Comparing Block-based Programming Models for Two-armed Robots

Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ronald Garcia, and David C. Shepherd

Abstract—Modern industrial robots can work alongside human workers and coordinate with other robots. This means they can perform complex tasks, but doing so requires complex programming. Therefore, robots are typically programmed by experts, but there are not enough to meet the growing demand for robots. To reduce the need for experts, researchers have tried to make robot programming accessible to factory workers without programming experience. However, none of that previous work supports coordinating multiple robot arms that work on the same task. In this paper we present four block-based programming language designs that enable end-users to program two-armed robots. We analyze the benefits and trade-offs of each design on expressiveness and user cognition, and evaluate the designs based on a survey of 273 professional participants of whom 110 had no previous programming experience. We further present an interactive experiment based on a prototype implementation of the design we deem best. This experiment confirmed that novices can successfully use our prototype to complete realistic robotics tasks. This work contributes to making coordinated programming of robots accessible to end-users. It further explores how visual programming elements can make traditionally challenging programming tasks more beginner-friendly.

Index Terms—Programming environments, User interfaces, Robot programming, Parallel programming, Block-based programming

1 Introduction

Robot technology has advanced substantially and now supports robots that can safely work alongside humans. These robots, called *collaborative robots*, work faster and better than either humans or robots alone [1], [2]. Collaborative robots have become cheap and effective enough to make them suitable for many tasks where robots were previously uneconomic [3]. When multiple collaborative robots interact with each other, they can solve tasks even more effectively [4]. For example, they can hold, weld, screw or fold a single object in parallel, or solve new tasks, like jointly lifting heavy loads.

Unfortunately, collaborative robots are difficult to program, and doing so requires complex programming tools. When multiple robot arms need to interact, they cannot be programmed independently. There must be coordination between the computations on collaborating robots, which adds further complexity. For this reason, robots are programmed by experts, but expert programmers are expensive and there are not enough of them available to program all robots [5].

If the workers collaborating with the robots could program the robots themselves, then there would be enough programmers, and it would be cheaper. But these workers are usually end-users without programming experience or education. They need programming tools that are easier to use and learn.

Researchers have tried to create easy-to-use programming tools. Much of this work uses block-based programming languages. Block-based languages appeal to beginners due to their graphical, friendly design. Block-based languages make it easier for beginners to learn programming [6]. There are also block-based languages for programming collaborative robots [7]. However, coordinated programming of multiple collaborative robot arms is not supported by block-based languages.

In this paper, we explore novel techniques to support coordinated programming in block-based languages. After creating several candidate designs, we conducted a preliminary study of whether end-users could understand the behaviour of coordinated programs presented to them. We found that they best understood those designs where the program layout resembles the physical layout of the collaborative robot. Based on this finding, we derived four candidate design options that differ with respect to their synchronization model and program flow presentation. Choosing an explicit synchronization model significantly extends the expressiveness of the designs compared to using implicit synchronization. The presentation of program flow does not affect the semantics or expressiveness of the designs.

For each of our candidate designs we identified potential benefits and trade-offs in terms of program comprehension and usability. We empirically evaluated these trade-offs through a survey of 273 professional users like engineers, managers, and robot workers (110 without previous programming experience). We found that participants understood all variants equally well, but gave higher usability ratings for those that present program flow vertically. Figure 1 shows these findings mapped to the two-dimensional design space we explored in our survey. We built a functional prototype of the design that uses explicit synchronization and vertical program flow. An interactive experiment on 11 industrial participants confirmed that most could use our prototype to successfully write coordinated programs for a multi-armed robot.

- *N. Ritschel, R. Holmes and R. Garcia are with the Department of Computer Science, The University of British Columbia, Vancouver, BC V6T 1Z4 Canada.*
- *V. Kovalenko is with JetBrains Research, JetBrains N.V., 1017 ZM Amsterdam, The Netherlands.*
- *D. Shepherd is with the College of Engineering, Virginia Commonwealth University, Richmond, VA 23284 USA.*

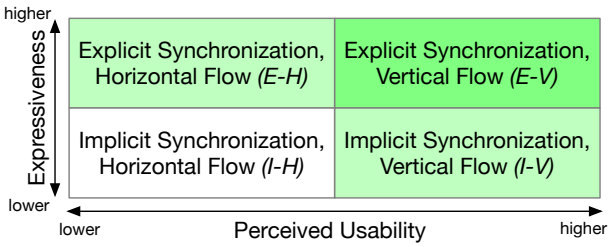


Fig. 1: Design space based on synchronization model and program flow direction. Each of the designs shown in Figures 6, 7, 8 and 9 corresponds to one cell of this table.

2 Related Work

Our work draws ideas from robot programming, visual programming and parallel programming. It further builds on established technology from end-user programming and computer science education. In this Section we discuss these influences on our work.

2.1 End-user Robot Programming

Most robot programming tools expect users to have a background in both computer science and robotics. Even these users need extensive training to program robots effectively [8].

Biggs and MacDonald surveyed tools that try to make robot programming more accessible for beginners [8]. They categorized the approaches taken by these tools as *manual* or *automatic*. Manual tools use domain-specific languages that are simpler, have more scaffolding and use higher-level commands than expert languages. Automatic tools on the other hand try to eliminate the need for programming altogether. One example of an automated approach allows users to move the robot arm by hand and then replay the movements later [5]. This approach, called *demonstration-based learning*, has also been combined with object, task or gesture recognition to make it even more effective [9].

Even beginner-friendly tools can have a range of target audiences. Some systems, like *Lego Mindstorms EV3* [10] and *MORPHA* [11], target absolute novices with no programming experience. Other systems, such as *Polyscope* [12], are more complex and target intermediate users. Systems for intermediate users not only allow users to write more complex programs but also to target more advanced robots.

This work was heavily influenced by *CoBlox*. CoBlox is a programming environment designed to allow beginners to program collaborative robot arms [7]. In CoBlox, users can manually move the robot arm to its intended location and then capture this position to use it in a block-based program. CoBlox was the first environment to combine a manual programming environment with techniques from demonstration-based learning. The authors of CoBlox have evaluated it and found that it is easier to learn and use for end-users than other commercially available tools [7].

CoBlox uses a simple domain-specific programming language. All programs are linear sequences of blocks that move the robot arm or hand. CoBlox only supports programming one robot arm at a time and cannot express coordination among arms. It is therefore not suitable for programming multiple robot arms that work in tandem.

2.2 Block-based and Visual Programming

The success of environments like *Scratch* [13] and *Blockly* [14] made block-based programming a popular topic of research. Studies have shown that block-based languages can effectively teach programming concepts to novices [15] and can be easier to learn than text-based languages [6]. Recent studies suggest that these strengths also apply to block-based robot programming, both in education [16] and industry [7].

Block-based languages are typically described as "graphical" [13] or "visual" [17]. This suggests that they are related to the research area of *visual programming languages* [18]. Block-based languages use similar techniques to visual programming languages: They use visual channels like color and shape to encode information, and they use graphical user interactions like drag-and-drop. However, they do not apply the same overall strategies as visual programming languages: Unlike visual languages, they do usually not attempt to use a *concrete* visual presentation of data in its domain. They further do not support the *direct* manipulation of data, or try to make semantic information more *explicit* [19]. Only some block-based languages, like Scratch, attempt to give users more immediate, *live feedback* than traditional text-based languages [20]. Block-based languages are text-based languages, where the text is encapsulated in graphical blocks. We therefore do not believe that block-based languages should be categorized as visual programming languages.

The authors of most block-based languages expect users to eventually transition to text-based programming [21]. This is because they are often used in computer science education. Their goal is to prepare students to program in traditional programming languages. This explains why block-based languages do not use the same strategies as visual languages: If they look too different from text, it is harder for users to transition to text-based languages later.

This work targets industrial end-users. Most industrial end-users do not intend to learn programming beyond what is necessary for their current task. Therefore we do not need end-users to be able transition to text-based programming languages. We can apply all of the previously mentioned visual design strategies to our language design.

2.3 Parallelism in Block-based and Visual Languages

Coordinating multiple robot arms is very similar to coordinating parallel programs. In both cases, concurrent actions need to be executed with specific timing constraints to serve their intended purpose. Parallel programming is typically considered an advanced topic in computer science education. It is only taught after students have already mastered sequential programs. We want to enable novices to learn sequential and coordinated programming at the same time.

