International Journal of Software and Systems Modeling (SoSyM) manuscript No.
(will be inserted by the editor)

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

Swaib Dragule · Thorsten Berger · Claudio Menghi · Patrizio Pelliccione

Received: date / Accepted: date

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 towards 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

S. Dragule
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
Department of Computer Science,
Makerere University, Kampala, Uganda
E-mail: dragule@chalmers.se

T. Berger
Center of Computer Science
Ruhr University Bochum, Germany;
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
E-mail: thorsten.berger@rub.de

C. Menghi
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg, Luxembourg
E-mail: claudio.menghi@uni.lu

P. Pelliccione
DISIM, University of L'Aquila, Italy;
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
E-mail: patrizio.pelliccione@univaq.it

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 are projected to grow by 50% each year.

Different techniques have been proposed for engineering the various aspects of robotic behavior [37, 39,27,32,25,26,108,92], such as interoperability at the

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

human-robot (or human-swarm) level [53,43] and at the software-component level in middlewares [78], or multi-robot target detection and tracking [92].

Engineering robotics control software is challenging [37]. Specifying the behavior of a robot, typically called the robot's mission, is far from trivial. Specifically, a *mission* is a description of the high-level behavior a robot must perform [39,71,37,27]. As such, a mission coordinates the so-called skills of robots, which represent lower-level behaviors.

Developing missions requires substantial expertise [15,3]. For instance, logical languages, such as LTL, CTL, or other intricate, logic-based formalisms to specify missions, are complex for users with low expertise in formal and logical languages [65,71,70].

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 feature in 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 [113, 11,71,37]. Researchers and practitioners have invested substantial effort into achieving end-user-oriented programming environments for robots [113,10,11,18,83]. In fact, almost every commercial mobile robot nowadays comes with a *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 to improve our empirical understanding of the current state-of-the-art in mission specification. Specifically, the 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 are created, designed, or particularized. We consider a mission specification environment as a collection of tools that facilitates the 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 [50, 82]—a common method to analyze and document a particular domain [24,30,109,58,59].

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 identified such environments from various sources, including the Google search engine and the research literature.

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

With our analysis we identified a total of 30 environments and designed a feature model with 137 features, 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, the language concepts utilized by end-users. We also present and discuss the representation of these features in the individual environments and languages. 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 in 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 needs.

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,

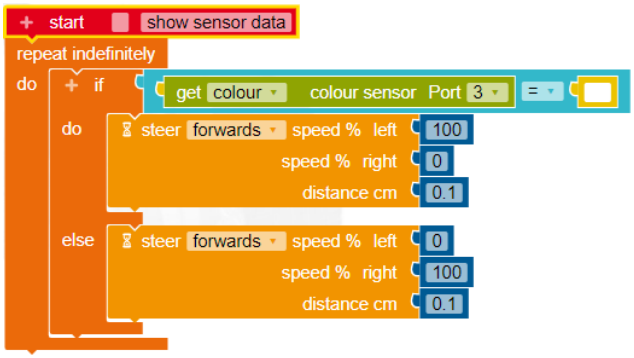

Fig. 1 Block-based mission of a robot patrolling a perimeter wall, expressed in Open Roberta

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 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.

For illustration, let us consider a very simple mission for a line-following Lego EV3 robot, specified in one of our studied environments, Open Roberta. This mission has been specified using the Open Roberta environment, as shown in Fig. 1 with a corresponding textual code in Listing 1. The code in the block-based syntax coordinates sensor input and tasks (here, robot movements) in a certain imperative way. Figure 2 shows the mission executed in Open Roberta's simulator. Listing 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 mission specification (Fig. 1), contains intricate boilerplate code hidden by the latter's language.

Fig. 2 Simulation of the line-following Lego EV3 robot

3 Methodology

We now explain our methodology for identifying end-user-oriented mission specification environments (Sec. 3.1) and for classifying and analyzing their features (Sec. 3.2).

