

Open Research Online

The Open University's repository of research publications and other research outputs

EUD-MARS: End-User Development of Model-Driven Adaptive Robotics Software Systems

Journal Item

How to cite:

Akiki, Pierre; Akiki, Paul; Bandara, Arosha and Yu, Yijun (2020). EUD-MARS: End-User Development of Model-Driven Adaptive Robotics Software Systems. Science of Computer Programming, 200, article no. 102534.

For guidance on citations see [FAQs](#).

© 2020 Elsevier



<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Version: Accepted Manuscript

Link(s) to article on publisher's website:

<http://dx.doi.org/doi:10.1016/j.scico.2020.102534>

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data [policy](#) on reuse of materials please consult the policies page.

oro.open.ac.uk

EUD-MARS: End-User Development of Model-Driven Adaptive Robotics Software Systems

Pierre A. Akiki ^{a*}, Paul A. Akiki ^b, Arosha K. Bandara ^b, and Yijun Yu ^b

^a Department of Computer Science, Notre Dame University–Louaize, Zouk Mosbeh, Lebanon

^b School of Computing and Communications, The Open University, Milton Keynes, United Kingdom

ABSTRACT

Empowering end-users to program robots is becoming more significant. Introducing software engineering principles into end-user programming could improve the quality of the developed software applications. For example, model-driven development improves technology independence and adaptive systems act upon changes in their context of use. However, end-users need to apply such principles in a non-daunting manner and without incurring a steep learning curve. This paper presents EUD-MARS that aims to provide end-users with a simple approach for developing model-driven adaptive robotics software. End-users include people like hobbyists and students who are not professional programmers but are interested in programming robots. EUD-MARS supports robots like hobby drones and educational humanoids that are available for end-users. It offers a tool for software developers and another one for end-users. We evaluated EUD-MARS from three perspectives. First, we used EUD-MARS to program different types of robots and assessed its visual programming language against existing design principles. Second, we asked software developers to use EUD-MARS to configure robots and obtained their feedback on strengths and points for improvement. Third, we observed how end-users explain and develop EUD-MARS programs, and obtained their feedback mainly on understandability, ease of programming, and desirability. These evaluations yielded positive indications of EUD-MARS.

KEYWORDS

End-user development, Model-driven, Adaptive, Robots, Visual languages, Visual development environments

1. Introduction

The role of end-users in software development is becoming increasingly important. End-users are gaining access to a wide variety of devices that they can program. Robots are gaining wider adoption and can operate as Internet-of-Things (IoT) objects (things), because of their ability to sense their environment, perform computations, and exchange data over a network. End-users have a primary role to play in controlling robots, whether for personal or professional use. For example, drones are serving a variety of applications like aerial filming and entertainment [1]. End-users have already been engaged in various types of programming activities in areas such as business automation [2], IoT [3], and spreadsheets [4]. The adoption of software engineering principles in end-user software-development improves the quality of software products [5]. Applying model-driven development makes software systems resilient to changes in technology and requirements. Supporting adaptation allows the developed systems to act dynamically upon changes that occur in their context of use (user, platform, and environment). However, in order for end-users to apply these principles, they require an approach that is not daunting and does not incur a steep learning curve. This is the role of End-User Development of Model-driven Adaptive Robotics Software (EUD-MARS), which is the approach that we present in this paper.

1.1. What is EUD-MARS?

EUD-MARS is an approach that empowers end-users to apply model-driven principles to the development of adaptive robotics software systems. This approach does not require end-users to have advanced technical skills that would incur a steep learning curve.

Model-driven software-development approaches promote the use of multiple levels of abstraction, which enable an abstract model to have multiple concrete implementations in different technologies. For example, an abstract model can represent a program that controls various robots. This model would map to several concrete models that are compatible with different hardware technologies and application programming interfaces (APIs). This is possible as long as the target ro-

* Corresponding author.

E-mail addresses: pakiki@ndu.edu.lb (Pierre A. Akiki), paul.akiki@open.ac.uk (Paul A. Akiki), arosha.bandara@open.ac.uk (A. K. Bandara), yijun.yu@open.ac.uk (Y. Yu)

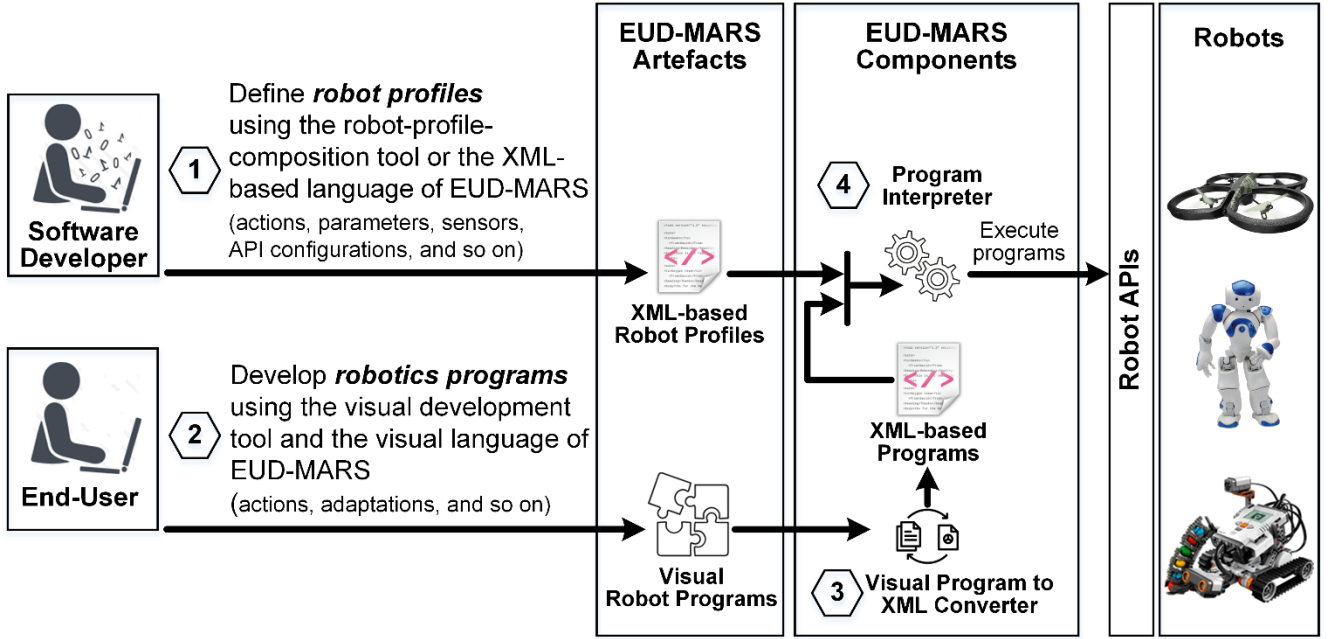


Fig. 1. Overview of the proposed EUD-MARS approach.

bots support the same types of actions. For example, end-users can execute the same program on two drones that have similar features, even if these drones are from different manufacturers. EUD-MARS hides the technicalities of model-driven development from end-users in an underlying layer, which professional software developers can control. To avoid possible confusion, we should note that the term model-driven in this paper refers to software models not hardware system models. EUD-MARS targets software systems that end-users develop to control robots, but it does not target the design of robot hardware.

Adaptive systems alter their behavior based on changes in their context of use. The development of adaptive systems can be a challenging task, even for professional software developers. EUD-MARS provides visual blocks that enable end-users, who do not possess advanced technical skills, to incorporate adaptive behavior into their programs. End-users can use events in their EUD-MARS programs to include change notifications, which trigger based on occurrences such as value changes. Robot sensors are the main detectors of changes. Events provide end-users access to the output of robot sensors but can also notify of changes from other sources such as controllers (e.g., key pressed). After detecting changes with events, end-users can adapt the behavior of one or more robots by invoking or restricting desired actions. The detection of changes occurs in real-time. This enables robots to adapt to new situations related to context-of-use aspects (user, platform, and environment). The following are few examples of such situations: a different type of user taking control (user), malfunction in one of the robot's parts (platform), low battery level on a robot (platform), and obstacles in a robot's surroundings (environment). For example, a robot could block some of its features from certain users based on their access rights. A robot could hand its task over to another robot if one of its parts is malfunctioning or if its battery level is low. When a robot faces an obstacle, such as an object or another robot, it could stop moving to avoid a collision.

EUD-MARS follows the approach shown in Fig. 1. (1) Professional software developers start the process by defining robot profiles that embody EUD-MARS concepts. Software developers can define these profiles using either an XML-based language or a visual tool that produces an output in this language. Robot profiles contain technical descriptions of the robots that the end-users are expecting to program. These profiles serve as a foundation that enables end-users to program robots without having to use complicated technical artifacts such as XML and code. (2) With EUD-MARS, end-users program robots via a visual programming language that we created using Blockly [6]. (3) When an end-user executes a visual program, a converter transforms this program into an XML-based representation. The same converter transforms the visual programs that use different robot profiles to XML. (4) Then, an interpreter reads the XML-based program (from the converter) and the XML-based representation of the robot profiles, and it dynamically (at runtime) executes the program's instructions on the robots.

Dividing the work between the converter and the interpreter creates a separation of concerns. The converter performs a transformation between a program's Blockly-based visual representation and the XML-based representation of EUD-MARS that the interpreter interprets and executes. Since the converter and the interpreter work dynamically, EUD-MARS can reflect program changes such as adaptation constraints on the fly without having to generate code that requires com-

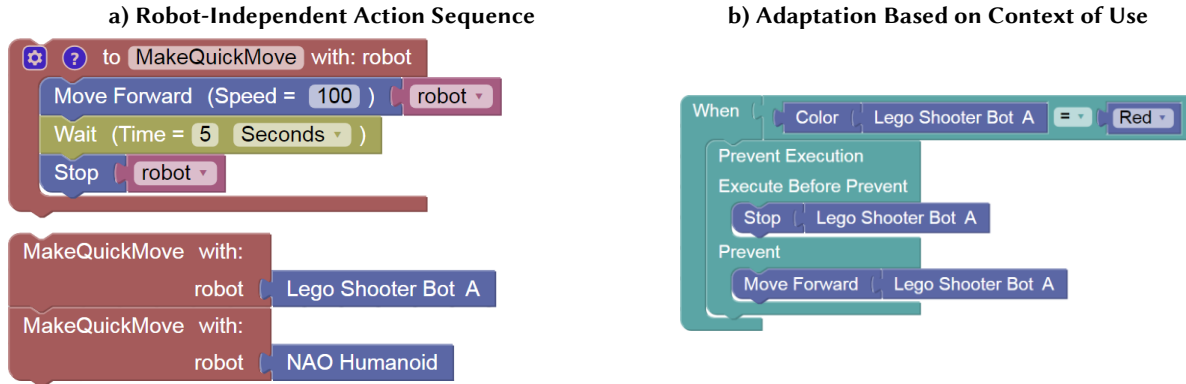


Fig. 2. A simple example of an EUD-MARS visual program containing an (a) action sequence and an (b) adaptation. The action sequence is robot-independent in the sense that it is not coupled with one type or brand of a robot but can be executed on different robots that support a common set of actions (in this example “move forward” and “stop”). The adaptation stops a robot and prevents it from moving forward if the robot’s color sensor detects a red color.

pilation. The interpreter uses robot APIs to execute actions on robots. Software developers can obtain an API from a robot’s manufacturer or develop it themselves. EUD-MARS programs can control multiple robots, which are working either individually or collaboratively. We use the word “robot” in this paper to refer to a variety of robots such as wheeled robots (e.g., rovers), legged robots (e.g., humanoids), and flying robots (e.g., drones).

1.2. EUD-MARS example

The example EUD-MARS visual program in Fig. 2 shows a sequence of actions that can be executed on multiple robots (Fig. 2a), and an adaptation of a robot’s behavior based on a change in its context of use (Fig. 2b).

The action sequence “MakeQuickMove” (Fig. 2a) calls the actions “move forward” and “stop” and a robot-independent time delay utility function. The latter actions apply to different types of robots. Hence, it is possible to execute this action sequence on similar robots developed using different hardware. These robots can be, for example, two rovers developed with different robotics kits like Lego Mindstorms and VEX. It is also possible to execute this action sequence on different robots that are partially similar. For example, both rovers and drones can move forward and stop. In this example, we can see that the program is executing the same “MakeQuickMove” action sequence twice, once with a Lego Mindstorms shooter robot and another time with a NAO [7] humanoid robot. We should emphasize that the action sequence depends on the capabilities of the target robot(s). In this case, if a robot does not support moving forward and stopping, then it would not be possible to execute the “MakeQuickMove” action sequence on it.

The adaptation (Fig. 2b) in this example activates upon the detection of red color. The “When” event and its condition represent this detection. When the robot “Lego Shooter Bot A” detects red color (through its color sensor), the program stops this robot and then prevents it from moving forward. The “Prevent Execution” block performs the prevention. Under this block, the end-user programmers specify the action that they wish to *execute before prevention* (if any), e.g., “stop”, and the action that they wish to *prevent* from executing, e.g., “move forward”. One case for applying this particular adaptation involves preventing a robot from damaging itself by colliding into certain types of obstacles.

1.3. Contributions of EUD-MARS

There are existing approaches that aim to enable end-users to visually program robots. Some approaches aim to support professional programmers in the development of model-driven and adaptive robotics software. EUD-MARS generally differs from existing approaches in its aim to enable end-users, who are not professional programmers, to visually program model-driven adaptive robotics software. EUD-MARS makes the following contributions:

1. Concepts that integrate visual programming with model-driven software development to make the latter accessible for end-users, thereby enabling them to benefit from this useful software engineering technique;
2. Concepts that integrate visual programming with adaptive software development, and enable end-users to develop adaptive robotics software in a simplified manner;
3. An approach that divides the work required to develop robotics software between professional software developers and end-user programmers; this way, software developers would prepare the technical underpinnings through extensible robot profiles and API configurations that lay the foundation for end-users to develop model-driven adaptive robotics software systems for various types of robots without having to deal with complex technical details;

4. A tool to support software developers in the definition and maintenance of robot profiles and configurations;
5. A tool to support end-user programmers in the development of robotics programs using the concepts and visual language offered by EUD-MARS.

We evaluated EUD-MARS from different perspectives as follows:

- We conducted a technical evaluation by using EUD-MARS to program different types of robots, and by assessing the paradigm (interlocking blocks) of its visual programming language against the recommendations of the cognitive dimensions framework [101].
- We asked software developers to prepare robot profiles and API configurations using EUD-MARS and to provide feedback on strengths and points for improvement concerning the overall approach and its supporting tool for software developers.
- We asked end-users to explain and develop EUD-MARS programs, and to provide feedback mainly on understandability, ease-of-programming, and desirability. End-users found the visual programs to be overall understandable and easy to program, and they perceived the approach to be desirable.

Google developed Blockly as a JavaScript library for creating block-based visual programming languages. We should note that although EUD-MARS uses Blockly for representing its visual programming language, it does not rely on any technical concepts from Blockly. EUD-MARS merely uses Blockly as a way of visualizing its concepts (contributions 1 and 2). Even the program interpreter and the component that converts visual programs to XML (refer to Fig. 1) are EUD-MARS components that do not rely on Blockly. Furthermore, in this paper, we do not aim to develop new robot APIs. Hence, we used existing robot APIs for evaluating the contributions of EUD-MARS.

We should note that EUD-MARS does not aim to replace traditional code-based (text-based non-visual) programming languages that professional programmers use for developing robotics systems. Code-based languages have many merits and offer numerous advanced features such as object-orientation and complex data structures. EUD-MARS does not aim to offer all these advanced features; for example, its visual language is not fully object-oriented. The main aim of EUD-MARS is to provide a simple way for end-users to develop model-driven adaptive robotics software systems without going through the technical complications that require the expertise of professional programmers. These end-users can be anyone who is not a professional programmer but is interested in programming robots. One example can be hobbyists of any age who wish to program robots for entertainment purposes. Another example can be children who are programming robots as part of an educational workshop. Since EUD-MARS primarily targets end-users, it supports robots that are usually available for end-users. For example, these robots include hobby drones like Parrot Bebop, educational humanoids like NAO, and robots that hobbyists can put together using a kit like Lego Mindstorms. EUD-MARS does not currently target complex robots, e.g., those used for industrial purposes, which professional programmers typically program using code-based languages. In some cases, end-users can control certain robots through preprogrammed tools or physical controllers. Nonetheless, there are many cases where end-users would like to program their robots to perform custom operations. This can be for fun, education, or some other real-life benefit. End-users can realize various applications by programming robots visually. For example, young school students can program a prebuilt humanoid robot like NAO to perform dance moves or kick a ball. Older school students can put together a rover robot using a kit like Lego Mindstorms and program it to drive within a closed area to identify objects of a certain color. Drone hobbyists can develop a custom program that makes their drones fly in a certain pattern. In either case, end-users could decide to substitute their robot's hardware and would benefit from the model-driven nature of EUD-MARS. They would also encounter scenarios that require adaptation and would benefit from the adaptation features that EUD-MARS supports.

1.4. Structure of the article

Section 2 presents the strengths and shortcomings of the related work. Section 3 presents the concepts behind EUD-MARS as UML class diagrams and explains each one. These concepts realize contributions 1, 2, and 3 stated in Section 1.3. Section 4 demonstrates an example EUD-MARS program using visual blocks and XML. This example elaborates further on the concepts presented in Section 3. Section 5 presents two tools that support programmers and end-users who wish to use EUD-MARS. These tools realize contributions 4 and 5 stated in Section 1.3. We present the results of the technical, software-developer, and end-user evaluations in Sections 6, 7, and 8 respectively. Section 9 presents our conclusions and future work.

2. Related work

The work presented in this paper spans the following areas: visual programming, end-user programming, robot programming, and model-driven and adaptive robotics software systems. A variety of related work has focused on one or more of these areas. There are visual programming tools and techniques for end-users. Some of these tools and techniques are general, while others are robotics-specific. Some techniques apply model-driven development to robotics software. There are reference architectures and approaches for developing various types of adaptive software systems including

robotics software. There are middleware systems that aim to make it easier for professional software developers to program robots. Although the work presented in this paper spans various areas, it focuses on the intersection between model-driven and adaptive robotics software development, and end-user software development using visual programming languages. This section presents an overview of the strengths and shortcomings of the related work and compares it to EUD-MARS. There are existing surveys of literature that offer more information on the following areas of related work:

- *end-user software development* [8,9],
- *robot programming systems and middleware* [10–13],
- *model-driven development* [14,15], and
- *adaptive software systems* [16–18].

2.1. Visual end-user programming approaches

Several existing visual programming approaches target end-users. Some of these approaches are general and not directly related to robots. For example, *TouchDevelop* enables end-users to develop software applications visually by using smartphones [19]. *Alice* aims to simplify early programming courses [20]. *MashSheet* [21] and *Vegemite* [22] target the development of mashups. Other approaches are directly related to the end-user development of robotics software. What follows is an overview of the approaches related to robotics software development.

Microsoft Robotics Developer Studio (MRDS) supports the visual development of robotics software, using the Microsoft Visual Programming Language (MVPL) [23]. This language provides visual programming blocks that programmers can tie to each other using line connections. MRDS offers an advanced 3D simulator, which has a physics engine and allows the placement of multiple robots in a realistic simulation environment. Both hobbyists and professional software developers can program robots with MRDS. Hobbyists can perform visual programming using a drag and drop interface, and professional software developers can write code using Visual Studio.

EV3 is a programming tool that supports the development of robotics software for controlling robots constructed using the Lego Mindstorms robotics kit [24]. Lego primarily intended the Mindstorms kit to be an educational tool, but both professionals and hobbyists can use it. This tool has a visual programming paradigm that incorporates configurable programming blocks. Programmers place these blocks next to each other in the intended order of execution. EV3 compiles the visual programs and deploys them to Lego Mindstorms robots. Users can execute a program from a screen located on the brick (contains the processor) of their robot.

End-users such as children and teenagers use *Scratch* to learn computer programming and create real-world applications [25]. Scratch's environment offers visual building blocks that fit together to compose a program. Scratch mostly supports working with on-screen objects, e.g., animated characters (sprites). Nonetheless, third-party extensions allow Scratch to support the programming of physical robots such as Edbot [26], Lego Mindstorms robots, and drones.

RoboBlockly [27] is a web-based programming tool for learning programming by developing software applications that control robots. Teachers also use RoboBlockly to teach mathematics and science. This tool uses the Blockly library as part of its visual programming editor. RoboBlockly generates code in C++ that controls robots created using different types of hardware such as Linkbot and Lego Mindstorms. Other researchers have developed another Blockly-based programming language with a similar name called *Robot Blockly* [28]. The latter targets the development of programs for controlling industrial robots, particularly a single-armed robot called Roberta. *MORPHA* also targets visual programming for industrial robots, and it presents a style guide of icons and symbols for visual robotics programs [29].

Some tools support the development of IoT and robotics software systems that run on general-purpose hardware platforms such as Arduino and Raspberry Pi. It is possible to use these tools for programming robots that roboticists built from scratch and for extending existing robots such as drones with new features. Examples of Arduino-specific approaches include: *Ardublock* [30], *Modkit* [31], and *Sense* [32]. *Node-RED* offers a web-based tool for programming devices and services using visual blocks [33]. *NETLab Toolkit* supports the development of IoT applications for different hardware platforms including Arduino and Raspberry Pi [34]. Both Node-RED and NETLab Toolkit provide visual programming languages that use a box-and-line notation.

Code3 is a system that supports the visual programming of mobile manipulator robots [35]. This system targets both non-roboticists and roboticists, and it aims to reduce the time (weeks) that is usually required to learn traditional robotics programming systems. Code3 integrates CustomLandmarks, CustomActions, and CodeIt, which are three components that its creators deem necessary for programming mobile manipulation tasks. CustomLandmarks allow users to create landmarks that a robot can locate. CustomActions use programming by demonstration to make the manipulation of simple actions more accessible to novice users. CustomActions in Code3 build upon the work of Alexandrova et al. [36]. CodeIt extends the work of Huang et al. [37] to help users in specifying logic and complex task structures that are not possible to accomplish with CustomActions.

