Interacting with Computers, 2025, 1–18
https://doi.org/10.1093/iwc/iwaf028
Article

Block-based or graph-based? Why not both? Designing a hybrid programming environment for end-users

Nico Ritschel[1], Reid Holmes[1], Felipe Fronchetti[2], Ronald Garcia[1] and David C. Shepherd[3]

[1]University of British Columbia, Vancouver, Canada
[2]Virginia Commonwealth University, Richmond, United States
[3]Louisiana State University, Baton Rouge, United States
Corresponding author: ritschel@cs.ubc.ca

End-user programmers need programming tools that are easy to learn and use. Development environments for end-users often support one of two visual modalities: block-based programming or data-flow programming. In this work, we discuss differences in how these modalities represent programs, and why existing block-based programming tools are better suited for imperative tasks while data-flow programming better supports nested expressions. We focus on robot programming as an end-user scenario that requires both imperative and expressions-based code in the same program. To study how end-user tools can better support this scenario, we propose two programming system designs: one that changes how blocks represent nested expressions, and one that combines block-based and data-flow programming in the same hybrid environment. We compared these designs in a controlled experiment with 113 end-user participants who solved programming and program comprehension tasks using one of the two environments. Both groups indicated a small preference for the hybrid system in direct comparison, but participants who used blocks to solve tasks performed better on average than hybrid system users and gave higher usability ratings. These findings suggest that despite the appeal of data-flow programming, a well-adapted block-based programming interface can lead end-users to more programming success.

RESEARCH HIGHLIGHTS

- Block-based programming is a modality commonly found in imperative programming tools for end-users.
- Data-flow graphs are common as well, and a better fit for programs with nested expressions.
- We propose end-user friendly designs for programs that combine imperative and expression-based code.
- We find that users perform better using a block-based design compared to a block-and-graph hybrid.
- This finding suggests that consistency is more important for end-users than fitness for purpose.

Keywords: application specific development environments; visual languages; empirical studies in HCI.

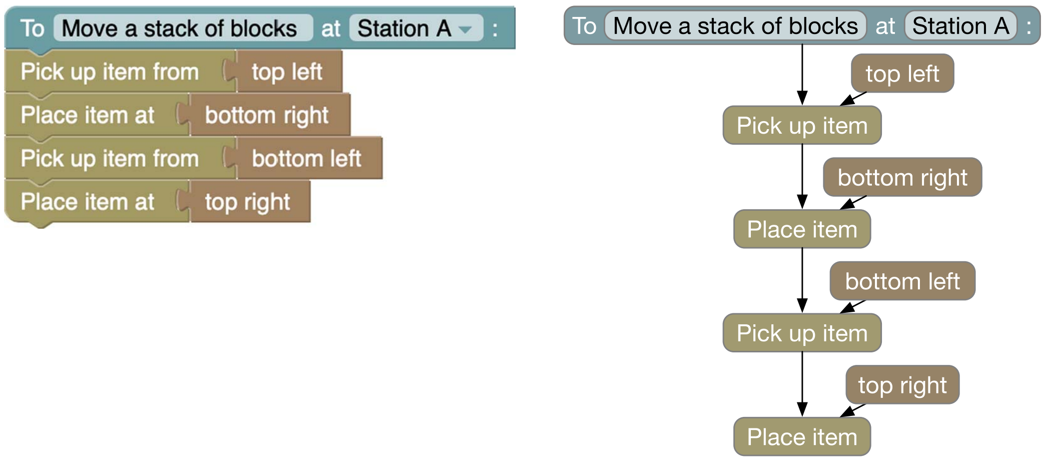
1 Introduction

Millions of people write code as part of their job, but only a few of them are professional software developers (Ko *et al.*, 2011, US Department of Labor, 2021). The vast majority of programmers are *end-users*, who receive no formal programming-related education and only limited training. Due to this lack of formal expertise, end-users would struggle to use most development environments and general purpose programming languages, because they are designed for professional software developers. Instead, end-users rely on domain-specific tools and languages that are designed to be easy to learn and use (Dorn, 2010, Wiedenbeck *et al.*, 1995).

End-user tools typically provide rich user interfaces that leverage visual aids and notations, making them expensive to develop from scratch. To save development cost and effort, many tool developers build on established frameworks that are suitable for a wide range of programming domains. *Block-based programming* and *data-flow programming* are two of the most commonly used frameworks for this purpose. The former uses visual puzzle blocks to represent program syntax, while the latter represents programs as visual directed graphs that illustrate information flow.

Previous work has explored how blocks and data-flow graphs can benefit different domains, but has only considered them independently. Data-flow programming has long been used in industrial languages and applications (Kelly *et al.*, 1961, Morrison, 1994), and has recently been re-popularized and applied to new domains, such as game development (Bertolini, 2018, Sewell, 2015) and workflow automation (Fagan, 2007). Data-flow representations tend to be used for data and event processing, because they are well-suited to illustrating the step-wise evaluation of mathematical and logical expressions. Block-based programming on the other hand has its roots in computer science education (Maloney *et al.*, 2010, Weintrop, 2019), but has recently been proposed as a solution for end-user programming as well, targeting areas such as home automation (Gonçalves *et al.*, 2021) and industrial robot programming (ABB Ltd, 2020). Blocks are typically used to used to describe imperative, story-like sequences of commands, like animations (Conway, 1998, Maloney *et al.*, 2010) or robot

Received: July 26, 2024. **Revised:** March 11, 2025. **Accepted:** March 22, 2025


To Move a stack of blocks at Station A :
Pick up item from top left
Place item at bottom right
Pick up item from bottom left
Place item at top right
To Move a stack of blocks at Station A :
top left
Pick up item
bottom right
Place item
bottom left
Pick up item
top right
Place item

(a) Block-based representation (b) Data-flow graph representation

FIGURE 1. An imperative block-based program for a mobile robot. The block representation (a) matches the flow of program execution and text. The graph representation (b) can replicate this, but adds visual clutter rather than further clarity.

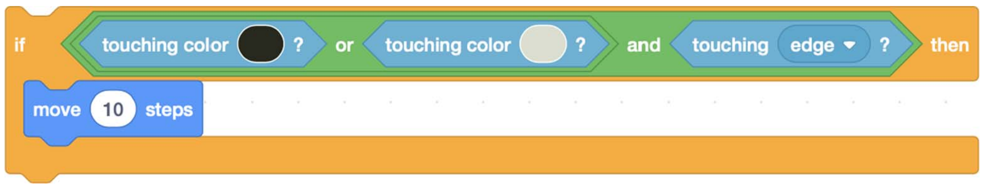


FIGURE 2. A conditional expression in the block-based language Scratch: the nested structure is compressed into a single line and hard to read and edit.

manufacturing steps (ABB Ltd, 2020, Weintrop *et al.*, 2017). However, previous work has chosen one of the two frameworks or the other, without explicitly discussing the trade-offs that come with this choice, or evaluating how their individual strengths or limitations fit a specific programming domain.

In this work, we investigate the comparative benefits and shortcomings of block-based and data-flow programming for end-user programming. For imperative programs, existing block-based environments resemble the flow of natural language instructions, making them intuitive to read for beginners (see Figure 1a). Data-flow programming can imitate this style (see Figure 1b), but adds visual clutter with little apparent benefit. Conversely, existing block-based tools are poorly optimized for representing nested expressions. As Figure 2 illustrates, even when block-based expressions grow in size and complexity, they are usually compressed into a single line and difficult to read and edit. Some existing block-based systems have found visual workarounds for this issue (Bak *et al.*, 2018, Mattioli & Paternò, 2020), but expressions still follow a pre-determined, rigid shape that can be hard to read. Data-flow programming is better suited for describing such expressions (see Figure 3), as it reveals the flow of information and illustrates how individual sub-expressions contribute to a resulting value (Johnston *et al.*, 2004).

Ideally, an end-user programming tool should support both imperative and expression-based code. When users define *what* a program should do, an imperative style is often easier, whereas defining *when* to do it often requires them to create logical conditions in the form of nested expressions. In this work, we consider a concrete use case where both aspects of a program are important and can become complex enough to challenge end-users: programming a mobile robot that moves between workstations and operates machinery. Previous work has identified the value

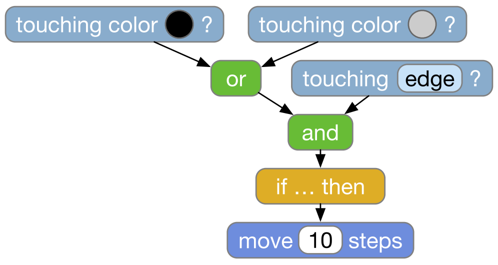


FIGURE 3. Mock-up of a data-flow graph representation equivalent to the program in Figure 2. The data-flow and structure of the nested expression is visually highlighted and easier to read.

of empowering end-users to program mobile robot assistants that can can freely and autonomously conduct work across workstations (Burger *et al.*, 2020, Ritschel *et al.*, 2022). Existing approaches have demonstrated that block-based programming can support end-users as they define *what* actions such a robot should perform (Ritschel *et al.*, 2022, Weintrop *et al.*, 2017). However, operating machines, for example by first loading and later unloading them, requires more careful planning of *when* a robot should execute these operations. A widely used strategy for program planning in end-user tools are *triggers*, a simplified form of event-based programming where users combine external input signals into logical expressions (Ovadia, 2014, Ur *et al.*, 2014). Existing end-user automation tools typically combine triggers with simple, atomic *actions* that are executed when a trigger expression is satisfied. When integrated into existing block-based robot programming tools, triggers could allow users to plan the execution of their code, while robot command blocks allow them to define complex, highly customizable actions.

Triggers require the definition of nested expressions, which traditional block-based programming tools do not support well. We consider two options to overcome this limitation: one is to adapt block-based programming to better support nested expressions, and the other is to build a *hybrid* system that supports both blocks and data-flow graphs, using each for different parts of the same program. To investigate the suitability of the two options further, we created two prototype environments that each implement one of them and represent triggers either as blocks or as a data-flow graph. We conducted a between-subject controlled experiment to compare the two prototypes. We recruited 113 end-users, via the *Prolific* online platform, to compare them. Each participant was trained to use their respective environment and was asked to complete two complex tasks that required writing both nested expressions for triggers and imperative robot code. We then evaluated their ability to comprehend isolated, complex examples of triggers that were represented using blocks or using a graph. Finally, we asked participants to rate their assigned environment in a survey, and showed them the alternative design to give them an opportunity to compare the two options.

We find that participants performed better on average when using a purely block-based environment, both when writing programs and when trying to understand them. These results are complemented by the participants' subjective ratings, which were higher for those who used blocks exclusively. However, when asked to compare the environment they used with the one they did not, both participants who used blocks and those that used data-flow graphs preferred the graph-based environment. Our observations illustrate the appeal that data-flow programming has on end-users, but also highlight its practical limitations compared to purely block-based tools. For the domain of robotics, we find that introducing graphs as a situational alternative does not clearly benefit end-users. For other domains, our findings suggest that designers should deeply consider whether the advantages of any novel representation can outperform blocks for end-user programming tasks.

2 Background

In this section, we present an overview of related work in end-user programming, as well as block-based and data-flow programming. Block-based programming and data-flow programming have been studied and evaluated independently in previous work, so we first present them separately, in the context of end-user programming. We then provide an overview of the smaller body of empirical and analytical evaluations of either of the two modalities, and their relation to our work.

