Can Guided Decomposition Help End-Users Write Larger Block-Based Programs? A Mobile Robot Experiment

NICO RITSCHEL, The University of British Columbia, Canada
FELIPE FRONCHETTI, Virginia Commonwealth University, USA
REID HOLMES, The University of British Columbia, Canada
RONALD GARCIA, The University of British Columbia, Canada
DAVID C. SHEPHERD, Virginia Commonwealth University, USA

Block-based programming environments, already popular in computer science education, have been successfully used to make programming accessible to end-users in domains like robotics, mobile apps, and even DevOps. Most studies of these applications have examined small programs that fit within a single screen, yet real-world programs often grow large, and editing these large block-based programs quickly becomes unwieldy. Traditional programming language features, like functions, allow programmers to decompose their programs. Unfortunately, both previous work, and our own findings, suggest that end-users rarely use these features, resulting in large monolithic code blocks that are hard to understand. In this work, we introduce a block-based system that provides users with a hierarchical, domain-specific program structure and requires them to decompose their programs accordingly. Through a user study with 92 users, we compared this approach, which we call *guided program decomposition*, to a traditional system that supports functions, but does not require decomposition. We found that while almost all users could successfully complete smaller tasks, those who decomposed their programs were significantly more successful as the tasks grew larger. As expected, most users without guided decomposition did not decompose their programs, resulting in poor performance on larger problems. In comparison, users of guided decomposition performed significantly better on the same tasks. Though this study investigated only a limited selection of tasks in one specific domain, it suggests that guided decomposition can benefit end-user programmers. While no single decomposition strategy fits all domains, we believe that similar domain-specific sub-hierarchies could be found for other application areas, increasing the scale of code end-users can create and understand.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; **Procedures, functions and subroutines**; *Domain specific languages.*

Additional Key Words and Phrases: Block-based programming, program decomposition, mobile robots

ACM Reference Format:
Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C. Shepherd. 2022. Can Guided Decomposition Help End-Users Write Larger Block-Based Programs? A Mobile Robot Experiment. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 133 (October 2022), 26 pages. https://doi.org/10.1145/3563296

Authors' addresses: Nico Ritschel, The University of British Columbia, Vancouver, BC, Canada, ritschel@cs.ubc.ca; Felipe Fronchetti, Virginia Commonwealth University, Richmond, VA, USA, fronchettl@vcu.edu; Reid Holmes, The University of British Columbia, Vancouver, BC, Canada, rtholmes@cs.ubc.ca; Ronald Garcia, The University of British Columbia, Vancouver, BC, Canada, rxg@cs.ubc.ca; David C. Shepherd, Virginia Commonwealth University, Richmond, VA, USA, shepherdd@vcu.edu.

2475-1421/2022/10-ART133
https://doi.org/10.1145/3563296

1 INTRODUCTION

Programming has become integral for the work of millions of employees. Previous work estimated that in 2012, between 55 and 90 million employees in the U.S. performed basic programming tasks as part of their work [Ko et al. 2011; Scaffidi et al. 2005], a number that has likely grown over the past decade. However, as statistics from the US Department of Labor [2021] show, most of these employees are not professional developers. They are *end-user programmers*, who have not received formal education or training for performing programming-related tasks.

Because end-user programmers have limited formal programming training, they need tools that are specifically designed with their needs in mind [Dorn 2010; Wiedenbeck et al. 1995]. Although an extensive and diverse corpus of programming tools for end-users exist [Barricelli et al. 2019], *block-based programming environments* have emerged as one particularly successful form of beginner-friendly programming tool. Block-based environments provide a user interface that represents program syntax using graphical blocks and lets users compose blocks via drag and drop. These features are typically combined with a simplified high-level programming language to mitigate many of the frustrations that novice programmers typically encounter [Maloney et al. 2010].

Though block-based environments were originally conceived for computer science education [Maloney et al. 2010], they have been successfully used to support novice programmers in other domains. For example, end-users have automated their homes [Gonçalves et al. 2021], developed augmented reality apps [Mota et al. 2018], created mobile apps [Wolber et al. 2011], and programmed industrial robots [ABB Group 2021; Weintrop et al. 2017] using block-based environments.

Previous studies have shown that end-users find block-based languages easy to learn and use [Gonçalves et al. 2021; Mota et al. 2018; Weintrop 2019; Weintrop et al. 2018], although these evaluations were almost exclusively based on small programs that fit within a single screen. Code is easier to understand if it fits on a single screen without scrolling. Linting tools and coding styles have long been used to restrict the length of coding units created by professional software developers [Abbes et al. 2011; Charalampidou et al. 2015; Fowler 1999]. To write larger programs that are still understandable, developers must decompose their programs into short, related units of functionality (e.g., functions). Unfortunately, decomposing programs in this manner requires expertise that end-users typically lack.

Although most block-based environments provide procedural abstractions, their users rarely utilize them to structure their programs. Instead, they create programs that exhibit properties commonly understood to be problematic, employing long methods and code clones [Hermans et al. 2016; Robles et al. 2017]. Block-based languages support procedural decomposition and abstraction in theory, but they do little to encourage their use. Many evaluations of block-based programs include programs written for educational and other non-professional purposes. However, all available evidence suggests that users of block-based systems tend not to decompose their programs [Amanullah and Bell 2019], even though decomposition is crucial to successfully write larger programs.

In this paper, we present a block-based programming environment that guides users as they decompose their programs. The system specifically targets the programming of *mobile robot workers* (which we introduce in Section 2.1). Unlike other block-based programming tools, our environment presents users with a pre-defined program hierarchy and requires that end-users systematically decompose their programs. 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 make these restrictions more intuitive for users and make programs easier to navigate, the environment is divided into two side-by-side canvases, one of which is used to define tasks and the other one to compose them.

To evaluate whether our environment can make larger programs easier to understand for end-users, and thereby improve their programming performance, 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 [Gonçalves et al. 2021; Mota et al. 2018; Weintrop 2019; Weintrop et al. 2018], 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.

The primary contributions of this work are:

(1) The design and implementation of a programming environment that guides end-users as they decompose their programs.
(2) A study of 92 novice participants that found that few of those who used a traditional block-based environment used abstractions to decompose their programs.
(3) Evidence from said study that suggests that a system that makes decomposition mandatory and guides users as they apply it can improve their performance when solving tasks.

This work demonstrates that program decomposition is a relevant and difficult challenge for beginner programmers that must be addressed by tools that target end-users and aim to support large programs. The presented study only explored a limited set of tasks, but our findings suggest that enforcing decomposition and providing additional, domain-specific guidance to users can help them write better programs. We hope that these results encourage further work that explores domain-specific abstraction and decomposition mechanisms and applies them to a wider range of tasks and domains.

