Really Pushing My Buttons: Affordances in Block Interfaces

Austin Cory Bart, Luke Gusukuma, and Dennis Kafura
Computer Science
Virginia Tech
Blacksburg, Virginia 24060
Email: acbart@vt.edu, lukesg08@vt.edu, kafura@cs.vt.edu

***Abstract*—Block-based languages are a useful scaffold for novice programmers to create syntactically correct code. However, block environments that attempt to represent feature-rich languages, such as Python or Java, face serious usability and pedagogical challenges. How can the affordances and controls of a block be exposed, without over-complicating the block's interface and obscuring its meaning to the user? In this paper, we visually explore a number of interface mechanisms to reconfigure blocks. We outline research questions to compare and contrast the advantages and disadvantages of these mechanisms, and call on the community to pursue these trade-offs further.**

I. Introduction

Block-based interfaces are a powerful way to introduce students to coding, with a number of advantages [1]. These interfaces avoid issues with syntax and make the structural nature of code more obvious. With proper construction, it becomes considerably more difficult to make incorrect programs, even for beginners. Block environments make the collection of blocks available visually, allowing students to rely on recognition instead of recall [2].

However, there are advantages to conventional text-based languages. Text is a very concise medium that can express a large number of syntactic structures in a minimal amount of space. No special environments or graphical windows are required to edit text-based languages. Although it is not always obvious what language features are available to use, a knowledgeable programmer can immediately take advantage of them simply by writing text.

Although some block-based editors feature their own, blocks-first editor, there has been interest in environments that match blocks to text-based languages. For these environments, blocks are simply an interface for editing a textual language such as Java [3], Python [4], or C++ [5]. Many of these environments have pedagogical goals, to help novice learners better understand the structural nature of programs [4], [6]. In other cases, the environment presents the dual interface as beneficial to even advanced programmers [7].

Block interfaces that seek to represent complex languages, with rich syntactical features, face serious usability issues. Blocks need to be represented in a way that highlights the structural nature of programs. However, the space taken up by the interface should not distract from the program itself. Advanced language features need to be available without overwhelming the users' cognitive load. If block environments could expose better affordances for advanced language features, they could gain more of the benefits of textual interfaces.

In this position paper, we consider different ways that a user can interact with blocks in order to manipulate and extend the block to take advantage of language features. We conclude with a call to action for more studies that evaluate the trade-offs among these different ways. The target audiences of the paper are developers and researchers interested in designing and evaluating new block interfaces, particularly for environments that use a block interface to represent a conventional text-based language.

II. Block Interfaces

Block interfaces typically have a few key, recurring elements. Consider the `REPEAT` block shown in Figure 1, from Google Blockly [1]. The text on the block indicates the primary purpose of the block, and its buttons give controls for manipulating the block. The shapes of the block and its connectors are used to show that blocks can be placed inside of it, expressions snapped onto the side, or the block itself can be wedged inside of other blocks. Finally, the color of the block gives information about its general classification, since related blocks will be similarly colored.

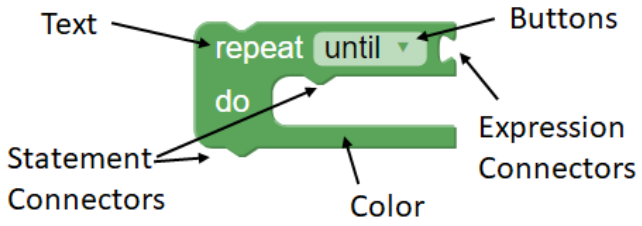

Fig. 1. An Example Block Annotated with its Information and Affordances

Prior research on blocks has identified other mechanisms for presenting semantic information to the user. Vasek studied how the shape of block connectors could be dynamically manipulated in order to reveal type information about the block [8]. OpenBlocks uses similar polymorphic block connectors that detect the input block type and adjust accordingly to present additional block information [9]. Poole developed a theoretical block interface that used color gradients to convey

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

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

type information about specific blocks [10]. Although there have been studies analyzing how blocks can be used to present information to the user, there has been little research about what affordances blocks should use to provide control over the blocks' features. Fraser published one of the few studies briefly comparing and contrasting various block affordances [11].