2.1 End-user Programming

End-users make up a substantial portion of the programming workforce, and have been targeted by a large body of work (Ko *et al.*, 2011, Wiedenbeck *et al.*, 1995). Most of this work focuses on making it easier for end-users to write and edit programs. To achieve this goal, *end-user programming* systems typically provide richer user interfaces, visual programming languages, and more scaffolding than systems intended for professionals (Barricelli *et al.*, 2019). These features can be expensive to design and develop, but also come with another cost: most end-user systems are domain-specific and tailored to a limited range of tasks (Ajaykumar *et al.*, 2021).

Most end-user systems target a domains with sufficient demand for dedicated tools, such as business, web development, automation, and robotics (Barricelli *et al.*, 2019). Some of these domains have established dedicated programming approaches, such as spreadsheets for business applications (Nardi & Miller, 1990), or gesture and demonstration-based programming for robotics (Biggs & MacDonald, 2003). Nonetheless, most end-user tools rely on more traditional approaches that are easier to adopt, such as the data-flow or block-based programming modalities we compare in this work (Ajaykumar *et al.*, 2021, Barricelli *et al.*, 2019). In practice, tool developers have little guidance on which of approach they should choose from the multitude of different design alternatives, and this absence motivates our work to better understand their trade-offs.

Writing programs is typically the core, but not the entirety of the software development work that end-users have to perform. As they gain more experience and work on larger code bases, end-user programmers often engage in tasks such as editing, refactoring, or debugging code. Some tools aim to support end-users with a wider range of software-related tasks, such as debugging, documenting, or maintaining their code (Dittrich *et al.*, 2013). These *end-user development* often provide even more complex and custom-tailored tools and visualizations (Dittrich *et al.*, 2013, Fleming *et al.*, 2013). Furthermore, previous work has proposed debugging features tailored towards the design of block-based and data-flow programming tools (Banken *et al.*, 2018, Kim *et al.*, 2018). However, we see these development questions as mostly independent from the underlying language design. The study we present in this work touches these topics, for example by giving participants the opportunity to test and debug their code, and tasks that ask them to extend existing code. However, it provides the same functionality to all participants, and does not aim to study the broader end-user development process .

2.2 Block-based Programming

Block-based programming represents the syntax of programs as visual puzzle blocks. Programmers can drag these blocks from a library of language fragments onto the programming canvas and attach them to previously placed blocks. By visualizing the legal ways in which blocks can be assembled using jigsaw-shaped notches, block-based languages guide novice programmers as they construct their programs (Maloney *et al.*, 2010, Weintrop, 2019). Compared to other types of structural editors that partially rely on text inputs (Hempel *et al.*, 2018, Voinov *et al.*, 2022), block-based editors rule out syntactic errors entirely. In addition, every program that can be visually assembled is also executable and can be tested immediately. These design aspects reduce frustration and encourage inexperienced programmers to tinker with their code (Fraser, 2015, Maloney *et al.*, 2010).

The *Scratch* (Maloney *et al.*, 2010) editor has popularized block-based programming over the past decade. Scratch remains the block-based system with the most users today, and was listed as the 20th-most popular programming language overall by the TIOBE Index in 2022 (TIOBE The Software Quality Company, 2023). Scratch and most other popular block-based environments, like *Snap!* (Harvey *et al.*, 2013) and *MIT App Inventor* (Wolber *et al.*, 2011), were designed to teach programming to students from K-12, as well as junior university students. As such, these environments are easy to learn, but are not necessarily useful for writing complex programs for production use. Instead, they are typically designed to offer their users explicit paths to transition to traditional, text-based languages (Fraser, 2015, Weintrop & Holbert, 2017). Some even offer ways to switch between block-based and text-based representations of the same program (Kelleher *et al.*, 2007), allowing students to transform their block programs to text.

More recently, end-user programming tools have started using block-based programming as well (ABB Ltd, 2020, Gonçalves *et al.*, 2021, Weintrop *et al.*, 2017). Early evaluations suggest that the benefits of blocks also apply to this domain (Gonçalves *et al.*, 2021, Ritschel *et al.*, 2022, Weintrop *et al.*, 2018). However, most block-based end-user systems and their evaluations focus on small fragments of code. The limitations of these tools typically become severe once programs grow large or complex. Some end-user systems have added further environmental support or visual programming elements on top of blocks to improve their readability (Fronchetti *et al.*, 2023, Ritschel *et al.*, 2022). However, these additions only target individual, domain-specific programming use cases. In Section 3, we describe the inherent limitations of blocks in more detail.

2.3 Data-flow Programming

Data-flow graphs visualize the flow of information in a program. Nodes represent fragments of computation, which are connected via directed edges that show how the results of one node's computation feed into another node. Typically, nodes correspond to a program's syntactic fragments, similar to each individual block in block-based programming. Edges on the other hand do not necessarily match the syntactic structure of the program, so nodes can have multiple outgoing edges or there might even be cycles in the graph. However, if computation is imperative and side-effects are ignored, data-flow graphs can appear quite similar to blocks, as in Figure 1. Such a tree-shaped data-flow graph can be trivially converted into blocks by replacing edges with block connectors, and vice-versa.

In contrast to blocks, data-flow programming has long been used to target industrial end-users. *Block diagrams*, not to be confused with block-based programming, were introduced in 1960 as a text-based description language for data-flow graphs used in electrical engineering (Kelly *et al.*, 1961). Subsequent refinements allowed the visualization of increasingly complex structured programs as flow charts (Morrison, 1994) and introduced them as a teaching tool (Calloni & Bagert, 1994). Although data-flow programming tools have remained niche until the early 2000s (Johnston *et al.*, 2004), selected systems like *LabView* (Vose, 1986) have gained popularity in domains like engineering and system design.

In recent years, data-flow programming has seen a resurgence, likely due to technological advances such as higher-resolution monitors and faster processing and rendering capabilities. One of the most successful modern data-flow languages is the *Blueprints Visual Scripting* language used in the Unreal Engine for game development (Sewell, 2015). This scripting language has influenced several similar languages used in game development, such as the *Unity Visual Scripting* language (Bertolini, 2018). In addition, data-flow programming has become a popular tool for managing high-level information flows and interactions, for example between different web services (Morrison, 2010), home automation systems (Brich *et al.*, 2017, Ghiani *et al.*, 2016), or Internet-of-Things devices (Lekić & Gardašević, 2018).

Similar to block-based code, data-flow programs can become difficult to read and edit as they grow. Laying out large graphs without long or crossing edges can be challenging, especially for novices. Some data-flow editors address this challenge by offering automated layout assistants (Sewell, 2015). However, these systems typically discards any manual effort that users have applied to structuring their code. It also does not fully alleviate the issue of long edges or hard to read large programs entirely if users do not manually decompose their code. We discuss the usability limitations of data-flow programming in more detail in Section 3.

2.4 Analysis and Comparison of Blocks and Data-flow Graphs

A substantial body of previous work has studied the effectiveness of data-flow-based systems (Burnett *et al.*, 1994, Green & Petre, 1996, Whitley, 1997, Whitley *et al.*, 2006) or block-based systems (Fronchetti *et al.*, 2023, Ghiani *et al.*, 2016, , Ritschel *et al.*, Weintrop *et al.*, 2018) in comparison to text. Other work has investigated the benefits of such systems in comparison to more traditional, form-based wizards (Desolda *et al.*, 2017). Most of this work contributes empirical evidence for the advantages of the respective visual modality, often finding substantial benefits with respect to learnability or usability. However, these benefits can be rarely traced back to specific design decisions, since the work only compares between the presence of one specific approach and its absence. Other work has compared design decisions explicitly, but only within one of the two modalities of blocks (Holwerda & Hermans, 2018, Ritschel *et al.*, 2022, 2020) and data-flow graphs (Green & Petre, 1996). Some of this work has identified the benefits of the respective systems, and discussed potential usability limitations and design challenges for future systems (Green & Petre, 1996, Holwerda & Hermans, 2018). Other work has explored ways to represent the same program using different syntactic choices (Ritschel *et al.*, 2022, 2020), but only within the same overall program representation.

We are not aware of any previous work that compares, or even considers, both block-based and data-flow programming as potential solutions for the same programming task. We speculate that this lack of comparative work is a result of block-based systems being historically focused on computer science education and only recently entering the domain of end-user programming. Nonetheless, more recent work shows clear overlaps in the domains that both systems target, resulting in block-based (Weintrop *et al.*, 2018) and data-flow-based (Alexandrova *et al.*, 2015) solutions to robotics programming, or blocks (Gonçalves *et al.*, 2021) and graphs (Brich *et al.*, 2017) being used in home automation. Because of the lack of comparative work, the individual advantages of the underlying modalities in these domains, and whether there are potential trade-offs between them, remain unclear. One of the contributions of this work is to address this lack of comparative evaluation.

3 Comparing Block-based and Data-flow Programming

In this section, we compare the strengths and limitations of block-based and data-flow programming with a focus on end-user applications. We build on the previous introduction of both programming modalities from Sections 2.2 and 2.3. Our comparison in this section is organized around the framework of *13 Cognitive Dimensions of Notations (CDN)* (Green, 1989). One of the earliest uses of the CDN has been the analysis and comparison of the design of graph-based (Green & Petre, 1996) systems. The structure and findings of this comparison inform the discussion we present in this section. The framework has also been used in previous work to compare among competing block-based (Holwerda & Hermans, 2018) design alternatives. We therefore see CDN as a useful tool to extend the previous work and conduct a similar comparison of the cognitive effects on programmers across the two modalities. While most CDN terminology is self-explanatory, we introduce

TABLE 1. Summarized comparison of block-based and data-flow programming along the cognitive dimensions discussed in this section.

Cognitive Dimension	Block-based Programming	Data-flow Programming
Closeness of Mapping	No improvements over pure text	Improved through the explicit representation of data-flow
Hard Mental Operations	Programmers must untangle nested expressions	Programmers (typically) need to manually arrange nodes
Hidden Dependencies	No improvements over pure text	Shared dependencies can be highlighted by re-using existing nodes
Diffuseness	Low number of syntactic elements that must be learned	Slightly higher than for blocks due to edges as additional syntactic element
Consistency	High for complete programs; handling of missing code can introduce inconsistencies	Very high for side-effect free programs; significantly lower when state has to be represented
Secondary Notation	Rarely supported and often hard to access	Manual arrangement of nodes adds an inherent secondary notation layer
Visibility	Often in conflict with secondary notation; degrades rapidly as programs grow	Requires more screen space and programmer effort, but can scale better for large programs
Viscosity	Low for imperative code, but can be very high for nested expressions	Low, but increases when nodes need to be re-arranged to maintain visibility
Premature Commitment	Rarely necessary, as blocks can be easily rearranged and reassembled	Often necessary, as users need to commit to an arrangement of nodes early-on

some terms as necessary, and highlight all CDN references in the text by *underlining and italicizing* them.

We do not explicitly cover all dimensions of the CDN, but only the 9 that are most directly influenced by the visual representation of programs. Our findings for these dimensions are summarized in Table 1. We skip the dimension of *progressive evaluation* support, as it is mostly influenced by a system's integration with its run-time environment, which is orthogonal to how programs are represented. We also ignore the *abstraction gradient* and *role-expressiveness* dimensions, which are primarily determined by programming language features such as abstraction mechanisms and can be represented equivalently in both modalities. Finally, we discuss the dimension of *error-proneness* inline when it is affected by design decisions that we made to improve cognitive effects along other dimensions affected by visual representation.

