Enriching Block-based End-user Programming with Visual Features

by

Nico Ritschel

B.Sc., Technische Universität Darmstadt, 2015
M.Sc., Technische Universität Darmstadt, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
November 2023
© Nico Ritschel 2023
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Enriching Block-based End-user Programming with Visual Features**

submitted by **Nico Ritschel** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Computer Science**.

**Examinining Committee:**

Reid Holmes, Professor, Computer Science, UBC  
*Supervisor*

Ronald Garcia, Associate Professor, Computer Science, UBC  
*Co-supervisor*

Ivan Beschastnikh, Associate Professor, Computer Science, UBC  
*Supervisory Committee Member*

Michael J. Feeley, Associate Professor, Computer Science, UBC  
*University Examiner*

Ning Nan, Associate Professor, Sauder School of Business, UBC  
*University Examiner*

Amy J. Ko, Professor, The Information School, University of Washington  
*External Examiner*
Abstract

Today, most programmers are not professional software developers, but end-users with limited training and experience in programming. End-user-friendly programming languages and tools aim to support this type of user, and many use visual programming aids to do so. Block-based programming is a popular visual programming style that has been effectively used in computer science education and is the foundation for many modern end-user programming tools. Because of the popularity of block-based programming, language designers can use a rich set of existing technologies that save them the effort of creating visual programming designs from scratch. However, many language designers ignore that block-based programming was created with learners in mind, who have different needs than end-users. Especially when programs grow in size and complexity, blocks offer little support to help end-users understand and edit programs effectively.

In our work, we augment block-based programming with visual features that extend the range of programs that end-users can comprehend and write. In particular, we create languages and environments for the domain of robotics programming that allow end-users to write larger and more expressive programs. We focus on three scenarios that represent challenges that end-users face in this domain: coordinating multiple robots that work in tandem, writing large programs that span several workstations in different locations, and reacting to external signals such as machines or user interactions. For each environment, we first discuss the limitations of existing work in the areas of block-based and end-user programming. We present and discuss the design of our visual extensions with the goal to maintain end-user-friendliness. Finally, we evaluate our work through empirical studies, both formative to inform our designs and summative to demonstrate their benefits. Our designs, and the empirical and analytical process that we applied to create them, both contribute to a stronger understanding of how to build end-user-centric tools. We further believe that although our work focuses on the domain of robotics, these contributions transfer to other areas of end-user programming as well.
Lay Summary

Learning how to program is hard, but many people must write programs as part of their daily work. Block-based programming is a popular tool to help these people because it is easy to use and can be adapted to many types of programs. However, it has only been used for small and simple programs before, and real programs are often large and complex. In this dissertation, we explore how useful block-based tools are for programming industrial robots, and what factors are holding them back. We then present ideas on how block-based tools can support large and complex programs without making them harder to learn. We present studies to compare different designs and find out if they reach the goals we have set out to achieve. We believe that our ideas and the way we study them can be useful for other areas where non-experts have to write programs as well.
Preface

Chapters 3, 4 and 5 of this dissertation have been adapted from work that was previously published or is currently under review for publication. All remaining chapters are my own original, unpublished work.

- Chapter 3 (excluding Section 3.4.2) is based on work that has been published [Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ronald Garcia, and David C. Shepherd. Comparing Block-based Programming Models for Two-armed Robots. *Transactions on Software Engineering (TSE)*, 2020]. I was the lead author of this work, have created the presented design prototypes, and have designed and conducted the presented study under supervision of Reid Holmes and Ronald Garcia. Vladimir Kovalenko implemented and evaluated the prototype that is presented in Section 3.4.1. David C. Shepherd was an external supervisor and has coordinated the collaboration on this project.

The work presented in Section 3.4.2 has been accepted for publication [Felipe Fronchetti, Nico Ritschel, Logan Schorr, Chandler Barfield, Gabriella Chang, Rodrigo Spinola, Reid Holmes, and David C. Shepherd. Side-by-Side: Block-based Programming for Two-Armed Robots: A Comparative Study. *International Conference of Software Engineering (ICSE)*, 2024]. Felipe Fronchetti, Logan Schorr, Chandler Barfield and Gabriella Chang developed and evaluated the presented prototype based on my initial work that is presented throughout the chapter under supervision of Rodrigo Spinola and David C. Shepherd. I have supported them in the design and evaluation of their work, but am not the lead author.

- Chapter 4 is based on work that has been published [Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C. Shepherd. Can Guided Decomposition Help End-users Write Larger Block-based Programs? a mobile robot experiment. *Proceedings of the ACM on Programming Languages* 6, *OOPSLA2*, Article 133, 2022]. I was the lead author of this work, have designed and developed the presented
programming environment, and designed and conducted the presented study under supervision of Reid Holmes and Ronald Garcia. Felipe Fronchetti supported the work by providing feedback and ideas on early design prototypes and the study design. David C. Shepherd was an external supervisor and has coordinated the collaboration on this project.

• Chapter 5 is based on unpublished work that is currently being revised for future submission [Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C. Shepherd. Blocks? Graphs? Why Not Both? Designing and Evaluating a Hybrid Programming Environment for End-users]. I was the lead author of this work, designed and developed both presented programming environments, and designed and conducted the presented study under supervision of Reid Holmes and Ronald Garcia. Felipe Fronchetti supported the work by investigating related work on graph-based languages and tools, and providing feedback and ideas on design prototypes and the study design. David C. Shepherd was an external supervisor and has coordinated the collaboration on this project.

The study presented in Chapter 3 has been reviewed by UBC’s Behavioural Research Ethics Board under BREB Number H18-02963. The studies presented in Chapters 4 and 5 have been reviewed by UBC’s Behavioural Research Ethics Board under BREB Number H21-01434.
Table of Contents

Abstract ................................................................. iii
Lay Summary .............................................................. iv
Preface ................................................................. v
Table of Contents ..................................................... vii
List of Tables ............................................................ x
List of Figures ........................................................... xi
Acknowledgments ....................................................... xvi
Dedication ............................................................... xviii

1 Introduction .......................................................... 1
  1.1 Connecting Block-based Programming and Visual Programming ............................ 5
  1.2 Contributions of This Work .................................................. 9

2 Background ............................................................ 13
  2.1 Block-based Programming .............................................. 13
    2.1.1 Early Block-Based Programming Languages ............................ 14
    2.1.2 Scratch as the Prototype for Modern Block-based Programming ......................... 20
    2.1.3 Modern Block-based Programming Beyond Scratch .......................... 23
    2.1.4 A Language-based Comparison of Block-based Programming Languages ................. 26
  2.2 Visual Programming .................................................... 28
    2.2.1 Classification and Strategies of Visual Programming Languages ......................... 28
    2.2.2 Block-based Programming and Visual Programming .............................. 32
3 Block-based Synchronization: Coordination of Two Robot Arms

3.1 Coordination and Parallelism in End-user Languages

3.1.1 Coordination in Block-based Languages

3.1.2 Coordination in Visual Languages

3.1.3 RobotStudio Online YuMi

3.2 Design Approach: Side-by-side Synchronization

3.2.1 Showing Coordinated Programs Side-By-Side

3.2.2 Synchronous Coordinated Movements

3.2.3 Timing and Synchronization of Asynchronous Movements

3.2.4 Vertical vs. Horizontal Program Flow

3.3 Comparative Survey

3.3.1 Research Questions

3.3.2 Survey Design

3.3.3 Survey Results

3.4 Further Studies on Coordination

3.4.1 Front-end Prototype Study

3.4.2 Large-scale Physical Robot Study

3.4.3 Limitations

3.5 Discussion and Summary

3.5.1 Using Visual Program Alignment To Represent Complex Program Semantics

3.5.2 Visual Support for Programming Beyond Two-Armed Robots

4 Block-based Decomposition: Mobile Robots Operating Multiple Workstations

4.1 Decomposition in End-user Languages

4.2 Design Approach: Guided Decomposition

4.2.1 Why Do End-Users Not Use Functions?

4.2.2 Domain-Specific Program Decomposition

4.2.3 Aligning Physical and Programmatic Scope

4.2.4 A Visual Programming Environment That Supports Decomposition

4.3 Experimental Evaluation

4.3.1 Research Questions
List of Tables

2.1 Desirable block-based language properties and comparison of selected languages based on these properties. Languages are sorted in order of their first release. ✓/X= Language does/does not have property. (√)= Language partially has property; from top to bottom: Alice only supports drag and drop editing for statements, not expressions; Snap! and AIA mix natural language and traditional programming syntax; Snap! and AIA provide feedback, but error messages often refer to compiled code rather than blocks. 27

2.2 All dimensions introduced by the 13 Cognitive Dimensions of Notation framework [1]. Descriptions are derived from Green et al.’s original definitions [2] and adapted for the purpose of evaluating (visual) programming languages. 38

3.1 Box-plots of the scores of novices for each task’s comprehension questions. Participant numbers (n) are reported in Table 3.7. 58

3.2 Perceived importance of individual design aspects for novice participants, overall and grouped by their preferred design: 1 is least important, 5 is most important. 61
## List of Figures

1.1 Range of potential programming modalities for end-users (from left to right): Programming by example, visual programming, block-based programming and text-based programming. This dissertation proposes to study the benefits of merging visual and block-based programming. ........................................... 3

1.2 A large block-based program for a smart home application [3]. The visual highlighting of the program structure does little to make the program more readable or easier to edit. ............... 4

1.3 Examples of programs with custom blocks in the VPL SimuLink and the BBPL Scratch. ................................................................. 8

1.4 Work presented in this dissertation. Aligned with the research questions stated in Section 1.2, we target three programming challenges and address them with prototypes that combine blocks and visual programming. Then, we analyze the prototypes analytically and empirically. ................................. 10

2.1 Example programs written in the two oldest block-based languages, illustrating the early language features described in this section. ................................................................. 14

2.2 The Alice 2.0 programming environment, which allows users to create animated 3D scenes using a mix of block-based and menu-based programming. ........................................................................ 18

2.3 Definition of the expression “x > 0” in Alice via four levels of nested drop-down menus. ................................................................. 19

2.4 Modern version of the Scratch environment that runs in a web browser and allows users to create sprite-based 2D animations. 21

2.5 Synchronization as proposed by the authors of Scratch [3], recreated in a modern version of the environment. Users need to manually define and update variables that can be used in “wait until” blocks. ................................................................. 22
2.6 Example of type coercion in Snap!: The function supplied to “map” requires two arguments instead of one, but instead of raising an error, Snap! duplicates the first argument to fill both holes. ........................................... 24
2.7 Example program in AIA with an error that prevents compilation. A colour expression must be connected to the program to fix the error, but this is not obvious to users as other expression blocks look identical. ................................. 25
2.8 Examples of visual languages in different categories. ....... 30
2.9 ABB RobotStudio: A modern development system for professional robot programmers [5] that features a complex simulator and development environment. ................................. 34
2.10 Polyscope: a simplified robot programming environment with visual features for intermediate users [6] and a less complex simulator interface. ................................. 36
2.11 CoBlox: a block-based robot programming environment for end-users [7] integrated into ABB RobotStudio. It only supports rudimentary commands, but is easier to use than any of the other shown systems. ................................. 37

3.1 Parallel programming in Alice using a “do together” block nested with “do in order” blocks. ................................. 43
3.2 Parallelism in Lego Mindstorms EV3. The sole purpose of the six blocks highlighted in yellow is to synchronize the top and bottom row. ................................. 44
3.3 RobotStudio Online YuMi programming environment for two-armed robots. The tool is primarily text-based, but has a set of buttons on the right side of the interface. ................................. 46
3.4 Example of a block-based program mimicking the physical layout of the targeted robot, with each arm’s program placed on the same side as the physical arm facing the user. ....... 47
3.5 Example programs using vertical flow. In both programs, one robot arm hands over an item to the other. .................. 50
3.6 Example programs using horizontal flow. In both programs, one robot arm hands over an item to the other, as for the vertical programs in Figure 3.5 .................. 52
3.7 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 3.5a, 3.5b, 3.6a and 3.6b. ........... 55
3.8 SUS scores computed for novice participants’ responses, plotted on the SUS percentile curve. Participants rated the vertical design candidates significantly better.  

3.9 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.  

3.10 Fully functional prototype environment Duplo based on the design prototype E-V.  

3.11 Participants’ completion times in ascending order. Each bar represents one participant, and its color indicates the assigned environment (blue for Duplo, yellow for ROY). A black circle marks participants who ran out of time before completing all sub-tasks.  

3.12 Design space based on synchronization model and program flow direction. Each of the designs shown in Figures 3.5a, 3.5b, 3.6a and 3.6b corresponds to one cell of this table.  

4.1 Proposed programming system showing two tasks and side-by-side canvases. The left canvas shows the main program and the right one shows the body of the currently selected task(s).  

4.2 Structures found in end-user programs. Traditional programming languages treat all of them as equal alternatives, but we speculate that a hierarchical structure benefits end-users the most.  

4.3 Grammar(s) of the programming language used in the proposed system. A program (left) can only contain calls to tasks at the top-level, and that task definitions (right) take place in a separate programming canvas with its own syntax. As Section 4.2.4 describes, there are some differences between the language’s semantics shown here and its block-based presentation shown in Figure 4.1. Also note that locations are defined via a visual location picker that is not represented in this grammar.  

4.4 The traditional block-based development environment used by the TR Cohort of the programming study. It offers the same command blocks and simulator, but uses a single canvas and provides free-form function blocks that are defined and used in this canvas.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Categorization of participants in the TR cohort and their programs for the three tasks our study.</td>
<td>87</td>
</tr>
<tr>
<td>4.6</td>
<td>Categorization of participants in the GD cohort and their programs for the three tasks our study.</td>
<td>88</td>
</tr>
<tr>
<td>4.7</td>
<td>User study results. For box plots: center lines show the medians, box limits indicate the 25th and 75th percentiles. Whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. Outliers are represented by dots, only outliers &lt; 20 minutes are shown.</td>
<td>92</td>
</tr>
<tr>
<td>4.8</td>
<td>Examples of programs written by participants for Task 2. (a) shows a subset of an unstructured program (57 blocks in total); (b) shows the top layer of a hierarchically structured program (61 blocks in total).</td>
<td>96</td>
</tr>
<tr>
<td>4.9</td>
<td>Examples of programs written by participants for Task 3. (a) shows a subset of an unstructured program (58 blocks in total); (b) shows the top layer of a hierarchically structured program (61 blocks in total).</td>
<td>97</td>
</tr>
<tr>
<td>4.10</td>
<td>Example of a program where movement and stationary instructions are entangled. It is not possible to entangle movement and stationary instructions in this way with our approach.</td>
<td>99</td>
</tr>
<tr>
<td>5.1</td>
<td>A nested expression in the block-based language Scratch: The entire expression is displayed in a single line and the nesting structure is difficult to identify.</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of flow in imperative block-based code and block-based trigger expressions.</td>
<td>105</td>
</tr>
<tr>
<td>5.3</td>
<td>The same program as in Fig. 5.2b represented as a data-flow graph. The edges make the data-flow explicit and programmers can manually group and arrange nodes.</td>
<td>107</td>
</tr>
<tr>
<td>5.4</td>
<td>Hybrid programming environment: the left canvas shows high-level program flow as blocks, the right canvas shows selected tasks as blocks and triggers as data-flow graphs.</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td>Procedure of the randomized experiment to evaluate the two prototype environments</td>
<td>111</td>
</tr>
<tr>
<td>5.6</td>
<td>A more complex trigger with nested operators, used to test participants’ program comprehension.</td>
<td>115</td>
</tr>
<tr>
<td>5.7</td>
<td>Completion probability over time for the two programming tasks. Shaded areas are 95% confidence intervals.</td>
<td>117</td>
</tr>
</tbody>
</table>
5.8 Distribution of responses for the Likert items of the survey, split by participant group (Blocks: N=58, Graph: N=52). For the bottom row, bars are aligned around neutral responses to indicate diverging preferences. ........................................ 119

6.1 Early design draft for a 3-armed extension of the vertical design with explicit synchronization that we present in Chapter 3. It uses 3 columns to represent the individual robot arms. ................................................................. 132

6.2 Early design draft for an environment that features both two-armed coordination as presented in Chapter 3 and guided decomposition as presented in Chapter 4. The left canvas shows the high-level program flow, which is composed of a series of two-armed programs that are defined in the right canvas. ................................................................. 133
Acknowledgments

First and foremost, I would like to thank both of my supervisors, Reid Holmes and Ron Garcia, for their guidance, encouragement and advice over the past six years. Both my own research and the world around it have taken turns that nobody could have foreseen when I started my degree. I am grateful that I had not just one, but two compassionate and dependable mentors as sources of support during this time.

I would also like to thank Ivan Beschastnikh for his advice and feedback on my work and the trajectory of my research. It has helped immensely to steer and improve this dissertation.

I have had many other supporters and collaborators throughout my degree that I would also like to recognize. The most influential one was without doubt David Shepherd, who has been a great mentor throughout all my work, almost to the degree of being a third informal supervisor. In addition, I would like to thank David’s student Felipe, who was a fantastic collaborator on many projects, and Alexander Summers, who has encouraged me to take on teaching, and who has put immense effort into mentoring me during my term as instructor.

Furthermore, I would like to thank Sebastian Erdweg and Mira Mezini, my previous supervisors during my time at TU Darmstadt. Although my area of research has shifted substantially since I worked with them, it was their encouragement that led me to take on my PhD journey, and I would certainly not be in the same place without them.

This research has been generously supported by ABB Inc. and the Natural Sciences and Engineering Research Council of Canada (NSERC). Through my collaborators, it has further been supported by the Natural Science Foundation (NSF).

I have also been supported by my parents and my family, who have been patiently waiting for me to finish my degree and have been there for me, both emotionally and financially, the entire time. I apologize for moving so far away, but enjoy every day we are able to see or talk to each other.

Finally, I would like to thank the many members of the Software Practices Lab and the many friends I made within the lab and beyond, of which
I only have space to address a few directly. James, talking to you has been an everlasting source of fun and your (terrible) jokes have lifted me up many times. Adam, I am glad for all the runs (both for coffee and in the literal sense) we had over the years. Giovanni, you were a great friend and playing board games with you was often a highlight of my week. Arthur, thanks for being my grad buddy and helping me out whenever you could. Felipe, your advice has helped me many times as I was stuck at research or in the middle of departmental bureaucracy. Paulette, thank you for always standing up to the department and fighting for our grad student concerns. Braxton, Felix, Joey and Lucas, even though we only shared a short period in the lab together, I still enjoyed all of it! Neil, Hayley, Siddhesh, Anna and Michael, you were amazing friends, and I will never forget all the amazing times and adventures we had together!
Dedication

TO ALEX
Meeting you is the best thing that happened on this journey.
Chapter 1

Introduction

Millions of people program at work or to solve problems in their daily lives, but only a small fraction are professional software developers. In the United States, approximately 1.7 million employees were categorized as software developers in 2020 [8]. Most of these employees have received a comprehensive education in computer science and hold at least a Bachelor’s degree in a related field [8]. However, previous research estimated that between 55 and 90 million employees in the U.S. had to program as part of their daily tasks in 2012 [9, 10], a number that has likely grown over the past decade. These programmers are not trained professionals, but end-user programmers who have received almost no programming-related education.

End-users must program because there are not enough professionals available to write all the code for them. Learning how to program is hard, and even beyond earning a formal degree, it takes novices several years of education and experience before they reach the skill level that is expected of a programming expert [11]. Further, the majority of everyday tasks requires domain knowledge that is at least as hard to acquire as the necessary programming skills. One way to solve this shortage is to make programming easier, and make end-users learn only the skills that are absolutely necessary to solve the task at hand [12]. This type of end-user with limited training is sometimes called vernacular developers, in contrast to the professional developers who traditionally write software [13].

Programming environments and languages must support end-user programmers to allow them to succeed. A substantial body of previous research has investigated how to teach end-user programmers to use tools that were designed for experts [14]. However, training alone can rarely cover the lack of knowledge and experience that end-users are confronted with when they use such systems. End-users are more likely to succeed when the tools they use are specifically designed with their needs in mind [14, 15].

Many beginner-friendly tools and languages use Visual Programming, a form of programming where some or all parts of a program are represented and edited graphically. By using this form of editing, the designers of these systems aim to support users beyond what pure text is capable
of [16]. There also exist a number of purely text-based historical [17] and recent [18] languages that aim to be beginner-friendly, but even those are often supported by environments that provide visual aids. The vast majority of visual modalities fall into the spectrum that Figure 1.1 illustrates. On the extremes, techniques like Programming by Example [19] or Natural Language Programming [20] try to overcome the need for traditional text-based programming entirely. Programming by example has users record and execute the intended program behaviour, and natural language programming interprets instructions in natural language without requiring users to learn any syntax beyond the language they already know. On the other end of the spectrum, work on visual aids for text-based programming like syntax highlighting or error highlighting aims to make programs and their behaviour easier to comprehend without changing the underlying programming language at all. Subtle visual aids have been shown to be useful for professional developers, but the support they provide does not appear to meet the needs of novice programmers [21].

In the middle of the visual modality spectrum lie Visual Programming Languages (VPLs) and Block-based Programming Languages (BBPLs). Both aim not to replace traditional programming, but to make it easier by combining it with visual elements. VPLs typically make aspects of a program more explicit, or changing them more direct [22, 23]. In the example in Figure 1.1, this aspect is the flow of data, but other VPLs might choose higher-level aspects, such as the scope of data [24], or aspects that are domain-specific like the layout of a website [25]. BBPLs are often seen as a specific type of VPLs, but in their pure form they are different. Instead of aspects of a program, they visualize the syntax of the underlying programming language and make programs easier to edit structurally by dragging and dropping syntactic elements as if they were puzzle pieces. [4]. This makes them quite versatile: virtually any text-based programming language can be converted into a BBPL [26].

Block-based programming has reached wide-spread adoption in recent years, particularly in programming education where its correspondence to text is often seen as a benefit. For example, the TIOBE Index of popular programming languages lists the block-based programming environment Scratch [4] as the 20th-most popular programming language overall in 2022 [27]. BBPLs are said to provide first-time programmers with more visual feedback than text-based languages, and to encourage experimentation in an environment that minimizes frustration [4, 28]. However, block-based systems like Scratch are not intended for use beyond learning – they primarily serve as a stepping stone in turning beginners into expert program-
Figure 1.1: Range of potential programming modalities for end-users (from left to right): Programming by example, visual programming, block-based programming and text-based programming. This dissertation proposes to study the benefits of merging visual and block-based programming.

Block-based programming is also used in several recent tools that were designed with end-user programmers in mind [3, 7, 32, 33]. Because end-user languages are often limited in scope, tool developers often prefer to use an established, standardized visual design that comes with tried and tested frameworks and tools. Mature block-based frameworks like Blockly [34], which have been evaluated both in terms of usability and the required implementation effort for the overall system, are a good match for these needs [33, 35, 36]. However, most end-user BBPLs are just variations of existing educational languages and build on the same principles without considering the differences between traditional learners and end-users. Therefore, they have limitations that make them hard to use in practice, for example when programs are larger or more complex than those used in teaching settings [37].

In theory, BBPLs can support almost any advanced programming language feature [26], but their usability benefits diminish if these features are incorporated naively. Instead, the visual representation of low-level program syntax can distract users in some cases and reduce the readability of programs and the usability of the overall system [38]. Consider the block-based program in Figure 1.2. It looks complex and requires substantial mental effort to read, although the underlying program is less than 20 lines long and mostly comprised of simple variable assignments and arithmetic. This pro-
Figure 1.2: A large block-based program for a smart home application [3]. The visual highlighting of the program structure does little to make the program more readable or easier to edit.

program illustrates the problem that block-based system designers face when they replicate text-based language designs and features, even if they do not address the needs of end-users.

In this dissertation, we investigate whether block-based systems can be designed to be end-user-centric, meaning that they address the needs of end-users rather than learners. In particular, we explore how BBPLs can be adapted to enable end-users to write a larger variety of more expressive programs without making them harder to learn or use.

Thesis statement: By incorporating end-user-centric design features from visual programming, block-based programming languages can enable end-users to write programs that solve larger and more complex tasks while retaining their usability, learnability and ease of implementation.

We limit the scope in which we investigate our thesis to one concrete area of industrial end-user programming: robotics. By focusing on one area in which end-users commonly perform programming work, we can tailor languages to their specific needs and use abstractions and terminology that they are familiar with. However, we consider robotics programming to be a
placeholder for other end-user programming domains, such as home automation [33, 39] or app development [35]. We believe that most observations we make and design considerations we derive apply to these other domains as well.

We selected robotics as our domain of inquiry for several reasons:

1. The target audience of robotics programming languages, and the settings in which they are used, are comparatively well-studied and well-defined. Previous work has outlined the settings in which industrial end-users must program robots [7, 40], which forms of training they can receive in practice [7, 12], and what their expectations are when programming and interacting with robots [41]. This foundation of research allows us to reduce the amount of preliminary work necessary to address the needs of this type of end-user.

2. Although modern robotics programming languages provide abstractions for low-level cyber-physical parameters, several aspects of robot programs challenge end-user programmers: Many non-trivial robot programs require some form of coordination or interaction between multiple robots or robots and their environment. In addition, even conceptually simple tasks like picking up and carrying an item can require a large number of programming steps and become complex when a robot must interact with its environment. Programming interfaces can address these challenges by providing abstractions and visual aids that offer additional support.

3. Robot programming is a domain with substantial related work in both language modalities that we aim to connect with each other. End-user robotics programming has been explored previously, by visual programming work [42, 43] and block-based programming work [36, 44]. We can use these design examples from both domains to guide our work. At the same time, most previous work has substantial usability or scalability limitations [45], leaving room for significant improvement.

