Considerations for using Block-Based Languages for Industrial Robot Programming – a Case Study

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
firstname.lastname@jku.at

Mario Winterer, Christian Salomon, Doris Hohensinger, Rudolf Ramler
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
firstname.lastname@scch.at

***Abstract*—The paradigm shift triggered by Industry 4.0 leads to a fast rising number of industrial machinery and collaborative robots that increases the need for flexible customization of production processes and automation workflows. End-user programming of industrial robots has become an essential capability for all areas in industry. Consequently, different visual programming languages have found their way into the domain of industrial robot programming. In this paper, we investigate the applicability of block-based programming languages for large and complex robot programs in realistic environments. Here, a key aspect of robot programming is not only the interaction with the physical environment, but also the robot's interaction with other shopfloor participants at the software control level. To this end, we analysed the requirements for programming a robot based a real world production cell and implemented the necessary programming constructs using Blockly, an open-source block-based visual language. We assessed the results comparing the implementation of a change in Blockly and the Sequential Function Chart-based language. We find that while Blockly is able to express large and complex real-world robot programs, a major contributing factor is not just the language itself but the presentation of the robot's run-time environment as well as support by the development environment (i.e., editor). Our preliminary user experiment has identified a set of challenges in understanding and changing such programs that we now plan to follow-up with a larger user study.**

***Index Terms*—Robot programming, end-user programming, manufacturing automation, block-based programming languages**

I. Introduction

Flexibility is key in industrial manufacturing systems and production lines in the era of Industry 4.0 [1]. There is a growing trend towards small batch sizes, individualization of products, and shorter life-cycles. The increasing demand for more flexibility is mastered by more capable and versatile hardware, controlled by highly configurable and adaptable software. Thus, flexibility is often directly connected to the ability to widely and easily adapt the software to expected as well as unexpected changes due to new product types or a changeover in the processing and logistic workflows.

Despite constant investments in smart and autonomous robot systems making use of emerging technologies such as artificial intelligence [2], there is still the need to involve humans in the process of adjusting and extending automated workflows. Besides many scenarios in today's manufacturing and production landscape, this need is also embraced in the development of collaborative robots that work alongside humans and support many different tasks by enabling automation. Adapting this automation support to individual needs and tasks is essential for a productive collaboration of humans and robots.

The paradigm shift triggered by Industry 4.0 leads to a fast rising number of machinery and robots and interconnected environments that bring human workers and robots closer together. Thus, more and more human operators are affected and many of them have a highly different technical background and only little or no programming skills. To enable flexible adoption and extension of robot programs even for untrained operators, programming environments and tool support have to be highly intuitive, easy to use, and widely accessible.

Therefore, visual languages for end-users are frequently applied in industrial robot programming. Prominent examples are icon-based flow charts like *MORPHA* [3] or the *ENGEL Programming Language* (*EPL*), which represent commands as colored graphical pictograms connected and aligned according to the program's control flow. Large and complex robot tasks are implemented in these languages. Human operators are expected to be able to understand and change these programs in order to adapt tasks to their needs and environments.

Nowadays, block-based visual programming languages have received much attention. These languages are typically used for educational purposes with the goal of making programming more accessible to a larger audience, especially for novice users and young learners. Despite their success in education, which includes teaching of programming robots like Nao [4] and Sphero [5], these languages are barely found in context of industrial projects. However, several studies have been conducted that indicate the usefulness of block-based programming languages also for industrial robots [6]–[9].

The studies conducted so far usually apply small-scale examples of robot programs suitable for controlled experiments. In our study we focuses on the question whether visual end-user programming using a block-based programming language is applicable for implementing *large and complex real-world robot programs*. A key aspect of the industrial robot environment under investigation in this paper is integration with

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH.

other shopfloor participants. Robots are not merely picking and placing parts but need to interact with other machines: query their state, wait for events, and send out signals in return. In order to obtain insights under realistic conditions, we ported an existing robot program that is actually used in industry to *Blockly* and investigate how an exemplary (but realistic) adaptation in Blockly and in the original EPL version may be implemented. We identify key considerations for supporting flexible programming of industrial robot behavior such as the need to support customization at multiple levels of abstraction or the integration of connected machines unknown to the programmer at the time of customization, both which go well beyond the actual control of the physical robot movements. We identified through preliminary user observations that identifying the relevant programming constructs from the many available elements in a heterogeneous production cell is one key concern.