2 BACKGROUND

In this section we briefly introduce the domain this work targets, one-armed mobile robot workers, and the challenges that programmers (and end-user programmers in particular) face when they structure their programs using block-based programming languages.

2.1 Domain of Inquiry: One-Armed Mobile Robot Workers

This work focuses on the programming of one-armed mobile robot workers. This type of mobile robot consists of two separate hardware components: a robot arm that can manipulate objects, and a mobile base that can autonomously navigate between locations. An example of such a mobile robot is shown in Figure 1. The robot is stopped at one station (e.g., the table on the right), but can also move to other stations (e.g., the table behind the robot). The use case for the mobile robots that we target assumes that a robot has a designated task, requiring it to navigate between multiple workstations and conducting work at each of them. This scenario was inspired by the use of mobile

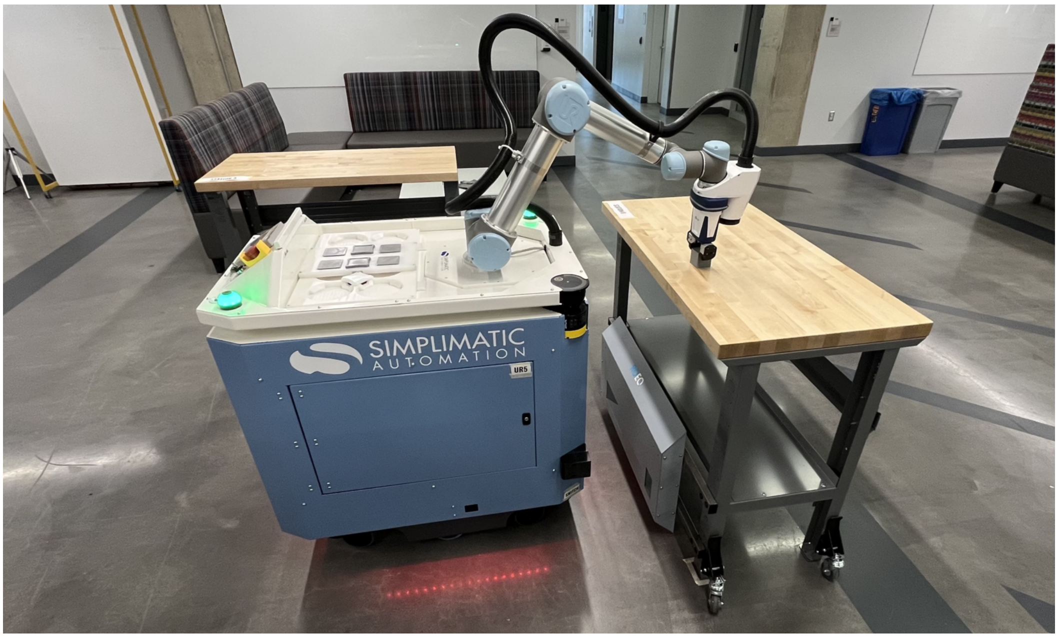


Fig. 1. Mobile robot with two components: a mobile base and a mounted robot arm for manipulating objects, stopped at one workstation (table right), but can also move to another workstation (table above left).

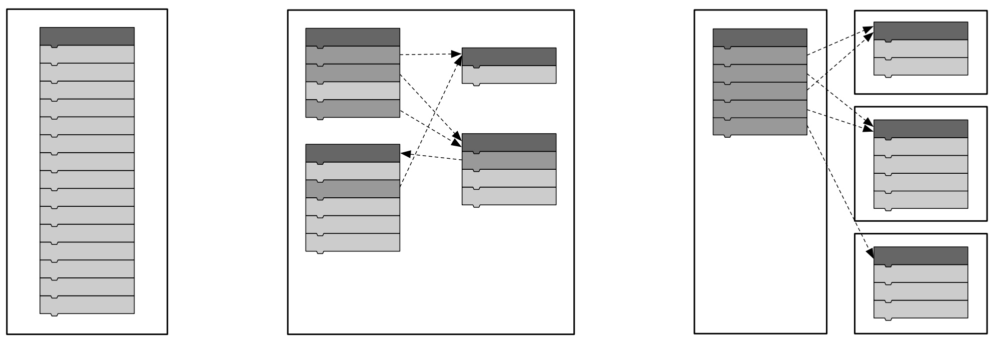
robots to automate lab work, such as the recent use of mobile robots to perform experiments around the clock [Burger et al. 2020].

Because mobile robot workers can perform a wide range of tasks and can be re-purposed without disassembling and reassembling their hardware, they are ideally suited to support end-user workers. Similar mobile robots have been used for assisting end-users before, but all programming was performed by experts [Hvilshøj et al. 2009]. Programming mobile robot workers is challenging, since tasks often require a large number of individual steps and depend on many environmental parameters, such as the locations of workstations and items available at them. Consequently, programs for these mobile robots are typically larger than those examined by past research investigating end-user block-based programming.

We target the mobile robot programming domain for two reasons: First, robot programming is a domain of relevance that has been targeted by previous work on end-user and block-based programming [Weintrop et al. 2017], but mobile robots, although already deployed in many warehouses and on factory floors [Siegwart et al. 2011], have not yet been targeted by end-user-centric work. Second, tasks for mobile robot workers are typically conceptually simple—moving boxes from station to station or re-arranging a stack of boxes—but require many individual instructions to complete. This makes them an ideal target for a system that explores program decomposition while otherwise using a fairly simple (and therefore easy to introduce) set of instructions.

2.2 Structuring Block-Based Programs

Most block-based languages support abstraction and program decomposition, often in the form of user-defined procedures or first-order functions. Section 7 provides a more extensive overview and discussion of these existing abstraction mechanisms.



(a) Unstructured programming

(b) Freely structured programming

(c) Hierarchically structured programming

Fig. 2. Styles of structuring programs.

Although users could benefit from structuring larger programs using the abstraction mechanisms that block-based languages offer, they rarely do so in practice [Hermans et al. 2016; Robles et al. 2017]. This leads to programs that contain long sequences of instructions and frequent code duplication. Work on professional developers suggests that this style of code can be detrimental to program comprehension, even for those professionals [Fowler 1999]. One expects inexperienced programmers, such as end-users, to be even more impacted by poorly structured code, and the observations we present in Section 5 support this assumption.

Drawing from previous work, and our own observations of end-users writing block-based programs, we have identified two common patterns for these 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 have illustrated it in Figure 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. Unfortunately, these users often struggle to find a structure that is compatible with their task, resulting in code that is inconsistently structured, more complicated than necessary, and therefore harder to understand and more error-prone.

