Vipo: Spatial-Visual Programming with Functions for Robot-IoT Workflows

Gaoping Huang[1], Pawan S. Rao[2], Meng-Han Wu[1], Xun Qian[2], Shimon Y. Nof[3], Karthik Ramani[1,2], Alexander J. Quinn[1]

[1]School of Electrical & Computer Engineering, Purdue University, West Lafayette, Indiana, USA
[2]School of Mechanical Engineering, Purdue University, West Lafayette, Indiana, USA
[3]School of Industrial Engineering, Purdue University, West Lafayette, Indiana, USA
{huang679, rao81, wu784, qian85, nof, ramani, aq}@purdue.edu

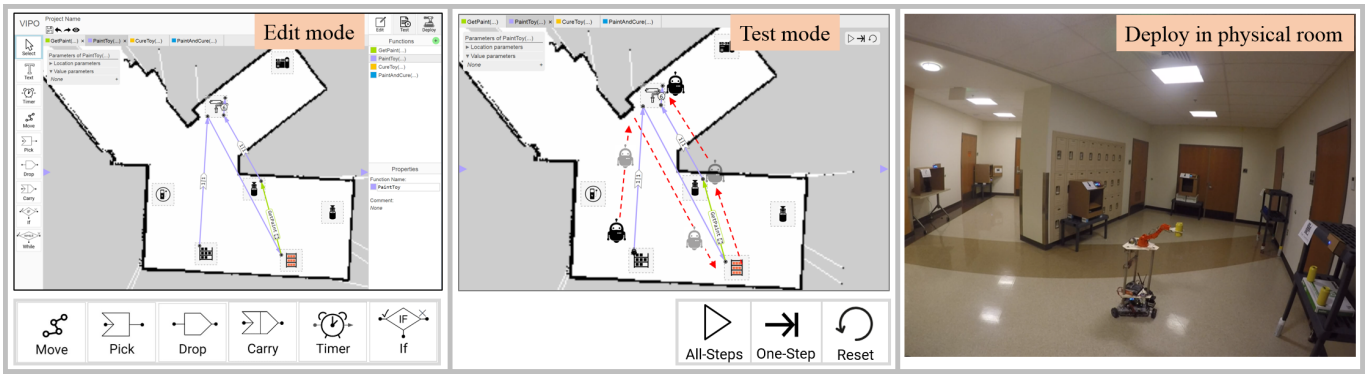

Figure 1. Vipo programs are created using standard programming constructs over a 2D floor map using the Vipo editor (left), then tested in simulation (center), and deployed to mobile robots that interact with IoT devices in the physical environment (right).

ABSTRACT

Mobile robots and IoT (Internet of Things) devices can increase productivity, but only if they can be programmed by workers who understand the domain. This is especially true in manufacturing. Visual programming in the spatial context of the operating environment can enable mental models at a familiar level of abstraction. However, spatial-visual programming is still in its infancy; existing systems lack IoT integration and fundamental constructs, such as functions, that are essential for code reuse, encapsulation, or recursive algorithms. We present Vipo, a spatial-visual programming system for robot-IoT workflows. Vipo was designed with input from managers at six factories using mobile robots. Our user study (n=22) evaluated efficiency, correctness, comprehensibility of spatial-visual programming with functions.

Author Keywords

Spatial visual programming; Robots; Internet of Things;

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376670

CCS Concepts

•**Human-centered computing** → **User interface programming;**

INTRODUCTION

Programming robots and Internet of Things (IoT) devices together gives rise to vast new opportunities for factory floors, and is part of a recent trend toward Internet of Robotic Things [16, 32]. As mobile robots and human workers become more tightly integrated within IoT environments, the task of instructing the machines has become increasingly complex. With more devices to coordinate, human operators must author workflows that are inherently computational in the context of dynamic spatial environments.

Factories typify the challenges of coordinating complex workflows with mobile robots delivering parts and interoperating with manufacturing equipment. As manufacturing processes ever more increasingly depend on customization and product changes, the effort needed to create or modify workflows becomes a bottleneck. Furthermore, some responsibility for programming robots and their interactions with IoT devices must shift to the workers directly involved with a given manufacturing process [7].

Bringing factory workers into the process will require the right level of abstraction and context. *Task-level programming*

offers a starting point. In this paradigm, expert programmers write code for robots to perform generalized tasks, which non-programmers can use to direct the robot [1, 9, 29]. However, as basic programming skills become more pervasive, the need becomes less focused on programmers vs. non-programmers, and more on enabling domain experts to efficiently specify workflows.