The remainder of this paper is organized as follows: In section II we describe block-based programming and the industrial context of this work. In Section III we present our research questions and the experiment setup. In Section IV we describe the implementation in Blockly and its toolbox, and we compare the resulting Blockly program and the implemented change task with the EPL counterpart in Section V. In Section VI we conclude the paper and outline avenues for future work.

II. Background

A. Block-Based Programming

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 [10], 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 to be remembered. Additionally, the listing of all block types helps the user to become familiar with language elements and to maintain overview of system components.
- Programming causes high cognitive load, in particular for new users. This is reduced, because block-based programs 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.

Block-based programming has gained increasing popularity over the recent years due to the emergence of new programming systems such as Scratch [11] and Blockly [12]. They are often used for educational purposes [13]–[15], e.g., when teaching programming to children. 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 write programs. However, study results show that block-based programming enables to create non-trivial programs even for inexperienced users [10]. Consequently, block-based programming has also been suggested for applications that require end-user programming (e.g., [16]).

B. Blockly

Blockly is a 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.

Fig. 1. Hello World example in Blockly.

Due to its accessibility, extensibility, and large community support numerous popular block-based programming environments, like MakeCode [17] and Scratch are based on the Blockly library. Fig. 1 shows a simple example program printing ten times “Hello World!” implemented in Blockly. The program contains the following types of elements:

- A subroutine (named “greet the world”) is a callable code part that may include sub-statements and therefore

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

supports *program organization*. In contrast to Scratch, a subroutine in Blockly and MakeCode visually frames its sub-statement

- *Control flow structure* like a conditional branch or loop ("repeat ... times do") can be used to dynamically change the behavior of the program dependent on *data elements*.
- *Instructions* like "set number to 10" or "print 'Hello World' " in Fig. 1 are either value assignments, system calls, or subroutine calls. If an instruction requires input arguments, the according block either supports quick selection via drop-down or provides dents to snap-in proper input blocks, e.g. variable *data elements*. In Blockly, even constant data elements are represented as separate snap-in elements, in MakeCode and Scratch constant values can directly be entered in the dents of the instruction block.

C. 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. 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. 2 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 (e.g. arguments of a subroutine call, edit the condition expression of a conditional statement, etc.). The toolbox offers language elements – including all flow control and machinery instructions – that can be used for programming.

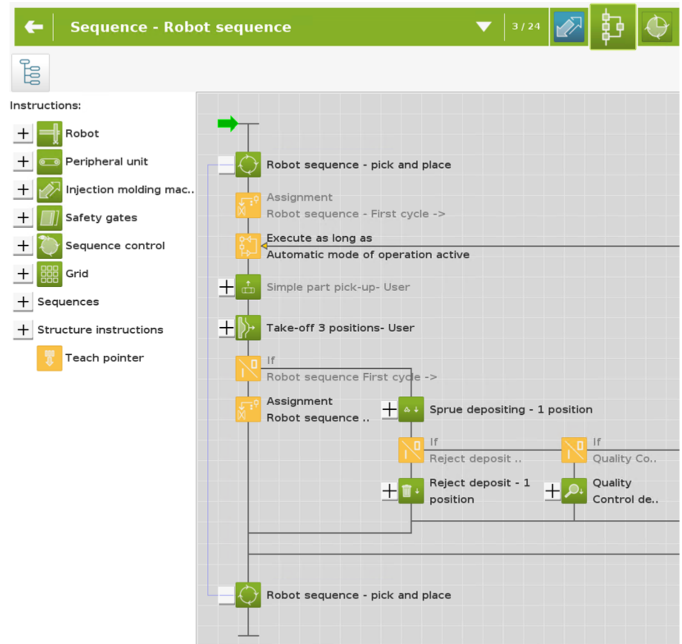

Fig. 2. ENGEL Sequence Editor with a toolbox on the left side (white background) and the EPL program on the right (grey background).

