The Journal of Systems & Software 195 (2023) 111547

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Assessing industrial end-user programming of robotic production cells: A controlled experiment☆

Christoph Mayr-Dorn [a,*], Mario Winterer [b], Christian Salomon [b], Doris Hohensinger [b], Harald Fürschuss [c]

[a] Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria
[b] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
[c] ENGEL Austria GmbH, Schwertberg, Austria

ARTICLE INFO

Article history:
Received 21 March 2022
Received in revised form 5 July 2022
Accepted 12 October 2022
Available online 20 October 2022

Keywords:
Robot programming
End-user programming
Manufacturing automation
Block-based programming languages
Sequential function charts

ABSTRACT

Adapting the behavior of robots and their interaction with other machines on the shop floor is typically accomplished by non-programmers. Often these non-programmers use visual languages to specify the robot's and/or machine's control logic. While visual languages are explored as a means to enable novices to program, there is little understanding of what problems novices face when tasked with realistic adaptation programming tasks on the shop floor. In this paper, we report the results of a controlled experiment where domain experts in the injection molding industry inspected and changed realistic programs involving a robot, injection molding machine, and additional external machines. We found that participants were comparably quick to understand the program behavior with a familiar sequential function chart-based language and a Blockly-based language used for the first time. We also observed that these non-programmers had difficulty in multiple aspects independent of language due to the interweaving of physical and software-centric interaction between robot and machine. We conclude that assistance needs to go beyond optimizing available language elements to include suggesting relevant programming elements and their sequence.

1. Introduction

Modern shop floors are highly automated. To this end, robots are one key mechanism to ensuring the material flow between production stations and/or execute manufacturing steps. For example, robots pick molded parts from an injection molding machine and place them on a conveyor belt.

These configurations of machines, robots, and behavior do not remain stable. Innovation, new products, and changing customer demand force these shop floors to change as well. Product traceability, for example, might require that molded parts need to be identifiable which can be accomplished by laser engraving the individual parts before packaging, thus requiring the robot to hand over a part to an engraving station and only then place it on a conveyor belt.

Implementing such changes typically needs expertise in the machines and robots that are part of the shop floor production cell. Often the domain experts that adapt the robot and machine behavior are not programmers. While end-user-centric programming languages aim to make programming more accessible to non-programmers, it is yet little understood what aspects end-users find challenging in a realistic, industrial production context.

In this paper, we aim to reduce this lack of understanding by studying in a controlled experiment how domain experts adapt the behavior of a realistic production cell, involving the interaction between an injection molding machine (IMM), robot, and other auxiliary machines. Specifically, we are interested in which aspects an end-user has problems with, and what kind of assistance the end user would require to execute a non-trivial program editing task.

In our controlled experiment, we tasked eleven domain experts to understand and edit a program in two visual languages: Blockly and EPL (a proprietary sequential flowchart-based language used in the injection molding industry). We found that while non-programmers find it easy to understand a program regardless of the language, they find it equally difficult in either language to correctly define all required robot moving and gripping commands as well as the robot's interaction with external machines.

Given that the participants never used Blockly before, we can confirm prior studies that Blockly is easy to understand by novices. On the other hand, we infer from our observations

☆ Editor: Prof Raffaela Mirandola.

* Corresponding author.
E-mail addresses: christoph.mayr-dorn@jku.at (C. Mayr-Dorn), mario.winterer@scch.at (M. Winterer), christian.salomon@scch.at (C. Salomon), doris.hohensinger@scch.at (D. Hohensinger).

https://doi.org/10.1016/j.jss.2022.111547

that visual languages by themselves provide too little support to enable non-programmers to implement complex, realistic tasks without additional assistance.

The remainder of this paper is structured as follows: We provide a short introduction to block-based programming languages in Section 2 together with a description of the industry context. Section 3 details the study design including research questions and experiment procedure. We show experiment results in Section 4, list participant feedback on usability aspects in Section 5 before discussing the results and their implications in Section 6. We review related work in Section 7 before concluding this paper with an outlook on future work in Section 8.

2. Background

2.1. Block-based programming languages

A block-based programming language is a type of visual programming language. It uses blocks to represent statements, i.e. the atomic conceptual elements of a programming language, in contrast to text-based languages where statements are mapped to words. Usually, an instruction is expressed by a block representation that has a specific shape and color-code related to its type. Blocks also contain a describing text and/or an icon as well as optional editable fields to allow users to provide additional input. Most blocks have characteristic dents or nobs (following the metaphor of puzzle pieces) that provide visual clues to the user about where matching blocks can be connected to combine elements to syntactically correct programs. Furthermore, the resulting programs appear as larger blocks themselves, containing groups of aligned (nested) blocks from which they are compiled.

Modern block-based programming editors offer support for drag-and-drop of blocks and for snapping matching blocks together. Individual blocks can be picked from a palette and inserted into a program by dropping them on another block where it will snap in place if the dents/nobs of the blocks match. This feature helps to intuitively explain coding concepts and to facilitate access to programming for novice users. According to Bau et al. (2017), block-based programming languages are advantageous over conventional textual languages with respect to learnability due to the following reasons:

- Programming languages usually require learning the programming vocabulary. However, blocks rely on recognition — not on recall, since blocks can be picked from palettes and need not be remembered. Additionally, the listing of all block types helps the user to become familiar with language elements and to maintain an overview of system components.
- Programming causes a high cognitive load, in particular for new users. This is reduced because block-based programs typically are structured into smaller and easily recognizable pieces.
- In contrast to conventional programming approaches, syntax errors can be avoided in block-based programming. The related environment prevents the user from connecting mismatching blocks when assembling elements to programs.

One example block-based language environment is Blockly. Specifically, it is an open-source JavaScript library for building block-based programming editors for the web, mainly developed by Google.[1] Blockly defines the general graphical syntax and provides some basic language blocks out of the box. It also has an API to define additional custom language elements easily. Another important part of the Blockly library is its ability to generate source code out of Blockly programs. Therefore, it provides extendable code generators for JavaScript, Python, Dart, Lua, and PHP.

We chose Blockly over other Block-based languages due to it being well documented, supported, extensible, intensively studied where multiple paper demonstrate its effectiveness when used by beginners/non-programmers especially in the domain of robot control (see also Section 7. Especially, Blockly has also been used as the basis for programming collaborative robots, e.g., ABB's CoBlox (Weintrop et al., 2017) or in the Niryo.One Studio.[2] Due to its accessibility, extensibility, and large community support numerous popular block-based programming environments, like MakeCode (Ball et al., 2019) and Scratch are based on the Blockly library. Other well-investigated languages such as Snap! are more targeted at children, hence, we deemed them less appropriate for application in an industrial setting.

2.2. Industry context

ENGEL is a world leader in manufacturing injection molding machines (IMM) used across many industry domains like consumer electronics, automotive, avionics, food industry, etc. for producing a huge variety of different plastic parts. ENGEL also offers several industrial robots – Cartesian coordinate robots as well as multi-axes articulated robots – which are usually delivered together with the machine as a *production cell* that can be integrated into larger production lines.

The possible tasks of these robots are manifold. The simplest tasks include removing the molded parts from the machinery and placing them on a conveyor belt, or picking up supplied parts and inserting them into the machinery to be included in the molded product (e.g. the metal part of a screwdriver). But there are also more complex scenarios, for example, production processes with multiple molding steps, in which semi-finished parts are removed, temporarily stacked and then re-inserted into the machinery. In several scenarios, collaborating robots are applied together in the same production cell, which needs coordinating interactions with each other and the machinery.

To address this need, ENGEL machines include an end-user programming environment called *Sequence Editor* to adapt workflows that are described using the *ENGEL Programming Language* (EPL), a visual programming language that is based on flow charts (see Fig. 1 for an excerpt). Flow-chart based programming languages are common in industrial environments, for example, icon-based flow charts like *MORPHA* (Bischoff et al., 2002) or sequential function charts as one of the five languages described in the IEC 61131-3 standard for PLC programming (Lewis and Lewis, 1998).

The Sequence Editor is used by a wide range of technicians from well-trained maintenance engineers to novice factory attendants. As most users have either no or only limited programming experience, they rely on proven, predefined robot programs that are provided by ENGEL.