We present Vipo, a *spatial-visual programming* system for robot-IoT workflows. With spatial-visual programming, programs are created using visual programming constructs drawn directly on a map of the operating environment. To start, a floor map of the physical space is uploaded into Vipo. Then, users can draw paths for robot movements, and specify loops and conditionals by connecting paths with shapes. Those constructs can also be found in other recently developed spatial-visual programming systems [11, 23, 10].

Vipo builds on those capabilities in two significant ways.

First, the Vipo language allows workers to write programs using *functions*. In conventional programming, support (or non-support) of functions is often viewed as an informal litmus test for a “real programming language". Far from just another language feature, functions are a crucial building block that enables encapsulation, reuse, and scale. Steps and data related to a meaningful sub-goal can be encapsulated in a function definition. Then, the function can be called repeatedly with different parameters. Functions can even be called recursively. Support for recursive function calls allows workers to express programs that could not be expressed without functions (or stack data structures).

Second, the Vipo architecture integrates IoT devices into the programming and execution environments with no prior configuration. Using the Vipo protocol, devices broadcast their location, capabilities, and resource status (e.g., power, supplies, etc.) in a format based on the Resource Description Framework (RDF). The Vipo architecture uses those messages to discover devices. Their status and capabilities are automatically integrated into the Vipo IDE, a web-based development environment used to create programs with Vipo. When the programmer specifies an action that involves an IoT device, its capabilities are populated into the editor, and its resource status can be checked to ensure the actions are possible. Building on the Robot Operating System (ROS) [31], the Vipo architecture compiles the user’s programs into a form that can be executed by robots, or simulated. Our research provides pathways for other researchers to develop more collaborative and interoperable settings that can improve productivity of humans and IoT devices in small- and medium-sized organizations.

The key contributions can be summarized as follows:

- The Vipo language is a spatial-visual language for programming robot-IoT automations with functions, as well as conditions, loops, movement, and IoT device operations.
- The Vipo architecture, including the Vipo protocol, enables real-time integration of mobile robots and IoT devices with dynamic state information (e.g., locations, capabilities, resource availability, etc.).
- The Vipo IDE integrates 1) code creation, 2) testing/simulation, 3) deployment, and 4) monitoring in an integrated development environment, as a demonstration of the overarching vision for factory or other robot-IoT automation.
- Our user study with 22 participants validated the comprehensibility, correctness, and efficiency of spatial-visual programming with functions.

RELATED WORK

Vipo builds on prior work in visual programming languages and interfaces, spatial-visual programming, and communication protocols between smart devices.

Visual Programming for Robots and IoT devices

Many approaches have been studied for authoring robot-IoT workflows, such as programming by demonstration [3], collaborative control [15], and visual programming. This work focuses on visual programming that enables logic and control flow. Visual programming interfaces make programming more approachable for non-experts and thus enable workers to author workflows for robots and/or IoT devices. Many visual programming interfaces have been developed to program tasks for IoT devices [5, 14, 4], robots [30, 18, 24, 22, 13, 17, 12], or for both robots and IoT devices [38, 26, 41, 35].

These interfaces are mainly built on two authoring approaches: *form-filling* and *visual programming languages*. Form-filling approach allows users to fill a predefined form by adding actions or triggers via drop-down menus [33, 38, 21]. On the other hand, visual programming languages provide visual constructs (e.g., functions, conditions, and loops) to wire the sensory data and actions of robots or IoT devices into tasks, such as Blockly [8]. Since form-filling is less flexible than visual programming languages in authoring dynamic and complex workflows, most interfaces mentioned above have adapted or designed visual programming languages to author workflows in various formats, such as blocks [24, 8], data-flow [22], flow-chart [5], event-based, [12], or state-flow [26].

Although these visual programming languages could represent workflows in many formats, they lack suitable visual notations to represent the activities occurring within a spatial environment, such as delivering parts by mobile robots and interacting with machines. We design Vipo to provide users with handy notations to program workflows while maintaining the power of a programming language in the spatial domain.

Spatial-Visual Programming

The ability to sense the spatial relationship between objects and environments is one of the key benefits of programming via augmented reality (e.g., V.Ra [10]), virtual reality (e.g., Ivy [14]), or 3D map (e.g., [39]). Such benefit is automatically gained when authoring in 3D space, but is less obvious to obtain in 2D space. Given that users are more familiar with authoring via 2D interface and have easier access to 2D authoring tools (e.g., computers), it would be valuable to embrace the benefit of spatial-awareness in a 2D space.