3.1 Closeness of Mapping and Hard Mental Operations

Block-based languages are designed to reduce the barrier of entry to programming by making programs more similar to natural language text (Conway, 1998, Maloney *et al.*, 2010). To achieve this goal, most block-based systems avoid special syntax, such as parentheses, that are unfamiliar to non-programmers. In addition, the systems typically structure program syntax in a way that matches the reading flow of natural language. Consider Figure 1 as an example, where a robot is instructed to execute a sequence of commands. The way blocks are arranged in Figure 1a allows programmers to read this program in a single pass, top-to-bottom and left-to-right, matching the flow of English language text: "To move stack of blocks at Station A: Pick up item from top left; place item at bottom right; ...". Many block-based systems even offer on-the-fly translation of programs, potentially mirroring the block shapes to adapt to languages like Arabic that are read right-to-left.

By making programs closely resemble the structure of text, block-based languages forego a common goal of visual languages, which is to improve the *closeness of mapping* between how a program and its semantics are visually represented. For example, visual languages might choose to represent the operations of the program in Figure 1a by visualizing locations or movement paths. This representation would simplify the potentially *hard mental operation* of visualizing the instruction substantially. Notably, some block-based systems provide minor visualizations that are embedded into blocks, such as the colors shown in Figure 2. However, they avoid more complex visual aids that would both increase the implementation cost of such a system or add visual elements that programmers have to learn. This strategy of keeping block-based frameworks reusable and easily recognizable by users has likely contributed to their success.

The approach that blocks use to match natural language works particularly well for imperative code. To illustrate how blocks represent nested logical expressions, Figure 2 shows a simple conditional statement in block-based system Scratch (Maloney *et al.*, 2010). Unlike traditional programming languages, Scratch represents this nested expression without parentheses or other special syntax. Instead, block boundaries mark the nesting structure of the sub-expressions, making the resulting expression resemble an English sentence: "If [you are] touching [the] color black or touching [the] color grey and touching [an] edge, then move 10 steps." Compared to the purely imperative example, the condition described by this sentence is harder to read and understand. More importantly though, the nesting of the logical clauses in the sentence is ambiguous without also inspecting the block boundaries in Figure 2. Requiring users to closely examine the nesting of blocks increases the system's *error-proneness*. Furthermore, it introduces a new *hard mental operation* that is similar to that of parsing a complex English sentence.

Data-flow graphs follow a different strategy than blocks, but typically aim for a similar goal. Instead of imitating natural language, they make programs resemble flow charts that use familiar symbols, such as arrows, to direct the reader's attention. For imperative programs, this approach can lead to a similar result as for blocks, as shown by the graph in Figure 1b that resembles the block-based program in Figure 1a. However, unlike for blocks, the readability of data-flow graphs highly depends on their visual layout. For example, the graph shown in Figure 1b could have been drawn with nodes placed at random locations. Without an automated system for arranging nodes, controlling the layout of graphs can be an additional *hard mental operation* that is not present in blocks.

Although visually laying out a program can add a mental burden for programmers, it can also be beneficial. For example, Figure 3 shows the data-flow equivalent of Figure 2. In this example, the arrow notation of the data-flow graph improves the *closeness of mapping* between notation and program execution flow. Therefore, reading the nesting and order of logical operations is substantially easier. This layout also reduces the risk of users misreading or misinterpreting the nesting structure, which reduces the system's *error-proneness*. However, this change in layout also comes at a space cost, which we discuss further in Section 3.4 as it affects a program's *visibility*.

3.2 Hidden Dependencies

One of the strengths of visual programming compared to text-based systems is their ability to explicitly represent a program's *hidden dependencies*. For example, in traditional programming, code might refer to variables or other names that are defined in different sections of a program. Although modern programming environments allow users to quickly navigate to these definitions, visual languages can represent these connections directly. Common visual techniques for this include adding edges to a graph, or automatically positioning related elements near each other.

In block-based programming systems, few examples make *hidden dependencies* explicit. Although most editors support highlighting related blocks, such as a variable definition and its references, this functionality does not go beyond the equivalent support provided by text-based systems. Some recent block-based systems have experimented with spatially aligning concurrent code in parallel programs (Fronchetti *et al.*, 2023, Ritschel *et al.*, 2020). However, these capabilities are typically custom-tailored to a specific subset of parallelism and require substantial compromises in which other language features they can be combined with. These limitations might also explain why dependency visualizations are not found in any mainstream block-based systems to date.

Data-flow programming can provide stronger highlighting of *hidden dependencies* than blocks. In Figure 3, the dependencies between the logical sub-expressions are more prominently represented and easier to identify. A reader can easily follow the incoming edges of a node to identify all other nodes that it depends on, and the visual arrangement of the graph can further support this clarity.

Some data-flow programming systems support nodes with multiple outgoing edges that can be re-used in other parts of the program. For example, if the expression "touching edge" is used in another part of the program, it could have a second edge pointing to a different node, while a block-based program would use an entirely new expression block. The feature allows program creators to make dependencies even more explicit than would be possible in text- or block-based systems without this visual feature. Furthermore, lower redundancy also reduces the *error-proneness* of the system, as the effects of changes and inconsistencies are easier to identify and avoid.

3.3 Diffuseness and Consistency

In the CDN, *diffuseness* and *consistency* are related, but different, dimensions. *Diffuseness* refers to the number of syntactical or visual elements that a language defines. This could be the number of different blocks or nodes in a language, or the number of additional visual elements used to represent their composition. *Consistency* refers how easily the meaning of one syntactical or visual element can be derived after understanding others. A diffuse language might also have high *consistency* if all of its elements follow an easily recognizable pattern, and one with low *diffuseness* might also have low *consistency* if its few elements require distinct effort to interpret. Both block-based and data-flow languages can be at arbitrary points along both dimensions with respect to the number and types of nodes or blocks they offer. For our comparison, we focus on the way both modalities represent program composition.

Because blocks resemble text, they traditionally have low *diffuseness* and high *consistency*. Blocks can be composed by connecting new blocks vertically or horizontally. For each block, the possible connectors are identifiable based on the block's shape, and almost all blocks in commonly used systems fall into one of two categories: statements that have connectors at the top and bottom, or expressions that have a connector on the left and potentially holes for adding sub-expressions. By keeping the two block categories distinct, users can easily identify which blocks might be used at each point of a program. Most block-based languages lack *consistency* regarding how unfilled holes are handled. Missing sub-expressions usually make an the surrounding expression block impossible to interpret, which many block-based systems handle by skipping its evaluation and continuing with the next statement. Branching statement blocks, such as conditional blocks, on the other hand generally support empty branches without warning or error.

Data-flow graphs introduce an additional visual element: directed edges. These edges ensure that *hidden dependencies* are explicitly represented in the program's code. This additional visual element increases the *diffuseness* of the program representation. Nonetheless, compared to other visual representations, the diffuseness of the overall system remains low.

At a first glance, data-flow languages exhibit greater *consistency* than block-based languages in that they only provide one type of connector between nodes, and that connector always behaves identically. However, this is only the case in the context of side-effect-free programs where all data-flow is clearly structured and known in advance. As an example where this breaks down, assume that there exists a type of node that writes values to memory that it shares with other nodes. Ignoring this potential data-flow would lead to an inconsistent graph, as it only models some types of data-flow but not others. This representation could mislead users into thinking that nodes are independent even if they are not.

To improve the *consistency* of data-flow graphs in the presence of side-effects, tool designers might introduce edges that, directly or through an intermediate node, connect all the potentially dependent computations. This design choice can further benefit users by highlighting otherwise *hidden dependencies*. However, unlike other data-flow edges, these edges are only informative and cannot be edited by users directly. A representation that uses different types of edges increases the *diffuseness* of the language, and could grow unwieldy if even a modest number of nodes share the same memory access. As a result, data-flow languages can only represent side-effects by making substantial sacrifices in terms of *diffuseness* or *consistency*.

3.4 Secondary Notation and Visibility

In the CDN terminology, *secondary notation* describes how users can annotate their code with *informal* elements. These annotations typically aim to make code easier to read or understand, both for its author and for potential third-party readers and editors. Almost all traditional programming languages offer text-based comments as the primary feature for this purpose. However, most languages also allow programmers to improve the readability of their code in other ways. For example, the ability to use custom names for code fragments, or the freedom to format code with white space, also provides opportunities for informal program annotation.

Visual programming languages, including data-flow languages, typically support a different style of *secondary notation* than text. Despite their graphically rich overall design, few visual languages support free-form drawings or graphical elements to be used for informal code annotations. In fact, not all visual languages even support textual comments. One likely reason for this lack of annotation support are *visibility* concerns. If formal and informal notations are not visually distinct enough, users might confuse them. If this is the case, the *secondary notation* can even harm the readability of the surrounding code by hiding intended semantics of a program behind visual clutter.

The main tool for *secondary notation* that many visual languages provide to programmers is the freedom to arrange their programs. If it is deliberately chosen rather than coincidental, the proximity of nodes and the layout of edges are themselves a form of secondary notation. For example, a user might cluster related elements on a 2D canvas, or order edges to represent the order of execution, as shown in Figure 3. This custom arrangement allows users to highlight program fragments they consider important, and to improve the *visibility* of the overall program's semantics.

Historically, many block-based systems have neglected *secondary notation* altogether (Holwerda & Hermans, 2018). Some popular systems have only offered a single form of non-functional annotation: leaving unattached (i.e., "commented out") code fragments on the editor canvas for later use. Although most modern block-based systems also support textual comments that are attached to a specific block, this feature is often hidden behind drop-down menus, discouraging their use. Notably, block-based systems do not support the custom arrangement of code or even manual white space or line breaks, due to their strict use of block shapes. In programs like the one shown in Figure 2, this limitation can drastically lower the *visibility* of nested blocks. Block-based tools are therefore particularly limited in how they can represent nested mathematical or logical expressions.

3.5 Viscosity and Premature Commitment

Most of the cognitive dimensions discussed previously are primarily related to the comprehension of existing programs. However, the ease with which end-user programmers can create and edit new programs is also an important usability factor. When a programming tool requires *premature commitment*, it forces the programmer to anticipate later editing steps early in the development process. For example, a programmer might have to commit to a structure or decomposition style before having a sense of the scale of their final program. The *viscosity* of a program representation describes the effort that is required to make changes to existing code. Program editing effort can grow if programmers have to make repeated, or related, edits to different but intertwined parts of a program. Therefore, redundant information and coupling lead to an increased *viscosity*, even in cases where the extra information might benefit the readability of a program.

When a tool both requires *premature commitment* and has a high *viscosity*, it creates challenges for novices that can cause them to get stuck during editing. Data-flow programming tools often fall into this category because users have to decide how they will lay out their graph as soon as they start working on their program. For example, users might have difficulties determining the appropriate distance between nodes or fail to anticipate if a node will be used across different parts of the same program. Data-flow programming tools also introduce editing effort by requiring users to edit nodes and edges separately. Often, when a node is supposed to be replaced, users must first delete it and then recreate all connections within the graph. This limitation also increases the *error-proneness* of the editing process.

Block-based tools aim to simplify editing and avoid *premature commitment* by allowing users to construct program fragments independently before assembling them. Blocks are re-arranged automatically to make space for inserted fragments, eliminating the effort of planning a program's layout. Blocks can further be re-arranged by simply dragging them between connectors. Any re-arrangement of blocks also applies to all of their attached children and successor blocks, which can eliminate redundant editing steps. However, the *viscosity* of blocks increases substantially when editing the nesting of blocks. For example, supposed a user wants to change the association of operators in Figure 2. What might be as simple as moving a parenthesis in text-based code can require users to fully disassemble and then re-assemble most of the conditional expression in blocks. In fact, it might be preferable for users to re-create the expression from scratch as it is requires more *hard mental operations* or editing steps than making changes to the existing expression.