Previous block-based programming languages have used a variety of techniques to make parallel programming beginner-friendly. Environments like Scratch [13] support concurrent, event-driven programming for handling user interactions. The *Parallel Snap!* [22] environment for programming distributed systems supports domain-specific high-level abstractions such as MapReduce or the producer-consumer problem. This allows these environments to cover

Fig. 2: Parallelism in Alice. While multi-threading and synchronization are abstracted into a single "do together" block, the nested, linear structure can make programs hard to read.

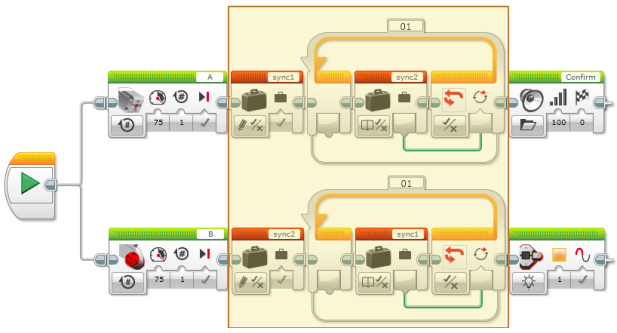


Fig. 3: Parallelism in Lego Mindstorms EV3. The sole purpose of all blocks highlighted in yellow is to synchronize the top and bottom row.

one specific type of parallelism very effectively. The block-based language *Alice* on the other hand uses a special "do together" block that enables generic multi-threading [23]. An example program that uses one is shown in Figure 2. Alice supports a wide range of parallel programs but also provides less scaffolding and visualization than Scratch or Parallel Snap!, making it potentially harder to learn.

Visual programming languages have also aimed to simplify parallel programming. The graph-based visual robotics language Lego Mindstorms EV3 allows each command node to have multiple outgoing edges. All nodes connected in this way are executed in parallel [10]. Figure 3 shows an example program that uses EV3. Both the top and the bottom row of the program are executed simultaneously.

An advantage of EV3's approach is that programs can usually be arranged to look like a timeline from left to right or top to bottom. However, this requires that the users manually arrange each node on the canvas to match the intended program flow. To ensure that two commands are executed simultaneously, users further need to manually synchronize them: They need to manually add a complex barrier construct (highlighted in yellow in Figure 3). Besides adding visual clutter, this also requires users to correctly use other non-trivial language features like variables and loops. Other visual data-flow languages that support parallel programming, like *VIVA* [24] or *CODE* [25], have similar design issues as EV3.

The block-based and the visual programming languages we described have significant drawbacks. Some only support specific forms of parallel programming that are not fit for coordinating robots. Others do not provide effective visualization, or require significant manual effort from the user. We aim to support better visualisation and propose a language design that fits the needs of robot programmers.

3 Design Considerations

The actions of multiple robots can be coordinated in many ways. The optimal choice depends on the task that the user is trying to complete. In this work we focus on coordinating two robot arms. Two robot arms can be coordinated in two distinct ways:

- **Synchronous coordination** - Some tasks require both robot arms to move at the same time and speed, although potentially in different directions. Examples for such tasks are carrying, turning, folding or tearing an item.
- **Asynchronous coordination** - Other tasks require both robot arms to move independently, potentially waiting for each other at some point before continuing. Examples for such tasks include handing an item from one arm to another, and holding an item in place with one arm while the other one stirs, screws, or welds.

In this section we present our process of designing a programming language that is end-user friendly and supports both synchronous and asynchronous coordination. We conducted small-scale preliminary studies with novice programmers to guide our design work. We then conducted a larger empirical evaluation of the resulting designs candidates. In this Section we focus on our analytical design considerations and the decisions they motivated.

The *13 Cognitive Dimensions of Notation (CDN)* [26] are a popular framework for analyzing visual languages. This framework describes how users understand and interact with with different visualizations. The CDN provides terminology to effectively describe and compare many different types of visual languages. We use its terminology in the following sections when describing our design decisions.[1]

3.1 Showing Coordinated Programs Side-By-Side

Robot programmers often have to write code based on imprecise, high-level task descriptions. They need to simultaneously determine the necessary steps for each robot arm and the necessary coordination between them. For beginners, this process can be challenging even for tasks where no coordination is necessary. The first goal of our design process was therefore to find ways for novices to effectively read and write code for two independent robot arms.

In early design drafts, we used a single, linear program where each command had a parameter that specified which arm should move. We also tried using separate `Move Right Arm` and `Move Left Arm` blocks. Both designs intertwine the movements of the two arms and force users to distinguish them in their heads. This is a *Hard Mental Operation*. Further, these designs also reduce the *Visibility* of each arm's individual actions.

1. We mark references to CDN terminology by underlining them.

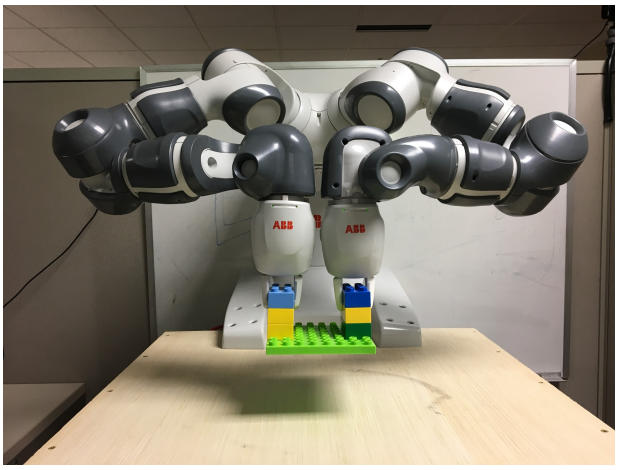


Fig. 4: Two-armed robot carrying an item with two arms.

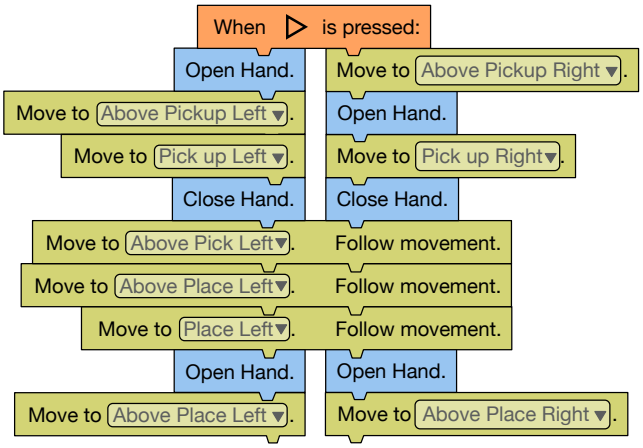


Fig. 5: Example program using two columns and top-to-bottom program flow. Both robot arms jointly pick and place an object using synchronous two-armed movements.

For our next design iteration, we considered the perspective of the user when programming and testing. Figure 4 shows the users' typical view of a two-armed robot: They are facing the robot from the front with one arm on the left and one on the right. To mimic this perspective in our programming language, we decided to separate the programs for each arm and place them side-by-side. To connect both programs, we use a single block that spans both columns and has two tabs for successor nodes. This integrates well with the interlocking jigsaw aesthetic of block-based languages. The resulting layout can be seen in the first 5 rows of Figure 5.

One important detail of the design shown in Figure 5 is that it requires slightly different blocks for each arm. For example, the `Open Hand` blocks in the two columns need to have their jigsaw tabs aligned either on the left or the right of each block. Using different blocks for each arm means that program fragments cannot be easily moved between arms. This leads to an increased *Viscosity* of our language. Therefore it is important to automatically adapt the blocks depending on which arm it is assigned to. One way to do so is to swap the alignment of blocks based on the side of the canvas they are placed on.

3.2 Synchronous Coordinated Movements

Two columns are an effective way to present two independent programs side-by-side. However, our goal was to support coordinated tasks where the robot arms don't act independently.

Consider a task where the robot in Figure 4 is supposed to place the held Lego piece on the table. This task requires both arms to move downward simultaneously. This could be implemented by two side-by-side `Move` blocks, one for each arm. However, a movement like this requires precision to make sure that timing, speed and distance of both arms fit exactly. This increases the system's *Error-proneness*. It also leaves the dependency of the two movements implicit, creating a *Hidden Dependency*.