Spatial-visual programming in 2D space is still in its infancy. A few research projects have explored this area. For example, Vizir [11] allows air traffic controllers (ATC) to author automation on top of a geographic airport map with a set of ATC-specific visual constructs. By placing the visual constructs above the map, Vizir tries to maximize the closeness of the control and actions, as well as the predictability of the automation. In addition, Ruru [13] designed spatial metaphors for input sensors to allow the position and orientation of an input relative to a device to be expressed visually. Kitty [23] allows users to sketch animated drawings to illustrate the spatial and temporal relationship between entities. RoboShop [25] supports robot housework assignment by sketching on a bird's-eye view of the environment.

Although these approaches enabled users to create simple workflows or illustrations, they did not support *functions* that are essential for a programming language. Functions are supported in most textual languages and non-spatial visual programming languages mentioned earlier. Given that manufacturing workflows have the tendency to become more complex and customizable [7], functions can play an important role due to its reusability, modularity, and flexibility. In this paper, we designed and implemented functions in a spatial space. Along with other spatial constructs, the Vipo language aims to build an accurate conceptual model of the spatial relationship between programmed tasks and the environment.

IoT Protocols & ROS

IoT protocol is one of the key components to bridge the digital programming interface and the physical execution environment. Specifically, to program a workflow, users need to have access to the capabilities and sensory data of robots and IoT devices. In addition, these capabilities and sensory data should be represented in a format that is understandable by users.

Many protocols have been proposed to facilitate communication between connected systems [37, 19, 36, 34, 40]. For example, MQTT [19] is a publish-subscribe messaging protocol that is suitable for mobile applications. In addition, several protocols have been created to describe data/message in specific formats. For example, RDF [27] is a standard model for data interchange on the Web, while IoT-Lite [2] is a variation of RDF to describe IoT resources, entities and services.

In this work, we take advantage of the Publisher/Subscriber functionality provided by ROS [31] to enable status sharing and task coordination. Furthermore, we adapt the RDF protocol to describe the status and capabilities in a way that Vipo can interpret. This modified RDF message enables IoT devices to be automatically discovered and added into Vipo.

DESIGN OF VIPO

The key contribution of this work is the introduction *functions*—including a notation and interactions—to spatial-visual programming for robot-IoT workflows. Later in this paper, we will explain the language design and architectural challenges that were entailed to bring functions to spatial-visual programming. To establish context for that discussion, we will first explain the design goals of Vipo, and the more foundational elements of the Vipo language.

Requirements and design goals

The requirements and design goals were gathered through a series of visits by a group of at least three researchers to six factories. These included manufacturers of construction equipment, automobiles, electronic equipment, and components.

The visits were motivated by other collaborations related to robot-IoT automation, but on each occasion, the researchers asked questions related to the firms' challenges regarding specification of robot-IoT workflows.

Representatives also visited the lab where Vipo was developed during the development of Vipo. The researchers gave demonstrations for plant managers and executives, and received feedback that guided our understanding of the requirements.

The visits provided design hints that guided the design of Vipo. For example, representatives from a consulting firm that supports small- and medium-sized enterprises informed us about the cost structure of equipment acquisition, which led to key decisions related to the Vipo architecture.

Language design

Transitional Constructs (move)

Transitional constructs represent operations that transit from a source to a destination. They are typically used to plan the motions of mobile robots, including "move", "pick", "drop", and "carry". All four constructs can be found from the toolbar on the left of Figure 10. For example, "pick" means the robot picks objects from a source and moves to a destination. "Drop" means the robot moves from source and drops objects to destination. When users need the robot to pick from source and drop at destination, they can use "carry", which is a combination of "pick" and "drop". Transitional constructs can represent the spatial relationship between devices, such as the direction and distance. Figure 2 shows four transitional constructs between the paint inventory to paint mixer.

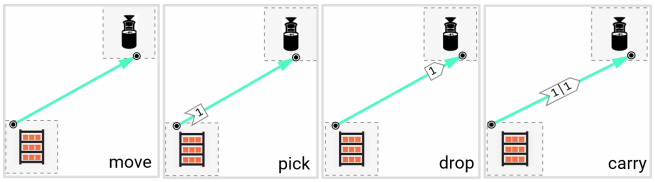

Figure 2. Examples of four transitional constructs.