4 Two Design Alternatives for Block-based Nested Expressions

As outlined in Section 3, the choice between block-based programming and data-flow programming comes with usability trade-offs. Different styles of programs are better represented in different modalities: blocks are a good fit for stateful, imperative code, while data-flow graphs are better at representing nested expressions. However, real-world programs commonly use *both* of these styles to define *what* and *when* code gets executed. As a consequence, even on a case-by-case basis, it is not always possible to choose one modality that is a strictly better fit than the other.

In this section, we focus on a domain that illustrates the trade-offs between block-based and data-flow programming within a single use case: mobile robotics. We then present two design alternatives for programming environments that are building on previous block-based end-user systems. Both designs aim to make it easier to define complex nested expressions. One design solely makes changes to the visual representation of expressions in block-based programming. The other design is a hybrid between block-based and data-flow editing and allows users to define expressions in a data-flow graph while all imperative commands remain as blocks. To support our discussion, we continue referencing the CDN framework in the same style as in Section 3, *underlining and italicizing* CDN terms in the text.

4.1 Use Case: Mobile Robotics

Programming industrial robots traditionally requires substantial expertise, both in programming and in the robotics domain. Automated no-code programming approaches like demonstration-based learning (Argall *et al.*, 2009) and gesture recognition (Gadre *et al.*, 2019) has simplified small, linear robot tasks like pick-and-place routines, but more complex use cases still require programs written by humans. A range of robot

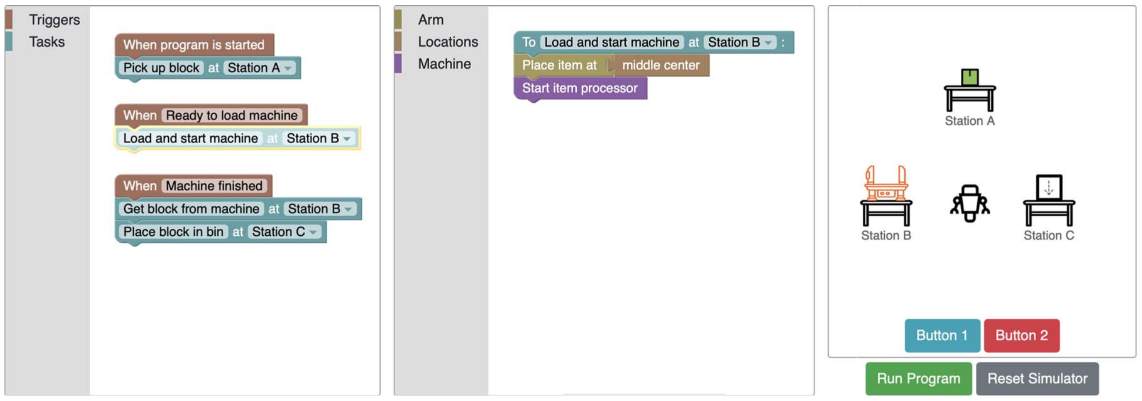

Triggers
Tasks
When program is started
Pick up block at Station A
When Ready to load machine
Load and start machine at Station B
When Machine finished
Get block from machine at Station B
Place block in bin at Station C
Arm
Locations
Machine
To Load and start machine at Station B :
Place item at middle center
Start item processor
Station A
Station B
Station C
Button 1
Button 2
Run Program
Reset Simulator

FIGURE 4. Programming in two canvases: users can select and assemble task and trigger blocks in the left canvas and edit the currently selected block (here: "Load and start machine") in the right canvas. A simple simulator on the very right allows users to test their programs.

programming environments specifically target end-users, usually targeting stationary robot arms (ABB Ltd, 2020, Ajaykumar *et al.*, 2021). As recent work has demonstrated, environments that combine block-based programming and demonstration-based programming are particularly effective in this task (Fronchetti *et al.*, 2023).

Recently, *mobile robot workers* have been identified as targets of interest for end-user programming as well (Burger *et al.*, 2020, Ritschel *et al.*, 2022). This type of robot consists of a mobile robot base with a mounted gripper arm that can move between a number of workstations, and is already commonly deployed in warehouses and factory floors. Previous work (Ritschel *et al.*, 2022) has focused on the challenges created by the size of mobile robot programs, which can become quite large, as the robot arm may perform a number of different tasks across several different work stations. However, a challenge that remains unaddressed is how to control the triggers for these workflows and interactions with the robot's environment through machine signals or human inputs in a novice-friendly way. Mobile robot tasks are highly stateful and are predominantly written in an imperative style, while triggers for event-based programming are typically represented as complex logical expressions. For this reason, we believe that this scenario is ideally suited to demonstrate the challenges of combining these two different programming styles in a single end-user environment.

4.2 Shared Design Approach: Editing in Multiple Programming Canvases

In this paper, we build on a programming system introduced by previous work (Ritschel *et al.*, 2022), which uses multiple programming canvases to modularize large programs into understandable chunks. Figure 4 illustrates our adaptation of this environment, which is the foundation for both design alternatives presented below. Users define the overall program flow in a canvas on the left side of the screen. Each task block that is added to this canvas is assigned to a single workstation and defined in its own *task editor* canvas, which is displayed on the right side of the screen once a task is selected. The task editor contains commands for instructing the robot pick and place items, load and unload machines, and turn on machines to process loaded items.

We extend the existing work by adding support for *triggers* to the previously presented programming system. As shown in Figure 4, triggers allow programs to be non-linear and controlled by environmental interactions, rather than a pre-determined sequence of robot commands. A robot controlled by triggers might react to external *signals*, such human inputs (e.g., a button being pressed), machine events (e.g., the processing of an item being finished), or environmental conditions (e.g., the robot gripper being empty). Programmers can combine multiple signals within a trigger through logical *combinators*, similar to how a programmer would compose a conditional expression as shown in Figures 2 and 3. The combination of triggers and tasks allows programmers to assemble complex, mostly autonomous workflows that can allow multi-tasking or reduce the need to re-program a robot.

Following the design principles introduced by the existing system, we divide our programming system into two canvases. Programmers can create tasks and triggers in the left programming canvas and assemble them into execution sequences. When a task or trigger is selected, its definition is displayed and can be edited in a separate canvas on the right. In the example shown in Figure 4, the "Load and activate machine" task is selected on the left side of the screen and its definition is displayed on the right side. The task editor is identical for our two design alternatives, which only differ in the way triggers are edited when they are selected.

We use a multi-canvas framework for our designs as it has several key benefits: First, the approach addresses decomposition, which is a challenge in an end-user context when programs become moderately large. The idea of programs being composed from multiple localized tasks naturally extends into a more complex composition of tasks and triggers. Second, the separate canvases allow users to edit two programs next to each other while making it easy to understand that they are not directly composable. Third, for the context of the experiment that we present in Section 5, the design allows us to create versions of the programming environment that are virtually identical except for how triggers are represented that we aim to evaluate.

4.3 Design Alternative 1: Adapting Block-based Programming

As we discussed in Section 3, block-based programming aims to imitate the flow of natural language text, which works well for commands. However, as with natural language, expressing nested clauses can grow unwieldy, and untangling them becomes a hard mental operation for readers. The trigger editor canvas on the right side of Figure 5 uses a design that aims to make block-based expressions easier-to-understand. Similar to a bullet point list, it spreads a nested expression over multiple lines. The jigsaw shapes that are typically used for statements in block-based languages represent the concatenation of sub-expressions. Deeper nesting

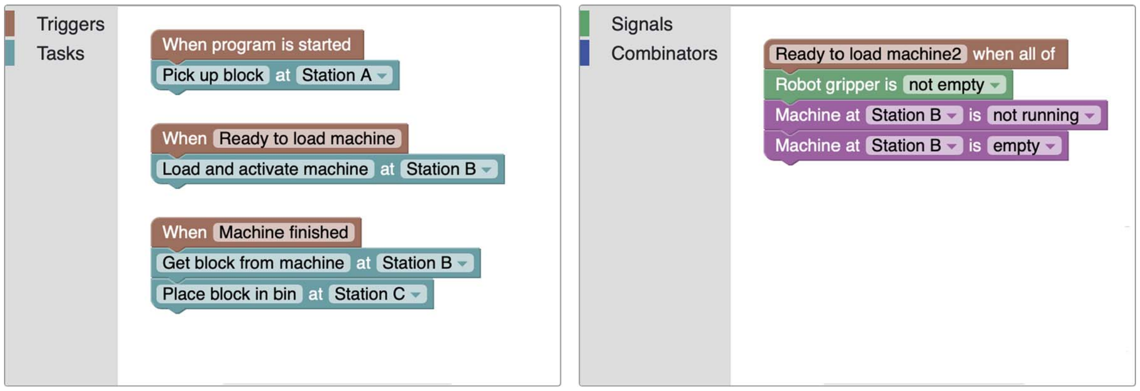

Triggers
Tasks
When program is started
Pick up block at Station A
When Ready to load machine
Load and activate machine at Station B
When Machine finished
Get block from machine at Station B
Place block in bin at Station C
Signals
Combinators
Ready to load machine2 when all of
Robot gripper is not empty
Machine at Station B is not running
Machine at Station B is empty

FIGURE 5. Purely block-based design alternative: Trigger expressions are represented as blocks, as shown in the right canvas. This design is consistent with the representation of tasks shown in Figure 4, but the the data-flow between the blocks can become harder to read as expressions grow.

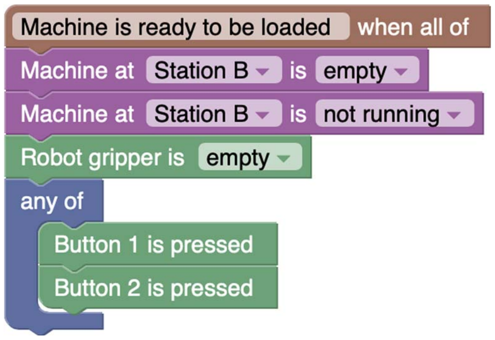

Machine is ready to be loaded when all of
Machine at Station B is empty
Machine at Station B is not running
Robot gripper is empty
any of
Button 1 is pressed
Button 2 is pressed

FIGURE 6. A more complex block-based trigger expression for a mobile robot, represented as blocks. The computational flow of the nested sub-expressions is not obvious for beginners.

levels are represented by inserting a new concatenation operator (here, "any of") and placing sub-expressions inside, similarly to how nested statements (e.g. the "move" block in Figure 2) are usually represented in block-based languages.

The trigger design shown in Figure 5 improves the *visibility* of the overall nesting structure of the program and the *role-expressiveness* of each block. This design is further from natural language than is customary for block-based programs, but it avoids using any special syntax such as parentheses. However, the design does not offer the same opportunity to use *secondary notation* as text-based languages, as users have no freedom to format their programs. Furthermore, the *hidden dependencies* between sub-expressions can make it hard for programmers to determine in which order blocks evaluated. This limitation becomes more apparent in the complex example trigger shown in Figure 6, which contains multiple levels of nesting, and in which blocks are not evaluated in linear order.

Another notable restriction of this block-based trigger representation is that expressions cannot be directly composed with imperative block-based code such as that for tasks in Figure 4. This might not be obvious for users, as the trigger blocks have the same visual style as task blocks. Other block-based systems that spread logical expressions across multiple lines also face this issue, and typically mitigate it by using different colour hues for statements and expressions (Bak *et al.*, 2018, Mattioli & Paternò, 2020). However, this visual distinction is not consistent with the usual design of blocks, where shapes alone determine composability, and colors can be mixed arbitrarily. Our prototype avoids this issue and keeps the overall system *consistent* by separating the two editing canvases and preventing attempts to mix and assemble imperative and expression-based code. Nonetheless, this limitation would affect other environments that try to use the same representation without a multi-canvas design.

