OpenVP: A Customizable Visual Programming Environment for Robotics Applications

Andrew Schoen
University of Wisconsin-Madison
Madison, Wisconsin, USA
schoen@cs.wisc.edu

Bilge Mutlu
University of Wisconsin-Madison
Madison, Wisconsin, USA
bilge@cs.wisc.edu

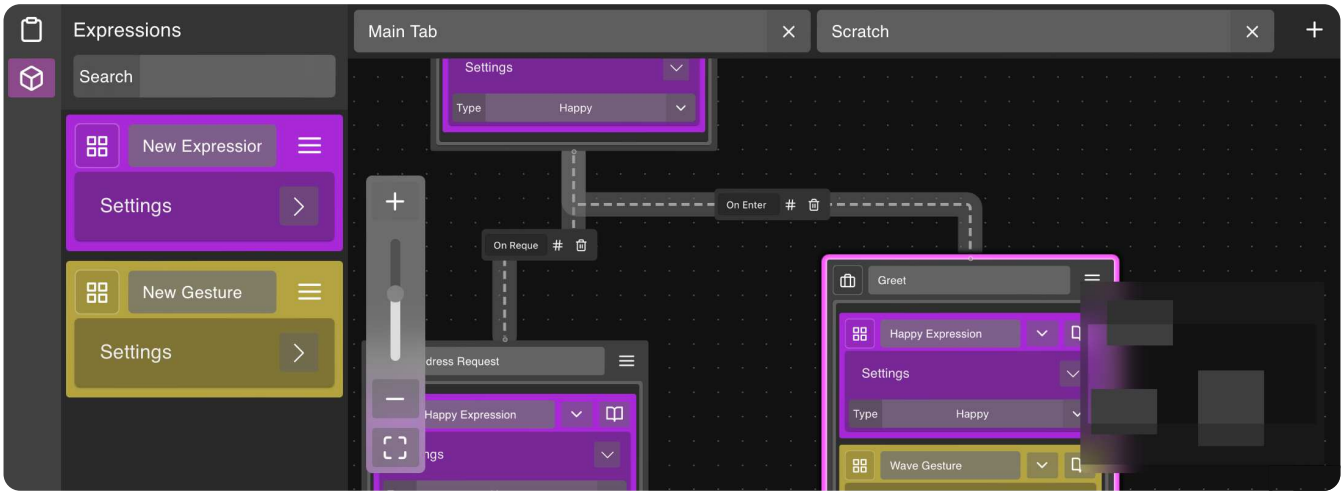

Figure 1: An example flow-based programming system designed with *OpenVP*, illustrating a simple logic about how a robot should behave if a patron enters a store.

ABSTRACT

Authored robotics applications have a diverse set of requirements for their authoring interfaces, being dependent on the underlying architecture of the program, the capabilities of the programmers and engineers using them, and the capabilities of the robot. Visual programming approaches have long been favored for both novice-level accessibility and clear graphical representations, but current tools are limited in their customizability and ability to be integrated holistically into larger design interfaces. *OpenVP* attempts to address this by providing a highly configurable and customizable component library that can be integrated easily into other modern web-based applications.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Visual languages**.

KEYWORDS

visual, programming, library, blocks

ACM Reference Format:
Andrew Schoen and Bilge Mutlu. 2024. OpenVP: A Customizable Visual Programming Environment for Robotics Applications. In *Proceedings of the 2024 ACM/IEEE International Conference on Human-Robot Interaction (HRI '24), March 11–14, 2024, Boulder, CO, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3610977.3637477

HRI '24, March 11–14, 2024, Boulder, CO, USA

ACM ISBN 979-8-4007-0322-5/24/03…$15.00
https://doi.org/10.1145/3610977.3637477

1 INTRODUCTION

Visual programming systems and representations can be used as a method of specifying more accessible robot program abstractions to users, which can then be transformed, translated, and executed on other systems built with more traditional, flexible programming approaches like C, C++, or Python. These visual user-facing programs can come in many forms, including more traditional imperative programs, state machines, or even behavior trees. Given the multimodal characteristics of robot programming, these user-facing programs may be combined with other configuration, visualization, feedback, and analysis subsystems to create complete robot programming systems. Existing well-established and ubiquitous tools like Blockly [1] and Scratch [5], while capable, do suffer in their limited interoperability with these other subsystems and relatively poor flexibility in usage.