To simplify defining synchronous movements across arms, we introduce new block types that span both columns. The first type, that we refer to as a `Follow` block, is shown in Figure 5. This block lets the user define the target location for one arm and computes the other arm's movements automatically. It makes both arms move in parallel by ensuring that their distance and speed are matched. The second new block type is a `Mirror` block. It also matches the speeds of the arms but makes them move in opposite directions. This is necessary for tasks like folding of objects.

An advantage of both new block types is that they fit well how beginners think about synchronous movements. Their designs and labels match the verbal descriptions that early participants intuitively gave for synchronous movements. This suggests that the *Closeness of Mapping* between these blocks and the way beginners think about tasks is beneficial for linking the code to the physical domain.

3.3 Timing and Synchronization of Asynchronous Movements

A typical robot program like the one in Figure 5 combines both synchronized and individual commands. When the arms move independently, the actions that are presented side-by-side may not always take the same amount of time. Later commands may however require a specific order or timing of arm movements. At some point, one of the two arms may need to wait for the other one to achieve this timing.

We showed our design draft as presented in Figure 5 to a small group of novice participants. We explained to them that the robot might take significantly less time to open its hand than it does to move its arm. We then asked them about their interpretation of the timing of the following commands. The answers of most participants suggested that they read the program as a timeline. They assumed that each row of blocks, starting from the top and moving down, is executed simultaneously for both the left and the right robot arm. This means that they expected the arms to wait if one is faster than the other.

Based on this feedback, we drafted a design that *implicitly* synchronizes the arms after each row of blocks. We added `Wait` blocks in cases where only one arm is required to move. These blocks ensure that all blocks are connected in the jigsaw-style of block-based languages. An example

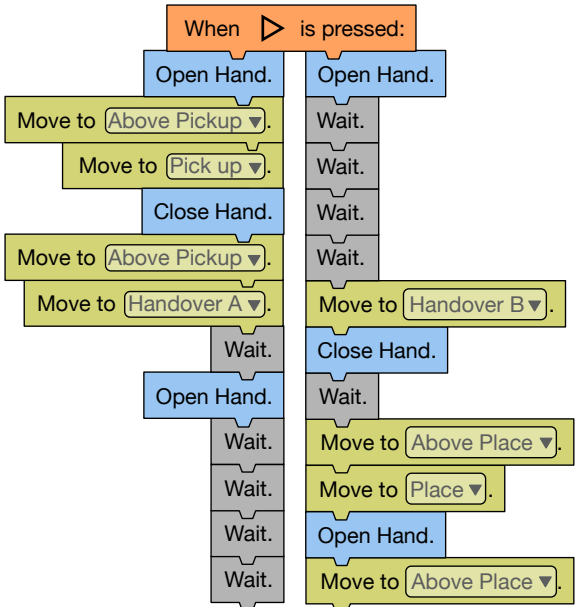


Fig. 6: Example program using implicit synchronization and vertical program flow *(I-V)*. One robot arm hands over an item to the other. Arms wait for each other after each command.

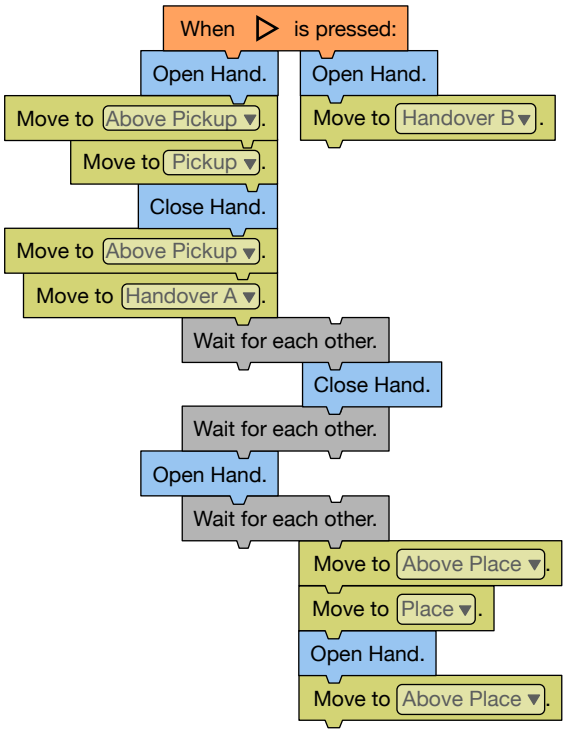


Fig. 7: Equivalent program to the one shown in Figure 6, using explicit synchronization and vertical program flow *(E-V)*. Arms are acting independently and only synchronized when they reach grey `Barrier` blocks. This program was used in Task 3b of our survey.

program using this design is shown in Figure 6. It tells one robot arm to pick up an item and hand it to the other arm, and then tells the other arm to grab it and put it down.

The design shown in Figure 6 has a significant limitation: Since every block has exactly one counterpart in the other arm's program, only one command can ever be executed simultaneously with another command. If a short command is combined with a long one, this can introduce unnecessary wait times. An example for this is the first four rows of the program in Figure 5: While the first four commands for each arm require no coordination, the arms would still wait for each other after every step. Therefore, the example program as it is written here wastes time whenever arm movements take longer to execute than opening a robot arm's hand. While efficiency could be improved by swapping the order of commands, this may not be obvious to beginners. There are also other cases where the order of commands is relevant and the design is not expressive enough to write more efficient programs.

A second drawback of the presented design becomes visible in Figure 5. If only one arm should be active for an extended period of time, multiple `Wait` blocks must be inserted for the other arm. This is necessary to retain *Consistency* with the block-based jigsaw design, which does not allow gaps between blocks. However, it adds visual clutter to the design and increases its *Diffuseness*. It can further reduce the Visibility of those command blocks that describe active robot behavior.

We considered ways to solve the problems of the presented design while keeping it readable like a timeline. One way is to modify the height of blocks based on how long they take to execute. Another is to add indicators that warn users about wait times. However, in practice it is not possible to predict the exact length of each robot command. It is even possible that the same command takes a different amount of time between multiple runs. We cannot therefore rely on this information.

An alternative approach to synchronization can be found in traditional parallel programming: barriers can force threads to wait until all of them have reached the same point in program execution. They synchronize concurrent programs *explicitly*. We have created an alternative design based on this approach. It does not require blocks to wait for each other after each row, but uses `Barrier` blocks instead. Figure 7 shows the hand-over program from Figure 6 rewritten in this design.

A design with explicit barriers allows the execution of any number of commands in parallel. This makes it strictly more expressive than the design shown before in Figure 6. It comes however with the drawback that programs cannot always be read as a timeline. Take the program from Figure 5 as an example: When interpreted based on this design, it is not clear which of the first commands for each arm are executed simultaneously.

As Figures 6 and 7 illustrate, programs may only need a small number of `Barrier` blocks. This is especially the case since two-column blocks like `Follow` blocks also act as a barrier. Users must think about timing when it matters for the program's correctness. This makes the Diffuseness of this design lower than for the previous design. However, this does not necessarily make programs shorter, as Figures 6 and 7 show.

The two presented designs that use implicit and explicit

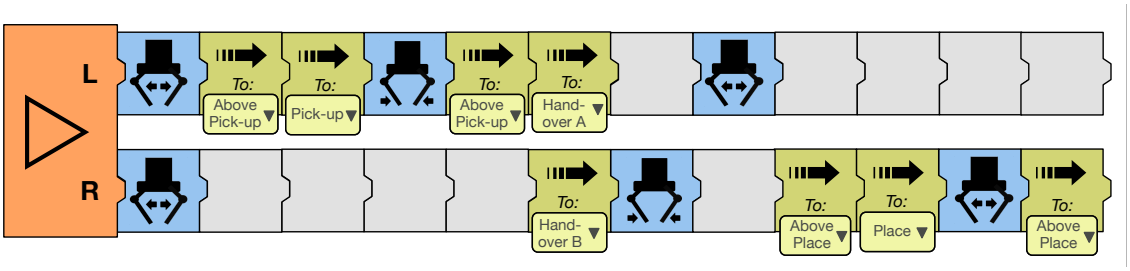


Fig. 8: Equivalent program to the one in Figure 6, using implicit synchronization and a horizontal program flow *(I-H)*.

