Languages for Specifying Missions of Robotic Applications

Swaib Dragule, Sergio García Gonzalo, Thorsten Berger, and Patrizio Pelliccione

Abstract Robot-application development is gaining increasing attention both from the research and industry communities. Robots are complex cyber-physical and safety-critical systems with various dimensions of heterogeneity and variability. They often integrate modules conceived by developers with different backgrounds. Programming robotic applications typically requires programming, mathematical or robotic expertise from end-users. In the near future, multipurpose robots will be used in tasks of everyday life in environments such as our houses, hotels, airports or museums. It would then be necessary to democratize the specification of missions that robots should accomplish. In other words, the specification of missions of robotic applications should be performed via easy-to-use and accessible ways and, at the same time, the specification should be accurate, unambiguous, and precise. This

Swaib 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

Sergio García Gonzalo
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
e-mail: sergio.garcia@gu.se

Thorsten Berger
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
e-mail: thorsten.berger@chalmers.se

Patrizio Pelliccione
Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila, L'Aquila, Italy
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
e-mail: patrizio.pelliccione@univaq.it

chapter presents Domain-Specific Languages (DSLs) for robot-mission specification, among others profiling them as internal or external, and also giving an overview of their tooling support. The types of robots supported by the respective languages and tools are mostly service mobile robots, including ground and flying types.

1 Introduction

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

Robots are complex cyber-physical and safety-critical systems, which challenges engineering their software [39, 16]. In addition, the robotics domain is divided into a large variety of sub-domains, including vertical ones (e.g., drivers, planning, navigation) and horizontal ones (e.g., defense, healthcare, logistics), with a vast amount of variability [40, 9], further complicating robotics software engineering. Due to this heterogeneity, a robot typically integrates modules conceived by developers with different backgrounds. For instance, electrical engineers design the robot's hardware, control engineers develop planning and control algorithms, and software engineers architect and quality-assure the software system. Coordinating the integration of all these modules from developers with different backgrounds is one of the major challenges that characterize the domain of robotics [16, 39]. Further challenges comprise identifying stable requirements, defining abstract models to cope with hardware and software heterogeneity, seamlessly transitioning from prototype testing and debugging to real systems, and deploying robotic applications in real-world environments.

A core activity when engineering robotics software is defining and implementing the robot's behavior. Specifically, in addition to building and integrating modules that define the lower-level behavior, the overall behavior of robots needs to be defined. This behavior, often called a *mission*, coordinates the lower-level behaviors that are typically defined in modules realizing the different skills. While this coordination has traditionally been implemented in plain code [48], this will not be feasible in the near future, when multipurpose robots will be used in our houses, hotels, hospitals, and so on, to accomplish tasks of the everyday life. For these reasons, the use of dedicated (domain-specific) languages is becoming increasingly popular [24, 33, 82]. These languages target end-users without robotic, ICT or mathematical expertise, and allow them to conveniently command and control robots. This trend is also expressed by the MAR roadmap, given the increasing involvement of robots in our society, especially service robots (i.e., robots that perform useful tasks for humans excluding industrial automation applications[2]). In fact, the MAR roadmap describes DSLs [93, 82, 55],

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

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

together with model-driven engineering [49, 86, 84, 18, 15], as core technologies required to achieve a separation of roles in the robotics domain while also improving, among others, modularity and system integration.

The specification of mission ranges from (i) very intricate and difficult-to-use [47, 5] logical languages, such as Linear-Temporal Logic or Computation Tree Logic [67, 31, 65, 43, 96], whose instances are directly fed into planners; via (ii) common notations for specifying behavior, such as Petri nets [94, 97] and state machines [13, 89, 56], which require low-level and step-by-step descriptions of missions; to (iii) robotics-specific DSLs tailored to the robot at hand [36, 83, 42, 26, 27, 70], which often allow a more high-level mission specification.

This chapter contributes to the state-of-the-art in mission specifications for robots. We present an overview of programming languages for robotic applications and respective IDEs (integrated development environments) in Sec. 2. Thereafter, we present DSLs for mission specification in Sec. 3, including internal and external DSLs, together with their tooling. In Sec. 4 we discuss how robots are usable in the everyday life, with specific reference to the PROMISE tool for specifying missions for multi-robot applications. We put PROMISE into practice by describing a real mission with PROMISE we realized, together with the rest of the robotic software, including a multi-layer architecture. We conclude and discuss areas for future work in Sec. 6.

2 Programming Languages and IDEs for Robotic Applications

The software of a robotic application can be conceptually organized into two main parts: (i) the software controlling the various modules (written once and embedded into the robot), and (ii) the software that permits the specification and execution of the mission (potentially changing from mission to mission, especially for multipurpose robots). Traditionally, these two parts are mixed for robots capable of doing specific tasks, where the mission specification only involves setting some parameters that are specific for the environment in which the mission will be executed. In this section, we briefly describe programming languages (Sec. 2.1) and IDEs (Sec. 2.2) used in robotics.

2.1 Programming Languages for Robotic Applications

Many different languages are used for the development of mobile robotic applications. Starting from the lowest level of abstraction, hardware-description languages (e.g., Verilog or VHDL) are mainly used by electronic engineers to “program” the low-level electronics of robots [71]. Hardware-description languages are commonly used to program field-programmable gate arrays [73], which are devices that make it possible to develop electronic hardware without having to produce a silicon chip.

At the level of microcontrollers, a widely used option is Arduino[3] [53]. It is an open-source electronics platform that consists of a board with assembled sensors (and potentially actuators) that can be controlled using specific software. Software for Arduino-based applications may be developed using an open-source IDE[4], which supports the languages C and C++, applying a wrapper around programs written in these languages and using special rules of code structuring. The hardware manufacturers typically also provide proprietary software, such as RAPID[5] technical reference manual from ABB and KRL[6] reference guide from Kuka.

More powerful machines in terms of computation—including single-board computer solutions such as Raspberry Pi—support Ubuntu distributions and, therefore, the Robot Operating System (ROS) [60]. ROS [76] is an open-source middleware offering a framework for structured communication among various robotic components using a peer-to-peer connection. ROS currently runs on Unix-based platforms, and software for ROS is primarily tested on Ubuntu. Therefore, a typical setup for a roboticist includes a certain version of Ubuntu[7] with a certain distribution of ROS.[8]

Most packages and libraries of ROS are developed using either C++ or Python so those languages are the most commonly used. However, ROS's communication system is language-agnostic, which enables several languages such as C++, Python, Octave, Java, and LISP to be used depending on the user's proficiency. ROS also offers modularized tool-based microkernel design to aggregate various tools performing specific tasks such as navigating source code tree, get and set configuration parameters, and visualize the peer-to-peer connection topology, among others [45, 59].
ROS has evolved with a number of distributions, supporting more than 20 robotic systems[9], including drones, arm robots, humanoids, and wheeled mobile-base robots. Among the robot-agnostic middleware, ROS is considered the de facto standard for robot application development [39], officially supporting more than 140 robots (including ground mobile robots, drones, cars, and humanoids) [29]. Examples of repositories from robotics companies that support the integration of ROS are the one from Kuka[10] or from Aldebaran and Softbank Robotics.[11]

MATLAB (and its open-source relatives, such as Octave) is a popular option among engineers for analyzing data and developing control systems. It has also been used for robotics software development [88], and there even exists a robotics-dedicated toolbox.[12] The toolbox contains tools that support functionalities ranging from producing advanced graphs to implementing control systems.

---

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

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

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

[6] http://robot.zaab.org/wp-content/uploads/2014/04/KRL-Reference-Guide-v4_1.pdf

[7] https://wiki.ubuntu.com/Releases

[8] https://wiki.ros.org/Distributions

[9] http://wiki.ros.org/Distributions

[10] https://wiki.ros.org/kuka

[11] https://wiki.ros.org/Aldebaran

[12] https://www.mathworks.com/products/robotics.html

Machine learning is another technique applied in the context of robotics, as is being used in decision making and image recognition. Machine-learning models are first trained using platforms such as Tensorflow or PyTorch and then implemented as ROS nodes [58]. These training platforms provide dedicated APIs, and they are commonly Python or C++-based. Finally, image processing is a key functionality in robotics, and the most used library in this domain is OpenCV[13] [20], written in C++. Its primary interface is written in and uses C++, but there are bindings for Python, Java, and MATLAB.

2.2 IDEs for Developing Robotic Applications

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

Eclipse, in particular with its tooling for model-driven software engineering (e.g., Eclipse Modeling Framework), has been used to realize DSLs and respective environments for building robotics applications in a model-driven way. For instance, Arias et al. [3] offer a complete robotics toolset upon Eclipse to support the engineering from design to code generation, called the ORCCAD model.

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

Table 1 summarizes all the IDEs with details on target users, languages supported, and features that go beyond a general IDE. To illustrate one of the IDEs, Fig. 1 shows a screenshot of the Robot Mesh Studio. The user interface is separated into three main panes. Pane A shows a description of the current mission—rich-text entered by the developer to describe and illustrate the mission (here, the visual recognition and lifting of an object by the robot). Pane B shows the actual mission expressed in an external DSL (with Blockly syntax) provided by the IDE; or alternatively the generated textual code. Pane C shows help text; or alternatively the interactive debugger or an overview on the current robot configuration.