Fig. 1 shows a screenshot of the Sequence Editor with a preset robot program. Each element in the program represents an instruction statement of the underlying robot control system and is visualized as icon with descriptive text. Clicking or touching an element opens a dialog to parameterize the selected element, for example, arguments of a subroutine call, edit the condition expression of a conditional statement in the condition editor (see Fig. 2), and so on. The toolbox offers language elements – including all flow control and machinery instructions – that can be used for programming.

[1] https://developers.google.com/blockly.

[2] https://niryo.com/2020/10/learn-robotics-with-our-ecosystem/.

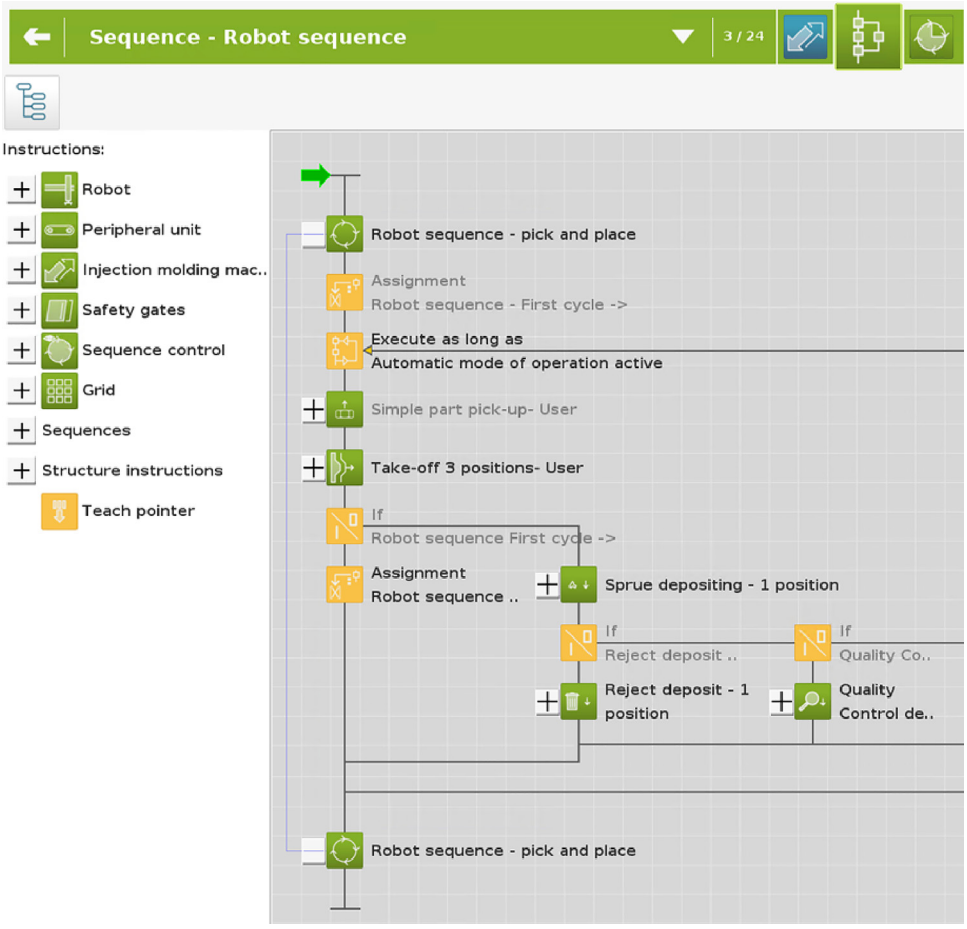

Fig. 1. ENGEL Sequence Editor with a toolbox on the left side (white background) and the EPL program on the right (gray background).

Creating new programs or adapting existing ones due to changing product or cell layout takes considerable time for multiple reasons. First, execution of such programs in a real environment takes considerable time as robot movements need to happen at a safe speed until the user is certain the robot movements are correct. While a virtual environment exists for the injection molding machine and the robot, this environment cannot visualize the physical interaction between robot and machine (i.e., collisions, etc.) and cannot simulate any non-integrated machines such as conveyor belts. Hence the virtual environment is not suitable for providing feedback of robot behavior that involves coordination with multiple shop floor participants. Program refinement hence involves multiple iterations of trial running the program at slow speeds to be able to abort it at any time, cumbersome setting up the correct physical pre-conditions such as available (raw) products to various positions (e.g., in the form or on the conveyor belt) to assure that the program behaves correctly in various situations and then adjustment of the program.

Although, we describe this activity by domain experts as "programming", often it is described as "configuring" in the industry to avoid implying the need of having a programmer involved. Most often, the goal is to extend and adapt existing programs rather than writing them from scratch to obtain results quickly and as the basic behavior remains the same. From scratch development is more common for highly customized production cells with rarely seen robot/machine combinations. The programming activities for such cases are then done by experts with a strong programming background and not conducted by the domain experts at the center of this work.

3. Study design

In highly automated, industrial cells, domain experts that need to adapt not only the behavior of individual machines or robots but also their interactions are often not programmers.

The *goal* of this study is to observe to what extent such non-programmers are able to understand and edit realistic production cell programs that involve the interaction of multiple machines and robots.

The *motivation* behind this study is to determine what aspects non-programmers find challenging during their tasks, the programming constructs that are unclear or lead to confusion, and the level of support needed to complete a task.

The *perspective* is of researchers and practitioners at machine vendors to take these insights as a basis for developing appropriate support mechanisms for the non-programmer stakeholder group.

The *context* of this study comprises eleven engineers tasked with determining the interaction between an injection molding machine, robot, and external machine with two different visual programming languages in the scope of a controlled experiment.

3.1. Research questions

In this work, we investigate the following research questions:

RQ1: ***What are the errors non-programmers make when programming interactions among machines and robots?***

Rationale: programming the interaction between machines and robots on the shop floor typically involves aspects such as signaling, waiting for things to happen, moving, or gripping. We are

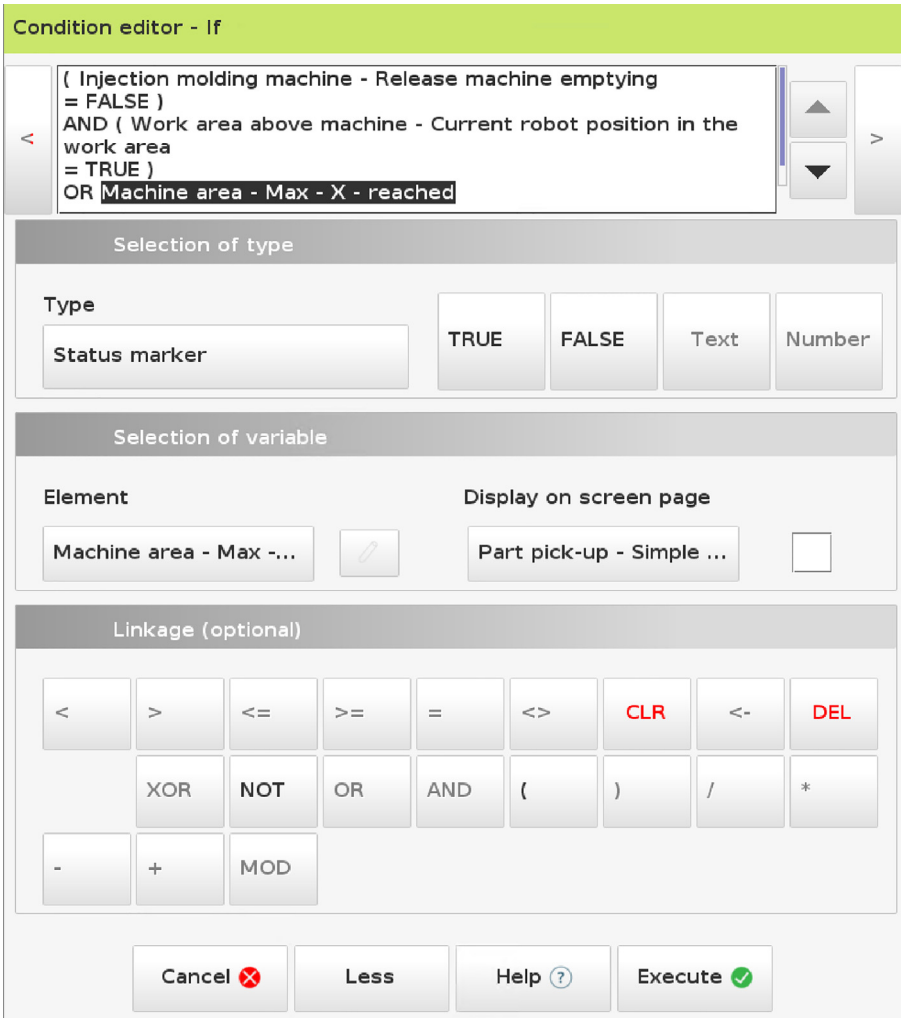

Fig. 2. ENGEL condition editor.

interested in better understanding whether any of these elements are more error-prone than others and thus might show potential for dedicated support mechanisms.