The observation that novice programmers struggle to use traditional abstraction techniques, such as functions, is not surprising. 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 [Kallia and Sentance 2017]. If parameters are used to call functions, learners struggle to understand the semantics of pass-by-value versus pass-by-reference [Kennedy and Kraemer 2018]. 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 [Kennedy and Kraemer 2018]. In addition, users might accidentally discover and try to use techniques like recursion, which can lead to confusion and misconceptions [Lahtinen et al. 2005].

While many different approaches exist to introduce functions and even recursion to beginner programmers [McCauley et al. 2015], they typically assume a classroom-like situation where learners can work on carefully selected problems and slowly gain an understanding of the underlying concepts. However, in the context of end-user programming, users rarely have sufficient time and support to learn and practice programming skills in such an environment. Instead, they get exposed to programming systems (including existing block-based systems) that offer them an almost unrestricted design space, illustrated by Figure 2b. While this freedom in program design can benefit experienced users, beginners are more likely to be overwhelmed by this flexibility. For example, though Figure 2b might show a carefully crafted program with nested function calls (illustrated as arrows), it might be much harder for an end-user to understand the behaviour of this program, let alone edit it or create a program with such a structure by themselves. A traditional programming environment can require end-users to make design decisions that they are unaware of or unable to assess without external help. Therefore, we speculate that a system that reduces the abstraction decision space and provides guidance to users, illustrated by Figure 2c, can benefit these users and improve their ability to reasonably structure their work and consequently construct larger programs.

3 APPROACH: GUIDED PROGRAM DECOMPOSITION

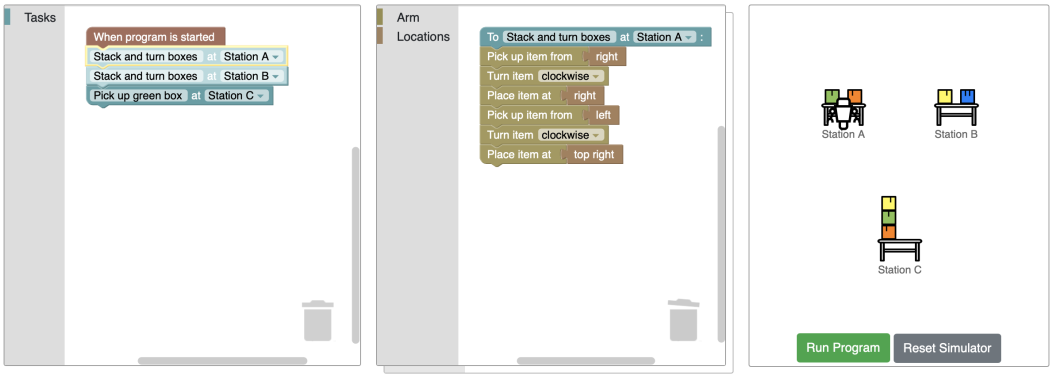
We propose a programming system that provides more guidance to novice users, such as end-users, than existing environments. The goal of this system is to reduce the design space to those decisions that these users are able to make effectively, and to explicitly give them the information needed to make 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 3, supports a fixed hierarchy of parameterless functions, which we call *tasks*. Figure 4 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.

We highlight our design decisions, and motivate and discuss them with respect to the framework of *13 Cognitive Dimensions of Notation (CDN)* [Green and Petre 1996], which provides terminology for analyzing visual languages. We aim to use CDN terminology in a way that is self-explanatory, but underline terms to indicate that they are based on an established definition. We further summarize key insights in boxes using the 💡 icon.

3.1 Why Do End-Users Not Use Functions?

Kallia and Sentance [2017] noted two factors that contribute to beginners' lack of function usage, which 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 [Chen et al. 2012] and likely more-so for novices [Rose et al. 2017]. 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 2b). The unconstrained nature of task decomposition can quickly lead to confusing call-chains (e.g., the calling relationships in Figure 2b), or even unintended recursive definitions (that likely lead to non-termination). These potential misuses of functions contribute to users



```
When program is started
  Stack and turn boxes at Station A
  Stack and turn boxes at Station B
  Pick up green box at Station C
```

```
To Stack and turn boxes at Station A :
  Pick up item from right
  Turn item clockwise
  Place item at right
  Pick up item from left
  Turn item clockwise
  Place item at top right
```

Fig. 3. Proposed programming system showing two tasks and side-by-side canvases. The left canvas shows the main program while the right one shows the body of the currently selected task(s). Users can test their programs using the simulator on the right side.

⟨*program*⟩⟶ **‘when started do’** ⟨*task*⟩*

⟨*task*⟩⟶ **‘do’** ⟨*taskname*⟩ **‘at’** ⟨*stationname*⟩

⟨*taskdef*⟩⟶ **‘to’** ⟨*taskname*⟩ **‘:’** ⟨*statement*⟩*

⟨*statement*⟩⟶ **‘pick from’** ⟨*location*⟩
| **‘place at’** ⟨*location*⟩
| **‘turn item’** (**‘clockwise’** | **‘counter-clockwise’**)

Fig. 4. 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 3.4 describes, there are some differences between the language's semantics shown here and its block-based presentation shown in Figure 3. Also note that locations are defined via a visual location picker that is not represented in this grammar.

creating programs structured like in Figure 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 [Hazzan 2008; Kramer 2007]. 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 5, 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, nor restricting the maximum length of continuous code blocks, can be a solution for this problem.

3.2 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 straight-forward 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 [Conway 1998; Harvey et al. 2013], but do so in more intrusive ways such as disabling blocks or showing error messages.

The system design allows only a fixed, two-layer call graph, which can limit the design freedom of experts but benefits the understanding of end-users and avoids accidental misuse of advanced language features like recursion.

3.3 Aligning Physical and Programmatic Scope

Complex scoping rules can be confusing and unintuitive for novice programmers [Kennedy and Kraemer 2018]. 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 [Ritschel et al. 2020; Weintrop et al. 2017], 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 [Weintrop et al. 2017]. While that work does not discuss their design decision in the context of CDN terminology, 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 more indirect 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 [Ritschel et al. 2020; Weintrop et al. 2017]. 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.

The system design allows end-users to define locations visually and per-task, which aligns their programmatic scope with the physical environment.