4.4 Design Alternative 2: Creating A Hybrid Programming Environment

The design we presented in Section 4.3 aims to make programs easier to understand while maintaining a single consistent block-based programming style. However, this approach has some disadvantages compared to data-flow programming. In addition to retaining limitations of blocks, like the inability to add *secondary notation*, the system does not visualize the implicit data-flow. This can make it difficult for novices to determine the order in which trigger blocks are evaluated. For example, in Figure 6, as nested expressions determine the values of the surrounding operators, information flows right-to-left and bottom to top, which is inverse to imperative code. Data-flow programming makes the order of computations and the flow of information explicit, making it easier to reason about complex expressions. We therefore speculate that this programming style could be beneficial, even if it comes at the cost of a lower *consistency* across the overall system.

It is often possible to convert programs between the two modalities, as illustrated by Figure 7, which is a direct conversion of the block-based program in Figure 6. However, their user interfaces differ too much to seamlessly switch between them while programming. Therefore, we propose an design as shown in Figure 8, which builds on the previously presented multi-canvas approach. Instead of using blocks throughout the entire system, this design combines block-based and data-flow program representations and shows them side-by-side. For each part of a program, the system uses the representation that provides the most benefit for the user. This allows users to edit tasks as blocks and triggers as graphs, without switching between programming environments.

5 Evaluation

To evaluate the trade-offs of our block-based and hybrid editing design alternatives, we conducted a controlled experiment. The goal of this experiment was to gain further empirical insights into the learnability and usability trade-offs between the block-based and the hybrid environment beyond our analysis presented throughout Sections 3 and 4. We discuss our experimental design and procedure in this section and present our findings in Section 6.

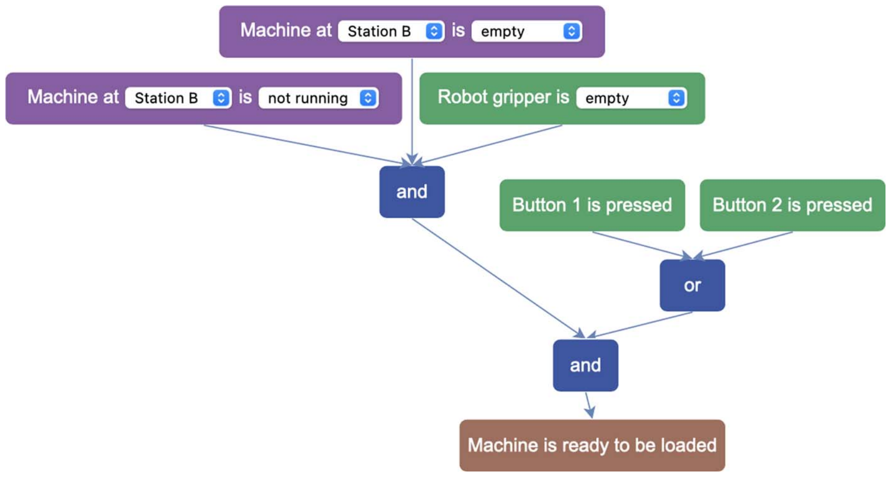

Machine at Station B is empty
Machine at Station B is not running
Robot gripper is empty
and
Button 1 is pressed
Button 2 is pressed
or
and
Machine is ready to be loaded

FIGURE 7. The same program as in Figure 6 represented as a data-flow graph. The edges make the data-flow explicit and programmers can manually group and arrange nodes.

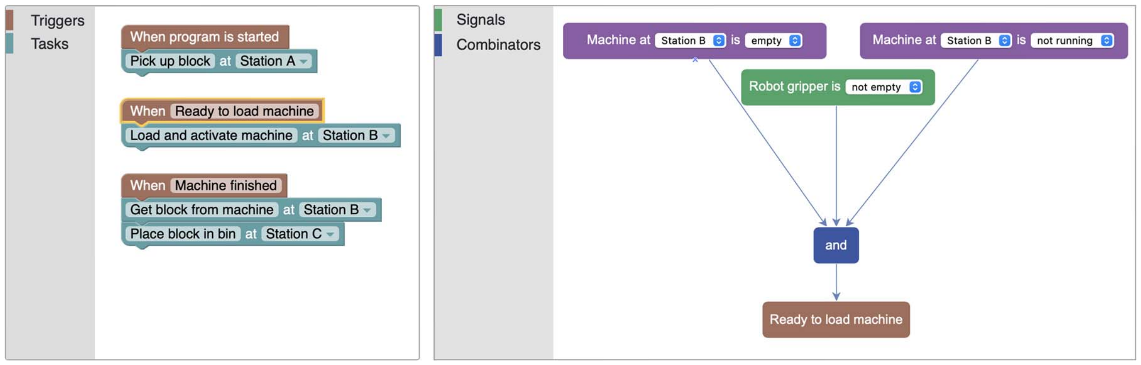

Triggers
Tasks
When program is started
Pick up block at Station A
When Ready to load machine
Load and activate machine at Station B
When Machine finished
Get block from machine at Station B
Place block in bin at Station C
Signals
Combinators
Machine at Station B is empty
Machine at Station B is not running
Robot gripper is not empty
and
Ready to load machine

FIGURE 8. Hybrid design alternative: triggers are represented as data-flow graph in the right canvas. This design is less consistent, but makes the data-flow between blocks more consistent and allows users to freely arrange graphs to support their understanding of the program.

5.1 Research Questions

We set out to answer the following research questions:

RQ1: *Does the introduction of triggers into a multi-canvas programming environment enable end-users to solve robotics tasks that involve environmental interactions?*

Through this research question, we aimed to validate the overall design approach for both programming environments that we outlined in Section 4.2. Previous work found that a programming environment with two separate programming canvases can help end-users structure complex programs and allow them to program mobile robots with high success (Ritschel *et al.*, 2020). Trigger-based programming extends the range of programs supported by this environment, and enables new practical use cases, as discussed in Section 4.2. However, although triggers have been used successfully in previous end-user programming system (Ur *et al.*, 2014, 2016), it remains unclear if end-users are able to use them effectively in the context of our system. The two approaches of trigger-based and multi-canvas programming might not work in tandem, and triggers may not be suitable for the domain we chose for the experiment. Furthermore, triggers increase the complexity of the overall system and might make it too difficult to learn for end-users, especially in the ad-hoc context of a study. We therefore aimed to use our results for RQ1 as a baseline of the performance we could expect from participants.

RQ2: *Does the representation of triggers affect the performance of participants as they read and write robot programs?*

Using our findings for RQ1, the primary goal of our study was to compare participant performance across the two prototype environments that we presented in Sections 4.3 and 4.4. By answering this research question, we set out to gain additional insight into the trade-offs between the two approaches and their empirical implications. In particular, we aimed to determine which environmental prototype is more effective at mitigating the limitations of block-based and data-flow programming in the chosen scenario of mobile robot programming.

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

In addition to collecting performance measurements, we also set out to investigate how our participants perceived their assigned environment. We asked participants whether they preferred one environment over the other, and whether they have any comments about their programming experience or preferences. By collecting their feedback and opinions, we hoped to better understand our findings for RQ1 and RQ2, and gather insights that might inform future designs or research on this topic.

5.2 Participants

We describe our power analysis, the method we used for recruiting, and the demographics of our participants below.

Power Analysis: We performed a power analysis (Cohen, 1992) to estimate the necessary number of participants to reach statistically meaningful results. Assuming $\alpha = 0.05$ and a medium effect size ($w = 0.3$), the power analysis determined that a minimum of 88 participants were necessary to reach a power of $1 - \beta = 0.8$ in a simple χ^2-test. Rounding conservatively, we therefore planned to recruit 100 participants.

Recruitment: Our study was advertised on Prolific (Palan & Schitter, 2018), an established online platform that pays participants a fixed compensation to participate in studies and other research activities The advertisement contained the study description, an estimate of the study duration, and details about the offered compensation, which was 12£.[1] We recruited English-speaking participants worldwide and in multiple waves to avoid over-representing specific countries or population groups. We discuss potential biases that might have been introduced by the Prolific platform and the recruitment process in Section 7.4.

Demographics: We include the data of 113 participants in the results that we present in Section 6. On average, these participants were 28.6 years old (SD=9.6); 44 (39%) identified as female while 68 (60%) identified as male. One participant opted not to disclose their gender. Participants listed 23 distinct countries of residence, and 78 (69%) identified themselves as employed at least part-time. The majority of our participants (69%) indicated a European country of residence, with Poland being the most commonly listed country for 22% of our participants. The remaining participants either indicated a country in North America (16%), another continent (9%), or did not provide any residency information.

5.3 Materials

For our study, we used the following programming systems and interactive tutorials.

Programming Systems: We implemented prototype versions for both programming systems designs presented in Section 4. Both use the same general overall set-up for mobile robot programs that previous work introduced and found to be effective at supporting end-users as they compose large programs (Ritschel *et al.*, 2022). The purely block-based programming system builds on multiple side-by-side instances of the framework *Blockly* (Fraser, 2013), while the hybrid environment connects Blockly with a graph editor that uses the framework *MXGraph* (JGraph Ltd, 2017). Both environments further feature an identical simulator that shows a simplified top-down view of a mobile robot and its environment. Programs can be immediately tested in the simulator, which provides feedback to participants through a graphical representation of the robot environment, as well as error messages when commands fail to be executed.

To keep the two programming environments under evaluation as similar as possible, the environments provide identical syntax and editing functionality for programming imperative robot tasks. For defining triggers, the trigger editors use equivalent syntax that only differs based on the used program representation. This simulator also provides the same functionality and a built-in testing mechanism that can determine whether a user has successfully solved a given task. Throughout the study, participants were able to reset the simulator and execute code as often as they wanted, encouraging them to experiment and test partial solutions.

Tutorials: We presented participants with three separate tutorial tasks. The first tutorial was designed to help with the pre-screening process by ensuring that potential participants show a minimum level of engagement with the presented systems. This tutorial asked potential participants to program a robot to pick and place a block, with detailed instructions how to do so. The tutorial was designed to take less than 5 minutes to complete, did not include any triggers, and was identical for all users.

The second and third tutorial gave participants a more thorough introduction to the programming environment they were assigned to use. The second tutorial introduced each participant to writing programs that span multiple workstations and are split across multiple tasks. The third and final tutorial introduced participants to triggers and how to program the robot to interact with machines. While the third tutorial was different for both participant groups due to the different methods used to program triggers, we designed both tutorial versions to be as similar as possible. Overall, we aimed for participants to spend less than 20 minutes total on the second and third tutorial combined.

5.4 Study Procedure

Because we recruited participants remotely through the Prolofic platform, we had no direct interaction with them other than addressing rare cases of technical difficulties. Instead, participants followed a fully automated study procedure that is outlined in Figure 9 and that we discuss in detail below.

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

Automated bots and inattentive participants are a known issue when using online panels and crowdworking platforms (Chmielewski & Kucker, 2020). We chose the Prolific platform instead of the more commonly used MTurk as the platform claims to carefully curate their participant pool and prevent automated or duplicate submissions (Palan & Schitter, 2018). To further ensure that participants engaged with the study, and have the technical capabilities to use the programming environment, we also included our first tutorial as part of the pre-screening process. This tutorial, described in Section 5.3, was intended to be easy to complete, but could not be finished by randomly clicking on the screen. Participants who successfully completed the first tutorial received a small compensation of 1£ and were added to the pool of pre-screened participants.