EPL supports a simple way of code-reuse in form of blueprints. These are special blocks that act as named container for a code fragment. Code is grouped together to a block that can be collapsed and expanded in the editor. The blueprint block also appears in the toolbox and can be applied anywhere in the program. In contrast to code reuse via subroutines, the application of blueprint blocks results in code duplication. A copy of the entire code fragment is produced and inserted at the applied location.

D. Levels of Customizability

Since a production cell may be used in many different production scenarios, the tasks of the robots have to be highly customizable. This customization occurs at multiple levels that reflect the types of user configuring, respectively, programming the robot. We distinguish between three levels:
Level 1 - Domain-specific program sequences: these consist of typical tasks a robot carries out inside an injection moulding cell, e.g., moving into the IMM via three predefined waypoints. These sequences make use of domain specific concepts, including, for example, the expected signals from and to the IMM, symbolic positions for picking and placing parts, or movement ranges. Sequences at this level are provided by the machine manufacturer and allow a quick setup of default robot programs, respectively, serve as a useful basis for adaptation.
Level 2 - Production site-specific program sequences: these consist of tasks a robot carries out in a specific production cell customized to the concrete context of the robot. Aspects customized at this level typically include whether and where there is a conveyor belt, whether there is a quality control station, or any other machine the robot needs to interact with.

At this level, only rough sequences are physically executed by the robot.

Level 3 - Product-specific programs: a this level, the production-specific programs are fine-tuned to the concrete product. As a change of the mould implies the production of a different product, typically the precise positions for picking a part, placing a part, and the range in which the robot gripper may rotate a part need to be adjusted as well. Changes at this level also may include using different attachments on the robot arm to actually grip the moulded parts. For example, instead of one large piece pickable with two vacuum suction cups, now 4 smaller pieces are produced that require mounting and controlling two additional vacuum suction cups. This rarely comes with a change of the robot's control flow but is absolutely crucial before the robot can become truly operational. In practice, adaptation at this level occurs via setting a subset of variables such as coordinates of symbolic locations, speed, thresholds, and other control flags that are explicitly declared as configuration parameters in the robot program without providing write access to the programs's detailed behavior.

In this paper, we focus on the customization need at the level of production site-specific programs.

To support use cases at the level of production site-specific programming, the programming language and environment, thus need to provide following abilities:

- access to domain-specific elements and sequences: the user is primarily a domain expert, thus expects certain predefined elements (positions, movements, signals, etc) without having to specify/create these themselves.
- to adjust these domain concepts, respectively extend and combine them.
- to interact with arbitrary, unforeseen machinery.
- to combine domain-specific elements (see item 1) and low-level language elements (see item 2) seamlessly.
- work on multiple levels of restricting customizability: e.g., ability to expose/highlight dynamically created configuration properties to allow program adjustments without accidentally changing the robot's behavior in an unpredictable manner.

III. Study Design

The *objective* of this study is to investigate the applicability of block-based programming languages such as Blockly for programming industrial robots. Our focus on real-world programs from industry highlighted several requirements and constraints that led to the following research questions. These research questions have been derived from the requirements and constraints for robot programming identified in the industrial context (see II-C).

RQ-1) Implementation: Can complex real-world robot programs be expressed in Blockly and its toolbox? In order to answer this research question we implement a representative, large and complex robot program in Blockly. We analyze if and to what extent all aspects of the semantics currently provided by an implementation in EPL can be expressed accordingly and to what extend EPL's toolbox can be replicated in Blockly's toolbox.