3.1 Identification of Environments (RQ1)

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

Inclusion and Exclusion Criteria During our systematic environment identification process (explained below), we applied the following inclusion and exclusion criteria to filter candidate environments.

```
#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 1 Target C code generated from the mission in Fig. 1

Table 1 Selected environments by data sources after applying inclusion and exclusion criteria. 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, 14 selected	**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, 20 selected (8 new)	MissionLab, Choregraphe, LEGO Mindstorms EV3, Sphero, TiViPE, Aseba, Robot Mesh Studio, Edison software, Makeblock 5, TRIK Studio, Ardublockly, FLYAQ, **PICAXE, Open Roberta, SparkiDuino, VEX Coding Studio, Metabot, Marty software, Tello Edu App**, and **Code Lab.**
Google search: 373 candidates, 23 selected (2 new)	MissionLab, Choregraphe, LEGO Mindstorms EV3, Sphero, TiViPE, Aseba, Robot Mesh Studio, Edison software, Makeblock 5, TRIK Studio, Ardublockly, MiniBloq, FLYAQ, PICAXE, Open Roberta, SparkiDuino, VEX Coding Studio, Metabot, Marty software, Tello Edu App, Code Lab, **BlocklyProp**, and **Ozoblockly.**
Snowballing: 80 candidates, 15 selected (3 new)	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 selected[1]	**Turtlebot3-blockly, Makecode**, and **Scratch EV3**

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

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.

Identification of Candidate Environments. We identified our subject environments from our data sources using the following steps, as illustrated in Fig. 3.

Authors' Experience. Based on our experience in robotics software engineering, we assembled a list of 26 candidate environments from which 14 were selected after applying the inclusion and exclusion criteria.

List of Mobile Robots. A list of 59 commercial mobile robots was created based on past experience of the authors (e.g., the authors were aware of many educational robots, such as Thymio, Sphero, or NAO) and a simple Google search for mobile robots. From the robots' web pages, we identified the software that was offered

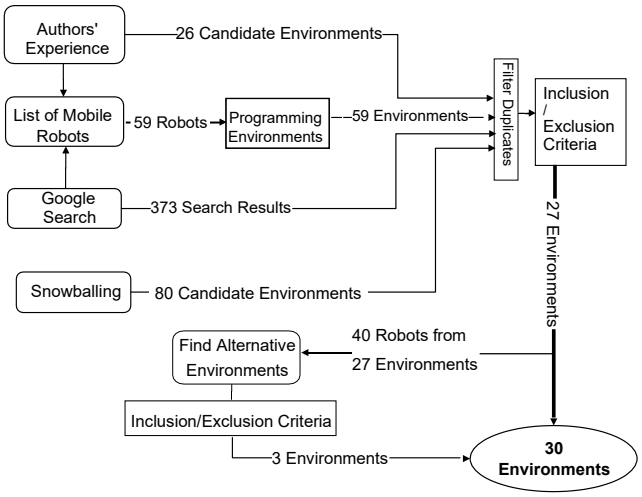

Fig. 3 Identification of environments

for programming the robot missions. We obtained 59 environment candidates, from which 20 environments were selected after applying the inclusion and exclusion criteria, eight of the 20 environments were new and not selected in the previous step.

Google Search. We searched with the search 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 (a common phenomenon with Google search). Out of the 373, we selected 23 environments after applying the inclusion and exclusion criteria; only 2 environments were new and not selected in a previous step.

Snowballing. Based on a list of six survey papers we were aware of based on 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 (backward snowballing) and all papers citing it (forward snowballing), while ignoring duplicates. We identified 12 from Biggs et al. [10], 44 from Bravo et al. [14], 14 from Jost et al. [48], 0 from Luckcuck et al. [65], 6 from Nordmann et al. [83], and 7 from Hentout et al. [42], totaling 80 candidates. Out of the 80 candidates, 15 were selected by applying the inclusion and exclusion criteria, with 3 being new and not identified in a previous step. Table 1 shows environments identified from particular data sources.

These four steps identified a total of 538 candidate environments, which led to 27 environments based on the inclusion and exclusion criteria. Note that, before applying the latter, we always removed duplicates as well.

Find Alternative Environments. Finally, 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. This way, we identified 40 robots from the 27 environments selected above, which yielded 3 more environments after applying the inclusion and exclusion criteria.

In summary, we identified 30 environments for analysis, as shown in Table 1 classified by data source.

3.2 Analysis of Identified Environments (RQ2)

Our analysis goal was to identify the characteristics that distinguish our subjects in the form of features [8] and to organize them in a feature model [50,82]. Performing such a feature-based analysis is a common method for describing the design space of DSLs [110, 109] and other technologies, such as model transformations [24], language workbenches [30] or variation control systems [58,59].

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 was 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 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 to better understand how the environments are used in specifying missions.

Through 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 a consensus.

4 The Environments (RQ1)

We now summarize the identified environments. We classify them by the kinds of syntax they offer: block-, flowchart-, graph-, text- or map-based syntaxes. Table 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.

Appendix B provides further details about each environment, and Appendix C additional online resources.

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 [45].

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 2. There is some attempt to use these languages for industrial use.[2]

The syntaxes of these block-based languages are typically implemented using the popular open-source libraries Blockly [88,17] and Scratch [51]. 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 and offers a code-generation framework to generate code in the target (general-purpose)

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

Table 2 Subject environments and their characteristics

Environment	Syntax	Runtime environment	Mobile robots supported	User domain
SparkiDuino 1.8.7.1 [4]	Block-based, text-based	Desktop	Sparki	Education
Ardublockly 2.4.220 [5,44]	Block-based	Desktop	Spartan	Education
Aseba 3 [105,67]	Block-based, text-based	Desktop	Thymio II	Education
BlocklyProp 1.1.1.455 [86]	Block-based	Desktop	ActivityBot, Scribbler 3 Robot	Education
Choregraphe 2.1 [96,90,77]	Graph-based	Desktop	NAO, Romeo	Education
Code Lab [2]	Block-based, text-based	Desktop, mobile-app	COZMO	Education
EasyC 5 [28]	flowchart-based	Desktop	VEX EDR & VEX IQ	Education
Edison software [75,7]	Block-based, text-based	Web	Edison robot	Education
Enchanting 0.2.4.3 [29,64]	Block-based	Desktop	LEGO Mindstorms NXT	Education
FLYAQ [33,25,12]	Custom map-based	Desktop, web	Parrot AR drone	Education, research
LEGO Mindstorms EV3 1.3.1 [31,16]	Block-based, text-based	Desktop, mobile-app	LEGO Mindstorms EV3	Education
Makeblock 5 [68,57]	Block-based, text-based	Desktop, web	Codey rocky, mbot, Airblock	Education
Makecode 1.0.11 [76]	Block-based	Web	LEGO Mindstorms EV3	Education
Marty software 3.0 [95]	Block-based, text-based	Web	Marty	Education
Metabot [87,74]	Block-based	Web	Metabot	Education
Ozoblockly [85,34]	Block-based	Web	Bit, Evo	Education
PICAXE 6 [89,47]	Block-based, flowchart-based, text-based	Desktop, mobile-app	PICAXE 20X2 Microbot	Education
Robot Mesh Studio 2.0.0.6 [73]	Block-based, flowchart-based, text-based	Desktop, web,	VEX IQ, VEX EDR, VEX V5	Education
Scratch EV3 [101,54]	Block-based	Web	LEGO Mindstorms EV3, WeDo 2.0	Education
Sphero 5.2.0 [103,46]	Block-based, text-based	Desktop, web, mobile-app	Sphero Bolt, Spark+, Sphero Mini	Education
Tello Edu App 1.1.2.23 [114, 41]	Block-based	Mobile-app	Tello drone	Education
TiViPE 2.1.3 [63,62]	Graph-based	Desktop	NAO	Education, research
Turtlebot3-blockly [107,55]	Block-based	Desktop	TurtleBot3	Education, research
VEX Coding Studio 18.08.2010.100 [97,20]	Block-based, text-based	Desktop	VEX IQ, VEX EDR	Education, research
MiniBloq 0.83 [81,49]	Block-based	Desktop	DuinoBot, Sparki	Education
MissionLab 7.0 [6,108]	Graph-based	Desktop	ATRV-jr, Urban robot, Amigobot, Pioneer AT, Nomad 150,and 200	Education, research
Open Roberta 3.0.3 [45,48,52]	Block-based	Desktop, web	Micro:bit, LEGO Mindstorms EV3 and NXT, NAO, WeDo, BoB3, Nepo4Aduino, Bot'n Roll, calliope mini	Education
RobotC 4 [94,99]	Block-based, text-based	Desktop	VEX IQ, VEX CORTEX, LEGO Mindstorms EV3 and NXT	Education
TRIK Studio 3.2.0 [106,80]	Graph-based, text-based	Desktop	LEGO Mindstorms EV3 and NXT	Education
PROMISE [35,36]	Graph-based, text-based	Desktop	TIAGo, ITA, Turtlebot2	Research

language [87]. Scratch is similar to Blockly, but developed by the MIT media laboratory [51]. The library can be extended to add custom, end-user-oriented blocks.

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 a 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. 4 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 the main program and a subroutine (FLASH). After creating the mission in the flowchart editor, a separate text file is generated when the mission is compiled.

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.

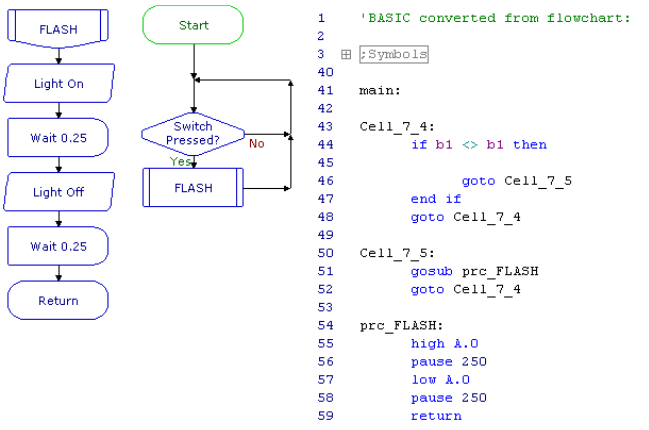

Fig. 4 Left: A mission specified in PICAXE's flowchart-based language. Right: generated target code in PICAXE's BASIC language (from [89])

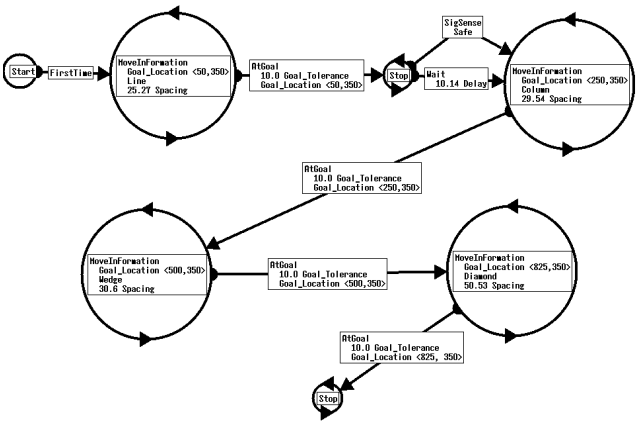

Fig. 5 Graph-based state transition diagram for a multi-robot scouting mission in MissionLab (from MacKenzie et al. [66])

Figure 5 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. 5 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 the rectangle boxes) representing conditions in which to advance to a new state. In our example, the robots start in line formation, then switches to column formation, then wedge formation, and finally the diamond formation.

4.4 Environments with Text-Based Languages

Most of the textual syntaxes offered by our 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 general-purpose language (GPL) in addition to its main DSL for mission specification. Notable exceptions are Aseba and PROMISE, whose DSLs also offer a textual syntax. In the other environments, the GPLs with the textual syntax used include, for instance, Python, C/C++, Java, Javascript, and BASIC. Figure 6 shows a text-based mission specified for a robot to follow a line using Edison software.

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

Figure 7 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.

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 on the environments' capabilities to use these languages. We identified 137 features that distinguish our environments and that we organized in

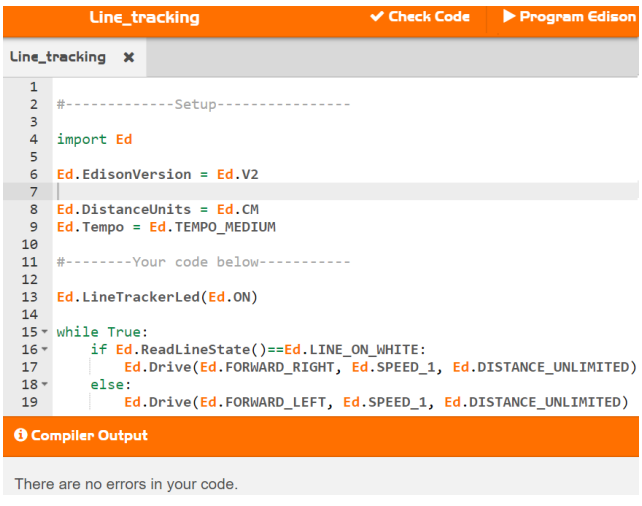

Fig. 6 A text-based mission for line tracing specified in Python within the environment Edison software (from its website [75])

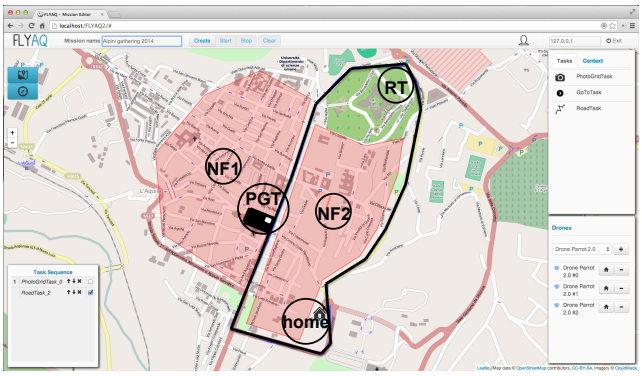

Fig. 7 A patrolling mission in FLYAQ (from Ruscio et al. [98]), where a drone follows a street, repeatedly takes photos (at specified distances), and avoids no-fly zones. NF1 and NF2 are no-fly zones, RT is road task to follow a street while PGT is photo grid task indicating where photos can be taken.

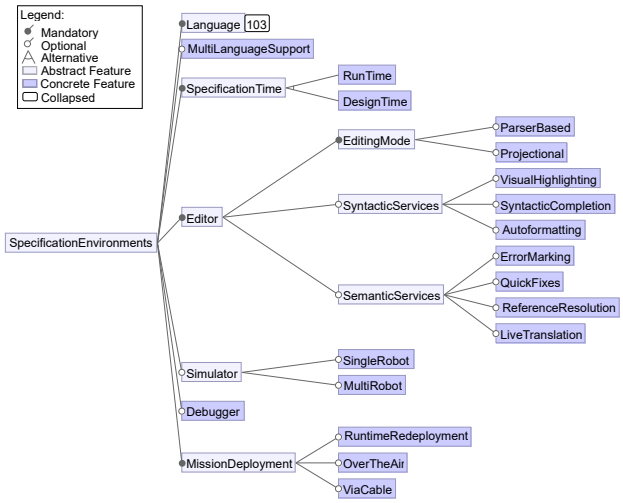

Fig. 8 Overview of the features (137 features in total)

a feature model. In the following, Sec. 5.1 presents the *high-level* features extracted from our subject environments, Sec. 5.2 presents the *language-specific* features we identified, and Sec. 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 A, in Table 5 (high-level environment features and language characteristics) and in Table 6 (language concepts offered by the DSLs).

5.1 Specification Environments