Several approaches support programming the NAO robot. SoftBank (previously Aldebaran) Robotics, the creators of NAO, developed a tool called *Choregraphe* for programming the NAO robot using a box-and-line visual notation and Python [38]. *Interaction Blocks* is a tool for developing robotics programs using visual blocks that serve as interaction design

patterns [39]. *TiViPE* supports the development of robotics programs using a network of connected parametrized visual components [40,41].

Stenmark et al. [42] presented a graphical interface for performing iconic programming of the ABB YuMi robot. The authors proposed combining programming by demonstration and parametrized skill representations. With this approach, users can select actions and store them as skills that they can reuse later.

ROBOTC [43] is a programming language that mainly supports educational robotics technologies such as VEX and Lego Mindstorms. Expert and novice programmers can program robots with ROBOTC using either code or a visual language. Programmers can display the output of their ROBOTC programs in a virtual simulation environment.

Some robot programming systems adopted flow-based visual programming languages [44]. *RoboFlow* enables end-users to control robots by developing flowchart-like programs composed of visual tokens [45]. An evaluation study showed that people with various backgrounds could quickly learn RoboFlow and use it to program robots effectively with a low error rate. *Interaction Composer* aims to enable the collaboration of programmers and end-users in designing human-robot interaction [46,47]. *Interaction Composer* supports the development of programs that follow a box-and-line notation that is similar to the one used by *EV3*.

The approaches discussed in this section have many strengths. Both end-users and professional software developers can use these approaches to program robots. These approaches support visual languages, which enable end-users to program robots without incurring a steep learning curve. This is especially true in comparison to code-based programming languages that target professional programmers. Some approaches do not directly target robotics programming but are extensible in this direction. These approaches could be more difficult to use than the ones that provide robotics-related functionality out of the box. We can see that the robot programming approaches discussed in this section are not model-driven. End-users would have to redevelop a program that they intend to execute on different robots. For example, assume that an end-user bought a NAO robot and built another humanoid robot using Lego Mindstorms. As humanoids, both robots share some abilities such as walking and talking. However, with traditional approaches, running the same set of instructions on both robots requires the end-user to develop two separate program versions. For example, end-users could develop the NAO program version with *Choregraphe* and the Mindstorms program version with *EV3*. By following a model-driven approach, an EUD-MARS program can run on different robots that support a common set of actions. EUD-MARS also enables software developers to integrate a variety of code-based robot APIs and map them to their visual block counterparts for end-users. This integration is done while maintaining the model-driven nature of EUD-MARS. Furthermore, unlike the existing approaches, EUD-MARS provides dedicated visual programming blocks for adaptation. Developing complex adaptations from scratch could require specifying many instructions to elicit and process context-of-use changes and adapt a robot's behavior accordingly. With EUD-MARS, end-users can simply use visual blocks out of the box to handle events and specify what actions they want to invoke when an event triggers.

2.2. Model-driven development approaches for robotics software

The Object Management Group (OMG) Robotics Domain Task Force [49] recommends extending OMG technologies to support the domain of robotics. For example, it is possible to use the model-driven architecture (MDA) as a reference for developing robotics software systems. Several approaches apply model-driven development in robotics and primarily target technical stakeholders. We provide an overview of these approaches; more information is available in an existing survey of literature [48].

Blanc et al. [50] applied model-driven principles to develop software for Sony's AIBO pet robot. This work represents an early attempt to incorporate models into robotics software development. It uses models to represent the characteristics, states, and behaviors of robots. The researchers developed prototypes to test their approach.

BRICS (Best Practice in Robotics) supports the generation of code from models via model-to-text transformations [51]. This saves software developers the time needed to write code manually and allows them to structure their designs. *BRICS* implements model transformations using *Epsilon* [52]. *BRICS* represents its software models using what its creators call "specific models". These models conform to an abstract meta-model that in turn conforms to the "Ecore meta-meta-model" [53]. *BRICS* separates concerns by differentiating five concerns (5Cs): computation, communication, coordination, configuration, and composition. Computation represents a system's core functionality (i.e., implementation of the domain knowledge). Communication provides computation components with data while taking into consideration quality-of-service properties like latency. Coordination determines how components work together. Configuration parameters influence behavior and performance. Composition couples components, while considering reuse and predictability of behavior.

RobotML is a domain-specific language (DSL) for specifying missions, environments, and robot behaviors [54]. This language aims to support the following qualities: ease of use, architecture-style neutrality, multiple platforms, and platform independence. *RobotML* supports platform-independent definitions of a system's internal structure (architecture), the ways components send and receive data, and the robotic behavior. It enriches its domain model with an ontology that represents the knowledge of robotics experts. *RobotML* comprises four packages: architecture, communications, behavior, and deployment. The architecture package defines the concepts used to develop a robotics system. These include concepts

that represent a system's environment, data types that represent the data exchanged among components, and concepts related to a robot's mission. The communication package represents data exchange and service calls through ports and connectors. The behavior package comprises concepts related to the evolution model that is defined using algorithms and finite state machines. The deployment package comprises concepts related to the target platform such as middleware or simulator. RobotML uses existing code generators to support various robotics middleware and simulation engines.

The *SmartSoft* project supports the development and integration of model-driven robotics software components [55,56]. *SmartSoft*'s development approach involves the levels of abstraction proposed by MDA. A platform-independent model (PIM) describes the system. A transformation converts the PIM into a platform-specific model (PSM), which is refined to fit the selected platform. Another transformation converts the PSM to a platform-specific code-implementation. The toolchain of *SmartSoft* offers the ability to define transformations between one level of abstraction and another.

V³CMM (3-View Component Meta-Model) provides platform independence by separating reusable platform-independent parts from their platform-dependent counterparts [57]. *V³CMM* conforms to a component meta-model with three views: structural, coordination, and algorithmic. The structural view comprises elements such as components, ports, and interfaces. State machines link this view to the coordination view. *V³CMM* represents its coordination and algorithmic views using modified versions of UML state machines and activity diagrams. The coordination view incorporates UML state-machine concepts such as state and transition. The algorithmic view enables developers to model and connect activities. Example activities include calling operations from other components and external libraries. *V³CMM* uses ATL [58] and JET [59] for model-to-model and model-to-ADA-code transformations respectively.

RobMoSys manages the interfaces between roles (participants) in its ecosystem [60]. These roles include, for example, component supplier, system builder, behavior developer, performance designer, and system architect. *RobMoSys* aims to manage the complexity in robotics system development, by separating the concerns among multiple roles. *RobMoSys* relies on model-driven engineering and offers supporting tools. For example, it offers visual design tools for models that represent concepts such as components and tasks. The tools of *RobMoSys* offer a workflow that allows people with different roles to provide their input at an adequate abstraction level.

The benefits of model-driven development are apparent in several areas such as testing [61,62] and user interfaces [63,64]. The use of model-driven development enabled the previously discussed approaches to offer technology independence concerning hardware platforms, programming languages, and middleware systems. Despite their benefits, these approaches primarily target professional software developers who have skills with traditional programming, modeling, and transformation languages. It is unlikely that end-users would be capable or even willing to learn such software development technologies since they would face many challenges and a very steep learning curve. Hence, *EUD-MARS* aims to provide the underlying technical benefits of model-driven software development in a way that is suitable for end-users. This is why *EUD-MARS* divides the work between professional software developers and end-users. Software developers lay the technical groundwork that enables end-users to develop model-driven robotics software using a simple visual programming language.

Some approaches apply model-driven development to robotics software and involve a variety of stakeholders including those who do not possess major technical expertise. *LightRocks* is a DSL for robot programming that offers three levels of abstraction, which embody the knowledge of different stakeholders including domain experts and technicians [65]. *LightRocks* uses UML/P statecharts out of which it generates code-based programs that are executable against robots. *LightRocks* was evaluated with two test cases involving (1) driving a wood screw into a cube and (2) plugging electrical parts on a rail. *Adam et al.* [66] presented an infrastructure for developing service robotics applications (e.g., for hospitals). Like *LightRocks*, this infrastructure aims to support the input of different stakeholders such as domain experts and robotics experts. It aims to enable domain experts to contribute their knowledge using DSLs instead of general programming languages. *Adam et al.* tested their overall approach in a hospital environment.

Although *LightRocks* and *Adam et al.* use abstraction to incorporate the skills of different stakeholders, they primarily do not aim to support end-users the way *EUD-MARS* does. These approaches considered the input of domain experts rather than just technical experts, but *EUD-MARS* aims to empower end-user to become programmers. From this perspective, we can see that *LightRocks* relies on statecharts and *Adam et al.* rely on code-based DSLs that would be more technically challenging for end-users in comparison to the block-based visual language of *EUD-MARS*.

FLYAQ is a tool that enables non-technical end-users, such rescue workers, to specify missions for a team of multicopters [67–69]. It provides a DSL called Monitoring Mission Language (MML) that supports graphical mission definitions. *FLYAQ* calculates waypoints and trajectories from missions and represents them using a language called QBL.

Although *FLYAQ* targets non-technical end-users, it primarily focuses on programming multicopters, whereas *EUD-MARS* works with a variety of robots. This is apparent in *FLYAQ*'s graphical tool, which although hides technical details from non-experts, integrates with a mapping system (Open Street Map) due to the tool's focus on multicopters. Hence, *FLYAQ* assumes that there are trajectories that it will calculate from missions and concepts like no-fly zones. *EUD-MARS* does not make assumptions tied to a single type of robot. For example, an end-user might decide to use *EUD-MARS* for programming a robotic arm intended for hobbyists. In this case, the robot is fixed and has no trajectories or no-fly zones on a map. Another example can be programming a humanoid robot that does not fly. *Ciccozzi et al.* [70] have extended *FLYAQ* to support another type of robot, namely underwater vehicles, within a multi-robot system. An extension to part

of the DSL was required to support the new type of robot. On the other hand, EUD-MARS is extensible by design through robot profiles that allow it to support various robots not necessarily considered during its initial design. Hence, it is not necessary to go back and extend DSLs and tools with new concepts for each new type of robot. EUD-MARS automatically and dynamically reflects added robot profiles in its visual language and end-user programming tool. We used several robots to assess EUD-MARS for this paper; it is possible to add more robots in the same way. The ability to specify concepts like no-fly zones and operations like movements on a dedicated visual representation of an environment, e.g., like a map in FLYAQ, has its advantages. However, it would be better if this were not at the cost of generality in terms of supporting different types of robots. For example, in a commercial vehicle tracking system that some of us worked on more than a decade ago, we used Google Maps to enable end-users to visually manage concepts like restricted geographical zones, trajectories, and points of interest for vehicles. Yet, this vehicle-tracking system only targeted cars and trucks. The visual block-based language of EUD-MARS is more general and less restrictive in terms of supporting different types of robots. Currently, EUD-MARS provides a basic two-dimensional simulator. However, it is possible to extend this simulator in the future to provide a basis for supporting programming by demonstration. Then, end-users could specify parts of their programs in robot-specific simulators without losing the generality of the visual programming language. Yet, this feature requires the design of a generic extensible simulation environment, which is out of the scope of this paper.

2.3. Reference architectures for adaptive software systems and adaptation approaches for robotics

Researchers have proposed several reference architectures for developing adaptive software systems. These architectures can serve as a reference for developing robotics software systems that are a type of autonomic (self-managing) software system. *MAPE-K* is a reference model for autonomic computing, which considers software systems to be a set of managed resources [71]. *MAPE-K* supports knowledge sharing among four functions: monitor, analyze, plan, and execute. The *Rainbow* framework uses a control loop to manage self-adaptive systems, and it realizes the functions of the *MAPE-K* loop [72]. *Rainbow* comprises the following components: model manager, constraint evaluator, and adaptation engine. The *Three Layer Architecture* is an architectural approach and a conceptual reference for self-managing software systems; it comprises three layers: component control, change management, and goal management [73].

These reference architectures are useful for professional software developers, who aim to develop adaptive software systems related to robotics or other areas. They standardize the layering of adaptive systems and offer a high-level design of the components that are necessary for implementing these systems. However, reference architectures are too theoretical for end-users to follow, even if they have basic software development skills. Hence, as with model-driven development, EUD-MARS aims to allow end-users to program adaptive robot behavior while hiding the unnecessary complexity. We do not claim that EUD-MARS will allow end-users to develop adaptive software systems that are as advanced as the ones developed by professional programmers. Nonetheless, we aim to provide end-users with the ability to develop basic adaptive robot behavior using predefined building blocks that hide the theoretical suggestions of reference architectures and the complexity of code-based implementations.

Some robot behavior adaptation approaches built upon *FLYAQ* (refer to Section 2.2), which attempts to involve different stakeholders in the robotics development process. *Bozhinoski et al.* [74] presented an approach for runtime adaptation of unmanned aerial vehicle (UAV) systems, which comprise humans, drones, and devices like cameras and sensors. Their adaptation model operates on a group of autonomous entities (computational or human actors) that collaborate to accomplish tasks. The authors plan to integrate their approach with *FLYAQ*. *Dragule et al.* [75] also proposed a *FLYAQ* extension to support the specification of adaptive and resilient missions.

Like *FLYAQ*, the adaptation work based on it focuses on a particular type of robot, namely swarms of UAVs organized in different topologies. Additionally, *Bozhinoski et al.* assume that each entity within their system implements the *MAPE* loop based on a state machine diagram that they defined. These approaches have their merits concerning adapting behavior within a UAV system, but the purpose of EUD-MARS is different. Although some collaboration among robots is possible with EUD-MARS, the primary objective of this system is not about supporting robot swarms. Additionally, with EUD-MARS, there is no assumption that robots implement the *MAPE* loop. The types of adaptations that EUD-MARS supports involve the execution of robot-supported actions based on changes in the context of use. This requires monitoring (e.g., through sensors) and execution (e.g., restrict value), but EUD-MARS does not expect non-technical end-users to program complex adaptations for large-scale scenarios. Nonetheless, if a robot's API supported a *MAPE*-based or other preprogrammed adaptation algorithm, EUD-MARS can invoke it as an action. The simplicity of the adaptation operations supported by EUD-MARS offers the advantage of making them applicable out-of-the-box to different types of robots and understandable and usable by end-users. These operations act as small blocks in the style of IFTTT (If This Then That) [76], but targeted towards robots and usable within a visual block-based program. Adding complexity to the way end-users program adaptations could make things more difficult for them and preconceiving special-purpose preprogrammed adaptation mechanisms could decrease the generality of the adaptation blocks. The application of a reference architecture like the *MAPE-K* and the realization of automated adaptation based on predefined algorithms could be typical in large-scale

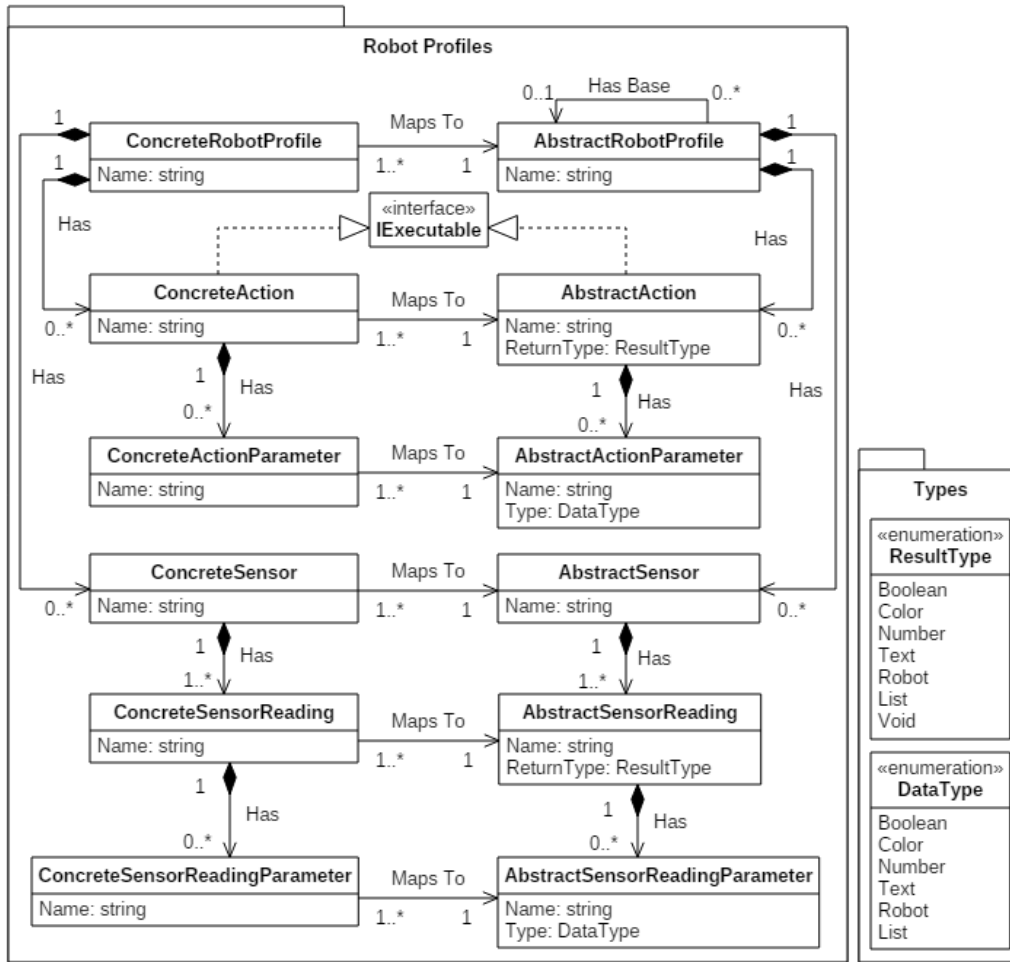


Fig. 3. Abstract and concrete robot profiles.

industrial systems. However, as we previously mentioned (refer to Section 1.3), EUD-MARS does not intend to replace these conventional approaches but rather to empower end-users with a simple enough approach.

2.4. Robotics middleware

The techniques for developing robotics software systems vary depending on the heterogeneity and diversity of the involved components. Robotics middleware systems are a type of software technology designed to manage complexity and heterogeneity in robotics. These systems aim to simplify software design and hide the complexity of low-level communication with robots by acting as an abstraction layer between operating systems and software applications. Robotics middleware systems generally follow a component-based development approach [77,78]. We present a brief overview of some of the existing robotics middleware and the systems that extend them. We primarily focus on the Robot Operating System (ROS) since it is likely the most widely used robotics middleware today.

ROS offers libraries and tools that support the development of robotics software using different programming languages (e.g., C++ and Python) [79,80]. It allows multiple development groups to collaborate and build upon each other's work. ROS programs communicate over a peer-to-peer network using the following types of communication components: topics, services, and actions. Topics support data streams between nodes. Nodes can publish and subscribe to topics using a central node and can communicate synchronously and asynchronously using services and actions respectively.

ROS offers a standardized way for developing robotics software. However, this does not necessarily provide hardware independence the way model-driven development does. Several researchers presented model-driven software development techniques that use ROS. *Hua et al.* [82] presented a model-driven approach that supports the development of software for controlling industrial robots by using model transformations to generate executable ROS code. *ReApp* is a ROS-based and model-based approach that attempts to address the problem of programming language diversity in robotics and makes the use of robots economically feasible for small and medium-sized enterprises (SMEs) [83]. *Bardaro et al.* [81] presented a robotics software development approach that separates the competencies of software engineers and robotics ex-

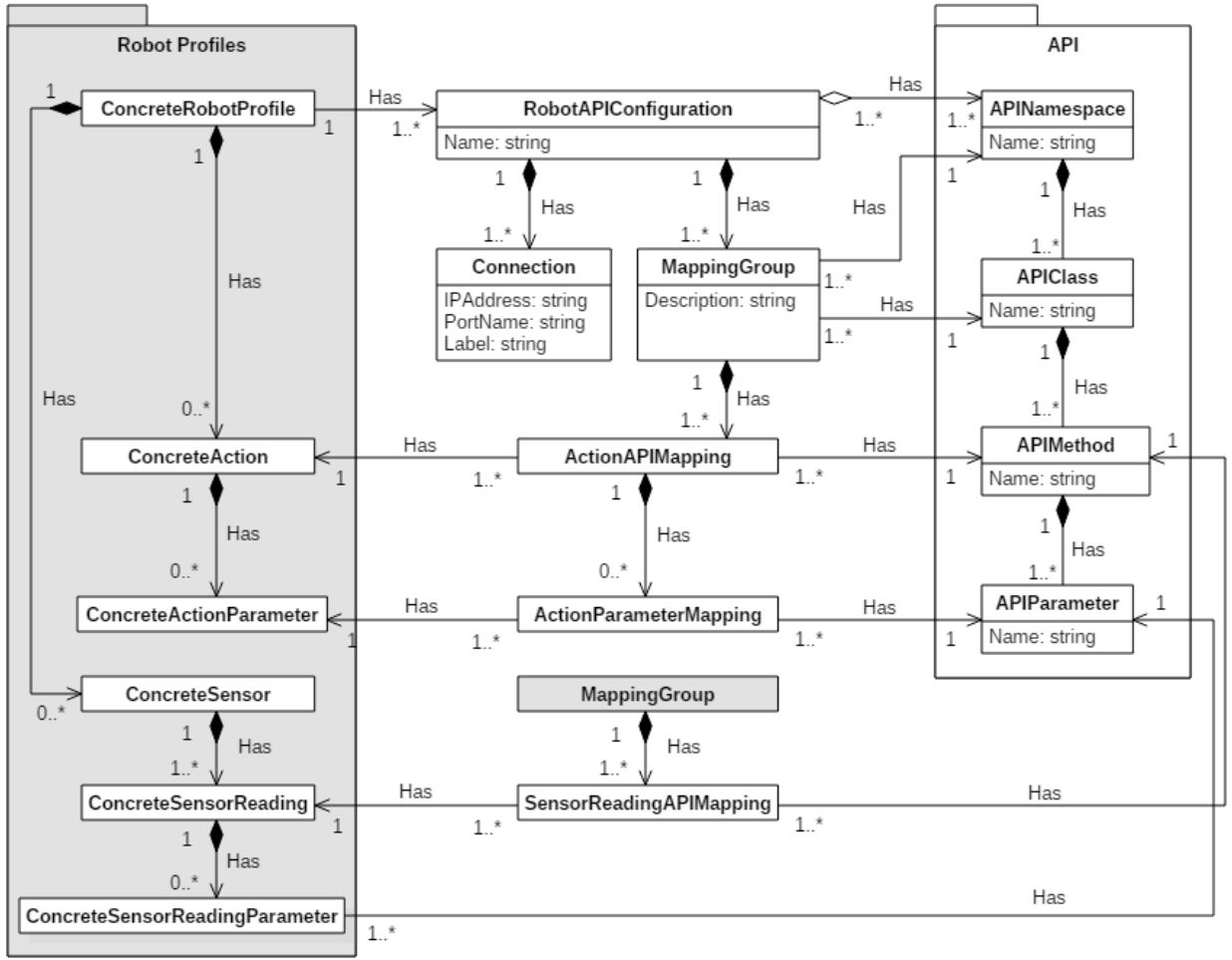


Fig. 4. Mapping between concrete robot profiles and robot APIs.

perts. EUD-MARS attempts to perform a separation of competencies that is similar to *Bardaro et al.* but between software developers and end-users. Unlike *Hua et al.*, EUD-MARS does not perform code generation from its models but rather interprets and executes them at runtime while taking into account constraints that adaptations impose.