3.4 A Visual Programming Environment That Supports Decomposition

Our design is implemented as a multi-canvas environment, as shown in Figure 3. The main program is 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 3 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

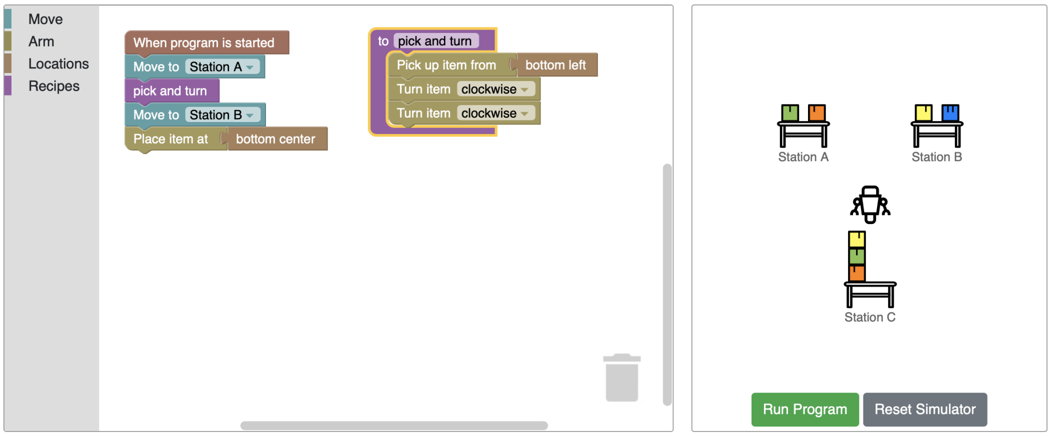


Fig. 5. The traditional block-based development environment used by the TR Cohort of the programming study.

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.

The system supports our overall design using a visual environment that uses two separate canvases and a side-by-side presentation of the program and the tasks it calls.

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 [Bragdon et al. 2010]. While 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 have to 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 [Elmqvist and Fekete 2009]. This approach tries to balance visibility with avoiding hard mental operations.

An interactive version of the presented environment is available online (select *Two-Canvas Mode*): https://vcuse.github.io/alvo/first-experiment/

4 EXPERIMENTAL SETUP

Previous studies of block-based environments and how end-users create programs in them have used small example tasks [Gonçalves et al. 2021; Weintrop et al. 2018]. 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, while the other used the environment with guided decomposition presented in Section 3. 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 design in detail.

4.1 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?

4.2 Study 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, while the second cohort (which we call *GD*) used the novel system with guided decomposition that we presented in Section 3.

Figure 5 shows the traditional block-based programming environment that participants in TR used; Figure 3 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 3. 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 (see Section 7 for some examples), we decided to call them *recipes* in our own interface to match the terminology used in a previous industrial robot programming environment [Weintrop et al. 2017]. 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 the statements listed in Figure 4.

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 8), 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 5) 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, while 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.

5 RESULTS

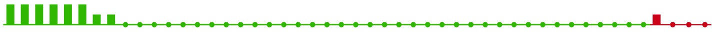
In the following section we discuss the results of our study, aligned with the research questions we presented in Section 4.1. As a reminder, we refer to the cohort of participants who used a traditional block-based programming system as *TR*, and to those we used our proposed novel system as *GD*. We highlight key results in boxes using the 🔍 icon.

An interactive version of all tutorials and tasks of our study, as well as the detailed results and metrics we have collected for each participant, are available online: https://vcuse.github.io/alvo/first-experiment/

5.1 RQ1: Program Decomposition in a Traditional System

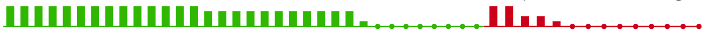
Figure 6 shows how participants in the TR cohort used functional abstraction, separated into Figures 6a, 6b, and 6c, which correspond to Tasks 1, 2, and 3, respectively. The right of each figure also shows how many participants re-used code as a result of applying functions.

For **Task 1**, the shortest task, participants used an average of 13.8 blocks (median: 13), of which 8.5 blocks (median: 8) were statements[1], and defined an average of 0.3 (median: 0) functions. Participants defined between 0 and 2 functions, as detailed in the following bar chart[2]:



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. While eight participants, as shown in Figure 6a, defined 1 or 2 functions, none of them re-used code by calling the same function more than once.

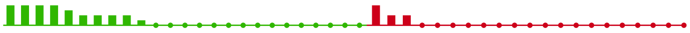
For **Task 2**, a larger task, participants used an average of 54.1 (median: 55) blocks, of which 29.9 (median: 31) were statements, and defined an average of 2.1 (median: 3) functions. Each participant defined between 0 and 4 functions, as detailed by the following bar chart:



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 6b, 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, while 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 6b, which meant that there were many missed opportunities for code reuse.

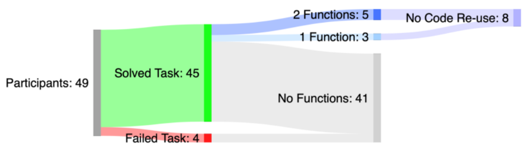
For **Task 3**, another larger task, the average solution contained 49.0 blocks (median: 49), of which 31.7 (median: 31) were statement blocks. Each participant defined between 0 and 4 functions, as detailed in the following bar chart:



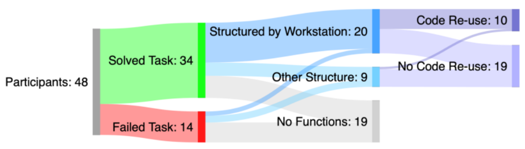
Only 13 (28%) of our participants used functions at all, a lower percentage than for Task 2, resulting in only 0.7 functions being used on average overall (median: 0). This led to less structure overall and even less code re-use than for Task 2, as shown in the middle-right of Figure 6c. Notably, 33

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

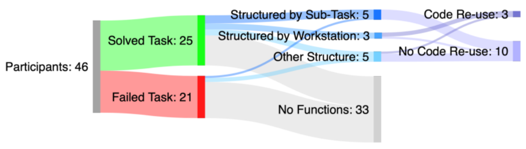
[2]Green bars (to the left) show successful tasks, red bars (to the right) show failed tasks. The height of the bar corresponds to the number of functions, dots represent solutions where no functions were used.



(a) Task 1

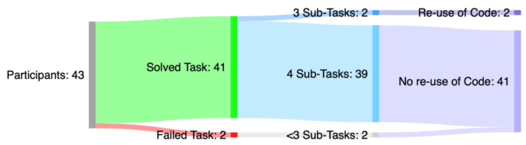


(b) Task 2

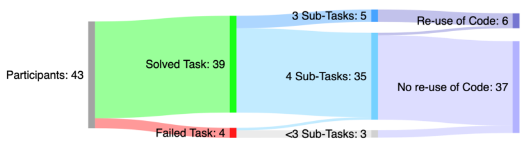


(c) Task 3

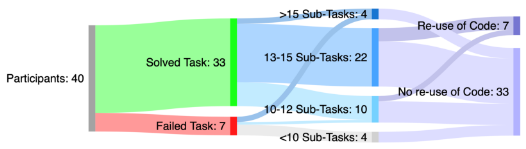
Fig. 6. Categorization of participants in the TR cohort and their programs for the three tasks our study.