Figure 8 shows the top-level features characterizing our subject environments: Language, MultiLanguageSupport, Editor, Simulator, Debugging, SpecificationTime, MissionDeployment. The support of these features by each environment is detailed in Table 5 (upper half) in Appendix A.

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 5.2 and 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 one 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 toolings in our environments offer 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) [112, 9, 100]. 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

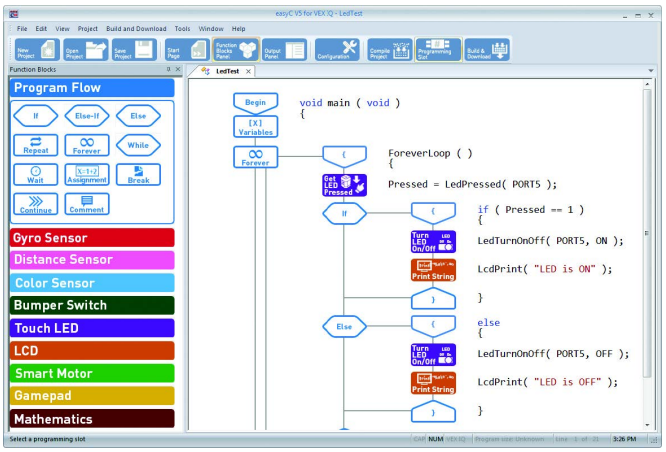

Fig. 9 Visual and textual syntax side-by-side in EasyC's projectional editor (from [28]).

(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 Fig. 9. 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 cannot fit if it is not syntactically correct. In textual notations such as Edpy in Edison software, projections of the next possible text to type are suggested while specifying a mission.

The majority of our environments (26) provides so-called syntactic services (feature SyntacticServices) [30]. These support developers in creating syntactically correct missions, according to the language's syntax. We identified three syntactic services:

- *Visual Highlighting* consists of 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 the 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) [30]. 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 to fix a problem when selected, such as interactive tooltips in PICAXE, pop up help, and autocomplete in Edpy of Edison software.
- *Reference resolution* links declarations to the usage of variables. For instance, invalid variable names are pointed out in MiniBloq.
- *Live translation* is the immediate generation of code from the mission as it is specified, which is displayed side-by-side to the graphical notation, such as in EasyC (Fig. 9).

As seen in Table 5, five of the environments offer semantic services to the end-user.

Simulator. As many as 10 of the 30 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[3] and Gazebo,[4] respectively).

Debugging. We identified debugging support in nine of the 30 environments. Specifically, we found a variety of debugging tools, including the live monitoring of sensor data, actuator states, and 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 specifications. Five of our environments (namely Turtlebot3-blockly, Sphero, EasyC, MissionLab, and Choregraphe), 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.
- *Via cable.* Supported by 23 environments, the cable options for mission deployment observed are USB cable, Ethernet cable, and custom cables.
- *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] https://ardupilot.org/mavproxy
[4] http://gazebosim.org

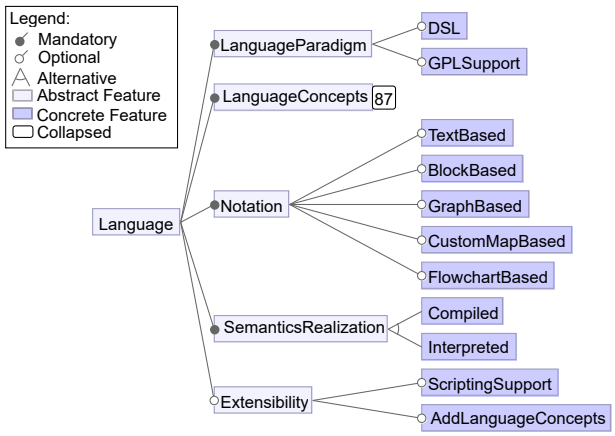

Fig. 10 General language characteristics identified

5.2 General Language Characteristics

As illustrated in the feature model in Fig. 10 and the feature matrix in Table 5 (Appendix A), we identified the following general characteristics in which the languages differ. These are represented by the features Notation, SemanticsRealization, LanguageParadigm, and Extensibility.

The actual concepts offered by the languages will be discussed in Sec. 5.3. Here, we discuss their core characteristics.

Notation. As already discussed above in Sec. 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. According to definitions provided in Sec. 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 to which the motor connects. As many as 13 of the 30 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, such as EasyC, as shown in Fig. 9.

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 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 GPLs usable directly in the environment. Examples of the latter are C/C++, Java, and Python, as shown in Table 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.

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, and 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 textual languages available in some environments. As shown in Fig. 11 and Table 6, we classified the concepts into the following features: MissionSpecificationParadigm, ControlFlow, Modularity, DataTypes, EventSupport, ReadSensor, Actions, ExceptionHandling, FileAccess, FunctionLibrary, Multithreading, and MultiRobotHardwareSupport. Below, we discuss details of the concepts.

MissionSpecificationParadigm. In the robotics domain, the two programming models typically used are imperative programming and reactive programming, with an explicit or implicit expression of control

Table 3 Kinds of notation supported by the environments. The visual notations typically belong to the primary DSLs of the environments; the textual notations typically to additional languages supported.

notation	environment
Visual	
Block-Based	SparkiDuino, Ardublockly, Aseba, BlocklyProp, 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++	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 4 Target general-purpose language when code is generated by the environment

target language	environment
C/C++	RobotC, BlocklyProp, Robot Mesh Studio, 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)

flow. In reactive control, the perception obtained via sensors and the actions is directly coupled, as can be seen in Aseba, where the control flow is also implicit. The idea is that a robot can respond timely in a dynamic environment. Imperative programming explicitly expresses the control flow via control-flow statements. Thereby, it is up to the programmer to encode (and assure) reactions to events using a viable control flow in the program. Another paradigm, recently discussed in the literature, is goal-based specification [71,70], where the goals are expressed (potential patterns over their order of fulfillment), but not the behavior necessary to achieve those goals.

Among the 30 environments, the majority (29) follow the imperative paradigm, with an explicit expression of control flow. Only two follow the reactive control paradigm: Aseba and MissionLab. In Aseba, events, which act as triggers, are matched with corresponding actions. See Fig. 12 for an example. MissionLab relies on state machines as the underlying modeling technique. As such, it can be classified as reactive, since state transitions are triggered by events upon the current state.

However, the boundaries between imperative and reactive programming are blurred. 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. An interesting environment is PROMISE, relying on a behavior tree language [35]. Behavior trees [22,39], originally coming from the games domain, encode control flow explicitly in a tree structure, which is executed via time-triggered traversal. While primarily imperative, by a respective ordering of the tree, reactions can be placed prominently in the tree structure to assure they are executed first.

ControlFlow. Not surprisingly, almost all (29) languages offer several kinds of statements for explicitly expressing control flow. Typical examples of conditionals we found are *if-do*, *if*, *if-else*, and *switch*. Explicit loop concepts are also common (28), 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. 13 (*repeat count*) and in RobotC (*repeat until*). Notably, even though, MissionLab with its

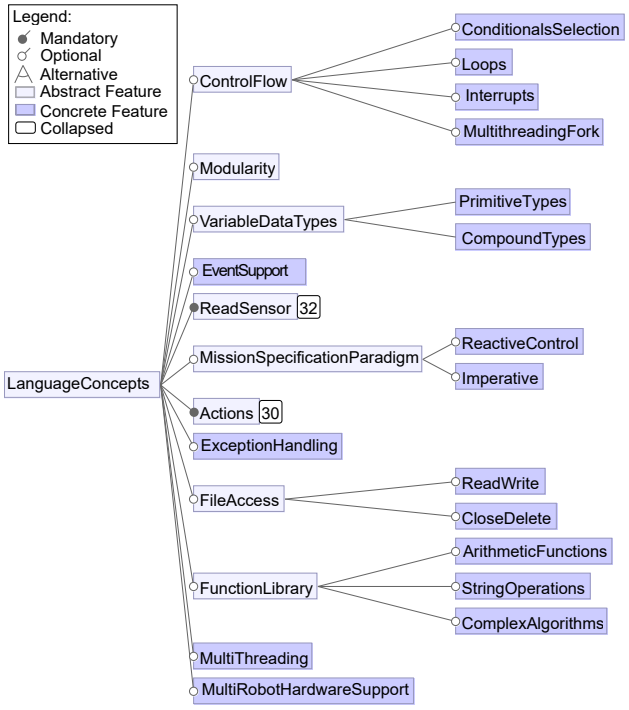

Fig. 11 Language concepts

state-machine-like language has no explicit concept for loops, they can still be expressed in the missions using a respective structuring of transitions (which can be as easy as a self-transition). 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, where the *start block* creates a thread and *sleep for x seconds* forces a thread to yield. See the feature Multithreading below for more information.

Modularity. The majority of the environments (17/30) offer modularization concepts to structure larger missions. We found functions that are graphically represented using dedicated blocks, such as functions or procedures, or modules in the environments. Each function gets input

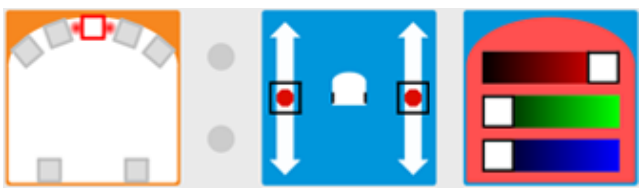

Fig. 12 An example of Aseba's block-based syntax for its language VPL—an event-based language consisting of event-action pairs. Here, when an object is detected (event), the top color (action) is set to red.

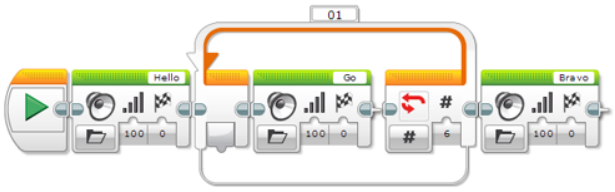

Fig. 13 Program control flow example in LEGO Mindstorms EV3: The robot says "Hello" once, then "Go" six times, then "Bravo" once.

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. These 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 modules with symbolic icons 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 can then be invoked and used.

DataTypes. The environments' 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, array, table,* and *list*. 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 enumeration (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 used to process data. For the latter, since it would be a subjective assessment, we do not detail which environment provides such algorithms in Table 6. For the others, essentially, we found the 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. 14 and Table 6. Specifically, actions can be of type:

- An instantaneous action (14) is executed immediately and only once, such as take photo in FLYAQ.
- A continuous action (14) 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.
- A delayed action (19) starts after a delay, which can be due to an event or specified time to wait.

Most environments support instantaneous actions. The other actions (continuous and delayed) typically require some notion of time and timer manipulation,

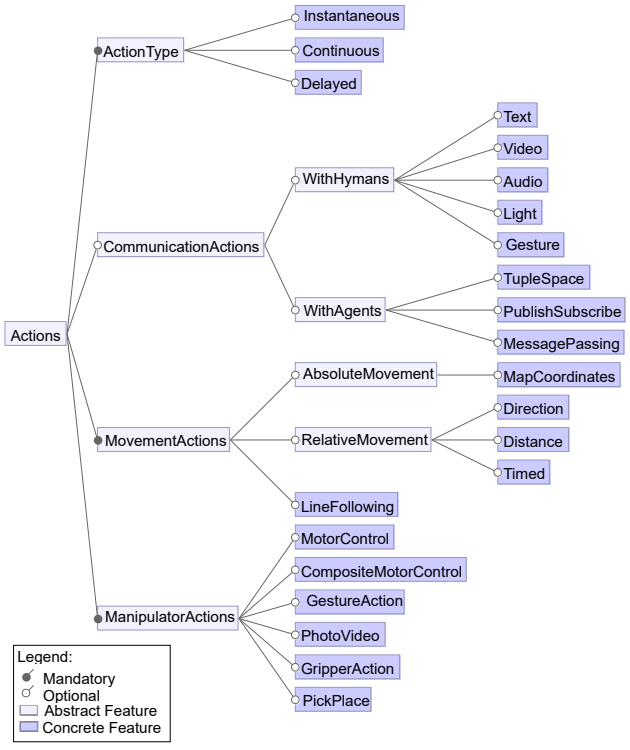

Fig. 14 Kinds of actions supported by the languages

where we found different realizations. In LEGO Mindstorms EV3, a *timer* block can be used together with a *loop* or *wait* block. Similar time constructs also exist in other environments, such as the statements *wait time* and *elapsed time* in Ardublockly, the statements *set roll time* in seconds as a variable, *time elapsed, get current time, set timeout,* and *set time interval* in Sphero, and finally the statements *set timer (seconds), timer,* and *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.

CommunicationActions. This includes interacting with humans (7) or other agents (12). Communication with humans can be of the form text, video, audio, light, or gesture. Communication with 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. Message-passing refers to a loose way of communicating, where robots send messages (e.g., via infrared), but have no guarantee of others receiving the message. It is also up to the developer to implement the reception of messages, for instance, Edison software provides a Boolean function to check whether an infrared message was received. 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 messages between robots. FLYAQ supports synchronization and communication messaging among drones at runtime. TRIK Studio supports sending messages to other robots. For what concerns communication to humans, examples are *speak short phrase* in Code Lab, *say text* to the environment in TRIK Studio and TiViPE. In Choregraphe, a robot can speak text to humans. In SparkiDuino, humans interact with the robot through *beep* and status led colors. 11 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, either with absolute (e.g., map coordinates) or relative (e.g., *direction, distance, or travel time*) parameters specifying the target. Few of the environments support

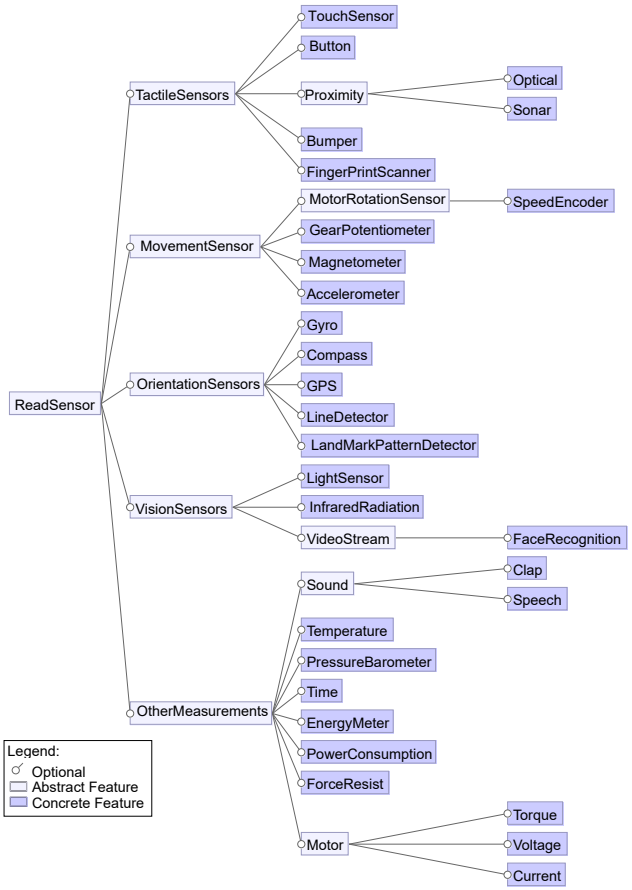

Fig. 15 Kinds of sensors supported by the languages