Even in the context of greater emphasis on voice-based or chat-based design of programs, there will still be a place for visual programming systems. For example, after designing an entirely voice-based prototype system, Porfirio and colleagues found that while spoken language had benefits, it was generally inefficient or poorly suited for complex specifications, leading the team to ultimately construct a multi-modal system instead [4].

In the process of designing various tools such as *CoFrame* [7] and *LivelyStudio* [6] in the robotics space, we have iterated and improved on a standalone component library for easily specifying highly customized and tightly integrated web-based visual programming environments within these larger, complete applications. This system, called *OpenVP*, is a React component library that can easily be integrated with other subsystems to create interactive robot programming systems. It provides a common block-based programming environment suitable for a range of program designs, such as imperative programming and flow-based or state machine programs.

2 PURPOSE

OpenVP has been designed with multiple goals in mind. These include:

- A high degree of straightforward customizability;
- Serializable and portable program representations;
- Tight integration with other interface subsystems (*e.g.*, visualization and feedback);
- Abstraction of basic interaction details;

OpenVP addresses these goals by creating a system by which a clearly defined *Program Specification* format can define the behavior, characteristics, and appearance of robotics-focused and visual programs and Domain-Specific Languages (DSLs). In the following sections we will articulate some of the characteristics of this system, and how it can be customized for a variety of applications.

3 CHARACTERISTICS

OpenVP has been designed for usability by both end-users and developers, and suitability to the robotics domain. This results in a number of high-level characteristics that guide its design.

3.1 Overview

Environment component serves as a single entry-point to using the system. The *Environment* contains a number of other built-in elements, shown in Figure 2. Central to the environment is the *Programming Canvas*, where the entire program can be visualized. Users can pan and zoom this infinite canvas to see a high-level overview of their program or focus on a particular block. The canvas also features a mini-map and navigation widget. To support editing, a number of other elements are present. The first on the left is a *Drawer Selector*. These buttons activate corresponding *Block Drawers*, which features a filterable list of blocks that can be dragged onto the canvas. Finally, a navigational *Tab Selector* at the top allows for creating, editing, and removing tabs.

3.1.1 Block Types. A key feature of *OpenVP* is the customizability of the system for a variety of possible programming paradigms or applications. As such, it is important to support configurability of the various block types that can be used in each instance. Therefore, *OpenVP* uses a two-part approach to configuring blocks, separating the program data model, which aims to be serializable for server-based applications, from the program representation data, which utilizes Javascript for greater customizability. These two components are called the *ProgramData* and *ProgramSpec*, respectively. This distinction is equivalent to the one between a DSL specification and a program written in that DSL. The *ProgramSpec* contains information about the drawers provided in the interface, as well as all types available to the users, while the *ProgramData* contains a serializable representation of the blocks and connections in the user's program. Edits made by users in the *Environment* are reflected as changes to the *ProgramData*.

Within the *ProgramSpec*, each type specification inherits from one of two primitive types, either *objects* or *functions*, and their specifications include information about the properties of each, as well as customizable rendering information for their *instance* and *reference* blocks (for *objects*), and *declarations* and *calls* (for *functions*). For example, it is possible to create three different *object* types (*e.g.*, a *ProgramType* for the top-level entry point, an *OperationType* to represent some behavior, and a *TargetType* to represent something for the *OperationType* to act upon. If desired, each of these types could include separate visuals for how *instances* and *references* are rendered. Similarly, it is possible to generate multiple types of the primitive *functions*, if needed. Note, the configurability of functions is still driven to a large part by the end-user, since a key feature of entries inheriting from the *function* primitive is that arguments can be added and removed to the declaration itself from the interface.

3.2 Drag and Drop

Drag and Drop is central to the design of *OpenVP*, borrowing from similar tools such as Blockly [1] and Scratch [5]. In *OpenVP*, users can select and drag blocks from the drawer into the *Program Canvas*. Depending on the needs of the application, some block variants can be designated as *canvas* blocks, meaning that they can be dragged directly onto the canvas and organized on the 2D grid. Other blocks can be limited to *non-canvas* blocks, meaning that they are only applicable as children to *canvas* blocks, or other *non-canvas* blocks. When dragging a block, valid drop points are highlighted in the interface, giving a visual reminder of where they can be deposited. Upon hovering the block onto a valid drop zone, a preview of that block in the specified location is shown. Hovering over drop zones without a held block provides a tool-tip that visually shows which blocks are valid at that location.