There are two reasons why we design distinct notations for "pick", "drop", and "carry", rather than reusing "carry" as a universal notation. First, these notations leverage spatial proximity. Specifically, "pick" is spatially closer to the source, "drop" is closer to the destination, and "carry" sits in the middle (Figure 2). Second, a distinct shape may enhance readability and memorabilityso that users can immediately tell if a notation is a "pick", "drop", or "carry". To strengthen the concept that "carry" is a combination of "pick" and "drop", the symbol of "carry" is also a combination of the symbols of "pick" and "drop".

In-place constructs (IoT operations)

Upon receiving materials from a robot, an IoT device can use them for in-place operations such as consuming, processing,

and/or packing. A “Timer" symbol reveals an estimated execution time of this operation. Upon clicking the Timer symbol, a popover allows users to choose an operation, duration, and other parameters. Figure 3 shows an example in which the paint mixer is programmed to mix paint for about 5 minutes.

Figure 3. Example of in-place construct: mix paint for 5 minutes. (a) start to create a timer, (b) use popover to select a machine capability and enter required parameters, (c) show estimated time in timer.

Control-flow constructs (if and loops)

Vipo supports control-flow through “if" and “while". Both constructs use a condition to select the next execution path. Conditions may include operators, numerical values, and/or device properties (e.g., printer status, oven temperature, available storage capacity on a shelf, etc.). Figure 5 shows an example in which the robot is asked to drop the paint can at mixer A if its jobStatus (i.e., completion progress) is greater than 80 percent, otherwise drop at mixer B.

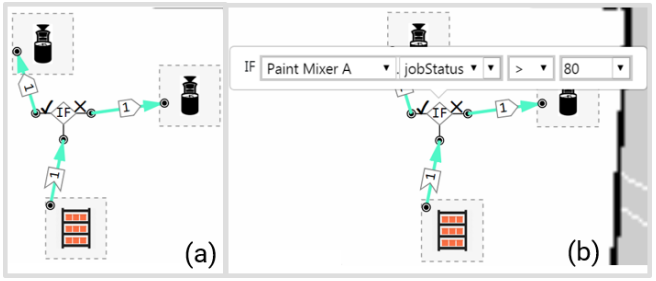

Figure 4. Example of "if". (a) The if condition decides which of two branches to follow, (b) Users select a property of a device, a logic operator, and enter a value to specify the if condition.

Besides “if" and “while", the aforementioned transitional constructs could also be considered as one kind of control flow constructs: “goto". In most cases, a workflow of a robot is a linear sequence of actions. However, sometimes users may draw a path (e.g., one of transitional constructs) that goes back to its prior action, which forms a loop. When this backward “goto" path is combined with “if", the workflow is functionally equivalent to a “while" loop.

Spatial Functions

This section shows how the Vipo language supports function definition and function call in the spatial domain.

Function definition

A function is used to represent a workflow that can be tested and executed alone, or reused by other functions via function call. A function can include one or more primitive constructs, or even functions. Each function has a distinct color to differentiate from other functions. The constructs belonging to a function have the same color as the function. To support people with color blindness, users can use different color value.

Functions are displayed as a list on the top-right panel of the editor (Figure 10). The user can click on a function to display its content in the main workspace within a tab. A click on another function opens a new tab. All functions for the same environment share the same layout map and machines, and are organized by tabs.

Similar to textual programming languages and other non-spatial visual languages, defining functions with parameters can make the workflow more flexible and customizable. There are two types of parameters: value and location; both of them start with a dollar sign “$".

- Value parameter (e.g., $\$n$). A value parameter is used to store a number. With the help of value parameter, users may replace a constant number with an algebraic expression. An algebraic expression can be a constant number, a variable, or algebraic operation on algebraic expressions (e.g., $\$n \times 2 + 3$). For example, the number of objects to pick/drop/carry can be “$\$n$" instead of a constant number (Figure 5). Likewise, the condition of “if" and “while" can use an algebraic expression.
- Location parameter (e.g., $\$start$, $\$end$). A location parameter is used to store a machine. With the help of location parameter, users may change some actions that happen on one machine to happen on another machine. For example, a robot may visit and interact with machine A, machine B, and machine C in linear order. Rather than visiting a sequence of fixed machines, users may convert machine B as a location parameter (e.g., $\$middle$). In such case, if we pass machine D into $\$middle$, the robot will eventually visit and interact with machine A, D, and C.