[13] https://opencv.org

Table 1 List of Dedicated Robotic IDEs.

Name	IDE Details
RobotC [79, 81]	C-based educational environment providing two notations, RobotC Graphical e.g. Fig. 7 and RobotC Natural Language e.g. Listing 4
Robot Mesh Studio [78]	IDE for programming educational robots from Arduino, Picaxe, Parallax, and Raspberry Pi micro-controllers. It offers two graphical DSLs: Flowol, a flowchart-based language, and a Blockly-based language. Texual languages: C++, Python
VEX Coding Studio [92, 19]	A robot vendor's environment for programming educational robot kits. The IDE offers Scratch-based syntax (VEXcode Blocks) and a text-based syntax (VEXcode Text).
PICAXE [72, 51]	For programming educational PICAXE micro-controller-based robots. It offers the PICAXE language in three syntaxes: PICAXE BASIC – textual, PICAXE Blockly – graphical, and PICAXE Flowchart syntax
ROS Development Studio [2]	An online IDE with ready-to-use tools, such as simulators and AI-based libraries. The ROS Development Studio supports all robots compatible with ROS and a variety of languages, such as C++, Python, Java, Matlab, and Lisp.
Microsoft Robotic Developer Studio (MRDS) [46]	Microsoft product for hobbyist, academic, and commercial robot application developers. The IDE supports programming robot applications in Microsoft's visual programming language (MVPL) and C#.
MATLAB and Simulink [74]	IDE offers hardware-agnostic robot control for Arduino and Raspberry Pi micro-controllers, that can be connected to ROS and ROS2. Code from a variety of embedded hardware, such as Field Programmable Gate Arrays (FPGAs), Programmable Logic Controllers (PLCs), and Graphics Processing Units (GPUs), can be generated to various target languages including C/C++, VHDL/Verilog, Structured Text, the PLC language, and Compute Unified Device Architecture (CUDA) language.
Webots [68]	An open-source, online IDE simulator that supports a number of robots and a range of languages such as C, C++, Python, Java, MATLAB, and ROS-supported languages C, C++, Python, Java, MATLAB.
Robot Task Commander (RTC) [44]	The IDE is meant for automated task planning for robot(s) using one or more computing devices over a network. It supports humanoid robots programmed using Python scripting language and RTC visual programming language.
The SmartMDSD Toolchain [87]	IDE for developing robot systems by providing building blocks that can be used for composing new systems from existing components. The IDE applies modeling techniques using tools such as Xtext, Xtend, and Sirius from Eclipse.
BRICS Integrated Development Environment (BRIDE) [11]	IDE for developing editors in robotics based on model-driven engineering principles. BRIDE incorporates the OROCOS and ROS frameworks. The ROS version offers features such as graphical modeling of ROS nodes, code generation in C++ or Python, and generation of launch files.
Universal Robotic Body Interface (URBI) [6]	Open-source IDE for programming robot controls, using client-server architecture. The server manages low-level hardware controls for sensors, camera, and speakers, and the client sends high-level behavior commands like "walk" to the server. Languages supported include C++, Urbiscipt scripting language, Matlab, Java and Lisp.
TeamBots [7, 59]	A Java-based environment for developing and executing control systems on teams of robots and on simulation using the application TBSim. The IDE provides a set of applications and packages for multi-agent mobile robots.
Pyro [12]	An educational IDE that abstracts low-level details, making it suitable for students learning to program robots using the C++, Java and Python. Pyro wraps Player/Stage and ARIA, for easy access to its users.
CopellaSim (VREP) [1]	A Multi-robot IDE, which uses distributed control architecture to model objects through: embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution. The IDE supports programming using C/C++, Python, Java, Lua, Matlab or Octave.

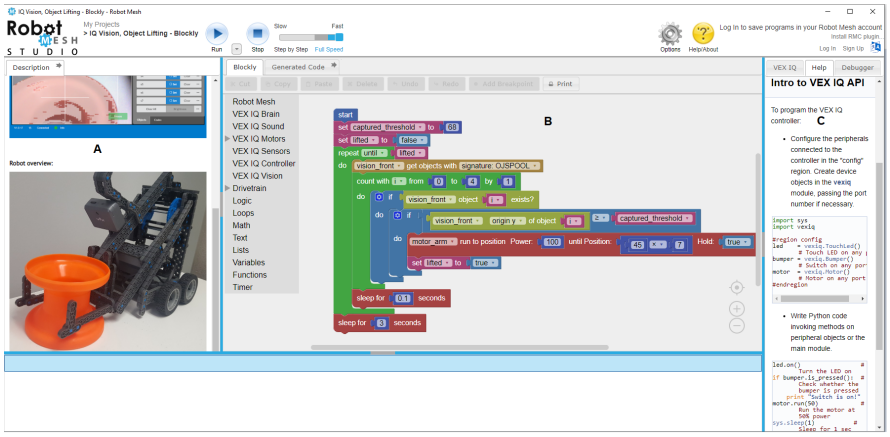


Fig. 1 Screenshot of the Robot Mesh Studio IDE [78] with three panes. Pane A provides an rich-text description of the mission, Pane B the actual mission expressed in a DSL with Blockly syntax, and Pane C shows help text or alternatively the debugger or an overview on the current robot configuration.

3 Robot Mission Specification

As robots become an integral part of the everyday life, we need better ways to instruct robots on the tasks they should accomplish. Mission specification is a process that relies on a strategy and mechanism that determines the steps a robot takes when performing a given task [91, 25, 28, 36].

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

Any mission specified using a DSL should be easily understood by experts in that domain—e.g., logistics, commerce, health. DSLs are recognised for their ability to abstract low-level details of robotic implementations and allowing users to specify their concerns from higher levels by using common terms in the domain. This abstraction further enhances effective communication of concepts with the domain experts. Due to these reasons, DSLs have been studied and proposed for mission specification by the community [36, 83, 42, 26, 27, 70].

DSLs typically work based on the underlying formalisms such as state machines, flow charts, and behavior trees [23, 41, 22]. We assume that the reader has already some knowledge on state machines and flow charts. Before presenting the selected DSLs we give a brief introduction to behavior trees, which are less widely known.

Table 2 List of DSLs, their notation used and their styles (internal or external DSL)

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

A behavior tree is a hierarchical model in which nodes of the tree are tasks to be executed [23, 41, 22]. Behavior trees emphasize modularity, coupled with two-way control transfer using function calls, unlike one-way (transitions) in finite-state machines. The modular character in behavior trees makes the reuse of behavior primitives feasible. Behavior trees have been applied in computing science, robotics, control systems, and video games.[14] Behavior trees consist of control-flow nodes (namely *Parallel*, *Fallback*, *Decorator*, and *Sequence*) and executor nodes (i.e., *Action* and *Condition*). An action node executes a task and returns success or failure, while the condition node tests if a certain condition is met.

In the following subsections, we describe a selection of internal and external DSLs for mission specification together with examples. Internal DSLs are extensions of a general-purpose (i.e., programming) language—often called host language. An external DSL is a language with independent syntax, semantics, and other related language resources, and designed with notation and abstractions suitable to the user domain.Table 2 shows an overview of these DSLs with the notations supported (visual or textual), and style (internal or external DSL).

3.1 Internal DSLs

Internal DSLs follow the host language's syntax, and their execution is limited to the host language's infrastructure. They provide features specific to given end-user domains, such as robotics engineering, which simplify specification of domain user's concerns. In each of the following internal DSLs, we look into the host language, the

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

developing organization (company), its semantics (compiled/interpreted), features specific to the internal DSL, and the end-user domain the language is targeting.

3.1.1 ROS Behavior Tree

ROS Behavior Tree [22] is an open-source C++ library for creating behavior trees. The DSL's aim is to be used by expert robot developers, who are conversant with the ROS framework and C++ or Python languages. Listing 1 shows sample code for creating a behavior tree.[15] It consists of header files and demonstrates how the action node and the condition node are executed in the behavior tree.

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

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

Listing 1 Creation of a new behavior tree using the ROS Behavior Tree DSL.

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

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

3.1.2 SMACH

SMACH [13] is a non-commercial application programming interface written in Python, based on hierarchical concurrent state machines. It allows executions to be controlled by a higher level task-planning system.

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

Figure 2 shows a state machine for a PR2 robot to recharge, specified in Python.[16] Listing 2 demonstrates how to create a state machine, adding states to the state machine. In the state execution (line 10), "event" depicts the condition to execute outcome1 if true, outcome2 otherwise.

```
1  #!/usr/bin/env python
2   import rospy
3   import smach
4  # creating a state
5   class Foo(smach.State):
6       def __init__(self, outcomes=['outcome1', 'outcome2']):
7  # Your state initialization goes here
8       def execute(self, userdata):
9  # Your state execution goes here
10        if event:
11            return 'outcome1'
12        else:
13            return 'outcome2'
14 # Adding states
15  sm = smach.StateMachine(outcomes=['outcome4','outcome5'])
16    with sm:
17      smach.StateMachine.add('FOO', Foo(),
18                             transitions={'outcome1':'BAR',
19                                          'outcome2':'outcome4'})
20      smach.StateMachine.add('BAR', Bar(),
21                             transitions={'outcome2':'FOO'})
```

Listing 2 Creation of a state and adding the state to a state machine in SMACH

3.1.3 C-based Agent Behavior Specification Language

The C-based Agent behavior specification language (CABSL) [80] enables the description of robot behaviors as a hierarchy of finite state machines. The control program executes behaviors based on the acquired sensor data, which maps the sensor

[16] https://wiki.ros.org/smach/Tutorials/Getting%20Started

data to actions the robot executes. In a state, when an action is taken, either the state generates an output or calls another state machine. Otherwise, there is a transition from one state to another state.