III. Representing Blocks

Whether writing in a block, text, or dual interface language, a program can typically be represented as an Abstract Syntax Tree. An *Abstract Syntax Tree* is a hierarchical representation of a program that models each element of the program in child-parent relationships. Block-based environments, in essence, are simply providing a mappable visualization of the programs' AST. In some cases, this mapping may be one-to-one; in others, abridgments or expansions of the original AST may be in effect. What does a block represent in relationship to the AST? Is each block an AST node? A pattern of nodes? A configuration of a node? These subtle differences have important implications for users of the environment and serious pedagogical ramifications for learning the language.

```
2 @blueprint.route('save/', **settings)
3 @blueprint.route('save', method='GET')
4 def save(url, user=None, *args, **options):
5     pass
```

Fig. 2. A Complex Python Function Declaration

Many languages have great flexibility and configurability reflected in their AST nodes. Consider the Python programming language. Both `IF` and `FOR` nodes have optional `ELSE` bodies – although the former is a commonly used construct, the latter is almost an esoteric language feature. `IF` statements and `TRY` blocks can have zero-or-more of their `ELIF` and `EXCEPT` clauses. Figure 2 shows how complex function declarations and invocations can get: zero-or-more parameters/arguments, default parameter values, keyword and varadic arguments, and optionally unlimited decorators. Any block language representing the Python language must decide how to expose these AST options to the user, or which features to not support.

IV. Block Affordances

In this section, we detail various control mechanisms for reconfiguring block interfaces. Each mechanism is described, visualized, and its advantages and disadvantages are examined.

A. Buttons on Blocks

Fig. 3. A list block controlled by buttons directly on the block

Perhaps the most straightforward way to expose features of a block is to simply put buttons on the blocks' interface. In Figure 3, plus and minus buttons are placed on the `LIST` block to allow the user to extend the number of slots available on the block. Many kinds of buttons can be used, such as drop-down menus for a set of options or check boxes for boolean toggles.

Advantages:

- Immediately exposes all of the features of the block.
- Well-design interfaces can be used quickly and easily.

Disadvantages:

- Additional buttons on a block take up space on the block, increasing its size. For particularly complex blocks, this cost can escalate quickly.
- Users need to mentally filter out the buttons from block elements meant to present information about the block (e.g., the text or color), which increases cognitive load.
- Buttons must be designed to be understandable, so that students do not need to refer to documentation.

B. Mutators

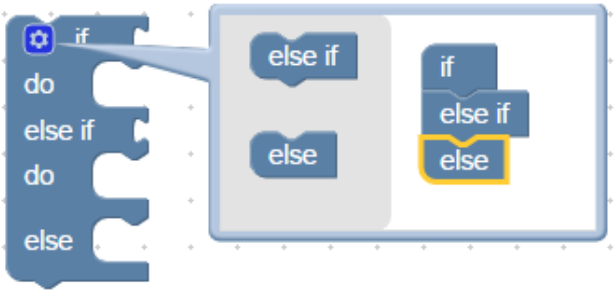

Fig. 4. An `IF` block controlled by a mutator

Mutators are an innovation of the Blockly interface library for encapsulating controls of a block in a single button. When clicked, that button opens up a larger submenu with additional options for controlling the block. Figure 4 shows how an `IF` block can be extended with `ELIF` and `ELSE` blocks.

Advantages:

- Because the controls are hidden, the impact on users' cognitive load is potentially reduced for complex blocks.
- The mutators' menu can have unbounded space.

Disadvantages:

- The mutator button may not be a clear mechanism for adjusting the block's features. In our experience, many students fail to realize that the default gear icon is clickable at all.
- The mutator button can still interfere with the users' comprehension of the block. When working with a student, one of the author's students described an `IF` block like the one shown in figure 4: "I think of this as a "DO block", because it says DO here." The student ignored everything on the line with the mutator button, resulting in a serious misconception about the blocks' features.
- The interface within the mutator's menu must still be carefully designed to be discoverable and usable.