RQ2: ***What level of assistance do non-programmers need?***

Rationale: we want to understand for a typical, realistic programming task, whether the programming language elements and current IDE support are sufficient to quickly and correctly produce programs or whether additional support such as warnings of missing elements, etc. would be helpful.

RQ3: ***Is a block-based programming language better suited to help non-programmers define system behavior than a sequential function chart-based programming language?***

Rationale: we want to understand if Blockly, which is specifically targeted to non-programmers, may help reduce errors. We also want to understand if certain errors are more likely to occur in one language than the other. Studying task performance in two languages also helps to differentiate errors due to insufficient language support from errors or problems attributable to the complexity of the environment.

3.2. Experiment procedure

For planning and executing this study, we followed the recommendations by Ko et al. (2015).

Every experiment session (one per participant) followed the same procedure.

(1) Welcome: the participants were informed that the purpose of this experiment was to obtain better insights into the support that end users require when using EPL and Blockly for programming an industrial cell. Afterwards each participant signed a consent for participation form. (5 min)
(2) Demographic details: we obtained background information on programming experience and domain experience. (3 min)
(3) Training Video: each participant was shown a short video introducing Blockly. The video demonstrated how to drag elements, how to navigate the programming pane, and explained the content of each toolbox category. (12 min)
(4) Warmup Task: each participant was asked to complete a warm-up task that required the participant to inspect the base program (see Section 3.3) in order to answer a few yes/no questions which were not further analyzed. (10 min)
(5) Experiment tasks. The experiment tasks consisted of two understandability (R1, R2) and two editing (E1, E2) tasks (see Section 3.4). Each participant carried out one understandability and one editing task in every language. The participant switched only once between the two task blocks in order to avoid confusion: hence either first all tasks with Blockly and then all tasks with EPL or vice versa. We ensured that across all participants every possible task permutation (out of 8 possible permutations - 2 languages $\times$ 2 understandability tasks $\times$ 2 editing tasks) occurred at

least once and at most twice. The program visualization was reset to its initial collapsed state after each task. (maximum 2x 5 min for understandability tasks and 2x 20 min for editing tasks)

(6) After each task, we obtained a complexity score and asked the participant about their perception of the task and the language, and saved their produced solution (4x 2 min).

(7) System usability score (SUS (Brooke, 1996)): at the end each user filled out a form containing the 10 system usability score questions for Blocky and EPL. (2 min)

The experiments were carried out on three days within a 10 day window at the beginning of March 2021, with no more than five sessions per day. As per the above outlined experiment procedure, each session lasted at most for 1 h 20 min.

The experiment environment consisted of a large auditorium room where the participant and the industry co-author could maintain sufficient distance due to Covid'19 safety measures. The participant conducted the tasks on a standard PC where the EPL language environment was smoothly running in a virtual machine, and Blockly running in a browser, the backend hosted by a server on the local network. Interaction with EPL and blockly occurred via a 21″ vertical touch display as available on the actual shop floor. In every session, one of the co-authors assumed the moderator role, while at least one other co-author took written notes. All scientific authors participated remotely via a conference call. During the session, the participant's touch screen was shared permanently and additionally recorded during the task execution.

Note that the participants were not able to execute their adapted program in order to obtain feedback. This was for two reasons: first, we have not completed a translation of Blockly to the underlying real-time robot control language. Hence, for the moment, the Blockly programs are not executable. Second, execution of such programs in a real environment takes considerable time as outlined in Section 2.2. The focus of completing the tasks was thus to obtain as complete and as correct programs as possible on the first "attempt" as this would ultimately result in a reduced amount of time-consuming iterations in the actual production cell in a real-world setting.

The experiment procedure was tested with two engineers at ENGEL that did not participate in the final study. The two test runs allowed us to fine-tune the expected task duration, correct misunderstandings in the task descriptions, and obtain feedback that the tasks are realistic.

3.3. Study input: Robot program

Programming tasks in an industrial environment mostly consist of understanding and adapting existing programs rather than having to build them from scratch.

We, therefore, selected a real (thus large and complex) EPL robot program that comes pre-shipped with ENGEL machines and thus is frequently used as is, or adapted by engineers at the customer's production sites and by ENGEL engineers. The program controls how a robot collaborates with an injection molding machine, specifically for inserting components into the mold – e.g., metal reinforcements into plastic components – before picking and placing solidified plastic parts.

The functionality and overall structure of a program in an IMM production cell can be mapped to the following main steps that together make up one production cycle:

(1) *Take insertion components.* Depending on the number of parts produced during a single run and the position of the insertion components, the robot has to pick up each component separately from a location outside the machine but within the production cell. An insertion component is a supplied part, e.g., the metal part of a screwdriver, that is inserted into the mold for adding plastic material, e.g., the screwdriver's grip. This step assumes that a robot has multiple grippers for simultaneously holding the insertion components and also being able to pick the parts from the previous run (see next step).

(2) *Pick solidified part from previous run.* This step includes moving safely towards (fragment shown in Fig. 3) and into the machinery area, avoiding collisions with obstacles like tie-bars, and synchronizing with the opening of the mold.

(3) *Insert components into mold.* If it is impossible to insert components while already holding parts, then this step may also be placed at the end of a cycle.

(4) *Detach and dispose sprue* after moving out of the machinery.

(5) *Place solidified parts.* This step requires safe movement of the object to the placement site. Parts are placed either

- at the indicated placement area (e.g. conveyor belt).
- at the quality inspection site, if an inspection of the part was requested.
- in garbage, if part was detected as faulty, otherwise.

Fig. 3 shows a small fragment of this robot program in the EPL notation. The entire program contains 170 EPL items in total and has a cyclomatic complexity[3] of 50 (32 conditional branches and 18 parallel branches).

In previous papers, we discussed the mapping of this program to Blockly (Winterer et al., 2020) and the main differences in how information is presented to the user. The two program versions (in Blockly and EPL) and the accompanying respective toolboxes are syntactically equivalent. This was achieved by first taking every EPL language element and identifying an equivalent, existing Blockly block, or creating one where no suitable one existed yet. Translation of the individual programs was done manually, but could have been automated with a custom transpilation tool given the abstract syntax tree remains the same. We then rewrote the programs to use Blockly concepts not available in EPL such as subroutines and arguments and conducted further refactoring to reduce code duplication. We additionally had our industry partner inspect the generated blockly programs to ensure they cover all relevant aspects. Fig. 4 depicts an excerpt of the program in Blockly. EPL might appear to be more abstract and to hide more details compared to Blockly, due to how an operation (e.g., 'move to robot to position') and its parameter(s) (i.e., a position) are visualized. In Blockly both are distinct textual blocks, while in EPL, the former is represented by an icon while the latter is provided by text to the right of the icon. While EPL does hide some information (and makes it available in a dialog upon selecting the icon), this information is not relevant for defining the logical behavior of the robot (but rather determines, e.g., whether certain parameters can be separately configured upon deployment, or whether certain information should show u p on some separate user interface page).

3.4. Understandability and editing tasks

The overall set of tasks consisted of one warmup task in Blockly, two understandability tasks of equal complexity (one to be executed with EPL and one with Blockly), and similarly two editing tasks of similar complexity (again one for EPL and one for Blockly).

Warmup Task

[3] As outlined in Winterer et al. (2020), we first translated the EPL program into an abstract syntax tree equivalent javascript version and then determined the cyclomatic complexity using jshint https://jshint.com/.