RQ-2) Readability and Understandability: How does Blockly affect readability and understandability of robot programs? In the context of software, readability is generally defined as a human judgment of how easy it is to understand the code of a program [18]. Readability is a prerequisite for understanding the code. The relationship between readability and understandability is similar to the relation between the syntax and semantic of a program; readability is affected by syntactic aspects while understandability is linked to semantic aspects [19].

RQ-3) Changeability: How does Blockly and EPL support customization and adaptation of robot programs? The changeability of a software system is the ease with which it can be modified to match changes in the requirements or the environment. Changeability includes the non-functional requirements for adaptability, flexibility, modifiability and robustness [20]. Various measures (e.g., modularization, introducing adjustable parameters, or design patterns) can be applied to reduce the impact of changes and the corresponding effort. Many of these measures are reflected in structural properties of a software system [21].

A. Experiments

In order to answer the research questions, we performed an experimental study in which we implemented a representative real-world robot program and corresponding toolbox in Blockly. This program is a re-implementation of a large and complex instance of a robot program that is actually used by ENGEL and many of its customers. The generated toolbox contains a comparable amount of domain specific programming elements. Aside from comparing the program in EPL and Blockly, we compare the steps taken when implementing a realistic change.

B. Study Object: Robot Program

For our experiments we selected a large and complex EPL robot program that is frequently used by ENGEL and at customer production sites to control robots that collaborate with injection molding machines. The program is applied when a robot has to insert 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.
2) *Pick solidified part from previous run.* This step includes moving safely towards (fragment shown in Fig. 3) and

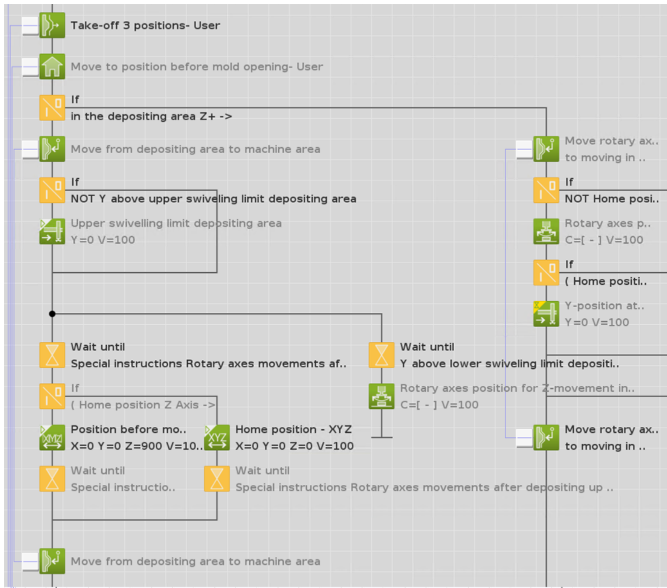

Fig. 3. ENGEL Robot Program.

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 of 50 (32 conditional branches and 18 parallel branches). The very high complexity of the program results from four main reasons. First, the program needs to distinguish between a first run where no parts are ready to be picked and placed but only inserting parts needs to be done (hence, skipping certain position and rotations and waiting signals). Second, the program needs to check for and react to failures in picking and releasing parts. Third, the robot might be restarted in an unknown state from a previous, aborted program run and potentially still have parts on its grippers. Forth, the program comes with multiple configuration and extension points to provide the flexibility to adapt to the wide range of predefined insert-pick-and-place scenarios without having to build these scenarios from scratch.

IV. Experiments and Results

Re-implementing the selected robot program in Blockly included two preparatory steps: Porting the EPL language structures to Blockly and porting the EPL toolbox to Blockly. We then implemented the program in Blockly and subsequently implemented the same change in EPL and in Blockly. Note that in previous work [22] we briefly reported on the effect of using Blockly on cyclomatic complexity and the use of subroutines for structuring robot programs. That previous work described in more detail the aspect of porting EPL to Blockly which we describe to some extent in the following section for sake of completeness. The focus in this paper is on the aspect of integrating the program logic within a complex production cell of multiple participants (i.e., focus on the toolbox) as well as the customizability of the program.