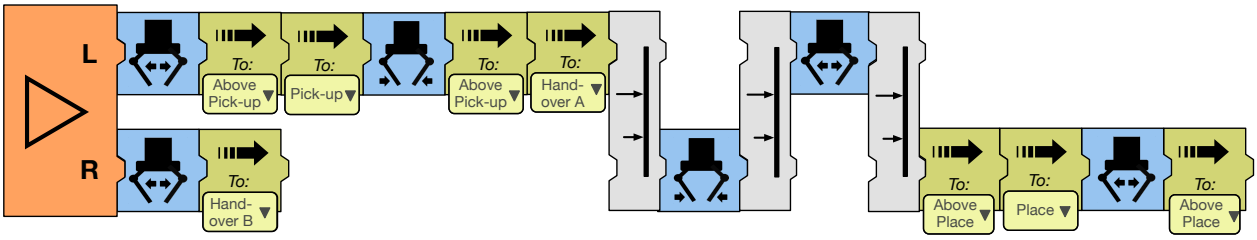


Fig. 9: Equivalent program to the one in Figure 6, using explicit synchronization and a horizontal program flow *(E-H)*.

synchronization both come with individual trade-offs. An analytical approach cannot reliably determine which of the two design alternatives is easier to comprehend. Answering this question empirically motivated us to conduct the survey we present in Section 4.

3.4 Vertical vs. Horizontal Program Flow

Our preliminary studies indicated that beginners intuitively read side-by-side programs, such as shown in Figures 5, like a timeline. This intuition might be influenced by previous experiences with widely available commercial tools for audio, video or animation editing. These tools use similar timeline visualizations and show events occurring in parallel on multiple time-aligned tracks.

Unlike our previously presented design drafts, most other tools present time as flowing horizontally. We are not aware of research that investigated if mapping time to a vertical or horizontal axis is more effective. We assume that most applications choose a horizontal design to maximize screen space or for other practical, layout-related reasons. Nonetheless, the likely pre-exposure of many users to horizontal timeline visualizations might influence their intuitions and comprehension.

Figures 8 and 9 show horizontal versions of the programs we previously presented in Figures 6 and 7. As the figures show, a horizontal layout requires more modifications than simply rotating the code: Since English text flows from the left to right, labels that are placed next to each other horizontally consume significantly more space than when they are placed vertically. This leads to screen space being used less effectively and drastically reduces the Visibility of the overall program context. We have therefore decided to use icons instead of text-based labels for all commands, with the exception of user-named locations.

Previous research has shown that icons are less effective for comprehension than textual labels [27]. However, our design drafts use fewer different symbols than this previous work. We therefore speculated that there might be an acceptable trade-off between using icons and and being able to present time flow horizontally.

In addition to icons, the horizontal design comes with another trade-off: Unlike in the vertical design, the two rows of the horizontal layout do not directly correspond to the positions of the left and right robot arm. Mapping the top row to the left arm and the bottom to the right one might be a Hard Mental Operation for users.

The trade-offs between the vertical and horizontal design alternatives are similar to those of the different synchronization alternatives. It is hard to draw conclusions by only using an analytical approach. We have therefore decided to evaluate the impact of both the synchronization model and the program flow orientation in the comparative survey we present in Section 4.

4 Comparative Survey

In Section 3 we presented four candidate designs for a novel robot programming language. We designed all of them to be novice-friendly and enable users to coordinate two interacting robot arms. They differ in how they model synchronization between arms and in how they present program flow. We discussed analytically how these differences might affect the comprehensibility of each design. However, we also wanted to empirically validate how well potential users can comprehend and use each design to decide which one is worth further development. To conduct a study on a larger scale, we designed an automated survey that we could distribute to a large number of participants.

The leftmost column in Table 1 shows an overview of our survey structure. We started with a brief demographic questionnaire to classify participants as novices and experts. We then asked them to complete three tasks to measure how well they are able to comprehend our design candidates. Finally, we asked the participants to rate the usability of the designs they saw and indicate which factors influenced their ratings.

Introduction	*273 Participants Started Survey*							
Demographics	110 Novices				*163 Experts*			
Task 1	50 Vertical		60 Horizontal		*77 Vertical*		*86 Horizontal*	
Task 2	45 Vertical		53 Horizontal		*55 Vertical*		*68 Horizontal*	
Task 2 (alt. design)	41 Vertical		50 Horizontal		*54 Vertical*		*65 Horizontal*	
Task 3	22 I-V	14 E-V	25 I-H	23 E-H	*20 I-V*	*31 E-V*	*28 I-H*	*32 E-H*
Task 3 (alt. design)	22 I-H	14 E-H	25 I-V	23 E-V	*18 I-H*	*31 E-H*	*27 I-V*	*31 E-V*
Usability Questionnaire	84 Novices Completed Questionnaire				*107 Experts Completed Questionnaire*			
Completed	*191 Participants Completed Survey*							

TABLE 1: Survey flow (from top to bottom) with participant numbers for each design treatment. Participants were first categorized as novices and experts and then randomly assigned one of the designs shown in Figures 6, 7, 8 and 9

4.1 Research Questions

In this Section we present the research questions guiding our survey. We further show how the overall structure of the survey contributes to answering each question. We then provide a more detailed description of each component of the survey.

RQ1: How well do novices comprehend our candidate designs?

As part of our design process we created mock-ups of sample programs written in each design candidate, like those shown in Figures 6, 7, 8 and 9. We used these mock-ups to validate our designs. We designed a series of three tasks, each based on one example program, and a series of comprehension questions. We divided participants into randomized groups that were each shown a different design candidate throughout the survey. By grading each participant's results for these tasks, we measured how well each design allowed them to comprehend realistic example programs written in it.

The first task did not use parallelism and was intended to introduce participants to block-based robot programming. The second task used synchronous parallelism and was based on the program shown in Figure 5. The third task used asynchronous parallelism and was based on the program shown in Figure 6.

RQ2: Which design(s) do novices prefer?

We see the differences in comprehensibility as a major factor in determining the quality of our designs. However, users may prefer a design for reasons unrelated to their ability to understand it. We therefore asked participants to fill out a short questionnaire asking how usable the prototype was.

We also wanted to give participants the opportunity to directly compare the designs. We showed each participant one alternative to the design they were originally assigned and let them rate its usability. We also asked them which one they would prefer overall. Table 1 shows the order in which each participant group was shown the tasks and design alternatives.

RQ3: Which factors influence the preference and overall opinion of novices?

We also wanted to find out which factors contributed the most to each participant's preferences. We pre-selected a number of differences, such as program flow orientation or using icons vs. text as labels, and asked participants how important each of these differences was to their decision. We also gave them the opportunity to name other factors and to provide us with an open-ended explanation of their decision. These questions were part of the usability questionnaire shown in Table 1.

RQ4: Do experts have different preferences than novices?

While our language designs are intended to be novice-friendly, they should also be usable by expert programmers. Since experts may have different needs and preferences than novice end-users, we included both in our study. We define novices as users with no significant experience with any programming language, no previous robotics programming experience, and no professional role related to software development. We end our analysis by comparing the preferences of both groups.

4.2 Recruitment

To recruit participants for our survey we reached out to both employees of a large, multi-national robotics company, and the broader public by advertising in online communities focused on engineering. We had 313 responses, with participants from a wide range of professional backgrounds (e.g. engineers, software developers, administrators). The results we present in Section 5 are based on those 273 participants who gave consent to have their data collected and who completed at least one task of the survey.

4.3 Study Protocol

We wanted to evaluate the participants' ability to comprehend programs written in our candidate designs on both a syntactic and a semantic level. For this purpose we designed three tasks with increasingly complex sample programs that were inspired by realistic usage scenarios. We divided participants into randomized groups, each of which was shown a mock-up based on a different design candidate matching one of the cells in Figure 1. For each question, we also gave participants a use case description. We then asked them to answer a series of questions to evaluate their comprehension. We intentionally focused on questions related to timing and parallelism, as these are the novel aspects of our design.

Before showing our participants the first task, we gave them an introduction to robot programming and the kind of robot that our designs target. We showed participants a one minute video. In this video a researcher makes the robot

perform a simple pick-and-place procedure with one arm. We did not show any programming interface to ensure that we didn't bias participants.

To avoid overwhelming participants, we ordered the three tasks following this video by increasing complexity. Each task consisted of a use case description, a mock-up program and a number of comprehension questions. Example questions were "select all rows/columns during which the right robot arm carries the cube" or "which blocks need to be modified to change the cube's target destination". Participants were allowed to leave questions unanswered and continue with the rest of the survey.