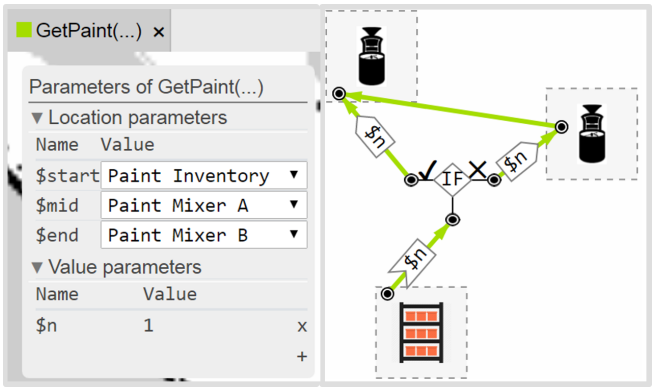

Figure 5. Function definition. Define a value parameter “$\$n$" for function “GetPaint" (left); use “$\$n$" in pick, drop, and if-condition (right).

Function call

A function can reuse existing workflows via function call. If users want to call a function, they can right-click the desired function in the function list and select "Call this function" from the context menu. Then users can click on the map to specify the start and end of the function. This drawing operation is the same as transitional constructs. The visual notation of a callee (i.e., the function being called) also looks like a transitional construct, as shown in Figure 6a. The visual notation includes the function name and an “Expand" button. Once clicking on the “Expand" button (Figure 6b), the internal definition of the callee will be displayed (Figure 6c). This allows users to quickly peek the definition without switching

between functions. The path color of a callee remains the same as the callee, which makes it distinct from the constructs belonging to the caller (i.e., the function that calls callee).

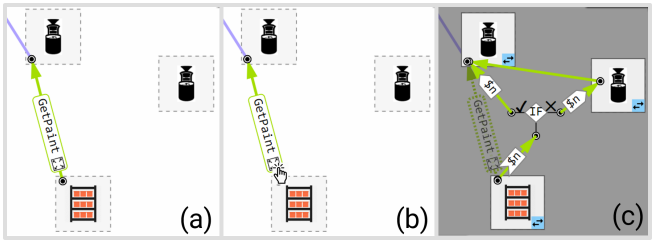

Figure 6. Function call and assign values. (a) call function "GetPaint", (b) click to expand the detail, (c) the detail of "GetPaint" is displayed within lightbox

To further customize the callee workflow, users can pass values and locations to its parameters. Upon clicking the callee notation, a popover appears to allow editing of the parameters. Similarly, users can select a different device for a location parameter. Once a location parameter is assigned to a new device, the workflow to be executed is changed to visit and interact with the new device. This change can also be visualized if users click the "Expand" button to see the expanded detail of callee. This change only affects execution of callees within the current caller; the function definition is not otherwise affected.

In addition, since users can expand a callee to see its internal detail, it enables a unique way of assigning location parameter. Users can click the "SwitchDevice" button near the bottom-right corner of each device. Then users can click on a different device and assign the new device to the corresponding location parameter, as shown in Figure 7. The originality of our approach is that a location parameter can be spatially visualized and also can be assigned to a new device spatially. Once the location parameter is successfully updated, the constructs that are related to this location parameter will be automatically switched to the new device.

ARCHITECTURE

The system can be treated as a three-layer architecture, as shown in the Figure 8. From top to bottom, it includes a task planning layer (i.e., Vipo), a task control layer (i.e., ROS Master), and a task execution layer (e.g., robots and IoT devices).

The task planning layer is part of the Vipo IDE, a web-based development and simulation environment that allows users to program tasks for robots/IoT devices. More details are given in later section.

The task control layer is ROS Master which acts as the bridge of the two-way communication between Vipo and the robots/IoT devices. When RDF message is sent from robots/IoT, ROS Master maintains a global context that keeps track of all the connected devices. ROS Master only sends the difference of two adjacent RDF messages from the same device to Vipo to reduce network traffic.

When a task script is sent from Vipo, ROS Master creates a thread for each new task. The task script receives the global context to fetch the details of the corresponding device. Each