A. Porting to Blockly

In this first step, the EPL source program was directly mapped to Blockly concepts, i.e. all commands and control structures were rewritten in Blockly syntax without any further modification or optimization to obtain two semantically equivalent languages that can use the same mapping to the underlying execution language Teachtalk. Additional custom blocks were defined as Blockly does not support all language concepts of the EPL language out of the box, i.e., *Parallel branch*, a *Section* element (to better structure code blocks), and *Section exit* (to skip ahead). However, necessary language extensions can be easily introduced due to the framework character of Blockly.

The largest part of custom blocks is related to the *domain specific commands* that are provided by EPL. These blocks include sending and waiting for signals, starting and waiting for timers, specific robot commands (e.g., move robot to a certain position, rotate robot's wrist axes, suspend robot movement on a signal, test if robot is at a certain position), special commands for peripheral devices (e.g., activate/deactivate suction gripper, check state of suction gripper).

A very obvious difference between EPL and Blockly is how statement related information is presented to the user. Blockly relies mainly on textual information and a puzzle-like presentation of input arguments, whereas EPL combines pictograms with complementary text. EPL has a large set of pictograms in only two colors, which makes it hard to tell them apart and quickly spot their meaning. As the additional text is not always supportive either, so identifying the pictogram or opening the property dialog (see II-C) is sometimes required to fully understand the sequence.

Another interesting aspect is the difference in expressiveness of both languages. Flow charts inherently focus on the control flow of execution through statement sequences, but they have no concept to visualize statement details, except of using descriptive text. Hence, EPL has no notation to visualize expressions, assignments (e.g., set number to 10), or arguments of subroutine calls. To this end, EPL relies on a separate editor that needs to be explicitly opened in a modal dialog. Blockly, in contrast, provides an explicit graphical notation for such concepts that seamless blends in with the regular control flow (see Fig. 1).

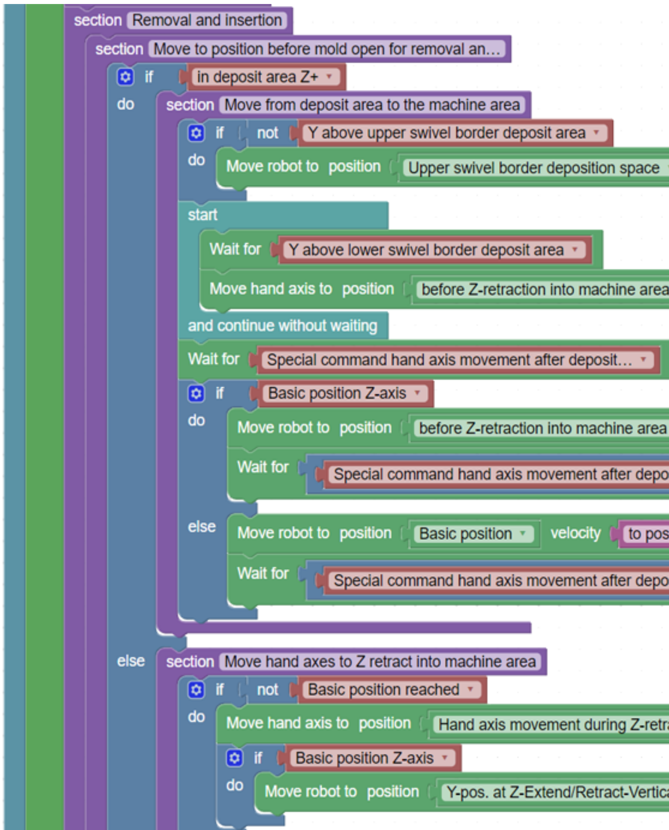

Fig. 4. ENGEL Robot Program ported to Blockly (detail, clipped).