After the second and third tasks, we showed participants an alternative mock-up using a different design candidate. This prepared them to compare the designs during our follow-up questionnaire. To avoid confusion between the shown designs, we decided not to show participants the alternative synchronization model but only the alternative program flow direction. Table 1 shows the resulting order in which each participant group was shown each design.

4.4 Comprehension Tasks

Task 1 was designed to be a warm-up task that only uses one arm and gives participants a chance to get familiar with the block-based programming environment. We showed participants a one-armed pick-and-place routine. A similar routine was used by Weintrop et. al. in their CoBlox experiments [7]. Since this task did not contain any coordination, we only distinguish between two different designs for this task, based on the used program flow direction. We asked two questions to determine whether participants had a basic understanding of the environment, one question about the (completely linear) timing of commands, and one question about how they would change the item's target destination.

Task 2 used parallelism, but was intentionally similar to Task 1 so that participants could identify the same overall structure. The vertical mock-up for this task was identical to the one presented in Figure 5. We asked participants two questions to evaluate if they had a basic understanding of the function of rows and columns in the given program. Then we asked them to identify what the purpose of the given program was. Finally we asked them two questions about the timing of parallel commands. The first of these questions allowed two possibly correct answers, based on the parallelism model the participant intuitively assumed. The second question then verified if participants were applying their chosen model consistently to the whole program.

Task 3 was the most complex task. It showed participants an uncommented 30 second video of a robot arm handing an item to another arm, and an incorrect program attempting to reproduce this behaviour. Figures 6 through 9 show the correct versions of this program in each of the evaluated design alternatives. The version used in the survey however had the `Open` and `Close` block in the middle of each program swapped. This error would cause one arm to drop the held item before the other one could grab it. We asked participants one general question about the timing of the program that was independent of the contained error. We also asked two questions about identifying the erroneous blocks. The final question suggested ways to fix the error, and asked participants to identify at least one that could work.

4.5 Usability and Preference Survey

To capture participant preferences, we solicited their opinions of the two design alternatives they were shown. After working on all three tasks of our survey, we asked them to rate the usability of each design via standardized questions from the *System Usability Scale (SUS)* [28]. Since some of the SUS questions only apply in a context where users are able to use a system actively, we asked only 6 of the 10 questions defined by the SUS and then computed an overall score. According to previous research [29], this gives results that are comparable to the full questionnaire after scaling the resulting score.

Besides rating usability, we also asked participants to indicate their overall preference between the two used design alternatives on a 5-point Likert scale. We also asked how important some of the design differences were to their preference. We further allowed participants to list other factors for their decision. We asked about three major differences between the vertical and horizontal designs: the direction of program flow, the type of labeling used for blocks (icons vs. text) and the uniformity (or lack of uniformity) of block sizes. Since each participant used either implicit or explicit synchronization, we rely on the usability survey to compare these design types.

5 Survey Results

In this Section we present the results of the comparative survey from Section 4. All of our results are based on the 273 participants who both gave consent to their data being collected and completed at least one of the tasks assigned to them. We classified 110 participants as novices and 163 as experts. The numbers of participants that finished each individual task are listed in Table 1. A total of 191 participants (84 novices and 107 experts) finished the entire survey.

5.1 RQ1: How well do novices comprehend our candidate designs?

Based on the 110 novice responses, Table 2 summarizes the distributions of comprehension scores for each task and design. The table also shows the average scores for each task. The overall average scores of participants on Task 2 and 3 were high, even though these tasks were non-trivial. However, novice participants performed substantially worse on Task 1 than Tasks 2 or 3.

We did not expect participants to perform worse on Task 1, since this task was intended to be the easiest and did not involve any form of coordination between arms. A closer inspection of the results shows that the low overall score can be attributed to a single comprehension question that only 32% of all novices answered correctly. This question asked participants how they would change the target destination of an object carried by the robot arm. It was the only question that involved the manipulation of locations. We assume that a lack of training on the meaning of locations is the most likely cause for this finding.

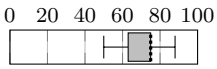
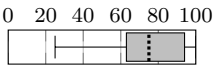
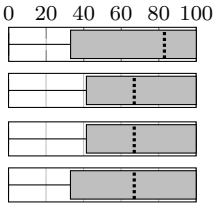
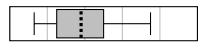
Novice Participants	Task 1: 1-armed Pick and Place	Task 2: 2-armed Pick and Place	Task 3a: 2-armed Handover
Implicit, Vertical			
Explicit, Vertical			
Implicit, Horizontal			
Explicit, Horizontal			
	Overall average: 55%	Overall average: 75%	Overall average: 67%

TABLE 2: Box-plots[2] of the scores of novices for each task's comprehension questions. Participant numbers (n) are reported in Table 1. Novices performed well on all tasks except Task 1 when using the horizontal design.

Participants who used the horizontal design performed worse on average on Task 1. The difference is substantial and statistically significant (29% worse; independent samples t-test assuming equal variances finds $t(108) = 90.54$, $p < .0001$). For Tasks 2 and 3, there is also a difference in the average values, but it is smaller and not statistically significant (<10% worse; independent samples t-test assuming equal variances for Task 2 finds $t(96) = 1.06$, $p = .15$; one-way ANOVA for Task 3 finds $F(3, 80) = 0.22$, $p = .88$).

RQ1 Summary: The data shows that participants' performance was high for all tasks, except for the warm-up task, independently of the used design.

5.2 RQ2: Which design(s) do novices prefer?

We measured participants' preferences using the SUS scale (see Section 4). The SUS scale, ranging from 0 to 100 points, cannot be interpreted linearly. Based on meta-studies, we can however convert SUS scores into a percentile rank which compares our designs to other systems that were evaluated using the same scale. While we used only 6 out of the 10 questions of the standard questionnaire, previous research indicates that the scaled scores are still equivalent to the same percentiles [29].

Figure 10 shows the raw scores of our designs on the SUS percentile curve. Meta-studies find that the average SUS score across studies is 66 [29]. The scores of all our designs are above this average, ranging from the 60th to the 80th SUS percentile. There is almost no difference (<1 point) between the average SUS scores of the designs using implicit and explicit synchronization. We do see however a substantial and statistically significant 8-point difference in favor of the vertical designs over the horizontal ones. A one-way ANOVA finds $F(3, 164) = 5.41$, $p = .0014$. A post-hoc Tukey HSD finds that vertical designs are always better than horizontal ones (all $p < .05$).

In addition to calculating SUS scores, we also asked participants directly which design they prefer. Of all participants, 57% answered that they slightly or strongly preferred the vertical design they were shown. Another 39% slightly or strongly preferred a horizontal design. Remarkably, only 4% of all participants had no preference at all.

RQ2 Summary: The data shows that the majority of novice participants preferred the vertical program flow, but have no preference between synchronization models.

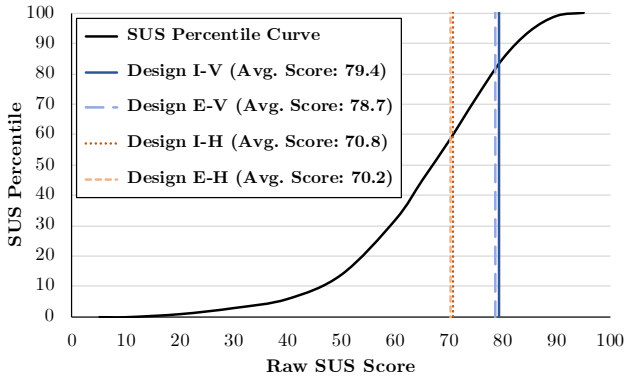


Fig. 10: SUS scores computed for novice participants' responses, plotted on the SUS percentile curve. Participants rated the vertical design candidates significantly better.

5.3 RQ3: Which factors influence the preference and overall opinion of novices?

Pre-Selected Factors: We showed all participants two designs: The one we assigned to them and an alternative one that uses the same synchronization model but a different program flow orientation. We asked them which factors had the biggest influence on their preference ratings. The first row of Table 3 summarizes how participants rated three pre-selected design differences on a 5-point Likert scale. Overall, participants found the differences in labeling (text vs. icons) to be the most important factor, closely followed by the orientation of program flow. It seemed less important for participants whether blocks had a uniform size.