As mentioned before, types inheriting from the *function* primitive allow editing of their arguments. Function *arguments* can be seen in the header of the corresponding function declaration. As would be expected, arguments within a function's context can be dragged anywhere within that function, but are not available to be dropped outside that context. Conversely, block *references* from outside that context can still be dragged in and used within a function.

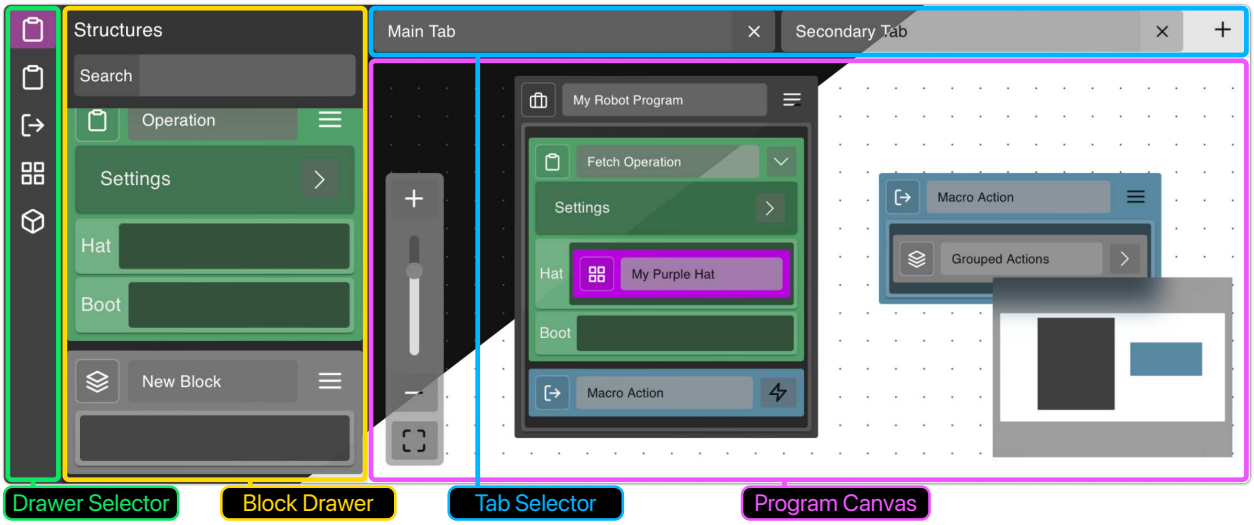

Figure 2: Overview of *OpenVP*'s *Environment* layout, highlighting the four main sections: the *Drawer Selector*, where the active drawer can be set, the *Block Drawer*, where blocks in the current set can be selected from, the *Tab Selector*, where individual tabs can be added, removed, hidden, and edited, and finally the *Program Canvas*, where the program is visualized and edited. Full customization of the theme is possible, as shown in the light/dark modes.

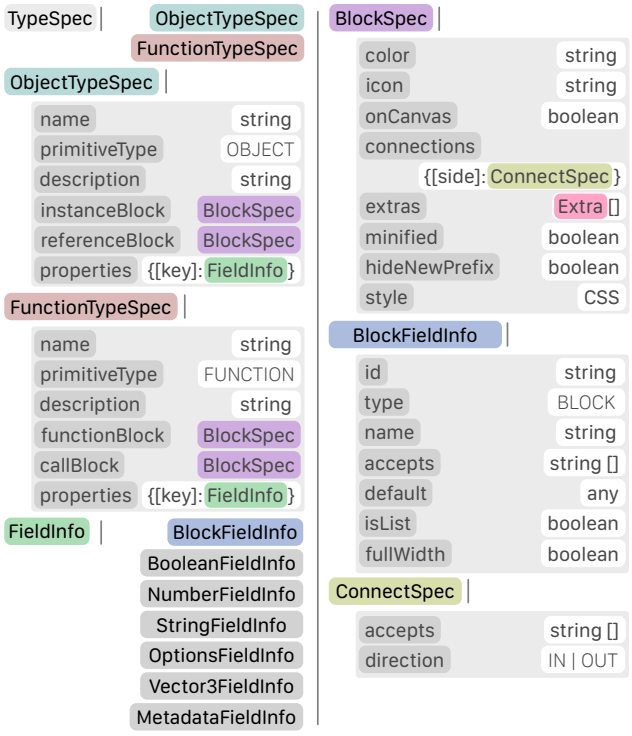

Figure 3: Overview of block customization via their associated *TypeSpec* data. For brevity, some variants are not included, notably non-block *FieldInfo* structs, (*e.g., NumberFieldInfo, StringFieldInfo*, etc.). Also not shown are the *Extra* fields, discussed elsewhere.

