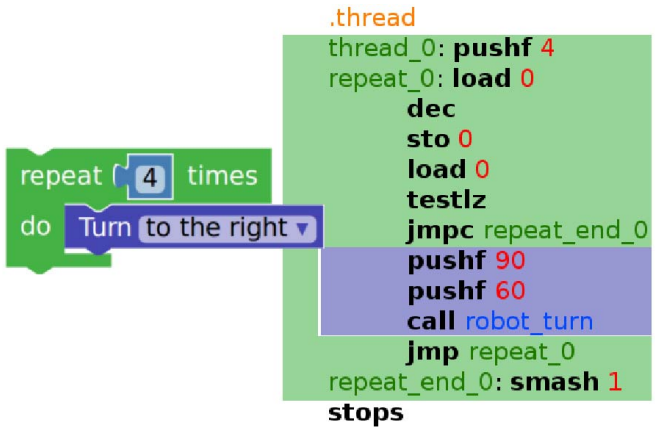


Figure A.2: Example of Metabot's visual notation (left), together with generated assembler code (right) from [137]

offer five levels of complexity, ranging from icon-based blocks to advanced programming constructs, which offer low-level control functions and advanced programming features.
Type of language: Block-based.

PICAXE. [70,71] is an environment for educational wheeled robots based on PICAXE microcontrollers, such as the PICAXE 20X2 microbot. The environment offers a language with syntaxes based on Blockly and a flowchart-like syntax, but it also comes with a Basic-style language with a textual syntax.
Type of language: Block-based, Flowchart-based, and text-based.

Robot Mesh Studio. [67] is used for programming the wheeled educational robots from VEX Robotics, such as the VEX V5, IQ, and EDR. It offers two languages: one with a flow-chart-like syntax (Flowol), and one based on Blockly. It also support C++ and Python for advanced programmers. The studio can be run online or on a Windows computer.
Type of language: Block-based, Flowchart-based, and text-based.

Scratch Ev3. [141] is the original Scratch from MIT, but tailored to support the educational robot kit Lego Mindstorms EV3. To this end, it offers dedicated language constructs for the EV3. Notably, a study has shown learning-related benefits of this Scratch-based environment over the original LEGO Mindstorms EV3 environment (see above) [142].
Type of language: Block-based.

Sphero. is an environment for programming the spherical educational robots Sphero BOLT, SPRK+, and Sphero Mini [143,144], which have a derivative resembling Star Wars' BB8 robot that was sold by the respective company under a license agreement. The language's visual syntax is based on Scratch, while JavaScript is also offered as a language with textual syntax.
Type of language: Block-based and text-based.

Tello Edu App. [145,146] is a Scratch-based environment that generates Python code for the educational drone Tello. It is essentially Scratch extended with a library that adds Tello-specific blocks (mainly drone flight controls).
Type of language: Block-based.

Figure A.3: The mission specification environment Marty software

TiViPE. [189] is a research environment for programming robots that provides support to wrap any program code modules (e.g., functions) of the supported programming language as nodes in a graph, with edges representing control and data-flows. The environment incorporates the API for the humanoid robot NAO, as demonstrated by Lourens et al [147].
Type of language: Graph-based.

Turtlebot3-blockly. [149,150] is a Blockly-based environment for programming the experimental robot turtlebot (essentially a Roomba without vacuum cleaning facilities, extensible with various sensors). It generates Python code for the turtlebot.
Type of language: Block-based.

VEX Coding Studio. [68,69] is the robot vendor's environment for programming the educational robot kits VEX EDR and VEX IQ (like EasyC). The language has a Scratch-based syntax (VEXcode Blocks) and a text-based syntax (VEXcode Text).
Type of language: Block-based and text-based.

FLYAQ. [34–36] is an experimental (research) environment to specify missions of drones, specifically the Parrot AR Drone2.0, while not being restricted to a drone model. It allows to specify missions and their parameters (e.g., flight locations), on a live map. It generates flight plans from a stack of languages, such as the monitoring mission language (which provides the user interface), to a behavioral language and a robot configuration language.
Type of language: Custom, map-based.

MiniBloq. [151,152] is an environment that can be used to program Arduino-board-based robots, such as the wheeled robot Sparki. Its language provides a custom syntax with relatively large icon-based blocks.
Type of language: Block-based.