Other existing middleware systems are similar to ROS in terms of their objectives but differ in terms of their capabilities and are less widely used. These systems primarily target professional software developers and are programmed using code-based languages. The Open Platform for Robotic Service (OPRoS) aims to facilitate the development of robotics software using off-the-shelf components [84,85]. Similarly, the Robot Technology (RT)-Middleware, *Miro* [86], and *Orocos* [87] support the development of component-based robotics software using the Common Object Request Broker Architecture (CORBA) [88,89]. Other examples of robotics middleware systems include *ORCA* [90] and *Player* [91].

The capabilities that ROS offers are certainly useful and currently adopted by many robotics software developers. However, ROS and the existing model-driven approaches that extend it do not directly support end-users through a visual programming language the way EUD-MARS does. Hence, ROS programmers typically use code-based programming languages that could be more daunting and introduce a higher learning curve for end-users. Furthermore, these approaches do not support adaptation operations out of the box in a way that is easily accessible to end-users. The same applies to the other middleware systems. As Fig. 1 shows, the underlying communication between EUD-MARS programs and robots goes through robot APIs. Software developers can build these APIs either from scratch or on top of one of the existing middleware systems such as ROS. EUD-MARS does not target this underlying low-level communication. Hence, it does not aim to be an alternative for existing middleware systems such as ROS. EUD-MARS rather aims to create a level of abstraction that allows programmers to set up the groundwork that enables end-users to develop model-driven adaptive robotics software in a simplified visual way. With model-driven development, it is possible to use alternative underlying implementations such as middleware. For example, *RobotML* uses code generators to support various robotics middleware (refer to Section 2.2). Instead of generating code, EUD-MARS interprets programs at runtime (refer to Fig. 1) and can execute them using different underlying implementations. In ROS, nodes use a central (Master) node to communicate. In

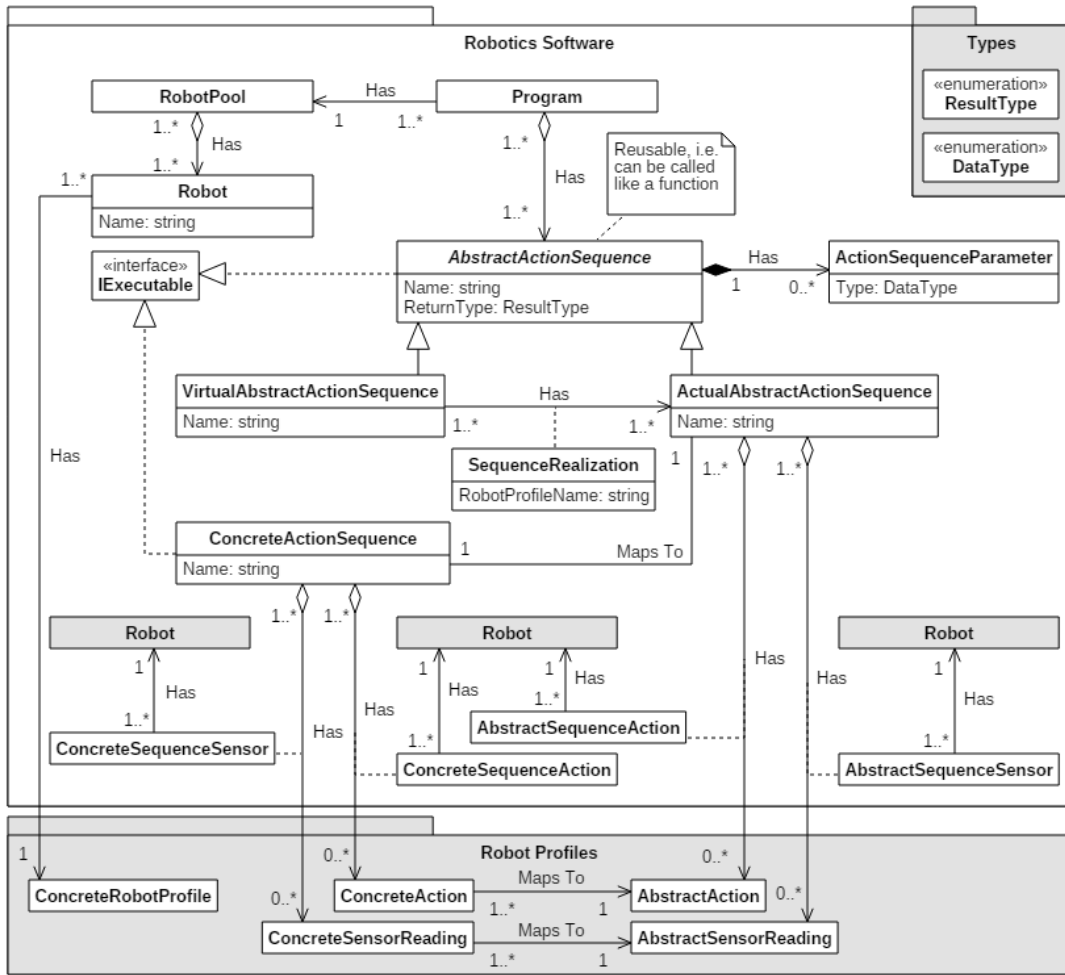


Fig. 5. Action sequences.

EUD-MARS, a program created by an end-user runs on a device (e.g., computer) and acts as a central node that can relay information among the robots. The next section presents the concepts of EUD-MARS and explains how these concepts support end-users in developing model-driven and adaptive robotics software systems.

3. EUD-MARS concepts

This section presents the technical concepts of EUD-MARS that include robot profiles, APIs, and the mappings between them. These concepts also include robotics program elements such as action sequences, events, variables, controls structures, utilities, and adaptations. We present EUD-MARS' concepts as UML class diagrams and explain them in the text. We repeated a few UML packages and classes, colored in gray, across figures and within the same figure to show associations more clearly and to avoid overlapping lines as much as possible.

3.1. Robot profiles and their mapping to APIs

Software developers prepare robot profiles and API configurations as a preliminary setup that enables end-users to program robots using the visual language of EUD-MARS. The class diagrams in Fig. 3 and Fig. 4 encompass the concepts of this preliminary setup.

As shown in Fig. 3, *RobotProfiles* describe the *Actions* and *Sensors* that the robots support. Examples of *Actions* include move-forward, stop, and speak. On the other hand, examples of *Sensors* include color, pressure, and proximity. An *Action* can have *Parameters*. For example, a "move forward" action requires the speed at which a robot will move. A *Sensor* provides *Readings* of data that it acquired from a robot's physical environment. One example *SensorReading* is a robot's proximity from physical objects. Like *Actions*, *SensorReadings* can also have parameters. Both *Actions* and *SensorReadings* have return types. The *DataType* and *ResultType* enumerations denote the parameter types and the return types respectively. We represent robot profiles on two levels of abstraction. *AbstractRobotProfiles* embody most of the information regarding

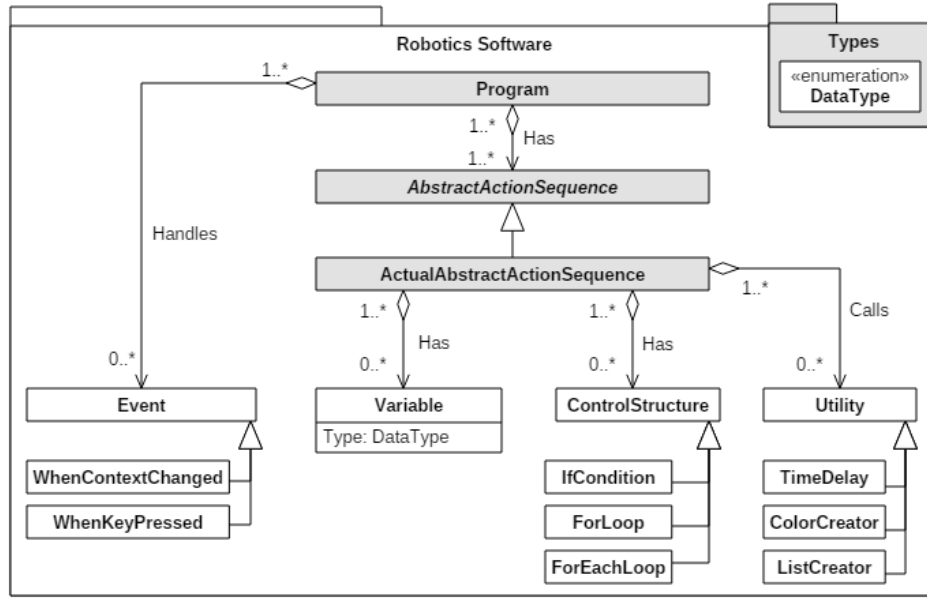


Fig. 6. Events, variables, control structures, and utilities.

a robot's *Actions* and *Sensors*. Every *AbstractRobotProfile* maps to one or more *ConcreteRobotProfiles*, which in turn map to APIs. For example, an *AbstractRobotProfile* called "Rover" can map to two *ConcreteRobotProfiles* representing Lego Mindstorms and VEX hardware implementations of a rover robot. *ConcreteSensorReadings* represent, in the robot profile, data reading operations performed by *Sensors*. Each *ConcreteSensorReading* maps to a method that realizes the reading operation in an API. When a *ConcreteRobotProfile* maps to an *AbstractRobotProfile*, it shall implement all its *Actions* and *Sensors*. Additionally, *ConcreteActions* and *ConcreteSensorReadings* have the same return type as their abstract counterparts. Having *ConcreteRobotProfiles* acquire the characteristics of an *AbstractRobotProfile* supports the principle of reuse. Additionally, either one of the concrete implementations can substitute the abstract concept. For example, both "Lego Mindstorms Rover" and "VEX Rover" can substitute "Rover". This is similar to the Liskov Substitution Principle from object-oriented software design. An *AbstractRobotProfile* can inherit from another *AbstractRobotProfile* by specifying it as a base. For example, an *AbstractRobotProfile* called "Driving Robot" can represent a robot that is capable of moving in different directions and stopping. The following robots all have driving capabilities: rover, vacuum cleaner, and drone. Hence, their *AbstractRobotProfiles* can inherit all the *Actions* and *Sensors* from the "Driving Robot" profile by specifying it as their base. *Actions* are executable ("*IExecutable*") indicating that end-users can execute them to command a robot. We differentiate executable commands from other elements because our system supports blocking the execution of such commands based on changes in the context of use as we explain later in the paper.

As shown in Fig. 4, *ConcreteRobotProfiles* map to code-based APIs that control robots. APIs typically comprise *Namespaces* (packages), *Classes*, and *Methods*. A *ConcreteRobotProfile* is associated with an *APIConfiguration*. In turn, the latter comprises *MappingGroups* containing *Action* and *SensorReading Mappings* that connect *ConcreteActions* and *ConcreteSensorReadings* to their respective *APIMethods*. *MappingGroups* also map the parameters of *ConcreteActions* to their matching *APIMethodParameters*. Each *RobotAPIConfiguration* is also associated with one or more *Connections*. Each *Connection* has an *IP address*, a *port name*, and a *label*. A *Connection* could require both an IP and a port or just one of them. The label identifies the use of the *Connection*. For example, the left movement motor of a Lego Mindstorms robot could be using "PortB" and has a label that is "LeftMovement".

We used the concept of a *RobotProfile* to represent the capabilities of robots because these profiles provide levels of abstraction (abstract and concrete). *RobotProfiles* also offer the extensibility for supporting different types of robots. Hence, EUD-MARS is by design independent of the types of robots that software developers will integrate into it. We derived the concept of an API from the real world where robot providers and third parties develop robotics APIs using code-based programming languages. These APIs enable professional software developers to control robots from their software applications. They can be bundled, for example, as reusable libraries. As part of the evaluation with robots, we extended and used APIs that are available for the robotics kit (Lego Mindstorms) and the robots (NAO and Parrot Bebop 2) that we used. In this paper, we did not target the integration of robotics middleware such as ROS with EUD-MARS, because we primarily aimed to realize scenarios for evaluating our approach with users and we were able to do this using the available APIs. Hence, EUD-MARS makes direct calls to the APIs that in turn communicate with the robots (e.g., through Bluetooth and TCP/UDP). Nonetheless, with our model-driven approach as demonstrated by adding existing APIs, it is possible to use

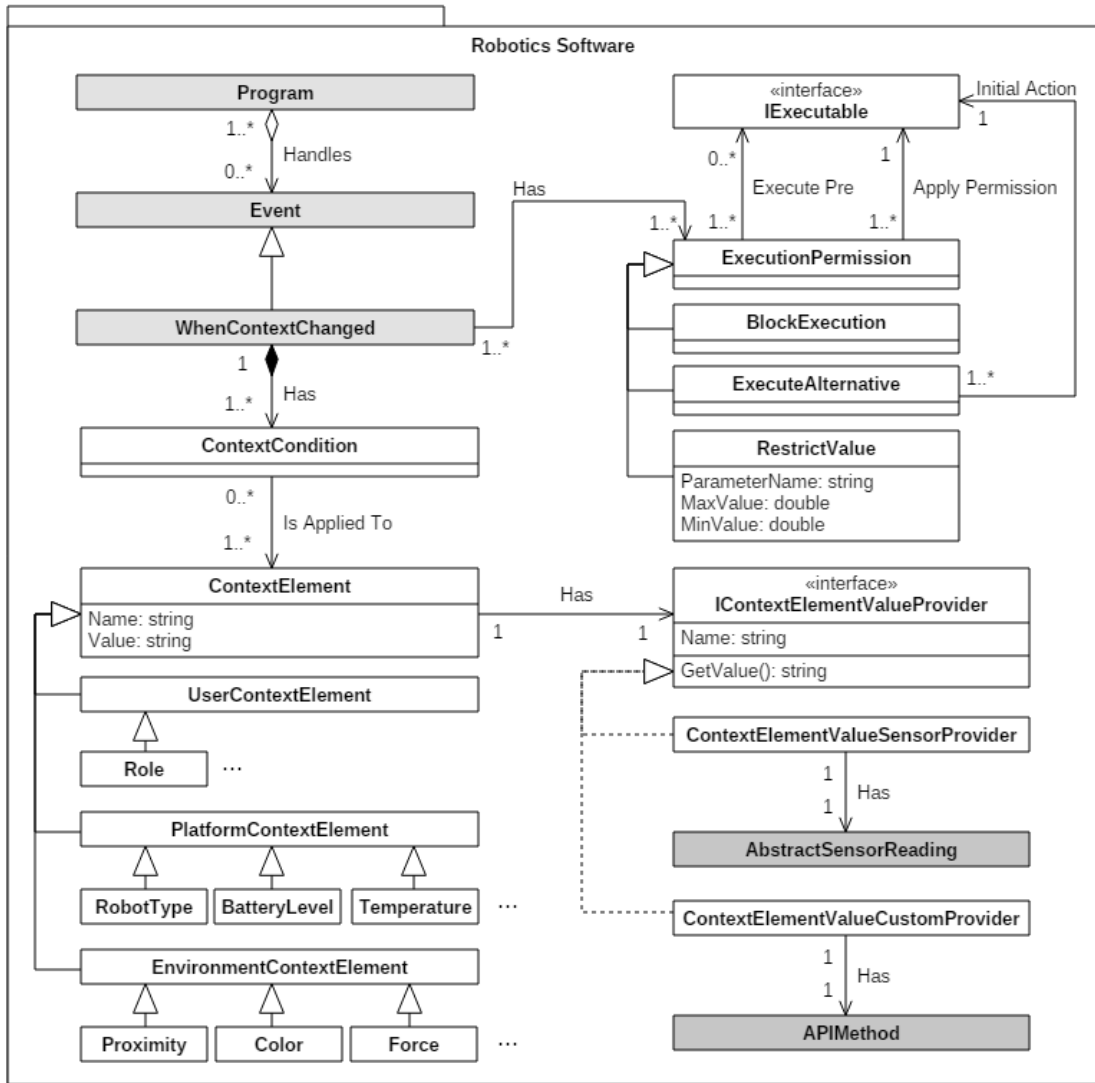


Fig. 7. Adaptation based on the context of use.

alternative underlying implementations to support middleware in the future. For example, it is currently possible to swap one API with another by simply changing the mappings between a *RobotProfile* and its corresponding API.

3.2. Action sequences

Each EUD-MARS *Program* has a *RobotPool* that contains the collection of *Robots* available for programming. These *Robots* are the ones that software developers preconfigured for end-users, by preparing *RobotProfiles* and *API Mappings* (refer to Section 3.1). EUD-MARS *Programs* also have *ActionSequences* that embody the *Actions* used to command a robot for performing a task. The class diagram in Fig. 5 encompasses the concepts related to *ActionSequences*.

ActionSequences can be abstract or concrete. There are two types of *AbstractActionSequences*. The first type is *VirtualAbstractActionSequence*, which represents an operation that multiple robots can perform each in their way. One example of this type of action sequence is “PerformGreeting”. A humanoid robot can wave its hand and say hello, whereas a drone can fly in a certain pattern to greet a person. The second type is *ActualAbstractActionSequence*, which end-users can place inside a *VirtualAbstractActionSequence* to specify how each robot carries out an operation such as “PerformGreeting”. Each *ActualAbstractActionSequence* contains *AbstractActions* and *AbstractSensorReadings* that a particular type of robot can perform. An *AbstractActionSequence* can have parameters of different types.

Each *ActualAbstractActionSequence* maps to a *ConcreteActionSequence*, which in turn contains *ConcreteActions* and *ConcreteSensorReadings* pertaining to a type of *Robot*. Like *Actions* (refer to Section 3.1), *ActionSequences* are also executable (i.e., implement *IExecutable*) and can thereby be blocked from executing based on changes in the context of use.

As shown in Fig. 5, each of the binary many-to-many relationships connecting *ActionSequences* to *Actions* and *SensorReadings* has an association class related to a *Robot* that represents an attribute on the relationship. This attribute denotes the *Robot* on which EUD-MARS shall execute each robot-specific *Action* in an *ActionSequence*.

We organized the visual programs of EUD-MARS into *ActionSequences*, as shown in Fig. 5, to create different levels of abstraction and to make the visual programs resemble a simplified form of code-based programs. The levels of abstraction are abstract and concrete as is the case with the robot profiles (refer to Section 3.1). Hence, abstract sequences are independent of the underlying technology whereas concrete sequences comprise elements (actions and readings) of concrete robot profiles that in turn map to APIs. As programming constructs, *ActionSequences* act like functions in code-based programs and can fulfill different purposes. *AbstractActualActionSequence* acts as a standard parameterized function that embodies reusable logic and is callable from other *ActionSequences*. *VirtualAbstractActionSequence* acts as a pure virtual method that does not have any implementation of its own. End-users provide different robot-specific implementations for a *VirtualAbstractActionSequence* using *ActualAbstractActionSequences*. This resembles the concept of overriding in code-based programs. Since the visual language does not support classes, end-users perform “overriding” by visually placing the *ActualAbstractActionSequences* within their corresponding *VirtualAbstractActionSequences* and selecting the target *RobotProfile* for each sequence. A visual program defined by end-users contains *ActionSequences* in a way that is similar to how procedural programs are comprised of functions.

3.3. Events, variables, control structures, and utilities

As with any program, visual EUD-MARS programs support *Events*, *Variables*, *ControlStructures*, and *Utilities*. End-users can use these concepts in *ActionSequences* alongside *Actions* and *SensorReadings*. The class diagram in Fig. 6 encompasses these concepts.

The execution environment of a program triggers *Events* when an occurrence happens. Examples of *Events* include detecting when the user presses a key or when there is a change in the context of use. End-users can handle *Events* to execute instructions. For example, when a robot detects that it is within a close distance to an object, it executes the “stop” *Action* to avoid a collision.

EUD-MARS programs support *If-Conditions*, *Loops*, and *Variables*. These act like their counterparts in code-based programming languages. *If-Conditions* evaluate to “true” or “false” indicating whether or not the instructions within shall execute. *For-Loops* iterate based on a counter while *ForEach-Loops* iterate around the elements of a list. Supporting these features in EUD-MARS enables end-users to develop useful programs. The type of a *Variable* is denoted by the *DataType* enumeration, just like the types of parameters (refer to Fig. 3 and Fig. 5).