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.

ManipulatorActions refer to skillful ways robots can control manipulators (e.g., a gripper or a robotic arm) to handle objects in the environment. Not surprisingly, the languages offer dedicated concepts in the form of statements that can be used, which we represent by the seven features under ManipulatorActions in Fig. 14. Specifically, we found a variety of very low-level motor actuation commands, which users can use to control manipulators, as well as commands (CompositeMotorControl) that provide complex movements involving multiple motors, e.g., to control grippers or robotic arms.

EventSupport. 24 of our languages provide event support, which concerns the languages' abilities to handle events, such as creating event handlers, declaring the types of events that can be recognized; and specifying the mechanisms of synchronizing events to subsequent actions. For instance, in VEX Coding Studio, the common language constructs for event support 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 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. The sensor concepts identified are shown in Fig. 15. We classified the sensors into: tactile sensors, movement sensors, orientation sensors, vision sensors, and sensors for other measurements. Tactile sensors measure physical interactions with the environment or humans, which includes touch sensors, buttons, proximity sensors (via optical detection or via sonar), bumpers, and even a fingerprint sensor (in BlocklyProp). Given the vast number of language concepts for reading different sensor data, we did not analyze the exact availability of individual sensors in the respective environments, but present the kinds of sensors we identified for which the languages provide support. Movement sensors measure the actual robot movement via, as we identified, a motor rotation sensor (with dedicated encoders to determine velocity), a gear potentiometer, a magnetometer, or an accelerometer. Orientation sensors we identified include gyro sensor, compass sensor, GPS, line detector, and landmark pattern detector—the latter two via visual information (so, they could also be classified as vision sensors). The vision sensors we found include light (intensity) sensors, infrared radiation sensors, cameras offering a video stream, and face recognition support based on the latter. Further support for reading sensor data that does not fall into any of these categories, organized in the feature OtherMeasurements include: sound sensors (some environments also offer support for recognizing claps or speech upon sound sensors), temperature/thermometer, barometer (air pressure sensor), timer, energy meter (reads battery energy), power meter (measures power consumption), force sensors, and various motor sensors (measuring motor torque, voltage, or current).

It is interesting to note that some languages allow to obtain sensor data via respective functions, while others only provide events for certain sensor data. For instance, some allow directly reading the state of a bumper sensor, while others abstract that away via functions that check for obstacles detected (e.g., Edison software). Some environments are flexible and allow

both, e.g., Choregraphe with its class *AL_Memory* [90], allowing reading for instance the foot bumper data, but also subscribing to events generated from it.

Finally, most languages offer additional libraries providing off-the-shelf algorithms for computations over sensor data (e.g., face recognition), contributing to the intelligence of the robot. In TiViPE and Choregraphe, the NAO robot senses sound data, and intelligently determines the direction of the sound. Picture frames are captured by the NAO robot at intervals to determine moving objects. The NAO robot tracks known people by comparing all people in a video with known ones, thereby automatically identifying unknown people. In Open Roberta and Choregraphe, the 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 colors at once, analyze objects for advanced tracking and path planning.

ExceptionHandling. We identified exception handling constructs in Open 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. 11 of the 30 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), 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()*, and *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. However, the reuse of missions across robot models is limited. There are always concepts in the languages specific to certain hardware. In the ideal case, when none such concepts are used, the robot model can just be changed, but we did not see that any of these environments support that. Mostly, changing the robot model requires recreating the mission. Sometimes, only the initialization block is robot-specific; in this case, the mission is reusable, which we observed in very few environments. As an example, consider Sphero, which has some missions that are compatible with more than one robot, even 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.

6 Findings and Implications

We now discuss our main findings and their implications for practitioners, language vendors, and the research community.

6.1 Language Engineering

The environments are especially interesting from a language engineering perspective. Specifically, none of them has been developed using standard language-engineering technology, such as language workbenches [30]. While we cannot judge their development processes, it does not seem that the vendors followed textbook methods [23,13,100,56], creating abstract and concrete syntax as well as static and dynamic semantics. Whether language-engineering methods and technologies are too difficult, do not provide sufficient benefit, or the vendors are just not aware of it, constitutes an interesting question for future studies.

In this respect, our study and the environments we studied can constitute a benchmark for the language-engineering community—steering research into lightweight language-engineering tools that are increasingly

adopted in practice, especially in the highly relevant domain of robotics, which needs effective mission-specification languages and would probably benefit from language-engineering technology.

Considering the implementation of our environments, it appears that Blockly and Scratch have become standard tools for engineering the visual syntax and the respective editing infrastructure. The vendors appear to be pragmatic here. They often used an off-the-shelf language and made it domain-specific by tailoring it down to the needs of mission specification, and by creating a visual, domain-specific syntax.

Notably, Blockly and Scratch appear to have made projectional editing popular and efficient to use. They are also a pragmatic way of realizing projectional editing [9,112,100], requiring much less intricate knowledge than using a projectional editing language workbench, such as Jetbrains Meta Programming System. Of course, Blockly and Scratch lack most of the advanced capabilities of projectional editing, such as language composition, views, and even more flexible syntax, combining textual and visual ones. To what extent lightweight libraries such as Blockly and Scratch can be extended towards these advanced capabilities is an interesting open question. Specifically, language composition and views could be powerful techniques to foster more extensive tailoring of mission-specification languages to the different end-users, who might want to use different language levels and views abstracting over complex missions (or projecting extra information relevant for advanced users).

Another advantage of projectional editing is its ability to use DSLs without having the traditional and heavyweight language infrastructure with transformations, where the feedback is provided only implicitly or late. In projectional editors, the feedback can be given immediately, easing the use (or combination) of different languages, as opposed to traditional model-driven engineering. An interesting direction would be to extend the feedback to runtime information, to help debug complex missions. An interesting work to consider here is that of Miguel Campusano et al. [19] on "live robot programming." The authors present a language that supports live feedback. It helps end-users in rapid creation and variation of robot behavior at run-time. This approach, however, does not provide the end-user with domain constructs to simplify the programming effort during mission specification.

Finally, most environments focus on languages that are very tailored-down versions of imperative programming languages. Others are built upon well-known specification languages, including state machines (MissionLab), behavior trees (PROMISE), and custom DSLs for drone operations (FLYAQ). As such, they all offer a very limited syntax and require a very exact specification adhering to this syntax. None is more flexible by allowing, for instance, natural-language-like specifications. In the future, the language-engineering community might want to look into frameworks or even language workbenches allowing these kinds of languages. An interesting work in this direction is done by Gorostiza et al. [40], who proposed a natural programming environment in which robot skills are accessed verbally to interact with end-users. The environment uses a dialog system to extract actions and conditions to create a sequence function chart. The challenge is still that the end-user cannot add new dialogue constructs for new tasks, making the languages inflexible.

6.2 Core Language Aspects

Actions We found that actions are often very concrete and every action has its own language concept. On the one hand, this is unavoidable. On the other hand, it would be beneficial to find a way to categorize or organize the various possible actions in groups in order to facilitate their definition, management, and treatment.

Furthermore, the language concepts for actions found in the environments are relatively basic. Consequently, they cannot sufficiently express what end-user might need. It is important that more vibrant libraries of controllers to specify behaviors with well-defined semantics are built to facilitate real-life mission specification for end-users.

Abstraction In general, the languages we surveyed have a rather low-level of abstraction. In most cases, the user is required to model in detail the behavior that the robots should perform to achieve the mission. This has some disadvantages:

- it is error-prone, and the user should know details about the language concepts used, which are not standard and in most cases biased to the robotics domain;
- 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
- it requires knowledge and expertise that the potential end-users will not necessarily have.

Goal-based and declarative mission specification languages look more promising and attractive.

Composition Mechanisms and Strategies. An important aspect to consider when scaling, maintaining, and evolving mission specifications is mission composition—the strategies and mechanisms to compose complex

missions from lower-level behavior (e.g., tasks). As discussed above, the majority of our environments offers relatively simple composition mechanisms known from imperative programming. Functions are the prime units of composition, brought into an execution order (e.g., sequence or loop) using plain imperative programming statements. The majority of our languages is imperative. We did not observe any more sophisticated composition mechanisms known from general-purpose languages, such as object-oriented or functional programming concepts.