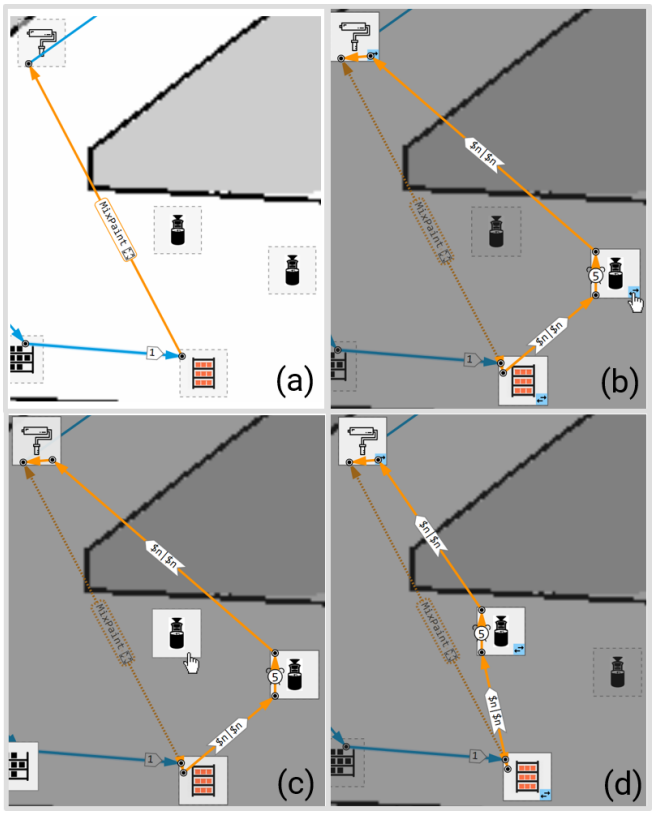

Figure 7. Assigning a new location in expanded callee view. a) The callee "MixPaint" before expanding, b) click on the "SwitchDevice" button in the expanded view of callee, c) click on a new device, d) location parameter is successfully changed and the constructs are switched to the new device automatically.

line of task script is translated into ROS-specific code and sent to a corresponding machine.

The task execution layer consists of physical or simulated devices (robots/IoT devices). Each device holds the spatial information (e.g., location), sensory data (e.g., temperature and job status), and functionalities (e.g., packing a box, 3D printing). IoT devices and robots periodically transmit RDF messages to ROS Master, while receiving command scripts from ROS Master.

In our system, we adopted a modified version of RDF to suit our application. Figure 9 presents a sample RDF message for a paint mixing machine. The modified RDF message has device-information fields (ID, name, description, location etc.), machine-specific *methods* and *properties*. *Methods* are the functions that the machine is capable of performing (like mix-paint, set-temperature, start, stop etc.) while *properties* are the real-time operation parameters (like job-status, temperature, coolant-level, run-time, health-status etc.)

RDF Messages serve a centrol role in Vipo. The fields discussed above are directly used by Vipo for different purposes. Specifically, fields like the location, iconUrl, and iconSize are used for rendering the icons on the map. The *properties* fields are used for populating the drop-downs of the control-flow constructs (e.g., jobStatus used in Figure 5). Moreover, the

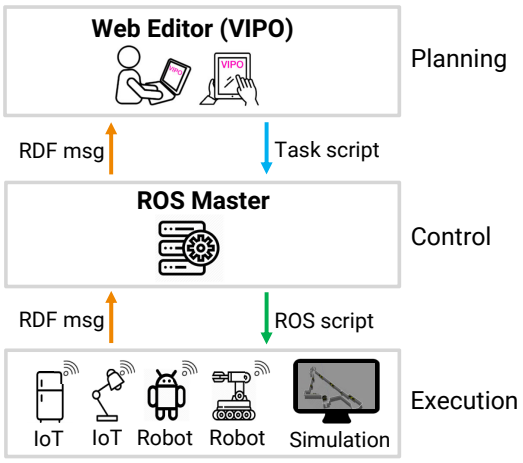

Figure 8. The three-layer architecture.

methods fields are used to populate the drop-downs of in-situ operation constructs as callable functions.

Given that the capabilities of IoT devices and robots often need users to specify some values, the *methods* fields of RDF message can automatically provide the parameter interface for capabilities. Vipo uses the *vipo_msg_type* field as the parameter interface that specifies what kind of value the method requires. For example, the "mix-paint" method of a paint mixer requires users to specify a time. Similarly, a "move" method of a robot needs to specify the location to move. So far we have supported four types: time, object, location, and value, which can be easily extended to support more types.

```
------------------
Property.msg
   string name
   float32 value

------------------
Method.msg
   string name
   string vipo_msg_type
   string topic_name

------------------
RdfMsg.msg
  string id
  string name
  string description
  float32[] location
  string size
  string imgUrl
  bool done
  bool error
  Property[] properties
  Method[] methods
```

```
---
id: "paintMixer02",
name: "Paint Mixer B",
description: "Mixes paint for a
              given duration of time",
location: [14.7755,-5.8476,0.5699],
size: "medium",
imgUrl: "/imgs/mixer.png",
done: false,
error: false,
properties:
 -
  name: "jobStatus",
  value: 50
methods:
 -
  name: "dispense",
  vipo_msg_type: "object",
  topic_name: "/dev05_dispense"
 -
  name: "mixPaint",
  vipo_msg_type: "time",
  topic_name: "/dev05_mixPaint"
---
```

Figure 9. The schema of modified RDF (left) and a sample RDF message (right).