3.3 Block Design

Blocks are highly customizable, with their appearance and behavior specified within the *ProgramSpec*. Each block is configured via a *TypeSpec*, which is described in Figure 3. Breaking down this specification, each variant (*ObjectTypeSpec* and *FunctionTypeSpec*) includes a set of two *BlockSpec* entries. Each of these entries can independently specify the color of the block, the icon, whether it appears on the canvas, any connections it can make with other blocks, menu items, whether newly spawned items have a "New" prefix attached to the name (*e.g.*, "New Operation" or "New Robot Function"), whether it features a compact design, or any other CSS style overrides that are desired.

3.3.1 Parameters. Considering the two *TypeSpec* variants, each block can specify the properties of that block through the inclusion of *FieldInfo* data. These structs come in a variety of forms, including *BlockFieldInfo*, *NumberFieldInfo*, *StringFieldInfo*, *OptionFieldInfo*, *BooleanFieldInfo*, *Vector3FieldInfo*, and *MetadataFieldInfo*. The contents of the *BlockFieldInfo* data structure is shown in Figure 3, which dictates how other blocks can be dropped into that block, either as a list of blocks or singular parameters.

3.3.2 Menus and Documentation. Block menus (Extras) can be configured separately for each *BlockSpec* entry and include a set of basic, prescribed functionality like *selection*, *deletion*, *documentation*, *copying*, and *cutting*, as well as more customized cases like *adding arguments to functions* and *custom Javascript functionality*.

The *TypeSpec* can also provide a *description*, which is a markdown-flavored text string, which can be used in the in-editor documentation. This markdown supports standard features, with customized links such that those to valid types will create a hyperlink to that type's documentation.

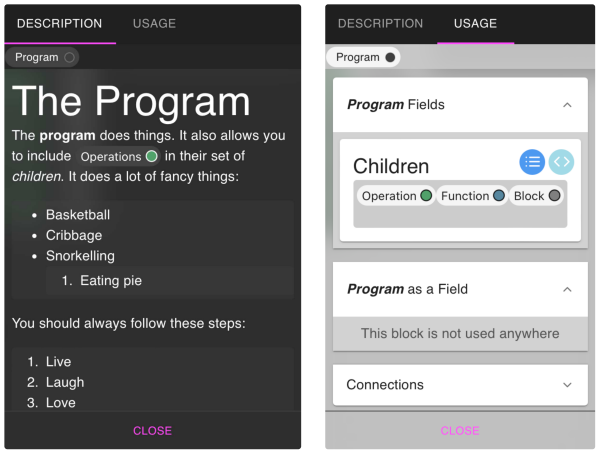

Figure 4: An example of a Documentation section generated for an example *Function* block. The documentation automatically curates how that block is used in other blocks and what blocks it uses. Additionally, the *Description* tab will render the textual markdown description from the *TypeSpec*.

3.4 Connections

For each of a *TypeSpec*'s *BlockSpec* structures, connections between that block and other canvas blocsk can be configured. This data structure includes a set of block types and directions that are allowed to connect. With this capability, it is possible to design flow-based programs, in addition to imperative ones. An example of such a design can be seen in Figure 5.

3.5 Integration

OpenVP was built with the understanding that it needs to operate within the context of a larger design application. This type of capability is essential if, for example, it is desired that when an error is found and selected, the corresponding block highlights. Alternatively, perhaps it is desired that while the actual process is running, the current progress of a given block could be updated. *OpenVP* solves this by making the internal data model, including both the *ProgramSpec* and *ProgramData*, accessible or editable from outside the component.

3.5.1 Data Store. The above functionality is achieved through the use of a flexible data store model called [3]. All the behavior for the store, including the internal actions, are contained within this store, which is provided in the library. Substitution of custom versions of this store as a property of the *Environment* component provides means of overriding behaviors and internal access. Full information configuring this is provided in the documentation.