Randomization and Training: After pre-screened participants provided consent to participate in the study and have their data collected, we assigned each of them randomly to one of the two environmental prototypes. Participants were not aware of the alternative programming environment until they reached the comparative survey, which was the last part of our study. After participants were assigned a programming system, we presented them with the remaining two tutorials that we describe in Section 5.3. Participants had to finish both tutorials to proceed with the study. We did not enforce a strict time limit for this part of the study.

Programming Tasks: To answer RQ1 and RQ2, we aimed to present participants with programming tasks that feature both triggers and robot programs that are complex enough to test the programming abilities they had acquired during their training. We also wanted to test both the practical usability of blocks and data-flow graphs, and how they support the participants when

[1] Prolific is based in the United Kingdom and calculates payments in British pounds. Participants from other countries received equivalent amounts in their local currency.

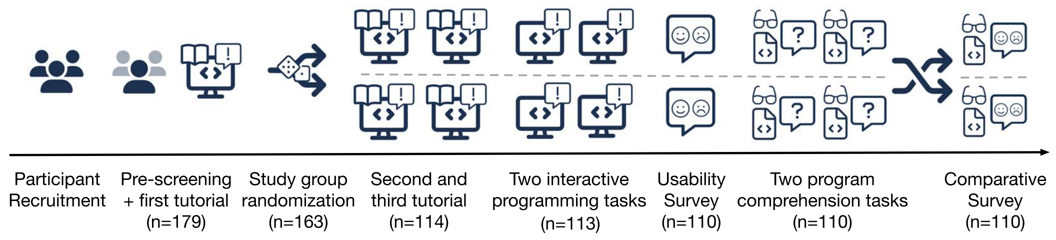

Participant Recruitment
Pre-screening + first tutorial (n=179)
Study group randomization (n=163)
Second and third tutorial (n=114)
Two interactive programming tasks (n=113)
Usability Survey (n=110)
Two program comprehension tasks (n=110)
Comparative Survey (n=110)

FIGURE 9. Procedure of the randomized experiment to evaluate the two prototype environments.

they need to reason about the activation of triggers. For this reason, we used tasks that were larger and more complex than those used in previous studies (Ritschel *et al.*, 2022, 2020, Weintrop *et al.*, 2017). As larger tasks take participants more time, we had to limit their number to ensure that participants would not get exhausted or drop out of the study. We eventually settled on two tasks in a fixed order that covered the concepts we set out to investigate:

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

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

After conducting several pilot experiments, we expected participants to be able to solve the two tasks in approximately 20 minutes per task. Some participants took longer than 20 minutes, but few of the successful participants exceeded 30 minutes. We therefore decided on 30 minutes as the cut-off time, at which point we would save the participant's work and redirect them to the next part of the study. Within this cut-off time, participants had an unlimited number of attempts at testing their solution in the simulator we provided to them.

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

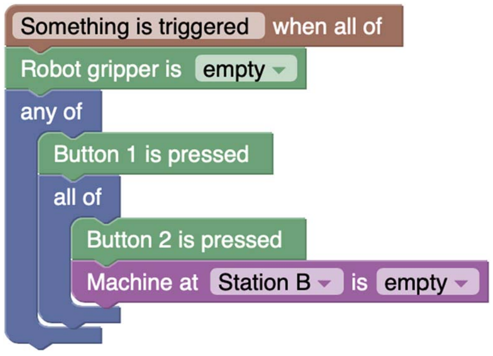


FIGURE 10. Block-based version of a more complex trigger with nested operators, used to test participants' program comprehension.

Program Comprehension Questions: The triggers that we asked participants to write during the two programming tasks were reasonably complex given the scope of the study, but real triggers might grow even larger. Due to the limited time participants were given to solve each task, we had to limit the number of items, workstations, and corresponding signals that we could ask them to consider. The code that participants had to write consisted of two to four signals and at most two operators in a single trigger, which is less than shown in Figures 6 and 7. Furthermore, only one of the two tasks required participants to read existing triggers.

To validate whether our findings, especially those for RQ2, hold for more complex triggers, we added two program comprehension tasks to the experiment. One task tested whether users could comprehend a trigger similar to the examples in Figures 6 and 7. Figure 10 shows the block-based version of the other task, which focused on whether users could understand a trigger with more nesting levels. We asked participants to select the correct natural language description of each program's behaviour out of four options in randomized order. Both of these triggers use signals and operators with which the participants are familiar, but do not combine them in a way that relates to any of the previously seen tasks or to each other. We intentionally chose this design so that participants would have to read the given code carefully without having to learn new concepts beyond those tested by the programming tasks.

Comparative Survey: As a final step in the experiment, we investigated how participants perceived the design of the environment to which they were *not* assigned. To answer this question, which is related to RQ3, we showed participants the same programs that they had just seen during the program comprehension component of the experiment, as well as the correct natural language description of the program. We then asked participants to rate this environment on three 5-point Likert items in comparison to the one they had seen up to this point. Like in the usability

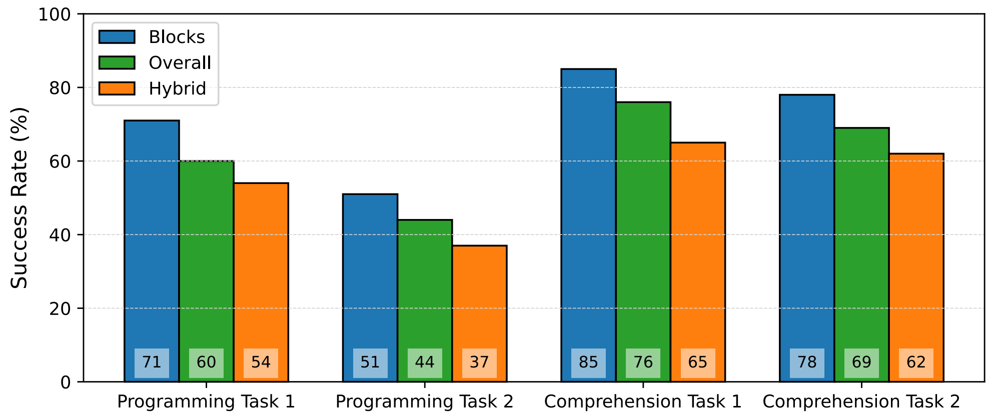


FIGURE 11. Completion rates for the two interactive programming tasks and the two comprehension tasks, both overall and for each participant group. Note that for the programming tasks, $n = 113$, and for the comprehension tasks, $n = 110$.

survey, we distinguished between the perceived learnability of the design and the ease of reading and writing programs. We also gave participants a final opportunity to provide written comments about their ratings.

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

6 Results

Following the study procedure outlined in Section 5.4, a total of 179 participants successfully completed the pre-screening and 163 of them agreed to participate in the randomized experiment. Of those participants, 49 decided to withdraw from the study before completing the additional programming tutorials. Unfortunately, we have no insight into why they decided to withdraw. It might have been because they struggled to follow the tutorials, or because the study required too much effort for the compensation we provided.

Participants who did complete the tutorials took 28.8 minutes on average (SD=10.2) to do so, with is slightly longer than anticipated. Upon closer inspection, the longer completion time can be attributed to the third tutorial, which introduced triggers. This tutorial took participants 16.2 minutes to complete on average (SD=6.0) while the first two tutorials stayed within their expected duration. The first tutorial took 5.1 minutes on average (SD=2.1) while the second tutorial took 7.6 minutes on average (SD=4.1). One participant was excluded for neither attempting any programming tasks nor providing any survey responses.

In the remainder of this Section, we present the data for the 113 participants who completed at least the programming component of the experiment. Of those participants, 3 did not complete the usability survey and program comprehension tasks that followed the interactive programming portion of the study. Consequently, we only include the data that we collected up to the point when those participants decided to exit the study.

When we refer to the evaluated groups, we will call them `Blocks` and `Hybrid` for brevity, as the only difference between them was the program representation used to define triggers. In the remainder of this section, we primarily report averages for descriptive statistics, but have also analyzed medians and standard deviations. For brevity, we only report those alternative metrics where we found them to be noteworthy, such as the number of program runs per participant. For all statistical tests, we use $p < 0.05$ as significance threshold.

6.1 Effectiveness of the Trigger-based Multi-Canvas Programming System (RQ1)

For each participant in the experiment, we recorded whether they finished each task and how long it took them if so. Figure 11 shows the success rates of the participants for both the programming and the comprehension tasks we asked them to complete. For the interactive programming tasks, 68 out of 113 participants (60%) were successful in completing Task 1, while 50 participants (44%) successfully completed Task 2. Participants who successfully completed Task 1 took 16.9 minutes on average (SD=5.80), and those who were successful on Task 2 took 18.6 minutes (SD=7.09).

We further recorded the participants' final programs as well as intermediate programs whenever they decided to run tests. The number of tests run ranges widely between participants for both tasks. On average, participants ran their programs 7.04 times (SD=5.83) for Task 1 and 8.43 times (SD=6.86) for Task 2, which includes the final, successful run if there was one. The median of runs is substantially lower at 4 runs for Task 1 and 6 runs for Task 2, illustrating that most participants only ran their code a few times while a few used more, with up to 34 attempts at most. We manually examined successful and unsuccessful participants' testing behaviour individually but did not find any notable differences.

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

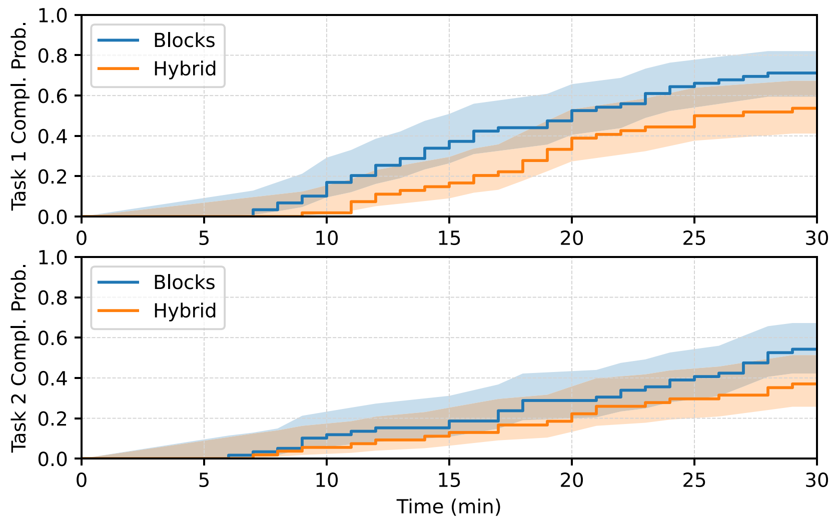


FIGURE 12. Completion probability over time for the two programming tasks. Shaded areas are 95% confidence intervals.

6.2 Performance Differences Between the Environments (RQ2)

In addition to analyzing the overall participant performance, we also analyzed the individual groups and compared their results. Of the 59 `Blocks` participants, 42 (71%) solved Task 1 and 30 (51%) solved Task 2. Of the 54 `Hybrid` participants, 29 (54%) solved Task 1 and 20 (37%) did the same for Task 2. The `Blocks` participants who successfully solved Task 1 took 16.0 minutes on average, compared to 18.1 minutes for `Hybrid` participants. For Task 2, successful `Blocks` participants took 18.7 minutes on average, while `Hybrid` participants took 18.5 minutes.

Participants were restricted in how long they could work on each task, and we suspect that this time restriction has influenced their success. Therefore, we performed survival analysis on our data, which allows us to consider both the success-over-time data of participants and their final success at the cut-off time (Goel *et al.*, 2010). The survival function $S(t)$ models the probability of a participant having *not yet successfully finished a task* at a given point in time. Since we are interested in the probability of participants having completed the task after a given time, we plot the respective completion curves $1 - S(t)$ in Figure 12.