We conducted two additional analyses: First, we investigated if participants that preferred different designs had different factors that influenced them. We grouped the factors by preferred design as shown in Table 3. An independent-sample t-test assuming equal variances finds no statistically significant difference between the two groups for any factor (from left to right: $t(78) = 0.35$, $p = .36$; $t(78) = 0.86$, $p = .20$; $t(78) = 0.52$, $p = .30$). Second, we verified that participants did not always prefer their assigned design over the alternative they were shown later. A one-way ANOVA finds no statistically significant difference between participants based on which design they were assigned ($F(3, 77) = 0.27$, $p = .85$).

2. For all box-plots in this paper, we use center lines to show the medians, box limits to indicate the 25th and 75th percentiles and whiskers extending to the 5th and 95th percentiles.

	Orientation	Icons/Text	Uniformity
	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5
Overall			
Pref. VT			
Pref. HR			

TABLE 3: Perceived importance of individual design aspects for novice participants, overall and grouped by their preferred design: 1 is least important, 5 is most important.

Qualitative Feedback: In addition to rating the importance of pre-selected design differences, we allowed participants to provide written feedback on what they liked about their preferred design. Almost all participants used this opportunity, resulting in comments from 78 of our 84 novice participants. We used open coding to categorize participants' responses based on named attributes and keywords. This resulted in 48 unique codes, each qualified as being positive/negative and assigned to one or more of the designs.

Positive comments about program comprehension were common for all design alternatives, but were found more frequently for the vertical designs: The most frequently used and uniformly distributed codes were *understandable* (45 overall mentions) and *simple* (44 overall mentions). The code *intuitive* was more frequently used for the vertical designs (30 mentions) than the horizontal ones (12 mentions). Some participants also mentioned that a vertical design was *compact* (6 mentions) while none said this about any horizontal design.

Participants' preference for the vertical design was further explained by their consistent comments on key issues. One participant commented, "Because I work vertically in Excel a lot, the vertical format was more intuitive.", which was echoed in many other comments, such as "I like the vertical due to the normal flow of scrolling down in the computer screen" and "information is normally presented top to bottom on a screen". The users' prior experience with vertical flow in other settings may have led to a preference for it in this setting. Another clear reason that users preferred the vertical flow was that "it is easier to associate left and right arms with left and right columns, rather than rows", mentioned by many users, who commented that "the natural left and right orientation of the vertical programming was a big positive" and "the left-right distinction is more clear rather than having to remember that top is left and bottom is right."

There were however codes that highlighted strengths of the horizontal design as well: The code *shows time flow* was used 8 times for horizontal designs compared to only 3 times for the vertical designs. Further, 21 participants complimented the *icon labels* used in the horizontal designs. In contrast, only 8 participants mentioned the *text labels* of the vertical designs positively, while 2 participants criticized them. These results are consistent with the previous observation that while the majority of participants preferred the vertical design, there is a minority with a substantial preference for the horizontal design.

Those users that preferred the horizontal design also explained their preference consistently. One participant commented "The icons in the horizontal example made it easy to understand in one glance which actions are performed"; this was echoed by others, such as "the horizontal flow's icons were very useful and intuitive" and "[I] like the horizontal's use of pictures over text". While the icons were the most positively mentioned property, a few users also commented that "it was easier to mentally determine the timing of the program's actions", "the horizontal design clearly gave me a timeline", and "I liked that it follows the design of most timelines, which feels most natural." However, perhaps the most relevant comment that several users made was the suggestion that we "bring the visual aids of the actions [(the icons]) from the horizontal version to the vertical version."

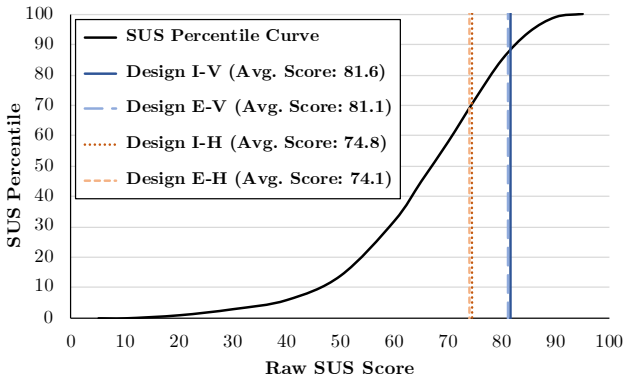


Fig. 11: SUS scores computed for expert participants' responses, plotted on the SUS percentile curve. Experts gave consistently higher ratings for all designs and had the same preferences as novices.

The synchronization model did not seem to influence the prevalence of any specific code. In addition, only a small number of comments mentioned the used synchronization model: Some participants said that they liked the used way of synchronization (5 mentions for the explicit and 4 for the implicit variant). This suggests that the synchronization model had only a minor influence on the opinion of participants.

RQ3 Summary: The data shows that the way blocks are labeled (text or icon-based) influenced participants' opinion the most, followed by the program flow orientation.

5.4 RQ4: Do experts have different preferences than novices?

Figure 11 shows the SUS scores we calculated based on the responses of expert participants. The experts not only performed better in all comprehension tasks, but also gave higher usability ratings for all design alternatives. They did however show the same preferences between the design alternatives as novices: The difference between the designs using implicit and explicit synchronization models is negligible (<1 point), while there is a substantial 7-point difference between the ones using vertical and horizontal program flow. A one-way ANOVA finds that there is a significant difference in our results ($F(3, 206) = 4.80$, $p = .0030$). Unlike for novices, a post-hoc Tukey HSD however only finds a statistically significant difference between Design E-V and the Designs I-H and E-H (both $p < .05$).

To identify factors that influenced expert users we used the same approach as for RQ3 on the results for expert users: In addition to collecting importance ratings for the pre-selected differences, we also coded 91 written comments. However, we did not see any noticeable difference in either of the results compared to novices.

RQ4 Summary: Overall, expert participants gave higher average usability ratings and showed the same preference for designs with vertical program flow as novices.

6 Interactive Prototype Study

In Sections 4 and 5 we presented a survey that compared four design candidates. We found that users could comprehend all designs equally well. We further found that the users preferred vertical program flow but had no preference between synchronization models. As discussed in Section 3, our designs with explicit synchronization are more expressive than those with implicit synchronization. We therefore decided to continue the development of our design candidate that presents program flow vertically and uses explicit synchronization.

While our survey measured how well participants could comprehend given mock-up programs, we were not able to verify if participants could write these programs themselves. To validate the usability of the design candidate that we implemented, we conducted an interactive follow-up experiment. This study was smaller than our survey but provided important insight into how novices could use our design to solve realistic robotics tasks.

6.1 Implementation

Our prototype implementation is based on a modified version of CoBlox, an existing environment for block-based robot programming [7]. CoBlox itself builds on the established block-based programming editor Blockly [14]. We modified CoBlox to add support for blocks with two inputs and outputs, allowing us to implement a two-column environment and support programs similar to our previously presented mock-ups. While the resulting programming environment is a fully functional, we were not able to support testing and debugging programs on a real robot due to time constraints.

6.2 Study Protocol

Our study assessed the following research question:

RQ5: Can users solve realistic coordinated robot programming tasks using our prototype?

We recruited 11 participants at a large office site of a multinational engineering company. Participants came from a diverse set of professional backgrounds, similar to those of our comparative survey. We classified participants as novices or experts based on the same classification questions as for our survey.

We showed all participants a short introduction video to demonstrate the prototype to them. The video explained the available blocks and showed how to write a simple program that picks up and then drops a cube. The participants had the opportunity to pause the video and ask the study supervisor clarification questions at any point.

After they finished watching the video, we asked participants to solve two tasks. For each task we gave users a text that described the intended behaviour of the robot. We also gave them a picture that showed a number of pre-defined locations that they could use in programs. The first task was to create a two-armed pick-and-place program, similar to Task 2 of our survey. The second task was to create a program to hand an item over from one arm to the other, similar to Task 3 of our survey. All participants solved the tasks in the same order.

We wanted to provide participants with a way to test their programs. Since we could not support testing on a real robot, we instead offered participants to manually "simulate" their current program for them: When participants asked to test their programs, the study supervisor would imitate the robot's movements with their arms. However, only 2 of the participants used this opportunity before submitting their final solutions.