An active graph in CABSL is a tree consisting of a set of state machines being executed. Each state machine can call any other state machine. The language is implemented and compiled in C++.[17] CABSL does not provide the functionality of replacing the behavior on the fly in case acquired sensor data requires a change in behavior.

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

3.2 External DSLs

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

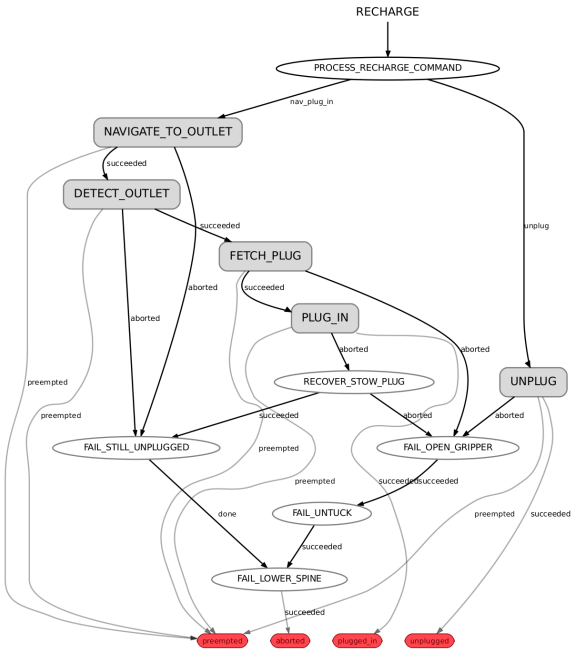


Fig. 2 State machine for a mission that instructs the PR2 robot to charge itself at a standard outlet.

[17] `https://github.com/bhuman/CABSL`

the external DSL, type of robots the DSL supports, and the domain the language is targeting.

3.2.1 NaoText

NaoText [42] is an external DSL developed by the research group QualiTune. The DSL is a role-based language for specifying collaborative missions for Nao robots using a textual notation. NaoText uses CPSTextInterpreter, which runs on the Java runtime environment using Maven to manage dependencies.[18]

The code below shows the declaration of a pass action in a soccer game between Nao robots.[19] Some of the domain terms used in specifying the mission in Listing 3 include striker, ballpossesor, and ballseeker.

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

Listing 3 A snippet of a mission for Pass Action in a soccer game between Nao robots.

3.2.2 EasyC

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

EasyC uses a graphical interface on top of Intelitek's own C library[20], which was custom made for the VEX Cortex and IQ robot controllers. Figure 3 is a screenshot of the DSL showing abstractions of the language concepts, a sample mission specified, and the generated C program.

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

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

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

3.2.3 BehaviorTree.CPP

BehaviorTree.CPP [30] is a C++ library for creating behavior trees. It is developed by a research group at the Eurecat Technology Center. The library provides a flexible framework to easily specify robot mission as behavior trees that can be loaded at run-time for execution. The nodes of the tree are either actions the robot can execute or conditions to be fulfilled before an action is taken.

The BehaviorTree.CPP DSL provides mechanisms to monitor, log, and debug the execution of a tree. The behavior trees for robot missions are executed using the C++ language run-time environment. Groot[21] provides a graphical editor for the C++ library to create and edit behavior trees. The primitives in Groot, built-in nodes, or custom nodes can be dragged and dropped to build a required behavior tree. Domain terms and expressions such as DetectObject, Grasp, GetMapLocation, and MoveTo, have been used in the DSL for mobile robots with the ability to move, recognize, and grasp objects.

3.2.4 Unreal Engine 4 Behavior Trees

The Unreal Engine 4 (UE4) Behavior Tree [8] is a commercial DSL developed and maintained by Epic Games, Inc. Behavior trees define the Unreal AI agent's processor, which makes decisions and executes various branches based on the outcome of those decisions. The Unreal Engine implements behavior trees using the Blackboard tool

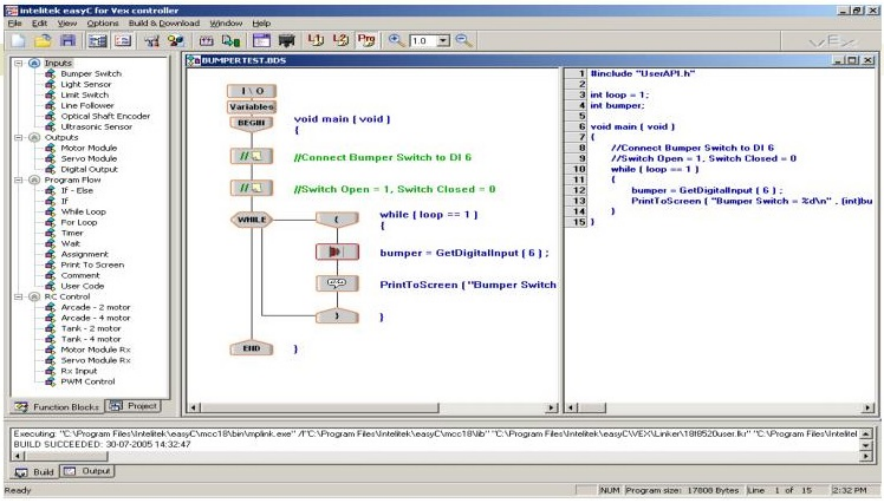


Fig. 3 A screenshot showing EasyC DSL: first column presents the language feature abstractions, second column is the active mission being specified while the last column shows the generated code

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

which acts as the "brain" of the AI character and stores key values that the behavior tree uses to make its decisions. A behavior tree task is an action the AI character can perform, for instance, move to a location or rotate to face an object.[22] Some examples of domain expressions are SetMovementSpeed, LookStraightAhead, and RapidMoveTo. The DSL has been used for simulation characters in video games, representing humans, helicopters, and vehicles. The Unreal Engine uses the Unreal scripting languages with a graphical editor for creating UE4 behavior trees, related blackboards for the behavior trees, and tasks—i.e., actions. The scripting languages are compiled using the UnrealScript Compiler.[23]

3.2.5 Choregraphe

Choregraphe [75, 69] is a commercial DSL produced and maintained by SoftBank Robotics for programming Aldebaran robots such as NAO. The language aids users to create animations, behaviors, and dialogues for the NAO humanoid robot—meant for experimentation and research. Choregraphe also offers simulation support for the NAO robot. The graphical DSL provides a flow-chart-like interface in which end-users specify missions by connecting boxes to construct a behavior for the robot.[24] Boxes are pre-programmed libraries, which abstract mission primitives. Some of the mission primitives include Play Sound, Set Speech Lang, and Speech Reco.

3.2.6 Microsoft Visual Programming Language

The Microsoft Visual Programming Language (MVPL) [46] is a DSL in Microsoft Robot Development Studio used for programming robotic applications based on the idea of boxes and arrows. The language concepts (activities) are represented by boxes while the arrows connect the boxes to build a program.[25] The MVPL data-flow diagram consists of a connected sequence of activities represented as blocks with inputs and outputs that can be connected to other activity blocks. A sample program is shown in Fig. 4. Activities can represent data flow control or processing functions, or user-defined activities, which the user creates in MVPL.

3.2.7 Open Roberta

Open Roberta [34, 52, 54] is a web-based educational DSL developed by the Fraunhofer Institute, which offers free use for individuals, but commercial use for institutional use. The Blockly-based DSL can be used to program a variety of robots:

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

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

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

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

Lego Mindstorms EV3 and NXT, Calliope mini, micro:bit, Bot'n Roll, NAO, and BOB3. This DSL provides a rich set of behavior abstractions and primitives, which are mainly categorized into actions (drive, turn, steer, show, play, say), sensors (touch, ultrasonic, color, infrared, temperature, gyro, timer), control (program control flows), logic (comparisons, AND, OR, Boolean), math (constants and arithmetic operators), text, colors, and variables. The specifics of these abstractions vary according to the robot for which the mission is specified. This DSL has a considerable potential in harnessing end-user programming, since it is a drag and drop graphical language with syntactic and semantic editor services. The DSL can either be run on the cloud or installed on a local server. Open Roberta generates Code in Python, Java, Javascript, and C/C++ depending on the target robot.