End-users can use several *Utilities* such as *TimeDelay*, *ColorCreator*, and *ListCreator*. These *Utilities* enable end-users to utilize functions that are common for various programs. The *TimeDelay* takes a time value (number) and a time unit such as seconds, and it delays the execution of instructions until the time has passed; this is similar to “wait” or “sleep” functions in code-based programming languages. The *TimeDelay* is useful for making a *Robot* wait for a specified time before it proceeds with an *Action*. One example is that if an end-user wants to orchestrate dance moves with pauses for a humanoid (e.g., NAO). The *ColorCreator* takes RGB values as numbers and returns the corresponding color value. End-users can assign the returned color value to a variable or pass it directly to an *ActionSequence* as a parameter. The *ColorCreator* enables end-users to create a specific color when it is not enough to choose from a predefined set in a color palette. One example of the use of colors is when a program is instructing a robot to identify a color through its color sensor. The *ListCreator* creates a list of values that the end-users could use within their programs. For example, end-users can create a list of numbers using the *ListCreator* and then loop around this list and pass each of its values to an *Action*. As a visual construct, the *ListCreator* allows the definition of an expandable number of input elements that end-users would supply with variables out of which they want to create a list.

3.4. Adaptation based on the context of use

In addition to being model-driven, another main feature of EUD-MARS programs is their support for adaptation based on changes in the context of use. The class diagram in Fig. 7 encompasses the concepts related to adaptation.

End-users can use *WhenContextChanged* events and *ContextConditions* to detect changes in a robot’s context of use. End-users can apply *ContextConditions* to a variety of *ContextElements* that have one of the following three types: user, platform, and environment. Calvary et al. [92] decomposed the context of use into these three facets. A *Role* is an example of a *UserContextElement*. End-users can use roles to distinguish between different types of users. For example, an “adult” user would have access to more features than a “child” user. Robot type, battery level, and temperature are examples of *PlatformContextElements*. For example, end-users could enable and disable actions based on the robot’s battery level and temperature. This allows the battery to last longer so the robot can perform critical tasks and allows the robot’s temperature to cool down. End-users can also setup adaptations based on changes in *EnvironmentContextElements* such as the proximity of a robot from objects detected by a proximity sensor, colors detected by a robot’s color sensor, and force applied on a

robot's pressure sensor. Robot sensors are the most common source of context-of-use change detection. These sensors inform a program of changes in real-time, so it can trigger immediate adaptations.

End-users can attach *WhenContextChanged Events* to several *ExecutionPermissions*, which include the following types: *BlockExecution*, *ExecuteAlternative*, and *RestrictValue*. *BlockExecution* prevents a robot from executing an *IExecutable* (action or action sequence; refer to Sections 3.1 and 3.2) based on the value of a *ContextElement*. For example, as long as the role of the user is "child" or as long as the robot's color sensor is detecting a green color, prevent the robot from shooting pellets. *ExecuteAlternative* executes an alternative *IExecutable* when the context changes. For example, when a robot receives a "move forward" command and is about to fall off an edge, alternative actions would be "move backward" and "stop moving". *RestrictValue* sets a minimum and a maximum for the value given to a parameter of an *IExecutable*. For example, end-users can restrict the speed at which a robot moves based on certain roles or battery levels.

Providers that implement the *IContextElementValueProvider* interface are responsible for providing the program with the values of *ContextElements*. A few of these values are preset and seldom change, e.g., roles, but most values are continuously changing in real-time, e.g., color, battery level, proximity, and pressure. Software developers predefine value providers that end-users can use in their programs out of the box. The type of a *Provider* can be either "sensor" or "custom" indicating that it reads values from either a *Sensor* or a different source respectively. *Sensors* and *SensorReadings* are already part of *RobotProfiles* (refer to Fig. 3). Hence, a *SensorProvider* refers to the *SensorReading* from which it obtains the value of a context element. Professional software developers preprogram *CustomProviders* and configure them by linking each one to an *APIMethod*. An API here represents reusable code; it encompasses the same concepts as a robot API (refer to Fig. 4). Currently, software developers can program *CustomProviders* as a C# class library or Python script that EUD-MARS can call dynamically the same way it does with robot APIs. Other programming languages can be similarly supported in the future. *CustomProviders* do not directly relate to a particular robot. An example of a *CustomProvider* is one that provides the role of a user (e.g., "child" or "adult"), by reading it from a file.

The adaptations supported by EUD-MARS involve modifying a robot's behavior based on changes in the context of use detected using *Events* (*WhenContextChanged*). When an end-user executes a program, EUD-MARS monitors the respective *Events* in parallel to detect changes in *ContextElement* values read by the corresponding *Providers*. New values could indicate a change in the context of use that requires an adaptation; this depends on the conditions that the end-user specified in the *Events* (e.g., proximity < 100). EUD-MARS monitors the new *ContextElement* values using APIs (*SensorProviders* or *CustomProviders*). The program interpreter of EUD-MARS either uses programming language events (e.g., in C#) or callback functions to detect new values. We aimed to simplify the way end-users program robot-behavior adaptations by providing adaptation-related blocks that they can place within *Event* blocks (e.g., Fig. 2b). Currently, EUD-MARS supports three types of adaptation blocks based on the previously discussed *ExecutionPermissions*: *BlockExecution*, *ExecuteAlternative*, and *RestrictValue*. During the execution of a program, the interpreter holds a set of the currently active *ExecutionPermissions*. Upon detecting the triggering of an *Event*, the interpreter updates this set by adding or removing permissions according to the context-of-use changes. For example, when the *Event* presented in Fig. 2b is triggered the program interpreter would stop "Lego Shooter Bot A" if it is currently moving forward. Then, the program interpreter would add to its set of execution permissions an entry indicating that it shall prevent the execution of the "move forward" action on "Lego Shooter Bot A" as long as the robot's color sensor is still detecting a red color. During this time, the interpreter monitors the colors detected by the sensor to remove the previously added execution permission when a color other than red is detected. Then, the program interpreter goes back to monitoring the original condition (color equals red). While the interpreter is executing a program, it checks if the execution of an *Action* violates an active execution permission to see whether it has to block the *Action*, execute an alternative, or restrict a value.

3.5. Communication among robots

A single EUD-MARS program can include multiple robots that communicate and exchange data. End-users can realize this communication in their programs by using a combination of *Events* and *Action* calls. Robots report changes that trigger *Events*. End-users can handle *Events* in their programs and call parametrized *Actions* to invoke desired behaviors and pass data to other robots. The host device on which an EUD-MARS program is running acts as a central node that enables robots to communicate and exchange data. What follows are example scenarios of communication among robots that are part of the robot pool of an EUD-MARS program.

Consider that there are two robots, e.g., drones, which end-users programmed to clear objects from a specific area. Assume that these robots want to alternate shifts depending on the battery level of the currently active robot. End-users can use an *Event* in their program to receive a notification when the battery level of the active robot is lower than the desired percentage. When this condition is true, the program calls an *ActionSequence* to command the idle robot to carry out the task in place of the active robot. Afterward, the program calls another *ActionSequence* to command the active robot with the low battery level to dock itself and recharge. This example explains how two collaborating robots convey, through the program, data (battery level) and commands (*ActionSequence* for carrying out the task and *ActionSequence* for charging the battery).


```

<AbstractRobotProfile Name="DrivingRobot">
  <Actions>
    <Action Name="MoveForward" ReturnType="Void">
      <Parameters>
        <Parameter Name="Speed" Type="Number" />
      </Parameters>
    </Action>
    <Action Name="MoveBackward" ReturnType="Void">
      <Parameters>
        <Parameter Name="Speed" Type="Number" />
      </Parameters>
    </Action>
    <Action Name="TurnRight" ReturnType="Void">
      <Parameters>
        <Parameter Name="Speed" Type="Number" />
      </Parameters>
    </Action>
    <Action Name="TurnLeft" ReturnType="Void">
      <Parameters>
        <Parameter Name="Speed" Type="Number" />
      </Parameters>
    </Action>
    <Action Name="Stop" ReturnType="Void" />
  </Actions>
  <Sensors>
    <Sensor Name="ProximitySensor">
      <Reading Name="Proximity" ReturnType="Number" />
    </Sensor>
  </Sensors>
</AbstractRobotProfile>
<AbstractRobotProfile Name="ShooterBot" BaseProfile="DrivingRobot">
  <Actions>
    <Action Name="Shoot" ReturnType="Void" />
  </Actions>
  <Sensors>
    <Sensor Name="LightSensor">
      <Reading Name="Color" ReturnType="Number" />
    </Sensor>
  </Sensors>
</AbstractRobotProfile>
<AbstractRobotProfile Name="DriverBot" BaseProfile="DrivingRobot" />
<AbstractRobotProfile Name="VacuumCleaner" BaseProfile="DrivingRobot">
  <Actions>
    <Action Name="PlayMelody" ReturnType="Void">
      <Parameters>
        <Parameter Name="MelodyNumber" Type="Number" />
      </Parameters>
    </Action>
  </Actions>
</AbstractRobotProfile>
<ConcreteRobotProfile Name="LegoShooterBot" AbstractProfileName="ShooterBot" APIConfigName="LegoNXShooterBotAPI" />
<ConcreteRobotProfile Name="LegoDriverBot" AbstractProfileName="DriverBot" APIConfigName="LegoNXDriverBotAPI" />
<ConcreteRobotProfile Name="iRobotVacuumCleaner" AbstractProfileName="VacuumCleaner" APIConfigName="iRobotCreateAPI" />

```

Fig. 8. Example abstract and concrete robot profiles.

Consider another example where robots need to communicate to benefit from each other's sensor capabilities. A drone can use its flying ability and sensors to scout an area and detect objects. Assume that this drone needs to report the location of the objects that it detected to ground rovers. End-user programs can use an *Event* to receive a notification when the drone detects an object. When this event triggers, the end-users can acquire the drone's location through a sensor. Then, they can pass this location to the ground rovers by invoking a parametrized *Action* (e.g., drive to a location). The data exchanged in this scenario are the locations of detected objects and the commands involve asking the rovers to drive to these locations.

3.6. How the EUD-MARS concepts support the development of model-driven and adaptive robotics software

This subsection provides a further explanation of how the EUD-MARS concepts, shown in Fig. 3 to Fig. 7, support the development of model-driven adaptive robotics software.

The separation of *RobotProfiles* into two levels of abstraction, namely *Concrete* and *Abstract* (refer to Fig. 3), allows a common abstract representation to target multiple concrete implementations. For example, it is possible to use this approach to support cross-platform robotics programming. Hence, an *AbstractRobotProfile* provides a common level of ab-

```

<ApiConfiguration Name="LegoNXShooterBotAPI">
  <PortConfigurations>
    <PortConfiguration PortName="COM4" Label="NXTBrick" />
    <PortConfiguration PortName="A" Label="ShooterMotor" />
    <PortConfiguration PortName="B" Label="LeftMovementMotor" />
    <PortConfiguration PortName="C" Label="RightMovementMotor" />
    <PortConfiguration PortName="Fourth" Label="ProximitySensor" />
  </PortConfigurations>
  <MappingGroups>
    <MappingGroup NamespaceName="ShooterBot.LegoNXT" ClassName="ShooterBot">
      <ActionMappings>
        <ActionMapping ActionName="MoveForward" MethodName="MoveForward">
          <ParameterMappings>
            <ParameterMapping ParameterName="Speed" APIParameterName="Speed" />
          </ParameterMappings>
        </ActionMapping>
        <ActionMapping ActionName="MoveBackward" MethodName="MoveBackward">
          <ParameterMappings>
            <ParameterMapping ParameterName="Speed" APIParameterName="Speed" />
          </ParameterMappings>
        </ActionMapping>
        <ActionMapping ActionName="TurnLeft" MethodName="TurnLeft">
          <ParameterMappings>
            <ParameterMapping ParameterName="Speed" APIParameterName="Speed" />
          </ParameterMappings>
        </ActionMapping>
        <ActionMapping ActionName="TurnRight" MethodName="TurnRight">
          <ParameterMappings>
            <ParameterMapping ParameterName="Speed" APIParameterName="Speed" />
          </ParameterMappings>
        </ActionMapping>
        <ActionMapping ActionName="Stop" MethodName="Stop" />
      </ActionMappings>
      <SensorReadingMappings>
        <SensorReadingMapping SensorName="ProximitySensor" ReadingName="Proximity"
          MethodName="GetProximity" />
        <SensorReadingMapping SensorName="LightSensor" ReadingName="Color"
          MethodName="GetColor" />
      </SensorReadingMappings>
    </MappingGroup>
  </MappingGroups>
</ApiConfiguration>

<!--Similar configurations (e.g., NXTDriverBotAPI and iRobotAPI) are added for the other robots-->

```

Fig. 9. Example mapping between concrete level and API.

straction above multiple *ConcreteRobotProfiles* that target specific hardware platforms. For example, the same *AbstractRobotProfile* can map to two *ConcreteRobotProfiles* representing two rovers, one developed using Lego Mindstorms and another developed using VEX. Furthermore, it is possible to use an *AbstractRobotProfile* as a base for another one. Hence, an *AbstractRobotProfile* can encompass functionalities that are common among multiple *AbstractRobotProfiles* that represent different robots with common functionalities. This applies if these robots use the same hardware platform and/or different ones. The *ConcreteRobotProfiles* in turn map to platform-specific *RobotAPIs* (refer to Fig. 4) that software developers created with code-based programming languages. Hence, our design also allows visual EUD-MARS programs to target multiple APIs written in different programming languages.

When end-users use the general concepts provided by the highest level of abstraction (*AbstractRobotProfile*), they would be indirectly writing a program that targets all the *ConcreteRobotProfiles* (and subsequently the *RobotAPIs*) that map to this *AbstractProfile*. The same principle of abstraction applies in the *ActionSequences* shown in Fig. 5, with the *Abstract*, *Virtual*, and *Actual ActionSequences* representing multiple levels of abstraction.

The concepts in Fig. 7 enable end-users to develop adaptive robotics software in a standard and simplified manner. This involves using *Events* that offer notifications when *ContextElements* change. Alongside these *Events*, end-users can execute and block certain actions to change the behavior of a robot. These features save the end-users having to write code-based programs from scratch to detect context-of-use changes and adapt a robot's behavior. Writing code-based programs is more complex and can significantly differ per robot. However, with the visual notation of EUD-MARS, the development of adaptive behavior for different types of robots becomes more standard since it merely includes a combination of *Events*, *Actions*, and *ExecutionPermissions*. As previously mentioned, we do not intend to support an adaptation mechanism that substitutes traditional technical solutions that follow reference architectures for adaptive systems (refer to Section 2.3). Yet, we aim to enable end-users to incorporate some adaptation capabilities into their programs in a simplified manner.

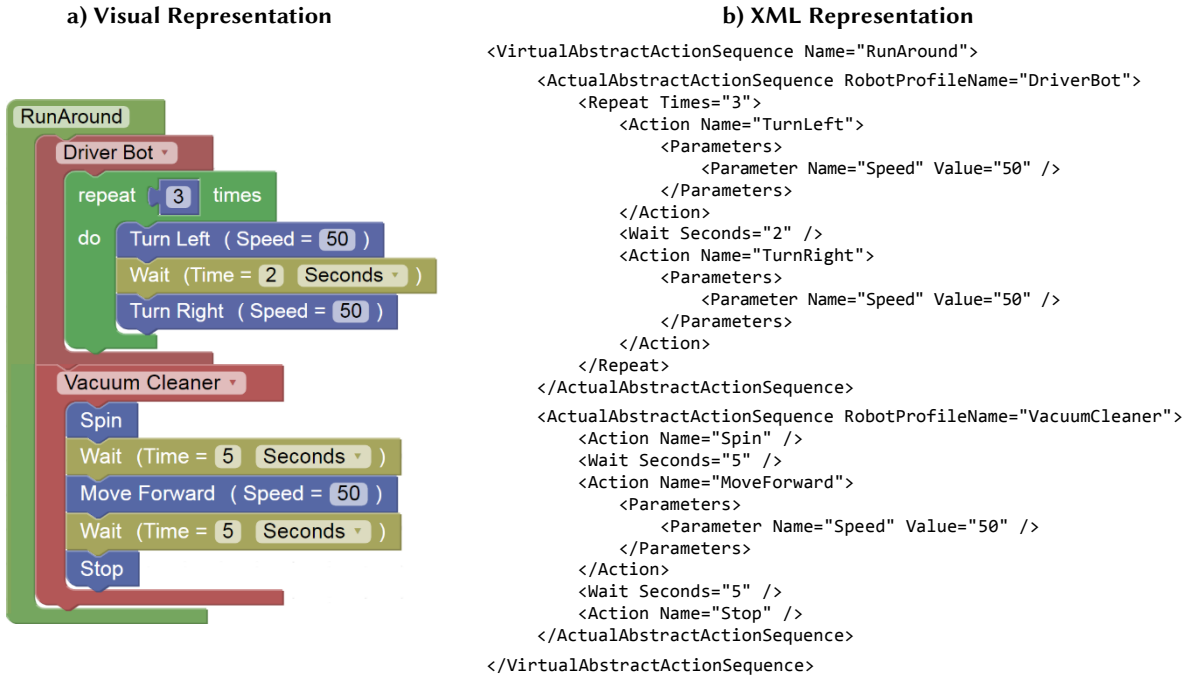


Fig. 10. Example virtual action sequence with different implementations. The (a) visual program is what end-users develop, while the (b) XML representation is what EUD-MARS generates from this program.

3.7. Currently supported concepts and extensibility

We presented the concepts currently supported by EUD-MARS in the meta-models shown in Fig. 3 to Fig. 7, and we explained them in Sections 3.1 to 3.6. These concepts include robot profiles and API configurations that professional programmers prepare, and the end-users' visual programming language constructs such as action sequences, adaptation, events, variables, control structures, and utilities.

The currently supported robot profile and API concepts make EUD-MARS extensible by design concerning the integration of various types of new robots. This extensibility is due to these concepts conforming to the same generic meta-model. Hence, for example, any new robot profile will have actions and sensors regardless of the type of robot. EUD-MARS' programming tool, visual-program-to-XML converter, and program interpreter (refer to Fig. 1) can automatically incorporate new robot profiles without the need for modifying their source code or having to define new converters and interpreters. The converter will still be able to transform the visual program to XML for the interpreter because the extensions introduced by the robot profiles conform to the same meta-model. For example, a drone and a rover are both robots, and "fly" and "move" are both actions. The end-user programming tool can display new robot profiles in the toolbox automatically since the concepts they comprise use the same style of visual blocks. For example, the visual block of an "action" presents its name, parameters (if any), and the robot on which to execute it.

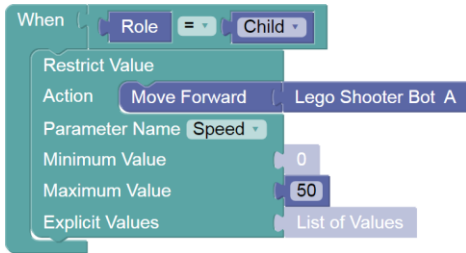
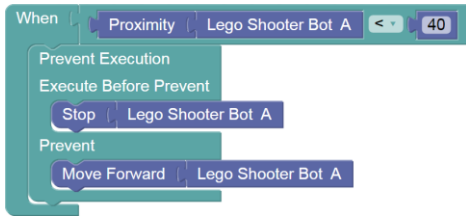
The currently supported programming language constructs allow end-users to create a variety of programs for controlling robots. However, we do not claim that this language is comprehensive. It is possible to extend the language's constructs further in the future. For example, it is possible to add more types of adaptation blocks, utilities, control structures, and even support for classes. Yet, what we have included so far is enough to assess adaptive model-driven software development scenarios with end-users and to evaluate our approach with different types of robots. Unlike robot profiles and API configurations, adding new constructs to the visual language requires extending the visual blocks, converter, and interpreter. Professional programmers can use JavaScript to define new visual blocks (based on Blockly). The end-user programming tool can display the new visual blocks in the toolbar, and thereby allow end-users to use them as part of their programs. Programmers can extend the converter and interpreter using JavaScript and C# respectively to recognize the new constructs.

Similarly, the interpreter requires extension if new API programming languages need to be supported. The interpreter that we developed for this paper directly communicates with APIs written in C# (libraries) and Python (scripts). In the future, we can adjust the interpreter to perform this communication through web-services to be able to support new languages without having to modify its code. Web services (e.g., RESTful services): (1) are a typical way of software system integration; (2) enable software systems to communicate regardless of each system's programming language and frame-

a) XML Representation of Context Elements and Providers

```
<ContextElements>
  <ContextElement Name="ShooterBotProximity">
    <Provider Type="Sensor" Name="ProximitySensor" ReadingName="Proximity" RobotName="ShooterBotA" />
  </ContextElement>
  <ContextElement Name="Role">
    <Provider Type="Custom" Name="RoleProvider" NamespaceName="MARS" ClassName="RoleProvider" MethodName="GetName" />
  </ContextElement>
</ContextElements>
```

b) Visual Representation of Context Condition Sequence



c) XML Representation of Context Condition Sequence

```
<ContextConditionSequence Name="ShooterBotCloseToObstacle">
  <ContextConditions>
    <ContextCondition>
      <ContextElement Name="ShooterBotProximity" RobotName="ShooterBotA"
        ComparisonOperator="<" Value="40" />
    </ContextCondition>
  </ContextConditions>
  <ExecutionPermissions>
    <PreventExecution>
      <ExecuteBeforePrevent>
        <Action Name="Stop" RobotName="ShooterBotA" />
      </ExecuteBeforePrevent>
      <Prevent>
        <Action Name="MoveForward" RobotName="ShooterBotA" />
      </Prevent>
    </PreventExecution>
  </ExecutionPermissions>
</ContextConditionSequence>

<ContextConditionSequence Name="RoleIsChild">
  <ContextConditions>
    <ContextCondition>
      <ContextElement Name="Role" ComparisonOperator="=" Value="Child" />
    </ContextCondition>
  </ContextConditions>
  <ExecutionPermissions>
    <RestrictValue>
      <ApplyPermissionOn>
        <Action Name="MoveForward" RobotName="ShooterBotA">
          <Parameters>
            <Parameter Name="Speed" MaxValue="50" />
          </Parameters>
        </Action>
      </ApplyPermissionOn>
    </RestrictValue>
  </ExecutionPermissions>
</ContextConditionSequence>
```

Fig. 11. Example adaptations based on the context of use. Software developers specify the (a) context elements and providers beforehand. Then, end-users develop the (b) visual programs and EUD-MARS generates (c) XML representations from these programs.

work; (3) wrap around the functions that shall be called and make them accessible to external systems through HTTP. Web services can be a typical solution to make EUD-MARS communicate with APIs written in languages that may not be directly callable from C# (the language in which we programmed the interpreter). Researchers have used web services with other robotics systems such as ROS [110,111]. As far as EUD-MARS is concerned, its API mapping mechanism (refer to Section 3.1 and Fig. 4) would work in the same way whether its interpreter is calling an API's functions directly or through functions defined in a web service.