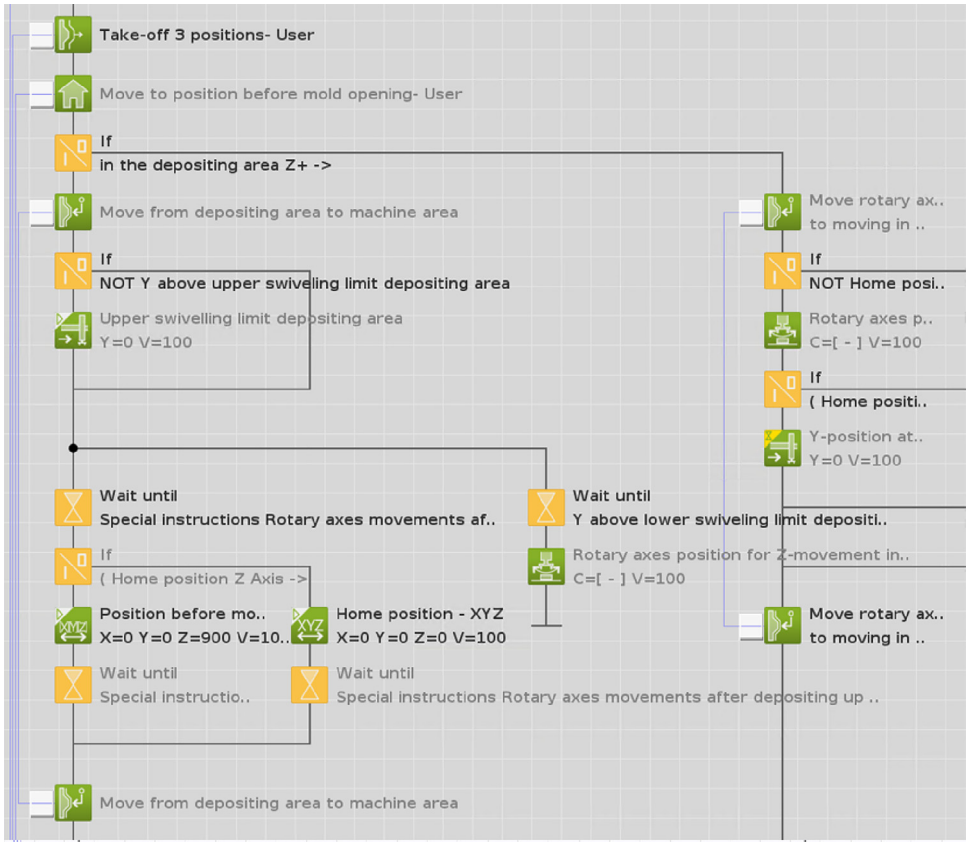

Fig. 3. ENGEL robot program.

Participants were tasked to answer four basic questions on the behavior of the program such as: "The robot places a part on an interim location and then retrieves another part from the machine. Is this true or false or unclear?"

Understandability Tasks

Participants were tasked to locate two operations in the program: locate the operation, where the robot places a part on the conveyor belt, respectively at the quality inspection site, and located the specific operation, where the movement to that position is stated.

Editing Tasks

The editing tasks consisted of integrating external machines into the default robot program. Both tasks included the following behavior aspects that are essential for integration: moving the robot, gripping and placing of parts, waiting for events, and triggering events.

The first editing task describes the integration of an engraving station before placing the freshly molded part on the conveyor belt. In our experiment, the purpose of the engraving station is to apply a unique identifier on a molded part with a laser. The new control logic to be added by the participants consists of the following coarse-grained requirements:

(1) The robot must check whether engraving is enabled or should be skipped
(2) The robot requires the ready signal from the engraving station before it may place a part.
(3) The robot must signal the engraving station when it has placed a part
(4) The robot must wait away from the engraving station during the engraving process for the engraving completed signal.
(5) The robot must signal the engraving station when it has picked the part.
(6) The robot must count the number of engraved parts.

The second editing task describes the integration of an oven that heats up raw parts before they are inserted into the mold for add-on molding. To this end, the participants were asked to implement the following coarse-grained requirements:

(1) The robot must check if the molding form is open
(2) The robot must check whether the oven is ready
(3) The robot must wait away from the oven for the heating completed signal.
(4) The robot must pick the raw part from the oven.
(5) The robot must signal the oven when it has picked the raw part.
(6) The robot must place the heated raw part in the mold.

Note that requirements for these two tasks come at similar but not identical levels of granularity to avoid as much as possible learning effects across the tasks. We collected the participant's perceived Single Ease Question (SEQ) (Sauro and Dumas, 2009) score after each task to control that both understandability tasks and both editing tasks were equally complex. SEQ is measured in the interval from 1 (very difficult) to 7 (very easy). The boxplot for each task is depicted in Fig. 5. As the SEQ results indicate, the editing tasks given during the study were more complex than the participants' regular programming/configuration activities. The provided tasks, however, represent relevant (future) scenarios that need to be supported and should be able to be carried out by this kind of experts.

3.5. Study participants

Our industry partner identified a set of 20 suitable participants that have experience in the injection molding domain and are

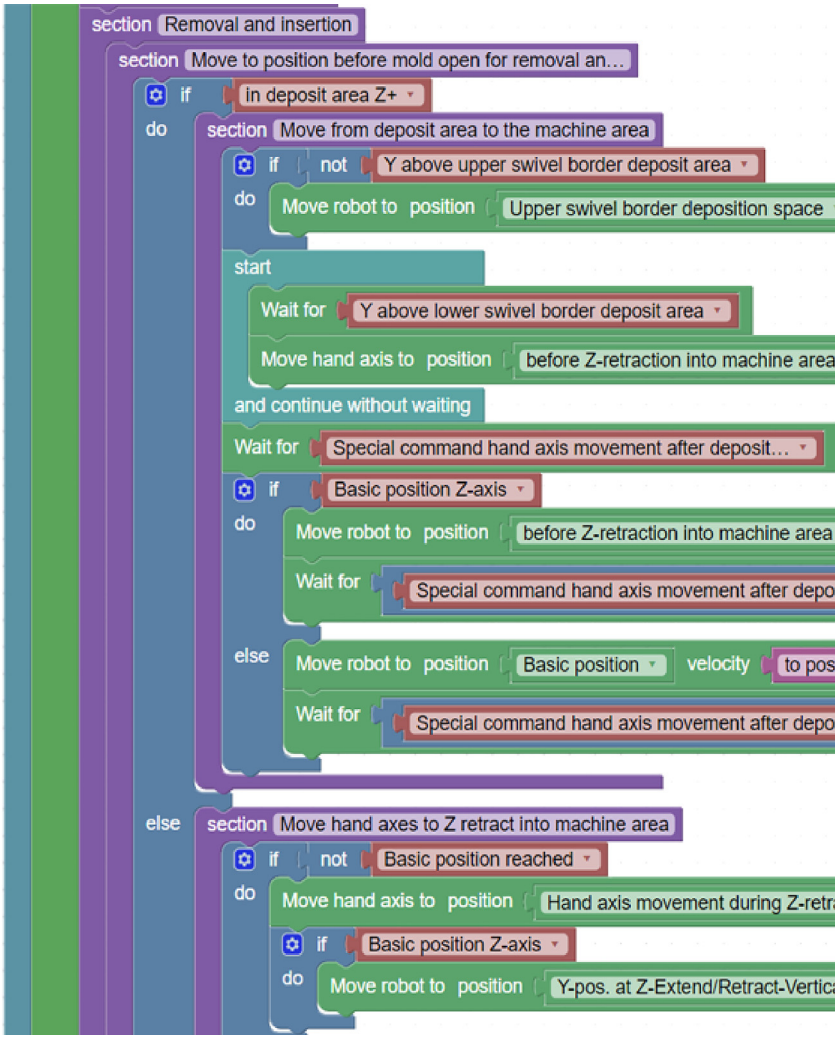

Fig. 4. ENGEL robot program translated to Blockly (detail, clipped).

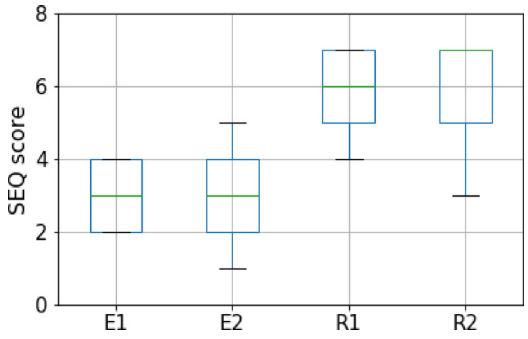

Fig. 5. Single easy question scores for understandability (R1, R2) and editing tasks (E1, E2).

familiar with the EPL editor but are not considered programmers. Out of these 20 candidates, 11 volunteered to participate in the study. On a scale of 1 (high) to 5 (no) experience with programming, 10 judged themselves at 4 or 5. The participants have between 6 and 35 years of experience (with a median of 20 years). On a scale 1 (daily), 2 (weekly), 3 (monthly), 4 (half-yearly), and 5 (less often) of using the EPL editor, all participants reported within the interval 1 to 3 (with a median of 2). Regarding the experience with a standard ENGEL 6-axis linear robot, and a scale of 1 (high) to 5 (no) experience, participants reported in the interval of 1 to 4 with a median of 3. None of the participants had ever seen Blockly before.