MissionLab. [83,96] is a research environment enabling mission specification through a state-machine-based visual language. Missions can be executed on a simulator or on the following wheeled robots used for smaller commercial applications: ATRV-Jr, Urban Robot, AmigoBot, Pioneer AT, and Nomad 150

& 200.
Type of language: Graph-based.
Open Roberta. [89–91] is a web-based, educational, and Blockly-based environment for programming a variety of robots: Lego Mindstorms EV3 amd NXT, Calliope mini, micro:bit, Bot'n Roll, NAO, and BOB3. It can either be run on the cloud or installed on a local server. The environment generates Code in Python, Java, Javascript, and C/C++ depending on the target robot.
Type of language: Block-based.
RobotC. [65, 66] is an educational environment providing a language that tries to be close to natural language, mainly through more natural language keywords and expressions (e.g., "Understood==True"). It allows programming the VEX, LEGO Mindstorms EV3 and NXT, and other Arduino-based robots.
Type of language: Block-based and text-based.
TRIK Studio. [153, 154] is an educational tree-based environment in which blocks connected to the chart are symbols of functions the block does. The studio provides an interactive simulation mode and supports multiple robot types, such as the drone Geoscan Pioneer and the wheeled robot kits LEGO Mindstorms EV3 and NXT.
Type of language: Graph-based and text-based.
PROMISE. [8] provides a graphical and a textual syntax for mission specification for multi-robot applications. The environment[1] allows the seamless integration and usage of both syntaxes to specify different aspects of the same mission. The language provides a list of operators—which are inspired by the behavior trees' operators [190]—that can be composed to encode complex missions. These operators are interconnected following a behavior tree style and notation. The language relies upon a catalog of patterns based on temporal logics, which encodes recurrent robotics missions from literature [1]. PROMISE automatically generates and forwards the missions to be achieved by the robotic application, decomposing the overall specification into robot-specific missions. PROMISE is intended to be robot-agnostic, so it could be integrated with any robot. *Type of language*: Graph-based and text-based.

A.2 Additional Online Resources

Table A.3 provides links to online resources for the respective specification environment resources

[1] https://github.com/SergioGarG/PROMISE_implementation

Table A.3: Links to respective specification environment resources

Environment	URL Link
Arcbotics' SparkiDuino	http://arcbotics.com/lessons/sparki/
Ardublockly	https://ardublockly.embeddedlog.com/ , https://modernroboticsinc.com/product-category/spartan/
Aseba	https://www.thymio.org/en:star
BlocklyPro	http://blockly.parallax.com/blockly/editor/blocklyc.jsp?project=27294#
Choregraphe	http://doc.aldebaran.com/1-14/software/installing.html
Code Lab	https://anki.com/en-us/cozmo/create-with-cozmo/constructor/create.html
EasyC	https://www.vexrobotics.com/easyc-v5.html
Edison software	https://meetedison.com/robot-programming-software/
Enchanting	http://enchanting.robotclub.ab.ca/tiki-index.php
FLYAQ	http://www.flyaq.it/
LEGO Mindstorms EV3	https://www.lego.com/en-us/mindstorms/downloads, https://education.lego.com/en-us/downloads/mindstorms-ev3/software#MicroPython
Makeblock 5	https://www.makeblock.com/software
Makecode	https://makecode.mindstorms.com/#editor
Marty software	http://martytherobot.com/users/using-marty/program/scratch/getting-started-with-scratch/
Metabot	http://blocks.metabot.fr/#
Ozoblockly	https://ozoblockly.com/editor?lang=en&robot=evo&mode=5
PICAXE	http://www.picaxe.com/software
Robot Mesh Studio	http://docs.robotmesh.com/ide-project-page
Scratch Ev3	https://scratch.mit.edu/projects/editor/?tutorial=ev3
Sphero	https://www.sphero.com/education/
Tello Edu App	https://play.google.com/store/apps/details?id=com.wistron.telloeduIN
TiViPE	https://www.tivipe.com/2016/08/30/merging-modules/#more-461
Turtlebot3-blockly	https://turtlebot-3-blockly-wiki.readthedocs.io/en/latest/
VEX Coding Studio	https://www.vexrobotics.com/vexedr/products/programming
MiniBloq	http://blog.minibloq.org/p/documentation.html
MissionLab	www.cc.gatech.edu/aimosaic/robot-lab/research/MissionLab/
Open Roberta	https://lab.open-roberta.org/
RobotC	http://www.robotc.net/graphical/
TRIK Studio	http://www.trikset.com/products/trik-studio#download
PROMISE	https://github.com/SergioGarG/PROMISE_implementation