4. An example EUD-MARS application

This section presents a working example that demonstrates the concepts explained in Section 3. We represented this example visually as well as using XML. The visual representation follows the notation that we created using Blockly. The XML representation is the underlying format of the visual programs of EUD-MARS, which end-users do not need to process. An EUD-MARS interpreter is responsible for interpreting the XML-based programs and executing them at runtime (refer Fig. 1). Software developers use XML to create artifacts such as robot profiles, which pave the way for end-users to

```

<Program>
  <RobotPool>
    <Robot Name="ShooterBotA" Profile="LegoShooterBot" />
    <Robot Name="ShooterBotB" Profile="LegoShooterBot" />
    <Robot Name="iRobotCreateWhite" Profile="iRobotVacuumCleaner" />
    <Robot Name="iRobotCreateBlue" Profile="iRobotVacuumCleaner" />
  </RobotPool>
  <Events>
    <!--Contains the definitions of the events, e.g., the ones used for adaptation-->
  </Events>
  <ActionSequences>
    <!--Contains the definitions of the virtual and actual action sequences-->
  </ActionSequences>
  <ExecutableCalls>
    <!--Contains calls to executable elements, e.g., action sequences-->
  </ExecutableCalls>
</Program>

```

Fig. 12. Example showing the XML representation of a robotics program with its different sections.

develop programs visually. We should note that software developers could either create these artifacts directly using XML or using a visual tool that we shall present in the next section.

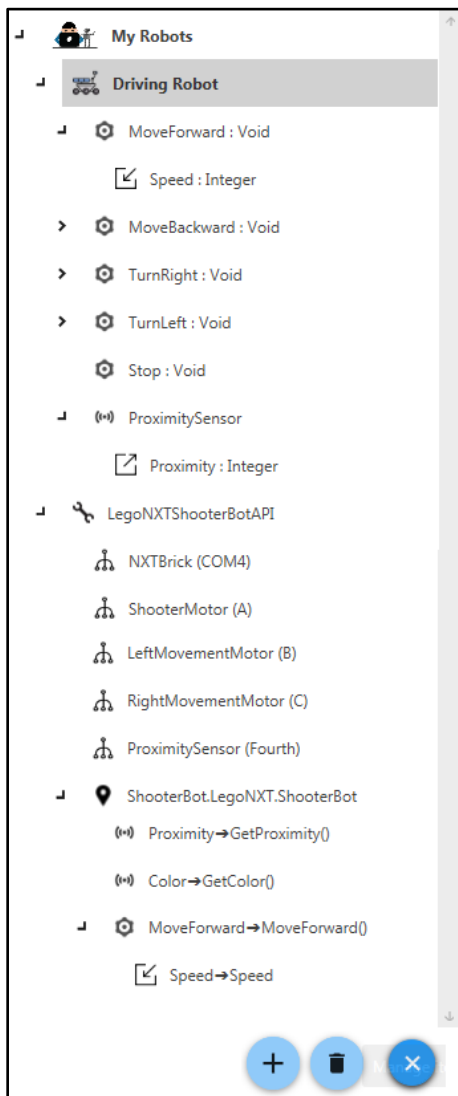
This example starts with the definition of *RobotProfiles* as shown in Fig. 8. The “DrivingRobot” *AbstractRobotProfile* has the following five actions: *MoveForward*, *MoveBackward*, *TurnRight*, *TurnLeft*, and *Stop*. All the actions, except *Stop*, take a parameter called speed. The “DrivingRobot” also has a “ProximitySensor” that reads the *proximity*. This example also includes three robot profiles that use “DrivingRobot” as a base profile. These profiles are “ShooterBot”, “DriverBot”, and “VacuumCleaner”. We can see that the “ShooterBot” adds a *Shoot* action and a *LightSensor* with a color reading. The “VacuumCleaner” adds a *PlayMelody* action that takes a melody number as a parameter. The example also shows three *ConcreteRobotProfiles* that connect the *AbstractRobotProfiles* to their respective APIs, by setting the API configuration names. The example shown in Fig. 9 represents an API configuration. This example shows *MappingGroups* that connect actions, parameters, and sensors to their corresponding API counterparts. This example also shows configurations of ports that programs connect to for commanding robots. The *ConcreteRobotProfiles*, shown in Fig. 8, have the same configuration of *Actions* and *Sensors* as their *AbstractRobotProfile* counterparts. Therefore, EUD-MARS performs automatic mappings by copying the *Actions* and *Sensors* from the *AbstractRobotProfiles* to their concrete counterparts. On the other hand, as shown in Fig. 9, software developers define mappings between *ConcreteRobotProfiles* and APIs, because the structure and names can differ between a *RobotProfile* and its corresponding API. We should note that in the example shown in Fig. 8, it happens to be that the *Actions* do not have return types (specified as void) and the *SensorReadings* do not have parameters, but this is not necessarily always the case.

The *VirtualAbstractActionSequence* in Fig. 10 represents a “RunAround” operation and contains two *ActualAbstractActionSequences*. Each of the latter provides an implementation for a different robot, namely “DriverBot” and “VacuumCleaner”. The “DriverBot” performs the “RunAround” operation by executing a loop that repeats three times the following actions: *Turn Left*, *Wait*, and *Turn Right*. On the other hand, the “VacuumCleaner” performs the “RunAround” operation by executing the following actions: *Spin*, *Wait*, *MoveForward*, and *Stop*.

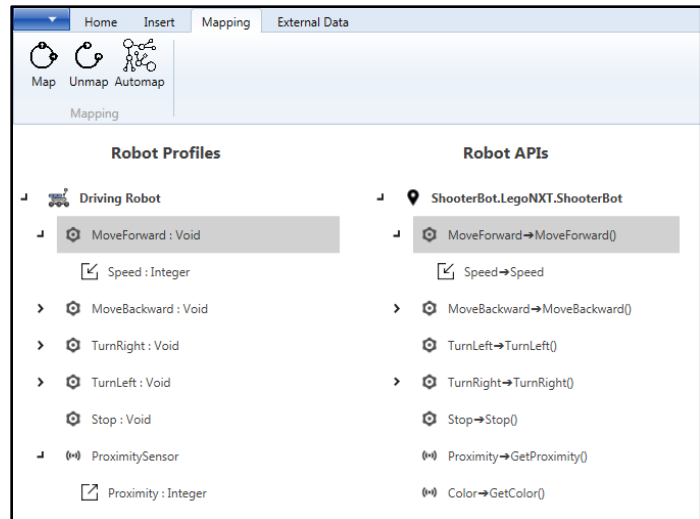
This example also contains an implementation of adaptive behavior, shown in Fig. 11. This implementation uses *ContextElements* and *ContextConditions* and applies *ExecutionPermissions* on certain actions when the context of use changes. The “ProximitySensor” of the robot called “ShooterBotA” provides the value for the “ShooterBotProximity” *ContextElement*. On the other hand, a class called “RoleProvider” provides the value for the “Role” (i.e., the role of current robot user) *ContextElement*. This example defines two *ContextConditions*. The first *ContextCondition* checks if the “ShooterBotProximity” is smaller than 40, then it triggers a stop action and prevents the robot from moving forward. This prevents the robot from bumping into obstacles. The second *ContextCondition* checks that if the “Role” is “Child”, then it restricts the value of the speed parameter on the “MoveForward” action to a maximum of 50. Here, we should note that a program does not necessarily have to apply *ExecutionPermissions* to the same robot that triggered a change in a *ContextCondition*. For example, if a robot detects a certain color, the program could prevent other robots from moving forward. We can consider this as a form of communication (refer to Section 3.5) among the collaborating robots. Hence, the abilities, e.g., sensors, of one robot could provide data for a *ContextCondition* that affects other robots, which might not have the same abilities. However, we should note that exploring collaboration among robots in-depth as was done in existing work, e.g., collaborative security [93], is not the aim of this paper.

We used “ShooterBotA” as a name for the robot that is part of the example shown in Fig. 11. Hence, the *RobotPool* in this example program can contain other shooter-bot robots with different names, and it can contain robots of completely different types. As the example in Fig. 12 shows, each program comprises a *RobotPool* containing all the robots that the program intends to control. We can see that there can be multiple instances of the same type of robot, which map to the same *ConcreteRobotProfile*. For example, both ShooterBotA and ShooterBotB map to the LegoShooterBot profile, and both

a) Project Explorer



b) Mapping Robot Profiles to APIs



c) Loading API Data from an Assembly

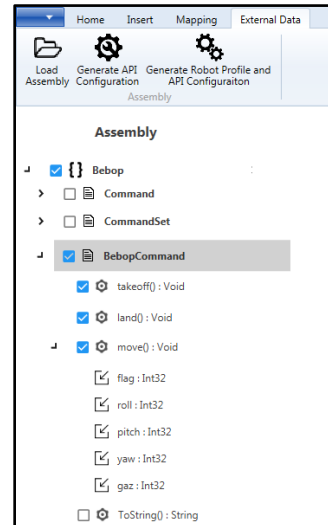


Fig. 13. EUD-MARS' tool for assisting software developers in composing robot profiles and API configurations.

iRobotCreateWhite and iRobotCreateBlue map to the iRobotVacuumCleaner profile. As Fig. 12 shows, a robotics program is composed of several sections. In addition to the *RobotPool*, a program contains the following sections: *Events*, *ActionSequences*, and *ExecutableCalls*. The *Events* section contains events such as the ones used for adaptation. The *ActionSequences* section contains the definitions of both the *Virtual* and *Actual ActionSequences*. The calls that end-users make to these *ActionSequences* are located under the *ExecutableCalls* section.

EUD-MARS generates XML representations like the one shown in Fig. 12 from the visual programs defined by end-users. When end-users are developing a program, they could define the different elements, action sequences, events, and calls using any visual organization that they choose. Upon saving a program and generating its XML representation, EUD-MARS regroups the elements under the different sections shown in Fig. 12. Robot profiles, mappings, sequences, and adaptations, presented in this section, are contained within programs such as the one shown in Fig. 12.

After demonstrating this example of EUD-MARS and its XML-based representation, we should emphasize that EUD-MARS only uses Blockly for the visual representation of its programs. It does not use Blockly for XML generation. The XML representation of the programs comprises the EUD-MARS concepts presented in Fig. 3 to Fig. 7. A converter component (from EUD-MARS) transforms the visual programs to an XML representation that is interpretable by an EUD-MARS interpreter. We should also reemphasize that we do not expect end-users to define robotics programs using XML. We provide end-users with a visual language that makes the development of robotics programs less challenging. The XML

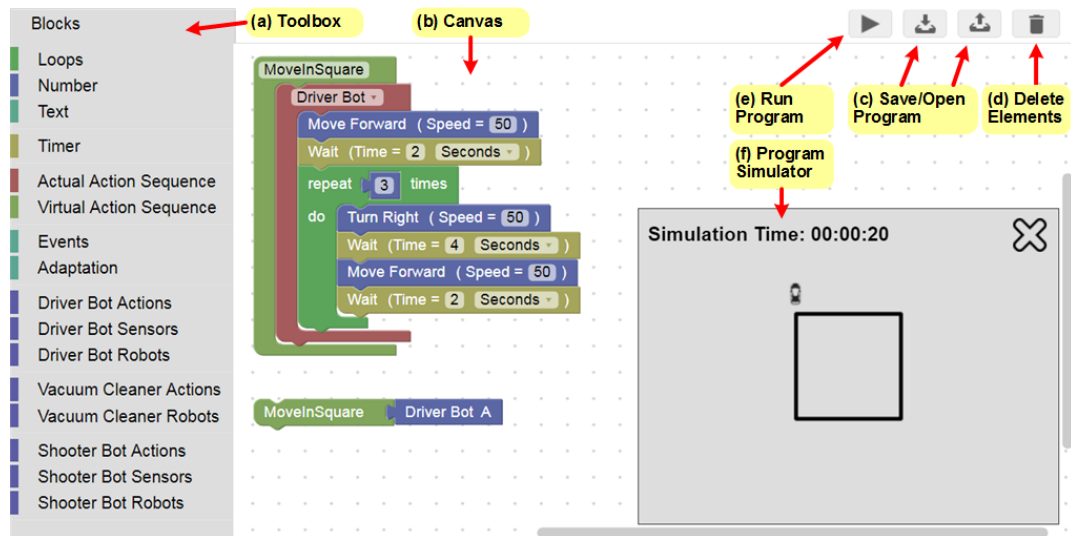


Fig. 14. EUD-MARS' visual programming environment for end-users (based on Blockly).

that we demonstrated in this section is the underlying representation of this visual language. Hence, with EUD-MARS, end-users do not have to create, read, or update any artifacts using XML or any other text-based language. The next section presents the tools that we devised to support the software developers and end-users who wish to use EUD-MARS.

5. Tool support

This section presents two tools that we developed. The first tool, shown in Fig. 13, supports software developers in visually composing robot profiles and API configurations. The second tool, shown in Fig. 14, supports end-users in developing model-driven adaptive robotics software using the visual language that we created based on Blockly.

5.1. Tool for software developers






As Fig. 1 shows, software developers are responsible for preparing profiles for the robots that end-users want to program. Software developers can define robot profiles using XML, like the examples shown in Fig. 8 and Fig. 9. They can also use the visual tool (Fig. 13) that we developed for this purpose.

We can see in Fig. 13a how this tool loads entire robot profiles and API configurations into its project explorer. The tree nodes of the project explorer have different icons and labels to show essential information. For example, the label of an “action” is composed of the action’s name and return type. The project explorer displays multiple projects. Each project is composed of the following items: abstract robot profiles, concrete robot profiles, API configurations, and all the respective sub-items. The project presented in Fig. 13a corresponds to the examples from Fig. 8 and Fig. 9.

It is possible to add and remove items as shown in Fig. 13a (bottom). When a software developer selects one of the nodes in the project explorer, the tool displays a form for editing the details of the item that this node represents. For example, upon selecting a sensor-reading-mapping it is possible to modify the sensor-name, reading-name, and the name of the API method that maps to the reading. Software developers can associate API configuration mappings with robot profiles using the visual tool shown in Fig. 13b. After the software developers select an API method (Fig. 13b left) and an action or a sensor reading (Fig. 13b right), they can map the two to each other with the click of a button.

Although it is possible to define robot profiles and API configurations manually as we previously mentioned, the tool is also capable of generating this data from existing robot API libraries (Fig. 13c). The tool loads the API libraries dynamically and reads their meta-data that includes the following information: namespace (package) names, class names, method names and return types, and parameter names and types. Upon loading this data, software developers can select the classes and methods that they need from the assembly. Then, they can use the tool to generate a skeleton for a robot profile and an API configuration that maps this profile to the actual API (classes, methods, etc.). Afterward, software developers can manually tailor the generated profiles and configurations according to their needs. Currently, it is possible to load API data from .NET assemblies (class libraries) using reflection. Nonetheless, in the future, it is possible to extend this tool to load API data from libraries defined with other technologies.

Table 1. Robots Used in the Technical Evaluation

Robot	Lego Mindstorms (Driver Bot)	Lego Mindstorms (Shooter Bot)	iRobot Create	NAO	Parrot Bebop 2
					
Actions	Move Backward, Move Forward, Stop, Turn Left, Turn Right	Move Backward, Move Forward, Shoot, Stop, Turn Left, Turn Right	Move Backward, Move Forward, Play Music, Set Mode, Spin, Stop, Turn Left, Turn Right	Blink, Get Date, Get Name, Is In Darkness, Look At, Move To, Play Sound, Point At, Say, Sit Down, Stand Up, etc.	Enable Video, Land, Move Backward, Move Down, Move Forward, Move Up, Pause, Take Off, Turn Left, Turn Right
Sensors	Ultrasonic	Ultrasonic, Color	Battery, Bumper, Buttons	Battery, Bumpers, Collision, Motor Heat, Posture, Sonar, Tactile, etc.	Barometer, GPS, Magnetometer, Ultrasonic

5.2. Tool for end-users

Tool-supported visual paradigms enable end-users to perform some of the programming tasks that professional programmers usually perform. Since we primarily intended EUD-MARS to be for end-users, we devised a visual programming tool to complement it.

This tool (Fig. 14) is web-based and uses Blockly. The latter is a JavaScript library created by Google to support the development of visual programming languages and tools [94]. Blockly represents coding concepts as interlocking blocks. Scratch, the end-user visual programming tool has been using interlocking blocks since before Blockly's initial release. Researchers have used Blockly to develop languages for several types of applications including model checking [95], web development [96], and data queries [97]. Blockly's paradigm is similar to jigsaw puzzles, which researchers introduced and evaluated in the area of end-user development [98,99]. We also used jigsaw puzzles as the visual notation for Visual Simple Transformations (ViSiT), which is an approach that empowers end-users to wire together previously incompatible IoT objects [100]. The previous work showed that end-users perceive visual interlocking blocks to be usable and learnable. Hence, we used this visualization paradigm for EUD-MARS since it is a promising choice.

Using the tool that we developed, end-users can drag visual programming blocks from a toolbar (Fig. 14a) onto a canvas (Fig. 14b). These blocks represent concepts such as actions, action sequences, events, and so on. End-users can save their visual programs to files and reload them later to continue their work (Fig. 14c). The toolbar contains some fixed elements such as the ones related to numbers and text. The toolbar also contains other elements such as actions, which are loaded dynamically based on the program's robot pool. End-users can drag and drop elements to move them around on the canvas. They can also delete individual elements or the entire program (Fig. 14d). End-users can zoom the canvas in and out to visualize the program in a better way on different resolutions.

End-users can run their programs directly from the tool (Fig. 14e). Since the programming environment is web-based, we developed a desktop-based interpreter (using C#), which takes the XML-based output of the programming environment and executes it on the target robots. This tool interprets the XML-based programs and maps the elements to their respective API counterparts, based on a configuration like the one presented in Fig. 9. Then, this tool loads the API libraries and makes dynamic function calls to execute the program's actions on the physical robots. As we explained in Section 1.1, when an end-user runs a program, the converter transforms the visual program to an XML-based representation that the interpreter reads and executes. The interpreter interprets programs represented using the XML-based format of EUD-MARS to stay independent from Blockly. The converter provides this independence by performing a transformation between the visual programs and the XML-based representation of EUD-MARS. This way, if we decide to substitute Blockly with a different library or even create another end-user development tool (e.g., for the mobile or the desktop) we will not have to change our interpreter. We would just need to extend the converter or develop another one to support different transformations. We developed the converter for this paper using JavaScript because our end-user development tool is web-based. This converter iterates through the elements of a visual program and generates the corresponding XML.

As we explained in Section 3, a program is composed of different constructs such as action sequences (e.g., Fig. 2a) and events (e.g., Fig. 2b). A program has an entry action sequence that the program interpreter will execute first. This entry action sequence can call other action sequences that can in turn call others and so on. We already explained in Section 3.4

that the interpreter handles events in parallel with the action sequences to monitor changes to the context of use and apply adaptations accordingly.

End-users can also execute their programs in a simple two-dimensional (2D) simulator (Fig. 14f). The simulator currently supports basic movements that are displayable in a 2D environment. For example, it is possible to show a rover robot moving around. However, currently, it is not possible to simulate drones flying up and down since that would require a more sophisticated 3D simulator.

6. Technical evaluation

As we mentioned in Section 1.3, EUD-MARS aims to support an extensible number of robot types that are usually available for end-users. Hence, this technical evaluation involved running EUD-MARS programs on a variety of such robot types. The visual language plays an important role in the ability of end-users to use EUD-MARS easily. Hence, we also assessed the chosen visual paradigm (interlocking blocks) of our language against the recommendations of the cognitive dimensions framework [101]. Furthermore, we conducted an efficiency evaluation that we report in the Appendix.

6.1. Variety of robots used

We evaluated the technique and tool presented in Sections 3 to 5 with the various robot types shown in Table 1. We built two of these robots ourselves using the Lego Mindstorms [102] robotics kit. We called the first Lego Mindstorms robot DriverBot because it can just move around on its three wheels. We called the second one ShooterBot because, in addition to moving around, it can shoot plastic pellets. The ShooterBot robot has two engines that move its tracks and one engine for operating the shooting mechanism. Both Lego Mindstorms robots have ultrasonic sensors that detect proximity from obstacles. The ShooterBot is also equipped with a sensor that detects colors. The third robot we used is the iRobot Create [103] programmable vacuum cleaner. This robot can perform various actions such as moving, spinning, and playing music. The fourth robot is the NAO [7] humanoid that can perform human-like actions such as walking, talking, and sitting. NAO is equipped with a wide variety of sensors for detecting battery level, collision, posture, and so on. The fifth robot is a Parrot Bebop 2 [104] drone. The Bebop 2 is also equipped with several sensors, and it can perform typical drone actions such as taking off, moving around, and landing.

The Lego Mindstorms, iRobot Create, and Parrot Bebop 2 were controlled using C# APIs. This is a natural selection since we used C# to develop the desktop-based interpreter discussed in Section 5. However, APIs are not necessarily limited to C#. For example, the manufacturer of NAO, SoftBank Robotics, does not provide .NET (e.g., C#) implementations for the recent versions of its API. The most common NAO API uses Python. Hence, we developed a C# API that internally calls the Python API provided by the NAO manufacturer. This bridges our tool with NAO's API, thereby allowing end-user programmers to control NAO robots using EUD-MARS.