B. Porting the Toolbox

The EPL toolbox combines two orthogonal views. One view consists of categories per domain subsystem such as the injection moulding machine, the robot, periphery (incl. conveyor belt), part arrangement options (i.e., how to place parts next to each other or on top of each other on the conveyor belt), reusable domain-specific sequences, and general structural elements (e.g., if/else, wait, while, parallel statements). The second view describes the "type" of variables: program switch (i.e., configuration flags), user-defined flags, machine flags, status flags, digital inputs/outputs, errors, current/expected robot head positions, current/expected robot arm positions, counters, and values (e.g., durations of cycles and movements). The toolbox displays in a tree-view at the top level the domain subsystem categories and below any available instances of the variable types (within each category).

For building the Blockly toolbox, we reused the custom blocks for each of the variable types (defined in the previous subsection) and dynamically loaded the available instances of these variables and their domain subsystem from a running injection moulding cell via the cell's API to its underlying programming and runtime environment (more precisely, we used a virtual twin for this purpose).

Doing so, we achieved a Blockly toolbox that mirrows the same richness and structure as the EPL editor's toolbox, allowing for user experiments under comparable conditions.

C. Implementing a realistic change

In this section we compare EPL and Blockly in the task of extending a default robot program to include an engraving station. In our case study, the purpose of the engraving station is to apply a unique identifier on a moulded part with a laser. Whether parts should be engraved should be configurable without having to edit the actual workflow (i.e., customization at level 3). The new control logic consists of the following coarse grained changes:

1) inserting a check whether engraving is enabled or should be skipped
2) waiting until the engraving station is ready
3) placing the part and moving into a waiting position
4) notifying the engraving station that a part is provided
5) waiting for the engraving station to finish
6) picking up the engraved part and moving away
7) notifying the engraving station that the part has been picked

Subsequently, the robot continues its sequence by placing the part at the configured placement area (e.g., conveyor belt). Engineers at ENGEL have confirmed that this scenario represents a typical case of how ENGEL or their customers would integrate external machines.

The changes involve a wide range of programming elements: defining new configuration variables (and checking them at runtime), waiting until signals arrive (typically achieved by waiting for a Boolean flag to become true), moving to various position, de/activating part gripper or suction cups, sending a signal to an external shopfloor participant (again typically achieved by setting a Boolean flag to true), executing parallel behavior (i.e., moving away and while doing so signaling the completion of having picked or placed a part).

This scenario assumes that the engraving duration is sufficiently short that the robot may finish placing the engraved part at its placement location and is able to pick the next produced part from the machine in time. If this is cannot be guaranteed, additional logic would be necessary to pick up the engraved part from the previous cycle, and only then place the part for engraving from the current cycle. This again requires to differentiate between the first cycle (when no previous part is available) and subsequent cycles.

Extending this scenario, we investigated the steps necessary when moving the complete graving procedure from only occurring just before placing parts on the conveyor belt to now also occurring before placing parts at the quality inspection site (but not when the part is faulty and needs to be discarded).

Revisiting the levels and required abilities defined in Section II-D, we notice that these changes occur at level 2 (i.e., additional customization is required for the precise part being handled and configuration whether engraving should be active). The changes include domain specific elements such as the robot movements and part picking/placing aspects. Interaction with unknown machinery (i.e., the engraving station is not a standard elements in an IMM cell and hence not part of

the domain specific vocabulary) occurs via setting of digital input and output signals (the Boolean flags).

In the next section, we report on early results from a small user study implementing the above described change.

D. Preliminary User study

We had one author, one research center employee, and one Master student implement the above described change in EPL as well as the same author and two Master students implement it in Blockly. Neither involved participant is a domain expert but obtained a basic introduction into EPL and Blockly. We observed (via screen sharing) what the participants did, what type of mistakes they made, and how long the individual steps of implementing the various fine-granular steps took. This setup is primarily intended to test useful evaluation tasks for a larger user study and we treat the gained insights as anecdotal evidence only. Consequently, we don't report quantitative numbers in this section but rather describe common participant behavior in a qualitative manner. These observations are not exhaustive due to the low number of participants.