Composition strategies in robotics typically classify into horizontal and vertical composition [69,60,93]. 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.

In almost all languages, given their modularity concepts, both kinds of composition are possible. However, the different kinds of decomposition are less obvious and enforced. In the imperative languages that are tailored versions of general-purpose languages, which is the majority of our environments, developers use control-flow statements for horizontal decomposition and functions for vertical decomposition. However, functions do not necessarily represent vertical decomposition—when they are just used for reusing code or making the mission more comprehensible by refactoring out code into functions. As such, the kind of decomposition is not immediately obvious. An interesting environment is PROMISE with its behavior-tree-like language [39,21], where horizontal and vertical decomposition is more explicit and encouraged by the language. Sibling tasks in the tree represent horizontal decomposition of these tasks. With respect to their parent, they are their vertical decomposition. Finally, Aseba's event-based language only supports horizontal decomposition, as seen by the event-action pairing *detect object – set top color red* in Fig. 12)

Intelligence. A core aspect to build into languages in the robotics domain is intelligence—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. Our feature model presents concepts such as event support in Fig. 11, delayed action type in Fig. 14 and reading sensor data in Fig. 15, which can facilitate intelligence in the robot systems. Specifically, some environments offer complex facilities, such as face recognition, in function libraries, as discussed in Sec. 5.3 (feature ReadSensor).

An important aspect is the programming model, which facilitates expressing the necessary intelligence. Pure imperative programming is the default, and while behavior trees, state machines, and the reactive control concepts in one of the languages (Aseba) are not more expressive, they provide a more abstract and restricted form of mission specification, forcing the developer to focus more on expressing the intelligence in an intuitive and comprehensible way.

Collaborative Multi-Robots. The large majority of our environments and their languages support one robot. Increasingly, the multiple robots need to collaborate to achieve complex missions. Environments such as FLYAQ, PROMISE, and MissionLab support collaborative mission specification. FLYAQ [12] facilitates the specification of missions for multiple drones. The end-user explicitly sequences the tasks for each robot, together with location details, thereby avoiding collisions. Some of the mission primitives used include: *Takeoff, Goto(location), DoPhoto, Land*. Since the distribution of tasks to drones is done manually, there are no language concepts to express multi-robot mission specifications as shown in Fig. 7. PROMISE [38] proposes a visual mission specification environment for multi-robots, however, the decomposition of the mission to local missions for each robot is also done manually. The operator parallel (*parallelOp*) takes robots as input and assigns a robot to each branch (each child). Beyond that, there is no more dedicated support for multi-robot missions, such as scheduling support or robot pooling for complex missions. This indicates that multi-robot mission specification and task distribution is not trivial, certainly not from an algorithm perspective, but also not from a language perspective. An interesting work in this direction is probably Doherty et al [26], who 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.

7 Practical Usage of the Survey

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

7.1 End-User—Teacher

A teacher has to instruct a robot development course to students with limited background in programming

languages. The teacher has to select a robotic mission specification environment based on the requirements:

1. *Simulation support:* The environment shall support simulating the mission execution and deploying the mission on the physical robots.
2. *Language control flows:* The language shall support specifying sequences of tasks repeated until a certain condition holds (loop statements) and executing alternative tasks depending on some conditions (conditional statements).
3. *Actions*: The language shall allow users to specify movement actions and robot to human communication.
4. *Runtime environment:* The mission specification environment shall run 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:

1. *Simulation support:* Table 5 shows that Aseba, FLYAQ, Makecode, Metabot, PICAXE, Robot Mesh Studio, MissionLab, Open Roberta, RobotC, and TRIK Studio provide simulation support and can be used within the course.
2. *Language control flows:* Table 6 shows that 26 of our 30 environments provide loop and conditional statements, and can be used within the course.
3. *Actions:* Since all environments offer movement actions, checking Fig. 14, selection of appropriate environment will depend on the environments that offer communication with humans. From Table 6, the following environments support robot to human communication: Choregraphe, Enchanting, SparkiDuino, MissionLab, Code Lab, Open Roberta, and TiViPE.
4. *Runtime environment:* Table 2 shows that Open Roberta, FLYAQ, Robot Mesh Studio and Sphero provide both the web interface and can be executed as stand-alone applications, and therefore can be used in the course.

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

7.2 End-User—Robot Manufacturers

Robots are usually ensembles of existing parts. It is handy for robotics engineers to check mission specification environments for the features that the robot should have. Let us consider a robot manufacturer interested in creating a new robot that can move on land and recognize objects as well as sound. The purpose of the robot is to aid in learning programming and research.

The robot manufacturer needs to know:

1. *Robot mobility, e.g., motor, steering:* What movement features exist in mobile robots for benchmarking;
2. *Actions, e.g., movement actions, instantaneous actions, relative movement actions, and manipulation actions:* Which robot actions a new robot can execute.
3. *Simulation support:* Which robots have a simulator integrated with the environment?

The robot manufacturer can be guided as follows:

1. *Mobile robots:* Table 2 contains a list of mobile robots, such as VEX robots, PICAXE, and LEGO robots. By further profiling the manufacturer specifications of such robots, a robot manufacturer can take informed decisions on what mobility features she can incorporate in the new robot.
2. *Actions:* Figure 14 and Table 6 can guide the manufacturer to analyze a variety of actions, which the robot can execute. Actions such as movement, instantaneous actions, relative actions, delayed actions and actuations can be performed by VEX robots.
3. *Simulation support:* For instance, Aseba's Thymio robot can be simulated, as well as nine other environments providing a simulator, as shown in Table 5.

The robot manufacturer can use VEX robotics. Specifically, VEX robots are open and can be programmed by many environments, such as VEX Coding Studio, EasyC, Robot Mesh Studio, and RobotC.

7.3 End-User—Language Engineer

A language engineer wants to develop a language for mission specification targeting children below seven years. The engineer 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 engineer has the following requirements:

1. *Notation:* The environment shall provide a visual language based on blocks and connections, since users can barely read and write.
2. *Simulation support:* The environment shall provide simulation support to allow children to play and simulate the behavior of the robots when executing different missions.
3. *Language control flows:* Choice of control flow from available options such as loops, conditional, and interrupts is also required.
4. *Actions:* Children shall be able to specify complex movement actions, such as making the robot dance. Furthermore, children should be allowed to communicate with the robots.

5. *Runtime environment:* The environment shall be executable as a stand-alone application.

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

1. *Notation:* Section 4 lists each environment's syntax(es). A block-based syntax, such as in SparkiDuino, Ardublockly, Aseba, BlocklyProp, Edison software, or LEGO Mindstorms EV3, fits the requirement (Table 2 lists all with a block-based syntax).
2. *Simulation support:* Based on Table 5, Aseba, FLYAQ, Makecode, Metabot, PICAXE, Robot Mesh Studio, MissionLab, Open Roberta, RobotC, PROMISE, and TRIK Studio provide simulation support.
3. *Language control flows:* Table 5 guide on the available control flow concepts offered by the languages. All environments offer loop control flows except for FLYAQ, and MissionLab (where loops can still be emulated with self-references). Almost all (29) environments also offer conditional control flows, such as *if*, *if-else*, and switch, while 20 of the environments offer interrupt controls. TRIK Studio provides multithreading fork control-flow support.
4. *Actions:* The language concepts summarized in Table 6can 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 SparkiDuino, BlocklyProp, Edison software, FLYAQ, LEGO Mindstorms EV3, Makeblock 5, Sphero, and Tello Edu App.
5. *Runtime environment:* Using Table 2, the engineer can determine features of stand-alone environments. Most of them offer support for stand-alone installations, except Edison software, Makecode, Marty software, Metabot, Ozoblockly, and Scratch EV3.

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 might not necessarily be the same. We resolved this issue by relying on multiple sources of data as well as searching for robots and then trying 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 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 BlocklyProp.

External Validity. Extracting the features using independent data collection based on documentation available in the public domain is a threat 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.

Furthermore, there is diversity in phrases and terms used to describe mission specifications. 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 explained in Sec. 3.1.

9 Related Work

Bravo et al. [14] review intuitive robot programming environments for education. They categorize their languages into textual, visual, and tangible languages. However, they do not discuss individual language features that facilitate end-user programming, as we do.

Biggs et al. [10] 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. [91] survey user expectancies from robots. They find that, at a 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 and closer to the end-user domains.

Abdelfetah Hentout et al. [42] survey development environments for robotics. They identify frameworks for programming robotic systems, but not targeting mission specifications.

Jost et al. [48] in their review of 10 visual programming environments for educational robots, discuss advantages of visual over textual environments, in order to present the Open Roberta project. They do not analyze any of the existing environments to the extent we do, however.

Luckcuck et al.'s survey [65] 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 [84, 83] 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 motivated our study.