1.1 Connecting Block-based Programming and Visual Programming

Due to the growing interest in block-based programming, recent work has explored their use outside of education. For example, personal end-users
can now use blocks to automate their homes [33, 39], and professional end-users can use them to develop augmented reality apps [35] and program industrial robots [7]. In all of these use cases, blocks are intended to be the only language that an end-user has to learn to solve the task at hand. This means that the ability to transition from blocks to text is not a major reason that might motivate system designers to use BBPLs. However, the similarity between block-based programming and text has other advantages that makes BBPLs attractive for end-user systems:

1. Block-based programming is an established technique. It allows both users and language designers to rely on best practices and transfer knowledge from one block-based tool to another. One can even speculate that the continued, widespread use of blocks in child education could expose potential end-users to blocks and make it easier for them to use block-based programming tools productively as adults.

2. Many visual languages use data structures like graphs to represent programs internally [46]. This requires different compilation or interpretation strategies than traditional programs that are represented as Abstract Syntax Trees (ASTs). The structure of block-based programs on the other hand matches that of the underlying AST. This means that tool creators can not only save the effort of creating a parser for their programming language, but can use established language frameworks and tools for AST manipulation and compilation. This makes block-based tools not only easy to use, but also easy to modify and extend.

3. The familiar visual design of block-based systems makes them also directly accessible to people with existing programming experience in text-based languages. These users can understand the underlying text-based code without having to learn an entirely new visual syntax. This means that professionals can easily read and edit block-based programs, and support or collaborate with end-users if necessary.

Evaluations conducted by the authors of end-user block-based systems consistently find them to be easier to learn and use than professional tools [33, 35, 36, 47]. However, these findings are always based on small tasks that might resemble realistic use cases but are still simplified so that they can be solved by a single user in a single session. There is little concrete evidence that end-users could solve larger or more complex tasks as well. Further, studies on educational block-based systems that are used outside traditional
learning environments have shown that even when BBPLs support encapsulation or procedural abstraction to enable the writing of larger programs, users rarely make use of these features [37, 38]. Similarly, some block-based tools support advanced programming language features like parallelism [4], object-oriented programming [48], or higher-order functions [49], but users often ignore or misunderstand them [38]. It therefore appears that existing block-based tools are better suited for writing short, procedural code snippets than large or complex programs.

The designers of VPLs have developed methods to represent encapsulation and abstraction that they have demonstrated to be more understandable to end-users than the traditional approaches used by text-based languages or BBPLs [2, 50]. We call these methods end-user-centric, as they typically target a specific audience of end-users with existing domain knowledge, unlike educational languages that aim to teach general purpose programming to users with a wide range of backgrounds.

For example, Figure 1.3 compares how the VPL SimuLink represents the application of a user-defined function in contrast to the BBPL Scratch. Although both languages use similar mechanisms for procedural abstraction, their designs differ in important details: SimuLink represents programs as flow graphs, and represents procedures as nodes in this graph. In SimuLink, edges into and out of a procedure represent inputs and outputs; the body is fully abstracted and can only be edited in a separate canvas that is independent from the main program. As a result, user-defined procedures in SimuLink are explicitly encapsulated and data flow into and out of each procedure is visualized. On the other hand, Scratch presents both the procedure definition and its usage on the same canvas, and even allows users to freely drag variables between them. Similar to text-based, dynamically typed programming languages, scoping restrictions are not enforced until an error occurs at run-time. This increases the likelihood that users introduce bugs and misunderstand where data can flow in the program.

Procedural abstraction is just one example of a language feature that visual languages represent through end-user-centric visualization. Visual languages have also demonstrated that they can represent features like parallelism [51] and higher-order functions [52] in ways that make them more explicit and easier for end-users to understand. As a result, many visual languages successfully scale to large and complex programs, and they have proven to be invaluable tools for real industrial end-users [53, 54].

In this dissertation, we propose to investigate whether we can design block-based programming languages that support larger, more complex programs in a similar way to VPLs. We believe that many of the end-user-
(a) Scratch program that uses a custom block to abstract a sequence of if-else conditionals. The definition is shown next to the main program.

(b) SimuLink program that uses a custom block to abstract a mathematical computation. Only the inputs and outputs of the block are shown, unless the user manually inspects the block.

Figure 1.3: Examples of programs with custom blocks in the VPL SimuLink and the BBPL Scratch.
centric methods and strategies that visual languages use to achieve this goal can be applied to blocks as well. We further hypothesize that by adding visual, end-user-centric features to blocks, language designers can provide benefits to end-users as they use BBPLs for their programming work.

1.2 Contributions of This Work

To support the thesis statement we presented above, we investigate the following research questions:

1. What usability limitations do end-users encounter when using existing block-based programming languages?

2. Can we extend BBPLs with end-user-centric features to overcome these limitations in the area of robotics without losing their end-user-friendliness?

3. What implications does the addition of end-user-centric features to BBPLs for robot programming have for their existing usability and implementation benefits?

In the rest of this section, we provide additional context and details regarding these questions and the work we propose.

Figure 1.4 outlines the structure of this dissertation. It refers to the research questions above. We have selected two types of robots that serve as use cases for our end-user programming tools: two-armed robots that are stationary, and mobile robots that can move within a defined perimeter and have one attached robot arm. Both of these robot types are commonly used in practice, but there exists little work that allows end-users to use them to their full potential. In particular, we have identified three programming challenges that end-users face and that existing tools do not address:

1. **synchronizing two robot arms to solve coordinated tasks.** This scenario is difficult for end-users because it directly forces them to write programs that contain parallelism and coordination. When they program two robot arms, users must actively determine when arms wait for each other or perform synchronized actions. Existing block-based tools have attempted to make this accessible. However, all existing pure block-based system designs suffer severe limitations regarding visibility or abstraction. To tackle this scenario, we present a design that uses a visual approach...
To address RQ1: We identify three challenges that end-users face when using block-based tools:

- coordinating parallel programs
- decomposing large programs
- embedding nested expressions into imperative programs

To address RQ2: We design prototypes of block-based programming tools that incorporate visual programming features to target each programming challenge in the context of robotics:

To address RQ3: We evaluate the prototypes during and after the design process, using:

- analytical discussions of features and trade-offs, using the "13 Cognitive Dimensions of Notations" framework
- empirical methods, such as controlled studies and experiments with end-user participants

Figure 1.4: Work presented in this dissertation. Aligned with the research questions stated in Section 1.2, we target three programming challenges and address them with prototypes that combine blocks and visual programming. Then, we analyze the prototypes analytically and empirically.
to improve visibility by displaying two programs side-by-side, and then highlights interaction points between the two robot arms to make them easier to understand for end-users.

2. **programming mobile robots to solve tasks across multiple workstations.** This scenario poses two separate challenges to end-users: First, when a robot’s task spans multiple workstations, the robot must operate in multiple different spatial contexts. For example, objects or tools that are necessary to solve a task may or may not be present at different workstations at given points in time, and end-users must understand this dependency between the program they create and the robot’s current surroundings. Second, multi-station robot programs often grow in size as the number of workstations and steps to complete increases. Without structuring programs and using appropriate abstraction, end-users can experience difficulties with reading or editing such large programs. In traditional programming languages, users solve both of these challenges using procedural abstraction. However, as we illustrated in Section 1.1, BBPLs do not sufficiently convey the purpose and correct usage of this form of abstraction to end-users. Instead of teaching users how to use abstraction through external means such as manuals or tutorials, we present a design for this scenario that guides users in decomposing their programs as they work on a task. Our design splits programs into low-level tasks that take place in a single location and are connected by a separate, high-level structure. These tasks are placed in separate editing canvases, which are arranged in a way that makes it easy to navigate and keep track of the different program components.

3. **programming mobile robots to interact with their environment.** This usage scenario extends the previous one by assuming that robots do not just perform tasks as a single predetermined sequence, but react to inputs from their environment. By enabling this interaction, we both extend the range of tasks that a robot can accomplish and potentially allow the same robot to work on multiple parallel tasks. This use case aligns closely with how workers conceptualize robots in industrial settings: they consider robots to be their assistants and would like to assign them tasks that help in their daily routine [41]. To allow a robot to work on multiple tasks, potentially even those that serve multiple different end-user programmers, requires a clear program structure and model of interaction with the environ-
ment. For this reason, we propose a design for an editing system that focuses on triggers to control and schedule the robot’s interactions with its surroundings. In particular, we propose a hybrid environment that connects block-based programming and visual dataflow programming for defining dependencies and conditions that trigger the robot to perform specific work.

We first answer research question RQ1 by analyzing each of the proposed usage scenarios and the limitations that end-users encounter when using existing tools. We then answer RQ2 and RQ3 by designing and evaluating language and environment prototypes to overcome these limitations. For our evaluation of each design, we combine our findings from RQ1 with an empirical assessment through user studies and experiments. Chapter 3 describes our work on two-armed synchronization and coordination, Chapter 4 the work on programming mobile robots across multiple workstations, and Chapter 5 the work on making mobile robots interact with their environment. In Chapter 6, we provide a discussion of our work and a reflection on which elements can be applied to other domains within and beyond robotics.
Chapter 2

Background

In this Chapter, we give an overview of the history of block-based programming and the accomplishments and limitations of several seminal block-based tools. We also present and categorize visual programming languages based on their properties and features, some of which we set out to transfer to block-based programming. Finally, we introduce the framework of Cognitive Dimensions of Notation (CDN) that provides us with terminology and categorizations that we use in later Chapters.

2.1 Block-based Programming

Block-based programming has gained substantial popularity over the past decade. It has been successfully used in Computer Science education in high-schools [28] and universities [55], and in personal [33] and industrial [7] end-user programming. This is remarkable, considering that early block-based languages did not receive much mainstream attention. The Scratch [4] environment, released in 2010, was the first block-based programming system that found widespread popularity, and it has inspired many other languages that apply block-based programming principles to new application domains.

In this Section, we present a selection of block-based programming languages that we consider to be influential or historically relevant. Centering around Scratch as the environment that influenced modern block-based programming the most, we describe languages before Scratch in Section 2.1.1, the principles introduced by Scratch itself in Section 2.1.2, and more recent work building on top of Scratch in Section 2.1.3. In addition to providing a historical background, we outline the contribution of each new block-based tool at a language or environmental level. In addition to individual contributions of previous tools, we highlight one exemplary challenge that almost all block-based tool designers face and that illustrates the range of choices they can make: how to visually represent types using blocks. Finally, we summarize the features and properties of each discussed tool in Section 2.1.4, although we revisit some of them in later chapters because they relate to our own contributions.
Figure 2.1: Example programs written in the two oldest block-based languages, illustrating the early language features described in this section.

2.1.1 Early Block-Based Programming Languages

The oldest language that we consider to be related to modern block-based programming environments is BridgeTalk, which was released in 1987 [56]. BridgeTalk was designed to allow users to convert informal problem descriptions into functional program code. Remarkably, BridgeTalk’s authors already distinguish between learners and end-users as two potential user groups that they aim to support: in their vision, aspiring programmers could learn BridgeTalk as an introductory language to prepare them for professional languages like Pascal, and end-user programmers could write all their programs in BridgeTalk without the need to “graduate” from it. However, though BridgeTalk underwent multiple design iterations and user studies, it was never fully implemented.

At a first glance, the language, shown in Figure 2.1a, looks remarkably similar to modern block-based languages: BridgeTalk uses puzzle-like shapes for blocks that visualize the structure of a program explicitly. As its authors
explain, they chose this design to address common issues they perceived when beginners write their first programs. For example, they found that beginners often struggle with understanding the flow between lines of nested code like the loop shown in Figure 2.1a. In addition, beginners are often unaware of the full syntax of a language, and don’t know whether they can translate a single step in their informal plan into a single instruction or must decompose it into multiple steps. BridgeTalk supports its users by providing a visual template for which syntactic elements are necessary and how programs can be composed from them.

Another noticeable design aspect of BridgeTalk is that it is much more verbose than text-based programming languages. Each syntactical element is labeled and explained in plain language that can be understood without external documentation. This is a stark contrast to other popular programming languages from the time BridgeTalk was developed, which relied on syntax that was efficient to write for experts, but hard to learn and read without memorizing abbreviated commands and syntax. BridgeTalk’s authors chose this design with two goals in mind. First, they wanted to give novice users confidence in their code, and avoid syntax that users could misinterpret if they did not have the necessary background knowledge. Second, they aimed to implicitly teach users the common programming terminology of the time as they were repeatedly using the same blocks labeled with it.

Finally, another design goal of BridgeTalk is to provide support for high-level abstractions to guide the thinking of programmers. From a modern perspective, where high-level language features are commonly found even in professional languages, this goal might not be immediately clear from examining programs like the one in Figure 2.1a. This program not only illustrates the detailed step-by-step computation of a while-loop, but also illustrates how the absence of non-integer variables requires programmers to resort to workarounds like using “magic numbers” to represent special states. However, this program demonstrates BridgeTalk’s high-level language goals in the contexts of its time, for example by showing how the language distinguishes between a “Counter Plan” and the “Running Total Plan”. Both of these blocks represent simple arithmetic addition, but BridgeTalk uses distinct blocks for them to illustrate that one is used as a loop iteration counter while the other is used for mathematical computation.

All of BridgeTalk’s design elements we describe above made their way into modern block-based languages. However, BridgeTalk has two substantial limitations compared to later languages that only become apparent when writing code instead of just reading it. First, BridgeTalk programs are not assembled by dragging and attaching blocks, but rather by iterating through
each block’s unfilled syntactic connectors and selecting a possible new block to fill each gap. Second, all blocks have a completely fixed shape, and blocks can only be nested in ways supported by their shape. This means that in Figure 2.1a, one cannot place a “Compute Plan” block in the “Get New Value” slot of the loop block because it does not have the correct height for the slot. It also means that loops cannot be nested inside other loops, and the same applies for conditionals. BridgeTalk’s authors discuss potential workarounds for this issue but also argue that larger programs or deeper nesting of blocks might confuse novices. They claim that discouraging novices from writing code that is too complex for them to understand could be a feature instead of a limitation. However, as newer block-based languages have shown, BridgeTalk’s limitations were severe enough to outweigh this benefit.

After several years without follow-up work related to BridgeTalk being released, Sten Minör re-introduced the idea of composing programs through visual blocks in 1992 [57]. This work is primarily an analysis and critical discussion of structural editing tools of its time and does not contribute any new language designs or evaluations. However, it presents brief program snippets that resemble BridgeTalk’s design ideas, and introduces the idea of using a mouse pointer to drag and drop blocks onto the programming canvas. Remarkably, Minör does not explicitly reference BridgeTalk, and none of the later work on blocks we discuss in this Chapter cites his work either. It appears that although Minör’s work fills a gap in the development of block-based tools, it has had little influence on other tools, and that parallel developments like the introduction of drag-and-drop were coincidental.

Following Minör’s conceptual work, LogoBlocks [58] was the next fully-fledged language that re-introduced the idea of programming with blocks. Notably, like Minör, the authors of LogoBlocks do not cite BridgeTalk as a source of inspiration, and although they share the same design ideas, there appears to be no direct connection between the two languages. Instead, the authors of LogoBlocks were inspired by the text-based programming language Logo [17] and the learning theory of constructionism. The theory of constructionism focuses primarily on educating children and aims to support learning through experimentation rather than following a fixed curriculum of instructions and learning materials [59]. Logo and LogoBlocks support this idea by providing only a small number of syntactic elements that are easy to memorize and supported by the visual shapes of blocks. Through this syntax, children can quickly progress from guided learning to self-driven experimentation and create sharable artifacts as part of their learning process [60].

BridgeTalk and LogoBlocks are designed for different audiences and pur-
poses: BridgeTalk’s targets adults to introduce them to established programming concepts and patterns, while LogoBlocks targets children and is only loosely connected to traditional programming languages. These differences might also explain the different domains that the two languages target. BridgeTalk aims to support arbitrary computation, and LogoBlocks focuses on the more tangible use case of programming Lego robots. Despite their different origins, the two languages share a remarkable number of design elements, including the ideas of using puzzle pieces to represent syntax and providing high-level abstractions to their users. However, LogoBlocks overcomes BridgeTalk’s two main limitations: it allows users to assemble programs via drag-and-drop, and it allows blocks to scale in size as needed as long as they are connected to an existing block through the right connector. LogoBlocks is also the first language that uses different shapes to visually distinguish between integer and boolean variables, as shown in Figure 2.1b. The round outer connector of the boolean expression “sensora < 10” is easily distinguishable from the rectangular connector of the integers “sensora” and “10” it contains. LogoBlocks also uses colours to highlight semantic differences between blocks, for example the constant “10” and the mutable sensor variable “sensora”.

Another notable distinction between BridgeTalk and LogoBlocks is the use of textual labels. As the two programs in Figure 2.1 illustrate, LogoBlocks can represent a larger program in substantially less screen space than BridgeTalk. To achieve this goal, LogoBlocks uses abbreviations for some of Logo’s commands, as shown in Figure 2.1b where the blocks labeled “a” or “b” are short forms of Logo’s “motor a, on” statement to turn on a motor. This is a stark contrast to BridgeTalk’s verbose block descriptions, and can make LogoBlocks much harder to read without background knowledge.

Alice 2.0 is a block-based educational storytelling environment that was released in 2006 and received wide-spread attention in the computer science education community [61]. It extends the text-based scripting language Alice, which was released in 1998, and provides a block-based programming environment for it [62]. Alice primarily targets high-school students and allows them to program the behaviour of characters and objects in animated 3D scenes. This re-framing of programming as a way to tell stories was considered to be one of Alice’s primary contributions at the time [63] and has inspired a number of later, not necessarily block-based, educational environments and computer science curricula [64, 65].

Focusing on the block-based aspects of Alice 2.0, one of the most notable design choices is the closeness of blocks to text and the goal of allowing users to transition to traditional programming. In fact, Alice is the first instance
of a system that was not initially designed for block-based programming, but converted from text at a later point in time. Alice 2.0 was specifically designed to support the transition between blocks and text, and both use the same underlying language. This programming language is object-oriented and similar to Java, but replaces some keywords with natural language and removes others entirely, like parentheses and semicolons. Before execution, programs are converted into actual Java code, which modern versions of Alice allow users to view as well [66]. Users can therefore follow the entire pipeline from blocks to simplified text-based code to traditional Java within a single programming environment.

Despite its merits, Alice also serves as a first example of some of the inherent limitations of block-based programming. Alice is statically typed and aims to support object-oriented programming similar to Java. However, to be friendlier to learners, Alice aims to reduce the number of different types that users must consider. As Figure 2.2 illustrates, most commands in Alice are method calls on objects in the current scene, using parameters with a set of primitive types like strings, numbers, colours and directions. Each of these blocks uses a different shape to indicate what its type is and where it can be inserted into the surrounding program. However, Alice’s type

Figure 2.2: The Alice 2.0 programming environment, which allows users to create animated 3D scenes using a mix of block-based and menu-based programming.
Figure 2.3: Definition of the expression “x > 0” in Alice via four levels of nested drop-down menus.
system also supports user-defined classes and subtyping using inheritance. Therefore, an unbounded number of types can appear in an Alice program, so simple shapes cannot indicate which types are compatible.

Alice overcomes the challenge of supporting object-oriented block types by abandoning the block-based puzzle metaphor when composing values. Expressions that appear within these statements cannot be dragged and dropped into place like the statement blocks themselves, but must be selected using drop-down menus. These menus are automatically filtered to show only values with legal types. Although this aspect of Alice’s design ensures that users cannot compose ill-typed programs, it also leads to a less consistent and often tedious user experience than in most other block-based languages. For example, as Figure 2.3 shows, users must navigate through four levels of nested menus to define a simple conditional expression like “x > 0”. Solving the visual challenges of supporting object-oriented programming remains an open problem with which even modern block-based languages struggle, as we describe in Section 2.1.3.

2.1.2 Scratch as the Prototype for Modern Block-based Programming

To this day, the Scratch programming environment remains the most widely used block-based programming system. Figure 2.4 shows a screenshot of a modern iteration of Scratch. The left side of the environment shows the toolbox from which users can drag blocks into the programming canvas in the center. When blocks are attached to a trigger block like the “when space key pressed” block in Figure 2.4, they become an executable program that runs whenever the corresponding event occurs. Most blocks contain commands that manipulate a 2D sprite on a canvas on the right side of the environment, but they can also execute arbitrary code such as playing sounds or showing prompts to the user.

Scratch is not directly based on an existing text-based language, but it features a wide variety of commonly used programming constructs like variables, conditionals, loops and procedure-like custom blocks. Like LogoB-locks, Scratch’s authors clearly name the text-based programming language Logo [17] and the underlying philosophy of constructionism [60] as their primary inspiration. Following the ideas of constructionism, the creators of Scratch name tinkerable as one of their fundamental design principles [4]. By calling Scratch tinkerable, its authors mean that the environment encourages experimentation: since Scratch is designed to prevent any syntactical errors, it allows users to run their code at any point in time. This way,
they can test programs even if they are incomplete. In addition, Scratch ignores unused code blocks that are disconnected from a program. Therefore, users can quickly swap out commands without having to explicitly delete or disable them like in text-based languages.

Another design principle of Scratch is *simplicity*, which is similar to BridgeTalk’s goal of providing high-level abstractions. For example, Scratch offers users a separate block to change the coordinates of a sprite to an absolute value or to a relative one based on its current position, and also provides separate blocks for changing the x-coordinate, the y-coordinate or both coordinates. All of these commands could be handled by a single block and some arithmetic computation, but Scratch aims to provide easier to understand alternatives. Compared to previous languages, Scratch also adds features to simplify its language for beginners: It is the first language to consistently label all of its blocks in natural language instead of typical programming language syntax. It is also the first language to provide automated hygiene for names. When users in Scratch rename variables and function blocks, the
environment automatically finds and renames references to these blocks as well. Most professional code editors provide this feature of hygienic renaming as a refactoring [67], but Scratch makes this way of managing names the default.

Despite their efforts to make Scratch simple to use, Scratch’s authors point out that their language is sub-optimal in some regards. Scratch provides almost 100 primitive blocks, and despite efforts to organize them into categories, the list of options can easily overwhelm users who are new to the environment. Restricting the number of primitive blocks is essential for block-based languages since, unlike text-based languages, they typically expose users to a list of all the syntactic options available to them. Scratch’s authors recognize this issue in their original presentation of the environment and suggest that the number of primitive blocks should be minimized if possible. They also pose block selection as an open problem that could be targeted by future work. However, the number of available blocks in Scratch, as well as the design of its toolbox, have not changed significantly in the years following its initial release.

To demonstrate how hard it can be to find a small, expressive set of primitive blocks, consider a Scratch feature that was prominently presented by its authors from the start and is still present to this day: concurrency. Scratch allows users to write multi-threaded programs, for example by creating multiple triggers that react to the same event. It also provides a block
to broadcast a message to other threads, and a block to pause a thread until a specific message is received or a condition is satisfied. Scratch’s authors claim that these blocks are expressive enough to program a wide range of concurrent programs. For this reason they have intentionally refrained from adding further, higher-level primitives for concurrency. However, in practice it is often non-trivial to synchronize threads in Scratch by using just the available primitives. For example, Figure 2.5 shows how Scratch’s authors propose to synchronize two concurrent threads using a lock. This solution not only adds substantial clutter to the code but also requires users to be familiar with variables. Experienced programmers might also believe that the shown solution could still suffer from a race condition if a thread switch occurs between the “wait” and the “set busy to 1” block. Only with additional knowledge about Scratch’s concurrency semantics, which restricts thread switches to specific points of a program, does this example become fully comprehensible. This feature of Scratch is not only difficult to learn, but can actively mislead learners if they believe that the lock shown in Figure 2.5 is thread-safe in all programming languages.

2.1.3 Modern Block-based Programming Beyond Scratch

Scratch was initially released as a stand-alone desktop application and quickly attracted interest from a wider audience. Google’s Blockly framework, released in 2012, creates a block-based environment similar to Scratch directly in a web browser [34]. Blockly was originally designed to compile block-based programs to JavaScript code, which it can run directly from within the programming environment. However, it also supports other target programming languages and can therefore be used to program almost any system as long as there exists an infrastructure to compile or run the produced code. Blockly is still under active development today and most block-based environments, including modern Scratch, use it as their technological foundation.

A popular alternative to Scratch is Snap!, released in 2013 by UC Berkeley [49]. Snap! replicates most of the features of Scratch, but also adds advanced programming language features to the system. Most notably, Snap! supports collections, higher-order functions, basic object-oriented programming, and continuations. Snap! is primarily used in supervised educational settings such as college classes [68]. Therefore, although it demonstrates that block-based languages can support advanced programming language features, the environment itself does little to explain their purpose to beginners. Learners probably cannot understand features like continuations just
Figure 2.6: Example of type coercion in Snap!: The function supplied to “map” requires two arguments instead of one, but instead of raising an error, Snap! duplicates the first argument to fill both holes.

by tinkering with the blocks, and Snap!’s built-in documentation even states that “continuations are too complicated to explain in a help screen.”

Snap!’s direct translation of advanced programming concepts into blocks exhibits further drawbacks regarding how types are represented. Snap!’s support for lists and higher-order functions exposes users to an unbounded number of types that can occur in a program. Some later work has proposed designs for representing a wider range of types in blocks [69], but Snap! does not use any comparable system. Instead, it uses a single shape for all types and, unlike Alice or Scratch, does not prevent users from assembling ill-typed programs. Snap! attempts to convert types implicitly as programs execute, which avoids frustrating errors in some cases, but can also lead to unexpected semantics. This design decision does not necessarily help beginner programmers, as studies have found that similar behaviour in traditional languages like JavaScript can confuse even professional developers [70]. Take for example the program in Figure 2.6, which uses the higher-order function “map” that expects a function with one parameter as its argument. When the built-in addition operator, which requires two arguments, is inserted instead, Snap! attempts to infer the user’s intention. The output of the program suggests that Snap! fills all arguments of the addition operator with the list argument. Although this allows the program to run and the results might even match the user’s expectations, it is not obvious that this way to use the “map” function is not the intended one. Therefore, beginners might inadvertently learn a wrong lesson when tinkering with Snap!’s features.