3.2.8 FLYAQ

FLYAQ [28, 32, 25, 14] is an open-source research prototype developed and maintained by a team of researchers that provides an extensible DSL for specifying missions for a variety of robots, including quadrotors. The monitoring mission language (MML) allows specification of mission context such as obstacles, flight path (i.e., starting point, action points, ending point), and no-fly zones on a live map. The executable code is automatically generated to be executed by a robot or a swarm of robots as shown in Fig. 5. The DSL is suitable for missions such as surveillance, public order management, and agriculture. The concrete syntax (i.e., the map) used

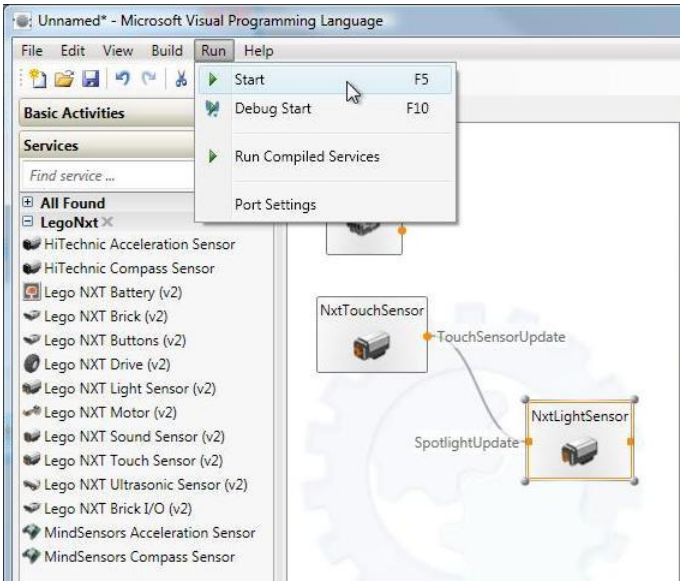


Fig. 4 A screenshot of the MVPL DSL showing a data flow program to connect to switch on light

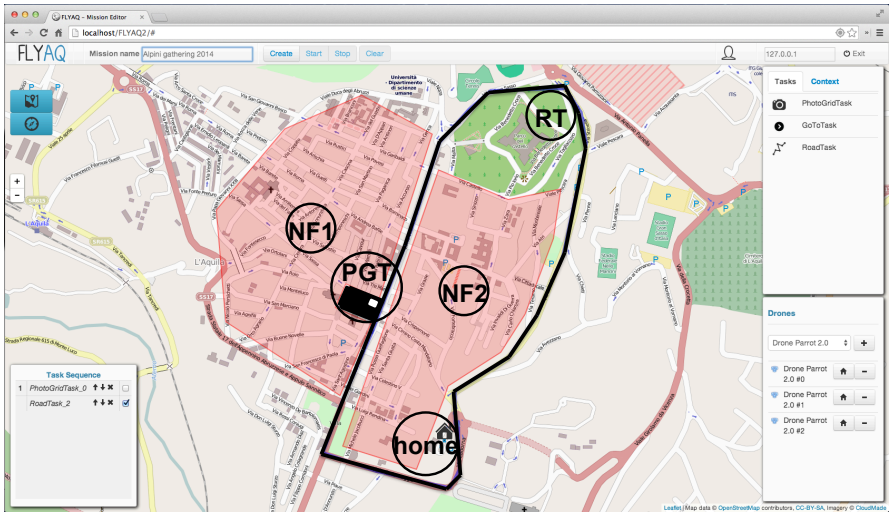


Fig. 5 Specifying a mission for a drone to hover in given space while avoiding no-fly zones [26].

for specifying the mission context makes the language reachable for end-users. The FLYAQ virtual machine provides a ready-to-run version for the end-user.

3.2.9 Aseba

Aseba [90, 63] is a DSL with variants of visual and text syntaxes, created by Mobsya under Creative Commons Attribution-ShareAlike 3.0 License. The language syntax variants are the Visual programming language (VPL), a Blockly-based language, a Scratch-based language, and the Aseba textual language. VPL provides events and action-based programming in which, for a given event, there is a corresponding action. These events are triggered by data from sensor readings. Examples of events are press button, obstacle detector, ground detector, robot tapped, and hand clap. In turn, examples of actions are set motor speed, set top or bottom color, and play music. The same language concepts can be programmed using the other DSLs of Aseba. Some common behaviors[26] associated with the Thymio robot are: friendly (follow hand and react to another Thymio robot), explore (avoid obstacles and stop when the ground is dark), fearful (goes away when approached and scream when cornered), attentive (changes color and moves depending on the number of claps detected), investigator (follows a black track), and obedience (reacts to button and remote control).

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

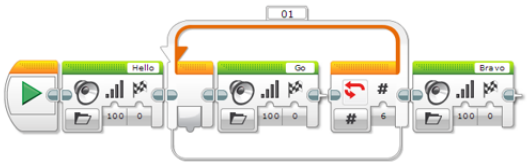


Fig. 6 Specifying a loop in LEGO Mindstorms EV3.

3.2.10 LEGO Mindstorms EV3

The LEGO Mindstorms EV3 builder [61, 17] makes it possible to create robots that can do a number of things such as walk, talk, or drive. The graphical DSL provides a rich set of language constructs categorized into action, flow, sensor, data, and advanced blocks. For instance, the action blocks include move steering block, display block, and sound block, which can be used for specifying a mission by kids learning how to program. The DSL is a visual language with blocks connected to form missions. Figure 6 shows a mission specification in which the robot says "Hello" once, then "Go" six times, and then "Bravo" once. The sound blocks are used for creating the respective sounds while the flow block—the loop is used for repeating the "Go" sound. Each block is an icon of the function it executes.

3.2.11 MissionLab

MissionLab [4, 91] was created by the Mobile Robot Laboratory at Georgia Tech and is a research prototype DSL that facilitates mission specification through a state-machine-based visual language. The DSL uses assemblage and temporal sequencing constructs to create a temporal chain of behaviors as a mission. The assemblage construct defines behavior primitives and coordination mechanisms. During mission specification, the assemblage is instantiated. The temporal sequencing creates states with perceptual triggers to enable transitions between states. MissionLab provides a graphical editor-based configuration description language (CDL) to specify multi-agent missions. Missions can be executed on a simulator or on the following wheeled robots used for smaller commercial applications: ATRV-Jr, Urban Robot, AmigoBot, Pioneer AT, and Nomad 150 & 200.

3.2.12 RobotC

Figure 7 shows a sample program demonstrating how RobotC graphical language is used to write a robot program that counts and displays the number of times a button is pressed. The language primitives are in form of blocks, which users drag and drop to build a program, making it easy for novice programmers write robot missions.

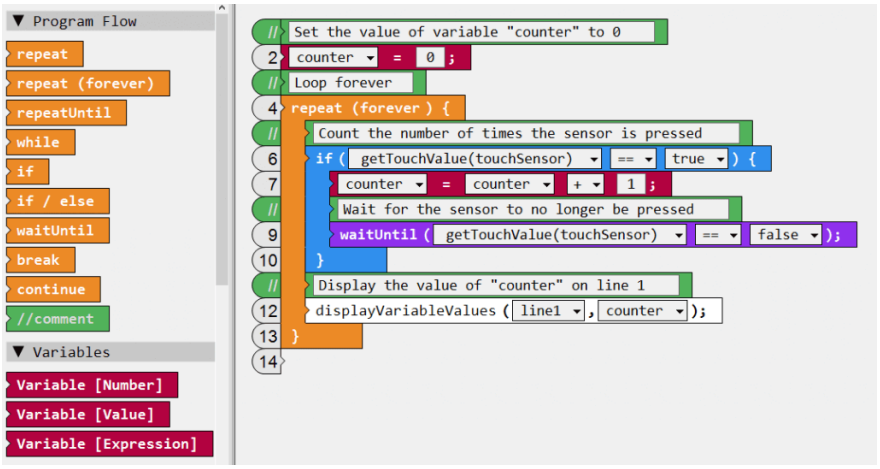


Fig. 7 Screenshot of RobotC Graphical DSL's click-and-drag command blocks [79].

Listing 4 illustrates the same program in textual version of RobotC Natural language.[27] The expressions preserve the C language syntax while the natural language makes it easy for novice programmers to comprehend the programs.

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

Listing 4 Snippet of a program in RobotC Natural Language

4 Making Robots Usable in the Everyday Life

Mobile robots are increasingly used in everyday life to autonomously realize missions such as exploring rooms, delivering goods, or following certain paths for surveillance. The current robotic market is asking for a radical shift in the development of robotic applications where mission specification is performed by end-users that are not

[27] http://www.robotc.net/NaturalLanguage/

highly qualified and specialized in robotics or ICT. To this end, in the context of the Co4Robots EU H2020 project,[28] we developed our contribution in two steps.

- First (Sec. 4.1), with the aim of understanding the missions that are currently expressed in practice, we surveyed the state of the art and formulated and formalized a catalog of 22 mission specification patterns for mobile robots. We also provide tooling for instantiating, composing, and compiling the patterns to create mission specifications [66, 67].
- Second (Sec. 4.2), using specification patterns as main building blocks, we proposed a DSL that enables non-technical users to specify missions for a team of autonomous robots in a user-friendly and effective way [36, 37].[29]

4.1 Mission Specification Patterns

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

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

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

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

[28] `http://www.co4robots.eu`

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

the specification of complex missions by the conjunction or disjunction of the patterns [66].