3.6. Data processing

We followed a data processing approach informed by thematic analysis. In an open coding phase, we collected notes from all co-authors that participated in the experiment sessions and identified the first set of emerging challenges such as usability problems, mistakes, and degrees of assistance from these notes which also covered explicit participant feedback.

With this initial set of codes, we conducted another round of coding refinement by revisiting the notes and also by inspecting the videos, now also explicitly determining how often the various codes emerged for which participant. During video inspection, we also focused on the level of assistance given, in which sequence participants executed the various fine granular steps of the editing tasks and the correctness of the task result. Towards determining the assistance needed, we defined the following six levels:

(1) No assistance required
(2) Assistance on how to use a language element
(3) Assistance where an element is found in the respective toolbox

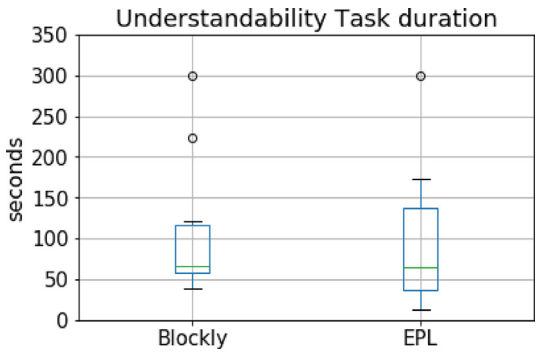

Fig. 6. Understandability task duration for n = 11 participants.

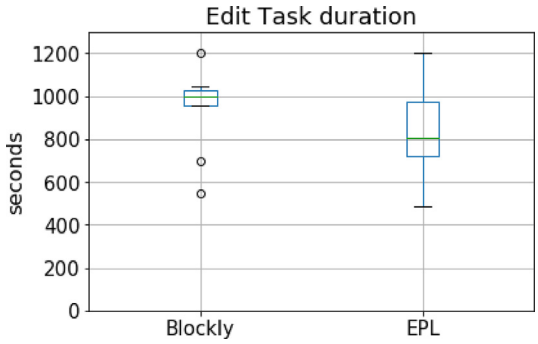

Fig. 7. Edit task duration for n = 9 participants.

(4) Assistance by pointing out that an element is missing (but not directly identifying which one).
(5) Assistance what element might be suitable to include next
(6) Assistance by explaining what element to select and where to insert step by step

We report the qualitative insights in the next sections.

4. Results

4.1. Results - Understandability tasks

Out of the 11 participants, 10 managed to correctly complete the understandability task within the allocated five minutes. The completion duration for Blockly and EPL are similar, with the average completion time for Blockly being 1:27 min (std of 0:52 min), and 1:14 min (std of 0:52 min) for EPL, respectively. The minimum achieved duration for Blockly of 38 s and 13 s for EPL seems to show that one influencing factor is the simpler mechanism for expanding of program sections in EPL by simply tapping on the "+" icons in contrast to Blockly's access via the context menu. However, the average time and standard deviation indicate that this aspect affects only the very quickest participants. Fig. 6 provides a box plot of the understandability task durations for the 11 participants. The respective aborted EPL and Blockly task instances for one participant are visible as outliers at the 5 min (i.e., 300 s) mark.

Key observation: participants were able to complete the understandability task in similar time, even though they have worked with Blockly for the first time.

4.2. Results - Editing tasks

The results in this section stem from the nine participants (n = 9) that attempted the editing tasks E1 and E2. Two candidates considered these tasks too complicated and chose not to do them.

Table 1
Assistance vs. Correctness for each participant (n = 9) that attempted the editing tasks, separated by language.

Blockly		EPL	
Assistance	Correctness	Assistance	Correctness
3	83%	1	100%
4	58%	3	71%
2	64%	3	46%
6	21%	5	92%
5	54%	3	71%
5	57%	5	42%
3	92%	3	86%
3	71%	5	100%
5	100%	5	57%

All participants required guidance during the editing tasks to some degree in both languages: three times the maximum type of assistance was equally high for EPL and Blockly. Twice a participant required more guidance in EPL, and four times the participant required more guidance in Blockly. Table 1 provides for each participant their provided maximum assistance level for Blockly and EPL to continue with their tasks and what percentage of correctness they ultimately achieved. Fig. 8 visualizes these data points as a scatter plot. The scatter plot also shows that the assistance level does not correlate with correctness. This is not surprising as assistance was applied to allow the participants to continue with their task, but not to solve or correct it for them. As the assistance levels outlined in the previous section represent a Likert scale and not an ordinal scale, providing an average over the maximum assistance level would be unsound.

Based on the prior expertise of participants with EPL and no expertise with Blockly, we attribute some of the assistance needed to the novelty of the Blockly programming environment and language. As quite significant assistance was often needed, we suspect that the complexity of the overall editing tasks is a major contributing factor. Assistance for what element to best use next (i.e., level 5) was required four times for EPL and three times for Blockly (with a single level 6 assistance necessary for Blockly).

A slight difference is also apparent when studying the task duration. Fig. 7 provides a box plot of the edit task durations for nine participants. We note that participants required more time for the task with Blockly (avg 15:37 min, std 3:05 min) compared to EPL (avg 14:06 min, std 3:13 min).

Key observation: participants are able to execute the editing tasks in Blockly with comparable levels of assistance but taking longer compared to EPL. Overall, participants needed considerable assistance when integrating coordination logic with robot behavior logic.

Note especially that the following observations are made independent of the programming language (i.e., these errors and difficulties occurred with Blockly and EPL). We discuss Blockly and EPL specific aspects later in the scope of potential for improvement in the next section.

We categorized what are common mistakes from analyzing the participants' solutions in terms of correct sending of signals, correct waiting for external or internal signals, moving the robot in the cell, as well as placing or picking an item. The overarching error type here was participants forgetting to include these program elements, but not so much an error by using them at an incorrect place in the sequence. The lack of the latter type of errors might be due to a combination of clear sequential behavior for some logic (e.g., waiting for a signal and only then moving) with freedom of sequence due to implicit parallelism for other parts of the logic (e.g., moving away and sending a signal can

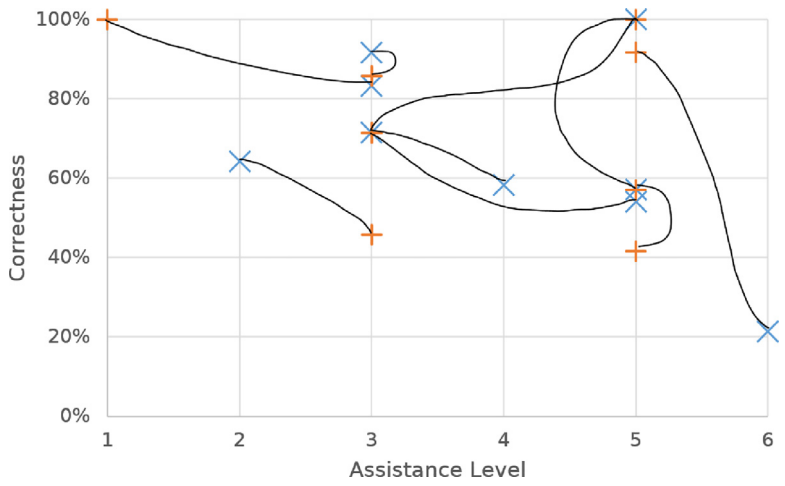

Fig. 8. Assistance vs. Correctness scatter plot of EPL (orange '+') and Blockly (blue 'x') tasks with data points from the same participant (n = 9) identified via connecting lines. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
Absolute and average number of mistakes per type in the respective language out of possible incorrectly sending, waiting, moving, or picking/placing.

	Send (17x)	Wait (13x)	Move (21x)	Drop/pick (13x)
EPL	5.0	2.0	6.0	4.0
	29%	15%	29%	31%
Blockly	4.0	2.5	5.0	4.0
	24%	19%	24%	31%

be typically placed in any order as the moving command just starts a movement with immediate execution of the subsequent command such as signal sending while the movement to the desired position is continued asynchronously.)