Observations of using EPL:

- Finding the correct places to change a program requires expanding most subsequences (from their collapsed default visualization). Participants often lost track where in the overall robot program they currently are and needed to collapse a significant amount of structure before diving into details again.
- Participants miss-interpreted conditional statements. EPL prints the textual description of an if-condition along the horizontally diverging else-branch, thereby leading participants to confuse the else-branch with the then-branch.
- Participants often took considerable time choosing the correct element from the toolbox. One potential reason is that its not immediately clear for the programmer whether an element (e.g., a signal or variable) exists in the predefined toolbox or needs to be newly defined which is then placed in a separate category (i.e., the "custom" category). A potential second reason is that users were not sure which type of variable (of the many listed in Section IV-B) needs to be used in wait or conditional statements.
- Some variables of similar name are provided by different shopfloor participants. E.g., both IMM and disposal station provide a similar named signal that the next part should be discarded. Participants have difficulties to assess which signal is the correct one, i.e., the one that will be used at runtime.
- Conceptual mismatch of reading from a digital input (which is a generic toolbox element) and writing to a digital output (which is a custom element of the periphery category). Such mismatches cause users to select the wrong elements or spend considerable time searching.

Observations of using Blockly:

- Similar to EPL, users need to expand collapsed subsequences in Blockly. On the one hand, this lead to less overload as all elements are strictly vertically layouted, but expanding (and collapsing) subelements takes multiple clicks (into an element's context menu) as opposed to EPL's single-click activation of the "+" icon. This slows down navigating in Blockly.
- In Blockly, the element for executing one or more commands asynchronously with respect to the main control flow (i.e., "without waiting for completion do [...]") is placed above the main control flow (see Fig. 4 middle). This makes it more difficult to "detect" parallel behavior compared to EPL where such behavior is visualized as a parallel, vertical branch that branches off from the main control flow (see Fig. 3 left middle). In both languages, however, the visualization makes it difficult to perceive, which commands are then executed in parallel at exactly the same time.

V. Discussion

In this section we discuss our experiences and findings with respect to the three research questions stated in Section III.

RQ-1) Can complex real-world robot programs be expressed in Blockly and its toolbox?

In our experiments we show that with a few additional custom blocks it is possible to express real-world robot programs in Blockly without notable drawbacks. Although concepts like concurrency are not implicitly supported by the language, they can be replicated by appropriate custom blocks.

RQ-2) How does Blockly affect readability and understandability of robot programs?

Although similar in visual code size, we consider Blockly more readable than EPL due to its use of text in favour of icons, supported by colors. Nevertheless, more complex programs with nested control structures tend to become clumsy and confusing. In addition, complex expressions might blow up the code and distract the user from the control flow itself, causing more confusion. On the other hand, using the same visual syntax for both, control flow and expressions flattens the learning curve.

A non-negligible concern is representing the environment in which the program (here the robot) is interacting with. The environments complexity defines the amount of predefined elements in the toolbox and hence affects the effort and time a programmer needs to find and use desired elements. Both Blockly and EPL, however, support simple copy/paste commands to use variables and code sequences without having to access the toolbox.

RQ-3) How does Blockly and EPL support customization of robot programs?

When it comes to editing, Blockly's puzzle-like, more physical structure is advantageous. Block sequences can be selected, copied and moved around easily. As the workspace is two-dimensional, it is possible to deposit unused sequences anywhere and reuse them later.

A note on language vs language environment

The ability to change a program is often not solely determined by the language but also by the language environment (i.e., the respective editor) and often a combination of both.

A key difference we noticed between Blockly and EPL is the ability to bring the program into a syntactically incomplete/incorrect state. First, Blockly allows moving groups of connected elements out of the program flow and let them "float" on the canvas (this is possible for every element type, not just functions). EPL limits the programmer to cutting a set of elements and inserting them at another position (effectively a single slot clipboard). Elements also need to be inserted at valid positions in the flow, leaving the program always in a executable state. Second, Blockly allows statements to remain incomplete (e.g., an assignment of a variable where the value slot is left empty, or a comparison where the comparator is left empty). In EPL, this is not possible, and even if the separate pop out editor would allow this, a programmer would have difficulty spotting missing values or constructs as EPL's main flow-chart view doesn't reveal such details. In Blockly, empty slots are easily detectable. Such a shortcoming is not specific to EPL but to any flow-chart-centric language that is not well suited to visualize conditions, assignments, and the like.