(a) Task 1



(b) Task 2



(c) Task 3

Fig. 7. Categorization of participants in the GD cohort and their programs for the three tasks our study.

participants (72%) constructed their entire program as a single, continuous block of code, as seen in the middle-bottom of Figure 6c. 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.

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.

5.2 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 6b and 6c, 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.

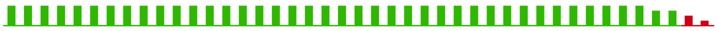
While we saw a correlation between function usage and success rates for both Tasks 2 and 3, we only saw a substantial difference in how fast participants solved a task for Task 2. For Task 2, the participants that used functions finished in 9.1 minutes on average, while the participants that did not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

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

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

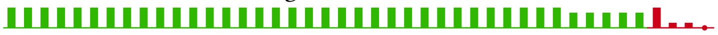
Figure 7 shows how participants in the GD cohort decomposed their code to solve each task. By design, the development environment with guided program decomposition required all participants in this cohort to decompose their programs, including the toy-sized Task 1. To avoid ambiguity between our three study tasks and the "tasks" that the environment offered participants as a way to decompose their programs (see Section 3) as "sub-tasks". In Figure 7 we categorize the usage of these sub-tasks based on their number, since unlike for RQ1, the environment does not allow differences in decomposition style.

For **Task 1**, participants used an average of 16.3 blocks (median: 17), of which 7.8 (median: 8) were statements[3], and defined an average of 3.8 sub-tasks (median: 4). The total number of blocks is higher than for the TR cohort, which is primarily due to the overhead introduced 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, as detailed in the following bar chart:



As visualized on the right side of Figure 7a, 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.

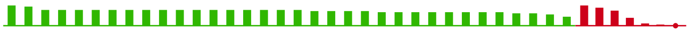
For **Task 2**, participants used 60.3 blocks on average (median: 63), of which 28.1 (median: 31) were statements, and defined an average of 3.7 sub-tasks (median: 4). Similar to Task 1, the average program length is therefore slightly higher than for the other cohort, although the average number of statements is slightly lower. The average number of sequential statements per code chunk (main program or sub-task) is 6.1 blocks (median: 6). Each participant defined between 0 and 4 sub-tasks, as detailed in the following bar chart:



[3]When we use the term "statements" here, we include sub-task calls and robot instructions, but not locations or headers.

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. Further, 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 functions, as detailed in the following bar chart:



As visualized in the middle of Figure 7c, participants needed many functions to solve this task. Nine participants (22%) solved the task successfully with 12 sub-tasks or less, 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 at. 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. While all of them were structured around workstations, some participants were following 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 contains 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.

Users with guidance employ substantially more program decomposition but still struggle with re-using code.

5.4 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 from Section 5.2 and comparing them with the charts from Section 5.3. Participants in the GD cohort had much more success, as indicated by the almost entirely green charts in Section 5.3. Figure 8 shows a more direct comparison of the two cohorts; the left side of this figure details success rates while the right side details task times.

For the first task, which was relatively simple, the success rates of participants in both cohorts were high (92% and 95%, shown on left of Figure 8). However, for Task 2, the TR cohort's performance dropped significantly (71%) while 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 8, there were no practical differences between tutorial times except on Task 3. To investigate this difference,

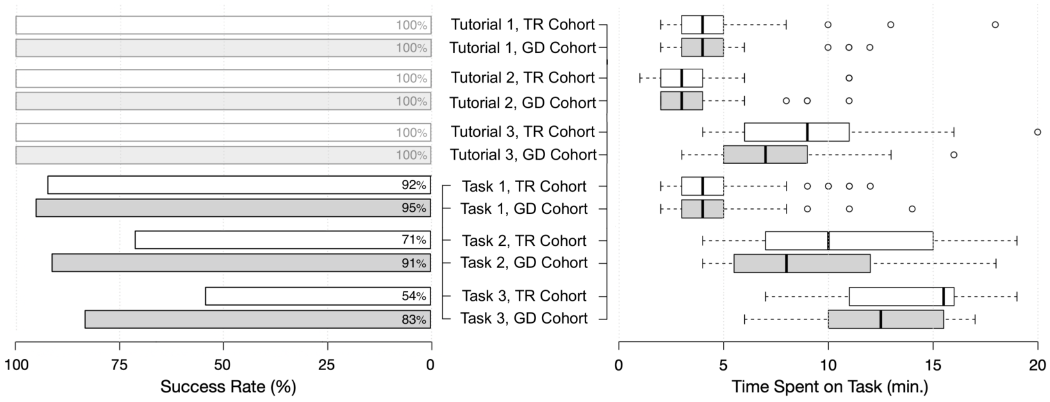


Fig. 8. User study results. For box plots: center lines show the medians, box limits indicate the 25th and 75th percentiles. Whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles. Outliers are represented by dots, only outliers < 20 minutes are shown.

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.). 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: $t_{\text{Task1}}(90) = -0.151, p = .880$; $t_{\text{Task2}}(89) = -1.89, p = .062$; $t_{\text{Task3}}(84) = -2.57, p = .006$.

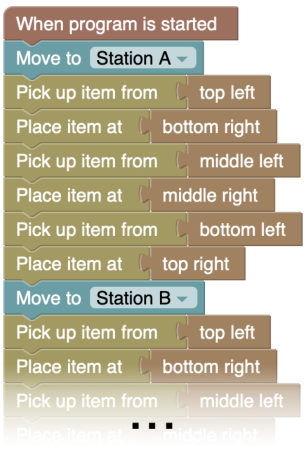
Users of the scaffolded environment were significantly more successful when solving larger tasks.

6 DISCUSSION

In this section, we discuss the implications and limitations of our approach and the findings from Section 5. As in Section 3, we use terminology from the framework of 13 Cognitive Dimensions of Notation (CDN) [Green and Petre 1996] and highlight these by underlining the CDN terms.