We measured the time that participants needed to complete each task. We further graded their final results for correctness. After they completed both tasks, we gave participants an SUS usability questionnaire, similar to the one we used in our survey. Since this experiment involved interaction with our system, we included all 10 official SUS questions [28].

6.3 Study Results

All of our 11 participants finished the interactive study. We classified 7 of them as experts and 4 as novices. Only two of the experts had previously used any robot programming tools while the others had some other previous programming experience.

A total of 10 participants (3 novices) correctly solved Task 1 while 7 participants (2 novices) correctly solved Task 2. Participants took 4 to 5 minutes on average for each task, with the longest taking 11 minutes for one task. Participants gave our system an average usability score of 76.8 which is corresponds to the 70th SUS percentile. There was no substantial difference between the ratings of novices and experts.

While most programs that participants wrote were correct, not all were as efficient as possible: For Task 1, many participants added additional `Barrier` blocks around two-armed `Follow` blocks. This suggests that participants did not fully trust the synchronous behavior of these blocks. Similarly, 2 participants solved Task 2 without running any commands in parallel, as shown in Figure 12. We did not instruct participants to focus on efficiency. Therefore, they might have chosen this solution since it was easier to comprehend for them than a more efficient alternative.

Task 2 had a lower success rate than Task 1. Most of the incorrect solutions for Task 2 share a similar issue, as shown in Figure 13: Instead of coordinating both robot arms, participants tried to write two sequential, independent programs to solve this task. None of the participants with this type of issue asked the study supervisor to simulate their programs. Therefore they might have not been aware that their solution is incorrect. We assume that all participants

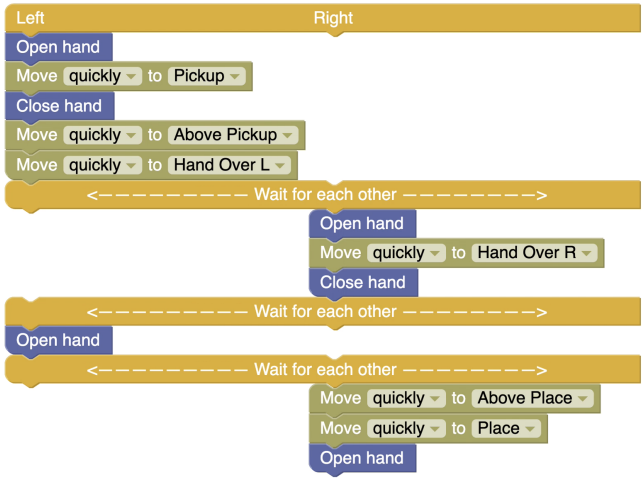


Fig. 12: A correct but inefficient solution for the handover task of our interactive study. None of the commands are executed in parallel, which adds unnecessary wait times but does not affect the overall result.

would have tested their solution at least once if they had a real robot available to them. Therefore this type of issue might be more related to our study protocol than to the evaluated prototype.

In summary, **most participants were able to solve two realistic coordinated robot programming tasks using our prototype.**

7 Discussion

We believe this work can influence both end-user robot programming and block-based language design. We also believe that our design and evaluation methodology has shown potential to be applied successfully to other software engineering domains beyond robotics. In this Section we discuss these implications of our work, and also assess some of the limitations of our empirical results.

7.1 Complexity in End-User Robot Programming

Previous research has shown that end-users can learn to successfully program industrial robots. Our own observations seem to confirm these previous results. However, previous work was limited to simple tasks that used only a single robot arm. This paper shows that our novel designs can be used by novice end-users to program non-trivial use cases that involve the coordination of multiple robot arms.

Although we asked novices to solve more complex programming tasks than previous work, we did not give them substantially more guidance or learning time. This did not seem to negatively impact their learning or performance. We further found that end-users do not need to have mastered single-armed programming before they can learn how to write more complex programs. We therefore suspect that previous work has not yet pushed novices to the limit of their learning abilities.

We believe that future research should attempt to push novice programming even further. End users might be able to tackle even more complex tasks and larger programs.

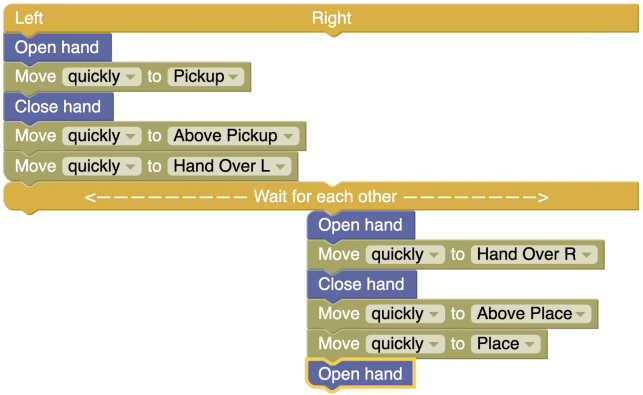


Fig. 13: An incorrect solution for the handover task of our interactive study. Instead of a single coordinated program, this participant has written two independent programs separated by a `Barrier` block.

As our survey showed, powerful end-user tools can also be attractive to experts. Areas we consider worth further investigation are event and error handling, representing more complex movement logic, and coordination beyond two robot arms.

7.2 Richer Visualizations in Block-based Languages

The design candidate that we have implemented coordinates robot arms explicitly through barriers. Barriers are an established concept in traditional parallel programming to synchronize threads. However, using barriers correctly in parallel programs is usually considered to be a difficult task that is too advanced to teach to novices.

Our experiments have shown that beginners were able to understand barriers quickly and with minimal external guidance. They were able to comprehend programs with barriers just as fast as those that used implicit line-by-line synchronization. This is surprising since implicit synchronization is less expressive and therefore seems like it should be easier to understand. Barriers also did not seem to match the way that participants of our preliminary studies intuitively read coordinated programs.

One explanation for our findings could be the way we chose to visualize barriers: We showed the commands for each arm side-by-side and visualized the barrier as a single block that connects both arms. Unlike in traditional text-based programs, novices therefore do not need to manually find and align concurrent actions. We believe this to be a powerful visual feature of our language that has not been used as effectively in previous work.

We see barriers as just one example of how an effective visualization can make advanced language features easier to comprehend. Block-based languages could benefit from using stronger visualizations that go beyond encapsulating commands from text-based languages in blocks. This particularly applies to languages that do not aim to be similar to a specific text-based language.

7.3 Design and Evaluation Methodology

Earlier in this work we have described our overall design process: We started by conducting small-scale preliminary

studies and developing early design candidates based on novice feedback. We then analyzed the potential strengths and drawbacks of these designs by using the CDN as an established cognitive framework. Based on the results of this analysis, we then selected design candidates for a further empirical evaluation.

Recent work has already demonstrated that the CDN can be used to compare different block-based languages [30]. The CDN can reveal how design decisions can potentially impact users and explain their trade-offs. It however cannot tell if users prefer one of the resulting overall designs, or can comprehend some designs easier than others. We therefore believe that combining an empirical and analytical evaluation leads to stronger and more useful results than using only one approach.

7.4 Limitations

Participant population: Our comparative survey was open to the general public. However, due to the channels through which it was advertised, we expect the majority of our participants to have a North American or European background. This may have introduced cultural biases, such as a preferred reading direction for the mostly North American or European participants. It is further likely that a disproportionate number of participants are employees of the same multi-national engineering company. The participants we recruited for our interactive prototype study were also employees of this company in a single location. This might have caused our participants to be a more homogeneous group than the general population.

User Training: Both our survey and our prototype evaluation used training methods for our participants that were practical but not necessarily realistic. For our survey we had to limit the amount of time spent on introducing users to our designs. This might have negatively affected their comprehension. It might for example explain why users had trouble solving Task 1 of our survey, even though it was designed as a warm-up task. Despite the intended beginner-friendliness of our designs, we would expect real industrial end-users to be trained more thoroughly. In particular, we would expect any real-world training to involve interactive testing and experimentation due to fewer time constraints.

Task Choice: We have motivated our choice of tasks for each experiment. We are however aware that they only cover some of the potential usage scenarios for our designs. We have verified that programs for other common tasks can be implemented in our designs. Further studies in the field are required to validate whether our findings can be generalized to these tasks.