We thoroughly validated the patterns [67]. We evaluated the benefits of using our patterns for designing missions by collecting 441 mission requirements in natural language: 436 obtained from robotics development environments used by practitioners, and five defined in collaboration with two well-known robotics companies. Further information about the theoretical aspects might be found in [67], and about the tool in [66], while details about the specification of each pattern might be found in the website[30].

4.2 PROMISE

In order to support the specification of more complex missions with respect to those that can be specified using the specification patterns, and in order to enable the specification of missions for multiple robots, we proposed a Domain-Specific Language called PROMISE. PROMISE considers the mission specification patterns as atomic tasks that can be executed by robots and proposes sophisticated composition operators for describing complex and multi-robot missions. These operators are inspired by behaviour trees [21, 50] in their style and notation. The DSL is integrated into a framework,[31] which allows the seamless specification and execution of a mission. The framework contains:

1. The realization of the language using Eclipse and two plugins for language workbench, namely Xtext[32] and Sirius.[33] In this way, mission specification can be performed through textual and graphical interfaces, which are synchronized.
2. A compiler implemented using Xtend[34] for mission code generation.
3. An interpreter, which parses the mission code and gives the low-level commands to each robot accordingly.

While the DSL support is provided by a standalone tool and can be integrated within a variety of frameworks, the current implementation has been integrated with a software platform [35] that provides a set of functionalities, including motion control, collision avoidance, image recognition, SLAM, and planning. This software platform has been implemented in ROS.

Our DSL has been successfully validated through experimentation with both simulation and real robots. Footage of the validation through experimentation we have conducted can be found on the dedicated website. The experimentation led to a demonstration of several missions to the Co4Robots consortium, which triggered important feedback. For instance, an industrial partner from the Bosch Center of

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

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

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

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

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

Artificial Intelligence suggested that practitioners from their logistics facilities would appreciate a response from the tool stating a natural English description of the mission that had been specified. An example of such a description is provided in Section 5. We targeted specific robots during the experimentation, however, PROMISE is intended to be robot-agnostic, so it could be integrated with any robot by modifying the interpreter with the interfaces required for the new robot. The experimentation enabled us to validate PROMISE from the point of view of expressiveness by measuring the ability to write missions defined by practitioners, as we will detail in Section 5.

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

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

Further information regarding PROMISE and the validation procedure we followed during its development might be found in our previous study [36], in a tool paper [37], and in the DSL's dedicated webpage.

5 Putting PROMISE into Practice

In the previous section, we introduced the methods and mechanisms we developed to make robots usable in the everyday life in a descriptive way. In the following, we present an example of a mission and its specification using PROMISE together with a comprehensive discussion of the context in which it has been defined. This

example originates from our work in the Co4Robots project,[35] which aimed at of developing a full functioning robot that integrates several robotic skills that we have developed, including navigation, self-localization, and planning. Its focus is on robotic applications realized on top of robotic platforms provided by our industrial partners, including a TIAGo robot[36] and an ITA robot[37], both in real life and simulation. To test our developments, when we could not directly access any of these robots, we used an economic and easy-to-use robot, the Turtlebot 2.[38] It does not provide a wide range of functionalities, but allows easy prototyping, while testing recognition and navigation skills before deployment to the production-level robots TIAGo or ITA.

Our example scenario is inspired by a mission proposed for the 2018 edition of the well-known robotics competition Robocup@Home. We replicated and made available in PROMISE's repository several missions proposed in the rules of this Robocup@Home'18 [64]. Concretely, we use here the restaurant simulation scenario as an example. In this scenario, two robots collaborate to help clients in a simulated restaurant at the same time. The robots are required to ask the customers for their order and deliver drinks or snacks provided by a barman (i.e., the human *operator*), while people walk around. Both robots must work in parallel.

For our project, we have used Python as the development programming language since it is one of the most common languages used in robotics together with C++ [39], as discussed in Sec. 2. It is also well-supported by the Robot Operating System (ROS) [76] middleware. Many libraries, such as for testing or developing dedicated skills, are also written in Python. As anticipated above, we make use of ROS since it is the most widespread middleware and it is used by the Turtlebot 2 and TIAGo.

Next, also influenced by our middleware choice, we have designed a three-layered software architecture for the software, because it supports the separation of concerns among processes with different layers of abstraction [35]. We have also opted to adhere to a component-based approach, mostly because ROS enforces the component-based software development with its clustering of software modules into packages and nodes. If properly performed, the step of designing and adhering to a software architecture simplifies the later integration of robotic skills while promoting their documentation. As a mainstream IDE, we used Eclipse. In Section 2.2 we present and discuss popular IDEs that support users in developing robotics software, distinguishing between mainstream IDEs, such as Eclipse, which are extensible via plugins for various robotics aspects, and dedicated robotics IDEs.

Figure 8 shows the representation of the restaurant scenario using the graphical syntax of PROMISE. The image has been edited with circled numbers to label nodes and ease the explanation of the mission. In turn, Fig. 9 shows the textual representation of the same mission. This figure also contains circled numbers, which label the same nodes and therefore supports the reader while linking the graphical-textual mapping.

---

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

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

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

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

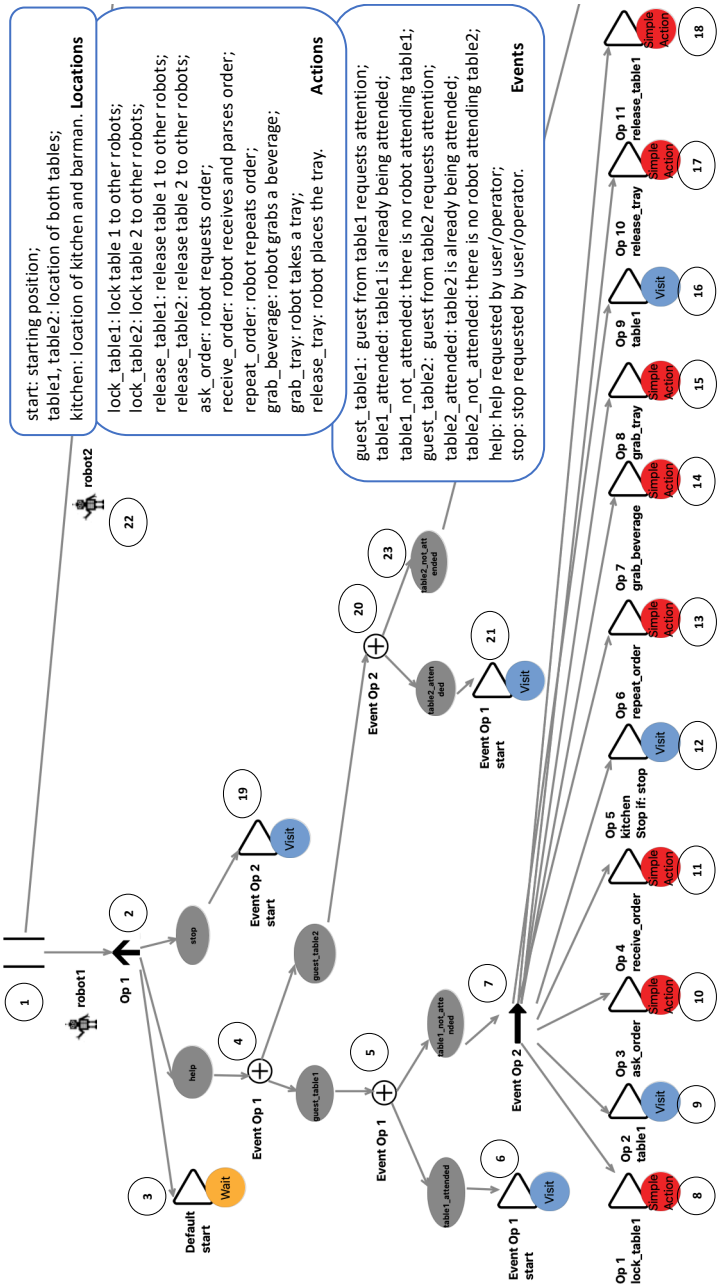


Fig. 8 Running example specified with the graphical syntax of PROMISE.

Running example: mission defined using PROMISE.
The root of the mission specification, i.e., the operator *parallel*, is identified by the

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

Fig. 9 Running example specified with the graphical syntax of PROMISE.

node ① and specifies that *robot1* and *robot2* must perform their missions in parallel. A robot is assigned to each branch associated with this operator, as indicated with labels in the edges between ① and ② in Fig. 8, and with the name of the assigned robot (i.e., *robot1* and *robot2*) in Fig. 9. Since the mission for *robot2* (㉒) is a replica of the one for *robot1*, we only show the latter for the sake of conciseness.

The operator labeled with ② is the *eventHandler*—more information regarding PROMISE's operators is available in [36]. It has a default behavior; in our example, it forces the robot to wait in location *start* (③). This behavior is paused when one of the events that are assigned to the *eventHandler* is detected by the robot. The default robot's behavior (③) is resumed whenever any of the behaviors triggered by an event is finished (either succeeding or failing).

Each event is assigned to a child of the *eventHandler* (as represented in Fig. 8) as gray circles and invoked in Fig. 9 by the keyword `except`. If the event "help" is

detected, the first operator *condition* (④) is executed. This operator evaluates its associated events in order, and if they hold, it triggers the behaviors associated with them. In this case, the operator *condition* evaluates whether the request of help comes from $table1$ or $table2$.