Sun et al. [104] use models to raise the level of abstraction of implementation details to support developers in solving challenges (e.g., maintenance), supporting multiple platforms, and validating timing requirements. However, the level of abstraction is not to the graphical level where novice developers can easily comprehend the implementation details.

Ghzouli et al.'s [39] work analyzes behavior tree language concepts, such as root node, composite nodes (sequence, selector, decorator and parallel nodes), and leaf nodes, which are well suited for robotics especially the specification of robot missions. The authors also study the use of these language concepts in open-source robotic applications, where the robot behavior is represented in behavior tree models.

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 to hard-code their missions at manufacturing time. It is also unrealistic to keep relying on robotic and software engineers to program these missions. With the increasing presence of robots in our everyday life, more research and development effort has focused on enabling end-users to specify robotic missions. Recognizing that visual environments are more motivating for end-users as they reduce the burden of memorizing intricate syntax in textual languages (e.g., C++ and Java) [72], many end-user-oriented mission specification environments have been presented. However, to the best of our knowledge, there was no study identifying and organizing the features provided by such environments and languages.

In our survey, we studied the design space of 30 specification environments providing dedicated end-user-oriented languages for mission specification. We presented the design space as a feature model and further analyzed how the environments provide these features and how they differ from each other.

In summary, we found many typical constructs (e.g., control-flow statements) from general-purpose languages, provided using visual syntax. Many environments appear to have taken a general-purpose language and stripped it down to the needs of mission specification, implementing the syntax for the remaining language concepts often using Blockly or Scratch. 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 provide 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.

Most languages use control-flow statements for horizontal decomposition and functions for vertical decomposition. The environments provide computation algorithms over sensor data to intuitively realize intelligence in the robots. While collaborative multi-robot systems do not offer automated task scheduling among robots, end-users explicitly assign tasks for each robot in the team.

Even though, complex and powerful algorithms are hidden behind single language concepts, we found the abstraction level of the languages in general relatively low, especially with respect to specifying the mission, which coordinates the skills and actions of the robot. Goal-based and declarative mission specification languages look more promising and attractive.

In summary, the language engineering community and researchers, should use the findings as a benchmark and apply tools such as language workbenches and domain modeling to develop better DSLs for robot mission specifications.

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. Ideally, future languages can be customizable to the individual users' needs, establishing language product lines [110, 109] for robotics mission specification.

A possible route is to assess the syntax with respect to Moody's notational design principles [79]. 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.

Acknowledgments

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

References

1. Ali, K.S., Balch, T.R., Cameron, J.M., Chen, Z., Endo, Y., Halliburton, W.C., Kaess, M., Kira, Z., Lee, J.B., MacKenzie, D.C., Martinson, E.B., Merrill, E.P., Ranganathan, A., Sgorbissa, A., Stoytchev, A., Ulam, P., Wagner, A.: User manual for MissionLab version 7.0. Tech. rep., Georgia Tech Mobile Robot Laboratory. URL `https://www.cc.gatech.edu/aimosaic/robot-lab/research/MissionLab/mlab_manual-7.0.pdf`
2. Anki: Codelab for cozmo robot (2020). URL `http://www.anki.com/en-us/cozmo/code-lab/how-it-works`
3. Aragão, M., Moreno, P., Bernardino, A.: Middleware interoperability for robotics: A ROS–YARP framework. Frontiers Robotics AI **3**, 64 (2016). DOI 10.3389/frobt.2016.00064. URL `https://doi.org/10.3389/frobt.2016.00064`
4. Arcbotics: Arcbotics, learning with robots (2020). URL `http://arcbotics.com/lessons/sparki/`
5. Ardublockly: Ardublockly (2020). URL `https://ardublockly.embeddedlog.com/`
6. Arkin, R.: Missionlab: Multiagent robotics meets visual programming. Working notes of Tutorial on Mobile Robot Programming Paradigms, ICRA **15** (2002)
7. Bacca-Cortés, B., Florián-Gaviria, B., García, S., Rueda, S.: Development of a platform for teaching basic programming using mobile robots. Revista Facultad de Ingeniería **26**(45), 61–70 (2017)
8. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature? a qualitative study of features in industrial software product lines. In: International Software Product Line Conference (SPLC) (2015)
9. Berger, T., Völter, M., Jensen, H.P., Dangprasert, T., Siegmund, J.: Efficiency of projectional editing: A controlled experiment. In: International Symposium on the Foundations of Software Engineering (FSE). ACM (2016)
10. Biggs, G., Macdonald, B.: A survey of robot programming systems. In: Proceedings of the Australasian conference on robotics and automation, p. 27 (2003)
11. Bozhinoski, D., Bucchiarone, A., Malavolta, I., Marconi, A., Pelliccione, P.: Leveraging Collective Run-Time Adaptation for UAV-Based Systems. Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE (2016)
12. Bozhinoski, D., Di Ruscio, D., Malavolta, I., Pelliccione, P., Tivoli, M.: Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters. International Conference on Automated Software Engineering (ASE). IEEE/ACM (2015)
13. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
14. Bravo, F.A., Gonzalez, A.M., Gonzalez, E.: A review of intuitive robot programming environments for educational purposes. In: Conference on Automatic Control (CCAC). IEEE (2018)
15. Brugali, D., Prassler, E.: Software engineering for robotics. IEEE Robotics and Automation Magazine **16**(1), 9–15 (2009). DOI 10.1109/MRA.2009.932127
16. Burnett, W.: Alternative programming languages for lego mindstorms (2020). URL `http://www.legoengineering.com/alternative-programming-languages/`
17. Burnett, W.: Blockly (2020). URL `https://developers.google.com/blockly/`
18. Button, R.W., Kamp, J., Curtin, T.B., Dryden, J.: A survey of missions for unmanned undersea vehicles. Tech. rep. (2009)
19. Campusano, M., Fabry, J.: Live robot programming: The language, its implementation, and robot api independence. Science of Computer Programming **133**, 1–19 (2017)
20. Caron, D.: Competitive robotics brings out the best in students. Tech Directions **69**(6), 21–24 (2010)
21. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction (2018). DOI 10.1201/9780429489105
22. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI: An Introduction. CRC Press (2018)
23. Combemale, B., France, R., Jézéquel, J.M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering modeling languages: Turning domain knowledge into tools. CRC Press (2016)
24. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**, 621–645 (2006)

25. Di Ruscio, D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. CEUR Workshop Proceedings **1319**, 16–29 (2014)
26. Doherty, P., Heintz, F., Kvarnström, J.: High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. Unmanned Systems **01**(01), 75–119 (2013)
27. Dragule, S., Garcia, S., Berger, T., Pelliccione, P.: Languages for specifying missions of robotic applications. In: Software Engineering for Robotics. Springer (2020)
28. EasyC: Easyc v5 for cortex and vex iq (2020). URL `http://www.intelitek.com/engineering/easyc/`
29. Enchatnting: http://enchanting.robotclub.ab.ca/tiki-index.php (2020)
30. Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al.: The State of the Art in Language Workbenches. In: Software Language Engineering. Springer (2013)
31. EV3, L.M.: Apps (2020). URL `https://www.lego.com/en-us/mindstorms/downloads/download-software`
32. Fernández-Perdomo, E., Cabrera-Gómez, J., Domínguez-Brito, A.C., Hernández-Sosa, D.: Mission specification in underwater robotics. Journal of Physical Agents **4**(1), 25–34 (2010)
33. FLYAQ: Graphical specification and execution of missions (2020). URL `http://www.flyaq.it/`
34. Fojtik, R.: The ozobot and education of programming. New Trends and Issues Proceedings on Humanities and Social Sciences **4**(5) (2017). DOI 10.18844/prosoc.v4i5.2666
35. García, S., Pelliccione, P., Menghi, C., Berger, T., Bures, T.: High-level mission specification for multiple robots. In: International Conference on Software Language Engineering. ACM (2019)
36. Garcia, S., Pelliccione, P., Menghi, C., Berger, T., Bures, T.: Promise: High-level mission specification for multiple robots. In: International Conference on Software Engineering (ICSE), Demonstrations Track (2020)
37. Garcia, S., Strueber, D., Brugali, D., Berger, T., Pelliccione, P.: Robotics software engineering: A perspective from the service robotics domain. In: International Symposium on the Foundations of Software Engineering (FSE). ACM (2020)
38. García-Zubía, J., Angulo, I., Martínez-Pieper, G., Orduña, P., Rodríguez-Gil, L., Hernandez-Jayo, U.: Learning to program in k12 using a remote controlled robot: Roboblock. In: Online Engineering & Internet of Things, pp. 344–358. Springer (2018). DOI 10.1007/978-3-319-64352-6_33
39. Ghzouli, R., Berger, T., Johnsen, E.B., Dragule, S., Wasowski, A.: Behavior trees in action: A study of robotics applications. In: International Conference on Software Language Engineering (SLE). ACM (2020)
40. Gorostiza, J.F., Salichs, M.A.: End-user programming of a social robot by dialog. Robotics and Autonomous Systems **59**(12), 1102–1114 (2011). DOI 10.1016/j.robot.2011.07.009
41. Guide, A., Guide, B., Foundation, F.: Program tello drone to do back ips with scratch! (2020). URL `https://www.aerial-guide.com/article/program-tello-drone-to-do-backflips-with-scratch`
42. Hentout, A., Maoudj, A., Bouzouia, B.: A survey of development frameworks for robotics. In: International Conference on Modelling, Identification and Control (ICMIC). IEEE (2016)
43. Hocraffer, A., Nam, C.S.: A meta-analysis of human-system interfaces in unmanned aerial vehicle (uav) swarm management. Applied ergonomics **58**, 66–80 (2017)
44. Holwerda, R., Hermans, F.: A usability analysis of blocks-based programming editors using cognitive dimensions. In: Symposium on visual languages and human-centric computing (VL/HCC), pp. 217–225. IEEE (2018)
45. IAIS, F.: Open roberta lab (2020). URL `https://lab.open-roberta.org/`
46. Ioannou, M., Bratitsis, T.: Teaching the notion of speed in kindergarten using the sphero sprk robot. In: International Conference on Advanced Learning Technologies. IEEE (2017)
47. Jarvinen, E.M., Karsikas, A., Hintikka, J.: Children as innovators in action–a study of microcontrollers in finnish comprehensive schools. Journal of Technology Education **18**, 37–52 (2007)
48. Jost, B., Ketterl, M., Budde, R., Leimbach, T.: Graphical programming environments for educational robots: Open roberta - yet another one? In: International Symposium on Multimedia. IEEE (2014)
49. Junior, L.A., Neto, O.T., Hernandez, M.F., Martins, P.S., Roger, L.L., Guerra, F.A.: A low-cost and simple arduino-based educational robotics kit. Journal of Selected Areas in Robotics and Control (JSRC) **3**(12), 12 (2013)
50. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. software engineering institute. Universitas Carnegie Mellon, Pittsburgh, Pennsylvania (1990)
51. Kaučič, B., Asič, T.: Improving introductory programming with scratch? In: Proceedings of the International Convention MIPRO. IEEE (2011)
52. Ketterl, M., Jost, B., Leimbach, T., Budde, R.: Open roberta – a web based approach to visually program real educational robots. International Journal of Learning and Media **8** (2016). DOI 10.7146/lom.v8i14.22183
53. Kolling, A., Walker, P., Chakraborty, N., Sycara, K., Lewis, M.: Human interaction with robot swarms: A survey. IEEE Transactions on Human-Machine Systems **46**(1), 9–26 (2016)
54. Korkmaz, Ö.: 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 **4**(3), 73–88 (2016)
55. Krishnan, A.: Turtlebot3blockly documentation (2020). URL `https://readthedocs.org/projects/turtlebot-3-blockly-wiki/downloads/pdf/latest/`
56. Lämmel, R.: Software languages: Syntax, semantics, and metaprogramming. Springer (2018)
57. Lane, A., Meyer, B., Mullins, J.: Robotics with Enchanting and LEGO NXT-A Project Based Introduction to Programming, version 1.1 edn. © 2012 Monash University under a Creative Commons (2012)
58. Linsbauer, L., Berger, T., Grünbacher, P.: A classification of variation control systems. In: International Conference on Generative Programming: Concepts & Experience (GPCE). ACM (2017)
59. Linsbauer, L., Schwaegerl, F., Berger, T., Gruenbacher, P.: Concepts of variation control systems. Journal of Systems and Software (2020). Preprint
60. Lotz, A.: Managing non-functional communication aspects in the entire life-cycle of a component-based robotic software system. Dissertation, Technische Universität München, München (2018)
61. Lourens, T.: Tivipe - tino's visual programming environment. In: International Computer Software and Applications Conference (COMPSAC). IEEE (2004)
62. Lourens, T.: Programming robots using tivipe - a step by step approach using aldebaran's nao robots. Tech. rep., TiViPE, Netherlands (2011)

63. Lourens, T., Barakova, E.: User-friendly robot environment for creation of social scenarios. In: International Work-Conference on the Interplay Between Natural and Artificial Computation, (IWINAC). Springer (2011)
64. Lozenko, G.F., , Vadim Olegovich Dzhenzher, L.V.D.: Training future teachers in computer skills in extra-curricular activity with schoolchildren. Life Science Journal **11**(8s), 203 (2014)
65. Luckcuck, M., Farrell, M., Dennis, L., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. Computing Surveys **52**(5) (2019). DOI 10.1145/3342355
66. MacKenzie, D.C., Arkin, R.C., Cameron, J.M.: Multiagent mission specification and execution. Autonomous Robots **4**(1), 29–52 (1997). DOI 10.1023/A:1008807102993
67. Magnenat, S., Rétornaz, P., Bonani, M., Longchamp, V., Mondada, F.: Aseba: A modular architecture for event-based control of complex robots. Transactions on Mechatronics **16**(2), 321–329 (2011). DOI 10.1109/TMECH.2010.2042722
68. Makeblock: Makeblock (2020). URL `https://www.makeblock.com/software`
69. Medeiros, A.A.: A survey of control architectures for autonomous mobile robots. Journal of the Brazilian Computer Society **4**(3) (1998)
70. Menghi, C., Tsigkanos, C., Berger, T., Pelliccione, P.: PsALM: Specification of dependable robotic missions. In: International Conference on Software Engineering (ICSE), Demonstrations Track. IEEE / ACM (2019)
71. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. Transactions on Software Engineering pp. 1–1 (2019). DOI 10.1109/TSE.2019.2945329
72. Menzies, T.: Evaluation issues for visual programming languages. In: Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies, pp. 93–101. World Scientific (2002)
73. Mesh, R.: Robot mesh studio (2020). URL `http://docs.robotmesh.com/ide-project-page`
74. Metabot: Metabot blockly (2020). URL `http://blocks.metabot.fr`
75. Microbric: Edison software (2020). URL `http://meetedison.com/robot-programming-software/`
76. Microsoft: mindstorms (2020). URL `http://makecode.mindstorms.com`
77. Miskam, M.A., Shamsuddin, S., Yussof, H., Omar, A.R., Muda, M.Z.: Programming platform for nao robot in cognitive interaction applications. In: International Symposium on Robotics and Manufacturing Automation (ROMA). IEEE (2014)
78. Mohamed, N., Al-Jaroodi, J., Jawhar, I.: Middleware for robotics: A survey. In: International Conference on Robotics, Automation and Mechatronics, RAM, pp. 736–742. IEEE (2008)
79. Moody, D.: The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. Transactions on Software Engineering **35**(6), 756–779 (2009). DOI 10.1109/TSE.2009.67
80. Mordvinov, D., Litvinov, Y., Bryksin, T.: Trik studio: Technical introduction. In: Conference of Open Innovations Association (FRUCT). IEEE (2017)
81. Multiplo: Minibloq (2020). URL `http://blog.minibloq.org/`
82. Nesic, D., Krueger, J., Stanciulescu, S., Berger, T.: Principles of feature modeling. In: International Symposium on the Foundations of Software Engineering (FSE). ACM (2019)
83. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. Journal of Software Engineering for Robotics **7**(1), 75–99 (2016)
84. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). Springer (2014)
85. Ozobot: Ozoblockly (2020). URL `http://ozoblockly.com/editor?lang=en&robot=bit&mode=2`
86. Parallax: Getting started with blocklyprop (2019). URL `https://learn.parallax.com/tutorials/language/blocklyprop/getting-started-blocklyprop`
87. Passault, G., Rouxel, Q., Petit, F., Ly, O.: Metabot: A low-cost legged robotics platform for education. In: International Conference on Autonomous Robot Systems and Competitions (ICARSC). IEEE (2016)
88. Pasternak, E., Fenichel, R., Marshall, A.N.: Tips for creating a block language with blockly. In: Blocks and Beyond Workshop, B and B. IEEE (2017). DOI 10.1109/BLOCKS.2017.8120404
89. PICAXE: Picaxe (2020). URL `http://www.picaxe.com/software`
90. Pot, E., Monceaux, J., Gelin, R., Maisonnier, B.: Choregraphe: a graphical tool for humanoid robot programming. In: International Symposium on Robot and Human Interactive Communication. IEEE (2009)
91. Ray, C., Mondada, F., Siegwart, R.: What do people expect from robots? In: International Conference on Intelligent Robots and Systems. IEEE (2008)
92. Robin, C., Lacroix, S.: Multi-robot target detection and tracking: taxonomy and survey. Autonomous Robots **40**, 729–760 (2016)
93. Robmosys: Task-level composition for robotic behavior. `https://robmosys.eu/wiki/composition:task:start` (2019)
94. Robotc: Robotc's graphical feature (2020). URL `http://www.robotc.net/graphical/`
95. Robotical: Marty application (2020). URL `http://martytherobot.com/users/using-marty/program/scratch/getting-started-with-scratch/`
96. Robotics, S.: Documentation (2020). URL `http://doc.aldebaran.com/1-14/software/choregraphe/interface.html`
97. Robotics, V.: Vex robotics (2020). URL `http://www.vexrobotics.com`
98. Ruscio, D.D., Malavolta, I., Pelliccione, P., Tivoli, M.: Automatic generation of detailed flight plans from high-level mission descriptions. In: International Conference on Model Driven Engineering Languages and Systems. ACM (2016)
99. Salcedo, S.L., Idrobo, A.M.O.: New tools and methodologies for programming languages learning using the scribbler robot and alice. IEEE (2011)
100. Schauss, S., Lämmel, R., Härtel, J., Heinz, M., Klein, K., Härtel, L., Berger, T.: A chrestomathy of dsl implementations. In: International Conference on Software Language Engineering (SLE). ACM (2017)
101. Scratch: http://scratch.mit.edu/projects/editor/?tutorial=ev3 (2020)
102. Shin, J., Siegwart, R., Magnenat, S.: Visual programming language for thymio ii robot. In: Conference on Interaction Design and Children (IDC'14). ETH Zürich (2014)
103. Sphero: Sphero app (2020). URL `http://www.sphero.com/education/`
104. Sun, Y., Gray, J., Bulheller, K., von Baillou, N.: A model-driven approach to support engineering changes in industrial robotics software. In: International Conference on Model Driven Engineering Languages and Systems. Springer

105. Thymio: The aseba language (2020). URL http://www.thymio.org/en:start
106. Trikset: Trik studio (2020). URL http://www.trikset.com/products/trik-studio
107. TurtleBot3Blockly: Turtlebot3 blockly (2020). URL http://turtlebot-3-blockly-wiki.readthedocs.io/en/latest/launchBlockly.html
108. Ulam, P., Endo, Y., Wagner, A., Arkin, R.: Integrated mission specification and task allocation for robot teams - design and implementation. In: International Conference on Robotics and Automation, pp. 4428–4435. IEEE (2007)
109. Vacchi, E., Cazzola, W.: Neverlang: A framework for feature-oriented language development. Computer Languages, Systems & Structures **43**, 1–40 (2015)
110. Vacchi, E., Cazzola, W., Pillay, S., Combemale, B.: Variability support in domain-specific language development. In: International Conference on Software Language Engineering (SLE). Springer (2013)
111. Vandevelde, C., Wyffels, F., Ciocci, M.C., Vanderborght, B., Saldien, J.: Design and evaluation of a diy construction system for educational robot kits. International Journal of Technology and Design Education **26**(4), 521–540 (2016). DOI 10.1007/s10798-015-9324-1
112. Völter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: International Conference on Software Language Engineering (SLE). Springer (2014)
113. Weintrop, D., Afzal, A., Salac, J., Francis, P., Li, B., Shepherd, D.C., Franklin, D.: Evaluating coblox: A comparative study of robotics programming environments for adult novices. Conference on Human Factors in Computing Systems (CHI). ACM (2018)
114. Wiedu: Tello eduapp (2020). URL http://www.wiedu.com/telloedu/index_en.html

A Environment-Feature Matrix

Table 5 shows an overview of features in the specification environments. Table 6 reports the features related to language concepts.

	Ardublockly	Aseba	BlocklyProp	Choregraphe	CodeLab	EasyC	EdisonSoftware	Enchanting	FLYAQ	LegoMindstormsEV3	Makeblock5	Makecode	MartySoftware	Metabot	Ozoblockly	PICAXE	RobotMeshStudio	ScratchEv3	SparkiDuino	Sphero	TelloEduApp	TiViPE	Turtlebot3Blockly	VexCodingStudio	MiniBloq	MissionLab	OpenRoberta	RobotC	TrikStudio	PROMISE	Frequency
Specification environments																															
MultiLanguageSupport	x	x	x	x	x	x	✓	x	x	x	✓	x	x	x	x	✓	✓	x	✓	✓	x	x	x	✓	x	x	x	✓	x	x	8
Editor Modes																															
Projectional	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	30
Parser-based	x	✓	x	x	x	✓	✓	x	x	x	✓	✓	x	x	x	✓	✓	x	✓	✓	x	x	x	✓	x	x	x	✓	x	✓	12
Semantic services																															
Error Marking	✓	✓	✓	x	x	x	✓	x	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	✓	x	x	x	x	x	✓	7
Quick fixes	x	x	x	✓	x	x	✓	x	x	x	x	x	x	x	x	✓	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	5
Reference resolution	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	x	x	✓	2
Live translation	x	✓	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	2
Syntactic services																															
Visual highlighting	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	x	✓	✓	✓	✓	26
Syntactic completion	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x	x	x	✓	x	✓	x	x	✓	✓	7
Auto formatting	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	✓	x	x	x	x	✓	x	x	x	x	x	x	5
Simulation																															
Single Robot	x	✓	x	x	x	x	x	x	✓	x	x	✓	x	✓	x	✓	✓	x	x	x	x	x	x	x	x	✓	✓	✓	✓	x	10
Multi-robot	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	2
Runtime specification[1]	x	x	x	✓	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	✓	x	x	✓	x	x	x	x	5
Debugging	x	✓	x	✓	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x	x	x	✓	x	✓	✓	✓	✓	x	9