The downside of requiring always syntactically correct programs is that it forces the programmer to "obey" to the program, rather than following their mental model. For example, when writing a condition expression, EPL requires the programmer to already know which variables are relevant as inspecting the program is not possible without closing the condition editor (loosing all changes when the constraint is not syntactically correct).

VI. Conclusion

In this paper we investigated Blockly, a block-based visual programming language in terms of applicability for programming industrial robots. Therefore, we ported a real-world robot program from EPL, a flow-chart based visual language, to Blockly and assessed different quality aspects like readability, understandability, and customizability. Although Blockly was invented to teach programming to beginners by simple examples, our study shows that it is possible to express even large and highly complex real-world robot programs with the language concepts offered by the block-based language. We found that visual code size and general clarity are comparable to the EPL program, whereas adaptability is better.

The gained insights are based on anecdotal evidence from a few observations. We, hence, plan to conduct a larger user study to more quantitatively compare the time and amount of errors participants make when implementing a change in Blockly and EPL, respectively.

References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & information systems engineering*, vol. 6, no. 4, pp. 239–242, 2014.

[2] G. Fragapane, D. Ivanov, M. Peron, F. Sgarbossa, and J. O. Strandhagen, "Increasing flexibility and productivity in industry 4.0 production networks with autonomous mobile robots and smart intralogistics," *Annals of Operations Research*, pp. 1–19, 2020.

[3] R. Bischoff, A. Kazi, and M. Seyfarth, "The morpha style guide for icon-based programming," in *Proc. 11th IEEE Intl. Workshop on Robot and Human Interactive Communication*. IEEE, 2002, pp. 482–487.

[4] C. J. Sutherland and B. A. MacDonald, "Naoblocks: A case study of developing a children's robot programming environment," in *2018 15th Intl. Conference on Ubiquitous Robots (UR)*. IEEE, 2018, pp. 431–436.

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

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

[7] J. Blume, "iprogram: intuitive programming of an industrial hri cell," in *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 2013, pp. 85–86.

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

[9] N. Ritschel, V. Kovalenko, R. Holmes, R. Garcia, and D. C. Shepherd, "Comparing block-based programming models for two-armed robots," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[10] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: blocks and beyond," *Communications of the ACM*, vol. 60, no. 6, pp. 72–80, 2017.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010.

[12] N. Fraser, "Ten things we've learned from Blockly," in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 2015, pp. 49–50.

[13] J. Trower and J. Gray, "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*, 2015, pp. 5–5.

[14] D. Topalli and N. E. Cagiltay, "Improving programming skills in engineering education through problem-based game projects with scratch," *Computers & Education*, vol. 120, pp. 64–74, 2018.

[15] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: urban youth learning programming with scratch," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, pp. 367–371.

[16] D. Weintrop, D. C. Shepherd, P. Francis, and D. Franklin, "Blockly goes to work: Block-based programming for industrial robots," in *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 2017, pp. 29–36.

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

[18] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[19] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 73–82.

[20] K. M. Adams, "Adaptability, flexibility, modifiability and scalability, and robustness," in *Nonfunctional Requirements in Systems Analysis and Design*. Springer, 2015, pp. 169–182.

[21] E. Arisholm, "Empirical assessment of the impact of structural properties on the changeability of object-oriented software," *Information and software technology*, vol. 48, no. 11, pp. 1046–1055, 2006.

[22] M. Winterer, C. Salomon, J. Köberle, R. Ramler, and M. Schittengruber, "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 8-11, 2020*. IEEE, 2020, pp. 1231–1234.