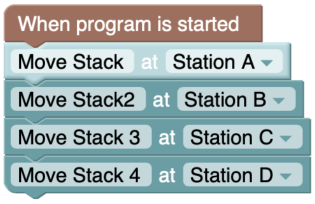
6.1 Benefits of Program Decomposition

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



```
When program is started
Move to Station A
Pick up item from top left
Place item at bottom right
Pick up item from middle left
Place item at middle right
Pick up item from bottom left
Place item at top right
Move to Station B
Pick up item from top left
Place item at bottom right
Pick up item from middle left
...
```

(a) TR Cohort: No structure (excerpt)



```
When program is started
Move Stack at Station A
Move Stack2 at Station B
Move Stack 3 at Station C
Move Stack 4 at Station D
```

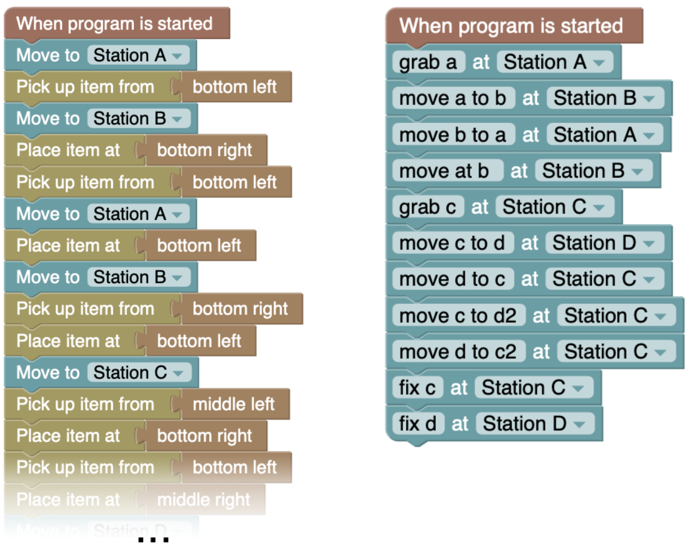
(b) GD Cohort: Hierarchical structure

Fig. 9. 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).

an environment that required them to decompose their programs and guided them towards a reasonable domain-specific decomposition, we saw their success rate 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 [Fowler 1999] 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 9 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 10a 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 10b 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 although 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 Figure 10a, 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 10a would have to perform the hard mental operation of chunking and summarizing the program's parts manually. In Figure 10b 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,



```
When program is started
Move to Station A
Pick up item from bottom left
Move to Station B
Place item at bottom right
Pick up item from bottom left
Move to Station A
Place item at bottom left
Move to Station B
Pick up item from bottom right
Place item at bottom left
Move to Station C
Pick up item from middle left
Place item at bottom right
Pick up item from bottom left
Place item at middle right
Move to Station D
...
```

(a) TR Cohort: No structure (excerpt)

```
When program is started
grab a at Station A
move a to b at Station B
move b to a at Station A
move at b at Station B
grab c at Station C
move c to d at Station D
move d to c at Station C
move c to d2 at Station C
move d to c2 at Station C
fix c at Station C
fix d at Station D
```

(b) GD Cohort: Hierarchical structure

Fig. 10. 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).

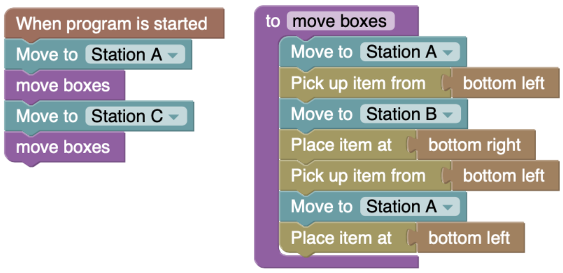
especially for novices [Holwerda and Hermans 2018]. However, especially block-based languages often lack places where users can use secondary notation or introduce barriers that make them difficult to access for beginners.

Figure 10 shows another example of the same pattern, but for Task 3. As described in Section 4.2, this task was intentionally chosen to contain a lot of steps that move the robot between workstations, resulting in a very fine-grained task separation when following our guided decomposition approach. Therefore, the program shown in Figure 10b 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 10a 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.

6.2 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 in a block-based environment, such as Task 1 in our study [Gonçalves et al. 2021; Mota et al. 2018; Weintrop et al. 2018]. 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, as suggested by the quantitative data, 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



```
When program is started
Move to Station A
move boxes
Move to Station C
move boxes
```

```
to move boxes
    Move to Station A
    Pick up item from bottom left
    Move to Station B
    Place item at bottom right
    Pick up item from bottom left
    Move to Station A
    Place item at bottom left
```

Fig. 11. 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.

explanation. Further examination of the data, comparing 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 11, 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 10b 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.

6.3 Domain-Specific Task Support: Beyond Mobile Robot Workers

Previous work on two-armed robots has demonstrated that customized programming environments for these types of robots can allow end-users to write complex, coordinated programs [Ritschel et al. 2020]. Similarly, this work demonstrates that a system that guides users in decomposing their programs can support end-users as their programs grow in size. Other end-user domains, such as home automation, app development or data collection from websites, might require a different approach to structuring programs. In particular, mapping between the kinds of goals end-user programmers have within a domain and the support for tasks, or functions, within an end-user programming environment will depend on the domain being supported. 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.

6.4 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 metric that both cohorts struggled with: 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 particular had obvious opportunities for code re-use, as illustrated by Figure 9: 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 on. This 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 make some attempts to make code re-use easier for participants: As Figure 3 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 or confusing to end-users. Finding a more effective way to encourage code re-use, especially in the presence of parameters, remains an open problem for future work.

6.5 Limitations

Here we discuss some of the limitations of the study we conducted.

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 while 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) [Paolacci et al. 2010]. We therefore believe that our AMT participants are an acceptable representation of the end-user demographic we aimed to investigate.

Recruiting Process: We recruited the 92 participants of our study in two waves, 55 and 37 participants respectively. While we planned to have 50-100 participants in total, 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 have 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 within the scope we have outlined in Section 2.1. 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 have evaluated our proposed system as a whole, which consists of several components that we have described in Section 3. While we have discussed the potential benefits of each component, our experiment cannot validate to which degree each component is responsible for the differences that we have observed between the two cohorts. In addition, we have 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. As we discuss in Section 7.2, 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.

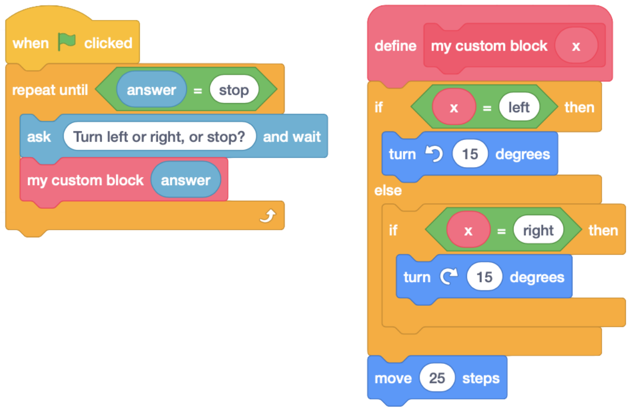
Further, there are other potentially relevant factors when evaluating the quality of a real program than those chosen by us for this experiment. For example, run-time performance might be relevant in practice, and it might be a goal for real programmers to minimize the time a robot spends moving between 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 used functions, 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.

7 RELATED WORK

Many block-based environments and programming languages exist, although only a minority of them target end-user programmers. A few block-based environments support abstraction via functions and other language features. In this section, we primarily focus on discussing this facet of related work, although we also discuss visual end-user languages broadly, and end-user *robot* programming in particular.