We fit Kaplan–Meier estimators (Goel *et al.*, 2010) on the success rates and times of the participants. Figure 12 shows the resulting 95% confidence intervals for the completion probabilities. We also performed a log-rank test (Wellek, 1993) for each task to determine if the difference between the two curves is statistically significant. We found χ^2_{Task1}(df=1; N=113) = 4.92, p = 0.03 and χ^2_{Task2}(df=1; N=113) = 3.08, p = 0.08. Therefore, only the difference for Task 1 is statistically significant.

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

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

6.3 Participants' Perception of the Environments (RQ3)

In addition to measuring participant performance, we also asked them to rate the environment they used regarding its learnability and the readability and writability of programs. We did so with two sets of questions to separate the task editor and the trigger editor. We provided participants with three 5-point Likert items, which we code as 1 for a strongly negative rating and 5 for a strongly positive rating. The resulting matrix of responses is shown in the first two rows of Figure 13, with each stacked bar chart comparing the responses for the `Blocks` and `Hybrid` groups.

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

The collected usability ratings should not be interpreted on an absolute scale as users rate usability with a substantial bias towards higher ratings (Bangor *et al.*, 2008). We did not use a standardized questionnaire, and therefore only use the usability ratings to compare the two participant groups to each other. This type of comparison can also be affected by statistical biases, but the effects are typically small (De Winter & Dodou, 2010).

We performed a one-way MANOVA across all six Likert items to determine if there is a statistically significant difference in overall responses between the groups. This analysis finds F(df=6, 103) = 2.34, p = 0.04, indicating statistical significance and justifying a closer inspection of the individual items. Individual significance tests for the task items find: $F_{\text{TaskLearn}}$(df=1, 108) = 0.06, p = 0.81, F_{TaskRead}(df=1, 108) = 0.02, p = 0.88, and $F_{\text{TaskWrite}}$(df=1, 108) = 0.01, p

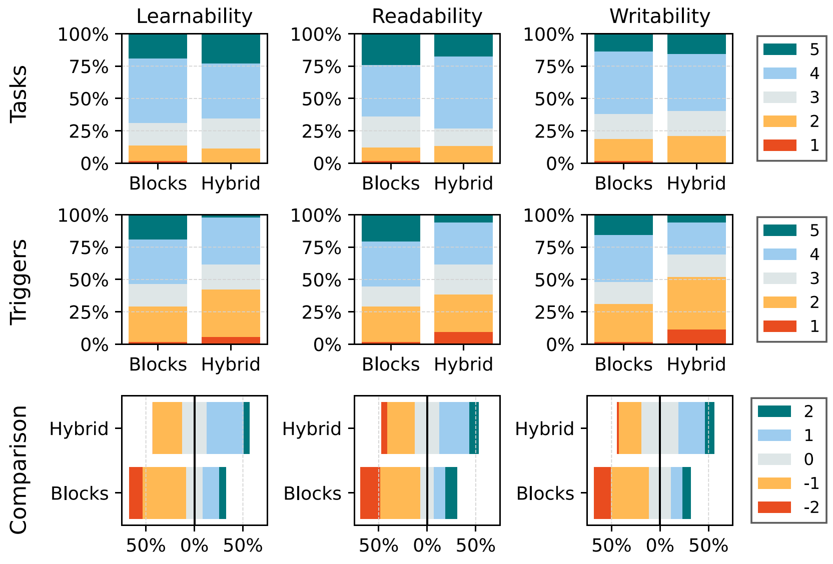


FIGURE 13. Distribution of responses for the Likert items of the survey, split by participant group (Blocks: N=58, Hybrid: N=52). Each chart compares the responses for one survey item, resulting in a 3x3 matrix. The first two rows are traditional stacked bar charts. For the bottom row, bars are aligned around neutral responses to indicate diverging preferences.

= 0.95. For the trigger items, they find $F_{\text{TrigLearn}}$(df=1, 108) = 5.58, p = 0.02, F_{TrigRead}(df=1, 108) = 5.00, p = 0.03, and $F_{\text{TrigWrite}}$(df=1, 108) = 8.13, p ¡ 0.01. Therefore, the differences between all three ratings of the trigger component are statistically significant, and those for the task items are not.

In addition to asking participants to rate their assigned environment, we also showed them triggers as they would be represented in the alternative environment. We asked them to indicate which of the representations they prefer, using the same three items (learnability, readability, and writability) as for the previous survey. We code responses on a scale from -2 (strong preference for the other environment) to 2 (strong preference for assigned environment). Figure 13 shows the results for these survey items in the bottom row, centered around a neutral rating.

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

7 Discussion

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

7.1 Success and Usability: Blocks Beat Hybrid

The results of our experiment are in line with the findings of previous studies (Ritschel *et al.*, 2022, Weintrop *et al.*, 2018) that an end-user friendly programming environment allows end-users to program mobile robots with little training. Our success rates are lower than those found by previous work on similar programming environments, which were 76% (Weintrop *et al.*, 2018) and 83% (Ritschel *et al.*, 2022) for their most complex tasks. However, even those tasks were substantially simpler than any of the tasks we asked participants to complete during our study, none of which the previously evaluated systems supported. Examining Figure 12, it is likely that a more generous time limit might have allowed more participants to finish the tasks. The lower success rate might also be attributed to the limited training that we were able to provide to participants throughout the study. This potentially indicates that the complexity of the presented system is approaching the limit of what participants can be expected to learn from scratch in a single session. A more critical interpretation might also attribute the lower success rate to triggers being an less suitable end-user friendly program representation than the previously evaluated design elements, which were mostly centered around imperative tasks. Considering all these factors, we consider a success rate of 40% still as a promising result that can motivate both future design improvements and further studies on the long-term usability of the presented systems.

Our findings also support our assumption that the trade-offs we presented in Section 4 are relevant for end-users and affect how they write code. We found that participants were faster and more successful writing programs in an environment that uses blocks as its only representation for code. This might be unsurprising, considering the additional effort required to learn two distinct editing styles. Some participants echoed this sentiment, writing about block-based trigger editing that *"[i]t's more similar to programming tasks"*, and about graph-based editing that *"[t]he lines and structure are relatively complex."* However, few comments mention issues regarding the different editing experiences between the two styles, with the exception of a few rare comments like *"I couldn't find how I could change the lines, so I ended up scrapping it and starting over."* Instead, several participants commented on data-flow graphs being difficult to read in general, criticizing *"random arrows [that] had two separate ends and were very confusing,"* and that *"it appears like there is a hierarchy even when there is not."*

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

7.2 (Why) Do End-users Prefer Graphs?

One finding of the survey does not fit with the other results: in a direct comparison, end-users prefer data-flow graphs over blocks. One reason for this observation could be that data-flow graphs look more interesting or visually appealing. A recent study on automated code generation has made similar observations, with users indicating that they prefer to use a code generator even though it had no measurable impact on their productivity (Vaithilingam *et al.*, 2022). Therefore, novelty alone (from the perspective of the user) might be a reason why our participants showed a preference for the data-flow graph representation.

Another potential reason for participants' preference could be that we only presented users with data-flow graphs such as the one shown in Figure 7, which we manually arranged to be easy to read for the given task. Compared to the block-based equivalent program, some users wrote that *"[t]he other one gets more clean and organized."* and that it *"looks much better and more intuitive than what I used."* We speculate that users compared our arrangement to the block-based program in Figure 6, which leaves no room for visual customization or aids. Some users also explicitly stated that the block-based code *"looks very messy"* and stated about graphs that *"in larger scale this looks better to program."*

7.3 Implications for Other Domains

As we outlined in Section 2, both block and data-flow programming have been adapted to a variety of end-user programming domains. However, to our knowledge there exist no hybrid systems like the one we presented here. Our findings can be interpreted as justifying this lack of hybrid systems. It appears that even minor adjustments to the visual presentation of block-based languages can achieve superior results despite theoretical limitations and caveats.

Remarkably, our results also provide initial evidence that end-users are better at understanding blocks than data-flow graphs even when each is used in isolation. We observed this in our program comprehension tasks, and the participant comments support this finding. This seems to contradict the more widespread commercial adoption of data-flow graphs compared to blocks. Although blocks are used in select commercial robotics applications (ABB Ltd, 2020), data-flow graphs have found wide-spread use across several other domains, particularly in game development (Bertolini, 2018, Sewell, 2015) and Internet-of-Things programming (Lekić & Gardašević, 2018). Our findings raise the question of whether exploring block-based approaches might bring benefits for end-users in these domains as well.

7.4 Limitations

Here we discuss limitations of the experiment that we conducted.

Participant Recruitment: We recruited participants using the Prolific online platform as it made it feasible to pick a large sample of end-users without being limited to a single geographical location or employer. As described in Section 5, we applied a variety of measures to reduce the risks of using a crowdworking platform. We also provide a range of demographic details about the participants in Section 6 that lead us to believe that we captured a wide range of participant backgrounds. Nonetheless, we are aware that the sample might not fully capture the large variety of potential end-users who might be asked to write robot programs in practice. In particular, although we aimed to recruit a worldwide sample of participants, we did not translate the programming environments we investigated, and only recruited English-speaking participants. Consequently, the vast majority of participants indicated that they live in Europe or North America, which might have introduced cultural biases, such as a preference for a specific spatial layout of the program code (Gevers & Lammertyn, 2005).

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

Task Selection: To ensure that we could ask the participants to complete reasonably complex programming tasks, we had to limit the number of tasks we asked them to solve. We could have selected a larger pool of tasks and randomly assigned them to participants, but this would have required splitting the participants into more groups and limited the statistical power of our results. As a result, we had to select only two programming and two comprehension tasks to represent a wider range of practical scenarios. We justified our selection in Section 5, but the external validity of our results is limited without further validation on different tasks.

Training and Time Limits: Due to the limited time available during the study, we had to restrict both the training we could provide to the participants and the time that they were allowed to work on each task. Participants might have performed better overall if they had received more extensive training, and more participants might have completed the tasks if given more time. Furthermore, since our task order was not randomized, participants might have also gotten more familiar with the system throughout our study. This might have lead to unintentional training effects, and a better performance on later tasks. However, these limitations affected both participant groups to a similar extent, leading us to believe that they would not have significantly altered the outcome of the study.

8 Conclusion

In this paper we have discussed the trade-offs between block-based and data-flow program representations in the context of imperative programming and evaluating nested expressions. We also presented and evaluated two design alternatives that aim to overcome the limitations of both representations, and found that a purely block-based programming environment with visual adaptations supports end-users better than one that is a hybrid of both modalities. Our evaluation provides initial evidence that, even when only considering the programming of nested expressions, blocks might still have a greater benefit for end-users than data-flow graphs. Though we performed our evaluation on the domain of robotics programming, we believe that our findings apply to many related areas of end-user programming that use graph-based languages, such as home and web automation, game development, or managing Internet-of-Things devices. We believe that these observations can inform the design of future end-user tools in those areas, as well as additional research on how to create new, novice-friendly, programming modalities and interface designs.

Acknowledgments

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.

Data Availability Statement

The data underlying this article are available in the Zenodo public online repository (https://doi.org/10.5281/zenodo.14232315).

References

ABB Ltd (2020) *Wizard Easy Programming*.

Ajaykumar, G., Steele, M., & Huang, C.-M. (2021) A survey on end-user robot programming. *ACM Computing Surveys (CSUR)*, **54**, 1–36.

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

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009) A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, **57**, 469–483. https://doi.org/10.1016/j.robot.2008.10.024.