Communication Between Layers

Each layer sends and receives messages from the adjacent layer(s). Communication is bidirectional.

Bottom-up: Broadcasting spatial and contextual information. The goal of this communication is to broadcast the spatial and contextual information from the execution layer to the planning layer. The broadcasted messages are used to setup the programming environment and reflect the real-time status of robots/IoT devices. The format of the message is based on a modified version of RDF, as described earlier.

ROS Master forwards these RDF messages to Vipo, which further renders each robot/IoT at the corresponding location defined in RDF messages. Moreover, the functionalities in RDF messages are converted to callable functions that can be used to program a task.

Top-Down: Deploying Task. Programs created by workers are transmitted from the planning layer to the execution layer. First, the Vipo architecture compiles the visual program into a textual task script. It is then sent to ROS Master, which converts each line of script into ROS-specific commands, which it sends to the corresponding robot or IoT device. As the devices execute the commands, execution status (e.g., success, error) is sent back to ROS Master as an RDF message.

Vipo IDE - TASK PLANNING LAYER

Vipo IDE has three work modes: Edit, Test, and Deploy, as shown in the top-right corner of Figure 10. In Edit mode, users can program workflows in the Vipo language. In Test mode, users can simulate what the workflow would do before being deployed to a physical environment. In Deploy mode, the visual programs are compiled and sent to robots/IoT devices for execution. At the same time, the real-time status of robots/IoT devices is monitored and viewed in the editor. This section introduces how to setup the interface for a new environment, and then introduces three work modes of the Vipo IDE.

Figure 10. The Vipo IDE displays a toolbar (left), three modes (top-right), and a 2D layout map with IoT machines at the corresponding location (center).

Setup

Vipo receives the information from the execution layer to set up the environment.

The layout map as the background canvas

At the start of the application, the map of the environment is displayed, as shown in Figure 10. The map is generated by

the robot after scanning the environment (e.g., via LIDAR). At the same time, while the robot is scanning, the locations of IoT devices are obtained (similar to [20]). Both the map and the locations are sent to ROS Master and eventually to Vipo, as described in the architecture.

IoT Device Registration

Similar to the map generation, IoT devices periodically broadcast their contextual information (e.g., id, name, status, and supported capabilities) to ROS Master then to Vipo. Thereafter, Vipo automatically renders the IoT devices as icons at the corresponding locations on the 2D layout map, as shown in Figure 10. The icon image is also defined by the IoT itself.

Edit Mode – Program with The Vipo language

Based on the layout map and icons of IoT devices in the environment, we designed the Vipo language to program workflows using spatially oriented constructs. The syntax and programming workflow were introduced above.

Test Mode – Simulate Execution

Testing is a mandatory activity before deployment. Users can click the "Test" button to enter test mode and simulate how the workflow will be executed by a robot. Three control buttons are provided, including playing all steps at once, playing one step at a time, and starting over from the beginning (Figure 11a). A robot moves along the path with animation. If there is a control-flow constructs, the condition is evaluated to choose which branch to follow (Figure 11c). The condition may involve the properties of machines, which are dynamic and external. To enable testing, users can enter test value to mock the properties (Figure 11b). By passing different test values, the robot is able to move along both branches of an "If" statement so that the test coverage is more comprehensive.