We analyzed the participants' solutions and detected that coincidentally elements for sending, waiting, moving, and picking/placing were all equally often forgotten. Concretely, all these mistakes were made at least once by 4 out of the 9 participants that attempted the editing task. Some participants made no mistakes, others made all these mistakes (either in Blockly and/or in EPL).

To obtain more insights, we determined how often a particular program element was forgotten (i.e., not just at least once per program, but across all occurrences in the program). When comparing these mistakes per programming language (see Table 2) we noticed no discernible difference.

Key observation: participants make the same type of mistakes to a similar degree independent of language.

The observations reported above relied primarily on the final outcome of the editing task. Additional observations are available in the form of hand-taken notes and videos that give insights into the challenges that the participants faced during the tasks. We observed multiple participants making the following errors (correcting some of them until the end of their task):

(1) Participants found it unclear which element to use: whether a signal, a marker, or a configuration flag to use in a condition.
(2) Participants found the difference of how to include an input signal compared to an output signal confusing. The former can only be used as part of a "wait-condition" or an "if-condition" while the later is provided as a dedicated block/element available for direct inclusion (in Blockly and EPL).
(3) Participants occasionally utilized an "if-condition" element for the purpose of the "waiting" element (which also is defined as a condition) and vice versa.
(4) Participants applied a loop element instead of a wait (expecting some sort of loop until behavior).
(5) Some participants confused input signals and output signals.
(6) Some participants utilized a waiting element without selecting a signal.

A further challenge consisted in determining whether a particular programming element such as a position, sending/receiving signals, or operation, was available as a pre-defined, domain-specific element in the toolbox, or needed to be custom-defined (e.g., activating a digital output to signal the oven instead of having a dedicated "oven ready" command).

Finally, we observed the approach participants followed for solving the editing tasks in terms of the order in which the various elements were included in the program. While the number of participants was too small to extract fine granular approaches, we, nevertheless, observed three coarse-grained strategies: (1) working through the desired logic step by step as provided in the task descriptions, (2) starting with and focusing primarily on the movements and pick and place commands, and (3) starting with and focusing primarily on the sending and receiving of signals. There was no clear preferred approach, but we noted that more than half of the participants only inserted the pick/place commands at the very end of the task duration (or not at all).

Key observation: participants tend to confuse elements that require the specification of conditions.

5. Usability aspects

5.1. Quantitative measurements

The collected, aggregated SUS scores (Brooke, 1996) from all participants (n = 11) are displayed in Fig. 9. SUS scores range from 0 (extremely difficult to use) to 100 (extremely easy to use). The primary goal of obtaining SUS scores was to ensure the applied languages are usable enough so as not to negatively impact the result by hindering the user in completing their tasks. The mean SUS score for EPL is 55.6 (std 15.7) and 51.6 for Blockly (std 13.1), respectively. Bangor et al. (2009) determined that SUS results highly correlate with adjective ratings on a Likert scale of

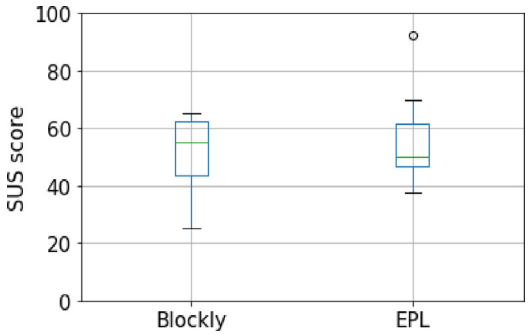

Fig. 9. System usability scores for EPL and Blockly (n = 11).

size seven from "worst imaginable" to "best imaginable". In their analysis, the fourth Likert scale entry (here "Ok") correlates with a SUS scale of 50.9. Hence, both EPL and Blockly can be considered equally usable from this perspective, however, with definite room for improvement.

5.2. Qualitative participant feedback and observations

Participants commented on and observers independently noticed the following phenomena that can be traced back to Blockly not being optimized for touch displays. (Note that to the best of our knowledge no study has investigated the impact of touch versus mouse usage of Blockly, see also the state-of-the-art discussion in Section 7).

(1) participants found the unfold/collapse functionality very tedious to use in Blockly as it requires accessing the elements context menu (which in turn requires a long press to activate).
(2) participants occasionally deactivated instead of expanded elements (as the deactivation command is placed beneath the expand command in the context menu).
(3) participants occasionally highlighted the text of an element when trying to select it for movement when they touched the element too long without dragging it.

With respect to EPL, the majority of participants had problems using the condition editor (see Fig. 2) which introduces a separate set of concepts to learn as the main EPL sequence editor focuses on control flow and not expressions. The main problems were accidentally removing part or all of the condition definition, not finding the right elements to use as part of a condition, or not knowing how to correctly structure the condition. Several participants explicitly noted during the experiments feedback phase, that specifying conditions is much easier in Blockly compared to EPL, where such expressions are part of the language.

These usability shortcomings primarily influence how long users need for a task, but do not affect the task's correctness or influence the required assistance.

6. Discussion and implications

Based on our experiment results, we come to the following conclusions for answering the three research questions.

With respect to **RQ1**: *What are the errors non-programmers make when programming interactions among machines and robots?* we note that non-programmers make the four most often occurring mistakes (i.e., moving, picking/placing, sending events, and waiting for events) similarly often and that there is no difference between the two languages.

Executing the programs in a simulation environment or the real shop floor definitely would help to detect these errors, however, such simulations that integrate machines from different vendors and their integration on the shop floor are still not widely found, or, with respect to in-situ testing, take a considerable time for each testing iteration.

With respect to the often (temporarily) forgotten place or pick command, we hypothesize that this may be due to a mental model that considers placing or picking as part of the movement as picking or placing is the purpose of the movement. This can be compared to the mental model of oneself picking a glass of water which is typically not explicitly perceived as a decomposition into a human arm movement part and a glass gripping part. One implication thereof could be the provisioning of an integrated move and pick/place command as part of the programming language elements.

With respect to **RQ2**: *What level of assistance do non-programmers need?* we conclude that the complexity of simultaneously integrating machine/robot interaction at the physical level (movements, picking/placing) and software level (signals, events) requires that non-programmers would benefit strongly from additional assistant mechanisms aside from merely optimizing the language elements.

Specifically, our observations and explicit participant feedback highlighted the need to support the following aspects:

(1) Selecting/simplifying the choice of programming elements as domain experts spent significant time finding elements. Possible support may consist in filtering out or, inversely, highlighting certain elements based on the programming context.
(2) Supporting the engineer in more easily distinguishing between temporal/asynchronous control flow (i.e., waiting for the occurrence of an event) and conditional control flow (i.e., if/then/else and loops constructs). In Blockly, an alternative visualization for parallel behavior such as in Ritschel et al. (2020) would be possible.
(3) Providing in addition to low-level, fine granular commands also higher level, customizable commands. For example, as already suggested above, providing an integrated move and pick/place command, or an "wait at position until event" command. While more investigations involving domain experts and analyzing existing programs would identify a suitable set of additional elements, the question remains on how to make non-programmers aware of these elements and how/when to use them.
(4) Integrating machine self-descriptions as dynamic programming elements. It is infeasible to a-priori include custom programming elements for every machine that may possibly be integrated into a production cell. Yet participants commented on their desire to have elements for interacting with the external machine (here oven and engraving station) via domain-specific elements rather than having to use low-level signals and having to define meaningful positions themselves. One possible approach is to have the language environment make use of machine description to produce respective programming elements dynamically upon "discovering" a new machine on the shop floor. Even more powerful are mechanisms that then suggest or check the order in which machine operations and events are applied in a program. In an industry context, one could apply OPC UA programs to this end. Put simply, OPC UA programs describe in a standardized way the signals/messages/methods and their valid usage context to interact with a machine and hence would be suitable to dynamically and automatically derive programming language elements and further support the interaction with a previously unknown machine.

With respect to **RQ3**: *Is a block-based programming language better suited to help non-programmers define system behavior than a sequential function chart-based programming language?* we find no strong evidence that Blockly is a better choice overall to enable non-programmers to express programs in industrial robotic production cells.