7.1 Blocks: Education vs. End-Users

The most popular block-based programming environment as of today is *Scratch* [Maloney et al. 2010], which is listed as the 22nd-most popular programming language overall in 2020 [Company 2021]. Scratch primarily targets young programming learners, who can use it to animate sprites and build simple, interactive 2D scenes. Scratch is currently built using the *Blockly* [Fraser et al. 2013] framework, which is the foundation of most block-based systems today. Blockly allows users to create their own block-based programming language and environment for a given domain. Two examples that show the range and extensibility of Blockly as a framework for educational tooling are *App Inventor for Android* [Wolber et al. 2011] for developing mobile apps and the educational robotics programming kit *OpenRoberta* [Jost et al. 2014].



```
when 🏁 clicked
repeat until (answer = stop)
    ask Turn left or right, or stop? and wait
    my custom block (answer)
↺

define my custom block (x)
if (x = left) then
    turn ⟲ 15 degrees
else
    if (x = right) then
        turn ⟳ 15 degrees
move 25 steps
```

Fig. 12. Scratch "custom block" for procedural abstraction.

In these educational environments there is often a close connection to text-based editing, as the goal for computer science students is to eventually transition from blocks to text-based source code, transferring their previously acquired knowledge [Fraser 2015]. Some block-based systems, like OpenRoberta, *Alice* [Conway 1998], and Pencil-Code [Weintrop and Holbert 2017] even foster this transition by allowing users to view the text-based equivalent of their block-based code directly inside the programming environment.

Only few block-based systems explicitly target end-users, like the robot programming environment *CoBlox* [Weintrop et al. 2017] or the system *Casa Assistiva EUD* [Gonçalves et al. 2021] for smart home control. These environments show less resemblance to text-based programming. In the example of CoBlox, users define target locations for a robot by manually moving it to its intended physical position. The entire process of capturing the robot's coordinates and assigning them to a variable takes is invisible to the user and not represented in the block-based code. This design decision comes with a trade-off: removing the concept of variables entirely makes the programming environment easier to learn, but limits its expressiveness.

7.2 Abstraction in Block-Based Programming

Several popular block-based programming languages make efforts to support larger programs by defining functions. For example, the block-based environment Scratch [Maloney et al. 2010] supports procedural abstraction through user-defined "custom blocks", as shown in Figure 12. Other block-based programming languages, like App Inventor for Android [Wolber et al. 2011] and OpenRoberta [Jost et al. 2014] even support functions that return a value.

Unfortunately, when block-based environments allow users to define functions, usability issues abound. Since all functions exist on the same canvas, they can, and often are, placed arbitrarily on the canvas, potentially confusing the user. Worse yet, as Scratch does not enforce scoping, a function parameter like x in Figure 12 could be easily dragged into the main program body and misused outside of its definition range, which Scratch silently ignores.

Usability issues like these may be the reason why few block-based programmers appear to use abstraction, even when it is offered. For example, a survey of Scratch programs in the wild found that code clones and other code smells are common in block-based code written by beginners [Hermans et al. 2016; Robles et al. 2017].

Nonetheless, some systems attempt to support even complex forms of abstraction. *Snap!* supports higher-order functions [Harvey et al. 2013], and App Inventor for Android supports object-oriented programming [Wolber et al. 2011]. Unfortunately, attempting to fit these abstractions into a block-based environment results in further usability concerns, like the need for type coercion, a language feature that is often seen as problematic and confusing [Pradel and Sen 2015]. We believe that making abstractions like these more usable in block-based environments is a promising avenue for future work.

7.3 End-User Robotics Programming

To use *expert* robot programming tools, users typically need extensive background knowledge in both computer science and robotics. Fortunately, there are two types of systems that aim to make robot programming accessible to end-users: those that are *manual* and those that are *automatic* [Biggs and MacDonald 2003].

Manual systems make programming simpler by adding support features for beginners and creating domain-specific languages with simple, high-level commands (e.g., move arm). An example of a manual system is CoBlox, a block-based programming environment for one-armed robots [Weintrop et al. 2018]. Automatic systems, on the other hand, use techniques that require no traditional programming at all, such as *demonstration-based learning*. In demonstration-based learning, users guide a robot arm by hand and then replay these movements as a program [Pan et al. 2010]. Other techniques like object or gesture recognition have also been applied successfully [Argall et al. 2009].

Both manual and automatic robot programming systems can target a variety of end-user types. Systems such as *Lego Mindstorms EV3* [Benedetelli 2013] (manual; using a visual programming language) target users with no previous programming experience, whereas systems like *Polyscope* [Universal Robots 2013] (manual; using a tree-based programming language), target intermediate users with some knowledge of robotics. The chosen target audience for an end-user tool typically affects its usability but also the complexity of the programs it can generate.

In the context of this work, all the presented tools face similar issues as block-based programming languages when they have to support larger programs. Flat programs without abstraction do not scale well beyond small tasks, no matter if they are represented as blocks or nodes in a visual graph. For programs learned by demonstration, which may not have a visual representation at all, large programs are especially difficult to teach and modify. Therefore, we see potential for further research on how the block-based representations presented in this work could be used to better represent programs for existing approaches.

8 CONCLUSION

Technology has become an integral part of the workplace, and as systems become more pervasive, the line between users and programmers continues to blur. Training more professional software developers is not enough to keep up with this progression. Instead, it is crucial to enable end-users to write code without having to become programming experts first. Block-based programming has been shown it can enable this need, but is held back as the designers of block-based systems rely heavily on established programming concepts.

Existing block-based tools have been designed primarily for teaching and require extension to move into the industrial workplace. The results of our study demonstrate that with traditional block-based environments, few end-users are able to complete larger development tasks. To overcome this limitation, block-based tools must become more mature and offer end-users help in similar ways as traditional development environments assist their professional users. Our results suggest that an environment with additional guidance on how to decompose programs can be a first step in this direction, as we have observed that it helps end-users substantially when writing larger programs. It remains an open question for future work to explore guided decomposition in other domains, or even find strategies that generalize to most end-user domains. However, we hope that this work can serve as inspiration for language and tool designers to push end-user programming beyond its current limitations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thorough and insightful feedback. This work has been funded by the National Science Foundation through grants 2024561 and 2050958, as well as an NSERC Collaborative Research and Development Grant.

REFERENCES

ABB Group. 2021. Wizard easy programming. *https://new.abb.com/ products/robotics/application-software/wizard* (2021).

Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. 181–190.

Kashif Amanullah and Tim Bell. 2019. 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)*. 1–8.

Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems* 57, 5 (2009), 469–483.

Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software* 149 (2019), 101–137.