Bak, N., Chang, B.-M., & Choi, K. (2018) Smart block: a visual programming environment for smartthings. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. **2**, pp. 32–37. IEEE.

Bangor, A., Kortum, P. T., and Miller, J. T. (2008) An empirical evaluation of the system usability scale. *International Journal of Human–Computer Interaction*, **24**(6): 574–594.

Banken, H., Meijer, E., & Gousios, G. (2018) Debugging data flows in reactive programs. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 752–763.

Barricelli, B. R., Cassano, F., Fogli, D., & Piccinno, A. (2019) End-user development, end-user programming and end-user software engineering: a systematic mapping study. *Journal of Systems and Software*, **149**, 101–137. https://doi.org/10.1016/j.jss.2018.11.041.

Bertolini, L. (2018) *Hands-On Game Development Without Coding: Create 2D and 3D Games With Visual Scripting in Unity*. Packt Publishing Ltd.

Biggs, G., & MacDonald, B. (2003) A survey of robot programming systems. In *Proceedings of the Australasian Conference on Robotics and Automation*, 1–3.

Brich, J., Walch, M., Rietzler, M., Weber, M., & Schaub, F. (2017) Exploring end user programming needs in home automation. *Transactions on Computer–Human Interaction (TOCHI)*, **24**, 1–35. https://doi.org/10.1145/3057858.

Burger, B., Maffettone, P. M., Gusev, V. V., Aitchison, C. M., Bai, Y., Wang, X., Li, X., Alston, B. M., Li, B., Clowes, R., Rankin, N., Harris, B., Sprick, R. S., & Cooper, A. I. (2020) A mobile robotic chemist. *Nature*, **583**, 237–241. https://doi.org/10.1038/s41586-020-2442-2.

Burnett, M., Hossli, R., Pulliam, T., VanVoorst, B., & Yang, X. (1994) Toward visual programming languages for steering scientific computations. *IEEE Computing in Science and Engineering*, **1**, 44–62. https://doi.org/10.1109/99.338768.

Calloni, B. A., & Bagert, D. J. (1994) Iconic programming in baccii vs. textual programming: which is a better learning environment? In *Proceedings of the Symposium on Computer Science Education (SIGCSE)*, pp. 188–192.

Chmielewski, M., & Kucker, S. C. (2020) An mturk crisis? Shifts in data quality and the impact on study results. *Social Psychological and Personality Science*, **11**, 464–473. https://doi.org/10.1177/1948550619875149.

Cohen, J. (1992) Statistical power analysis. *Current Directions in Psychological Science*, **1**, 98–101. https://doi.org/10.1111/1467-8721.ep10768783.

TIOBE The Software Quality Company (2023) TIOBE index. https://www.Tiobe.Com/tiobe-index.

Conway, M. J. (1998) *Alice: Easy-to-Learn Three-Dimensional Scripting for Novices*. PhD thesis, University of Virginia.

De Winter, J. C., & Dodou, D. (2010) Five-point likert items: t test versus Mann–Whitney–Wilcoxon. *Practical Assessment, Research, and Evaluation*, **15**, 1–12.

Desolda, G., Ardito, C., & Matera, M. (2017) Empowering end users to customize their smart environments: model, composition paradigms, and domain-specific tools. *ACM Transactions on Computer–Human Interaction (TOCHI)*, **24**, 1–52. https://doi.org/10.1145/3057859.

Dittrich, Y., Burnett, M., Morch, A., & Redmiles, D. (2013) *End-User Development*. Springer.

Dorn, B. J. (2010) *A Case-Based Approach for Supporting the Informal Computing Education of End-User Programmers*. PhD thesis, Georgia Institute of Technology.

Fagan, J. C. (2007) Mashing up multiple web feeds using Yahoo! Pipes. *Computers in Libraries*, **27**, 10–17.

Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., & Kwan, I. (2013) An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **22**, 1–41. https://doi.org/10.1145/2430545.2430551.

Fraser, N. (2015) Ten things we've learned from blockly. In *Blocks and Beyond Workshop (B&B)*, 49–50.

Fraser, N., *et al.* (2013) *Blockly: A Visual Programming Editor*.

Fronchetti, F., Ritschel, N., Schorr, L., Barfield, C., Chang, G., Spinola, R., Holmes, R., & Shepherd, D. C. (2023) Block-based programming for two-armed robots: a comparative study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 494–505. IEEE Computer Society.

Gadre, S. Y., Rosen, E., Chien, G., Phillips, E., Tellex, S., & Konidaris, G. (2019) End-user robot programming using mixed reality. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2707–2713.

Gevers, W., & Lammertyn, J. (2005) The hunt for snarc. *Psychology Science*, **47**, 10–21.

Ghiani, G., Manca, M., Paternò, F., & Santoro, C. (2016) End-user personalization of context-dependent applications in aal scenarios. In *Proceedings of the International Conference on Human–Computer Interaction With Mobile Devices and Services (MOBILEHCI)*, 1081–1084.

Goel, M. K., Khanna, P., & Kishore, J. (2010) Understanding survival analysis: Kaplan–Meier estimate. *International Journal of Ayurveda Research*, **1**, 274–278. https://doi.org/10.4103/0974-7788.76794.

Gonçalves, M. C., Lara, O. N., de Bettio, R. W., & Freire, A. P. (2021) End-user development of smart home rules using block-based programming: a comparative usability evaluation with programmers and non-programmers. *Behaviour and Information Technology*, **40**, 974–996. https://doi.org/10.1080/0144929X.2021.1921028.

Green, T. R. (1989) Cognitive dimensions of notations. *People and Computers V*, 443–460.

Green, T. R. G., & Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, **7**, 131–174. https://doi.org/10.1006/jvlc.1996.0009.

Harvey, B., Garcia, D. D., Barnes, T., Titterton, N., Armendariz, D., Segars, L., Lemon, E., Morris, S., & Paley, J. (2013) Snap! (build your own blocks). In *Proceeding of the Technical Symposium on Computer Science Education (SIGCSE)*, 759–759.

Hempel, B., Lubin, J., Lu, G., & Chugh, R. (2018) Deuce: a lightweight user interface for structured editing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 654–664.

Holwerda, R. and Hermans, F. (2018) A usability analysis of blocks-based programming editors using cognitive dimensions. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 217–225. IEEE.

JGraph Ltd. (2017) *mxgraph*. https://jgraph.github.io/mxgraph/.

Johnston, W. M., Hanna, J. P., & Millar, R. J. (2004) Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, **36**, 1–34. https://doi.org/10.1145/1013208.1013209.

Kelleher, C., Pausch, R., & Kiesler, S. (2007) Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 1455–1464.

Kelly, J. L., Lochbaum, C., & Vyssotsky, V. A. (1961) A block diagram compiler. *The Bell System Technical Journal*, **40**, 669–676. https://doi.org/10.1002/j.1538-7305.1961.tb03236.x.

Kim, C., Yuan, J., Vasconcelos, L., Shin, M., & Hill, R. B. (2018) Debugging during block-based programming. *Instructional Science*, **46**, 767–787. https://doi.org/10.1007/s11251-018-9453-5.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., *et al.* (2011) The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, **43**, 1–44. https://doi.org/10.1145/1922649.1922658.

Lekić, M., & Gardašević, G. (2018) Iot sensor integration to node-red platform. In *Proceedings of the International Symposium Infoteh-Jahorina (Infoteh)*, 1–5.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010) The scratch programming language and environment. *Transactions on Computing Education (TOCE)*, **10**, 1–15. https://doi.org/10.1145/1868358.1868363.

Mattioli, A., & Paternò, F. (2020) A visual environment for end-user creation of iot customization rules with recommendation support. In *Proceedings of the 2020 International Conference on Advanced Visual Interfaces*, pp. 1–5.

Morrison, J. P. (1994) Flow-based programming. In *Proceedings of the International Workshop on Software Engineering for Parallel and Distributed Systems*, 25–29.

Morrison, J. P. (2010) *Flow-Based Programming: A New Approach to Application Development*. CreateSpace.

Nardi, B. A., Miller, J. R.*et al.* (1990) *The Spreadsheet Interface: A Basis for End User Programming*, vol. 10. Hewlett-Packard Laboratories.

US Department of Labor (2021) *Occupational Outlook Handbook*.

Ovadia, S. (2014) Automate the internet with if this then that (IFTTT). *Behavioral & Social Sciences Librarian*, **33**, 208–211. https://doi.org/10.1080/01639269.2014.964593.

Palan, S., & Schitter, C. (2018) Prolific. ac—a subject pool for online experiments. *Journal of Behavioral and Experimental Finance*, **17**, 22–27. https://doi.org/10.1016/j.jbef.2017.12.004.

Ritschel, N., Fronchetti, F., Holmes, R., Garcia, R., & Shepherd, D. C. (2022) Can guided decomposition help end-users write larger block-based programs? A mobile robot experiment. *Proceedings of the ACM on Programming Languages (PACMPL)*, **6**, 233–258. https://doi.org/10.1145/3563296.

Ritschel, N., Holmes, R., Garcia, R., Fronchetti, F., & Shepherd, D. (2025) *Replication Package: Block-Based or Graph-Based? Why Not Both? Designing a Hybrid Programming Environment for End-Users*.

Ritschel, N., Kovalenko, V., Holmes, R., Garcia, R., & Shepherd, D. C. (2020) Comparing block-based programming models for two-armed robots. *Transactions on Software Engineering (TSE)*.

Ritschel, N., Sawant, A. A., Weintrop, D., Holmes, R., Bacchelli, A., Garcia, R., KR, C., Mandal, A., Francis, P., and Shepherd, D. C. Training industrial end-user programmers with interactive tutorials. *Software: Practice and Experience (SPE)*, **53**(3): 729–747.

Sewell, B. (2015) *Blueprints Visual Scripting for Unreal Engine*. Packt Publishing Ltd.

Ur, B., McManus, E., Pak Yong Ho, M., & Littman, M. L. (2014) Practical trigger-action programming in the smart home. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pp. 803–812.

Ur, B., Pak Yong Ho, M., Brawner, S., Lee, J., Mennicken, S., Picard, N., Schulze, D., & Littman, M. L. (2016) Trigger-action programming in the wild: an analysis of 200,000 IFTTT recipes. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pp. 3227–3231.

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022) Expectation vs. experience: evaluating the usability of code generation tools powered by large language models. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pp. 1–7.

Voinov, P., Rigger, M., & Su, Z. (2022) Forest: structural code editing with multiple cursors. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pp. 137–152.

Vose, G. M. (1986) Labview: laboratory virtual instrument engineering workbench. *Byte*, **11**, 84–92.

Weintrop, D. (2019) Block-based programming in computer science education. *Communications of the ACM*, **62**, 22–25. https://doi.org/10.1145/3341221.

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

Weintrop, D., & Holbert, N. (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.

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

Wellek, S. (1993) A log-rank test for equivalence of two survivor functions. *Biometrics*, **49**, 877–881. https://doi.org/10.2307/2532208.

Whitley, K. N. (1997) Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, **8**, 109–142. https://doi.org/10.1006/jvlc.1996.0030.

Whitley, K. N., Novick, L. R., & Fisher, D. (2006) Evidence in favor of visual representation for the dataflow paradigm: an experiment testing labview's comprehensibility. *International Journal of Human–Computer Studies*, **64**, 281–303. https://doi.org/10.1016/j.ijhcs.2005.06.005.

Wiedenbeck, S., Zila, P. L., & McConnell, D. S. (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.

Wolber, D., Abelson, H., Spertus, E., & Looney, L. (2011) *App Inventor*. O'Reilly Media, Inc.