With a large number of elements in the toolbox and multiple levels of abstraction (i.e., domain-specific vs. generic elements), the advantage of visual languages to enable recognition over recall diminishes as the user increasingly has to remember what is possible (i.e., which elements are available at which abstraction level). In line with our answer to RQ2, reducing the number of elements, filtering/highlighting of elements, or recommendation of elements (Rahman et al., 2016) might be potential approaches to reduce the cognitive load of the user, respectively, reduce the time to find the right element.

One interesting future research direction that takes a rather different approach is the comparison to defining robot behavior with languages such as PDDL (the Planning Domain Description Language) (McDermott et al., 1998) that focuses on predicates and actions (with their pre/post conditions and effects) for merely defining what is possible in the production cell, and then let a planner determine a suitable (robot movement, picking/placing, signaling, and waiting) plan.

With these findings and the current focus on non-programmers, one might raise the hypothesis that regular programmers make the same mistakes or would have different problems. We have only anecdotal evidence that motivates further research along these lines: when testing the experiment tasks also with Computer Science Bachelor students and work colleagues, we noticed that they made fewer mistakes than the participants. One possible reason for this performance difference could be the algorithmic thinking that comes with a computer science education and the practice of abstracting from and copying of existing code pieces quite common in software engineering. On the other hand, a lack of domain experience could have required the testers to follow the task descriptions more carefully and thus reduce the chance of missing some operations. These initial observations, however, are too few in number, and the tests where not done in a rigorous enough manner to draw robust conclusions but point to interesting follow-up work.

In summary, we conclude from the somewhat unexpected finding of observing no discernible different in performance between EPL and Blockly, that for more complicated (realistic) tasks, the chosen language itself (while still important) does not seem to make a sufficiently large impact. Rather, supporting the ability for non-programmers to design and adapt complicated scenarios will need assistance mechanisms which can guide the domain experts along the lines outlined in the discussion of RQ2 above.

6.1. Threats to validity

Due to the low number of available participants, we chose to have every participant work with both languages rather than splitting them into two groups. This may lead to learning effects where executing one task affects the time and/or correctness of the other task. In order to avoid such learning effects, we carefully designed the tasks to require similar programming elements but semantically different descriptions. In addition, we randomized the order and combination of tasks and language to minimize the impact of any remaining learning effect.

Due to the nature of a controlled experiment, especially the goal to evaluate realistic tasks, we had to ground these in a particular domain, here injection molding. Hence we are careful to generalize our findings as some of the encountered difficulties might not occur in other domains, or inversely, additional difficulties emerge. For example, the actual gripping of a part is not a major challenge in this domain as the parts are always located at very precise, fixed positions. However, the necessity to define the interaction between machines and controlling the robot physical behavior is a common aspect in most automated shop floors.

While Blockly supports easy transformation into programs, such a translation becomes a matter of safety as there needs to be considerable effort ensuring that the produced code is safe to control the robot while the human is testing it on an actual machine. Such effort would also require significant costs on the side of our industry partner, not just of writing the translation. Consequently, the inability to execute the program under development, hence the lack of a rapid write-test loop, requires to look at programming errors differently and hence cannot be compared with traditional programming where rapid, frequent execution will reveal errors early. As long as programs need to be slowly tried and tested on actual hardware due to the lack of an integrated simulation of the production cell, reducing the number of initial errors and getting the program as correct as possible on the first try is key to achieving short reconfiguration and adaptation times. Hence, we highlight that the findings in this paper on the comparison and usage of block-based and sequence chart-based programming cannot be reliably transferred to traditional programming environments.

Another limitation is the restriction to two languages. Testing a comprehensive set of languages in an industry setting indeed would be required to obtain more robust results, but is difficult for several reasons: (1) the problem of finding sufficient candidates to avoid learning bias as we cannot repeat the same tasks in too many languages by the same participant, (2) the ability of our industry partner to provide so many candidates in the first place, (3) the effort of building comparable versions of EPL in those various languages. Yet as we outlined in the experiment design section, the majority of block-based languages work and behave very similar.

7. Related work

Block-based programming has gained increasing popularity over the recent years due to the emergence of new programming systems such as Scratch (Maloney et al., 2010) and Blockly (Fraser, 2015). They are often used for educational purposes (Trower and Gray, 2015; Topalli and Cagiltay, 2018; Maloney et al., 2008), e.g., when teaching programming to children. Despite their success in education, which includes the teaching of programming robots like Nao (Sutherland and MacDonald, 2018) and Sphero (Corral et al., 2019), these languages are barely found in the context of industrial projects. Sometimes, in particular due to the often colorful look-and-feel of the programming environments and the provided examples, block-based programming is perceived more as edutainment for children rather than a serious approach to writing programs. Study results, however, show that block-based programming enables creating non-trivial programs even for inexperienced users (Bau et al., 2017). Consequently, block-based programming has also been suggested for applications that require end-user programming (e.g., Weintrop et al., 2017). To this end, several studies have been conducted that indicate the usefulness of block-based programming languages also for industrial robots (Coronado et al., 2019; Blume, 2013; Weintrop et al., 2018; Ritschel et al., 2020).

Several approaches address the challenge of supporting the user in understanding what the robot will do.

Multiple papers focus on integrating the visualization of feedback on force and torque vectors (Mateo et al., 2014) or movements positions (Shepherd et al., 2019; Gadre et al., 2019) with virtual reality overlay of the production cell. These approaches, however, focus on the robot behavior only but do not address the

integration (and thus interaction) of the robot with other shop floor participants which are not visualized or controlled.

Another frequently investigated end-user focused approach is robot programming by demonstration (Gadre et al., 2019; Ajaykumar and Huang, 2020; Alexandrova et al., 2015; Kumaar and Tsb, 2011). Here, the user shows the robot what to do by moving the robot arm to desired positions by hand and recording these positions. Alternative approaches program the robot by placing physical markers in the robot's environment which the robot then visually detects and interprets at runtime (Sefidgar et al., 2017). Similarly, there are languages that focus on generalization and transferring to other environments of movements and placing actions (Paxton et al., 2018).

The primary downside of such approaches is that the programming elements are limited to robot movements and physical actions. Waiting for signals, sending of signals, or other control flow logic needs to be provided differently, thus requiring the end-user to switch between different programming modes.

This work differs in multiple ways from previous evaluations of using Blockly for robot programming (e.g., Weintrop et al. (2018), Ritschel et al. (2020)) in that the focus is not purely on the robot's physical behavior (i.e., moving and gripping/releasing) but on its interaction with other machines in a production cell. Additionally, the chosen experiment tasks are taken from a realistic environment and reflect task complexity as it would occur in a real setting.

In our previous work (Mayr-Dorn et al., 2021) we introduced three levels of customizability at which a user would interact with machines in a typical production cell. To this end, we distinguish between Level 1 *Domain-specific program sequences* which consists of programming language elements that describe typically useful commands in an IMM production cell such as the signals from the IMM form or robot positions in and around the IMM. At Level 2 *Production site-specific program sequences*, a programmer customizes a program done at level 1 to match the exact cell configuration, while at Level 3 *Product-specific programs*, the final tuning to the exact product is conducted. Programming tasks and abstractions available in those related robot programming studies, however, can be described to be at Level 0, providing basic elements to move the robot, activate a gripper, etc. without any relation to a domain-specific usage of the robot.

Our study identified a few aspects of using Blockly on a touch display. So far hardly any study has investigated the suitability of using touch displays for Blockly. Their focus was on different aspects such as collaborative (i.e., simultaneous multi-user) programming on a large touchscreen (Selwyn-Smith et al., 2017), enabling visually impaired people to use a screen reader to read Blockly programs (Caraco et al., 2019), and the use of a virtual onscreen keyboard to select and insert Blockly element instead of having to drag and drop elements with a finger (Almusaly et al., 2018). While Scratch has been used as the foundation for an Android-based visual programming environment for programming robots (Mateo et al., 2014), the approach has not been evaluated with real users with respect to usability. The focus of this study was primarily on programming robots for polishing, milling, or grinding tasks and their execution integration via augmented reality.

8. Conclusions

In this paper, we presented the results of a controlled experiment in which we aimed to obtain better insights into the problems non-programmers encounter when editing realistic, non-trivial programs in a production context. Quantitative and qualitative data showed that while Blockly is easy to understand, it does not necessarily result in fewer errors or quicker task completion times. Rather, the complexity of the domain is a significant factor as shown by the amount and distribution of error where we detected no significant difference between Blockly and a sequential function chart-based language.