Although Snap!’s implicit type conversions are intended to avoid user-facing error messages and the resulting frustration, users can still write Snap! programs that produce error messages. These error messages are not created by Snap!’s user interface, but by the JavaScript engine that
executes the code that Snap! generated. Therefore, they refer to code that is not actually visible to users and can be hard to interpret, especially for beginners that are only familiar with Snap!’s block-based language and not JavaScript. The existence of syntax errors further violates the design principle introduced by Scratch that every program that can be created is executable. Snap! therefore foregoes many of the benefits of Scratch by focusing on features over design consistency.

The final block-based programming environment we discuss in this section is *App Inventor for Android (AIA)*. This popular block-based programming system primarily targets adult learners and allows them to write mobile phone apps [48]. AIA provides both a graphical editor for the app’s user interface and a block-based editor for the app’s functionality. AIA particularly focuses on making programs tinkerable by supporting live programming [71]. By using a special companion app, users can test their apps immediately on their phone and even inject new code while the app is executing.

Like Snap!, AIA supports several programming language features beyond Scratch, such as collections and objects. AIA takes a more measured approach to adding features, such as omitting higher-order functions and providing no way for users to define custom classes. It also does not perform any implicit type conversions, but instead enforces type correctness when blocks are attached to one another. However, AIA does not visually represent block types like Scratch or Alice. This approach can lead to confusing situations as demonstrated by Figure 2.7. The left block has an empty socket that must be filled with an appropriate value. AIA indicates this using a red circle and a tooltip. However, all three blocks on the right look like they could fit in the empty socket, despite only one having the correct type. Users might try to attach the others, which seems visually possible but is not supported by the editor. This underlying type check that explains this
mismatch is completely hidden from the user. Furthermore, a user cannot test a program before all errors in the program are fixed. Even when AIA programs are free of compile-time errors, they can still raise run-time errors, which often refer to the compiled Java code instead of the blocks. This behaviour can cause frustration and limits how easily users can experiment with their programs in practice, despite support for tinkering being among AIA’s primary goals [71].

2.1.4 A Language-based Comparison of Block-based Programming Languages

Guided by the languages we presented and discussed in the previous sections, we have identified a list of shared properties of most modern block-based languages. In Table 2.1 we show a summary of these properties and whether the languages succeed or fail at achieving them. We do not list Blockly in Table 2.1 since it is intended as a framework to develop block-based languages rather than a language itself. Its most significant contribution is to enable block-based programming in a web browser without installing additional software. AIA, Snap! and more recent versions of Scratch are web-based, but we omit this property in Table 2.1 since it is less related to language design than the environment’s implementation. We also omit the simplicity of the language’s syntax. This property is highly relevant when designing block-based languages, but we believe that it is possible to perform a meaningful comparison between the ones we discussed because they serve vastly different purposes. For example, LogoBlocks uses fewer different blocks than any of the more recent languages but is also limited to a very specific domain.

As the direct comparison shows, early block-based languages have consistently improved upon each other. Scratch combined the strengths of its predecessors and was the first environment that combined their desirable properties with the exception of supporting complex data types. Based on the languages we have presented, it appears that supporting complex data types and representing those types as shapes are mutually exclusive properties. Alice is the only language that achieved both, but at the price of using drop-down menus instead of drag and drop to assemble blocks. Remarkably, the more recent environments also forego some desirable properties, like programs being always executable when they can be assembled in the editor. This might suggest that modern languages consider functionality to be as or even more important than beginner-friendliness or consistency with their own design principles.
<table>
<thead>
<tr>
<th>Feature</th>
<th>BridgeTalk</th>
<th>LogoBlocks</th>
<th>Alice</th>
<th>Scratch</th>
<th>Snap!</th>
<th>AIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration of language editor and compiler</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deep nesting of blocks is possible</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Syntax-directed editing using drag and drop</td>
<td>X</td>
<td>✓</td>
<td>(✓)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Support for complex data types</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shapes accurately represent compatible data types</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Automated hygiene for all names</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Programs are readable like natural language</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>(✓)</td>
<td>(✓)</td>
</tr>
<tr>
<td>Programs are always valid and executable</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Immediate feedback between editor and run-time</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>(✓)</td>
<td>(✓)</td>
</tr>
</tbody>
</table>

**Table 2.1:** Desirable block-based language properties and comparison of selected languages based on these properties. Languages are sorted in order of their first release. ✓/X= Language does/does not have property. (✓)= Language partially has property; from top to bottom: Alice only supports drag and drop editing for statements, not expressions; Snap! and AIA mix natural language and traditional programming syntax; Snap! and AIA provide feedback, but error messages often refer to compiled code rather than blocks.
2.2 Visual Programming

Text has always been the dominating modality for creating computer programs. Although programming environments have evolved to provide visual aids and tools, a text editor usually remains the centerpiece of professional software development. However, visual programming environments are used by millions of non-professionals. Spreadsheet software like Microsoft Excel is one of the most popular programming tools for end-users, and its cell-based programming paradigm can hardly be represented as plain text [72]. Creating and representing programs visually is not a recent idea either: the visual graphics editor Sketchpad from 1964 is often considered to be the first example of visual programming [73]. Since then, designers have created thousands of other visual languages, many of which are highly domain-specific or target audiences that are not professional programmers [46].

In the remainder of this section we give an overview of the domain of visual languages. As there exists a much larger body of work for visual languages than for blocks, but only a small fraction is directly relevant for our work, we do not provide a detailed historical overview here. Such overviews can be found in the multitude of surveys that have been conducted on visual programming over the years [46, 74, 75]. Instead, we focus primarily on how the wide range of previous visual languages can be categorized, as well as the strategies found in those languages that can be useful for our work. Section 2.2.1 presents this overview of previous work, and Section 2.2.2 discusses the relationship between block-based programming and visual programming.

2.2.1 Classification and Strategies of Visual Programming Languages

The range of visual languages that are used for programming or have been previously proposed is substantial. Visual programming in practice ranges from spreadsheet-based approaches like Microsoft Excel to flow graph languages like LabView [76], and dozens of other language designs. In 2017, Erwig et al. performed the largest and most recent survey of visual languages to date [46]. They surveyed 484 languages that were published between 1995 and 2014, 340 of which they classified as visual programming languages. Notably, Erwig et al. did not include any block-based languages in their study, as they do not consider them to be visual languages. We discuss this decision further in Section 2.2.2. The authors assigned a combination of 17 tags to each language, which allowed them to also categorize languages that combine multiple design paradigms. By filtering languages based on cer-
tain tag combinations, they were able to identify higher-level patterns like “flowchart” or “spreadsheet”. Their results can be seen as an overview of the domain of visual programming languages.

Erwig et al. initially distinguish VPLs according to syntax:

- **Graph languages** that define objects and how they relate to each other, allowing programs to be interpreted as graphs. They can be distinguished further based on typical graph properties like whether edges are directed or which graph components are labeled.

- **Partition languages** that divide space into regions, suggesting a geometric interpretation. The survey sub-categorizes these languages based on if they are open or closed, the former meaning that they allow space to remain undivided as ”background” (as in a Euler diagram) and the latter, meaning that they require a division of all available space (as in a spreadsheet).

The general categorization resembles that of other authors [74, 77]. The survey found that 91% of the languages can be classified as graph or partition languages, with 36% being only graph-based and 23% being only partition-based. It also found that the number of unclassified languages decreased over time, showing a trend towards standardization. This standardization also means that the binary categorization suffices to describe the majority of the recent visual programming languages.

In addition to syntax, Erwig et al. classify languages semantically as:

- **Graphical or external** based on whether their semantic domain matches the graphical representation used in the language or not. A language that displays image manipulations visually is considered graphical, as are spreadsheets since numbers and formulas are their semantic domain.

- **Static or dynamic**, depending on whether the language models a data structure or describes computation.

Erwig et al. remark that although the division into static and dynamic languages is ultimately the deciding factor of whether a language is a programming language, categorizing a language can be rather subjective. For example, there is often a duality between data structures and implicit computation embedded in them (e.g., state machines). For their survey, the authors decided to classify all languages that could be seen as describing computation as dynamic. The survey found that 72% of visual languages
are purely external and 21% purely graphical, and that 70% of languages are dynamic and 30% are static. These findings indicate that the majority of visual languages are indeed programming languages, and that most are used in an external (i.e., not domain-inherent) visual context.

By combining their tags into higher-level categories, Erwig et al. found that the majority of their studied corpus of data is made up by a small set of visual language types. Figure 2.8 shows an example language from each of these categories:

- 41% were classified as flowchart languages that describe graph-based computation,
- 25% were identified as tabular languages that use closed partitioning (of which 15% are spreadsheet languages that also use formulas),
- and 11% were categorized as Euler diagram languages that use open partitioning and labels (of which 6% also use graph-based elements).
Erwig et al.’s classification can provide an overview of what types of visual paradigms exist in the wild, but not why language authors have chosen these specific designs. Assuming that most visual languages were designed deliberately and carefully, it seems reasonable to assume that their overall goal was to support their users in a way that goes beyond the capabilities of pure text. However, visual languages are not the only means to achieve this goal. Statecharts \cite{78} and the more modern Unified Modelling Language (UML) \cite{79} are just two examples of visualizations for program structures and behaviour that were designed to aid developers during the program design process. More subtle visual aids like syntax or error highlighting are even more common, and found in almost every modern editor. This raises the question of whether there is a fundamental difference between program visualization and visual programming.

Burnett \cite{23} described four strategies that visual languages use to support the programmer. Whether, and to what extent, these strategies are present in visualizations can be used to distinguish visual programming from other uses of visualization:

1. **Concreteness** means presenting data to the user in a way that they are familiar with or that matches their mental image without the need to abstract from it. For example, a visual language could represent a 2-dimensional object concretely by drawing it instead of listing geometric properties.

2. **Directness** aims to enable manipulation of data in a way that resembles the user’s intention as directly as possible. For example, a visual language could allow users to change the position or size of a geometric object via drag and drop instead of editing a numeric “height” property.

3. **Explicitness** tries to explicitly present semantic information to the user without the necessity to infer data or perform mental calculations. For example, a visual language could visualize a relation between two objects by drawing an explicit connection between them instead of making the user infer that connection themselves based on textual descriptions.

4. **Liveness** attempts to give the user immediate and automated feedback about any modification to the program. For example, a visual language could reflect any updates to an object’s properties immediately when the change occurs instead of waiting for the user to manually refresh the program.
Not all visual programming languages use all of the aforementioned strategies to the same extent, but missing a strategy entirely can be an indication that a system does not fall into this category. For example, syntax highlighting does not increase the concreteness or directness of text, and is therefore better categorized as a visual aid and not as a visual language. Program visualizations like UML on the other hand do not use directness or liveness and should therefore not be considered languages themselves.

2.2.2 Block-based Programming and Visual Programming

Surveying work on both block-based programming languages and visual programming languages reveals an interesting discrepancy between the two domains: block-based programming language creators and those who research block-based languages frequently classify them as visual programming languages and therefore see themselves as part of this domain [4, 28, 80]. However, Erwig et al.‘s survey of visual languages does not consider any block-based languages to be part of this domain [46]. It is not difficult to imagine block-based languages as a graph-based language that uses blocks as nodes and their connections as edges. In fact, previous work has used such a representation to statically analyze block-based code [81]. However, at the same time blocks are intentionally similar to traditional text, and can be seen as a pure visual aid to make editing or reading programs easier for beginners. We investigate this disagreement further in the remainder of this section.

To understand why researchers might not consider block-based languages to be visual programming languages, it is useful to apply Burnett’s classification of visual languages [23]. We discuss this classification in detail in Section 2.2.1 but as a reminder: the four strategies that visual languages should apply are concreteness, directness, explicitness and liveness.

At first glance it can be challenging to see where block-based languages apply any of the above strategies. Some languages create blocks that represent graphical sprites (e.g., Scratch, Snap!) or other objects (e.g., Alice, AIA), but all of the properties of the sprites and objects are represented as plain text instead of a more concrete form.\footnote{One notable exception are colours in Scratch, Snap! and AIA that are represented as a coloured square instead of a name or RGB values} Furthermore, none of the languages use techniques, like bi-directional editing [82], that would allow users to manipulate sprites or objects directly and reflect these changes in the block-based program. However, block-based languages do employ some VPL strategies in the context of the syntax of the underlying text language. For example, the puzzle shapes of blocks explicitly visualize the syntax of
the underlying language and give users a more concrete way to see how they can combine primitives. In languages where data types are accurately represented by shapes, one could also argue that blocks make the language’s type system more manifest. Scratch and AIA apply another VPL strategy when they emphasize the idea of tinkerability: tinkerability is similar to liveness in that it allows users to quickly switch between programming and testing and get immediate feedback on their code. Nonetheless, the block-based languages we investigated only partially match the definition of VPLs above. It is therefore understandable why researchers disagree on whether they consider block-based programming to be part of the VPL domain.

When comparing block-based and visual languages, one may ask the question: why do block-based languages not apply more of the VPL strategies? We believe that the answer to this question reveals the most fundamental difference between visual and block-based languages: modern block-based programming, which was heavily influenced by languages like Scratch, was designed with the goal to closely resemble text-based programming. As most language designers intend blocks to be used in educational settings, they want users to have an exit strategy [29] and be able transfer their knowledge when they transition to text-based languages. However, it is important to note that this overall goal does not always align with the ways that block-based programming is used today. For example, languages for personal and industrial use cases may not even have a text-based equivalent that users could learn after blocks [7, 33].

In addition to pedagogical reasons, block-based language developers gain benefits from keeping their languages similar to text: it simplifies their translation to text-based code. In all block-based languages we have investigated in our review, it is straightforward to strip away the visual layer and treat the text contained in the blocks as code. In many cases, blocks can be compiled to textual code line-by-line. However, even if the translation is more elaborate, block-based languages form a tree that approximates an Abstract Syntax Tree (AST). There are substantially more and mature frameworks and techniques to transform and generate code from ASTs than for any visual language formalisms. Language designers can benefit from these tools and create new languages faster and with a focus on design aspects rather than implementation constraints. As a consequence, there are even efforts to make block-based language development itself accessible to non-professionals by providing block-based design tools [83, 84].
2.3 Domain of Inquiry: End-user Robotics Programming

Traditional programming environments for industrial robots are complex and difficult to use. Figure 2.9 shows RobotStudio [5], one such system, that targets professional developers. RobotStudio provides code editing and project management tools, similar to modern Integrated Development Environments (IDEs). It further allows users to test their programs through a complex 3D simulator, and via debugging tools that are integrated with real robot hardware. Other robotics IDEs, such as the ROS Development Studio [85] provide similar features and a have comparable level of complexity.

Robotics IDEs like RobotStudio provide a substantial amount of features to support their users. However, these features were designed with users in mind that have a thorough understanding of both programming in general and in the context of robotics. RobotStudio uses the propriety language RAPID [86], while other tools provide frameworks that build on text-based general purpose programming languages like C++ or Python. They all have in common that they were designed to give programmers low-level
control over the robot they target, sacrificing usability and learnability for this purpose.

Researchers and industrial developers have both tried to make robot programming accessible to end-users as well. They typically propose new programming systems along the spectrum we presented in Figure 1.1, using text-based, block-based, visual or fully automated programming approaches [16, 87]. Automated approaches, which aim to eliminate the need for traditional programming altogether, are particularly popular in robotics. For this reason, Biggs and MacDonald’s survey of programming systems from 2003 sorted all robot programming systems into just two categories manual or automatic [88]. The category of manual systems includes all tools that use any form of visual, block-based or text-based programming. As the survey did not explicitly focus on end-user systems, professional robotics programming languages like RAPID [86] are also covered by this category.

Automatic tools strive to eliminate the need for traditional programming by using alternative modalities, such as lead-through programming [89] or demonstration-based learning [90]. In lead-through programming, users physically guide a robot arm by moving it along a path that is then recorded and replayed when a program is executed. This form of programming is ideal for end-users as it is very direct and requires little training. However, it is limited to simple programs and collaborative robot hardware, which is safe to be used around humans. Demonstration-based learning is conceptually similar, but does not require users to directly interact with the robot arm that executes their program. Instead, users record the intended motion using a teach pendant, a different robot model, or gesture recognition. Although this form of programming is still easy to learn, it is less direct and often less precise [91]. A number of recent systems have been proposed to solve this issue, for example by using mixed and virtual reality [92–94], but their practical benefits remain untested.

Compared to automated programming, manual systems typically support a wider range of programs and do not require specific robot hardware. However, like other end-user and educational systems, they must make usability trade-offs based on their intended audience. For example, the block-based programming tools Open Roberta [44] and BEESM [39] target learners with no experience in robotics. They provide high-level commands and basic simulators to test programs, which makes them similarly easy to learn and use as other block-based tools such as Scratch. However, the functionality and range of robots that they support is limited, and they face similar issues as other block-based tools that we discussed in Section 2.1.3. Other manual systems, such as the RC+ Express [95] and Polyscope [6], propose more
complex programming environments and target intermediate users with programming experience. Figure 2.10 shows Polyscope, which features a simple simulator and a tree-based visual programming language.

A particular system that has influenced our work is CoBlox, a block-based programming environment for collaborative robot arms [7]. CoBlox, shown in Figure 2.11, combines manual and automated robot programming in a single system. Users can use a block-based programming environment to define the overall structure of their programs and then use lead-through programming to define the target position for each arm movement. CoBlox is embedded into RobotStudio and can target both real robots and the same simulator as programs written in RAPID. An evaluation by the authors of CoBlox found that end-users can learn how to use CoBlox faster and solve simple programming tasks more effectively than with other commercially available tools [36]. Although this initial evaluation was performed using a robot simulator, a later evaluation has confirmed that end-users can also program real robots using CoBlox after completing a short training [47].

Despite its promising initial evaluations, CoBlox is inherently limited...
Figure 2.11: CoBlox: a block-based robot programming environment for end-users [7] integrated into ABB RobotStudio. It only supports rudimentary commands, but is easier to use than any of the other shown systems.

in its feature-set. It only supports linear programs that contain a number of movements and target a single robot arm. Features like variables and loops are supported in theory, but CoBlox does not provide any support or instructions for end-users on how to use them. They were also never used in any of the evaluations of CoBlox [36, 47], and all used tasks were only a few blocks long. However, we see the positive results of its usability evaluations as initial evidence that block-based programming is suitable to support end-users as they program industrial robots. This work has motivated us to investigate how block-based robotics can support a richer set of tasks.

2.4 Cognitive Dimensions

Some areas of VPL research are relevant to block-based languages, regardless of how one considers the relation between the two domains. One stream of research that is particularly useful is the framework of 13 Cognitive Dimensions of Notation (CDN) [1, 2], which can be applied to both BBPLs and VPLs. Table 2.2 lists the 13 dimensions introduced by the CDN framework, as well as a brief description of the language aspect that each of them addresses. Since its inception, the CDN has been used to rationalize and
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Gradient</td>
<td>What levels and forms of abstraction does the language support or require?</td>
</tr>
<tr>
<td>Closeness of Mapping</td>
<td>How directly does the language represent the notation of the underlying domain?</td>
</tr>
<tr>
<td>Consistency</td>
<td>How difficult is it to infer parts of the language from knowledge of other parts?</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>How many different elements (e.g., keywords, syntactic rules, visual elements) make up the language or its visual representation?</td>
</tr>
<tr>
<td>Error-proneness</td>
<td>To what degree does the language’s design cause or prevent user errors?</td>
</tr>
<tr>
<td>Hard Mental Operations</td>
<td>How complex is the mental model that a user has to maintain as they read or write programs in the languages?</td>
</tr>
<tr>
<td>Hidden Dependencies</td>
<td>How difficult is it to identify which parts of a program relate to or depend on each other?</td>
</tr>
<tr>
<td>Premature Commitment</td>
<td>Does the language require users to make decisions (e.g., on program structure) before they are aware of all relevant information?</td>
</tr>
<tr>
<td>Progressive Evaluation</td>
<td>Is it possible to evaluate or receive feedback on partially complete code?</td>
</tr>
<tr>
<td>Role-expressiveness</td>
<td>How clear is the structure of programs and the purpose of individual code fragments?</td>
</tr>
<tr>
<td>Secondary Notation</td>
<td>Are there ways to annotate or structure code informally, beyond the language’s syntax?</td>
</tr>
<tr>
<td>Viscosity</td>
<td>How much effort is required to make changes to existing programs?</td>
</tr>
<tr>
<td>Visibility</td>
<td>How difficult is it to read a program, and navigate it if does not fit onto a single screen?</td>
</tr>
</tbody>
</table>

Table 2.2: All dimensions introduced by the 13 Cognitive Dimensions of Notation framework [1]. Descriptions are derived from Green et al.’s original definitions [2] and adapted for the purpose of evaluating (visual) programming languages.
and compare the designs of over 500 visual notations and languages [96]. Recent work has also applied the CDN to block-based languages [97, 98] and found it particularly useful when evaluating differences between block-based languages and trade-offs in their designs.

Some dimensions of the CDN can be directly mapped to block-based language terminology. For example, the dimension of whether a language supports Progressive Evaluation is highly related to the idea of a language’s liveness or tinkerability. Other dimensions describe problems that are particularly simple to solve in block-based languages due to their close relation to text-based program code. For example, all block-based languages since Alice that we investigated address the dimension of enabling Secondary Notation by allowing users to add comments to their code. However, some cognitive dimensions are particularly useful for discussing the benefits and limitations that arise from block-based language design choices.

Take Error-proneness as one example of a cognitive dimension that can be applied to multiple aspects of block-based languages. Error-proneness measures the degree to which a notation leads users to make mistakes. That block-based languages have a less error-prone syntax is one of the most obvious benefits compared to textual languages. Modern environments use additional measures to avoid errors on a visual level. For example, Scratch and most other recent environments automatically align blocks so that disconnected blocks cannot overlap or be closely adjacent to one another, so users do not misinterpret them as being connected [29]. However, different block-based languages prevent or discourage semantic mistakes to significantly degrees. For example, all block-based languages since Scratch that we investigated support the capture-avoiding renaming of variables, but only AIA warns users when they try to use variables outside of scope [97]. As another example, Snap! automatically attempts to convert mismatched block types, similar to runtime type coercions in languages like JavaScript. Just like type coercions, this feature that can simplify some programs but is also known to cause hard-to-predict behaviour that often leads to bugs [70].

We use the CDN primarily because of the terminology it provides for comparing visual programming language features and related notation. In later chapters, we refer to the dimensions listed in Table 2.2 without further introduction, but we highlight them to clarify that we refer to terms of the CDN.
Chapter 3

Block-based Synchronization: Coordination of Two Robot Arms

One of the challenges of industrial robot programming is controlling how a robot interacts with its environment. In traditional systems programming, the preferred strategy to control such interactions is to hide the details of the environment behind abstractions as much as possible. This is possible to some degree for robot programming as well, for example when it comes to controlling physical parameters that control the robot’s motors or sensors. However, a robot’s high-level interactions with its environment, like how it finds target locations, picks up objects, or interacts with machines, are often at the core of robot programming and cannot be hidden from the user.

A specific case in robot-environment interaction is that of two robots working cooperatively. When multiple robots interact with one another, they can solve tasks more effectively than if each was performing work independently [99]. However, programming this interaction is difficult, even for professionals: similar to when processes and threads interact in traditional system design, the individual programs must be carefully ordered to prevent race conditions and unexpected robot behaviour. At the same time, traditional strategies to present and teach parallel programming [100] can be difficult to transfer to robots. The state of a program, in particular, can be challenging to reason about in robot programming, as interactions do not just affect isolated variables, but can change the entire physical state of the robot’s environment.

Block-based programming languages like Scratch or Alice support concurrency similar to professional programming languages, but do little to make it easier to understand for beginners. In Section 2.1.2, we have already illustrated how it is possible to synchronize two Scratch threads through a lock. In this chapter, we revisit this style of synchronization and provide an overview of other approaches, including some from the broader areas
of visual programming and program visualization. We then present our
own contribution: a number of novel designs to visually represent program
coordination in block-based languages, and their analytical and empirical
evaluation. The work is based on the papers “Comparing Block-based Pro-
gramming Models for Two-armed Robots” (published in Transactions of
Software Engineering in 2020) [101] and “Side-by-Side: A Case Study En-
abling Parallel Programming of Robotic Arms by End-Users” (submitted to
the International Conference of Software Engineering 2024) [2]. In the first
paper, we explored four candidate design options that all display two block-
based programs side-by-side and visually connect them at points where they
interact. The four designs differ with respect to their synchronization model
and program flow presentation. We conducted a controlled survey of 273
professional users (110 without previous programming experience) to deter-
mine which one is the easiest to understand and which one our participants
prefer. We then implemented a front-end prototype of the best design and
conducted a smaller interactive experiment with 11 industrial participants
to confirm that it is viable in practice. The second paper went beyond
front-end experiments and evaluated a fully functional environment proto-
type that controls a physical two-armed robot. We found that participants
were substantially more successful at solving realistic assembly tasks using
our block-based system than when they used a text-based, commercially
available tool with similar functionality.

3.1 Coordination and Parallelism in End-user
Languages

Parallel programming is typically considered an advanced topic in computer
science education. It is only taught after students have mastered sequential
programs. This path to understanding parallelism is not suitable for end-
users who have a specific task in mind that they want to solve. They must
be able to learn sequential and coordinated programming at the same time.