Daniele Benedetelli. 2013. *Lego Mindstorms EV3 Laboratory: Build, Program, and Experiment with Five Wicked Cool Robots.* No Starch Press.

Geoffrey Biggs and Bruce MacDonald. 2003. A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*. 1–3.

Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code Bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 2503–2512.

Benjamin Burger, Phillip M Maffettone, Vladimir V Gusev, Catherine M Aitchison, Yang Bai, Xiaoyan Wang, Xiaobo Li, Ben M Alston, Buyi Li, Rob Clowes, et al. 2020. A mobile robotic chemist. *Nature* 583, 7815 (2020), 237–241.

Sofia Charalampidou, Apostolos Ampatzoglou, and Paris Avgeriou. 2015. Size and cohesion metrics as indicators of the long method bad smell: An empirical study. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*. 1–10.

Chiu-Liang Chen, Shun-Yin Cheng, and Janet Mei-Chuen Lin. 2012. A study of misconceptions and missing conceptions of novice Java programmers. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*. 1.

TIOBE The Software Quality Company. 2021. TIOBE Index. *https://www. tiobe.com/tiobe-index* (2021).

Matthew John Conway. 1998. *Alice: Easy-to-learn three-dimensional scripting for novices.* Ph. D. Dissertation. University of Virginia.

Brian James Dorn. 2010. *A case-based approach for supporting the informal computing education of end-user programmers.* Ph. D. Dissertation. Georgia Institute of Technology.

Niklas Elmqvist and Jean-Daniel Fekete. 2009. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *Transactions on Visualization and Computer Graphics (TVCG)* 16, 3 (2009), 439–454.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Neil Fraser. 2015. Ten things we've learned from Blockly. In *Proceedings of the Blocks and Beyond Workshop (B&B)*. 49–50.

Neil Fraser et al. 2013. Blockly: A visual programming editor. *https://code.google. com/p/blockly* (2013).

Mateus Carvalho Gonçalves, Otávio Neves Lara, Raphael Winckler de Bettio, and André Pimenta Freire. 2021. End-user development of smart home rules using block-based programming: A comparative usability evaluation with programmers and non-programmers. *Behaviour & Information Technology* (2021), 1–23.

Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

Brian Harvey, Daniel D Garcia, Tiffany Barnes, Nathaniel Titterton, Daniel Armendariz, Luke Segars, Eugene Lemon, Sean Morris, and Josh Paley. 2013. Snap!(build your own blocks). In *Proceeding of the Technical Symposium on Computer Science Education (SIGCSE)*. 759–759.

Orit Hazzan. 2008. Reflections on teaching abstraction and other soft ideas. *ACM SIGCSE Bulletin* 40, 2 (2008), 40–43.

Felienne Hermans, Kathryn T Stolee, and David Hoepelman. 2016. Smells in block-based programming languages. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 68–72.

Robert Holwerda and Felienne Hermans. 2018. A usability analysis of blocks-based programming editors using cognitive dimensions. In *symposium on visual languages and human-centric computing (VL/HCC)*. 217–225.

Mads Hvilshøj, Simon Bøgh, Ole Madsen, and Morten Kristiansen. 2009. The mobile robot "Little Helper": Concepts, ideas and working principles. In *Conference on Emerging Technologies & Factory Automation*. 1–4.

Beate Jost, Markus Ketterl, Reinhard Budde, and Thorsten Leimbach. 2014. Graphical programming environments for educational robots: Open roberta-yet another one?. In *International Symposium on Multimedia*. 381–386.

Maria Kallia and Sue Sentance. 2017. Computing teachers' perspectives on threshold concepts: Functions and procedural abstraction. In *Proceedings of the Workshop on Primary and Secondary Computing Education*. 15–24.

Cazembe Kennedy and Eileen T Kraemer. 2018. What are they thinking? Eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the Koli Calling International Conference on Computing Education Research*. 1–10.

Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44.

Jeff Kramer. 2007. Is abstraction the key to computing? *Commun. ACM* 50, 4 (2007), 36–42.

Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin* 37, 3 (2005), 14–18.

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

Renée McCauley, Scott Grissom, Sue Fitzgerald, and Laurie Murphy. 2015. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education* 25, 1 (2015), 37–66.

José Miguel Mota, Iván Ruiz-Rube, Juan Manuel Dodero, and Inmaculada Arnedillo-Sánchez. 2018. Augmented reality mobile app development for all. *Computers & Electrical Engineering* 65 (2018), 250–260.

Zengxi Pan, Joseph Polden, Nathan Larkin, Stephen Van Duin, and John Norrish. 2010. Recent progress on programming methods for industrial robots. In *International Symposium on Robotics and ROBOTIK*. 1–8.

Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. 2010. Running experiments on amazon mechanical turk. *Judgment and Decision making* 5, 5 (2010), 411–419.

Michael Pradel and Koushik Sen. 2015. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ron Garcia, and David C Shepherd. 2020. Comparing Block-based Programming Models for Two-armed Robots. *IEEE Transactions on Software Engineering (TSE)* (2020).

Gregorio Robles, Jesús Moreno-León, Efthimia Aivaloglou, and Felienne Hermans. 2017. Software clones in scratch projects: On the presence of copy-and-paste in computational thinking learning. In *International Workshop on Software Clones (IWSC)*. 1–7.

Simon Rose, MP Jacob Habgood, and Tim Jay. 2017. An exploration of the role of visual programming tools in the development of young children's computational thinking. *Electronic journal of e-learning* 15, 4 (2017), pp297–309.

Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. The '55m end-user programmers' estimate revisited. *Institute for Software Research, International, Carnegie Mellon University* (2005).

Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. 2011. *Introduction to autonomous mobile robots*. MIT press.

Universal Robots. 2013. PolyScope Manual.

US Department of Labor. 2021. Occupational outlook handbook. *https://www.bls.gov/ooh/* (2021).

David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.

David Weintrop, Afsoon Afzal, Jean Salac, Patrick Francis, Boyang Li, David C Shepherd, and Diana Franklin. 2018. Evaluating CoBlox: A comparative study of robotics programming environments for adult novices. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 1–12.

David Weintrop and Nathan Holbert. 2017. From blocks to text and back: Programming patterns in a dual-modality environment. In *Proceeding of the Technical Symposium on Computer Science Education (SIGCSE)*. 633–638.

David Weintrop, David C Shepherd, Patrick Francis, and Diana Franklin. 2017. Blockly goes to work: Block-based programming for industrial robots. In *Proceedings of the Blocks and Beyond Workshop (B&B)*. 29–36.

Susan Wiedenbeck, Patti L Zila, and Daniel S McConnell. 1995. End-user training: An empirical study comparing on-line practice methods. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 74–81.

David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. 2011. *App Inventor*. O'Reilly Media, Inc.