Our future work is two-fold. First, we plan to investigate in more detail which additional concepts would be useful to include in another iteration of the Blockly-based version (as we cannot extend EPL), e.g., a combined move and grip/drop block, and study how these are accepted and applied by study participants. Second, we will investigate how programming elements can be dynamically derived from run-time discoverable machine data, and where available, assistance mechanisms for recommending and inspecting the code element order.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research reported in this article has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), Austria, the Federal Ministry for Digital and Economic Affairs (BMDW), Austria, and the State of Upper Austria in the frame of SCCH, Austria, a center in the COMET - Competence Centers for Excellent Technologies Programme, Austria managed by Austrian Research Promotion Agency FFG, Austria.

References

Ajaykumar, G., Huang, C.-M., 2020. User needs and design opportunities in end-user robot programming. In: Companion of the 2020 ACM/IEEE International Conference on Human-Robot Interaction, Ser. HRI '20. Association for Computing Machinery, New York, NY, USA, pp. 93–95, [Online]. Available: http://dx.doi.org/10.1145/3371382.3378300.

Alexandrova, S., Tatlock, Z., Cakmak, M., 2015. Roboflow: A flow-based visual programming language for mobile manipulation tasks. In: 2015 IEEE International Conference on Robotics and Automation. ICRA, pp. 5537–5544.

Almusaly, I., Metoyer, R., Jensen, C., 2018. Evaluation of a visual programming keyboard on touchscreen devices. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 57–64.

Ball, T., Chatra, A., de Halleux, P., Hodges, S., Moskal, M., Russell, J., 2019. Microsoft makecode: embedded programming for education, in blocks and typescript. In: Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E. pp. 7–12.

Bangor, A., Kortum, P., Miller, J., 2009. Determining what individual sus scores mean: Adding an adjective rating scale. J. Usability Stud. 4 (3), 114–123.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., Turbak, F., 2017. Learnable programming: blocks and beyond. Commun. ACM 60 (6), 72–80.

Bischoff, R., Kazi, A., Seyfarth, M., 2002. The morpha style guide for icon-based programming. In: Proceedings. 11th IEEE International Workshop on Robot and Human Interactive Communication. IEEE, pp. 482–487.

Blume, J., 2013. Iprogram: intuitive programming of an industrial hri cell. In: 2013 8th ACM/IEEE International Conference on Human-Robot Interaction. HRI, IEEE, pp. 85–86.

Brooke, J., 1996. SUS: a "quick and dirty" usability scale. Usability Eval. Ind. 189.

Caraco, L.B., Deibel, S., Ma, Y., Milne, L.R., 2019. Making the blockly library accessible via touchscreen. In: the 21st International ACM SIGACCESS Conference on Computers and Accessibility, Ser. ASSETS '19. Association for Computing Machinery, New York, NY, USA, pp. 648–650, [Online]. Available: http://dx.doi.org/10.1145/3308561.3354589.

Coronado, E., Mastrogiovanni, F., Venture, G., 2019. Design of a human-centered robot framework for end-user programming and applications. In: ROMANSY 22–Robot Design, Dynamics and Control. Springer, pp. 450–457.

Corral, J.M.R., Ruíz-Rube, I., Balcells, A.C., Mota-Macías, J.M., Morgado-Estévez, A., Dodero, J.M., 2019. A study on the suitability of visual languages for non-expert robot programmers. IEEE Access 7, 17 535–17 550.

Fraser, N., 2015. Ten things we've learned from blockly. In: 2015 IEEE Blocks and beyond Workshop (Blocks and beyond). IEEE, pp. 49–50.

Gadre, S.Y., Rosen, E., Chien, G., Phillips, E., Tellex, S., Konidaris, G., 2019. End-user robot programming using mixed reality. In: 2019 International Conference on Robotics and Automation. ICRA, pp. 2707–2713.

Ko, A.J., LaToza, T.D., Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empir. Softw. Eng. 20 (1), 110–141.

Kumaar, A.A.N., Tsb, S., 2011. Mobile robot programming by demonstration. In: Proceedings of the 2011 Fourth International Conference on Emerging Trends in Engineering & Technology, Ser. ICETET '11. IEEE Computer Society, USA, pp. 206–209, [Online]. Available: http://dx.doi.org/10.1109/ICETET.2011.30.

Lewis, R., Lewis, R.W., 1998. Programming Industrial Control Systems using IEC 1131-3. IET, no. 50.

Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., Rusk, N., 2008. Programming by choice: urban youth learning programming with scratch. In: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education. pp. 367–371.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E., 2010. The scratch programming language and environment. ACM Trans. Comput. Educ. (TOCE) 10 (4), 1–15.

Mateo, C., Brunete, A., Gambao, E., Hernando, M., 2014. Hammer: An android based application for end-user industrial robot programming. In: 2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications. MESA, pp. 1–6.

Mayr-Dorn, Christoph, Winterer, Mario, Salomon, Christian, Hohensinger, Doris, Ramler, Rudolf, 2021. Considerations for using block-based languages for industrial robot programming - a case study. In: 3rd IEEE/ACM International Workshop on Robotics Software Engineering, RoSE@ICSE 2021, Madrid, Spain, June 2, 2021. IEEE, pp. 5–12. http://dx.doi.org/10.1109/RoSE52553.2021.00008, https://doi.org/10.1109/RoSE52553.2021.00008.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D., 1998. PDDL—the Planning Domain Definition Language. Tech. Rep. CVC TR98003/DCS TR1165, Yale Center for Computational Vision and Control, New Haven, CT.

Paxton, C., Jonathan, F., Hundt, A., Mutlu, B., Hager, G.D., 2018. Evaluating methods for end-user creation of robot task plans. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS, pp. 6086–6092.

Rahman, M.M., Roy, C.K., Lo, D., 2016. Rack: Automatic api recommendation using crowdsourced knowledge. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1. IEEE, pp. 349–359.

Ritschel, N., Kovalenko, V., Holmes, R., Garcia, R., Shepherd, D.C., 2020. Comparing block-based programming models for two-armed robots. IEEE Trans. Softw. Eng. 1.

Sauro, J., Dumas, J.S., 2009. Comparison of three one-question, post-task usability questionnaires. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Ser. CHI '09. Association for Computing Machinery, New York, NY, USA, pp. 1599–1608, [Online]. Available: http://dx.doi.org/10.1145/1518701.1518946.

Sefidgar, Y.S., Agarwal, P., Cakmak, M., 2017. Situated tangible robot programming. In: Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction, Ser. HRI '17. Association for Computing Machinery, New York, NY, USA, pp. 473–482, [Online]. Available: http://dx.doi.org/10.1145/2909824.3020240.

Selwyn-Smith, B., Homer, M., Anslow, C., 2017. Towards collaborative block-based programming on digital tabletops. In: 2017 IEEE Blocks and beyond Workshop (B B). pp. 57–60.

Shepherd, D.C., Kraft, N.A., Francis, P., 2019. Visualizing the "hidden" variables in robot programs. In: Proceedings of the 2nd International Workshop on Robotics Software Engineering, RoSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019. IEEE / ACM, pp. 13–16, [Online]. Available: http://dx.doi.org/10.1109/RoSE.2019.00007.

Sutherland, C.J., MacDonald, B.A., 2018. Naoblocks: A case study of developing a children's robot programming environment. In: 2018 15th International Conference on Ubiquitous Robots. UR, IEEE, pp. 431–436.

Topalli, D., Cagiltay, N.E., 2018. Improving programming skills in engineering education through problem-based game projects with scratch. Comput. Educ. 120, 64–74.

Trower, J., Gray, J., 2015. Blockly language creation and applications: Visual programming for media computation and bluetooth robotics control. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education. p. 5.

Weintrop, D., Afzal, A., Salac, J., Francis, P., Li, B., Shepherd, D.C., Franklin, D., 2018. Evaluating coblox: A comparative study of robotics programming environments for adult novices. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–12.

Weintrop, D., Shepherd, D.C., Francis, P., Franklin, D., 2017. Blockly goes to work: Block-based programming for industrial robots. In: 2017 IEEE Blocks and beyond Workshop (B & B). IEEE, pp. 29–36.

Winterer, M., Salomon, C., Köberle, J., Ramler, R., Schittengruber, M., 2020. An expert review on the applicability of Blockly for industrial robot programming. In: 25th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2020, Vienna, Austria, September (2020) 8-11. IEEE, pp. 1231–1234, [Online]. Available: http://dx.doi.org/10.1109/ETFA46521.2020.9212036.