Several block-based languages try to support parallel programming and
simplify it to some degree. However, to our knowledge, no previous block-
based work has attempted to visualize parallelism or represent it in a way
that goes beyond the capabilities of text. Therefore, they do little to support

2The author of this dissertation is not the first author of the second paper, but has
contributed to the study design, evaluation and presentation of the results. We focus
primarily on the first paper in this chapter, but summarize the findings of the second
dpaper in Section 3.4.2
end-users as they reason about which parallel program (inter)actions occur simultaneously. We present related block-based work and discuss its limitations in Section 3.1.1. Some prior work attempts to support parallelism in the visual programming language space, but none of those we surveyed are a good fit for enabling end-users to coordinate multiple robots. We present and discuss those attempts in Section 3.1.2. Finally, we briefly introduce RobotStudio Online YuMi, a commercially available, primarily text-based tool that has inspired our work and that we used as a baseline for our evaluation in Section 3.1.3.

3.1.1 Coordination in Block-based Languages

Previous block-based systems use a variety of techniques to make coordinated parallel programming beginner-friendly. In Chapter 2, we introduced the block-based languages Scratch and Alice, both of which support parallelism primarily in the context of animation. As presented there, Scratch supports the parallel execution of multiple triggers. Figure 2.5 illustrates the synchronization of two such triggers: similar to traditional thread synchronization, users can set up a simple lock and make the threads wait for the lock to be available. An expert in systems programming might expect such a lock set-up and find it easy to read and understand. However, for an end-user who might still be struggling with more fundamental programming concepts like mutable variables, this program can be confusing and distract from the actual commands that are executed once the synchronization has occurred. Scratch’s approach is also error-prone, as end-users must either replicate the lock (e.g. after looking up such a set-up in a tutorial or online reference) or devise their own, potentially flawed, implementation.

Alice proposed a different technique for synchronization that requires less low-level programming. Therefore, it might be a better fit for beginners. It features “do together” blocks that enable parallel animations [61]. This type of block, shown in Figure 3.1, allows users to animate multiple objects at once by placing commands inside a “do together” block. To implement more complex sequences of ordered and parallel animations, users can nest “do together” and “do in order” blocks. At the end of each block, Alice automatically ensures that all nested blocks have finished executing.

Both Scratch and Alice provide general-purpose abstractions for parallelism that can be used for different scenarios across multiple domains. However, neither Scratch nor Alice provide users with visual feedback or scaffolding to clarify which commands are executed synchronously or in which order. In contrast to Scratch and Alice, the Parallel Snap! [102] environment for
programming distributed systems focuses on domain-specific high-level abstractions such as MapReduce [103] or the producer-consumer pattern. This focus allows the environment to make these specific abstractions for parallelism very end-user friendly and easy to understand. Parallel Snap! also uses animated visualizations to demonstrate parallelism to users. However, Parallel Snap! does not integrate visual elements directly into the block-based programming language. Users must execute their programs to see the effects of the parallel computation. Parallel Snap! therefore extends block-based programming with novel program visualizations rather than visual programming features.

3.1.2 Coordination in Visual Languages

Visual programming languages aim to simplify parallel programming as well. In particular, visual data-flow languages, which are often purely functional, have explored the parallel computation of data [74]. As data-flow languages like VIVA [104] or CODE [105] demonstrate, a compiler or interpreter for these languages can compute data dependencies automatically and coordinate parallel computation without any additional user input. However, this technique does not generalize to other visual languages, even those that are represented by a flow graph.

One example of a graph-based visual robotics language with more com-
plex coordination is Lego Mindstorms EV3. This language, which represents programs as a control-flow graph, allows each node to have multiple outgoing edges. All nodes connected in this way execute in parallel \cite{106}. Figure 3.2 shows an example program that uses parallelism in EV3. Both the top and the bottom row of the program are executed concurrently.

As Figure 3.2 illustrates, EV3’s approach allows users to arrange their programs 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 execute simultaneously, users must manually synchronize them. They need to manually add a complex barrier construct (highlighted in yellow in Figure 3.2). Besides adding visual clutter, this also requires users to correctly use other non-trivial language features like variables and loops.

EV3 demonstrates that visualizing parallel computation alone is not sufficient to make coordination accessible to end-users. EV3 users might be able to write concurrent programs, but they must manually combine low-level instructions and ensure that they work reliably and as intended. This issue is comparable to the one we previously discussed for the block-based language Scratch, where synchronizing multiple threads is also tedious and requires a lot of low-level effort. However, since EV3 programs target robots, this shortcoming becomes even more critical. For example, although handlers for different user interactions in Scratch might not necessarily interact, mu-
Multiple commands in EV3 that control the same robot must almost certainly be coordinated explicitly.

3.1.3 RobotStudio Online YuMi

RobotStudio Online YuMi (ROY) is a programming interface created by robot manufacturer ABB to control their two-armed robot YuMi. It is designed to demonstrate the range of tasks that two robot arms can solve when they cooperate. Therefore, ROY is specifically designed to be beginner-friendly and accessible to end-users who would be overwhelmed by traditional tools like ABB’s RobotStudio [5].

ROY’s programming approach is primarily text-based, but it uses a number of graphical interface elements to make programming easier for beginners. As Figure 3.3 illustrates, ROY’s main interface contains three panels: two text-based program editors and a panel on the right with graphical buttons. The two editors contain the program code for each robotic arm that is targeted by the environment. Each program is written directly in the RAPID programming language, although boilerplate code, such as function headers and variable definitions, is hidden from the user. To assist users as they write RAPID code, ROY provides a series of buttons on the right to define commonly used robot commands. In addition to opening and closing the grippers attached to each arm, some buttons capture the current position of each robot arm and generate a movement command that uses the captured position as a target. This way of capturing locations by directly manipulating a physical robot is called lead-through programming and is commonly used in end-user robotics systems [7, 90]. As users compile and run their code, the two programs are combined with their respective boilerplate code and automatically deployed onto their respective robot arms.

Although ROY’s graphical programming support appears to enable the text-free creation of programs similar to block-based or visual programming, it has a major limitation. All of the described buttons can only be used to insert new code at the currently selected line number. To edit existing commands, users must manually edit the text, and to change captured locations, they must first switch into a separate “Edit this arm” interface that shows only one arm’s program and displays the normally hidden boilerplate code including variable definitions. Existing variables can be overridden using this interface without having to manually re-write the program, but this feature is easy to miss and switching between different modes complicates the programming significantly.
3.2 Design Approach: Side-by-side Synchronization

Robot programs can be coordinated in many ways, depending on the task that the user is trying to complete. In our work we focus on coordinating two robot arms 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 of 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 of 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 focus on our analytical design considerations and the decisions they motivated. For our design discussions, we use the
Robot programmers often must 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. Furthermore, these designs also reduce the Visibility of each arm’s individual actions.

To improve our prototype, we took inspiration from the overall design
of ROY (shown in Figure 3.3). Displaying the programs for the two robot arms side-by-side not only uses the available screen space efficiently, but also matches the perspective of the user when programming and testing. Figure 3.4a 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 connect both programs, we use a single trigger 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 3.4b.

One important detail of the design shown in Figure 3.4b is that it requires slightly different blocks for each arm. For example, the Open Hand blocks in the two columns must 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. One way to overcome this limitation is to automatically adapt the blocks depending on which arm it is assigned to, mirroring their layout as they get dragged across the center of the canvas. However, in our later prototype implementations we found this behaviour difficult to achieve in practice. We therefore opted for a visually less clean but practically feasible design that does not center blocks around the middle and aligns them on the left for both arms.

3.2.2 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 3.4a 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 the 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, which we refer to as a Follow block, is shown in Figure 3.4b. 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.

### 3.2.3 Timing and Synchronization of Asynchronous Movements

A typical robot program like the one in Figure 3.4b 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 3.4b 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 given by 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 program using this design is shown in Figure 3.5a. 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 3.5a 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 3.4b. Although the first four commands for each arm require
(a) Design (I-V): after each command, the arms implicitly wait for each other. (b) Design (E-V): explicit barrier blocks mark points of synchronization.

Figure 3.5: Example programs using vertical flow. In both programs, one robot arm hands over an item to the other.

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. Efficiency could be improved by swapping the order of commands, but 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 3.4b. 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. The same command might even take a different amount of time across 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 3.5b shows the hand-over program from Figure 3.5a re-written 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 3.5a. It comes however with the drawback that programs cannot always be read as a timeline. Take the program from Figure 3.4b 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 3.5a and 3.5b 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 3.5a and 3.5b show.

The two presented designs that use implicit and explicit 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 3.3.

### 3.2.4 Vertical vs. Horizontal Program Flow

Our preliminary studies indicated that beginners intuitively read side-by-side programs, such as shown in Figures 3.4b 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 time-line 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 3.6a and 3.6b show horizontal versions of the programs we previously presented in Figures 3.5a and 3.5b. 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 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 [107]. 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 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 therefore decided to evaluate the impact of both the synchronization model and the program flow orientation in the comparative survey we present in Section 3.3.

### 3.3 Comparative Survey

We 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.

#### 3.3.1 Research Questions

Our study was designed to answer the following research questions:

**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 3.5a, 3.5b, 3.6a and 3.6b. 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 3.4b. The third task used asynchronous parallelism and was based on the program shown in Figure 3.5a.

**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 par-
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 3.7 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 or 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 3.7.

**RQ4: Do experts have different preferences than novices?**

Although 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.

### 3.3.2 Survey Design

Table 3.7 shows an overview of our survey structure and the participant count for each part of the survey. As we recruited participants, we provided them with a brief demographic questionnaire to classify them as novices and experts. We then asked them to complete three tasks to measure how well they could 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.

**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). Of those participants, we include 273 who completed at least one task of the survey in our results.
<table>
<thead>
<tr>
<th>Introduction and Randomization:</th>
<th>273 Participants Started Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recruitment</td>
<td>110 Novices</td>
</tr>
<tr>
<td>Task 1</td>
<td>50 Vertical 60 Horizontal</td>
</tr>
<tr>
<td>Task 2</td>
<td>45 Vertical 53 Horizontal</td>
</tr>
<tr>
<td>Task 2 (alt.)</td>
<td>41 Vertical 50 Horizontal</td>
</tr>
<tr>
<td>Usability Questionnaire</td>
<td>84 Completed Questionnaire</td>
</tr>
<tr>
<td>Completed</td>
<td>191 Participants Completed Survey</td>
</tr>
</tbody>
</table>

**Figure 3.7:** 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 3.5a, 3.5b, 3.6a and 3.6b.

**Introduction and Randomization:** Immediately after our participants had completed our demographic questionnaire, we showed them a one minute long training video. This video introduced them to the overall idea of robot programming and the kind of robot that our designs target. It also showed them how a robot performs a simple pick-and-place procedure with one arm. It did not show any programming interface to ensure that we didn’t bias participants.

After participants had finished watching the video, we divided them into randomized groups. Each was assigned to a different design candidate among the four we have presented previously. All groups followed the same remaining survey procedure except for the mock-up programs they were shown and minor changes in survey texts to match the individual layout of each design.

**Tasks:** 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. 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.

To avoid overwhelming participants, we ordered the three tasks follow-
ing 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.

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 [36]. 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 3.4b. 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 the purpose of the given program. 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 3.5a through 3.6b 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.

After the second and third tasks, we showed participants an alterna-
tive 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 3.7 shows the resulting order in which each participant group was shown each design.

**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) [108]. Since some of the SUS questions only apply in a context where users can 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 [109], 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 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.

### 3.3.3 Survey Results

All of the following 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 3.7. A total of 191 participants (84 novices and 107 experts) finished the entire survey.

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

Based on the 110 novice responses, Table 3.1 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.
Table 3.1: Box-plots of the scores of novices for each task’s comprehension questions. Participant numbers (n) are reported in Table 3.7.

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.

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.

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

We measured participants’ preferences using the SUS scale, which ranges from 0 to 100 points. This scale should be interpreted linearly, but we can instead convert SUS scores into a percentile rank which compares our
designs to other systems that were evaluated using the same scale. We used only 6 of the 10 questions from the standard questionnaire, but previous research indicates that the scaled scores are still equivalent to the same percentiles [109].

Figure 3.8 shows the raw scores of our designs on the SUS percentile curve. Meta-studies find that the average SUS score across studies is 66 [109]. 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.

**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 3.2 summarizes how participants rated three pre-selected design differences as Likert items on a 5-point 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 whether participants that preferred different designs had different factors that influenced them. We grouped the factors by preferred design as shown in Table 3.2. 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 as-
signed 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 ($F(3, 77) = 0.27, p = .85$).

**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), but none said this about any
Table 3.2: Perceived importance of individual design aspects for novice participants, overall and grouped by their preferred design: 1 is least important, 5 is most important.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>Icons/Text</th>
<th>Uniformity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>1 2 3 4 5</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>Pref. VT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pref. HR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, and 2 participants criticized them. These results are consistent with the previous observation that although 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”. Although 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.”

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.

**RQ4: Do experts have different preferences than novices?**

Figure 3.9 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), but 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 applied the same approach as for RQ3 to 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.

### 3.4 Further Studies on Coordination

In our previously presented survey, 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 3.2, 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.

Our survey measured how well participants could comprehend given mock-up programs, but 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 two follow-up studies. The first study is a small-scale user study on a front-end prototype implementation of the design we deemed best, and included in our paper “Comparing Block-based Programming Models for Two-armed Robots”. The second experiment is a larger controlled experiment conducted on a physical robot and is the primary contribution of the paper “Side-by-Side: A Case Study Enabling Parallel Programming of Robotic Arms by End-Users” that we contributed to. In the following, we describe the first study in detail and provide a brief
summary of the findings of the second study.

3.4.1 Front-end Prototype Study

To complement the findings of our previously presented survey, we conducted an interactive study using a front-end prototype of the design we deemed best. This study was smaller than our survey but provided important insight into how novices could use our design to solve realistic robotics tasks. It 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 also 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 [108].

64
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, but the others had some other previous programming experience.

A total of 10 participants (3 novices) correctly solved Task 1 and 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. We therefore conclude that most participants could solve two realistic coordinated robot programming tasks using our prototype.

### 3.4.2 Large-scale Physical Robot Study

Based on the promising findings of our survey and prototype study, we implemented a fully functional prototype with the name *Duplo*. As illustrated in Figure 3.10, the system uses the program flow and synchronization mechanism that we presented in our design candidate *E-V* in Figure 3.5b. However, due to technical limitations, we were unable to implement blocks that span across the programs for both arms or the visual mirroring of blocks depending on the side of the canvas they are placed on.

To evaluate *Duplo*, we compared it to RobotStudio Online YuMi [110] that we described in Section 3.1.3. ROY is the only robotics tool of which
we are aware that targets end-users and supports coordinated programs that target two robot arms at once. In addition, ROY provides a similar set of features as Duplo, with many commands being available in both languages. This makes ROY an ideal candidate for evaluating the impact that the block-based editing paradigm and the synchronized blocks have on end-users programming real-world robots.

To compare the two programming environments, we performed a randomized controlled experiment with 52 participants. Our participants, primarily university students with little to no programming experience, were randomly assigned to one of the two programming systems and given a brief 15-minute long introduction on how to use it. They were then presented with a 105-minute long programming task that consisted of multiple stages in which participants had to coordinate two robot arms to jointly carry and assemble a series of items. We recorded whether participants completed each stage of the programming task, how long they took to do so, the number and kind of programming mistakes they made, and how many attempts they needed to test their programs.

Figure 3.11 summarizes our findings regarding completion times and success of both participant groups. We found that on average, participants who used the Duplo block-based environment could solve the given tasks faster and with greater success. Compared to only 46% of ROY users who successfully completed all stages of the programming task, 80% of Duplo users solved all stages. Participants who used Duplo were also faster, requiring only 53 minutes on average compared to 75 minutes on average for ROY users. A closer analysis of the two participant groups shows that Duplo users made fewer programming mistakes and produced cleaner code with
fewer unused commands or redundant definitions of robot locations. These differences might explain their better overall performance and demonstrate the benefits of our design for coordinated block-based programming.

3.4.3 Limitations

Our studies have a number of limitations, which we discuss at this point.

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. 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 first prototype study were also employees of this company in a single location, and those recruited for our second, larger study were all undergraduate students at a North American university. Despite being broad overall, these recruiting methods might have introduced biases, of which we consider the cultural ones to be most relevant for our results. In particular, North American or European users of our tools might prefer a different reading direction than those from other countries. It remains an open question whether our findings generalize to users from different cultural backgrounds.

User Training: All of our studies 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, and for our other studies we had to limit how much time users can spend familiarizing themselves with the given environment and tasks. 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: Time limitations have not just affected the training but also the tasks that we could provide to our users. We have aimed to select realistic tasks of different styles and sizes across our studies, and motivated our choices for each of them 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 beyond those used by us.
**Figure 3.12:** Design space based on synchronization model and program flow direction. Each of the designs shown in Figures 3.5a, 3.5b, 3.6a and 3.6b corresponds to one cell of this table.

**Attrition Bias:** The survey results we presented 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 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 is negligible.

### 3.5 Discussion and Summary

In this Chapter, we have presented and analyzed four block-based design prototypes that use visual programming elements, and evaluated them analytically and empirically. Figure 3.12 shows an overview of the findings of our survey: explicit synchronization allows users to express a wider range of programs, and vertical program flow was perceived as more usable by our participants. This outcome motivated us to conduct two follow-up studies. Both of these studies confirmed that the design with explicit synchronization and vertical flow allows end-users to successfully program robot arms. In the remainder of this section, we discuss these findings and how they might apply to other programming areas as well.
3.5.1 Using Visual Program Alignment To Represent Complex Program Semantics

Our analytical approach and preliminary studies alone do not explain why our explicit synchronization was accessible to end-users. The design uses explicit barriers for synchronization, which 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 could understand barriers quickly and with minimal external guidance. They could 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.

We believe that the addition of visual elements to blocks has played a crucial role in making our design end-user friendly. We showed the commands for each arm side-by-side and visualized the barrier as a single block that connects both arms. The design of barriers in our follow-up studies did not have the same visual representation of barriers as a single block, but still aligned them vertically and therefore created a visual connection. Unlike in traditional text-based programs, novices do not need to manually find and align concurrent actions. We believe this to be a powerful visual feature of our language that was not present in any previous work. The feature is also unique to block-based programming as it cannot be easily replicated in text-based tools where programmers manually determine the layout of program code.

3.5.2 Visual Support for Programming Beyond Two-Armed Robots

Our findings advance the area of robotics by making the programming of two-armed robots accessible to end-users. Building on previous research that has shown that end-users can learn to successfully program single-armed robots [7, 36], our own observations confirm and extend those previous results. Remarkably, although we asked novices to solve more complex programming tasks than the previous work, we did not give them substantially more guidance or learning time. This did not seem to negatively impact their learning or performance.
Based on our findings, we suspect that neither the previous work we surveyed in Section 3.1 nor our own designs and tasks have pushed novices to the limit of their learning abilities. Our findings also suggest that with the right level of support, end-users can understand and even write programs that feature extensive parallelism. It appears that end-users do not need to have mastered simpler use cases, such as single-armed programming, before they can learn how to write complex programs that feature coordination. It is up to future work to explore whether it is possible to introduce even more complex models of parallelism to end-user languages and how to represent them visually.

All our design candidates have two inherent limitations that we consider challenging to overcome but that would substantially improve their expressive power. First, all are restricted to coordinating two robot arms at a time, and second, they do not support any flow of data between the parallel programs. We discuss the challenges that arise with the addition of more arms in Section 6.3. However, incorporating data-flow is a particularly complex extension, as the presence of variables and mutation alone is a common source of confusion for beginner programmers [111]. Our designs avoid these language features as they are not necessary for the use case we targeted. Instead, we focus only on physical state, for example in the form of whether a robot hand is open or closed. This state is inherently arm-specific and easy to inspect and reason about for end-users. We believe that any form of data-flow that is more complex and introduces new sources of errors will pose a substantial new challenge that goes beyond the scope of our current prototypes and likely requires substantial follow-up work.
Chapter 4

Block-based Decomposition: Mobile Robots Operating Multiple Workstations

Stationary robot arms can support factory workers with a wide range of tasks, but their lack of mobility can hinder their use in practice. Their limited range can make it tedious to move items between different processing or assembly steps. More importantly though, when robots are fixed to a pre-determined location, they cannot easily be re-purposed or autonomously switch between multiple tasks. This means that stationary robots are primarily useful in contexts where their task is known ahead of time and the chance of any changes to their environment is low [112].

Mobile robots that can move autonomously between physical locations have increased in popularity over the previous years. [40,112] Compared to stationary robots, this makes them substantially more flexible and allows them to solve tasks that are spread across multiple work sites. In addition, mobile robots can autonomously switch between different tasks and work sites. For example, when a mobile robot is idle, it can temporarily work on a different task. Therefore, a single mobile robot can do the same work as multiple stationary robots.

Assigning multiple responsibilities and more complex tasks to a single robot means that the complexity of the corresponding program grows as well. When tasks are distributed over multiple sites, programmers must keep track of which operations are executed at each site and at which points the robot has to switch its location. Particularly for end-users, managing large programs can be hard. They must simultaneously focus on both the bigger picture and the individual steps of the task that they want to solve.

Traditionally, programming languages offer support for structuring code and decomposing it into smaller, mostly self-contained fragments such as functions or methods. Most block-based languages support abstraction and program decomposition as well, often in the form of user-defined procedures.
or first-order functions. However, just because abstraction mechanisms are offered in principle does not mean that users can apply them to the concrete program they try to solve. All available evidence suggests that users of block-based systems tend not to decompose their programs [113], even though decomposition is crucial to successfully write larger programs. We therefore speculate that existing block-based systems do not sufficiently support their users through the set of features they offer to them.

In this chapter, we present a block-based programming environment that guides users as they decompose their programs. This work is based on the paper “Can Guided Decomposition Help End-Users Write Larger Block-Based Programs? A Mobile Robot Experiment” (published and presented in OOPSLA 2022) [114]. The system we present specifically targets the creation of programs for mobile robot workers. These consist of high-level commands that move the mobile robot between different workstations and low-level commands that direct the robot to execute local tasks such as picking up an item. The presented programming environment makes use of this common task structure and explicitly divides programs into a high-level movement component and a low-level local component. Instead of traditional functions, the environment features “tasks”, which resemble parameter-less procedures, but impose domain-specific restrictions that limit how they can be composed and which instructions can be used within and outside of them. To support this language design visually and make programs easier to navigate, the environment is divided into two side-by-side canvases, one for high-level and one for low-level programming. Figure 4.1 shows our prototype and an example program with a single task selected and displayed on the right canvas.

We hypothesize that compared to a traditional block-based environment, our design candidate encourages users to give their programs a sensible, domain-specific structure. We further hypothesize that this structure encourages users to re-use code rather than create code clones. These speculative benefits come with a potential cost: in comparison to free-form procedural abstraction as it is supported by current block-based languages, users must follow the program structure that our environment imposes on them, which might not always be best. However, we hypothesize that for mobile robots, this cost is minimal in practice, as few end-users would be able to find a better solution in a traditional environment than the one our environment imposes on programs.

To validate our hypotheses and evaluate whether our environment can make larger programs easier to understand and write for end-users, we conducted an experimental evaluation with 92 novice participants recruited via
AMT. We randomly assigned each participant to either use our guided programming environment or a traditional block-based programming environment with support for custom blocks. We asked all participants to solve a series of three tasks that cover cases of interest: a small task that would not typically require decomposition, a large task with an obvious and easy to decompose structure, and another large task with a more challenging structure that is not optimally suited for our guided approach. Consistent with previous studies [28, 35, 36, 115], we found that participants from both cohorts performed well on the small task. However, for the tasks that required larger programs (and even the one that was easy to decompose), many participants in the cohort that used the traditional environment did not use abstractions or structure their programs at all. These participants were less successful at completing the given tasks. In comparison, the participants who used the environment with guided program decomposition were over 20% more likely to complete their tasks successfully, even for the task where this decomposition was sub-optimal.

Figure 4.1: Proposed programming system showing two tasks and side-by-side canvases. The left canvas shows the main program and the right one shows the body of the currently selected task(s).
4.1 Decomposition in End-user Languages

Block-based programming was created with learners in mind, who usually write brief and self-contained program snippets that serve a single purpose. Nonetheless, several popular block-based programming languages try to support larger programs by allowing users to define custom blocks. These blocks work similarly to custom functions and are highly expressive in theory, but as we outlined in Section 1.1, their integration into block-based environments is often shallow. For example, Figure 1.3a shows the definition and usage of a custom block in Scratch [4], which resemble the definition and call of a procedure in a traditional, text-based language. Also as in text-based languages, there is no explicit separation between the function body and the surrounding program built into the environment. This means that users can freely move code between the different scopes of the program and create programs that any expert programmer would immediately identify as faulty. For example, Scratch does not enforce scoping: it allows a function parameter like $x$ in Figure 1.3a to be dragged into the main program body and misused outside of its definition range. Scratch does not provide any feedback in these cases, and its design feature of not presenting run-time errors to the programmer can cause even more confusion as beginners try to understand the intended semantics of procedures.

Some block-based programming languages go even further than Scratch in terms of replicating established abstraction and decomposition mechanisms from traditional programming languages. OpenRoberta [44] supports first-order functions that return a value, Snap! supports higher-order functions [49], and App Inventor for Android supports object-oriented programming [48]. However, none of these languages attempts to guide programmers in how to use these features or to visualize their semantics. Therefore, unless these tools are accompanied by external support, such as a tutor or an extensive tutorial, they are hardly a good fit for end-users or other novice users.

Unfortunately, there is limited empirical data available on how end-users decompose their code when they are left to their own devices. Surveys of public repositories for block-based code suggest that few programs use any decomposition at all, but it is unclear what percentage of those code bases are written by adults or meant to be used productively [37, 38]. The same surveys also find that many block-based programs contain long sequences of instructions and frequent code duplication. Work that focuses on professional developers suggests that this style of code can be detrimental to program comprehension, even for those professionals [116]. One expects in-
Figure 4.2: Structures found in end-user programs. Traditional programming languages treat all of them as equal alternatives, but we speculate that a hierarchical structure benefits end-users the most.