In case "guest_table1" holds (i.e., the request of help comes from $table1$), another operator *condition* (⑤) is executed. This operator evaluates whether this table is already being attended by another robot ("table1_attended") and in this case, makes the robot return to the starting position $start$. This behavior is encoded by the instantiation of an operator *delegate* with a task *Visit* (⑥).

The next operator *condition* (⑤) evaluates "table1_not_attended," and, if it holds, the execution of an operator *sequence* (⑦) is triggered. This operator executes in sequence a set of operators. Concretely, the sequence of operators starts with ⑧, which "locks" $table1$ from the rest of the robotic team by forwarding a message (in this case, other robots will recognize it with the event "table1_attended'). The robot will then move to $table1$ (⑨), ask the order (⑩), and receive and parse it (⑪). The robot will then move to $kitchen$ (⑫) to interact with the barman (i.e., the human operator). Note that this specific task can be stopped by the user or human operator by means of the event "stop" (see Fig. 8). Once the robot has reached location $kitchen$, it will repeat the order to the barman (⑬), after which the robot will grab beverages (⑭) and a tray with the ordered snacks (⑮). The robot will then return to $table1$ (⑯) with the order, where it will place the tray (⑰). The sequence of tasks finishes with the robot "releasing" the table for other robots, in a similar way as to how it locked it.

The operator *condition* ⑳ is a replica of ⑤—see the conditions in the textual representation in Fig. 9 "if guest_table1" and "if guest_table2"—and, therefore, we do not show its whole graphical representation for the sake of conciseness.

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

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

The mission of the example was modeled through mission specification from a natural English description, in this case, from the rules of the Robocup@Home'18 [64].

Once the mission was modeled, we proceeded to validate it through experimentation in an iterative way. The first step we took was simulation,[39] for which we used simulated models of the facilities and robotic models provided by the industrial partners of Co4Robots for Gazebo [57]. Once the simulation was performed and the mission specification validated we proceeded with validation in real life. As explained above, we purchased a Turtlebot2 for experimentation. We validated the same restaurant scenario with the Turtlebot in the facilities of the University of Gothenburg.[40] The last step was to conduct a demonstration in presence of the project consortium at the facilities of PAL Robotics, for which we used a TIAGo robot.[41] Through this process, we demonstrated the ability of PROMISE to specify complex missions from textual descriptions. We also demonstrated its capability to operate with different robots by accordingly modifying the interpreter of its framework—see Section 4.2.

We invite the interested reader to learn more about the validation procedures we followed during the development of PROMISE in our published studies [36, 38] and on its dedicated website.

6 Discussion and Perspectives for Future Research

As discussed in this paper, in the last years there have been many contributions from the research community to propose domain-specific languages for mission specification [36, 14], the description of missions in natural language [62], and visual and end-user-oriented mission environments [95, 10, 77, 70].

The approaches surveyed here greatly contribute to the field, however the mission specification-problem still requires solutions able to make robots usable in everyday life for accomplishing complex missions. Here in the following we highlight the limitations of current approaches and we devise perspectives for future research. As stated also in the Multi-Annual Roadmap For Robotics in Europe (MAR) [85], in order to reduce costs and establish a vibrant component market, there is a need for instruments for supporting mission *reuse* and diversification, as well as coping with the *variability* of conditions of application scenarios occurring in real environments. This is also testified by our findings during our collaboration with practitioners in the robotic domain: the complexity does not reside in commanding a robot with a set of tasks but in making the robotic application robust enough to be able to cope with the variability that characterises the real environments in which the robots are required to operate, especially those that involve humans [39].

To the best of our knowledge few approaches try to address the reusability and variability envisioned by the MAR. PROMISE and the specification patterns are greatly contributing, however there are some aspects that should be investigated in the future. Here in the following we devise important research directions, which we

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

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

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

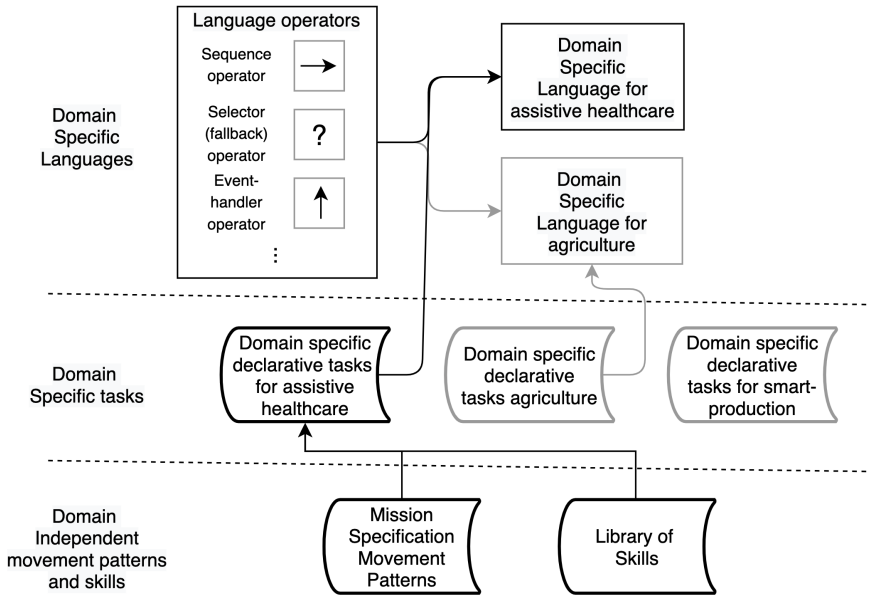


Fig. 10 Mission specification.

identified based on our collaboration with companies in the Co4Robot project and additional collaborations in the healthcare domain. Specifically, we believe that the main research directions go in the following directions:

- *Reusability*: the DSLs we will develop for enabling end-users to specify missions will make use of libraries of tasks and skills. They will also integrate with libraries produced by other projects and initiatives, like RobMoSys.[42]
- *Variability of the real world*: the DSL will be conceived to enable the specification of the variability of conditions of complex real-word scenarios.
- *Fleet specification of a mission*: the end-user that will specify the mission does not need to assign tasks to specific robots, but the mission specification will represent the “needs" of the end-user and robots will be automatically assigned, and potentially re-assigned during the mission execution, according to the capabilities or robots and various quality parameters.
- *Human-robot collaboration*: the mission specification will include also humans, with two different roles, namely, *operators*, able to perform actions needed to successfully accomplish the mission, and *patients*, which will require actions from robots.

In order to support what we believe we might need, various libraries of pre-defined solution schemes that can be reused, instantiated, and composed by means of properly

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

defined operators need to be implemented. As shown in Fig. 10[43] we envision three different types of libraries organized on two levels, one being application-domain-independent (specific for service robots), and the other one being domain-specific, e.g., assistive healthcare, agriculture or smart production.

- *Mission specification Movement patterns* are pre-defined solutions concerning movements of robots, and provide the bridge between a mission requirement expressed in structured English (subset of English with a well-defined semantics) and a formulation in temporal logic. An initial result in this direction consists in the specification patterns described in Section 4.1.
- *Library of skills* contains the implementation of the modules for enabling the robot to do specific actions, like grasp object with constraint, low dexterity, soft grasping, image recognition, gesture recognition, and so on, that are compliant with the RobMoSys platform.
- *Domain-specific declarative tasks* are recurrent combinations of mission-specifications patterns and skills used to define declarative tasks for domain-specific operations. For instance, in the assistive healthcare domain, a declarative task can be "welcome," and would require patterns for movements and various skills such as human recognition, speech recognition, etc. The tasks are declarative since they specify only what the robot is able to do without saying how the robot will do that. Then, planners will compute how the task will be solved in the specific environment according to the capabilities of the robot that will be allocated to this task.
- *Domain-Specific Languages*, as for instance PROMISE (Section 4.2), enable operators who are not required to have expertise in programming nor robotics, to specify in an easy and correct way the mission they would like the robots to safely accomplish. Each domain-specific language will make use of the language operators that we will define. There will also be specific "dialects" for specializing the language to the various domains. In this way, healthcare operators will find in the domain specific language for assistive healthcare concepts that are specific of the domain, expressed in terms of domain-specific declarative tasks for assistive healthcare. The language enables the description of complex and sophisticated missions, which will also take into account non-functional properties, such as timing constraints. These properties are captured by composition operators, like *sequence*, *selector* (fallback), or *event-handler*, which are inspired by Behaviour trees [21, 50] or by PROMISE.[44] The DSL will help healthcare operators (with a sort of wizard or recommendation system) to deal with the variability that characterizes the environments in which missions are executed. This includes "exceptional" behaviours, such as a robot running out of battery, an unforeseen obstacle hampering the mission satisfaction, an object falling down from the hand of the robot, and so on. As testified by MAR [85] and also highlighted in a

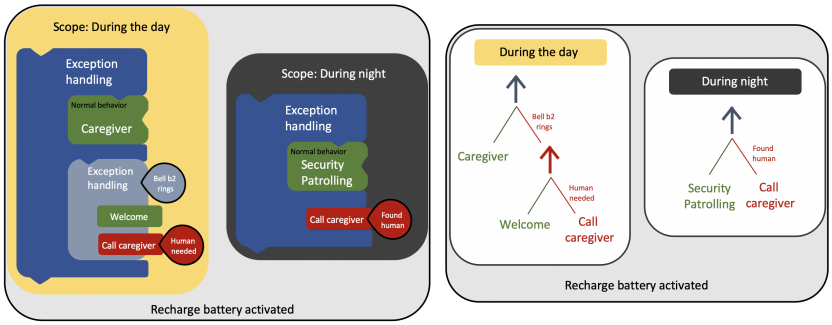
[43] We use the same terminology in "Architectural Pattern for Task-Plot Coordination" of the EU H2020 RobMoSys project: https://robmosys.eu/wiki/general_principles:architectural_patterns:robotic_behavior

[44] The PROMISE DSL has been developed in the context of the EU H2020 project Co4Robots [36]. PROMISE webpage: https://github.com/SergioGarG/PROMISE_implementation

recent study [39], one of the most difficult aspect in mission specification is to deal with the variability of real-world scenarios.