6.2. Choice of paradigm for the visual language

One of the reasons for using Blockly to develop the visual language of EUD-MARS is the maturity of this library and its web-based nature. Another important criterion is also the usability of Blockly's visual paradigm that constitutes interlocking blocks. A similar type of blocks, namely jigsaw puzzles, proved to be understandable by end-users in a previous work that we did on end-user development for IoT [100]. In that work, we assessed our puzzle notation based on the cognitive dimensions framework and with end-users. Here, we briefly explain how the Blockly-based visual language of EUD-MARS also adheres to the criteria of the cognitive dimensions framework.

We can say that the EUD-MARS language is *consistent* since it provides common visual blocks that represent concepts like action sequences and execution permissions. Once users learn how to use these blocks, they can apply their knowledge to different situations. Concerning *diffuseness*, each type of visual element has a particular color and shape. The reusability of action sequences reduces the number of blocks required in a program. This can improve a program's *visibility*. Furthermore, the Blockly-based visual blocks are comparable in size to the blocks used in existing visual development environments like Scratch. It is possible to perform a *progressive evaluation* of a program by calling each action sequence separately and observing its outcome on the target robots. This is similar to the progressive evaluation that software developers perform with code-based programming. In terms of the *abstraction gradient*, EUD-MARS is *abstraction-tolerant* since end-users can develop programs using the provided visual elements. Nonetheless, end-users can create new elements using action sequences. Then, they can call these new elements in their programs in a way that is similar to how they call the elements that EUD-MARS provides out of the box. The adopted visual block notation helps end-users to avoid possible "spaghetti" programs that might result from box-and-line notations that some existing systems use (refer to Section 2). It is possible to add comments as a *secondary notation* on any visual block. Once added, comments appear in the form of a clickable question mark that can bring up a text box for editing a comment's content.

7. Evaluation with software developers

Although EUD-MARS mainly targets end-users, software developers play an important role in setting up the robots before end-users can start programming. Hence, we conducted a study with software developers whereby they defined robot profiles and API configurations using EUD-MARS, and then they offered insights on our approach's strengths and points for improvement. This evaluation is important to elicit the feedback of software developers on the technique for setting-up new robot types and laying the groundwork for end-users to program model-driven adaptive robotics software systems. The following subsections report on this study's participants, design, and results.

7.1. Participants

This study had eight participants including two females and six males aged 23 to 31 (Mean=26.62, SD=2.87). The participants are actively working as software developers either in companies or as freelancers.

The participants have between 3 and 11 (Mean=5.75, SD=2.76) years of experience in the software industry. Their experience collectively included the development of various types of software applications including business applications and mobile games.

7.2. Design of the study

Each participant completed this study in around one and a half hours. Before the study, we introduced the participants to EUD-MARS and explained the role of software developers in creating robot profiles and API configurations. We also gave them an example of how to define profiles and configurations.

After this preparation, we asked the participants to use both the XML-based language and the visual tool to define the *DrivingRobot* profile of the *LegoNXTShooterBotAPI* shown in Fig. 8 and Fig. 9 respectively. We asked the participants to provide feedback on what they thought were the main strengths and the points that we can improve.

7.3. Results

The participants specified the following points as the main strengths of the language and tool.

Base profiles (refer to Fig. 8) **are very useful**. Since robots could share many features, using base profiles saves time and improves maintainability by allowing a robot profile to inherit the features of another. Base profiles allow software developers to think about abstraction when preparing robot profiles, the way they do when designing and programming software applications.

The **ability to browse robot profiles and API configurations visually in the project explorer** offers familiarity when compared to traditional software development tools. This makes it possible to easily search and browse items. Furthermore, the visual style of the tree views, for example in the project explorer, is appealing and makes it easier to locate and select items. We should note that the style we adopted for this tool's UI follows the material design guidelines, and uses the Material Design In XAML Toolkit [105]. This gives the tree nodes a wide and flat visual appearance.

The **ability to load API data from existing libraries** reduces the manual work needed for identifying the classes and functions in these libraries by using a separate tool.

The **ability to generate robot profiles and API configurations from API data** is very useful even if some manual adjustments might be required. The initially generated profiles and configurations are a good starting point. Performing some manual adjustments on the generated data is easier than starting from scratch. Similarly, **the feature that enables automatic mapping of robot profiles to API configurations saves time**, even if it only finds partial matches.

In addition to highlighting the main strengths, the participants provided insights on points for improvement. What follows is an explanation of these points and suggestions on ways for potentially addressing them.

It would be useful to **have a mixed-mode view that combines the visual trees and the XML-based representation**. Generating the XML representation of the profiles and configurations from the visual tree saves time. Nonetheless, being able to swap between the two views would make this tool more powerful. This suggested feature is similar to what IDEs support in UI development. For example, IDEs usually combine a visual UI design tool with a UI development markup language such as HTML. In the robot-profile-composition tool, a mixed-mode view would allow software developers to use the XML mode when they become more familiar with it. The participants considered this especially useful for operations that they deem simpler to do using the markup language. One example of these operations is duplicating a robot profile and performing minor changes on the copy. Software developers would be able to duplicate a profile by copying and pasting its XML representation, and then they can quickly move through the copy to perform the necessary changes. The software developers' tool is already able to transform the visual representation to its XML counterpart. Furthermore, the end-user development tool is already able to interpret the XML representation of the robot profiles and APIs, to enable the definition and execution of robotics programs. Combining both interpreters in the software develop-

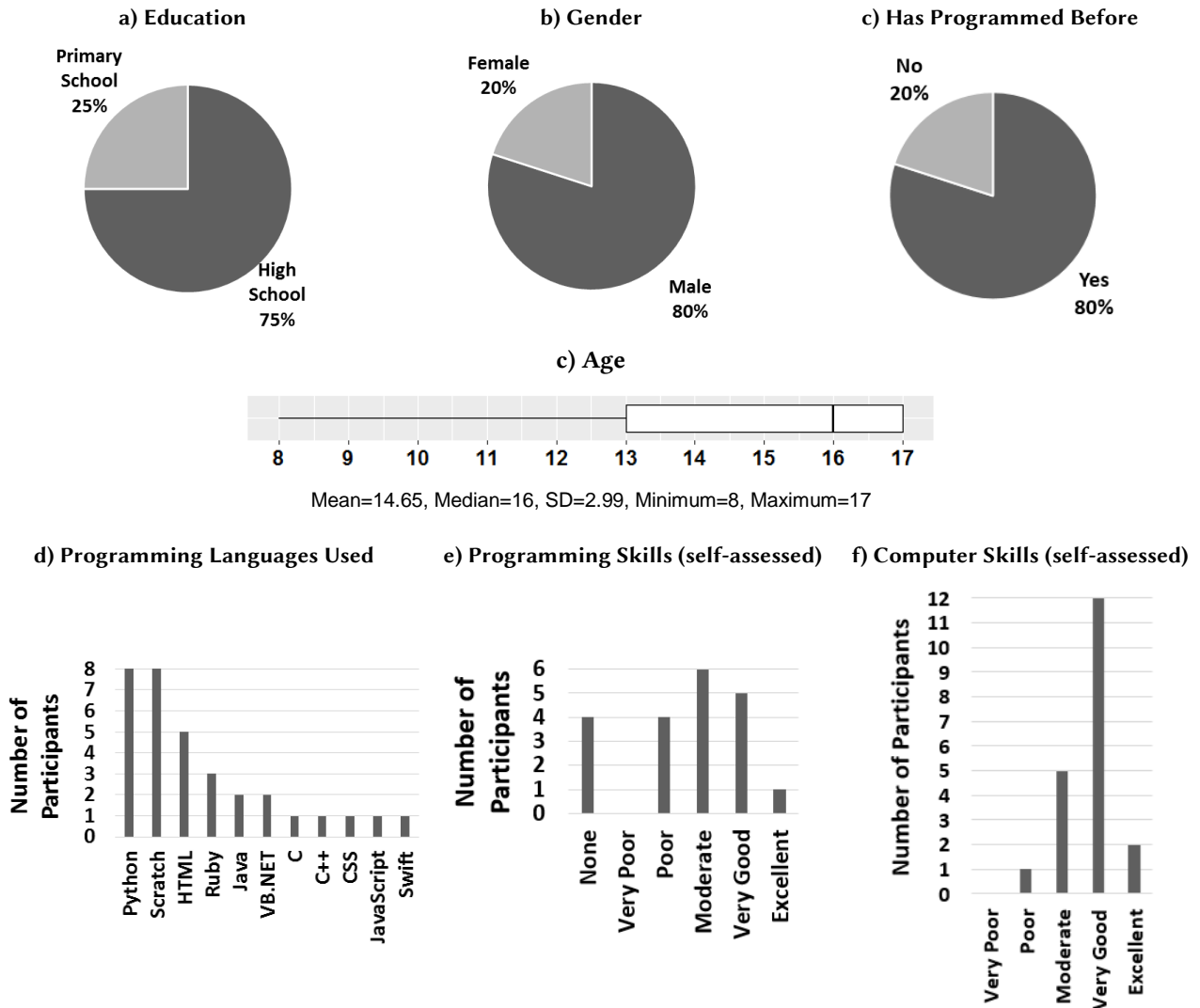


Fig. 15. Participants' education, gender, age, knowledge of programming, and computer skills.

ers' tool would realize this suggested feature through a form of bidirectional transformation between the visual representation and its XML counterpart.

It would be useful to **have a validation feature** that allows software developers to check if their API configurations still conform to updated versions of API libraries. It would also be useful if the tool **reflects or suggests API library updates automatically**. Considering that API libraries evolve, having these two features would save software developers the time required to manually check and apply updates. These features would also make the process of updating API configurations less error-prone. The tool is already able to load API definitions from their respective libraries. We just need to implement a comparison mechanism between the latest API definitions and the previously loaded ones. The automated comparison mechanism could involve prompts for obtaining the software developers' decisions on operations such as deleting and renaming items (e.g., classes and methods).

It would be useful if the tool could **generate source code skeletons** from the robot profiles and API configurations, even though this is not its primary purpose. Typically, software developers would have one or more API libraries and then they would define robot profiles and configurations to conform to these APIs. However, there are cases where the software developers are starting a new API library project. In such cases, it would be useful if they could use this tool to define the structure of the API, and then have the tool generate the skeleton of the API classes and methods in a target programming language. This is not the primary intention of this tool. However, it is possible to realize this feature by implementing an extensible code generation engine. This engine could incorporate transformation rules to convert robot profiles to different target programming languages. By doing so, the tool would allow software developers to use both API-first and profile-first approaches. Some IDEs support a similar feature for code and databases, whereby software de-

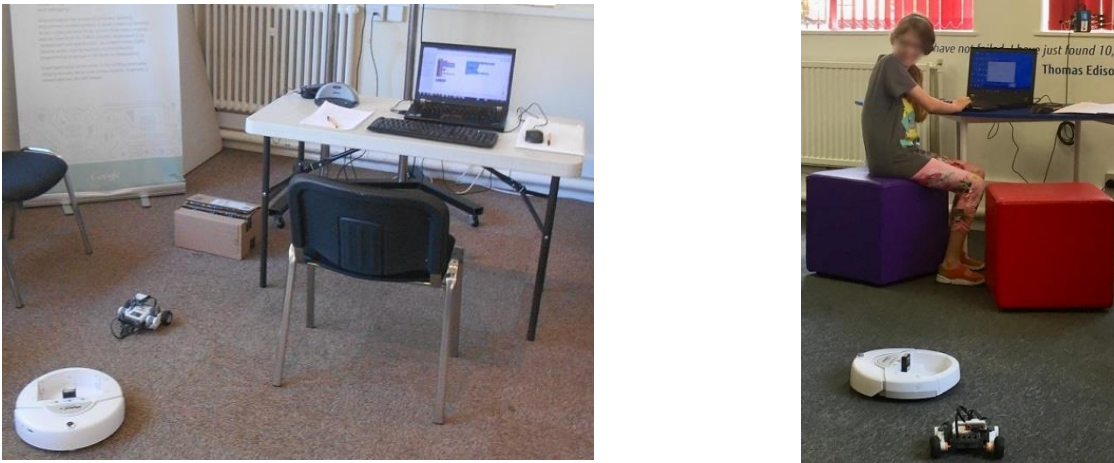


Fig. 16. Example settings from the study.

velopers can write the code first and generate a database based on the classes or design a database first and generate the code from it.

It would be useful to have **the ability to refer to copies of the API library files (assemblies) from the project explorer**. Software developers would be able to publish the copies directly with the robot profiles and API configurations for the end-user software-development tool to use. This also prevents changes to the external copies of API libraries from having any effect, until the software developers decide to upgrade to a new API version. This is simply realizable by copying the API libraries into the local directory of a robot profile project and having the tool refer to these copies rather than the original ones.

7.4. Threat to validity

This study involved a small number of participants. This could limit the generalizability of the results. However, these participants provided rich feedback from their perspectives as software developers, and they gave insights that showed the strengths and the points that we can improve. In this study, we primarily aimed to collect feedback from software developers on EUD-MARS after they have used it in a development scenario. We consider the results insightful and informative for creating a future version of the tool, which we can assess further in another study.

8. Evaluation with end-users

Since EUD-MARS mainly targets end-users, we conducted a study to assess how end-users perceive this approach's understandability, ease of use, and overall desirability. The evaluation based on these three metrics shows the ability of EUD-MARS to empower end-users to program model-driven adaptive robotics software systems. Assessing the understandability of the programs is important to see if EUD-MARS was able to provide end-users with a less technical and understandable representation of its model-driven and adaptation capabilities. Assessing the ease-of-use shows how easily end-users can use the visual language to develop programs that incorporate these capabilities. The assessment of desirability provides end-users with the means of expressing their opinion of the entire system using a set of descriptive keywords. The following subsections present the participants' background information, and the study's design and results.



8.1. Participants

We conducted this study with 20 participants. We recruited ten of these participants during a technology-related community outreach event that occurred within the National Museum of Computing at Bletchley Park in the United Kingdom. The event lasted for two days during which the participants volunteered to take part in this study and other related computer programming activities. We recruited five of the participants during a programming workshop for school students that took place at Notre Dame University–Louaize in Lebanon. We recruited the remaining five participants through personal contacts in Lebanon.

We should note that we conducted the study in English. The British participants are all native speakers. The Lebanese participants are proficient in English because they either are learning all their school courses in English or at least are studying English as one of two foreign languages that Lebanese schools commonly teach (the other language is French).

Question 1 (Action Sequences)

Assume that you have the following robots:

Humanoid
Drone

What do you think the following program does?

```


graph TD
    A[Perform Greeting] --> B[Humanoid]
    B --> C[Speak "Hello"]
    C --> D[Wave Right Arm]
    D --> E[Drone]
    E --> F[Fly Up]
    F --> G[Roll Left]
    G --> H[Roll Right]
    I[Perform Greeting] --> J[Humanoid]
    K[Perform Greeting] --> L[Drone]
  
```

How easy do you think this program is to understand?

Very Hard ☐ ☐ ☐ ☐ ☐ Very Easy

Question 2 (Adaptation)

Assume that you have the following robot:



Rover

What do you think the following program does?

```

graph TD
    A[When Distance from Object Rover < 10] --> B[Prevent Execution]
    B --> C[Execute Before Prevent]
    C --> D[Stop Rover]
    D --> E[Prevent]
    E --> F[Move Forward Rover]
  
```

How easy do you think this program is to understand?

Very Hard ☐ ☐ ☐ ☐ ☐ Very Easy

Fig. 17. Questions related to the understandability of EUD-MARS programs.

The participants consisted of children and teenagers, whose ages ranged between 8 and 17 years. They are primary school (25%) and high school (75%) students (Fig. 15a). The participants were 20% female and 80% male (Fig. 15b). Concerning their programming knowledge, 80% of the participants stated that they have previously done some form of programming (Fig. 15c). The participants who have previously programmed indicated that they have worked with (at least tried) one or more of the programming languages shown in Fig. 15d. They also did a self-assessment of their programming skills and computer skills, as shown in Fig. 15e and Fig. 15f respectively. Most of the participants had programming skills that ranged from poor to excellent according to their self-assessment. Four participants said that they had no prior programming experience. However, as we previously mentioned, we recruited most of the participants during technology-related community outreach events. These events included general introductory programming lessons and activities in Python and C#. Hence, the participants who had no prior programming experience gained a general understanding of programming and some experience during the events, before participating in the study. Only one of the four participants who did not have prior programming experience did not attend the technology-related events, because he was one of the five participants recruited through personal contacts. The events also helped the participants who had previously programmed to emphasize their knowledge of programming. We should also note here that considering the participants' age range, their knowledge of programming comes mostly from introductory school sessions or workshops that target school students. Therefore, we can say that the participants have some knowledge of programming, but they are not professional programmers.

Some existing research work targets young learners [106,107], while others target both children and less technical adults [108]. We should note that EUD-MARS does not only target children and teenagers; it targets any end-users who are not professional programmers but are interested in programming robots. The participants of this study are children

Involved Robots



Lego Mindstorms (Driver Bot)

iRobot Create (Vacuum Cleaner)
labeled Roomba because the name is more common

Question 1 (Action Sequences)

Scenario

Add a virtual action sequence and name it “Dance”.

The Roomba’s dance moves are as follows:

- Spin
- Wait for 10 seconds
- Stop

The Driver Bot’s dance moves are as follows:

- Move forward
- Wait for 2 seconds
- Move backward
- Wait for 2 seconds
- Stop

How easy do you think this scenario is to program?
Very Hard ○ ○ ○ ○ ○ **Very Easy**

Expected Output

The visual representation shows a green 'Dance' block containing two sub-sequences. The first sub-sequence is for the 'Roomba' and includes a 'Spin' block, a 'Wait (Time = 10 Seconds)' block, and a 'Stop' block. The second sub-sequence is for the 'Driver Bot' and includes a 'Move Forward (Speed = 100)' block, a 'Wait (Time = 2 Seconds)' block, a 'Move Backward (Speed = 100)' block, another 'Wait (Time = 2 Seconds)' block, and a 'Stop' block. To the right, there are two small icons: a green 'Dance' block with 'Roomba' and a blue 'Dance' block with 'Driver Bot'.

Question 2 (Adaptation)

Scenario

Add a “When” event.

Assume that when the Driver Bot’s sensor “Distance from Object” provides a value that is < 40, you want to perform the following:

- Stop the Driver Bot
- Spin the Roomba
- Prevent the Driver Bot from Moving Forward

How easy do you think this scenario is to program?
Very Hard ○ ○ ○ ○ ○ **Very Easy**

Expected Output

The visual representation shows a 'When' block with the condition 'Distance from Object | Driver Bot < 40'. Below this, there are three sections: 'Prevent Execution' (empty), 'Execute Before Prevent' (containing 'Stop | Driver Bot' and 'Spin | Roomba'), and 'Prevent' (containing 'Move Forward | Driver Bot').

Fig. 18. Questions related to programming with EUD-MARS.

and teenagers who are not professional programmers but have an interest in programming. Hence, we consider these participants to be a representative end-user group.

8.2. Design of the study

First, we asked the participants to provide some background information about themselves. Then, we gave an introductory explanation of EUD-MARS. Finally, the participants carried out tasks related to EUD-MARS and answered feedback questions. We present in Fig. 16 two examples of this study’s settings.

The introductory explanation that we gave to the participants included example programs developed using EUD-MARS. These programs introduced the participants to the way EUD-MARS works and the visual elements that are available in its toolbox. This explanation also helped those with little or no programming experience to understand more about visual programming. The introduction took on average 15 minutes per participant. Afterward, each participant took on average 20 minutes to complete the rest of this study that involved carrying out tasks and providing feedback. We did not

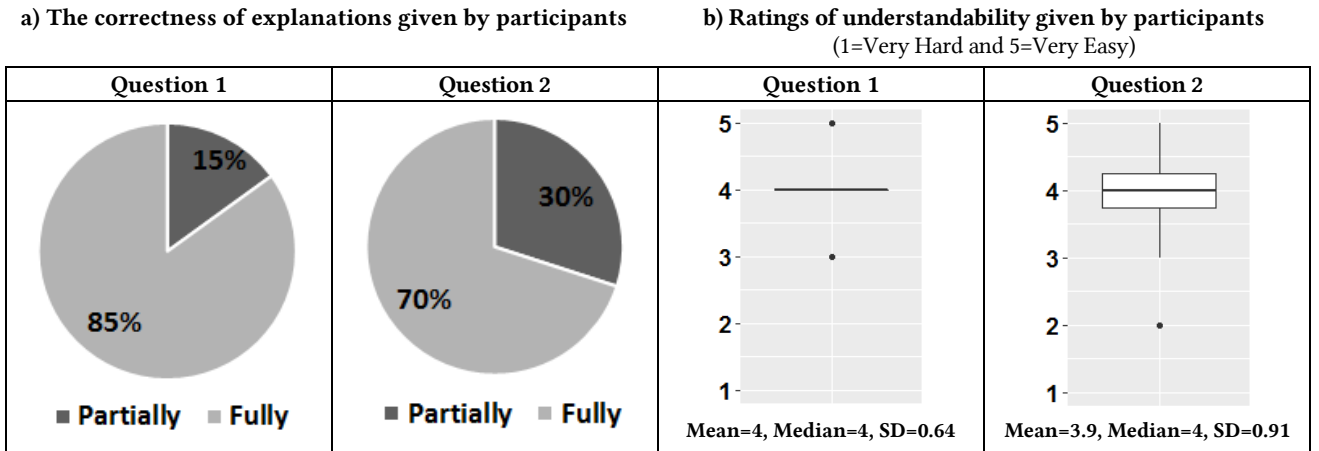


Fig. 19. Results of the end-user evaluation: correctness of explanations of programs and ratings of understandability.