Drawing from previous work, and our own observations of end-users writing block-based programs, we identify two common patterns in end-user programs: one subset of users always creates programs as a single, linear sequence of commands without any attempt made to decompose the code. We call this style unstructured programming and illustrate it in Figure 4.2a. The other subset of users appears to be aware of the need for decomposition and attempts to use the tools available to them (e.g. functions) to structure their code. They create structures as shown in Figure 4.2b, which we call free-form programming. These structures can take any shape that is supported by the programming language, potentially mixing different styles and using features ad-hoc instead of planning ahead. As a result, the code is often inconsistently structured, more complicated than necessary, and therefore harder to understand and more error-prone.

A wide and unrestricted program design space can benefit experienced programmers, such as end-users, to be even more obstructed by poorly structured code, and the observations we present in Section 4.3 support this assumption.

In fact, learning how to use functions has been identified as a Threshold Concept in computer science education: understanding how to use functional
abstraction dramatically benefits learners, but is challenging to learn [117]. If parameters are used to call functions, learners struggle to understand the semantics of pass-by-value versus pass-by-reference [118]. Users also struggle to understand the related concepts of variable scoping, such as whether they can access global variables within a function, and are confused by corner cases such as variable shadowing [118]. In addition, users might accidentally discover and try to use techniques like recursion, which can lead to confusion and misconceptions [119]. Therefore, we speculate that a system that reduces the abstraction decision space and provides guidance to users, illustrated by Figure 4.2c, can benefit these users and improve their ability to reasonably structure their work and consequently construct larger programs.

4.2 Design Approach: Guided Decomposition

We propose a programming system that provides more guidance to novice users, such as end-users, than existing environments. The goal of this system is to reduce the design space to those decisions that these users can make effectively, and to explicitly give them the information needed to make them. We believe that programming domains typically come with a set of embedded structures and hierarchies that can facilitate decomposition. End-users that are familiar with a domain have an informal understanding of these decomposition rules and how to apply them. A system that targets end-users should build on top of this understanding, provide the necessary formalization for domain-specific conventions, and enable programmers to decompose their code in a way that matches these conventions.

Our system, shown in Figure 4.1, supports a fixed hierarchy of parameterless functions, which we call tasks. Figure 4.3 shows the grammar of the underlying programming language and illustrates how tasks effectively divide the language into two distinct sub-languages: one where tasks are composed and one where they are defined. Our system further supports this hierarchical design with an environment that splits the program editor into two side-by-side canvases, where one provides the user with an overview of their program and the other shows the current task. Our hypothesis is that this approach for program decomposition and code navigation offers users a simplified, yet powerful way to structure their programs in accordance with their understanding of the problem they are trying to solve. Note that we use the Cognitive Dimensions of Notation throughout this section. We introduced this framework in Section 2.4 and underline and italicize terms
Figure 4.3: Grammar(s) of the programming language used in the proposed system. A program (left) can only contain calls to tasks at the top-level, and that task definitions (right) take place in a separate programming canvas with its own syntax. As Section 4.2.4 describes, there are some differences between the language’s semantics shown here and its block-based presentation shown in Figure 4.1. Also note that locations are defined via a visual location picker that is not represented in this grammar.

4.2.1 Why Do End-Users Not Use Functions?

Kallia and Sentance [117] noted two factors that contribute to beginners’ lack of function usage, and that we believe must be overcome if we want to make them accessible to end-users. First, functions are abstract programming mechanisms that do not directly match the tasks users are trying to accomplish. Translating a high-level task into appropriate sub-tasks, that is performing functional decomposition, is a Hard Mental Operation, even for professional programmers [120] and likely more so for novices [121]. As they decompose their main task into sub-tasks, users may become overzealous, for example creating many functions with only a single block (top right of Figure 4.2b). The unconstrained nature of task decomposition can quickly lead to confusing callchains (e.g., the calling relationships in Figure 4.2b), or even unintended recursive definitions (that likely lead to non-termination). These potential misuses of functions contribute to users creating programs structured like in Figure 4.2b, where functions can confuse inexperienced users more than they help them.

Second, beginner programmers with no prior experience using functions may not see their benefit until it is too late [122, 123]. Novice programmers may perceive functions as requiring Premature Commitment, as they have an up-front cost in terms of adding blocks and the understanding program...
flow, and they must decide whether to use them before they know how large their program will ultimately become. Thus, users begin writing programs that have no structure, and as these programs grow they become unwieldy.

Block-based systems have a low *Viscosity* in theory as they allow users to quickly re-arrange programs via drag-and-drop and have built-in hygiene for names. However, compared to refactorings that are found in professional development tools, these features provide little guidance to programmers. This is especially unhelpful for novices, who might not have a clear vision of how to improve a program’s structure, even if they are aware of its current issues. As we will see in Section 4.3, inexperienced users tend to simply split programs into arbitrary chunks, which provides minor *Visibility* benefits, but little benefit for their understanding of the code. For this reason, we also do not believe that simply highlighting overly long functions, or restricting the maximum length of continuous code blocks, can be a solution to this problem.

**4.2.2 Domain-Specific Program Decomposition**

To overcome the factors that prevent inexperienced users from using functional abstraction, we have developed an alternative design for a block-based environment that encourages the systematic decomposition of programs. Our language design can be summarized with two main points: first, to give decomposition more meaning to end-users, and to better align with their programming goals, we introduce “tasks”. Tasks are parameterless procedures that are assigned to a single workstation and can only contain instructions that relate to this workstation. Second, we make tasks mandatory by splitting the environment and the underlying language into two distinct components: the main program editor, which can only be used to compose tasks, and a task editor, which defines the instructions of each task.

Tasks are designed to explicitly support end-users as they create programs for the mobile robotics domain. By explicitly catering to the kinds of programs end-users create for this domain, our design aims to naturally guide end-users to create tasks that promote the single responsibility principle. Each task is assigned to a specific workstation, and the instructions available to define a task are limited to those that can be executed locally at this workstation. Therefore, the task body primarily contains instructions that move the robot’s arm and manipulate the workstation (e.g. by picking and placing items). Movement between workstations happens implicitly as the robot switches between tasks.

To reduce the decision space for inexperienced users, our system design
imposes a domain-specific program structure. We believe the hierarchical program structure with a fixed two-layer call graph makes our system easy to understand, yet expressive enough to decompose larger mobile robot worker tasks. We therefore do not offer any additional, more complex mechanisms for decomposing programs beyond tasks, such as globally or locally defined functions. The system imposes certain restrictions on users. First, users must decompose all programs into tasks, even small programs where an entirely flat program structure would be straightforward to read. This limitation can reduce the visibility of small programs, but also prevents programmers from making a premature commitment to a flat program structure when programs must eventually grow larger. We further believe that making large programs easier to comprehend supports end-users where they need it the most. Second, the hierarchical design with two fixed levels rules out programs with multiple decomposition layers (e.g., through nested function calls) or recursion. This can be a limitation for experienced programmers and lead to redundant code, potentially increasing the resulting language’s viscosity compared to one without these restrictions. However, we consider the limitations to be beneficial in the context of end-users, who are more likely to make mistakes when the control flow of a program becomes complex. In fact, many other block-based systems also prevent or warn users about using recursion [49, 62], but do so in more intrusive ways such as disabling blocks or showing error messages.

4.2.3 Aligning Physical and Programmatic Scope

Complex scoping rules can be confusing and unintuitive for novice programmers [118]. Therefore, we have a system that simplifies the concept of programmatic scope and matches it with the physical scope in which the mobile robot conducts work. Since each task is localized at a single workstation, a task should only contain commands that can be executed within the physical scope of its assigned workstation. For example, movement commands cannot take place within a task, and if there are different types of workstations, there might be other restrictions based on the physically available tools and the workstation layout. This is analogous to how most programming languages have built-in mechanisms to limit the lexical scope that is accessible from within a given block of code. However, when those rules become complex, they can increase the error-proneness of code and without sufficient feedback, understanding them can become a hard mental operation. Block-based languages can go beyond the traditional checks that text-based languages perform to enforce scoping rules. For example, these systems can
actively filter the syntax they present to users and provide only the valid options. In a structured programming system like the one we propose, we can leverage this feature.

To provide end-users with further guidance, our system design presents only those commands that are within the physical scope of a robot task. Similar to previous work on block-based robot programming [7, 101], our system supports only one variable type: locations. These locations, for example the target coordinates where an item is supposed to be placed, can be defined using the programming environment. Previous work has allowed users to select target locations physically by moving the robot arm into the intended position [7]. That work does not discuss their design decision in the context of CDN terminology, but we consider it highly beneficial for the Closeness of Mapping that users experience between the programming environment and the physical system. For our system, which currently relies on a virtual robot simulator, we had to replace this approach with a reasonable alternative. We chose a visual picker for target locations, which is a less direct representation than physical human-robot interaction, but still substantially more direct than a purely text-based specification. We have further provided useful feedback in the form of meaningful domain-specific error messages and warnings such as “Robot is already carrying an item!” when programs try to carry multiple things at once. We cannot compare the visual location picker used by our system to the physical approach used in previous work. However, we believe that both are reasonable, user-friendly choices.

In previous work, all locations were always defined globally [7, 101]. However, since our system allows programs to span multiple workstations, this approach is no longer appropriate. Instead, locations must be specific to the workstation at which they are accessible. Since our system assigns each task to exactly one workstation, we decided to make user-defined locations task-specific, similar to local variables defined within traditional functions. This straight-forward mechanism, made possible by matching physical and programmatic scoping, saves users the Hard Mental Operation of manually filtering the available locations.

4.2.4 A Visual Programming Environment That Supports Decomposition

Our design is implemented as a multi-canvas environment, as shown in Figure 4.1. The main program appears in the left panel of the figure. The middle panel of the figure shows the task that has been selected in the main
program. A robot simulator is shown in the right panel of Figure 4.1 as our prototype implementation uses a simulation environment. This environment offers high Visibility, concurrently displaying the main body of the program on the left canvas and the body of the currently selected task in the middle canvas. As users edit tasks, we concretize their editing by including the currently selected workstation in the task definition’s header. This has no impact on the language’s semantics or the ability to re-use a task for other workstations. However, the concretization might mislead users into thinking that they are only changing one instance of a task at a time. To highlight that they are in fact changing all instances of a task at once, other instances are also highlighted in the main program when any of them is selected.

Previous work has either made all function definitions visible at once on the same canvas (e.g. Scratch), or attempted to spread code over an arbitrary number of small, isolated canvases [124]. Although the latter approach shares some of the benefits to our design, it suffers either from Visibility issues when too much code needs to be displayed, or introduces additional Hard Mental Operations as users must customize their environment manually. Our design on the other hand uses a fixed number of canvases that are populated automatically and that follow the overview-detail paradigm that is commonly used in interface design and information visualization [125]. This approach tries to balance Visibility with avoiding Hard Mental Operations.

4.3 Experimental Evaluation

Previous studies of block-based environments and how end-users create programs in them have used small example tasks [36, 115]. Due to their small size (usually 20 blocks or less), these programs are not complex enough to require the use of decomposition. Our study focuses on how end-users perform when tackling larger tasks that do not fit on a single screen.

In this study of 92 self-identified end-users recruited via AMT, we trained participants to use a block-based programming environment. We then asked them to solve 3 tasks of increasing size and difficulty. One participant group used a traditional block-based programming environment with support for parameterless functions, and the other used the environment with guided decomposition presented in Section 4.2. We measured how participants decomposed their programs in each environment, and how the use of decomposition affected their success in solving the given tasks. In the remainder of this section, we describe our study in detail.
Figure 4.4: The traditional block-based development environment used by the TR Cohort of the programming study. It offers the same command blocks and simulator, but uses a single canvas and provides free-form function blocks that are defined and used in this canvas.

4.3.1 Research Questions

Our study investigated the following research questions:

**RQ1** How do end-users decompose their programs in a traditional block-based environment?

**RQ2** Does the use of program decomposition impact the task success rate and task completion time of end-user programmers when writing larger programs?

**RQ3** How does an environment with guided program decomposition change the way end-users write larger block-based programs?

**RQ4** Does guided program decomposition impact the task success rate and task completion time of end-user programmers when writing larger programs?

4.3.2 Experimental Design

We recruited our participants through the platform *Amazon Mechanical Turk (AMT)*, which pays users to conduct online tasks. Using a short pre-questionnaire, we selected only those users that indicated that they had less than one year of programming experience and did not have experience
programming industrial robots. Participants that passed this screen were randomly divided into two cohorts. The first cohort (which we call TR from now on) used a traditional block-based environment to complete the study, and the second cohort (which we call GD) used the novel system with guided decomposition that we presented in Section 4.2.

Figure 4.4 shows the traditional block-based programming environment that participants in TR used; Figure 4.1 shows the novel environment that those in GD used. Both environments include an editor on the left side and 2D simulator on the right side. The editor for the GD cohort required two canvases, as described in Section 4.2. The editor for cohort TR was a single canvas with the option to add simple, parameterless functions as they are supported by many existing block-based environments. As existing block-based environments use a wide range of terms to describe functions to their users, we decided to call them recipes in our own interface to match the terminology used in a previous industrial robot programming environment [7]. This environment also featured a separate “Move to Station X” command to replace the implicit robot movements in the environment with guided decomposition. All remaining commands, such as those for picking and placing items, were identical to the other environment, matching the statements listed in Figure 4.3.

We used a series of 3 tutorials to train all participants to use a block-based programming system. The first two tutorials focused on the system’s core functionality: commands to move from station to station and to pick up, carry, turn, and place items. The third tutorial taught users to decompose programs (using functions or tasks, respectively), and to re-use code by calling the same function or task multiple times. The tutorials’ content and flow were as identical as possible for each cohort. All tutorials are available on the previously linked website. Although we intended the tutorials to take approximately 15 minutes in total, we did not limit the time that participants were allowed to spend on each tutorial. A small number participants in both cohorts exceeded the intended tutorial time (see Figure 4.7), but the majority completed them without our initial time estimate.

After participants completed the tutorials, we asked them to solve a series of 3 tasks. Each task consisted of a brief description and an image showing the intended outcome of the task. A simulator (see Figure 4.4) allowed participants to test their solutions, and they were allowed an unlimited number of attempts. However, we did limit the time that participants were allowed to take for each task, giving them one final chance to submit their solution after they exceeded the maximum allowed time. Independent of whether a participant’s solution was correct, we saved their final attempt
for later evaluation.

The three tasks, in the order we gave them to participants, were:

**Task 1:** A short, toy-sized task with a time limit of 10 minutes that we intended as a warm-up for participants. Users were asked to move two boxes between stations, which can be accomplished using just 13 blocks without any decomposition, or using 17 blocks in the system that requires task decomposition.

**Task 2:** A larger task with a time limit of 15 minutes that we designed to be repetitive and therefore benefit substantially from code re-use. Users were asked to move a stack of 3 boxes, one at a time, from one side of a workstation to the other, for four different workstations. The task provided participants with a functioning program for a single workstation, consisting of 15 blocks, which they had to apply identically or with slight variations to other workstations. We specifically chose this task because it consists of a number of spatially isolated and therefore easy to separate sub-tasks. This meant that users who chose to decompose their programs could avoid redundant work by re-using or at least efficiently duplicating code. We believe that this task is representative of the type of work for which our proposed domain-specific decomposition is the most effective. We therefore expected users of the traditional environment who decomposed their programs to do so in a similar way as we imposed on users of the other system. The task also featured some clear redundancy and therefore potential for re-using code in both systems, reducing the minimum number of necessary blocks from 56 without any decomposition (or 62 in the system that required task-based decomposition) to just 46 in an optimal solution.

**Task 3:** Another large task with a time limit of 15 minutes, although for this task the ideal structure was less obvious since the components of the task were not spatially isolated. Users were asked to move boxes between stations, each time swapping places with another box. For this task, participants started without being given code, but we asked them to solve an easier part of the problem with a single box on each side first, and then approach a more complex version with two boxes that could be solved by re-using parts of their code. We believe that this task is representative of a type of work where our proposed decomposition strategy is sub-optimal, since it requires splitting tasks into more (and smaller) sub-programs than users might find intuitive. This also becomes clear when comparing block numbers: A solution without any decomposition can solve this task using 48 blocks, and an optimal solution with code re-use requires 40 blocks; the strategy imposed by our presented system requires 54 blocks with or 64 without re-use. We
therefore see it as useful for both evaluating if such overly strict decomposition requirements harm our participants’ performance, but also whether those users without guidance are able to find a (potentially different) style of decomposition that is useful for them.

4.3.3 Experimental Results

In the following section we discuss the results of our study, aligned with the research questions we presented in Section 4.3.1. As a reminder, we refer to the cohort of participants who used a traditional block-based programming system as TR, and to those we used our proposed novel system as GD.

RQ1: Program Decomposition in a Traditional System

Figure 4.5 shows how participants in the TR cohort used functional abstraction, separated into Figures 4.5a, 4.5b, and 4.5c which correspond to Tasks 1, 2, and 3, respectively. The right of each figure also shows how many participants re-used code as a result of applying functions.

For Task 1, the shortest task, participants used an average of 13.8 blocks (median: 13), of which 8.5 blocks (median: 8) were statements3 and defined an average of 0.3 (median: 0) functions. Participants defined between 0 and 2 functions, with 8 using at least one. The following bar chart shows the full distribution of function usage4:

Only 8 (16%) out of 49 participants used functions for this task at all, and 41 (84%) wrote their program as one continuous block of code. Therefore, the average number of sequential statement blocks contained in a single chunk of code (either the main program or a function) is 8.1 blocks (median: 8), which is close to the total average program size. As shown in Figure 4.5a, none of the 8 participants that defined 1 or 2 functions re-used code (by calling the same function more than once).

For Task 2, a larger task, participants used an average of 54.1 (median: 55) blocks, of which 29.9 (median: 31) were statements, and defined an average of 2.1 (median: 3) functions. Each participant defined between 0 and 4 functions, with 29 using at least one. The distribution is shown in detail in the following bar chart:

3When we refer to statements in this sub-section, we include function calls and instructions for the robot, but not locations or function headers

4Light green bars (to the left) show successful tasks, dark red bars (to the right) show failed tasks. The height of the bar corresponds to the number of functions, dots represent solutions where no functions were used.
A higher percentage of participants used functions for this task. However, 19 (39%) did not use any functions, specifying their entire program as a single, continuous block. As shown in the middle of Figure 4.5b, of the programs that contained functions, the majority (69%) were structured in a way where each function describes the robot’s actions at a workstation. This style of program decomposition is the one that we impose on users in the guided environment. The remaining 9 programs (31%) were decomposed in different ways, for example by extracting specific pick-and-place sequences. We tried to categorize these programs but were unable to find any patterns of note that appeared in 3 or more programs. Overall, participants used an average of 21.1 sequential statement blocks (median: 10) per code chunk (function or main program), with those participants who did not use any functions heavily skewing the distribution.

Of the 29 function-using participants, 25 (85%) gave their functions custom names. The remaining 4 (15%) did not change the default names assigned by the editor (i.e. “do something” with numeric suffixes to ensure uniqueness). However, only 10 of the 29 participants that used functions (34%) called any of them more than once, as shown on the right of Figure 4.5b, which meant that there were many missed opportunities for code reuse.

For Task 3, another larger task, the average solution contained 49.0 blocks (median: 49), of which 31.7 (median: 31) were statement blocks. Each participant defined between 0 and 4 functions, with 13 participants using at least one, as detailed in the following bar chart:

![Bar chart showing function usage for Task 3]

Only 13 (28%) of our participants used functions at all, a lower percentage than for Task 2, resulting in only 0.7 functions being used on average overall (median: 0). This led to less structure overall and even less code re-use than for Task 2, as shown in the middle-right of Figure 4.5c. Notably, 33 participants (72%) constructed their entire program as a single, continuous block of code, as seen in the middle-bottom of Figure 4.5c. Of the remaining participants who did decompose their programs, 5 (38%) used a very coarse structure that split their code into exactly two sub-programs, following the two parts outlined in the task’s description text. Only 3 participants (23%) used a workstation-based style that resembled the one used by the environment with guided decomposition. The remaining 5 participants (38%) used different styles that split the task into 3 or more parts and showed no common decomposition pattern. Overall, the average number of sequential statements used per code chunk (main program or function) is 25.4 blocks (median: 27), which mostly representative of those participants who did not
use functions. When just considering function users, the sequential statement number is 11.9 (median: 11), which still suggests of a fairly coarse decomposition of code.

**RQ1 Summary:** When using a traditional block-based system, most end-users did not use functions to decompose larger tasks, resulting in a single, unstructured code block for 65% of the submitted solutions for Tasks 2 and 3.

**RQ2: Impact of Traditional Decomposition on Success and Time**

Tasks 2 and 3 were complex, and thus we hypothesized that participants that used functions would have better success. As can be seen in the middle of Figures 4.5b and 4.5c most participants that failed did not use functions to structure their programs. For Task 2, of the 29 participants who used
functions only 5 (17%) failed, and of the 19 participants that did not utilize functions, 9 (47%) failed. For Task 3, of the 13 participants who used functions only 3 (23%) failed, and of the 33 participants that did not utilize functions, 18 (55%) failed. Taking both tasks together, when a participant used at least one function to solve a complex task, they succeeded 81% of the time, whereas when they did not, they succeeded 52% of the time.

Although we saw a correlation between function usage and success rates for both Tasks 2 and 3, we only saw a substantial difference in how fast participants solved a task for Task 2. For Task 2, the participants that used functions finished in 9.1 minutes on average, and the participants that did not use functions finished in 13.4 minutes. For Task 3, the participants that used functions finished in 14.5 minutes, and the participants that did not
use functions finished in 14.0 minutes.

**RQ2 Summary:** End-users that added functions to their programs were more likely to be successful when working on larger tasks.

**RQ3: Program Decomposition in an Environment with Domain-Specific Guidance**

Figure 4.6 shows how participants in the GD cohort decomposed their code to solve each task. By design, the development environment with guided program decomposition required all participants in this cohort to decompose their programs, including the toy-sized Task 1. To avoid ambiguity between our three study tasks and the “tasks” that the environment offered participants as a way to decompose their programs (see Section 4.2), we refer to the latter as “sub-tasks”. Since the environment with guided decomposition does not allow differences in decomposition style, we divide participants based on the number of sub-tasks they created in Figure 4.6.

For Task 1, participants used an average of 16.3 blocks (median: 17), of which 7.8 (median: 8) were statements, and defined an average of 3.8 sub-tasks (median: 4). The total number of blocks is higher than for the TR cohort. After a manual inspection of the code written by participants, we find that this difference can be primarily attributed to the overhead caused by additional sub-task definitions. The average number of sequential statements per code chunk (main program or sub-task) is 1.6 blocks (median: 1). Each sub-task only contained an average of 2.0 blocks (median: 2) and 1.0 statement blocks (median: 1). This level of decomposition is quite extreme, and demonstrates that the system we propose is not optimized for such small programs where decomposition is arguably unnecessary. Each participant defined between 1 and 4 sub-tasks, with all successful participants using 3 or 4. The distribution of sub-task usage is detailed in the following bar chart.

As visualized on the right side of Figure 4.6a, the vast majority of participants did not re-use code. However, although this task was short, it did have potential for re-using a sub-task, which two participants successfully utilized.

For Task 2, participants used 60.3 blocks on average (median: 63), of which 28.1 (median: 31) were statements, and defined an average of 3.7 sub-tasks (median: 4). Similar to Task 1, the average program length is therefore slightly higher than for the other cohort, although the average number of statements is slightly lower. The average number of sequential statements

---

5When we use the term “statements” here, we include sub-task calls and robot instructions, but not locations or headers.
per code chunk (main program or sub-task) is 6.1 blocks (median: 6). Each participant defined between 0 and 4 sub-tasks, with all successful ones using 3 or more. The full distribution is detailed in the following bar chart:

Overall, we saw a very high consistency in how participants solved this task. Almost all participants split their program into four about equally-sized sub-tasks that contained the code for a single workstation. However, although two of the workstations had identical instructions, only 6 participants (14%) re-used code between them. Furthermore, all but 5 participants (12%) edited the default names for their functions, making use of this opportunity to document their code.

For **Task 3**, participants averaged 59.9 blocks (median: 65), of which an average of 31.2 blocks (median: 31) were statements, and defined an average of 12.3 functions (median: 14). As for the previous tasks, the total program size is larger than for the TR cohort. Each participant defined between 0 and 18 sub-tasks, with none of the successful participants using fewer than 10. The distribution is detailed in the following bar chart:

As visualized in the middle of Figure 4.6c, participants needed many functions to solve this task. Nine participants (22%) solved the task successfully with 12 sub-tasks or fewer, 22 (55%) with 13-15 sub-tasks, and 2 (5%) used more than 15 sub-tasks. As a result, the average number of sequential statements per code chunk (main program or sub-task) is only 3.6 on average (median: 3), which is much lower than for the TR cohort. A total of 28 participants (70%) used names other than the default names for their sub-tasks, again indicating an interest in documenting code in this way.

Despite the restrictions on how programs can be decomposed in the guided environment, participants’ programs showed some variation in solving this task. All of them were structured around workstations, but some participants followed a more stringent order of operations while others attempted to optimize their solutions. For example, approximately two thirds of the programs written by successful participants contain redundant movements between workstations that are likely explained by them trying to keep the order of operations executed by the robot as uniform as possible. However, only 7 (18%) of the participants re-used sub-tasks at all, despite the potential for re-use enabled by this uniformity.

**RQ3 Summary:** Users with guidance employ substantially more program decomposition but still only rarely re-use code.
RQ4: Impact of Decomposition Guidance on Success and Time

The effect of the decomposition guidance provided to cohort GD on success can perhaps best be seen by re-examing the bar charts we produced for RQ2 and comparing them with the charts for RQ3. Participants in the GD cohort had much more success, as indicated by the almost entirely green charts in Section 4.3.3. Figure 4.7 shows a more direct comparison of the two cohorts; the left side of this figure details success rates and the right side details task times.