Attrition Bias: The survey results presented in Section 5 include the partial results of 81 participants who completed some but not all tasks. Participants may have left the survey because they were frustrated and found the survey tasks too challenging. This might have led to the artificially inflated comprehension scores or usability ratings for later tasks. To test for this bias, we have examined the partial results of participants who did not complete the survey. We did not find a notable difference in their scores. We found no correlation between the assigned designs and drop-out rates. We therefore assume any bias caused by participants who dropped out as negligible.

8 Conclusion

As robots become more powerful and accessible, the tools used to program them must keep pace. For programming environments that target end-users, it is even more important to find a balance between usability and expressiveness than for expert tools. In this work we have presented four design alternatives that enriched a block-based programming language with visual elements to enable novice users to create coordinated programs. We have analyzed which of their properties influence their usability and expressiveness and evaluated the designs empirically on 110 professional end-users to find the trade-off that works best for this audience. Our results showed that novice end-users were able to comprehend an expressive synchronization model that coordinates robot arms through explicit barriers. The data further showed that end-users prefer to read programs that flow vertically over horizontal ones. An interactive prototype study with 11 professional participants has confirmed that our top design led to a viable prototype that allowed participants to solve realistic programming tasks.

Based on our findings, we believe that there is more potential for designs that expand the expressiveness of end-user robot programming. As our work has demonstrated, end-users are able to tackle the traditionally challenging problem of writing coordinated programs when given appropriately designed programming language features.

Acknowledgment

The authors would like to thank ABB Inc. for generously providing funding and hardware.

References

[1] J. E. Colgate, J. Edward, M. A. Peshkin, and W. Wannasuphoprasit, "Cobots: Robots for collaboration with human operators," in *Proceedings of the 1996 ASME International Mechanical Engineering Congress and Exposition*. ASME, 1996, pp. 433–439.

[2] P. J. Hinds, T. L. Roberts, and H. Jones, "Whose job is it anyway? a study of human-robot interaction in a collaborative task," *Human-Computer Interaction*, vol. 19, no. 1, pp. 151–181, 2004.

[3] J. Krüger, T. K. Lien, and A. Verl, "Cooperation of human and machines in assembly lines," *CIRP annals*, vol. 58, no. 2, pp. 628–646, 2009.

[4] S. Kock, T. Vittor, B. Matthias, H. Jerregard, M. Källman, I. Lundberg, R. Mellander, and M. Hedelind, "Robot concept for scalable, flexible assembly automation: A technology study on a harmless dual-armed robot," in *Proceedings of the 2011 International Symposium on Assembly and Manufacturing (ISAM)*. IEEE, 2011, pp. 1–5.

[5] Z. Pan, J. Polden, N. Larkin, S. Van Duin, and J. Norrish, "Recent progress on programming methods for industrial robots," in *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*. VDE, 2010, pp. 1–8.

[6] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the eleventh annual international conference on international computing education research*. ACM, 2015, pp. 91–99.

[7] 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 *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, pp. 366:1–366:12.

[8] G. Biggs and B. MacDonald, "A survey of robot programming systems," in *Proceedings of the Australasian conference on robotics and automation*, 2003, pp. 1–3.

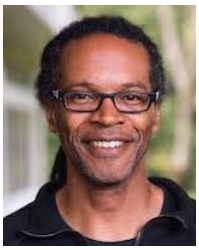
[9] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[10] D. Benedetelli, *Lego Mindstorms EV3 Laboratory: Build, Program, and Experiment with Five Wicked Cool Robots*. No Starch Press, 2013.
[11] R. Bischoff, A. Kazi, and M. Seyfarth, "The MORPHA style guide for icon-based programming," in *Proceedings of the 11th International Workshop on Robot and Human Interactive Communication*. IEEE, 2002, pp. 482–487.
[12] Universal Robots, "PolyScope manual," 2013.
[13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.
[14] N. Fraser *et al.*, "Blockly: A visual programming editor," *URL: https://code.google.com/p/blockly*, 2013.
[15] S. Grover and S. Basu, "Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic," in *Proceedings of the 2017 SIGCSE technical symposium on computer science education*. ACM, 2017, pp. 267–272.
[16] 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.
[17] D. Weintrop, "Block-based programming in computer science education," *Communications of the ACM*, vol. 62, no. 8, pp. 22–25, 2019.
[18] B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages & Computing*, vol. 1, no. 1, pp. 97–123, 1990.
[19] M. M. Burnett, "Visual programming," *Wiley Encyclopedia of Electrical and Electronics Engineering*, 2001.
[20] J. H. Maloney and R. B. Smith, "Directness and liveness in the morphic user interface construction environment," in *ACM Symposium on User Interface Software and Technology*, vol. 95, 1995, pp. 21–28.
[21] 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.
[22] A. Feng, E. Tilevich, and W.-c. Feng, "Block-based programming abstractions for explicit parallel computing," in *Proceedings of the 2015 Blocks and Beyond Workshop*. IEEE, 2015, pp. 71–75.
[23] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3D tool for introductory programming concepts," *Journal of Computing Sciences in Colleges*, vol. 15, no. 5, pp. 107–116, 2000.
[24] S. L. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.
[25] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton, "Visual programming and debugging for parallel computing," *Parallel & Distributed Technology: Systems & Applications*, vol. 3, no. 1, pp. 75–83, 1995.
[26] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.
[27] S. Wiedenbeck, "The use of icons and labels in an end user application program: an empirical study of learning and retention," *Behaviour & Information Technology*, vol. 18, no. 2, pp. 68–82, 1999.
[28] J. Brooke *et al.*, "SUS - a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.
[29] J. R. Lewis and J. Sauro, "The factor structure of the system usability scale," in *International conference on human centered design*. Springer, 2009, pp. 94–103.
[30] R. Holwerda and F. Hermans, "A usability analysis of blocks-based programming editors using cognitive dimensions," in *Proceedings of the 2018 Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 217–225.

Nico Ritschel is a Ph.D. student in the Department of Computer Science at the University of British Columbia, and part of the Software Practices Lab. He is co-supervised by Reid Holmes and Ronald Garcia. He is interested in both the theoretical foundations of programming but also the impact that programming languages and tools have on people. Currently, his research focuses on designing programming tools that target novices and allow them to create complex programs with minimal learning effort. He received an M.Sc. and B.Sc. in Computer Science from the Technical University of Darmstadt.

Vladimir Kovalenko is a Senior Researcher at JetBrains Research in Amsterdam, The Netherlands. As of 2020, he is finishing a Ph.D. in Software Engineering at Delft University of Technology, where he worked full-time for 3 years. His academic work is supervised by Alberto Bacchelli and Arie van Deursen. His interests are centered around the idea of making software development process more efficient with smarter team collaboration tools, and designing data-driven features for next-generation software development team collaboration tools. Vladimir received a M.Sc. in Software Engineering from Academic University of the Russian Academy of Sciences, and a B.Sc. in Astrophysics from Saint Petersburg Polytechnic University.

Reid Holmes is an Associate Professor in the Department of Computer Science at the University of British Columbia. His research interests include understanding how software engineers build and maintain complex systems; this understanding is generally translated into tools and techniques that can be validated in practice. He was previously an Assistant Professor at the University of Waterloo and a postdoctoral fellow at the University of Washington. He earned his Ph.D. at the University of Calgary, and his M.Sc. and B.Sc. at the University of British Columbia.



Ronald Garcia is an Associate Professor of Computer Science at the University of British Columbia. His research investigates how fundamental concepts in the theory, implementation, and practice of programming languages can improve the software development process. His research has focused on static and dynamic type-based reasoning, metaprogramming, and generic programming. Prior to his appointment at UBC, he was a Computing Innovation Fellow at Carnegie Mellon University and Postdoctoral Fellow at Rice University. He received a Ph.D. in Computer Science from Indiana University, and an M.Sc. and B.Sc. in Electrical Engineering from the University of Notre Dame.

David C. Shepherd is an Associate Professor in the Department of Computer Science at Virginia Commonwealth University. His current work focuses on enabling end-user programming for industrial machines and increasing diversity in computer science. He earned his Ph.D. and M.S. in Computer Science at the University of Delaware, and his B.S. in Computer Science at Virginia Commonwealth University. David has since worked as a postdoctoral fellow in the Department of Computer Science at the University of British Columbia, and as a Senior Principal Scientist at ABB Corporate Research.