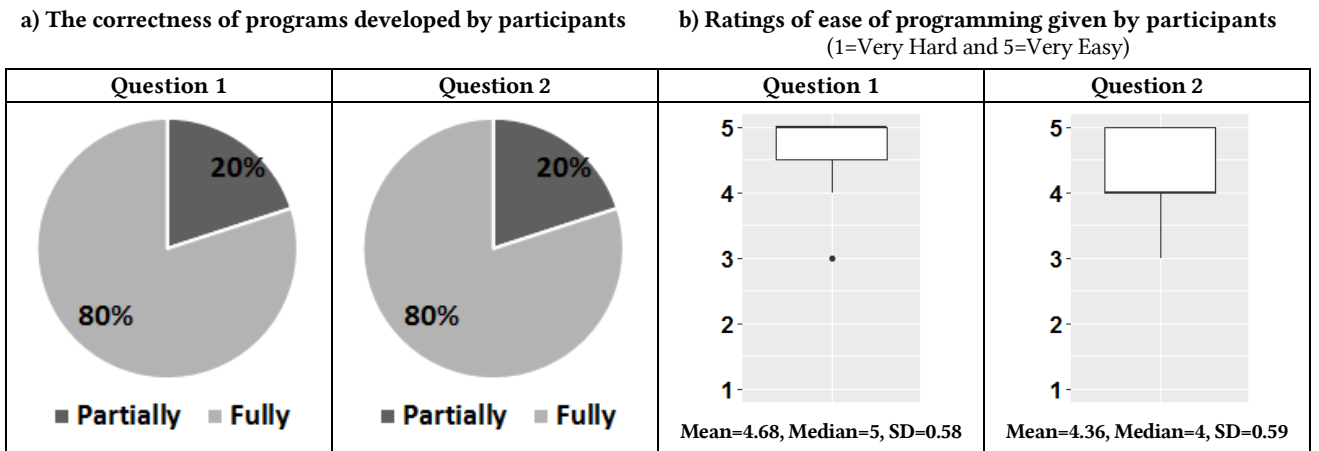


Fig. 20. Results of the end-user evaluation: correctness of developed programs and ratings of ease of programming.

set a maximum time that a participant can spend on a task. Since the participants were not familiar with EUD-MARS, the introductory explanation assisted them in completing the given tasks.

The first part of this study involved asking the participants to read two visual programs and explain what they think each of these programs does, as shown in Fig. 21. One of these programs demonstrated different types of action sequences and calls to action sequences (Fig. 21-Question 1), while the other program demonstrated adaptive behavior based on a change in the context of use (Fig. 21-Question 2). The participants rated, on a five-point scale, how easy it was to understand the two visual programs.

The second part of this study involved asking the participants to develop two programs using the visual programming environment of EUD-MARS, as shown in Fig. 22. The first program involved using different types of action sequences and actions to implement a dance functionality on two types of robots (Fig. 22-Question 1). The second program involved adapting a robot's behavior when its proximity from nearby objects is less than a given value (Fig. 22-Question 2). Upon detecting the proximity change, another robot is also prompted to execute an action; this is a form of communication among robots (refer to Section 3.5). The participants also rated, on a five-point scale, how easy it was to develop the two programs.

We recorded the participants' explanations and the programs that they developed to determine to what extent they were able to answer the questions. Finally, we asked the participants to choose three product reaction cards (PRCs) [109] that they thought best described EUD-MARS. We gave the participants a subset of 10 positive and 10 negative PRCs from the original 118 terms. The positive PRCs were as follows: appealing, desirable, easy-to-use, effective, exciting, friendly, fun, straightforward, useful, and valuable. The negative PRCs were as follows: boring, confusing, difficult, hard-to-use, intimidating, rigid, too-technical, unapproachable, unattractive, and undesirable. We gave the participants the choice of selecting PRCs that are positive, negative, or a mixture of both.

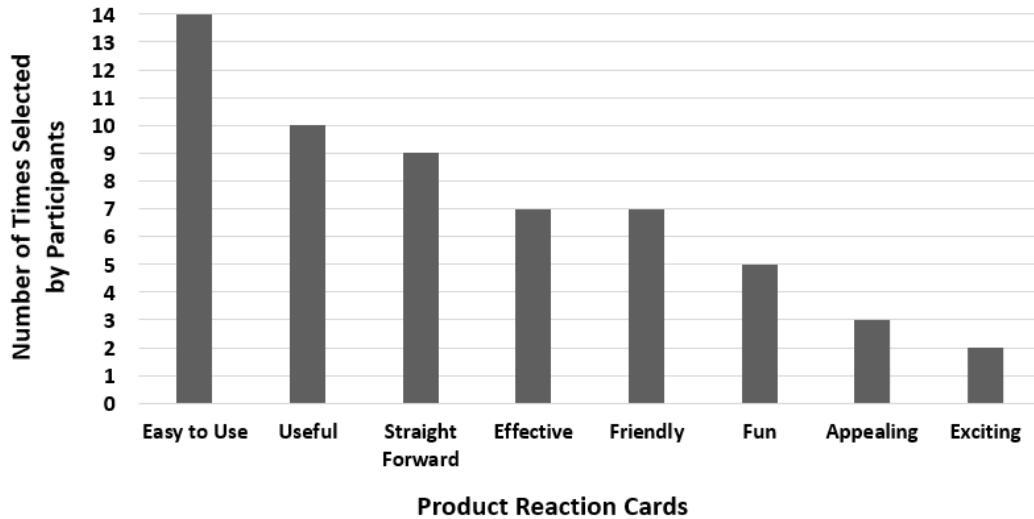


Fig. 21. Product reaction cards selected by the participants to describe EUD-MARS.

8.3. Results: Understandability, ease of programming, and overall desirability

This study showed positive results in terms of EUD-MARS' understandability, ease of use for programming, and overall desirability to people who are not professional programmers.

As Fig. 19a shows, the majority of the participants were able to explain fully the two EUD-MARS programs that we presented to them. The first (Fig. 17-Question 1) and the second (Fig. 17-Question 2) programs were explained in full by 85% and 70% of the participants respectively. The remaining participants provided a partial explanation of what the programs do. We decided on whether an answer fully or partially explains a program by comparing the participants' explanations to ones that we wrote and discussed before conducting the study. Our explanations covered all the concepts in the programs that we presented to the participants. Upon examining the participants' answers, we discussed whether their explanations also covered these concepts and therefore fully explained the programs, or partially explained the programs due to concepts missing from the explanation. What follows are examples of the most commonly missed concepts that led us to classify some explanations as partial. In question one, the participants who gave a partial explanation mostly missed the calls to the "Perform Greeting" action sequence with a humanoid and a drone as parameters. On the other hand, the participants who partially explained question two mostly missed the purpose of the "Prevent" section under the "Prevent Execution" block. As Fig. 19b shows, the average rating that the participants gave on the understandability of the two programs was 4/5, indicating that the programs are easy to understand.

Concerning the programming tasks (Fig. 18-Questions 1 and 2), as Fig. 20a shows, 80% of the participants were able to complete fully both tasks. We also decided on whether a participant's program fully answers a question, by comparing it to a program that we developed beforehand and assessing whether it fulfills the given requirements. The participants rated the ease of programming of both tasks to be on average higher than 4/5 (Fig. 20b). This indicates that they found the tasks to be easy to accomplish using EUD-MARS.

We can say that the rate of successful answers that the participants gave in the explanation and programming parts complies with their perception of EUD-MARS' understandability and ease of programming. We can see that in both the explanation and programming parts, the participants rated the adaptation questions as slightly less understandable and slightly harder to program. Nonetheless, the overall averages of both parts were comparable and positive.

The PRCs selected by the participants to describe the overall desirability of EUD-MARS were all positive as shown in Fig. 21. The top three most selected PRCs are as follows: easy to use, useful, and straightforward. These PRCs provide a positive indication of the participants' perceived usability and utility (usefulness) of EUD-MARS.

8.4. Results: Observations and participant suggestions

One of the things we observed during the study is that when composing an event (When) with a condition on a context element, e.g., a robot's distance from an object, several participants attached the context element block directly to the event block. They needed to attach a condition block to the event block, and then add the context element to the condition block. The EUD-MARS programming tool does not allow its end-user to attached incompatible blocks together, and it

provides warning messages to explain the reason. Nonetheless, to clarify things further for the end-users, we aim to add a faded placeholder (i.e., shadow block) next to the event block to indicate that a condition block is required there. We perceive the placeholder to be a solution because during the study the participants understood the meaning of the faded placeholder blocks that existed next to the action blocks.

The participants were overall keen on entering correct parameter values for the selected robot actions, even when the instructions did not specify what values to enter. For example, the instructions indicated a time value when instructing the participants to add a time delay (Wait) block. However, the instructions did not provide explicit values for the speed parameter of the MoveForward and MoveBackward actions. Yet, the majority of the participants noticed this parameter and were proactive in asking what range of values they can use. Afterward, they chose a speed that they considered appropriate. This indicates that the parameters are obvious enough for the end-users.

Although the participants had no prior experience with the EUD-MARS programming tool, many of them were able to work with it quite comfortably. For example, they navigated quickly through the toolbox. They copied, pasted, and deleted blocks when required, and even managed to call the action sequences that they added. This ease of work is likely due to the prior experience of some participants had with Scratch. This experience likely gave the participants an element of familiarity with the style of the blocks and the overall mode of interaction with the tool. This is one of the advantages of choosing a common visual notation, namely interlocking blocks, to develop the visual language of EUD-MARS.

In the first programming question, one participant added the required actual action sequences for the driver bot and the vacuum cleaner under two separate virtual action sequences. He used the word “Dance”, which was given in the instructions, to name both virtual action sequences. We aim to help end-users avoid this issue in the future by adding validation that would suggest merging virtual action sequences that have the same name.

One participant suggested adding the word “if” next to the combo box that shows the list of robots in actual action sequences. He mentioned that seeing “if”, e.g., “if Driver Bot”, makes the purpose of the actual action sequence more understandable. Adding “if” as a label could be considered in the future, but it would require evaluation to check if end-users will confuse it with the standard “if” condition. Another participant suggested adding text next to events (When) to clarify that a sensor block is required. We will address this issue by adding a faded placeholder next to events.

8.5. Threats to validity

The understandability and programming questions that we gave the participants cover part of the functionality supported by EUD-MARS. Nonetheless, these questions cover both major parts of EUD-MARS that involve supporting the development of adaptive software and the use of model-driven development as an underlying approach, while hiding the complexity from end-users. We aimed to obtain an indication about the understandability of EUD-MARS’ visual language, whether this language and its supporting tool are easy to use for programming and EUD-MARS’ overall desirability. The positive results that the study yielded indicate that end-users will appreciate the remaining functionality since it follows the same paradigm. If more learning time is dedicated, e.g., through lectures or video tutorials, we assume that end-users would be able to use EUD-MARS for developing programs that are more complex.

The study involved 20 participants; this could limit the generalizability of the results. However, the sample of participants was diverse in terms of the participants’ ages (8 to 17 years), programming skills (none to excellent), and nationalities (British and Lebanese).

9. Conclusions and future work

This paper presented an approach called End-User Development of Model-driven Adaptive Robotics Software (EUD-MARS). This approach allows end-users, who are not professional programmers, to develop model-driven adaptive robotics software without requiring advanced technical skills. Model-driven development enables programs to run on a variety of robot types and hardware platforms. Adaptation capabilities enable robots to change their behavior based on changes in the context of use (user, platform, and environment).

In the EUD-MARS approach, software developers are responsible for preparing robot profiles that enable end-users to program a set of robots. The preparation of these profiles involves specifying what actions and sensors robots support and configuring code-based APIs to connect them to EUD-MARS. We provided software developers with an XML-based language and a visual tool, which they can use to prepare robot profiles and API configurations. We created a visual language, based on Blockly, which enables end-users to develop robotics software without requiring a significant amount of technical knowledge. We developed a tool that enables end-users to develop robotics programs using our visual language and to execute their programs on robots.

We evaluated EUD-MARS from a technical perspective by using it to control a variety of robots. These robots included a driver bot and a shooter bot that we developed using Lego Mindstorms, a vacuum cleaner (iRobot Create), a humanoid (NAO), and a drone (Parrot Bebop 2). We also assessed the choice of interlocking blocks (Blockly) as a visual notation based on the recommendations of the cognitive dimensions framework.

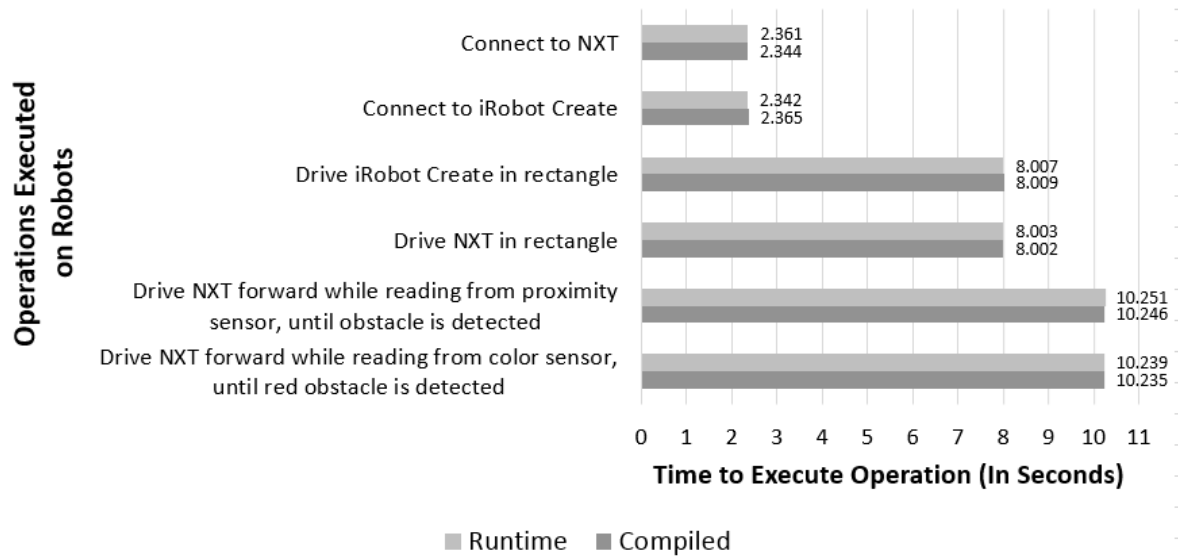


Fig. 22. Efficiency evaluation: comparing between runtime and compiled code execution times.

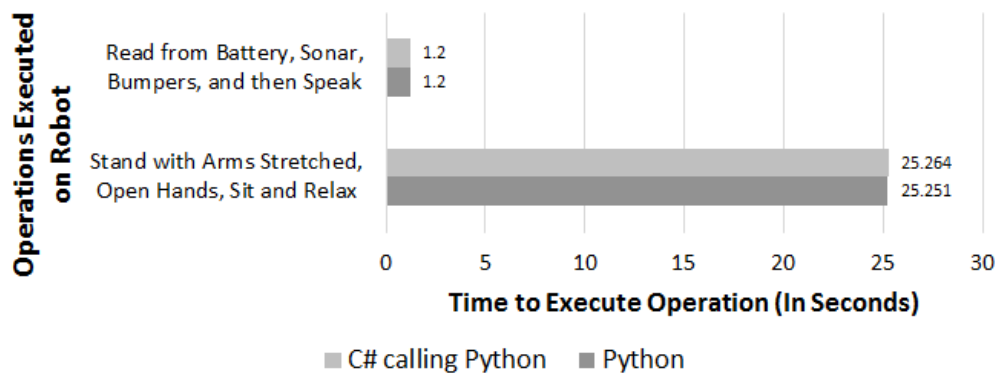


Fig. 23. Efficiency evaluation: comparing direct calls from Python and C# calling Python when no C# API is available (e.g., the case of the NAO robot).

We conducted a study with software developers to elicit their feedback on the XML-based language and visual tool that we created for defining robot profiles and API configurations. We asked the participants to use both the language and the tool to define a robot profile and API mappings. Then, we asked them to provide feedback on what they thought were the main strengths and the points that we can improve. The participants stated several perceived points of strength in our approach, and they offered insights for improvement that could inform future work.

We conducted a study with end-users to assess mainly their ability to explain and develop EUD-MARS programs. The participants were primary school and high school students from the United Kingdom and Lebanon. We asked the participants to explain EUD-MARS programs and to use the tool that we created to develop programs. We also asked them to answer questions about their perception of EUD-MARS' understandability, ease of use for programming, and overall desirability. The study yielded positive results concerning the participants' ability to explain and develop EUD-MARS programs, and their overall perception of our approach.

As mentioned in the paper, we noted observations and participant suggestions that we could use for future improvements. Furthermore, in the future, it would be interesting to research conflict prevention in real-world situations when two or more robots are collaborating to fulfill a task. We could also conduct further research on how to use the capabilities of heterogeneous robots to solve a complex task collaboratively. It will also be interesting to extend EUD-MARS' end-user programming tool by adding an advanced three-dimensional simulator that could support a wide variety of robots.

Acknowledgments

The authors would like to thank the financial support by the European Research Council (ERC), EU Horizon 2020, and the Engineering and Physical Sciences Research Council (EPSRC) that enabled the collaboration. We also acknowledge the eSTeEM project at the School of Computing and Communications, Open University, and the National Museum of Computing at Bletchley Park for supporting the engagement with schoolchildren in the United Kingdom.

Appendix

Efficiency evaluation

As Fig. 1 shows, an EUD-MARS interpreter reads the configurations and programs and executes the instructions on the designated robots at runtime. Hence, this evaluation involved comparing the efficiency of interpreted programs (runtime execution) with that of compiled code. This evaluation also involved testing calls made from the EUD-MARS program interpreter (written in C#) to an API written in another language (Python). The evaluation was carried out on a computer that has an Intel Core i5 2.50 GHz CPU, 12 GB of RAM, and is running Windows 7.

We developed the interpreter using C# and invoked API calls at runtime using reflection. The latter allows the retrieval of meta-data from assemblies and types. It also supports the dynamic creation of instances of objects and the invocation of methods. The efficiency evaluation checks if the use of runtime interpretation and execution would have any significant performance implications. The chart presented in Fig. 22 compares operations that we executed on different robots in two ways. One way involved executing the operations from a compiled C# program, which made function calls to each robot's API. Another way involved executing the operations from the EUD-MARS environment, which interpreted the visual program and transformed it into API calls at runtime. Here, we should note that the numbers include the time to complete the whole operation, e.g., physical movement by the robot, and not just the time for sending the command. The operations include both the execution of actions and reading from sensors. We can see that the efficiency of the EUD-MARS programs is almost the same as that of the compiled programs. This indicates that the dynamic interpretation and execution of the visual programs do not degrade performance.

As mentioned in Section 6.1, we developed a C# API that is capable of calling Python code to command the NAO robot. We compared the efficiency of calling the Python code from C# and calling it directly without C#. The chart presented in Fig. 23 compares two sets of operations that we executed twice, once from a C# API that calls external Python functions and another time by calling the Python functions directly. We can see that calling the external Python functions from the C# API does not add any significant overhead. Here, we should note that software developers could typically call Python from C# using the Iron Python language, which makes Python programs compatible with the .NET framework. They can also call the Python process from C# and pass as a parameter the path of the code file that they wish to execute. Although the first method is easier to use, the second method allows software developers to call recent versions of Python that Iron Python might not support. Hence, we used the second method to build the NAO robot API for EUD-MARS. If in the future we use web-services for API calls (refer to Section 3.7), the overhead would just be the HTTP call.

References

- [1] S.J. Kim, Y. Jeong, S. Park, K. Ryu, G. Oh, A Survey of Drone use for Entertainment and AVR (Augmented and Virtual Reality), in: T. Jung, M.C. tom Dieck (Eds.), *Augment. Real. Virtual Real. Empower. Hum. Place Bus.*, Springer International Publishing, Cham, 2018: pp. 339–352. https://doi.org/10.1007/978-3-319-64027-3_23.
- [2] Microsoft, Power Apps, (2019). <https://docs.microsoft.com/en-us/powerapps/powerapps-overview>.
- [3] M. Burnett, T. Kulesza, End-User Development in Internet of Things: We the People, in: *Proc. Workshop End User Dev. Internet Things Era*, ACM, Seoul, Korea, 2015: pp. 81–86.
- [4] M. Burnett, C. Cook, G. Rothermel, End-User Software Engineering, *Commun. ACM.* 47 (2004) 53–58.
- [5] A.J. Ko, R. Abraham, M.M. Burnett, B.A. Myers, Guest editors' introduction: End-user software engineering, *IEEE Softw.* 26 (2009) 16–17.
- [6] Blockly, (n.d.). <https://developers.google.com/blockly> (accessed October 28, 2019).
- [7] NAO, (n.d.). <https://www.softbankrobotics.com/emea/en/robots/nao> (accessed October 28, 2019).
- [8] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, others, The State of the Art in End-User Software Engineering, *ACM Comput. Surv. CSUR.* 43 (2011) 21:1–21:44.
- [9] H. Lieberman, F. Paternò, M. Klann, V. Wulf, End-User Development: An Emerging Paradigm, in: *End User Dev.*, Springer, 2006: pp. 1–8.
- [10] G. Biggs, B. Macdonald, A Survey of Robot Programming Systems, in: *Proc. Australas. Conf. Robot. Autom.* CSIRO, 2003: p. 27.
- [11] A. Elkady, T. Sobh, Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography, *J. Robot.* 2012 (2012) 15.
- [12] N. Mohamed, J. Al-Jaroodi, I. Jawhar, A review of middleware for networked robots, *Int. J. Comput. Sci. Netw. Secur.* 9 (2009) 139–148.