For the first task, which was relatively simple, both cohorts had high success rates (92% and 95%, shown on left of Figure 4.7). However, for Task 2, the TR cohort’s performance dropped significantly (71%) and the GD cohort’s performance remained high (91%). Though Task 3 proved to be the most difficult task for both cohorts, 83% of the participants in the GD cohort still managed to solve it successfully, versus 54% for the TR cohort. We performed a chi-squared test of independence to analyze the statistical significance of the relation between participants’ cohort and their success on each task. For Task 1, we did not find the relation to be significant: $\chi^2_{\text{Task1}}(1, N = 92) = 0.46, p = .496$. For Task 2 and Task 3, we did find the relation to be significant: $\chi^2_{\text{Task2}}(1, N = 91) = 5.64, p = .018$; $\chi^2_{\text{Task3}}(1, N = 86) = 7.72, p = .005$.

To ensure that the slight difference in tutorial presentation between cohorts did not lead to some participant group having more time or experience with the system, we also analyzed how long participants spent working on both the tutorials and tasks. As shown on the right of Figure 4.7, there were no practical differences between tutorial times except on Task 3. To investigate this difference, we performed an independent two-tailed t-test, but we did not find a statistically significant difference: $t_{\text{Tut1}}(90) = -0.65, p = .517$; $t_{\text{Tut2}}(90) = 1.40, p = .165$; $t_{\text{Tut3}}(90) = -1.20, p = .233$.

For times spent on tasks, the differences we observed showed a similar trend as the success rates. For the first task, the average time of 4.7 minutes (median: 4 min.) was similar to that of the TR cohort (average: 4.8 min., median: 4 min.). However, for Task 2, participants spent 9.2 minutes on average (median: 8 min.), which is less time than the average of the TR cohort (average: 10.8 min., median: 10 min.). For Task 3, the difference between the cohorts was even more pronounced, as participants only spent 12.4 minutes on this task on average (median: 12.5 min.) compared to the TR cohort’s 14.2 minutes (median: 15.5 min.).
**Figure 4.7:** User study results. For box plots: center lines show the medians, box limits indicate the 25th and 75th percentiles. Whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. Outliers are represented by dots, only outliers < 20 minutes are shown.
An independent two-tailed t-test further found a statistically significant relation between the cohorts and task times for Task 3, but not for Task 1 and Task 2: $t_{\text{Task1}}(90) = -0.151, p = .880$; $t_{\text{Task2}}(89) = -1.89, p = .062$; $t_{\text{Task3}}(84) = -2.57, p = .006$.

**RQ4 Summary:** Users of the guided environment were significantly more successful when solving larger tasks.

### 4.3.4 Limitations

Here we discuss some of the limitations of our study.

**Participant Population:** We recruited participants via the AMT platform. We did not collect detailed demographic information from participants beyond screening them on programming experience. Previous work suggests that although the pool of AMT workers is not entirely representative of the general population of the US, potential biases are comparable to those of other recruitment methods (e.g. recruiting students) [126]. Although we do not have clear evidence that AMT participants are an acceptable proxy for the overall end-user demographic we aimed to investigate, we believe that their results provide initial evidence we hope to be investigated further through future, more in-depth studies.

**Recruiting Process:** We recruited the 92 participants of our study in two waves, 55 and 37 participants respectively. We planned to have 50-100 participants in total, but we began evaluating the results of our first 55 participants when making the decision to add a second wave of participants, introducing a risk of bias. We found little difference in all aspects of the two waves, but out of an abundance of caution we separated the two waves in the supplemental data to ensure transparency.

**Tasks and Training:** We aimed to train participants in a way that is time-efficient and comparable to the quality of training that end-users might receive in real-life. In addition, we designed the training methods to be as similar as possible across both of our participant cohorts. We assume that participants would perform better across all our tasks with more extensive training, but the high success rate on Task 1 suggests that our training method was sufficient for teaching participants the foundations of programming mobile robot workers. Another potential limitation is that the small number of tasks we used in our study cannot fully represent the wide range of possible programming tasks that end-users might encounter in industrial practice, even within the domain of programming a mobile robot worker. These tasks and their wording might not match real industrial practice, and real applications might provide more detailed or precise instructions.
that make it easier to determine a program’s optimal structure. Our study further does not investigate how large or complex end-user programs can become, and whether our decomposition approach can scale further or if it is only applicable to a limited range of program sizes. Beyond the scope we have targeted, there are also further scenarios in which mobile robots can be used (e.g., where the robot conducts work while moving instead of stopping at dedicated workstations). These scenarios are not compatible with the language and decomposition strategy that we have presented here, and it is not certain that our work or findings are transferable to them.

Experimental Measures: For our experiment, we evaluated our proposed system as a whole, which consists of several components that we described in Section 4.2. We discussed the potential benefits of each component, but our experiment cannot validate to which degree each component is responsible for the differences that we observed between the two cohorts. In addition, we compared our proposed system to one that supports parameterless functions in a style that is conceptually similar to ours, but not necessarily representative of all abstraction and decomposition mechanisms offered by block-based systems. Other block-based systems do offer features like function parameters, and the lack of those features might have held back participants of the TR cohort as they wrote their programs.

Furthermore, other factors are other potentially when evaluating the quality of a real program than those chosen by us for this experiment. For example, run-time performance might be relevant in practice, and it might be a goal for real programmers to minimize the time a robot spends moving between stations. Since such factors are highly dependent on the given situation and robot model, we decided not to instruct our participants to care for them, and therefore also do not include them in our evaluation. For our qualitative evaluation, we focused on the number of functions used, the overall program structure and code re-use. We also reported on other metrics, like the number of overall blocks or statements in a program. We believe that the performance differences we observed, despite the almost identical program size between cohorts, shows that block numbers alone are not a sufficient indicator for how easily an end-user can understand a given program.

4.4 Discussion and Summary

In this section, we discuss the implications of our approach and the findings from Section 4.3. As in Section 4.2, we use terminology from the frame-
4.4.1 Benefits of Program Decomposition

Our findings, both for RQ2 and RQ4, confirm that program decomposition can substantially improve the performance of end-users as they solve moderately large programming tasks. For the TR cohort, we found that users that decomposed their programs were more successful in solving tasks. This could imply that decomposition helped them program more effectively, but also leaves room for the interpretation that users who performed better at programming also found it easier to decompose their programs. Our findings for the GD cohort provide a clearer insight into this connection between decomposition and programming performance. By giving our participants an environment that required them to decompose their programs and guided them towards a reasonable domain-specific decomposition, we saw their success rates improve substantially.

It remains an open question how exactly program decomposition benefits beginners, and whether advice given to professionals on how to ideally structure a code base [116] also applies to end-users. However, a closer look at some example programs suggests that any structure at all, even if it was not necessarily the most intuitive or most concise, already has positive effects on a programs comprehensibility. For example, Figure 4.8 shows two programs created by participants during our user study (Task 2). Both of these programs correctly solve the task, yet they highlight the differences in program comprehensibility. Figure 4.8a shows the solution of a participant in the TR cohort: an unstructured program with 57 individual blocks. Programs similar to this one were written by many of the TR cohort’s participants across all three tasks. In stark contrast, Figure 4.8b shows a solution from a participant in the GD cohort. The main program body is only 4 lines long, and each task’s name summarizes (at least superficially) what is happening inside the task’s body. Therefore, even though the total length of the program is slightly longer at 61 blocks, and even though the bodies of “Move Stack” and “Move Stack2” are identical and could have been replaced by a single, re-used task, it is still easier to understand its structure and relate it to the given task.

Notably, block-based programs already provide certain Visibility benefits to users compared to text-based languages. For example, even in an unstructured program like in Figure 4.8a, the different block colours can help users distinguish the individual commands and identify the points at
which the robot moves between workstations. However, as our experiment demonstrates, this visual aid alone is not a substantial enough measure to highlight a program’s structure. We believe that beyond guiding users to use an at least rudimentary level of abstraction, another important benefit of decomposition is the ability to name program components. For example, readers of the program in Figure 4.8a would need to perform the Hard Mental Operation of chunking and summarizing the program’s parts manually. In Figure 4.8b on the other hand, the user was able to describe that each task moves a stack of boxes. Previous work has argued for the importance of Secondary Notation like names and comments, especially for novices [98]. However, especially block-based languages often lack places where users can use Secondary Notation or introduce barriers that make them difficult to access for beginners.

Figure 4.9 shows another example of the same pattern, but for Task 3. As described in Section 4.3.2, this task was intentionally chosen to contain a lot of steps that move the robot between workstations, resulting in a
Figure 4.9: Examples of programs written by participants for Task 3. (a) shows a subset of an unstructured program (58 blocks in total); (b) shows the top layer of a hierarchically structured program (61 blocks in total).

very fine-grained task separation when following our guided decomposition approach. Therefore, the program shown in Figure 4.9b is rather long and not structured in a way that an expert would likely consider optimal. Yet, compared to the unstructured programming style shown in Figure 4.9a that the majority of participants used for this task, the program is still more readable, and as for the previous example, the participant has used task names to summarize their code. We believe that this example illustrates how even a (from an expert perspective) sub-optimal program structure can help end-users understand their programs better.
4.4.2 Function Usage in Block-Based Systems

Our results validate previous findings and demonstrate that end-users can quickly learn how to solve small, toy-sized tasks, such as Task 1 in our study, in a block-based environment [35, 36, 115]. However, we extended these findings showing that, when not supported by the programming environment, end-users became less successful as tasks grew larger, such as for Task 2 and Task 3. We believe this is because they either do not use functions, or when they do use functions, they struggle to use them systematically.

When considering why end-users in the TR cohort tended not to use functions, even though our study explicitly trained them to do so, a lack of exposure to these features is not a reasonable explanation. Upon further comparison of the results for Task 2 (which offered a straightforward way to structure programs) and Task 3 (where the ideal structure was less obvious), we found that users were much more likely to structure their programs in Task 2. This supports our assumption that end-users understand the need to structure their programs, and attempt to do so when they are able. However, in all but the most straightforward applications, the optimal program structure might only become apparent after a large amount of code has already been written. At this point, a novice user might have already Prematurely Committed to a specific structure (or to no particular structure) and it might be difficult for them to manually re-structure their code.

What causes these end-users to struggle with functions? There are many missteps that end-users can make when using unrestricted functions. Consider Figure 4.10, where movement and stationary instructions are entangled within a single function call. When viewing the main program, the end-user cannot predict which functions might contain a hidden robot base move, giving each function call a potential side-effect. This Visibility issue is solved naturally in our prototype, by systematically requiring all robot base movements to be defined at the top level, as shown in Figure 4.9b by the “at Station X” directive. This fixed Abstraction Gradient constrains the programs end-users could potentially write, forcing them to avoid confusing side-effects.

4.4.3 Domain-Specific Task Support Beyond Mobile Robots

This work demonstrates that a system that guides users in decomposing their programs can support end-users as their programs grow in size. However, other end-user domains, such as home automation, app development or data
Figure 4.10: Example of a program where movement and stationary instructions are entangled. It is not possible to entangle movement and stationary instructions in this way with our approach.

collection from websites, might require a different approach to structuring programs. In particular, different domains likely need different forms of environmental support to map the kinds of goals end-user programmers have to abstractions such as tasks or functions. We believe that finding tasks that are compatible with the domain-specific expectations of end-users is a challenging, but not impossible exercise. Therefore, we hope that future work in identifying domain-specific tasks, for other domains, will allow those domains to benefit from techniques and customized tool support similar to what we present in this work.

4.4.4 Code Re-use: An Open Problem

Although participants in the GD cohort of our study performed better than those of the TR cohort on many metrics, there is one result that is similar for both cohorts: a lack of code re-use. Regardless of task, only a small fraction of either cohort called any function or task more than once (9% of TR, 12% of GD). As a result, the average sizes of programs with and without decomposition were about the same, which eliminates one of the major benefits of modularizing programs.

We did not expect users to employ substantial amounts of code re-use since they were not explicitly instructed to write code that is concise. Furthermore, both environments did not allow users to parameterize their functions or tasks, limiting the potential for re-usability. However, Task 2 in par-
ticular had obvious opportunities for code re-use, as illustrated by Figure 4.8; participants had to write an identical 12-block sequence of instructions twice for two different workstations, but very few used this potential to substantially reduce the amount of code they had to write. This complete absence of re-use in most users’ programs is surprising, considering that we explicitly taught participants how to re-use code as part of the tutorial sequence they had to complete.

Because we did not anticipate this lack of re-use, we did not ask participants about their reasons for not re-using code. One of the primary benefits of supporting re-use is the reduction of the language’s *Viscosity*, as each repetition requires additional work if code changes become necessary. We speculate that participants might not have seen a benefit in putting effort into re-using the same function or task, and for this limited study they might have been correct in this assessment. For instance, we did not tell participants that they had to write programs that were easy to maintain or modify later. This lack of instruction might have caused them to prefer code clones, as they might have been more familiar with the idea of copy-and-paste than they were with re-use.

We did try to make code re-use easier for participants. As Figure 4.1 shows, our guided environment highlighted all calls to the same task and showed the relevant panes on the right as a visual stack. However, this interface may have been insufficient, as the visual cue is subtle and therefore easy to miss. As we did not provide further explicit explanation, it might have even confused our participants. Finding a more effective way to encourage code re-use, especially in the presence of parameters, remains an open problem for future work.
Chapter 5

Block- vs. Graph-based Nested Expressions: Enabling Environment Interactions through Trigger-Action Programming

Human workers on factory floors or in warehouses often envision using mobile robots as co-workers to which they can assign tasks that would be tedious for a human to solve [41]. The environment introduced in Chapter 4 contributes to realizing this vision. Allowing end-users to program mobile robots and assign them tasks that span multiple physical locations gives them the opportunity to automate routines and workflows that previously required a human worker.

Program size is only one of the challenges when programming mobile robots. Another one is that interaction with the robot’s environment is often more complex when it takes place across a large physical space. For example, a mobile robot may be programmed to operate one or more machines and also remotely react to commands given by human workers. To offset the cost of a robot, it might even be used for multiple purposes, switching between tasks on demand [127]. These use cases cannot be addressed through a single, linear program, even if end-users can make changes over time. Instead, the programming system must represent environmental signals and interactions and allow end-users to connect them to robot programs.

The programming system we introduced in Chapter 4 focuses solely on programming what actions a mobile robot should execute. In this chapter, we extend this system with a focus on interaction with the robot’s environment, and add support for programming when actions should execute. We introduce support for triggers, a simplified form of event-based programming found in popular end-user tools like If This Then That (IFTTT) [128]. Trig-
gers can represent a range of useful environmental stimuli, for example by allowing a robot to reacting to machine signals, user inputs or sensor data. This enables new types of programs, such as those that operate machinery in multiple time-delayed steps, or those that contain multiple tasks that are executed on demand.

Building an end-user system that supports triggers raises a number of design challenges and decisions. In theory, the design of existing digital automation systems like IFTTT can serve as a template for triggers in robot programming. Once a program is deployed, a scheduling system can check trigger conditions and execute programs without further input from the end-user. However, two factors differentiate robot programming from digital automation. First, most digital tasks that tools like IFTTT are used for (e.g., sending an email or displaying a reminder message) are conceptually atomic, and either instantaneous or easy to resolve in parallel. On the other hand, even simple robotic tasks might involve dozens of individual steps and take minutes to complete. Therefore, it is likely that situations arise where a new trigger requires handling before the previous one has been fully resolved. Furthermore, it is hard or impossible to interleave multiple robotics tasks that are assigned to the same robot (or the same machines or workstations, if multiple robots are in use). Handling such situations, for example by queuing up triggers, in a way that is understandable for end-users can be difficult. Second, using block-based programming to represent triggers introduces a new challenge that takes place before program execution. Defining a trigger involves constructing a complex logical expression, a form of programming that is different than defining sequences of imperative commands, and that we speculate to be represented poorly in existing block-based languages.

In our work, we primarily focus on the challenge of representing expressions in blocks because compared to scheduling, this challenge is more closely related to our overall topic of making block-based programming accessible to end-users. We conducted small-scale preliminary evaluations on how to schedule triggers in a way that is understandable to end-users. Eventually, we chose a design that was reasonably easy to implement in a prototype and appeared simple enough for most of our preliminary participants to be used in our follow-up evaluations. However, in these evaluations, which we conducted at a far larger and more rigorous scale, we focus entirely on investigating the current means of representing nested expressions in block-based programming languages, and propose alternative designs.

In this chapter, we present two prototype environments: one that uses a block-based design that nests expressions vertically rather than horizontally,
and one that is a hybrid between blocks and data-flow programming. We choose data-flow programming as a potential alternative to blocks because it is a mature and popular visual paradigm and assumed to be a particularly good fit for computing expressions [129]. On the other hand, data-flow programming is less effective when modeling imperative, stateful computations like those commonly found in the context of robotics. A hybrid of the two representations has the potential to combine the best properties of both.

We conducted a controlled experiment that compares two environments for mobile robot programming based on the two design options outlined above: one exclusively uses blocks while the other is a hybrid of block-based and data-flow programming. We recruited 113 end-users via the Prolific online platform to compare the two prototypes. We trained each participant to use their respective environment and asked them to complete two complex tasks that required writing both nested expressions for triggers and imperative robot code. We then evaluated their ability to comprehend isolated, more complex examples of triggers that were represented as blocks or as a graph. Finally, we had them rate their environment individually and in comparison to the alternative.

We find that participants performed better on average when using a purely block-based environment, both when writing programs and when trying to understand them. These results are supported by the participants’ ratings, which were higher for those who used blocks exclusively. Remarkably, when asked to compare the environment they used with the alternative, participants who used blocks were more interested in using a graph-based environment than those who used data-flow graphs were in switching to blocks. We believe that our observations illustrate the attraction that data-flow programming has for end-users, but also provide evidence for their practical limitations. For the domain of robotics, we find that introducing graphs as a situational alternative does not ultimately benefit end-users. For other domains, our findings raise the questions about whether blocks are underused and whether they could outperform data-flow programming.

5.1 Limitations of Block-based and Data-flow Programming

In this section we analyze and compare the strengths and limitations of block-based programming and data-flow programming.
5.1.1 Block-based Programming

There is limited evidence about which design elements of block-based languages most benefit novice users. However, one of the design goals of Scratch and Alice was to make programs match the flow of natural language text [4, 62]. Therefore, to reach this goal, programs should be readable (and executed) in an order that follows the flow of the block shapes. For example, in Figure 5.2a, a robot is instructed to execute a sequence of commands. This flow is represented by the arrangement of the blocks, allowing the program to be read top-to-bottom and left-to-right: “To move stack of blocks at Station A: pick up item from top left; place item at bottom right; ...”. This flow matches the flow of English language text and can be adapted to other natural languages with different flow by mirroring the block shapes.

Block-based languages like Scratch strive to retain some similarity between blocks and text for expressions. Scratch does not use traditional syntax like parentheses, but instead relies entirely on block shapes and does not provide any customization options for how users can format their programs. An example of this approach’s limitations can be seen in Figure 5.1, which is representative for branching in Scratch and other popular block-based languages. In this example, the visual clutter and hard-to-identify block boundaries obscure the nesting structure of the expression, and editing it can require disassembling and rebuilding the entire nested block via drag-and-drop. The main benefit that blocks offer in this case is guidance on how the different types of sub-expressions can be composed (round shapes are integers, pointy shapes are booleans). However, unlike for text-based representations, programmers cannot add line breaks or spacing to improve the readability of the overall expression. Furthermore, the closeness to natural language does little to improve the program’s readability. In particular, since English does not provide a way to indicate how grammatical clauses

![Figure 5.1: A nested expression in the block-based language Scratch: The entire expression is displayed in a single line and the nesting structure is difficult to identify.](image)
Figure 5.2: Comparison of flow in imperative block-based code and block-based trigger expressions.

are nested, the text alone leaves ambiguity that is only resolved by closely examining the block shapes.

Figure 5.2b shows an alternative way to represent nested expressions in code, which is the one we use for the purely block-based prototype environment we evaluate in this paper. In this representation, expressions have the shape that block-based languages traditionally use for statements, placing one on each line and concatenating them top-to-bottom. Compared to Scratch, where an entire expression is placed in a single line like a run-on sentence, this style resembles a nested bullet point list in natural language text. By default, all blocks are combined using the “and” operator, but explicit operators such as “any of” or “all of” can be inserted to nest expressions. Since each sub-expression is presented on its own line, with operators adding indentation to inner expressions, the program is indented similarly to statements inside control-flow blocks. To avoid deeper nesting than necessary, the representation provides an explicit drop-down selection of negated values (e.g., “running” vs. “not running”) instead of using negation blocks.

The representation we chose in Figure 5.2b aims to improve the clarity of the contained expression, but does not resolve all potential limitations of representing expressions using blocks. It still forces programmers to read and write expressions inside-out instead of top-to-bottom. Another potential issue of this representation is that it re-uses the visual shape of statements for expressions, which might cause confusion when using both in the same environment. For example, a programmer might expect to be able to attach the blocks in Figure 5.2b to the program in Figure 5.2a.
5.1.2 Data-flow Programming

Data-flow graphs do not visualize the text-based syntactic structure of a program. Instead, they introduce a notation that captures the same semantics, but with a focus on the flow of information rather than order of execution. For a nested expression, a data-flow graph represents the flow of information from sub-expressions to their surrounding expressions, and therefore also their dependencies and the order in which they must be evaluated to match direction of the flow. As a result, data-flow graphs can leave parts of the actual execution order up to the system that evaluates them. Figure 5.3 shows the data-flow graph for the expression that the block-based program in Figure 5.2b computes, with data flowing from the individual environmental signals to the trigger at the bottom of the graph. The evaluation of the nodes does not need to follow the order that Figure 5.2b implies, but can be changed and optimized as needed.

When computing a single side-effect-free expression, such as the trigger shown in Figure 5.2b, the resulting data-flow graph is always tree-shaped. This tree closely matches the program’s syntax tree for most programming languages. For example, if one were to draw the syntax tree of the block-based program in Figure 5.2b it would be structurally almost identical to Figure 5.3. In fact, the only difference for this example is the presence of explicit and-operator nodes in Figure 5.3, which we added for clarity, but could be omitted.

The closeness of data-flow graphs and syntax trees makes it trivial to convert between a block-based expression as shown in Figure 5.2b and its graph representation as shown in Figure 5.3. From the programmer’s perspective, graph-based representations might be beneficial for understanding a program. Specifically, data-flow graphs make the order in which expressions get evaluated and how they depend on each other explicit and can be arranged so that the resulting flow can be read from top to bottom.

Data-flow graphs can be generated for all kinds of programs and not just those that consist of a single side-effect-free expression like that in Figure 5.2b. In the general case, the graph may not be tree-shaped, since data can flow to multiple target nodes, such as environmental state or multiple output values. However, the more complex structure makes data-flow graphs less useful as a visual tool. This is especially the case if computation is stateful and the scope of the state is potentially global, such as a robot’s physical environment. For example, none of the commands in the robot programs shown in Figure 5.2a share explicit data flow, but they all affect (and are affected by) the state of the robot’s physical environment. The
block-based program represents this dependency by defining a fixed order in which the commands must be executed. Without introducing additional abstraction, such as nodes that represent the environment and edges that model interactions with it, a data-flow graph would look almost identical.

In terms of user experience, data-flow graphs have further advantages and disadvantages. One major advantage is that they resemble flow charts, a widely used notation outside of programming that many potential end-users are familiar with. Furthermore, data-flow graphs can give users the freedom to arrange and format their code as they prefer, without affecting its semantics. For example, a user can group related sub-graphs together, similar to how nodes that contribute to the same combinators are grouped in Figure 5.3. A fundamental disadvantage of graph editing compared to arranging blocks is that editing a graph requires twice the work: in addition to adding or editing nodes, users must also add or update the corresponding edges. Graphs give users the opportunity to manually arrange their code, but maintaining this arrangement also induces effort as users likely must make manual adjustments as they edit their code. As Figure 5.3 shows, a graph can also demand substantially more space than blocks, and compressing it into a small canvas can make it harder to read.

Figure 5.3: The same program as in Fig. 5.2b represented as a data-flow graph. The edges make the data-flow explicit and programmers can manually group and arrange nodes.
5.2 Design Approach: A Hybrid System

Block-based and data-flow programming both have distinct benefits for end-users and seem to be best suited for different types of programs. Although it is often possible to easily convert programs between the two paradigms, their user interfaces differ too much to seamlessly switch between them while programming. End-users further might not be able to tell which approach is preferable for writing a given program or program component. Therefore, we propose an environment that selects the program representation that provides the most benefit for the user for each part of a program.

To achieve our goal of merging the two program representations, we draw inspiration from our previous work [114] (described in Chapter 4) that has introduced a user interface that splits programs over two distinct canvases: one that displays the high-level flow of the program and one that defines the localized tasks that this flow connects. Users can navigate the program hierarchically via these two canvases, selecting one task at a time on the left that then gets displayed on the right. The previous work has focused on guiding users as they decompose programs by restricting the scope of the available syntax and variables in each canvas, and found that this approach can benefit end-users significantly when they write large programs.

We extend this work by not only supporting sequentially composed tasks, but also triggers that define when task sequences get executed. Users can now use the left canvas to select both tasks and triggers, whose definition is displayed on the right. Figure 5.4 shows the environment, with a trigger selected in the left canvas and its definition displayed as a data-flow graph on the right. When a task is selected instead of a trigger, the right canvas displays its definition as a block-based program like in Figure 5.2a.