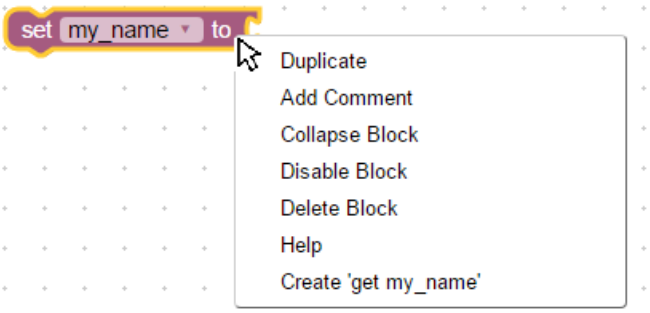

Fig. 5. An assignment block controlled by a right-click context menu

C. Right Click

Many conventional user interface systems use "context menus" activated by some secondary mouse click (e.g., right clicking, command clicking) in order to provide additional options. The Blockly interface library places many additional block controls (e.g., adding comments, code folding) in such menus, as shown in Figure 5.

Advantages:

- Context menus are a familiar user interface mechanism.
- The context menu can contain many mechanisms, efficiently using space.

Disadvantages:

- It may not be obvious that the context menu is available.
- The menu is tuned at providing clickable actions – not presenting information or other kinds of affordances.

D. Detect and React

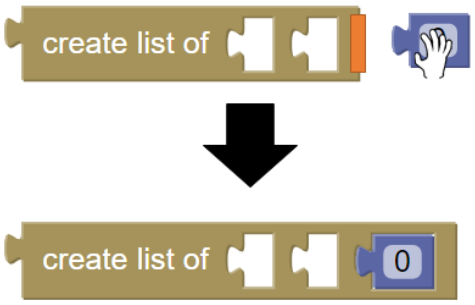

Fig. 6. A list block that expands when it detects the presence of a new potential element

Figure 6 demonstrates how a block can react to the users' actions and reconfigure itself dynamically, in this case adding a slot for a new item in a list. This specific mechanism is used in several environments, such as OpenBlocks [9]. Another example is a function invocation block with an optional return; the left-side plug (representing its ability to return a value) can optionally appear when brought towards a receptive block.

Advantages:

- Conserves space without the use of buttons.
- Can be used to immediately expose the result of taking the action, without actually commiting to the action.

Disadvantages:

- Not always a visible option to the user.
- Live previews from this feature can be jarring visually.
- Does not necessarily cover all possible affordance cases; in the `LIST` example above, there's no way to add elements without first creating the new value.
- More complex to implement.

E. Multiple Static Blocks

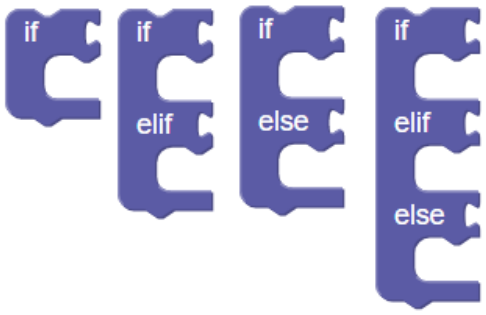

Fig. 7. Four different `IF` statements available to the user

Instead of blocks that are controllable via buttons, another option is to side-step the issue by simply providing static blocks with all desirable configurations. Figure 7 shows how major variations on an `IF` block can be made available to the user. Although not every possible configuration of an `IF` is made available as a single block, a useful subset can be provided. In some languages, the space of required blocks can be reduced further by relying on equivalent alternatives. For instance, `ELIF` blocks in Python are equivalent to nesting an `IF` block inside of the `ELSE` body of another `IF` block.

Advantages:

- Conserves space on the blocks and keeps them readable.

Disadvantages:

- Clutters the available block menu with the configurations.
- Students will be immediately exposed to many options when navigating the block menu, forcing them to choose the correct configuration early.
- Not all configurations are possible without providing a multitude of blocks.

F. Combo Blocks

Fig. 8. Component blocks to construct complex conditionals

Some complex blocks can have their components decomposed into separate blocks. Figure 8 shows 3 blocks: an `IF` block, an `ELSE IF` block, and an `ELSE` block. These three blocks can then be combined as needed. Incorrect combination

of blocks (e.g., two ELSE blocks) can be dynamically updated with an error mark to make it clear that the configuration is invalid. This is somewhat similar to how Snap! allows any expression block to be transformed into lambda expressions by encircling the expression with gray ring blocks [12].