Example of Mission specification. During the day "robot" welcomes newcomers when the bell "s2" of the door rings. According to the needs of the guests, "robot" will provide the needed information or ask them to enter the dining room, and if a human intervention is needed, "robot" informs a caregiver. When "robot" is in the dining room, it acts as a caregiver and interacts with people by calling them to drink and offering water that "tray" carries. During night, "robot" patrols for security and if it finds humans in the environment it calls an operator. Robots recharge autonomously while guaranteeing the welcoming and caregiving service. Notice that the example does not include quality aspects, such as timing constraints, since patterns including these aspects are not yet available, but they will be developed during the project execution.

Mission specification: A healthcare operator will specify the mission by means of the following domain specific macros: *Welcome*, *Security Patrolling*, *Caregiver*, and *Call caregiver*. The following figure shows the mission specified foreseeing two different graphical languages, one (a) based on the *blockly*[45] approach, and the other one (b) using PROMISE's style [36]. This is just to explain what we mean by graphical and easy-to-use language for mission specification.



a) Blockly style b) Promise style

Domain specific macros, mission specification patterns, and library of tasks: Behind the scene, i.e., invisible to the end-users, the macros will be built by using the mission- specification patterns and the tasks stored in the library. For instance, *welcoming* might be realized by composing the *sequenced visit specification pattern*[46] to reach from the current location of the robot the door (LTL formula: $\Diamond$(door_location)), with a *delayed action*[47] when the robot reaches the door to welcome and activate the speech recognition—LTL formula: $\Box$(door_location $\Rightarrow \Diamond$(welcome)), where "welcome" is a task in the library of tasks.

[45] `https://developers.google.com/blockly`

[46] `http://roboticpatterns.com/pattern/sequencedvisit/`

[47] `http://roboticpatterns.com/pattern/delayedreaction/`

Acknowledgments

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

References

1. Copella simulator (2020). URL https://www.coppeliarobotics.com/
2. Ros development studio (2020). URL https://www.theconstructsim.com/rds-ros-development-studio
3. Arias, S., Boudin, F., Pissard-gibollet, R., Simon, D., Arias, S., Boudin, F., Pissard-gibollet, R., Orccad, D.S., Arias, S., Boudin, F., Pissard-gibollet, R., Simon, D.: ORCCAD , robot controller model and its support using Eclipse Modeling tools (2010)
4. Arkin, R.: Missionlab: Multiagent robotics meets visual programming. Working notes of Tutorial on Mobile Robot Programming Paradigms, ICRA **15** (2002)
5. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Software Eng. **41**(7), 620–638 (2015)
6. Baillie, J.C.: URBI: towards a universal robotic body interface. pp. 33 – 51 Vol. 1 (2004). DOI 10.1109/ICHR.2004.1442112
7. Balch, T.: Teambots 2.0: https://www.cs.cmu.edu/~trb/TeamBots/ (2000)
8. Båtelsson, H.: Behavior trees in the unreal engine: Function and application (2016)
9. Berger, T., Steghöfer, J.P., Ziadi, T., Robin, J., Martinez, J.: The state of adoption and the challenges of systematic variability management in industry. Empirical Software Engineering **25**, 1755–1797 (2020)
10. Biggs, G., Macdonald, B.: A survey of robot programming systems. In: Proceedings of the Australasian Conference on Robotics and Automation, CSIRO, p. 27 (2003)
11. Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Bruyninckx, H., Soetens, P., Hägele, M., Pott, A., Breedveld, P., Broenink, J., Brugali, D., Tomatis, N.: BRICS - best practice in robotics. pp. 1–8 (2010)
12. Blank, D., Kumar, D., Meeden, L., Yanco, H.: Pyro: A Python-based versatile programming environment for teaching robotics. Journal on Educational Resources in Computing (JERIC) **3**(4), 1–es (2003)
13. Bohren, J., Cousins, S.: The SMACH high-level executive [ROS news]. IEEE Robotics & Automation Magazine **17**(4), 18–20 (2010)
14. Bozhinoski, D., Di Ruscio, D., Malavolta, I., Pelliccione, P., Tivoli, M.: Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 801–806 (2015)
15. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012)
16. Brugali, D., Agah, A., MacDonald, B., Nesnas, I.A., Smart, W.D.: Trends in robot software domain engineering. In: Software Engineering for Experimental Robotics, pp. 3–8. Springer (2007)
17. Burnett, W.: http://www.legoengineering.com/alternative-programming-languages/ (2018)

18. Bézivin, J.: On the unification power of models. Software and System Modeling **4**(2), 171–188 (2005)
19. Caron, D.: competitive robotics the best brings out in students, Tech Directions, v69 n6 p21-23 Jan 2010, https://eric.ed.gov/?id=EJ894879 pp. 21–24 (2010)
20. Cicolani, J.: Beginning Robotics with Raspberry Pi and Arduino: Using Python and OpenCV. Apress (2018)
21. Colledanchise, M.: Behavior trees in robotics. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2017)
22. Colledanchise, M., Ögren, P.: How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. IEEE Transactions on Robotics **33**(2), 372–389 (2017). DOI 10.1109/TRO.2016.2633567
23. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: An introduction. CRC Press (2018)
24. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices **35**(6), 26–36 (2000). DOI 10.1145/352029.352035. URL http://doi.acm.org/10.1145/352029.352035
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. Di Ruscio, 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, MODELS. ACM (2016)
27. Doherty, P., Heintz, F., Kvarnström, J.: High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. Unmanned Systems **1**(01), 75–119 (2013)
28. Dragule, S., Meyers, B., Pelliccione, P.: A generated property specification language for resilient multirobot missions. In: A. Romanovsky, E.A. Troubitsyna (eds.) Software Engineering for Resilient Systems, pp. 45–61. Springer International Publishing, Cham (2017)
29. Estefo, P., Simmonds, J., Robbes, R., Fabry, J.: The robot operating system: Package reuse and community dynamics. Journal of Systems and Software **151**, 226–242 (2019)
30. Faconti, D.: Models and Tools to design Robotic Behaviors. Tech. Rep. 732410, Eurecat Tecnológic, Barcelona Spain (2020). URL https://github.com/BehaviorTree/BehaviorTree.CPP/blob/master/MOOD2Be_final_report.pdf
31. Finucane, C., Jing, G., Kress-Gazit, H.: LTLMoP: Experimenting with language, temporal logic and robot control. In: International Conference on Intelligent Robots and Systems (IROS), pp. 1988–1993. IEEE (2010)
32. FLYAQ: http://www.flyaq.it/ (2019)
33. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley (2011)
34. Fraunhofer IAIS: https://lab.open-roberta.org/ (2019)
35. García, S., Menghi, C., Pelliccione, P., Berger, T., Wohlrab, R.: An architecture for decentralized, collaborative, and autonomous robots. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 75–7509. IEEE (2018)
36. García, S., Pelliccione, P., Menghi, C., Berger, T., Bures, T.: High-level mission specification for multiple robots. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019 (2019)
37. García, S., Pelliccione, P., Menghi, C., Berger, T., Bures, T.: Promise: High-level mission specification for multiple robots. In: 42nd International Conference on Software Engineering (ICSE 2020 Demos) (2020)
38. García, S., Pelliccione, P., Menghi, C., Berger, T., Bures, T.: Promise: High-level mission specification for multiple robots (2020)
39. García, S., Strüber, D., Brugali, D., Berger, T., Pelliccione, P.: Robotics software engineering: A perspective from the service robotics domain. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020) (2020)