This hybrid approach has several key benefits. First, it can provide the same guidance on program decomposition that we found to be beneficial in Chapter 4. This guidance is likely even more important for programs that contain triggers, as this new feature increases the amount of code end-users will need to write. Second, the environment can be realised by connecting two existing frameworks: The block-based programming system Blockly [34] and the graph editor MXGraph [130]. As cost plays an important role for professional tool developers, being able to rely on existing, feature-rich code bases increases the chances of a similar system being adopted commercially. Third, by keeping the two paradigms distinct, we can directly compare the hybrid to the purely block-based design we introduce in Section 5.1.1 without introducing other design differences that might affect our observations.
Figure 5.4: Hybrid programming environment: the left canvas shows high-level program flow as blocks, the right canvas shows selected tasks as blocks and triggers as data-flow graphs.
5.3 Experimental Evaluation

To evaluate the effect of block-based and hybrid editing on end-user programmers, we conducted a controlled experiment. We discuss our experimental design and procedure in this section.

5.3.1 Prototype Environments

We evaluated two prototype environments in the experiment: one purely block-based and one that combines blocks and data-flow graphs. In Section 5.1, we argue the blocks can represent the computation of nested expressions, but that this representation comes with caveats. We introduced a hybrid environment that aims to overcome the limitations of blocks by integrating them with data-flow graphs to represent the computation of triggers. From the experiment, we hoped to gain further empirical insights into the learnability and usability trade-offs between the block-based and the hybrid environment.

To keep the two programming environments under evaluation as similar as possible, they provide identical syntax and editing functionality for programming imperative robot tasks, and an almost identical syntax for defining triggers. We further provided both participant groups with an identical simulator that shows a simplified top-down view of a mobile robot and its environment. This simulator allowed them to quickly execute their programs and test their correctness within the same browser window that they used for writing their programs.

5.3.2 Research Questions

We set out to answer the following research questions:

**RQ1:** Can end-users create robot programs using a mix of trigger-based and task-based programming?

Through this research question, we aim to validate the overall design of the programming environment described in Section 5.2. Previous work found that a programming environment with two separate programming canvases can help end-users structure complex programs and allow them to program mobile robots with high success [101]. Trigger-based programming, the element we add to the design explored by the previous work, is a widely used paradigm in practice [131, 132]. Nonetheless, these approaches may not work in tandem and triggers may not be suitable for the domain we chose for the experiment.
RQ2: Does the representation of triggers affect participants’ performance as they read and write programs?

We also aim to compare participant performance across the two prototype environments. By answering this research question, we hope to gain additional insight into the trade-offs between the two approaches and whether a hybrid environment is effective in mitigating the trade-offs between blocks and data-flow graphs.

RQ3: How do participants perceive both representations of triggers after using one of them?

In addition to collecting empirical measurements, we also asked participants how they perceived their assigned environment. We further asked participants whether they prefer one environment over the other, and whether they have any comments about their programming experience or their preferences.

5.3.3 Experimental Design

We used the Prolific [133] online platform for the study, which pays participants a fixed compensation to participate in studies and other research activities. We had no direct interaction with the participants other than addressing rare cases of technical difficulties. Instead, participants followed a fully automated study procedure that is outlined in Figure 5.5 and that we discuss in detail below.

Recruitment We performed a power analysis [134] to estimate the number of participants necessary to reach statistically meaningful results. Setting \( \alpha = 0.05 \) and expecting an effect size of \( \omega = 0.3 \), the power analysis determined that a minimum of 88 participants are necessary to reach a power of \( 1 - \beta = 0.8 \) in a simple \( \chi^2 \)-test. Rounding conservatively, we therefore planned to recruit 100 participants.

Participants were recruited through an advertisement on Prolific that
contained the study description, an estimate of the study duration, and details about the offered compensation, which was $12.\text{£}^6$. We recruited participants worldwide and in multiple waves to avoid over-representing specific countries or population groups. We discuss potential biases that might have been introduced by the Prolific platform and the recruitment process in Section 5.3.5.

**Pre-screening** To ensure that the participants are end-users, we performed two filtering steps. First, we used a filter provided by Prolific to limit advertising to participants who did not report having computer programming skills when they created a profile on the platform. Second, we presented potential participants with a brief pre-screening survey that asked them whether they had previous experience with software development, robotics, block-based, or visual programming languages. Only participants who did not report any previous experience were allowed to participate in the study, while the remainder received $0.30\text{£}$ for their efforts.

Automated bots and inattentive participants are a known issue when using online panels and crowdworking platforms [135]. We chose the Prolific platform instead of the more commonly used AMT as the platform claims to carefully curate their participant pool and prevent automated or duplicate submissions [133]. To further ensure that participants engaged with the study and also have the technical capabilities to use the programming environment, we also included a brief introductory tutorial as part of the pre-screening process. This tutorial asked potential participants to program a robot to pick and place a block, with detailed instructions how to do so. The tutorial was designed to take less than 5 minutes to complete, did not include any triggers, and was identical for all users. The task was intended to be easy to complete, but could not be finished by randomly clicking on the screen. Those who successfully completed the first tutorial received a small compensation of $1\text{£}$ and were added to the pool of pre-screened participants.

**Randomization and Training** After pre-screened participants provided consent to participate in the study and have their data collected, we assigned each of them randomly to one of the two environmental prototypes. We then presented them with a series of two tutorials that built on top of the initial tutorial that they completed during the pre-screening process. These tutorials introduced them to more aspects of the programming environment. The second tutorial introduced each participant to writing programs that span multiple workstations and are split across multiple tasks. A third tutorial

---

6Prolific is based in the United Kingdom and calculates payments in British pounds. Participants from other countries received equivalent amounts in their local currency.
introduced them to triggers and how to program the robot to interact with machines. The third tutorial was different for both participant groups due to the different methods used to program triggers, but we designed both tutorial versions to be as similar as possible. Overall, we expected a participant to take less than 20 minutes for both tutorials, but did not enforce a cut-off time.

**Programming Tasks** To answer RQ1 and RQ2, we aimed to present participants with programming tasks that feature both triggers and robot programs that are complex enough to test the programming abilities they had acquired during their training. We also wanted to test both the practical usability of blocks and data-flow graphs, and how they support the participants when they need to reason about the activation of triggers. For this reason, we used tasks that were larger and more complex than those used in previous studies [7, 101, 114]. As larger tasks took participants longer to work on each of them, we had to limit their number to ensure that participants would not get exhausted or drop out of the study. We eventually settled on two tasks in a fixed order that covered the concepts we set out to investigate.

**Task 1** is large but conceptually simple. Two buttons are used to control a robot and make it pick up items from two different workstations. In both cases, the items should be carried to a machine that processes them, and after the machine finishes, the robot should pick up the items and transport them to a target destination. For this task, we decided to give straightforward natural language instructions for the required logic of the triggers, placing the focus on execution rather than creativity or abstract logical thinking.

**Task 2** is more complex than Task 1, as it requires one robot to simultaneously manage two item processing pipelines. We provided participants with a program for managing one of the two pipelines. Although it seems at first that duplicating this given code is all that would be necessary to solve the task, the resulting programs for each pipeline interfere and cause the robot to inadvertently place items from one pipeline onto the other. This interference can only be avoided by adding more logic to the triggers of the second pipeline. This task requires fewer programming steps than Task 1 due to the given partial solution, but resolving the interference (which is not explained in the task description) demands a more thorough understanding of the trigger semantics.

After we conducted several pilot experiments, we expected participants to be able to solve the two tasks in approximately 20 minutes per task. Some participants took longer than 20 minutes, but few of the successful partici-
pants exceeded 30 minutes. We therefore decided on 30 minutes as a cut-off time, at which point we would save the participant’s work and redirect them to the next part of the study. Within this cut-off time, participants had an unlimited number of attempts to test their solution using the simulator we provided them. By inspecting the state of the simulated environment, we were able to automatically determine whether a solution was correct, and forward the participant to the next task.

**Usability Survey** After participants finished working on both tasks, either successfully or by exceeding the time limit, we redirected them to a usability survey. To answer RQ3, we asked participants separately about their opinion on the learnability of the environment they used, and how easy it was to read and write programs. We provided participants with a set of 5-point Likert items for their rating. RQ3 focuses on how participants perceived triggers, but we asked participants to rate both the definition of triggers and construction of robot programs separately. We hoped that doing so would encourage participants to focus on only one of these two aspects of the environment for each question. We also gave participants the opportunity to explain their opinion or provide additional feedback using a free text question.

**Program Comprehension Questions** The triggers that we asked participants to write during the two programming tasks were reasonably complex given the scope of the study, but real triggers might grow even larger. Due to the limited time we could give the participants to solve each task, we had to limit the number of items, workstations, and corresponding signals that we could ask them to consider. The code that participants had to write consisted of two to four signals and at most two operators in a single trigger, which is less than shown in Figure 5.2b and 5.3. Furthermore, only one of the two tasks required participants to read existing triggers.

To evaluate whether our findings, especially those for RQ2, hold for more complex triggers, we added two program comprehension tasks to the experiment. One task tested whether users could comprehend a trigger similar to the examples in Figure 5.2b and 5.3. Figure 5.6 shows the block-based version of the other task, which focused on whether users could understand a trigger with more nesting levels. We asked participants to select the correct natural language description of each program’s behaviour out of four options in randomized order. Both of these triggers use signals and operators with which the participants are familiar, but do not combine them in a way that relates to any of the previously seen tasks or to each other. We intentionally chose this design so that participants would need to read the given code carefully without having to learn new concepts beyond those tested by the
**Figure 5.6:** A more complex trigger with nested operators, used to test participants’ program comprehension.

programming tasks.

**Comparative Survey** As the last step in the experiment, we investigated how participants perceived the design of the environment to which they were *not* assigned. To answer this question, which is related to RQ3, we showed participants the same programs that they had just seen during the program comprehension component of the experiment, as well as the correct natural language description of the program. We then asked participants to rate this environment on three 5-point Likert items in comparison to the one they had seen up to this point. Like in the usability survey, we distinguished between the perceived learnability of the design and the ease of reading and writing programs. We also gave participants a final opportunity to provide written comments about their ratings.

The comparative survey is an addition to the experiment that required little extra participant time as we re-used the same programs as for the previous component of the experiment. This comes with two important caveats that should be considered when interpreting the results. First, participants have not had a chance to actually use the alternative design and judge its usability beyond a first impression. Second, we expect participants to be biased as they have seen the other design first and are more familiar with it. However, both of these caveats apply equally to both experimental groups. Therefore, we compared responses between groups rather than relying on absolute rating numbers.
5.3.4 Experimental Results

Following the study procedure outlined in Figure 5.5, a total of 179 participants successfully completed the pre-screening and 163 of them agreed to participate in the randomized experiment. Of those participants, 49 decided to withdraw from the study before completing the additional programming tutorials. Unfortunately, we have no insight into why they decided not to participate further. It might have been because they struggled to follow the tutorials, or because the study required too much effort for the compensation we provided. Participants who did complete the tutorials took 28.8 minutes on average (SD=10.2) to do so, with is slightly longer than anticipated. Upon closer inspection, the longer completion time can be attributed to the third tutorial, which introduced triggers. It took participants 16.2 minutes to complete on average (SD: 6.0) while the first two tutorials stayed within their expected duration. The first tutorial took 5.1 minutes on average (SD=2.1) and the second tutorial took 7.6 minutes on average (SD=4.1). One participant was excluded for neither attempting any programming tasks nor providing any survey responses.

In the remainder of this Section, we present the data for the 113 participants who completed at least the programming component of the experiment. On average, these participants were 28.6 years old (SD=9.6); 44 (39%) identified as female and 68 (60%) identified as male. One participant opted not to disclose their gender. Participants listed 23 distinct countries of residence, and 78 (69%) identified themselves as at least part-time employed.

When we refer to the evaluated groups, we will call them Graph and Blocks for brevity, as the only difference between them was the programming paradigm used to define triggers. For all statistical tests, we use $p < 0.05$ as significance threshold.

RQ1: Overall Effectiveness of Trigger-based and Task-based Programming

For each participant in the experiment, we recorded whether they finished each programming task and how long it took them if so. For Task 1, 68 out of 113 participants (60%) were successful, and for Task 2 50 participants (44%) were successful.

We further collected the participants’ final programs as well as intermediate programs whenever they decided to run tests. The number of test runs ranges widely between participants for both tasks. On average, participants ran their programs 7.04 times (SD=5.83) for Task 1 and 8.43 times (SD=6.86) for Task 2, which includes the final, successful run if there was
Figure 5.7: Completion probability over time for the two programming tasks. Shaded areas are 95% confidence intervals.

one. The median of runs is substantially lower at 4 runs for Task 1 and 6 runs for Task 2, illustrating that most participants only ran their code a few times while a few used up to 34 attempts. We manually examined successful and unsuccessful participants’ testing behaviour individually but did not find any notable differences.

Finally, we also measured participants’ ability to comprehend two more complex examples of triggers. We found that 84 (76%) of the 110 participants who completed the program comprehension tasks correctly answered the first comprehension question and 78 (69%) correctly answered the second question. Most participants who correctly answered one of the two questions also answered the other one correctly; 68 participants (62%) correctly answered both questions.

**RQ1 Summary:** Overall, 60% of participants successfully solved the first and 44% solved the second task.

**RQ2: Performance Differences Between the Environments**

In addition to analyzing the overall participant performance, we also analyzed the individual groups and compared their results. Of the 59 Blocks
participants, 42 (71%) solved Task 1 and 30 (51%) solved Task 2. Of the 54 Graph participants, 29 (54%) solved Task 1 and 20 (37%) did the same for Task 2.

To analyze both the completion times and success rates of the participants under the given time limit, we performed survival analysis for both groups and tasks [136]. The survival function \( S(t) \) models the probability of a participants’ survival over a given period of time. In the context of the study, a participant “survives” as long as they have not yet successfully finished a task. Since we care for the probability of participants having completed the task after a given time frame, we plot \( 1 - S(t) \) in Figure 5.7.

We fit Kaplan–Meier estimators [136] on the success rates and times of the participants. Figure 5.7 shows the resulting 95% confidence intervals for the completion probabilities. We also performed a log-rank test [137] for each task to determine if the difference between the two curves is statistically significant. We found \( \chi^2_{\text{Task1}}(\text{df}=1; \text{N}=113) = 4.92, p = 0.03 \) and \( \chi^2_{\text{Task2}}(\text{df}=1; \text{N}=113) = 3.08, p = 0.08 \). Therefore, only the difference for Task 1 is statistically significant.

For both tasks, we found only minor differences in the testing behaviour of the two groups: Participants in the Blocks group ran their programs 6.93 times (median=4, SD=6.34) for Task 1 and 8.30 times (median=5, SD=7.46) for Task 2, compared Graph participants executing 7.15 runs (median=5, SD=5.45) for Task 1 and 8.61 runs (median=6, SD=6.44) for Task 2.

For the comprehension questions, we also found that participants in the Blocks group performed better than those in the Graph group. Participants in Blocks understood the trigger shown to them correctly in 50 (85%) and 46 (78%) out of 58 cases for the two respective tasks. The Graph users understood the trigger in 34 (65%) and 32 (62%) out of 52 cases for the two respective tasks. We find \( \chi^2_{\text{Comp1}}(\text{df}=1; \text{N}=110) = 6.59, p = 0.01 \) and \( \chi^2_{\text{Comp2}}(\text{df}=1; \text{N}=110) = 4.20, p = 0.04 \), and therefore a statistically significant difference for both tasks.

**RQ2 Summary:** Participants in Blocks were more successful at solving Task 1 as well as the program comprehension questions than those in Graph. We did not see a statistically significant difference between the groups for Task 2.

**RQ3: Participants’ Perception of the Environments**

In addition to measuring participant performance, we also asked them to rate the environment they used regarding its learnability and the readability and writability of programs. We did so with two sets of questions to separate the trigger editor from the surrounding programming environment for robot
tasks that was identical for both groups. We provided participants with three 5-point Likert items, which we code as 1 for a strongly negative rating and 5 for a strongly positive rating. The resulting matrix of responses is shown in Figure 5.8.

Participants from both groups used the same editing interface for robot tasks and gave it highly similar ratings as expected. The average ratings for the task editor are 3.75 for learnability (Blocks: 3.72, Graph: 3.77), 3.76 for readability (Blocks: 3.74, Graph: 3.77) and 3.54 for writability (Blocks: 3.55, Graph: 3.54). For the trigger component of the used environment, we find larger differences. The average ratings here are 3.17 for learnability (Blocks: 3.41, Graph: 2.92), 3.22 for readability (Blocks: 3.45, Graph: 2.96) and 3.04 for writability (Blocks: 3.34, Graph: 2.73). These ratings suggest that the participants found block-based triggers more usable overall,
with a particularly strong preference when rating how easy it was to write programs.

The collected usability ratings should not be interpreted on an absolute scale as users rate usability with a substantial bias towards higher ratings [138]. We did not use a standardized questionnaire, and therefore only use the usability ratings to compare the two participant groups to each other. This type of comparison can also be affected by statistical biases, but the effects are typically small [139].

We performed a one-way MANOVA across all six Likert items. This analysis finds that $F(df=6, 103) = 2.34, p = 0.04$, indicating a statistically significant difference between the groups. Individual significance tests for the task items find: $F_{\text{TaskLearn}}(df=1, 108) = 0.06, p = 0.81, F_{\text{TaskRead}}(df=1, 108) = 0.02, p = 0.88$, and $F_{\text{TaskWrite}}(df=1, 108) = 0.01, p = 0.95$. For the trigger items, they find $F_{\text{TrigLearn}}(df=1, 108) = 5.58, p = 0.02, F_{\text{TrigRead}}(df=1, 108) = 5.00, p = 0.03$, and $F_{\text{TrigWrite}}(df=1, 108) = 8.13, p < 0.01$. Therefore, the differences between all three ratings of the trigger component are statistically significant, and those for the task items are not.

In addition to asking participants to rate their assigned environment, we also showed them triggers as they would be represented in the alternative environment. We asked them to indicate which of the representations they prefer, using the same three items (learnability, readability, and writability) as for the previous survey. We code responses on a scale from -2 (strong preference for the other environment) to 2 (strong preference for assigned environment).

The participants in the Graph group gave average ratings of 0.19 for learnability, 0.09 for readability, and 0.19 for writability, indicating a weak preference for their own design. For the Blocks group on the other hand, participants gave average ratings of -0.41 for learnability, -0.46 for readability, and -0.45 for writability, indicating a preference for the representation used by the Graph group. A one-way MANOVA across the three items finds $F(3,106) = 3.49, p = 0.02$, and individual tests find $F_{\text{CompLearn}}(1, 108) = 9.05, p < 0.01, F_{\text{CompLearn}}(1, 108) = 5.95, p = 0.02, and F_{\text{CompLearn}}(1, 108) = 9.61, p < 0.01$ for the comparative ratings. Therefore, all differences are statistically significant.

**RQ3 Summary:** Participants in Blocks gave significantly higher usability ratings for their assigned trigger editor than those in Graph. However, both groups expressed a preference for the graph-based editor in direct comparison.
5.3.5 Limitations

Here we discuss limitations of the experiment.

**Participant Recruitment:** We recruited participants using the Prolific online platform as it made it feasible to pick a large sample of end-users without being limited to a single geographical location or employer. As described in Section 5.3, we applied a variety of measures to reduce the risks of using a crowdworking platform. We also provide a range of demographic details about the participants in Section 5.3.4 that lead us to believe that we captured a wide range of participant backgrounds. Nonetheless, we are aware that the sample might not fully capture the large variety of potential end-users who might be asked to write robot programs in practice.

**Attrition:** As stated in Section 5.3.4, a substantial number of users decided to withdraw from the experiment before completing the programming tutorials. These participants have been fairly evenly distributed between both participant groups (24 in Blocks, 27 in Graph). However, we have no information about why these participants withdrew, and our inability to analyze their data might introduce bias into the experiment.

**Task Selection:** To ensure that we could ask the participants to complete reasonably complex programming tasks, we had to limit the number of tasks we asked them to solve. We could have selected a larger pool of tasks and randomly assigned them to participants, but this would have required splitting the participants into more groups and limited the statistical power of our results. As a result, we had to select only two programming and two comprehension tasks to represent a wider range of practical scenarios. We justified our selection in Section 5.3, but the external validity of our results is limited without further validation on different tasks.

**Training and Time Limits:** Due to the limited time available during the study, we had to restrict both the training we could provide to the participants and the time that they were allowed to work on each task. Participants might have performed better overall if they had received more extensive training, and more participants might have completed the tasks if given more time. Similarly, participants might have been able to provide us with more detailed feedback and been able to provide a more distinct feedback on aspects of the system. For example, we see in Figure 5.8 that participants showed little distinction between learnability, readability and writability in their responses. With more time available to allow them to become familiar with the systems, and with room for more tasks that focus on one of the three aspects, we might have received more meaningful responses. However, these limitations affected both participant groups to a
similar extent, leading us to believe that they would not have significantly altered the outcome of the study.

5.4 Discussion and Summary

In this section, we discuss and interpret our findings. We also include selected participant quotes from the 46 total written comments collected using the survey at the end of the experiment.

5.4.1 Success and Usability: Blocks Beat Hybrid

The results of our experiment confirm the findings of previous studies [36, 114] that an end-user friendly programming environment allows end-users to program mobile robots with little training. Compared to those studies, we investigated substantially more complex tasks that involved writing robot programs as well as logic that determines when the program gets executed. Therefore, it is unsurprising that even the better-performing of the two participant groups had a lower success rate than participants in previous studies. Examining Figure 5.7, it is likely that a more generous time limit might have allowed more participants to finish the tasks. Nonetheless, we consider it remarkable that more than half of the participants could solve the tasks despite the necessary constraints.

Our findings also support our assumption that the trade-offs we presented in Section 5.1 are relevant for end-users and affect how they write code. We found that participants were faster and more successful writing programs in an environment that uses blocks as its only representation for code. This might be unsurprising, considering the additional effort required to learn two distinct editing styles. Some participants echoed this sentiment, writing about block-based trigger editing that “it’s more similar to programming tasks,” and about graph-based editing that “[t]he lines and structure are relatively complex.” However, few comments mention issues regarding the different editing experiences between the two styles, with the exception of a few rare comments like “I couldn’t find how I could change the lines, so I ended up scrapping it and starting over.” Instead, several participants commented on data-flow graphs being difficult to read in general, criticizing “random arrows [that] had two separate ends and were very confusing,” and that “it appears like there is a hierarchy even when there is not.”

Participants not only found it harder to write programs using data-flow graphs, but also struggled to read and understand them. This is not just
supported by comments like the previously mentioned ones, but also by the results of the program comprehension tasks. For both tasks, users that were presented with a data-flow graph performed significantly worse at identifying what the given trigger was supposed to do. Notably, for the comprehension tasks we showed users the trigger code in isolation. Therefore, we speculate that data-flow graphs might be harder to read for end-users even outside of the context of a hybrid programming environment.

5.4.2 (Why) Do End-users Prefer Graphs?

One finding of the survey does not fit with the other results: in a direct comparison, end-users prefer data-flow graphs over blocks. One reason for this observation could be that data-flow graphs look more interesting or visually appealing. All users had exposure to blocks, but those users who used them exclusively did never see data-flow graphs before the point where they had to rate them. They might have perceived them as a more elaborate visualization and the overall system as more complex or mature. Therefore, novelty alone (from the perspective of the user) might be a reason why our participants showed a preference for graphs.

Another potential reason for participants’ preference could be that we only presented users with data-flow graphs such as the one shown in Figure 5.3, which we manually arranged to be easy to read for the given task. Compared to the block-based equivalent program, some users wrote that “[i]t/the other one gets more clean and organized.” and that it “looks much better and more intuitive than what I used.” We speculate that users compared our arrangement to the block-based program in Figure 5.2b] which leaves no room for visual customization or aids. Some users also explicitly stated that the block-based code “looks very messy” and raised concerns that it “can get very spaghetti-y, very fast.” While some participants stated about graphs that “in larger scale this looks better to program”, we have discussed the higher demand for space as a potential downside of graphs when it comes to larger programs.

5.4.3 Implications for Other Domains

Both block and data-flow programming have been adapted to a variety of end-user programming domains. However, to our knowledge there exist no hybrid systems like the one we presented here. Our findings can be interpreted as justifying this lack of hybrid systems. It appears that even minor adjustments to the visual presentation of block-based languages such as
those we performed in our entirely block-based system can achieve superior results despite theoretical limitations and caveats.

Remarkably, our results also provide initial evidence that end-users are better at understanding blocks than data-flow graphs even when each is used in isolation. We observed this in our program comprehension tasks, and the participant comments support this finding. This seems to contradict the more widespread commercial adoption of data-flow graphs compared to blocks. Although blocks are used in select commercial robotics applications [32], data-flow graphs have found wide-spread use across several other domains, particularly in game development [140,141] and Internet-of-Things programming [142]. Our findings raise the question of whether exploring block-based approaches might bring benefits for end-users in these domains as well.
Chapter 6

Towards End-user-centric Designs for Domain-specific Programming Languages

In Chapters 3, 4 and 5, we present tools that expand the range of robotics applications that end-users can develop. All of the presented prototypes are founded in block-based programming and the underlying design principles and features we present in Chapter 2. However, each prototype extends beyond block-based programming alone and integrates concepts from visual programming into its design. In summary, the concepts used by each prototype are:

Chapter 3: The side-by-side presentation of two programs, their alignment to represent parallelism, and the use of synchronized commands that connect the two programs. In contrast to traditional block-based code that resembles text with a linear flow from top to bottom, this prototype adds a second dimension that represents the assignment of commands to a specific robot arm.

Chapter 4: The decomposition of a large robot program into multiple spatially localized tasks that can be navigated using an overview-detail editor. In contrast to traditional block-based environments that provide users with free-form functional decomposition and a full set of syntactic options at all times, this prototype guides users to ensure that their programs follow a specific, domain-appropriate structure.

Chapter 5: The representation of nested trigger expressions as data-flow graphs in combination with blocks for sequential commands. In contrast to traditional block-based systems that follow a single visual modality and represent all code using it, this prototype aims to maximize the benefits of multiple visual presentations for different elements of a program.
In this chapter, we reflect on the design approach we have taken to develop the prototypes we present in this work. We discuss the lessons we learned about empirically designing languages and tools that target end-users. We further discuss some of the limitations and caveats of this work, and how we can generalize the design elements and techniques we use to other settings within the domain of robotics, and to other domains where end-users program.