Mission deployment[2]																															
Runtime redeployment	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	✓	x	x	x	x	x	x	x	2
Over the air	x	x	x	✓	✓	x	x	x	x	✓	x	x	x	✓	✓	x	x	✓	x	x	✓	x	✓	x	x	x	x	x	✓	✓	10
Via cable	✓	✓	✓	✓	x	✓	✓	✓	x	✓	✓	✓	✓	✓	x	✓	✓	x	✓	x	✓	✓	✓	✓	✓	x	✓	✓	✓	x	23
Language characteristics																															
Language concepts[3]																															
Notation																															
Block-based	✓	✓	✓	x	✓	x	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	x	✓	✓	x	✓	24
Flowchart-based	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	✓	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	3
Graph-based	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	x	✓	x	x	✓	x	4
Text-based	x	✓	x	x	✓	x	✓	x	x	✓	✓	x	✓	x	x	✓	✓	x	✓	✓	x	x	x	✓	x	x	x	✓	✓	✓	14
Custom map-based	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1
Semantics																															
Compiled	✓	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	28
Interpreted	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	2
Language Paradigm																															
DSL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	30
GPL support	x	x	x	x	x	x	✓	x	x	x	✓	x	✓	x	x	x	✓	x	✓	✓	x	x	x	✓	x	x	x	✓	✓	x	9
Extensibility																															
Scripting support	✓	✓	✓	✓	x	✓	✓	x	x	x	x	✓	x	x	x	x	x	x	✓	x	x	x	✓	x	✓	x	✓	✓	x	x	12
Add language concept	✓	✓	x	x	x	✓	x	✓	✓	✓	✓	x	x	x	✓	x	x	✓	x	x	✓	✓	x	x	x	✓	✓	✓	✓	✓	16

Table 5 Feature matrix for specification environment features and general language characteristics

[1] All environments support design time specification [2] No environment supports runtime interference. [3] Language concepts in Table 6

	Ardublockly	Aseba	BlocklyProp	Choregraphe	CodeLab	EasyC	EdisonSoftware	Enchanting	FLYAQ	LegoMindstormsEV3	Makeblock5	Makecode	MartySoftware	Metabot	Ozoblockly	PICAXE	RobotMeshStudio	ScratchEv3	SparkiDuino	Sphero	TelloEduApp	TiViPE	Turtlebot3Blockly	VexCodingStudio	MiniBloq	MissionLab	OpenRoberta	RobotC	TrikStudio	PROMISE	Frequency
General concepts																															
Control flow																															
Conditionals	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	29
Loops	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	28
Interrupts	x	x	✓	✓	x	✓	✓	x	x	✓	✓	✓	✓	✓	✓	x	✓	✓	x	✓	x	✓	x	✓	x	✓	✓	✓	x	✓	20
Multithreading forks	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	✓	x	2
Modularity-modules[1]	✓	x	✓	✓	x	x	x	x	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	✓	x	✓	✓	✓	x	✓	✓	x	✓	x	17
Variable Data types																															
Primitive	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓	✓	✓	x	✓	✓	✓	x	25
Compound	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	x	✓	✓	✓	x	✓	✓	✓	x	25
Mission Specification paradigm																															
Reactive Control	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x	x	x	x	1
Imperative	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	29
Function Library																															
Arithmetic functions	✓	x	✓	✓	x	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	x	x	24
String operations	x	x	✓	x	x	x	x	x	x	x	✓	✓	✓	x	✓	✓	✓	✓	✓	x	x	x	x	x	x	✓	x	✓	x	x	10
Multithreading	✓	x	✓	✓	x	✓	✓	x	x	✓	✓	✓	✓	x	x	✓	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x	11
Multirobot Hardware Support	x	x	✓	✓	x	✓	x	x	x	x	✓	x	x	x	x	x	x	✓	x	✓	x	x	x	x	✓	✓	✓	x	✓	✓	11
File access																															
Read/write	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	✓	x	x	x	✓	x	x	✓	x	✓	✓	x	8
Open/close	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	x	x	x	✓	x	x	✓	x	x	x	✓	x	✓	x	x	6
ReadSensor[2]	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	x	27
Event support	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	x	x	✓	x	✓	✓	x	✓	✓	25
Exception Handling	x	x	x	✓	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	✓	5
Actions																															
Action type																															
Instantaneous	✓	x	✓	x	x	x	x	✓	✓	✓	x	✓	x	x	✓	x	x	✓	✓	x	x	✓	✓	x	x	✓	x	✓	x	✓	14
Continuous	x	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x	x	x	x	x	x	✓	x	✓	✓	x	x	x	✓	x	✓	✓	✓	14
Delayed	✓	✓	✓	✓	x	x	✓	✓	x	✓	✓	x	✓	✓	✓	✓	✓	x	x	✓	✓	x	✓	✓	✓	x	✓	x	x	x	19
Communication actions																															
With Human	x	x	x	✓	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x	✓	x	x	x	x	✓	✓	x	x	x	7
With Agent	x	x	✓	x	x	x	✓	x	✓	✓	✓	x	x	x	x	x	x	x	✓	✓	x	x	x	✓	x	✓	✓	x	✓	✓	12
Movement actions																															
Absolute	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	✓	✓	✓	x	x	✓	✓	✓	x	✓	x	✓	x	x	x	✓	11
Relative	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	x	28
Manipulator actions[3]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	30

Table 6 Feature matrix for language concepts

[1] Modules comprise functions and components [2] List of kinds of read sensor support in Table 15 [3] List of kinds of manipulation actions in Fig. 14

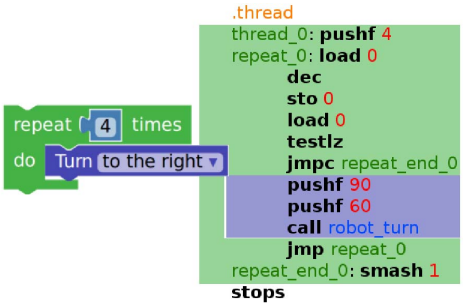

Fig. 17 Example of Metabot's visual notation (left), together with generated assembler code (right) from [87]

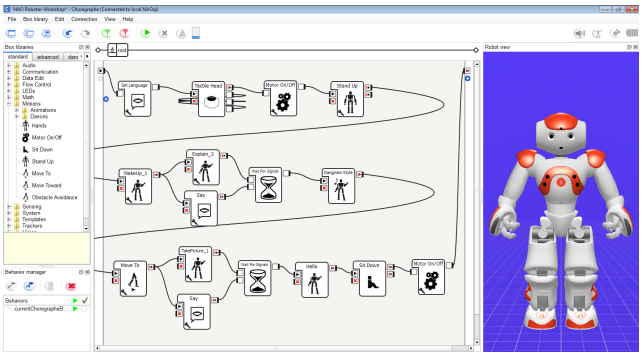

Fig. 16 The environment Choregraphe for the robot NAO

B Subject Environment Descriptions

This section provides a high-level textual description of the environments identified. In Table 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.

Ardublockly [5,44] is a Blockly-based environment supporting an educational, wheeled robot called Spartan [5,44,38], 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 [105,67] is a collection of environments with the same languages, but different syntaxes and, therefore, editors: VPL-based (Visual programming language) [102], 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.

BlocklyProp is a Blockly- and web-based environment for specifying missions for the wheeled educational robots ActivityBot robot and Scribbler robot [86].
Type of language: Block-based.

Choregraphe [96,90,77] 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. 16. 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 [2].
Type of language: Block-based and text-based.

EasyC [28] 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 [75,7] 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 and toy robot Lego Mindstorms NXT [29,64]—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 [31,16] 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. 13 in Sec. 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 [68,111]. 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 [76]. 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 [95] 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. 18. 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 [87,74]. Figure 17 shows a mission demonstrating the use of loop control structure with a corresponding assembler code generated [74,87].
Type of language: Block-based.

Ozoblockly [85,34] is a Blockly-based environment particularly for the educational, wheeled robot ozobot. The language and its visual syntax 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 [89,47] 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 [73] 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 [101] is the original Scratch from MIT, but tailored to support the educational robot kit Lego Mindstorms EV3. To

Fig. 18 The mission specification environment Marty software

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) [54].
Type of language: Block-based.

SparkiDuino [4] 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.

Sphero is an environment for programming the spherical educational robots Sphero BOLT, SPRK+, and Sphero Mini [103,46], 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 [114,41] 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.

TiViPE [61] 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 [63].
Type of language: Graph-based.

Turtlebot3-blockly [107,55] 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 [97,20] 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 [33,25,12] 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 [81,49] 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 [1,6,108] 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 [45,48,52] 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 [94,99] 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 [106,80] 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 [35,36] provides a graphical and a textual syntax for mission specification for multi-robot applications. The environment[5] 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 [21]—that can be composed to encode complex missions. These operators are interconnected following a behavior tree style and notation [39,21]. The language relies upon a catalog of patterns based on temporal logics, which encodes recurrent robotics missions from literature [71]. 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.

C Additional Online Resources

Table 7 provides links to online resources for the respective specification environment resources

D Author Biographies

Swaib Dragule is a PhD Fellow in Computer Science and Software Engineering at Chalmers University of Technology and Makerere University. He holds MSc. and BSc. in computer science. He is an academic staff of Makerere university, College of Computing and Information Sciences. His research interests are in programming languages, domain-specific languages, and robotics.

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

Table 7 Links to respective specification environment resources

Environment	URL Link
Ardublockly	https://ardublockly.embeddedlog.com/ , https://modernroboticsinc.com/product-category/spartan/
Aseba	https://www.thymio.org/en:star
BlocklyProp	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
SparkiDuino	http://arcbotics.com/lessons/sparki/
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	https://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

Thorsten Berger is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the PhD degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy—one of the highest recognitions for researchers in Sweden. He received two best-paper awards and one most influential paper award. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020. His research focuses on model-driven software engineering, program analysis, and empirical software engineering.

Claudio Menghi is a Research Associate at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), at the University of Luxembourg. After receiv-

ing his Ph.D. at Politecnico di Milano in 2015, he was a post-doctoral researcher at Chalmers University of Technology and University of Gothenburg. His research interests are in software engineering, with a special interest in cyber-physical systems (CPS) and formal verification.

Patrizio Pelliccione is an Associate Professor at University of L'Aquila and an Associate Professor at the Department of Computer Science and Engineering at Chalmers University of Technology and University of Gothenburg. He got his PhD in 2005 at the University of L'Aquila and since 2014 he is Docent in Software Engineering, title given by the University of Gothenburg. His research topics are in software architectures modeling and verification, autonomous systems, and formal methods. He has co-authored more than 120 publications in journals and international conferences and workshops. He has been on the program committees for several top conferences, is a reviewer for top journals, and has chaired the program committees of several international conferences. He is very active in European and national projects. In his research activity he has pursued extensive and wide collaboration with industry.