40. García, S., Strüber, D., Brugali, D., Di Fava, A., Schillinger, P., Pelliccione, P., Berger, T.: Variability modeling of service robots: Experiences and challenges. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 1–6 (2019)
41. Ghzouli, R., Berger, T., Johnsen, E.B., Dragule, S., Wasowski, A.: Behavior trees in action: A study of robotics applications. In: 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE) (2020)
42. Götz, S., Leuthäuser, M., Reimann, J., Schroeter, J., Wende, C., Wilke, C., Aßmann, U.: A role-based language for collaborative robot applications. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, pp. 1–15. Springer (2011)
43. Guo, M., Johansson, K.H., Dimarogonas, D.: Revising motion planning under linear temporal logic specifications in partially known workspaces. In: International Conference on Robotics and Automation (2013)
44. Hart, S., Dinh, P., Yamokoski, J.D., Wightman, B., Radford, N.: Robot task commander: A framework and ide for robot application development. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1547–1554 (2014)
45. Hentout, A., Maoudj, A., Bouzouia, B.: A survey of development frameworks for robotics. In: 2016 8th International Conference on Modelling, Identification and Control (ICMIC), pp. 67–72 (2016)
46. Ho, R.P.Y.: Configuration of robotics solutions in microsoft robotics developer studio, `http://aunilo.uum.edu.my/Find/Record/sg-ntu-dr.10356-20828` (2009)
47. Holzmann, G.J.: The logic of bugs. In: Symposium on Foundations of Software Engineering, SIGSOFT '02/FSE-10 (2002)
48. Hugues, L., Bredeche, N.: Simbad: an autonomous robot simulation package for education and research. In: International Conference on Simulation of Adaptive Behavior, pp. 831–842. Springer (2006)
49. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of mde in industry. In: ICSE, pp. 471–480 (2011). `http://doi.acm.org/10.1145/1985793.1985858`
50. Isla, D.: Handling complexity in the halo 2 ai. In: In Game Developers Conference (2005)
51. 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)
52. Jost, B., Ketterl, M., Budde, R., Leimbach, T.: Graphical Programming Environments for Educational Robots: Open Roberta - Yet Another One? In: 2014 IEEE International Symposium on Multimedia, pp. 381–386 (2014)
53. Juang, H.S., Lurrr, K.Y.: Design and control of a two-wheel self-balancing robot using the arduino microcontroller board. In: 2013 10th IEEE International Conference on Control and Automation (ICCA), pp. 634–639. IEEE (2013)
54. Ketterl, M., Leimbach, T., Budde, R.: Open Roberta (14), 1–22 (2015). URL `https://lab.open-roberta.org/`
55. Kleppe, A.G.: Software language engineering: creating domain-specific languages using metamodels. Addison-Wesley (2009)
56. Klotzbücher, M., Bruyninckx, H.: Coordinating robotic tasks and systems with rfsm statecharts (2012)
57. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566), vol. 3, pp. 2149–2154. IEEE (2004)
58. Kouzehgar, M., Tamilselvam, Y.K., Heredia, M.V., Elara, M.R.: Self-reconfigurable façade-cleaning robot equipped with deep-learning-based crack detection based on convolutional neural networks. Automation in Construction **108**, 102959 (2019)
59. Kramer, J., Scheutz, M.: Development environments for autonomous mobile robots: A survey. Autonomous Robots **22**(2), 101–132 (2007)

60. Krishna, B.S., Oviya, J., Gowri, S., Varshini, M.: Cloud robotics in industry using Raspberry Pi. In: 2016 Second International Conference on Science Technology Engineering and Management (ICONSTEM), pp. 543–547. IEEE (2016)
61. LEGO MINDSTORMS EV3: https://www.lego.com/en-us/mindstorms/downloads/download-software (2019)
62. Lignos, C., Raman, V., Finucane, C., Marcus, M., Kress-Gazit, H.: Provably correct reactive control from natural language. Autonomous Robots **38**(1), 89–105 (2015)
63. Magnenat, S., Rétornaz, P., Bonani, M., Longchamp, V., Mondada, F.: ASEBA: A modular architecture for event-based control of complex robots. IEEE/ASME Transactions on Mechatronics **16**(2), 321–329 (2011)
64. Matamoros, M., Rascon, C., Hart, J., Holz, D., van Beek, L.: Robocup@home 2018: Rules and regulations. `http://www.robocupathome.org/rules/2018_rulebook.pdf` (2018)
65. Menghi, C., García, S., Pelliccione, P., Tumova, J.: Multi-robot LTL planning under uncertainty. In: International Symposium on Formal Methods, pp. 399–417. Springer (2018)
66. Menghi, C., Tsigkanos, C., Berger, T., Pelliccione, P.: PsAlM: Specification of dependable robotic missions. In: International Conference on Software Engineering (ICSE): Companion Proceeedings (2019)
67. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. IEEE Transactions on Software Engineering, To appear. (2019)
68. Michel, O.: Cyberbotics Ltd. Webots™: professional mobile robot simulation. International Journal of Advanced Robotic Systems **1**(1), 5 (2004)
69. Miskam, M.A., Shamsuddin, S., Yussof, H., Omar, A.R., Muda, M.Z.: Programming platform for NAO robot in cognitive interaction applications. In: 2014 IEEE International Symposium on Robotics and Manufacturing Automation (ROMA), pp. 141–146. IEEE (2014)
70. 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)
71. Ohkawa, T., Uetake, D., Yokota, T., Ootsu, K., Baba, T.: Reconfigurable and hardwired orb engine on FPGA by java-to-hdl synthesizer for realtime application. ACM SIGARCH Computer Architecture News **41**(5), 77–82 (2014)
72. PICAXE: http://www.picaxe.com/software (2019)
73. Piltan, F., Sulaiman, N., Marhaban, M., Nowzary, A., Tohidian, M.: Design of FPGA based sliding mode controller for robot manipulator. International Journal of Robotic and Automation **2**(3), 183–204 (2011)
74. Piltan, F., Yarmahmoudi, M.H., Shamsodini, M., Mazlomian, E., Hosainpour, A.: PUMA-560 robot manipulator position computed torque control methods using Matlab/Simulink and their integration into graduate nonlinear control and Matlab courses. International Journal of Robotics and Automation **3**(3), 167–191 (2012)
75. Pot, E., Monceaux, J., Gelin, R., Maisonnier, B.: Choregraphe: a graphical tool for humanoid robot programming. In: RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication, pp. 46–51 (2009)
76. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. ICRA workshop on open source software **3**(3.2), 5 (2009)
77. Robert W. Button, John Kamp, Thomas B. Curtin, J.D.: A Survey of Missions for Unmanned Undersea Vehicles (2010)
78. Robot Mesh: http://docs.robotmesh.com/ide-project-page (2019)
79. ROBOTC: ROBOTC's Graphical feature (2019). URL `http://www.robotc.net/graphical/`
80. Röfer, T.: CABSL – C-Based agent behavior specification language. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **11175 LNAI**, 135–142 (2018). DOI 10.1007/978-3-030-00308-1_11
81. Salcedo, S.L., Idrobo, A.M.O.: New tools and methodologies for programming languages learning using the SCRIBBLER robot and Alice. Proceedings - Frontiers in Education Conference, FIE pp. 1–6 (2011)

82. Schauss, S., Lämmel, R., Härtel, J., Heinz, M., Klein, K., Härtel, L., Berger, T.: A Chrestomathy of DSL Implementations. In: 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE) (2017)
83. Schwartz, B., Nägele, L., Angerer, A., MacDonald, B.A.: Towards a graphical language for quadrotor missions. arXiv preprint arXiv:1412.1961 (2014)
84. Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5), 19–25 (2003). http://csdl.computer.org/comp/mags/so/2003/05/s5019abs.htm
85. SPARC: Robotics 2020 Multi-Annual Roadmap. https://eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf (2016)
86. Stahl, T., Völter, M.: Model-Driven Software Development. Wiley (2005)
87. Stampfer, D., Lotz, A., Lutz, M., Schlegel, C.: The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software. Journal of Software Engineering for Robotics (JOSER) **7**, 3–19 (2016)
88. Tamre, M., Hudjakov, R., Shvarts, D., Polder, A., Hiiemaa, M., Juurma, M.: Implementation of integrated wireless network and MatLab system to control autonomous mobile robot. International Journal of Innovative Technology and Interdisciplinary Sciences **1**(1), 18–25 (2018)
89. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: A new skill based robot programming language using UML/P Statecharts. In: 2013 IEEE International Conference on Robotics and Automation, pp. 461–466. IEEE (2013)
90. Thymio: https://www.thymio.org/en:start (2019)
91. Ulam, P., Endo, Y., Wagner, A., Arkin, R.: Integrated mission specification and task allocation for robot teams - Design and implementation. In: Proceedings - IEEE International Conference on Robotics and Automation, pp. 4428–4435 (2007)
92. VEX Robotics: https://www.vexrobotics.com (2019)
93. Voelter, M.: DSL Engineering. Designing, implementing and using domain specific languages (2013). URL http://www.dslbook.org/
94. Wang, F.Y., Kyriakopoulos, K.J., Tsolkas, A., Saridis, G.N.: A Petri-net coordination model for an intelligent mobile robot. IEEE Transactions on Systems, Man, and Cybernetics **21**(4), 777–789 (1991)
95. 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. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, pp. 366:1–366:12. ACM, New York, NY, USA (2018)
96. Wolff, E.M., Topcu, U., Murray, R.M.: Automaton-guided controller synthesis for nonlinear systems with temporal logic. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 4332–4339. IEEE (2013)
97. Ziparo, V.A., Iocchi, L., Nardi, D., Palamara, P.F., Costelha, H.: Petri net plans: a formal model for representation and execution of multi-robot plans. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1, pp. 79–86. International Foundation for Autonomous Agents and Multiagent Systems (2008)