- [13] M. Namoshe, N.S. Tlale, C.M. Kumile, G. Bright, Open middleware for robotics, in: 2008 15th Int. Conf. Mechatron. Mach. Vis. Pract., 2008: pp. 189–194. <https://doi.org/10.1109/MMVIP.2008.4749531>.
- [14] R. France, B. Rumpe, Model-Driven Development of Complex Software: A Research Roadmap, in: Proc. Workshop Future Softw. Eng., IEEE, Minneapolis, USA, 2007: pp. 37–54. <https://doi.org/10.1109/FOSE.2007.14>.
- [15] A.R. da Silva, Model-driven engineering: A survey supported by the unified conceptual model, *Comput. Lang. Syst. Struct.* 43 (2015) 139–155. <https://doi.org/10.1016/j.cl.2015.06.001>.
- [16] P.A. Akiki, A.K. Bandara, Y. Yu, Adaptive Model-Driven User Interface Development Systems, *ACM Comput. Surv.* 47 (2014) 64:1–64:33.
- [17] M.C. Huebscher, J.A. McCann, A Survey of Autonomic Computing—Degrees, Models, and Applications, *ACM Comput. Surv.* 40 (2008) 7:1–7:28. <https://doi.org/10.1145/1380584.1380585>.
- [18] M. Salehie, L. Tahvildari, Self-Adaptive Software: Landscape and Research Challenges, *ACM Trans. Auton. Adapt. Syst.* 4 (2009) 1–42. <https://doi.org/10.1145/1516533.1516538>.
- [19] B. Athreya, F. Bahmani, A. Diede, C. Scaffidi, End-user programmers on the loose: A study of programming on the phone for the phone, in: Vis. Lang. Hum.-Centric Comput. VLHCC 2012 IEEE Symp. On, IEEE, 2012: pp. 75–82.
- [20] W.P. Dann, S. Cooper, R. Pausch, Learning to Program with Alice, Pearson, 2011.
- [21] D.D. Hoang, H.-Y. Paik, A.H.H. Ngu, Spreadsheet as a Generic Purpose Mashup Development Environment, in: P.P. Maglio, M. Weske, J. Yang, M. Fantinato (Eds.), *Serv.-Oriented Comput.* 8th Int. Conf. ICSOC 2010 San Franc. CA USA Dec. 7–10 2010 Proc., Springer Berlin Heidelberg, Berlin, Heidelberg, 2010: pp. 273–287. http://dx.doi.org/10.1007/978-3-642-17358-5_19.
- [22] J. Lin, J. Wong, J. Nichols, A. Cypher, T.A. Lau, End-user Programming of Mashups with Vegemite, in: Proc. 14th Int. Conf. Intell. User Interfaces, ACM, New York, NY, USA, 2009: pp. 97–106. <https://doi.org/10.1145/1502650.1502667>.
- [23] J. Jackson, Microsoft robotics studio: A technical introduction, *IEEE Robot. Autom. Mag.* 14 (2007) 82–87.
- [24] M. Bell, J. FLOYD, J.F. Kelly, LEGO MINDSTORMS EV3, Springer, 2017.
- [25] J. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, The Scratch Programming Language and Environment, *ACM Trans. Comput. Educ. TOCE.* 10 (2010) 16:1–16:15.
- [26] Edbot, (n.d.). <http://ed.bot/edbot> (accessed October 22, 2019).
- [27] RoboBlockly, (n.d.). <http://roboblockly.ucdavis.edu> (accessed October 22, 2019).
- [28] D. Weintrop, D.C. Shepherd, P. Francis, D. Franklin, Blockly goes to work: Block-based programming for industrial robots, in: Blocks Workshop BB 2017 IEEE, IEEE, 2017: pp. 29–36.
- [29] R. Bischoff, A. Kazi, M. Seyfarth, The MORPHA style guide for icon-based programming, in: Proc. 11th IEEE Int. Workshop Robot Hum. Interact. Commun., 2002: pp. 482–487. <https://doi.org/10.1109/ROMAN.2002.1045668>.
- [30] H. Qichen, D. Li, Ardublock, 2014. <https://sourceforge.net/projects/ardublock>.
- [31] Modkit LLC., Modkit, (n.d.). <http://www.modkit.com> (accessed January 30, 2016).
- [32] G. Kortuem, A.K. Bandara, N. Smith, M. Richards, M. Petre, Educating the Internet-of-Things generation, *Computer.* 46 (2013) 53–61.
- [33] IBM Emerging Technology Services, Node-RED, (2013). <https://nodered.org/about> (accessed October 22, 2019).
- [34] P. van Allen, NETLab Toolkit, (2003). <http://www.netlabtoolkit.org> (accessed October 23, 2019).
- [35] J. Huang, M. Cakmak, Code3: A System for End-to-End Programming of Mobile Manipulator Robots for Novices and Experts, in: 2017 12th ACMIEEE Int. Conf. Hum.-Robot Interact. HRI, 2017: pp. 453–462.
- [36] S. Alexandrova, M. Cakmak, K. Hsiao, L. Takayama, Robot Programming by Demonstration with Interactive Action Visualizations, in: *Robot. Sci. Syst.*, 2014: pp. 48–56.
- [37] J. Huang, T. Lau, M. Cakmak, Design and evaluation of a rapid programming system for service robots, in: 2016 11th ACMIEEE Int. Conf. Hum.-Robot Interact. HRI, 2016: pp. 295–302. <https://doi.org/10.1109/HRI.2016.7451765>.
- [38] E. Pot, J. Monceaux, R. Gelin, B. Maisonnier, Choregraphe: a graphical tool for humanoid robot programming, in: RO-MAN 2009 - 18th IEEE Int. Symp. Robot Hum. Interact. Commun., 2009: pp. 46–51. <https://doi.org/10.1109/ROMAN.2009.5326209>.
- [39] A. Sappé, B. Mutlu, Design Patterns for Exploring and Prototyping Human-robot Interactions, in: Proc. 32Nd Annu. ACM Conf. Hum. Factors Comput. Syst., ACM, New York, NY, USA, 2014: pp. 1439–1448. <https://doi.org/10.1145/2556288.2557057>.
- [40] T. Lourens, TiViPE - Tino's Visual Programming Environment, in: Proc. 28th Annu. Int. Comput. Softw. Appl. Conf. 2004 COMPSAC 2004, 2004: pp. 10–15 vol.1. <https://doi.org/10.1109/CMPSAC.2004.1342799>.
- [41] T. Lourens, E. Barakova, User-Friendly Robot Environment for Creation of Social Scenarios, in: J.M. Ferrández, J.R. Álvarez Sánchez, F. de la Paz, F.J. Toledo (Eds.), *Found. Nat. Artif. Comput.*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011: pp. 212–221.
- [42] M. Stenmark, M. Haage, E.A. Topp, Simplified Programming of Re-usable Skills on a Safe Industrial Robot: Prototype and Evaluation, in: Proc. 2017 ACMIEEE Int. Conf. Hum.-Robot Interact., ACM, New York, NY, USA, 2017: pp. 463–472. <https://doi.org/10.1145/2909824.3020227>.
- [43] ROBOTC, (n.d.). <http://www.robotc.net> (accessed October 22, 2019).
- [44] D.D. Hils, Visual languages and computing survey: Data flow visual programming languages, *J. Vis. Lang. Comput.* 3 (1992) 69–101. [https://doi.org/10.1016/1045-926X\(92\)90034-J](https://doi.org/10.1016/1045-926X(92)90034-J).
- [45] S. Alexandrova, Z. Tatlock, M. Cakmak, RoboFlow: A flow-based visual programming language for mobile manipulation tasks, in: 2015 IEEE Int. Conf. Robot. Autom. ICRA, 2015: pp. 5537–5544. <https://doi.org/10.1109/ICRA.2015.7139973>.
- [46] D. Glas, S. Satake, T. Kanda, N. Hagita, An interaction design framework for social robots, in: *Robot. Sci. Syst.*, 2012: p. 89.
- [47] D.F. Glas, T. Kanda, H. Ishiguro, Human-robot interaction design using Interaction Composer eight years of lessons learned, in: 2016 11th ACMIEEE Int. Conf. Hum.-Robot Interact. HRI, 2016: pp. 303–310. <https://doi.org/10.1109/HRI.2016.7451766>.
- [48] N. Arne, H. Nico, W. Dennis, W. Sebastian, A survey on domain-specific modeling and languages in robotics, (2016).
- [49] OMG Robotics Domain Task Force, (n.d.). <https://www.omg.org/robotics> (accessed October 22, 2019).
- [50] X. Blanc, J. Delatour, T. Ziadi, Benefits of the MDE approach for the development of embedded and robotic systems, in: Proc. 2nd Natl. Workshop “Control Archit. Robots Models Exec. Distrib. Control Archit. CAR, 2007.

- [51] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, D. Brugali, The BRICS component model: a model-based development paradigm for complex robotics software systems, in: *Proc. 28th Annu. ACM Symp. Appl. Comput.*, ACM, 2013: pp. 1758–1764.
- [52] D.S. Kolovos, R.F. Paige, F.A. Polack, The epsilon transformation language, in: *Int. Conf. Theory Pract. Model Transform.*, Springer, 2008: pp. 46–60.
- [53] Eclipse Modeling Framework, (n.d.). <http://www.eclipse.org/modeling/emf> (accessed October 22, 2019).
- [54] S. Dhoub, S. Kchir, S. Stinckwich, T. Ziadi, M. Ziane, Robotml, a domain-specific language to design, simulate and deploy robotic applications, in: *Int. Conf. Simul. Model. Program. Auton. Robots*, Springer, 2012: pp. 149–160.
- [55] C. Schlegel, A. Steck, D. Brugali, A. Knoll, Design abstraction and processes in robotics: From code-driven to model-driven engineering, in: *Int. Conf. Simul. Model. Program. Auton. Robots*, Springer, 2010: pp. 324–335.
- [56] C. Schlegel, A. Steck, A. Lotz, Robotic software systems: From code-driven to model-driven software development, in: *Robot. Syst.-Appl. Control Program.*, InTech, 2012.
- [57] A. Diego, V.-C. Cristina, O. Francisco, P. Juan, Á. Bárbara, V3cmm: A 3-view component meta-model for model-driven robotic software development, *J. Softw. Eng. Robot.* 1 (2010) 3–17.
- [58] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, *Sci. Comput. Program.* 72 (2008) 31–39.
- [59] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.
- [60] RobMoSys, (n.d.). <https://robmosys.eu> (accessed May 20, 2020).
- [61] R. Heckel, M. Lohmann, Towards Model-Driven Testing, *Electron. Notes Theor. Comput. Sci.* 82 (2003) 33–43. [https://doi.org/10.1016/S1571-0661\(04\)81023-5](https://doi.org/10.1016/S1571-0661(04)81023-5).
- [62] M. Mussa, S. Ouchani, W.A. Sammane, A. Hamou-Lhadj, A Survey of Model-Driven Testing Techniques, in: *2009 Ninth Int. Conf. Qual. Softw.*, 2009: pp. 167–172. <https://doi.org/10.1109/QSIC.2009.30>.
- [63] P.A. Akiki, CHAIN: Developing model-driven contextual help for adaptive user interfaces, *J. Syst. Softw.* 135 (2018) 165–190. <https://doi.org/10.1016/j.jss.2017.10.017>.
- [64] P.A. Akiki, A.K. Bandara, Y. Yu, Engineering Adaptive Model-Driven User Interfaces, *IEEE Trans. Softw. Eng.* 42 (2016) 1118–1147. <https://doi.org/10.1109/TSE.2016.2553035>.
- [65] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, A. Wortmann, A New Skill Based Robot Programming Language Using UML/P Statecharts, in: *2013 IEEE Int. Conf. Robot. Autom.*, 2013: pp. 461–466.
- [66] K. Adam, A. Butting, R. Heim, O. Kautz, B. Rumpe, A. Wortmann, Model-Driven Separation of Concerns for Service Robotics, in: *Proc. Int. Workshop Domain-Specif. Model.*, Association for Computing Machinery, New York, NY, USA, 2016: pp. 22–27. <https://doi.org/10.1145/3023147.3023151>.
- [67] D. Bozhinoski, D.D. Ruscio, I. Malavolta, P. Pelliccione, M. Tivoli, FLYAQ: Enabling Non-expert Users to Specify and Generate Missions of Autonomous Multicopters, in: *2015 30th IEEEACM Int. Conf. Autom. Softw. Eng. ASE*, 2015: pp. 801–806.
- [68] D. Di Ruscio, I. Malavolta, P. Pelliccione, Engineering a Platform for Mission Planning of Autonomous and Resilient Quadrotors, in: A. Gorbenco, A. Romanovsky, V. Kharchenko (Eds.), *Softw. Eng. Resilient Syst.*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013: pp. 33–47.
- [69] D. Di Ruscio, I. Malavolta, P. Pelliccione, A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems., in: *MORSE STAF*, 2014: pp. 16–29.
- [70] F. Ciccozzi, D.D. Ruscio, I. Malavolta, P. Pelliccione, Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems, *IEEE Access.* 4 (2016) 6451–6466.
- [71] IBM, An Architectural Blueprint for Autonomic Computing, (2006). <http://bit.ly/MapeKLoop> (accessed April 5, 2013).
- [72] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, *Computer.* 37 (2004) 46–54. <https://doi.org/10.1109/MC.2004.175>.
- [73] J. Kramer, J. Magee, Self-Managed Systems: an Architectural Challenge, in: *Proc. Workshop Future Softw. Eng.*, IEEE, Minneapolis, USA, 2007: pp. 259–268. <https://doi.org/10.1109/FOSE.2007.19>.
- [74] D. Bozhinoski, A. Bucchiarone, I. Malavolta, A. Marconi, P. Pelliccione, Leveraging Collective Run-Time Adaptation for UAV-Based Systems, in: *2016 42th Euromicro Conf. Softw. Eng. Adv. Appl. SEAA*, 2016: pp. 214–221.
- [75] S. Dragule, B. Meyers, P. Pelliccione, A generated property specification language for resilient multirobot missions, in: *Int. Workshop Softw. Eng. Resilient Syst.*, Springer, 2017: pp. 45–61.
- [76] IFTTT (If This Then That), (n.d.). <https://ifttt.com> (accessed May 20, 2020).
- [77] D. Brugali, P. Scandurra, Component-based robotic engineering (part i)[tutorial], *IEEE Robot. Autom. Mag.* 16 (2009) 84–96.
- [78] D. Brugali, A. Shakhimardanov, Component-based robotic engineering (part ii), *IEEE Robot. Autom. Mag.* 17 (2010) 100–112.
- [79] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, ROS: an open-source Robot Operating System, in: *ICRA Workshop Open Source Softw.*, Kobe, Japan, 2009: p. 5.
- [80] Robot Operating System (ROS), *Robot Oper. Syst. ROS.* (n.d.). <https://www.ros.org> (accessed October 13, 2019).
- [81] G. Bardaro, A. Semperebon, M. Matteucci, A Use Case in Model-based Robot Development Using AADL and ROS, in: *Proc. 1st Int. Workshop Robot. Softw. Eng.*, ACM, New York, NY, USA, 2018: pp. 9–16. <https://doi.org/10.1145/3196558.3196560>.
- [82] Y. Hua, S. Zander, M. Bordignon, B. Hein, From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning, in: *2016 IEEE 21st Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, 2016: pp. 1–8. <https://doi.org/10.1109/ETFA.2016.7733579>.
- [83] M. Wenger, W. Eisenmenger, G. Neugschwandtner, B. Schneider, A. Zötl, A model based engineering tool for ROS component compositioning, configuration and generation of deployment information, in: *2016 IEEE 21st Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, 2016: pp. 1–8. <https://doi.org/10.1109/ETFA.2016.7733559>.
- [84] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, C.-H. Lee, OPRoS: A New Component-Based Robot Software Platform, *ETRI J.* 32 (2010) 646–656.

- [85] B. Song, S. Jung, C. Jang, S. Kim, An introduction to robot component model for opros (open platform for robotic services), in: Proc. Int. Conf. Simul. Model. Program. Auton. Robots Workshop, 2008: pp. 592–603.
- [86] H. Utz, S. Sablatnog, S. Enderle, G. Kraetzschmar, Miro - middleware for mobile robot applications, IEEE Trans. Robot. Autom. 18 (2002) 493–497. <https://doi.org/10.1109/TRA.2002.802930>.
- [87] H. Bruyninckx, Open robot control software: the OROCOS project, in: Robot. Autom. 2001 Proc. 2001 ICRA IEEE Int. Conf. On, IEEE, 2001: pp. 2523–2528.
- [88] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, Woo-Keun Yoon, RT-middleware: distributed component middleware for RT (robot technology), in: 2005 IEEE/RSJ Int. Conf. Intell. Robots Syst., 2005: pp. 3933–3938. <https://doi.org/10.1109/IROS.2005.1545521>.
- [89] H. Chishiro, Y. Fujita, A. Takeda, Y. Kojima, K. Funaoka, S. Kato, N. Yamasaki, Extended RT-Component Framework for RT-Middleware, in: 2009 IEEE Int. Symp. ObjectComponentService-Oriented Real-Time Distrib. Comput., 2009: pp. 161–168. <https://doi.org/10.1109/ISORC.2009.40>.
- [90] A. Makarenko, A. Brooks, T. Kaupp, Orca: Components for robotics, in: Int. Conf. Intell. Robots Syst. IROS, 2006: pp. 163–168.
- [91] T.H. Collett, B.A. MacDonald, B.P. Gerkey, Player 2.0: Toward a practical robot programming framework, in: Proc. Australas. Conf. Robot. Autom. ACRA 2005, Citeseer Citeseer, 2005: p. 145.
- [92] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, J. Vanderdonckt, A Unifying Reference Framework for Multi-Target User Interfaces, Interact. Comput. 15 (2003) 289–308. [https://doi.org/10.1016/S0953-5438\(03\)00010-9](https://doi.org/10.1016/S0953-5438(03)00010-9).
- [93] A. Bennaceur, T.T. Tun, A.K. Bandara, Y. Yu, B. Nuseibeh, Feature-driven Mediator Synthesis: Supporting Collaborative Security in the Internet of Things, ACM Trans. Cyber-Phys. Syst. 2 (2018) 21.
- [94] E. Pasternak, R. Fenichel, A.N. Marshall, Tips for Creating a Block language with Blockly, in: Blocks Workshop BB 2017 IEEE, IEEE, 2017: pp. 21–24.
- [95] S. Yamashita, M. Tsunoda, T. Yokogawa, Visual Programming Language for Model Checkers Based on Google Blockly, in: Int. Conf. Prod.-Focus. Softw. Process Improv., Springer, 2017: pp. 597–601.
- [96] A. Marron, G. Weiss, G. Wiener, A decentralized approach for programming interactive applications with javascript and blockly, in: Proc. 2nd Ed. Program. Syst. Lang. Appl. Based Actors Agents Decentralized Control Abstr., ACM, 2012: pp. 59–70.
- [97] P. Bottoni, M. Ceriani, Linked Data Queries as Jigsaw Puzzles: a Visual Interface for SPARQL Based on Blockly Library, in: Proc. 11th Biannu. Conf. Ital. SIGCHI Chapter, ACM, 2015: pp. 86–89.
- [98] J. Danado, F. Paternò, Puzzle: A mobile application development environment using a jigsaw metaphor, J. Vis. Lang. Comput. 25 (2014) 297–315.
- [99] J. Humble, A. Crabtree, T. Hemmings, K.-P. Åkesson, B. Koleva, T. Rodden, P. Hansson, “Playing with the Bits” User-Configuration of Ubiquitous Domestic Environments, in: A.K. Dey, A. Schmidt, J.F. McCarthy (Eds.), UbiComp 2003 Ubiquitous Comput. 5th Int. Conf. Seattle WA USA Oct. 12–15 2003 Proc., Springer Berlin Heidelberg, Berlin, Heidelberg, 2003: pp. 256–263. http://dx.doi.org/10.1007/978-3-540-39653-6_20.
- [100] P.A. Akiki, A.K. Bandara, Y. Yu, Visual Simple Transformations: Empowering End-Users to Wire Internet of Things Objects, ACM Trans Comput-Hum Interact. 24 (2017) 10:1–10:43. <https://doi.org/10.1145/3057857>.
- [101] T.R.G. Green, M. Petre, Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework, J. Vis. Lang. Comput. 7 (1996) 131–174.
- [102] Lego Mindstorms, (n.d.). <https://www.lego.com/en-gb/mindstorms> (accessed October 28, 2019).
- [103] iRobot Create, (n.d.). <https://www.irobot.com/about-irobot/stem/create-2.aspx> (accessed October 28, 2019).
- [104] Parrot Bebop 2, (n.d.). <https://www.parrot.com/uk/drones/parrot-bebop-2> (accessed October 28, 2019).
- [105] Material Design In XAML Toolkit, (n.d.). <http://materialdesigninxaml.net> (accessed August 6, 2019).
- [106] M. Seraj, C.S. Große, S. Autexier, R. Drechsler, Smart Homes Programming: Development and Evaluation of an Educational Programming Application for Young Learners, in: Proc. 18th ACM Int. Conf. Interact. Des. Child., ACM, New York, NY, USA, 2019: pp. 146–152. <https://doi.org/10.1145/3311927.3323157>.
- [107] A. Zimmermann-Niefield, M. Turner, B. Murphy, S.K. Kane, R.B. Shapiro, Youth Learning Machine Learning Through Building Models of Athletic Moves, in: Proc. 18th ACM Int. Conf. Interact. Des. Child., ACM, New York, NY, USA, 2019: pp. 121–132. <https://doi.org/10.1145/3311927.3323139>.
- [108] A. Millner, E. Baafi, Modkit: Blending and Extending Approachable Platforms for Creating Computer Programs and Interactive Objects, in: Proc. 10th Int. Conf. Interact. Des. Child., ACM, New York, NY, USA, 2011: pp. 250–253. <https://doi.org/10.1145/1999030.1999074>.
- [109] J. Benedek, T. Miner, Measuring Desirability: New Methods for Evaluating Desirability in a Usability Lab Setting, Proc. Usability Prof. Assoc. 2003 (2002) 8–12.
- [110] K. Anis, ROS As a Service: Web Services for Robot Operating System, J. Softw. Eng. Robot. 6 (2015) 1–14.
- [111] F.L. Keppmann, M. Maleshkova, A. Harth, Building REST APIs for the Robot Operating System-Mapping Concepts, Interaction., in: SALAD ESWC, 2015: pp. 10–19.