6.1 Empirical Programming Language Design for End-users

Designers of programming tools and languages often prioritize novel features in their work that make their systems more expressive and provide users with useful information about their programs. Factors such as how easy it is to learn and use a new system are often secondary, and are evaluated from the perspective of the authors, who are inherently familiar with their own designs. There exist a number of historical studies of how end-users and professional developers use programming tools. These studies have established general guidelines for designers. For example, that it is important to provide users with abstraction and encapsulation layers, and that it is easier to read instructions (in user interfaces or as programming language keywords) in natural language than in abstract notation. However, as programming languages, their features, and the ways in which they are used evolve over time, it becomes less clear if and how these guidelines apply to modern systems. Unfortunately, it is rare for studies to be replicated to see if they stand the test of time. Instead of informing design, modern empirical work is often only used to demonstrate the benefits of languages and tools after the design process has been completed, and its scope and rigor may be too limited to inform future work.

Recently, programming language researchers have intensified their efforts to incorporate methods from the field of Human-Computer Interaction (HCI) into their work. The most comprehensive and generally applicable language design process that has been presented is called the Programming Language Iterative Evaluation and Refinement System (PLIERS). This system proposes a methodology to develop programming languages iteratively and with feedback from potential users, evaluating languages not just summatively but also formatively.

PLIERS aims to be broadly useful for programming language research, but focuses on languages and tools for professional developers. In the context
of end-user-centric work, it can only be applied with caveats. For example, PLIERS assumes that potential users have previous programming experience that can provide them a frame of reference. If users have used established programming tools, they can actively compare new designs to the ones they have used previously and evaluate perceived trade-offs on their own. End-users, on the other hand, often have no previous exposure to programming and therefore need more guidance on which novel aspects of a tool they should evaluate and which ones are already established. In addition, they might also feel overwhelmed by terminology used in questions and might need more context or clearer examples to be able to provide valuable insights about specific language features.

PLIERS does mention one of the biggest challenges we observed in our work: the trade-off between creating a system that meets the needs of programmers by offering them a wide range of expressiveness, but also keeping the system simple enough to be usable. However, this trade-off is even more relevant for end-users than it is for professionals. It can be harder to find a compromise that is acceptable for end-users, since they need a lower barrier of entry than more experienced programmers, but still have expressiveness requirements that a system needs to meet. Professionals who plan to use a system over a long period of time are also more likely than end-users to actively look out for new features and be able to learn the ones they discover in a self-directed manner [147]. Addressing this challenge does not require an entirely new design approach, but additional attention to usability and short-term learnability, and might limit whether findings from studies on professional developers apply to end-user tools.

PLIERS also has limitations when applied to languages that are not text-based. In particular, PLIERS proposes the rapid development of mock-ups and prototypes to gather user feedback and compare design candidates. However, this form of iteration is far easier when tools are entirely text-based and little effort is needed to make superficial changes to designs. For example, it might be a matter of minutes to replace a keyword in a text-based programming language with one that might be easier to understand, but changing the design of a visual language can require substantial development effort. Similarly, it can be easier to provide text-based programs to users and have them understand how they could make changes to them as the modality is familiar to them. In contrast, the necessary interactions with visual tools are often harder to predict just from looking at a program or an image of a tool’s user interface.

Although we conducted most of our design work before PLIERS was published, we use similar techniques as those proposed by its authors. Our
language design process follows an iterative structure where early drafts are refined by direct end-user feedback or by analysing them through frameworks such as the 13 Cognitive Dimensions of Notations (CDN) [1]. Our work on side-by-side coordination in Chapter 3 is particularly influenced by both the CDN and by formative feedback gathered through a large end-user survey. PLIERS mentions analytical work and the CDN, but puts more emphasis on empirical studies to inform design decisions. However, based on our own work and experience, we believe that a thorough theoretical usability analysis is essential to inform the available design options and to guide empirical work. As the authors of PLIERS note themselves, user feedback is not always sound and reasonable, and we therefore believe in presenting users only with design alternatives that are consistent with the overall goals for the system.

In addition to formative studies, PLIERS outlines how user studies and experiments can evaluate new languages summatively. We have also conducted such summative evaluations in our work. In particular, we performed a controlled experiment to evaluate our environment for mobile robot programming in Chapter 4, and a similar one in our follow-up work on coordination in Section 3.4.2). PLIERS outlines some of the challenges of such evaluations, notably the difficulty of separating different language aspects to judge their impact separately. For example, in Chapter 4 we evaluated both language and environmental design choices at once, leaving it unclear which of them contributed how much to our overall results. The authors of PLIERS suggest conducting several isolated studies on features, which comes with the price of either having to recruit more participants or finding results that are statistically less meaningful. In our work, we decided to focus on a bigger-picture comparison to demonstrate the possible impact of guided decomposition for end-users, leaving a more detailed analysis of the individual design choices for future work.

Another insight that we gained through our work lies in the recruitment of study participants. PLIERS suggests recruiting senior undergraduate or graduate students as participants for studies as they have programming experience and are easily available to researchers at universities. Whether students can actually substitute for professional developers is questionable, but they are even less likely to be good replacements for end-users. This is not just because students might have more programming experience than end-users, but also because real end-users are likely more mature and have different needs [148]. In our work, we found that recruiting end-users through online platforms such as MTurk or Prolific is an effective way to find a large number of participants as long as measures are taken to counter the draw-
backs of these platforms. Bots and participants with low motivation to work on study tasks can bias the results of studies and limit the amount of useful feedback that can be gathered. However, from our experience, the use of pre-screenings and interesting, non-trivial tasks can lead to insightful results. Notably, we did recruit students (from different areas) for our follow-up work on two-armed coordination (see Chapter 3.4.2). For experiments that must take place in-person, students can be a reasonable compromise as long as they are screened for factors such as programming experience.

A challenge that the authors of PLIERS identified, but that we believe to be less significant for end-users, is the lack of time to train participants or ask them to complete complex tasks, which limits a study’s external validity. Time restrictions are a consistent limiting factor that we identified in all of our studies, but a limited amount of training and time for task completion is to be expected for end-user systems. When evaluating systems for professionals, it is often hard to predict how differently developers would interact with a system after weeks or months of active use. For end-users on the other hand, we do not typically expect them to develop the same degree of competence using a system, as programming is only a small part of their daily work. Conducting a shorter study with limited training and small tasks can therefore be a realistic simulation of their user experience.

6.2 Limitations of the Visual Design Approach

Our work introduces domain-specific visual elements into block-based programming while retaining most of the benefits of block-based systems. However, the extensions we present still come at a cost, which affects both the implementation and end-user experience. In this section, we discuss some of these limitations, while the larger question of whether our designs can generalize beyond a single domain is addressed in Sections 6.3 and 6.4.

Regarding implementation, our systems do introduce overhead, both in design and implementation, compared to creating an off-the-shelf block-based editor. There exist a number of tools to create block-based programming systems with minimal or even no programming knowledge [34,84]. Although our systems are using the same established frameworks targeted by these tools, they still require substantial implementation effort and custom solutions for design challenges like the ones we presented in each chapter. Nonetheless, our systems require a smaller amount of design and implementation effort to build than brand new visual languages. For example, the two-canvas system presented in Chapter 4 needs to coordinate multiple
block-based canvases, but does not re-implement the underlying user interfaces and workflows, such as dragging and attaching blocks. Similarly, the system we introduce in Chapter 5 combines block-based and graph-based editing, both of which rely on established editing tools that only need to be connected to create a single program across both editors.

Our designs also introduce new user experience challenges that arise independently of their implementation. Users who might have already used a traditional block-based environment might find the differences in how they interact with the system unexpected or frustrating. As we have illustrated in Chapter 2, traditional block-based systems also show a substantial range in their design, so we consider this a rather small risk. However, modern block-based tools are moving towards consolidation around a more rigid set of frameworks and design practices, which our work does not fully comply with. We therefore expect this trade-off between standardization and custom-tailored tools to become more relevant for future work.

Another aspect of user experience that our work does not yet address is accessibility. Although block-based programming may appear inherently inaccessible at first due to its focus on mouse-based and highly visual interactions, previous work has demonstrated that it can be adapted for visually impaired audiences [149, 150]. Some of these modifications, like using higher contrast colours or offering visual customization options, can be directly applied to our work as well. None of our presented designs extend the meaning of colours or block shapes beyond the way they are already present in traditional block-based systems. By reducing the amount and number of different types of blocks on screen, they might even help alleviate some of the difficulties that visually impaired users might have reading block-based programs. However, some features that were designed with blind users in mind require converting programs into text. These features are harder to replicate by our designs due to their greater emphasis on visual features and therefore reduced closeness to text-based code.

In the context of our chosen domain of robot programming, we believe that accessibility challenges are not limited to the programming interface, but also other human-robot interactions such as positioning or testing and debugging programs. Substantial additional work is needed to investigate the accessibility challenges of this domain and develop accessible design guidelines. In the scope of this thesis, we cannot address these issues, but nonetheless, we believe that our designs could be adapted for blind or visually impaired users. For example, our side-by-side design presented in Chapter 3 could be translated into a text-based grid that can be navigated with a four-way cursor and read out to the user. Previously proposed ideas such
as uniquely identifying blocks [149] could be used to indicate corresponding left- and right-armed command blocks to users. Notably, our design candidate with implicit line-by-line synchronization might be more suitable for such an interface as it is more regular in its grid-like structure. Our other designs in Chapters 4 and 5 could see similar navigation adaptions. In particular, our guided decomposition approach might be suitable for users of non-mouse-based interfaces because it reduces the number of available blocks in each canvas as well as the number of total blocks on screen.

6.3 Pushing our End-user Robotics Designs to New Use Cases

The designs that we present in our work address specific use cases for robots and are custom-tailored for their purposes. On a language level, we have typically chosen a subset of commands that is reasonable as a minimal viable prototype. Our designs might be easily adapted or extended by adding new types of blocks to support additional robot commands. However, more elaborate extensions that interact with the visual elements that we incorporated into the designs might be more challenging to realise and could lead to unexpected and severe drawbacks.

As an example of an extension we found surprisingly difficult to achieve, consider the design we introduce in Chapter 3 for coordinating two robot arms. Field studies have shown that few practical tasks require coordinating more than two robots directly [99]. Nonetheless, one might wonder whether more tracks could be added to extend our prototype beyond two-armed robots. We have created early drafts of how such a design could look, as illustrated in Figure 6.1. This example shows a program where three robot arms hand over an item from one robot arm to another and then to a third one. To support the third robot arm, we introduce a third column that blocks can be placed in, with programs still being executed top-to-bottom with explicit synchronization blocks to make the arms wait for each other.

Although the example in Figure 6.1 is conceptually similar to the two-armed program shown in Figure 3.5b, it suffers from significant usability drawbacks. First of all, the more robot arms a program gets distributed across, the more space remains unused, reducing the visibility of the code. The example is also harder to read sequentially, as it flows in two dimensions instead of just one. In this example, the program flow is still linear, from the top-left to the bottom-right, but other programs might flow back to the left or even jump arbitrarily between the columns. Although this example only
Figure 6.1: Early design draft for a 3-armed extension of the vertical design with explicit synchronization that we present in Chapter 3. It uses 3 columns to represent the individual robot arms.

shows three arms, one can imagine that any larger number of arms would further harm the program's readability.

Another challenge is finding a reasonable model of synchronization. One way to represent synchronized commands would be to introduce a global barrier that halts all programs for all arms. This form of synchronization is easy to understand, but might be unreasonably inefficient in practice. Therefore, in the example we show here, we chose a different form of synchronization where only two arms are waiting for each other at a time. This behaviour leaves room for concurrent commands that the third arm can execute. For example, if the first command “Move Arm to Second Handover Right” was a particularly complex movement that required time to perform, the arm could execute it while the other two arms perform the first handover. However, the way we represent this synchronization in blocks only works when two adjacent arms are synchronized as shown in Figure 6.1. If, for example, the left and right arm were to interact, a new block that skips the middle column would be required, which is difficult to represent.
Figure 6.2: Early design draft for an environment that features both two-armed coordination as presented in Chapter 3 and guided decomposition as presented in Chapter 4. The left canvas shows the high-level program flow, which is composed of a series of two-armed programs that are defined in the right canvas.

visually and might be misunderstood by users. Adding more arms would undoubtedly complicate such a design further.

Although our first draft as shown in Figure 6.1 is flawed, one should not conclude that the design we presented in Chapter 3 is inherently incompatible with more robot arms. To address the issues above, we have created further drafts with the goal to represent a pipeline of robot arms. These arms might be physically located in a way so that we only need to consider pairwise interactions, as shown in Figure 6.1. Such a pipeline could execute programs in parallel, with the left arm picking up a new item once it has handed the previous one over to the middle arm. Unlike the shown example, the robot arms would effectively execute a loop with the goal to optimize the throughput of executed commands (and therefore processed items). Such a design might be useful for certain real use cases of robot arms, and it might be worthwhile to investigate the representation of loops in block-based languages further. We have decided not to continue this design path in this dissertation, but consider it a promising area of future work.

Another interesting example of how our work could be generalized is to combine our work from Chapters 3 and 4. In Chapter 4, we present an environment that supports guided decomposition for mobile robot programs.
We discuss the importance of finding an appropriate form of decomposition for other programs in Section 4.4. However, coordination is one concrete example of a potential use case for guided decomposition. Figure 6.2 shows an early design draft for an environment for programming multiple robot arms using the same concepts as we presented in Chapter 4. In this design, tasks are parameterized by specifying the robot arms that are supposed to interact instead of workstations where tasks take place. Selecting a task in the left canvas shows its definition, which uses the side-by-side layout that we present in Chapter 4. By selecting different robot arms, tasks can be re-used for multiple purposes.

Compared to our previous draft shown in Figure 6.1, the design in Figure 6.2 limits all robot interactions to two arms at a time. However, it also structures programs around this limitation. This design principle is similar to the one we used for mobile robots in Chapter 4, where programs were decomposed based on space rather than coordination. In principle, this design could make programs more readable, but it might also structure them in a fragmented way if many different combinations of robot arms need to interact. Because of the implementation limitations that we discussed in Section 3.4.2, we could not pursue this design draft further as it would have required too many separate canvases and too much screen space. Nonetheless, we believe that this draft demonstrates the potential for guided decomposition in a different style than mobile robots.

Overall, the design drafts shown here demonstrate some of the possible design space for using our prototypes in the context of robot programming. In theory, one might also combine our trigger-based design prototypes shown in Chapter 5 to create a single environment that shows off all our work in one combined system. However, we consider it unlikely that a system that requires end-users to learn such a large number of features and concepts is effective in practice without adding further support. We therefore advocate for selecting the necessary features for a concrete use case carefully and opting for their careful integration over sheer quantity. For this reason, we have separated our prototypes rather than creating a single environment that aims to fit all purposes at once.

6.4 End-user Programming Beyond Robotics

Several of the visual features that we incorporated into block-based programming throughout our work are highly specific to the domain of robotics. The presentation of two programs side-by-side to match the alignment of
physical robot arms and decomposition of programs based on the physical workstations that a mobile robot operates are just two examples of these domain-specific adaptations. Although we have shown in previous chapters how each of these features benefits end-users in robotics, they also limit how our block-based designs could be used in other domains. For example, most programs outside of robotics do not consider physical space or the arrangement of hardware components. Even in cases where the layout of a system is relevant, such as in distributed computing, tool designers might intentionally hide these aspects of the system rather than incorporating them into their designs, for example by using abstractions to separate them from the rest of the program code.

We believe that some of our experimental findings have direct implications beyond specific domains. We discuss these implications in Sections 3.5.2, 4.4.3 and 5.4.3. However, we believe that there are more fundamental design features that go beyond our concrete prototypes and transfer to other domains as well. As in previous chapters, we use the terminology of the Cognitive Dimensions of Notations to discuss them here and italicize and underline CDN terms.

One of the most central concerns that of all our designs aim to address is the Visibility of programs and their semantics. The need for programs and their semantics to be visible is well-understood in the context of visual programming languages [2], but usually receives less focus in the development of block-based programming tools. Block-based programming provides some inherent visibility features, such as the colour coding of different blocks and the use of shapes to highlight connections between commands. However, a lesson we learned from our own language designs is that this support is not sufficient to make complex programs easy to read and understand. Visual programming features can make programs more visible on multiple levels, as our design for coordination illustrates. Using horizontal screen space more effectively keeps larger amounts of code visible, following the data visualization principle of “eye beats memory” [151]. Aligning and connecting the two coordinated programs also makes their interactions more visible and highlights the order in which they are executed. Our other designs follow a similar approach: they aim to maximize the amount of code that can be displayed and the ease of navigating it, but also address semantic concerns, for example by highlighting data flow in nested expressions.

As a consequence of increasing the visibility of complex parts of a program, all of our designs also reduce the Error-proneness of programs. Recall our initial example of a block-based program that is hard to read from Figure 1.2. In this program, blocks were ordered and scaled based on the
syntax of the underlying text-based program, resulting in often arbitrary connections between aligned blocks. In contrast, our designs direct the programmer’s attention to the parts of the program that are the hardest to comprehend and edit. In the example of our program decomposition design, we even decided to structure the entire programming interface and navigation around tasks and the physical space in which they take place. By making complex elements of a program explicit first-class features of the programming environment, we ensure that end-users do not neglect them or make errors based on a lack of information.

In addition to focusing the attention of end-user programmers on the hardest to understand parts of the programs they edit, all of our designs also aim to simplify these parts and make them easier to comprehend. By making properties of the programs explicit, an inherent goal of visual programming, they reduce the frequency of *Hard Mental Operations*. In most cases, we use the layout of program code to make mental operations easier, for example by arranging code in a way that matches the order of execution in both our designs for coordination and trigger evaluation. Compared to text, blocks provide the ability to do so as it is not necessary to arrange programs strictly top-to-bottom and left-to-right. However, we also go beyond layout as we support users in making decisions about their programs. Although our design for program decomposition is the most obvious example for simplifying the decision space for end-user programmers, we also use this technique more subtly in our other designs. For example, in our coordination prototype, we provide only a single mechanism to synchronize the two robot arms, compared to the low-level approaches offered by languages like Scratch and Mindstorms EV3. Similarly, in our trigger-based environment, we explicitly divide programs into tasks and triggers with distinct syntactic elements that cannot be combined arbitrarily. We believe that although this reduction of possible program semantics can limit the expressiveness of the overall system, it is often essential to support end-user friendliness.

One final aspect that is shared by our systems, but less explicit in our designs is a focus on limiting the language’s *Diffuseness*. A programming language is more diffuse the more different elements or concepts a user has to learn to understand it. Admittedly, as all our systems are prototypes, they are likely less diffuse on a syntactic level than a production system might be, as they do not need to support all potential commands that the robot hardware supports. However, our systems also follow this design principle on a different level, as they aim to re-use existing block-based concepts and features when there is no clear benefit to replacing them. For example, we considered the addition of arrows and other markers to highlight synchro-
nization points in our two-armed environment in Chapter 3, but found that aligning the blocks vertically is sufficient for most users and fits with the existing jigsaw puzzle design of blocks. Re-using design concepts not only reduces the learning effort for end-users, but also ensures that the implementation of the resulting systems is simpler and closer to that of purely block-based ones. Remarkably, the findings of our experiment in Chapter 5 support this design decision as users appear to prefer a more uniform block-based system over one that merges two different design approaches with the goal to combine their strengths.
Chapter 7

Conclusion

End-users already make up a large fraction of the software development ecosystem, and their number is likely to grow further. At the same time, only a fraction of programming tools target end-users, and there exist few established standards, or well-understood design principles, to support tool builders who create environments for end-user programmers. As a consequence, end-users often must rely on alternatives such as educational or professional tools that are not created with them in mind. This situation puts a burden both on end-users who lack support, and on tool designers who are expected to target a diverse range of potential users.

Adapting professional or educational tools for end-users is often challenging. Tool creators may underestimate this challenge due to the misconception that tools that are easy to use are also easy to design. State-of-the-art tools for professional developers are often complex because they are feature-rich, so it might seem that removing some of those features is all that is needed to make them easier to learn and use. Furthermore, tools and languages that are meant for students can appear like a blueprint for how to make programming more accessible for other audiences as well. However, both approaches commonly lead to systems that are inconsistent in their designs and do not allow end-user programmers to reach their full potential.

Without rigorous evaluation and feedback from end-users, it is not possible to make informed decisions regarding the trade-offs that arise when designing systems, but this is especially true in end-user-centric systems. Almost all designers of programming environments are experts in software development themselves, which makes it hard for them to intuitively judge the usability of their own tools. The end-user tools that see wide-spread adoption are usually those that have been tested thoroughly to truly fit the needs of their target audience. Unfortunately, successful tools are often proprietary and backed by strong commercial interests. Naturally, commercial vendors have little interest in sharing the insights gathered from their studies with potential competitors, or informing them of decisions made during development.

We believe that independent research into end-user-centric systems plays
an important role in making new designs and techniques available to a larger number of tool developers and users. In this work, we have explored how block-based programming, which is already the basis for a large number of systems, can be enriched with visual features. As stated in our thesis statement, we set out to expand the range of programs that end-users can write using blocks, supporting both larger and more complex tasks while maintaining ease of use. We focused on three areas that pose challenges for end-users: coordinating parallel programs, decomposing large programs, and embedding nested expressions into imperative programs. For each area, we developed design candidates, analyzed them, and evaluated them empirically through studies with end-user participants. Our findings show that this approach can benefit end-users and make them more likely to overcome the programming challenges they encounter, pushing the boundaries of what end-users are capable of.

In Chapter 3, we present an environment targeting two robot-arms that conduct work in tandem. Coordinating programs to perform tasks in parallel is challenging, even for professional software developers. For this reason, teaching environments often establish other fundamental concepts first, such as mutable state or the control-flow in multi-threaded programs. In our comparative study on different language designs, we have found that it is possible to introduce end-users to coordination in a way that does not require these fundamental prerequisites. We observed that end-users can understand the flow of two parallel programs as long as the code visually matches the physical layout of the robot arm set-up. We further found that as long as we direct the programmers’ attention to points where the arms interact, they can also understand the behaviour of asynchronous sequences of commands in-between. These empirical insights, combined with a language design based on analytical insights, allowed us to build a coordinated programming system that end-users were more effective at using than comparable commercial tools.

In Chapter 4, we introduce an environment that guides end-users in writing larger programs. Work that focuses on beginner programmers frequently ignores the challenges of growing code bases altogether. Even most end-user tools are evaluated on small, toy-sized programs only. At the same time, tools for professionals assume that developers are already familiar with abstraction and decomposition techniques. This leaves a gap that is difficult to bridge, especially for end-users who primarily care about a problem at hand rather than general-purpose software design patterns. In our work, we carefully analyzed the domain that our environment addresses, and designed a system that provides end-users with a way to structure and navigate their
code. In the controlled experiment we performed, we gained insights into how and why end-users fail to decompose their code, and our system allows them to overcome the challenges they face. As a result, many of them were able to write substantially larger programs than in a traditional system.

In Chapter 5, we extend the environment from Chapter 4 with support for environmental triggers. These triggers introduce a new style of programming to the system, as they are defined as logical expressions that may have several nesting levels. Existing block-based tools are ill-equipped for this style of programming. Data-flow programming on the other hand is specifically designed for this type of program, but a poor fit for imperative robot command sequences. In our work, we created an experimental new environment that combines block-based and data-flow programming in a single system. On paper, this hybrid system allows users to use style of editing the fits best to the task at hand, but it also has drawbacks due to its higher complexity and inconsistent program editing experience. In a controlled study, we compared the hybrid system with a purely block-based one and found that it was outperformed by the block-based tool. We believe that this result validates previous findings on the benefits of block-based programming. Furthermore, it illustrates the importance of evaluating system design prototypes empirically to determine the impact of design trade-offs.

We hope that our contributions support further research on the use of block-based designs in end-user programming. We have outlined whether and how it is possible to adapt the design approaches we chose for our work to other domains and programming challenges that end-users face. Many of our designs are only prototypes, and our experimental findings need further work to investigate how they compare to experiences of using such systems outside of controlled study environments. In particular, we expect that field studies could lead to new insights when they study end-users that work in a self-driven fashion on real tasks that are directly relevant to them. However, we hope that this work contributes towards supporting end-users in creating increasingly complex programs and accomplishing their real-world goals.
Bibliography


[7] David Weintrop, David C Shepherd, Patrick Francis, and Diana Franklin. Blockly goes to work: Block-based programming for industrial robots. In *Blocks and Beyond Workshop (B&E)*, pages 29–36, 2017. → pages xiii, 3, 5, 6, 13, 33, 36, 37, 45, 69, 80, 83, 113


[38] Felienne Hermans, Kathryn T Stolee, and David Hoepelman. Smells in block-based programming languages. In 2016 IEEE Symposium on
Visual Languages and Human-Centric Computing (VL/HCC), pages 68–72. IEEE, 2016. → pages 3, 7, 74


[47] Nico Ritschel, Anand Ashok Sawant, David Weintrop, Reid Holmes, Alberto Bacchelli, Ronald García, Chandrika KR, Avijit Mandal,


[58] Andrew Begel. Logoblocks, 1996. → page 16


148


[105] James C Browne, Syed I Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel


[113] Kashif Amanullah and Tim Bell. Evaluating the use of remixing in scratch projects based on repertoire, lines of code (loc), and elementary patterns. In Proceedings of the Frontiers in Education Conference (FIE), pages 1–8, 2019. → page 72


152


[124] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman,


[130] JGraph Ltd. mxgraph. URL: https://jgraph.github.io/mxgraph/, 2017. → page 108