3.5.2 External Blocks. Suppose a designer wishes to provide a visualization of a certain block from the *Environment*, but outside of the *Environment* itself. For this purpose, we provide an *ExternalBlock* component, which when connected to the correct data store, renders a full block, minus the *Environment*.

3.5.3 Execution Progress. Robotics generally involves a number of long-running processes, and it can sometimes be useful to receive feedback about these processes within the interface itself. Part of the store is reserved for configuring the progress of any blocks in the *Environment*. This is done with a lookup of block ids to numbers, an indeterminate label, or clock-sensitive Javascript functions.

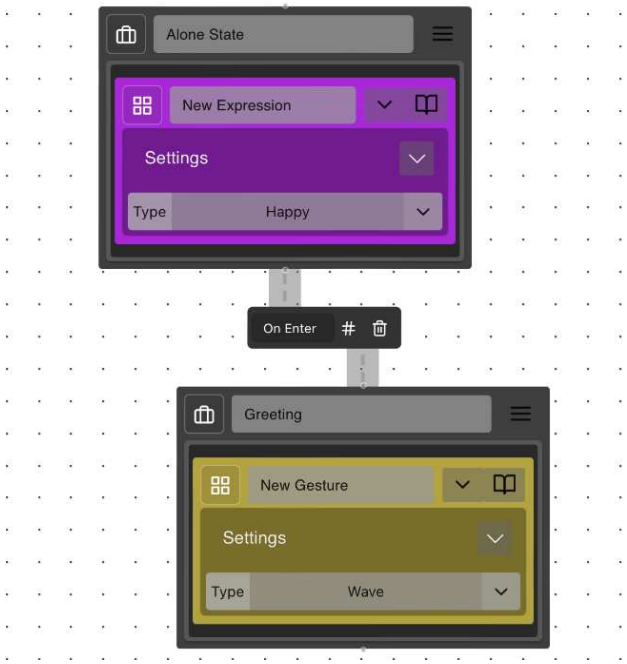

Figure 5: A small example flow-based program, illustrating the ability to draw connections between canvas-based nodes. Connectivity is configured within *BlockSpec* structs.

4 SOURCE CODE AND USAGE

The source code for *OpenVP* is provided freely on Github[1] and can be added to existing projects via the node package manager (NPM). Documentation is hosted on github[2].

OpenVP uses a MIT license. As a high-level library itself, it makes use of many other libraries, such as React Flow [2] and Zustand [3]. The former provides a fee-based Pro tier for usage by commercial entities (it is free for research and academic purposes), so all commercial usage of *OpenVP* should abide by those rules as well.

In conclusion, *OpenVP* seeks to provide an extensible, configurable, and forward-facing tool for visually specifying programs common in robotics applications. Early versions of this library have already been used in widely different robotics programming applications, such as *CoFrame* [7] and *Lively* [6]. It is our hope that by making this software available more widely, others can benefit from and contribute to its further development.

5 ACKNOWLEDGMENTS

This work received financial support from National Science Foundation Awards 1925043 and 2026478. We would like to thank Nathan White for feedback and implementation assistance.

[1] https://github.com/Wisc-HCI/open-vp
[2] https://wisc-hci.github.io/open-vp/

REFERENCES

[1] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 49–50.

[2] Webkid GmbH. 2019. Reactflow: Build Better Node-Based UIs with React Flow. https://reactflow.dev.

[3] Daishi Kato and Paul Henschel. 2019. Zustand: Bear necessities for state management in React. https://docs.pmnd.rs/zustand/.

[4] David Porfirio, Laura Stegner, Maya Cakmak, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. 2023. Sketching Robot Programs On the Fly. In *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction*. 584–593.

[5] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67. Publisher: ACM New York, NY, USA.

[6] Andrew Schoen, Dakota Sullivan, Ze Dong Zhang, Daniel Rakita, and Bilge Mutlu. 2023. Lively: Enabling Multimodal, Lifelike, and Extensible Real-time Robot Motion. In *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction*. 594–602.

[7] Andrew Schoen, Nathan White, Curt Henrichs, Amanda Siebert-Evenstone, David Shaffer, and Bilge Mutlu. 2022. CoFrame: A System for Training Novice Cobot Programmers. In *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 185–194.