Advantages:

- Reduces the number and size of blocks in the menu compared to providing multiple complete configurations.
- Blocks conserve space and have minimal cognitive load.

Disadvantages:

- Incorrect block configurations are possible; this may not be as direct at conveying correct code structures.

G. Keyboard-entry Blocks

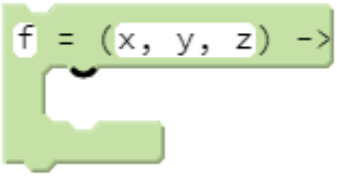

Fig. 9. Component blocks to constructor complex conditionals

More advanced syntactic features can also be exposed by embedding editable text boxes in the block. The parameters of the function declaration in Figure 9 (as used in the PencilCode editor [6]) are in an editable text box, making it very convenient to add new parameters. In some ways, this exposes the best flexibility of a text-based interface while retaining much of the power of blocks. BlockPy uses this mechanism in its Mutual Language Translation to convert unparseable textual code into editable, raw text blocks of code [4]. The GreenFoot Stride editor uses this at the core of the entire environment, allowing all blocks to be constructed by typing text [13].

Advantages:

- Extremely powerful and flexible interface that can expose any number of features.
- Maintains some structure surrounding the input.

Disadvantages:

- Difficult for the user to understand how to use.
- Breaks the blocks user interface convention, which some users may not be prepared for.

V. Research Studies

The possible mechanisms presented here have various advantages and disadvantages. The next step for the research community is to conduct studies to determine the trade-offs of the methods of extending and controlling blocks. We propose the following research questions as a starting point:

- How can buttons, mutators, and other controls be arranged on the block to minimize cognitive load?
- What visual cues can be used to make less visible affordances understandable to the user?
- Which of these mechanisms are most effective at exposing the true language features beneath?
- How quickly can students be exposed to different block configurations, and in what order should specific language features be exposed?
- Is there a critical point, in terms of program length or complexity, where blocks are no longer tenable as a programmatic interface?
- What affordances from more symbolic visual programming language blocks can be leveraged to address some of the gaps noted in text-based language blocks?

There are both Human-computer Interaction design issues, and questions about the impact on students' learning. Different environments, with different user bases and endgoals, may have need of different mechanisms, requiring study in many different contexts. Regardless, we expect that there are many generalizable lessons that could be learned.

VI. Conclusion

In this paper, we have presented a number of mechanisms that can be used to expose advanced language features of blocks. We have explained how these mechanisms relate to theoretical models of programs, and connected them to both usability and pedagogical research questions. We hope that the block-based research and design communities will use these as a starting point to explore these issues further and better understand the trade-offs inherent in designing blocks.

This work is supported in part by National Science Foundation grants DUE 1624320, DUE 1444094, and DGE 0822220.

References

[1] N. C. Brown, J. Mönig, A. Bau, and D. Weintrop, "Panel: Future directions of block-based programming," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016.

[2] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: blocks and beyond," *Communications of the ACM*, 2017.

[3] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-cs introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015.

[4] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura, "Blockpy: An open access data-science environment for introductory programmers," *IEEE Computer 2017*.

[5] J. Protzenko, "Pushing blocks all the way to c++," in *Blocks and Beyond Workshop, 2015 IEEE*.

[6] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*, 2015.

[7] J. Monig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in gp," in *Blocks and Beyond Workshop, 2015 IEEE*.

[8] M. Vasek, "Representing expressive types in blocks programming languages," 2012.

[9] R. V. Roque, "Openblocks: an extendable framework for graphical block programming systems," Ph.D. dissertation, MIT, 2007.

[10] M. Poole, "Design of a blocks-based environment for introductory programming in python," in *Blocks and Beyond Workshop, 2015 IEEE*.

[11] N. Fraser, "Ten things we've learned from blockly," in *Proceedings of the 2015 IEEE Blocks and Beyond Workshop*.

[12] B. Harvey and J. Mönig, "Lambda in blocks languages: Lessons learned," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*.

[13] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, 2015.