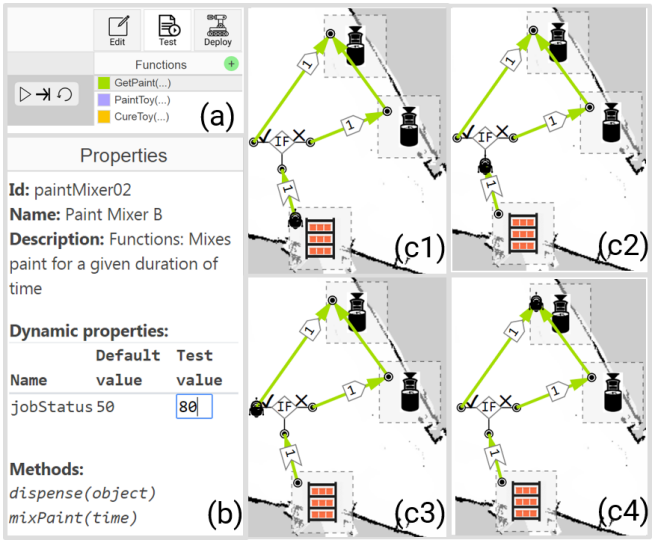

Figure 11. Test mode. a) Users switch to the Test mode and use three buttons to control the simulation, b) users set test values for dynamic properties of devices to simulate different execution results, c) once users click the play button, a robot moves along the path and chooses the proper branch to follow based on the if condition.

Before the robot moves to the next construct, Vipo checks the syntax of the next construct and evaluates its value. For example, if the number to pick/drop is missing or the condition of "If" statement is not completely filled, a red bulb icon will be displayed with error message (Figure 10). Moreover, if the algebraic expression has wrong syntax or the variable is not defined, the red bulb icon will also be shown to give the error message. The robot pauses movement until users fix the error.

Unlike other programming languages where the programmed script and the simulation result are visualized in separate interfaces, the Vipo IDE can show the programmed constructs and simulation result in the same interface. In other words, the simulation is running directly on top of the programming constructs within the environment. This direct coupling may enhance the predictability of the simulation result. In a loose sense, Vipo supports *"What-You-See-Is-What-You-Expect-To-Get"* in the environment.

Deploy Mode - Execute and Monitor Status

The Vipo IDE can monitor the real-time execution status of robots and machines, and allows users to visually associate the robot's movement with objects in the physical operating environment. If the physical robot is moving, it is shown in the interface at the corresponding location. Since the visual constructs of the task are also displayed in the interface, users are able to recognize which construct the robot is currently executing. This direct mapping between the execution in physical environment and the programming constructs in digital layout can help users better understand the current state and predict the next state.

Moreover, if a machine reports an error during execution, the Vipo IDE shows a red mark to highlight that machine. This real-time error reporting allows users to notice the error quickly and fix it to increase productivity.

The Edit mode and Deploy mode are separated in the environment, so that programmers can focus on authoring programs without the distraction of animated updates about the physical robot's location. When switching back to the Edit mode, the RDF messages at that time are cached and used to render the machines statically in Edit mode. Subsequent RDF messages are ignored until switching back to deploy mode.

USE CASES

To show how Vipo can be used to program tasks for robots/IoT devices, we present and explain two use cases of our system.

Scalability and Reusability - Recursive Function

The first use case solves the classic Tower of Hanoi puzzle[1], a commonly used example used to teach recursion in computer science courses. This puzzle represents a simple but non-trivial task for robots and IoT devices. The primary operations used for Towers of Hanoi—pick, move, and drop disk—are a natural fit for robotics, and have real-world analogs in factory warehouses (i.e., stacking crates). The three rods can be considered as IoT devices. Figure 12 shows a recursive solution expressed in the Vipo language.

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi

The ability to define parameters and use function calls enables users to program more complex workflow in a spatial domain, including recursive functions.

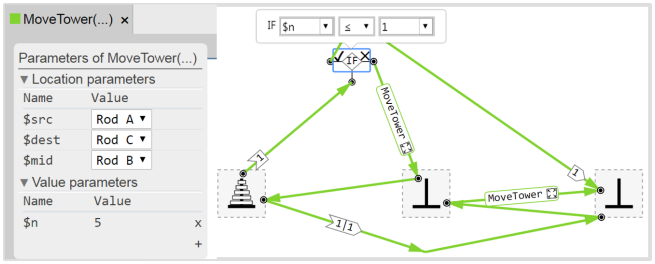

Figure 12. Recursive function calls enables a recursive solution to the classic Towers of Hanoi.

Factory Use Case

To validate the system developed in an industrial context, we deployed Vipo to author robot-IoT workflows for simple material-handling applications in a small-scale emulated painting factory. The industry considered is assumed to have an advanced factory setup with smart machinery and autonomous mobile robot (AMR). Workflow of a typical job order comprised of the following sequential steps:

1) Source the parts to be painted from the Item Inventory
2) Source paint cans from the Paint Inventory
3) Mix the paint to obtain a given shade using Paint Mixer
4) Paint the parts using the Painting Machine
5) Cure the painted parts at a given temperature in the Oven

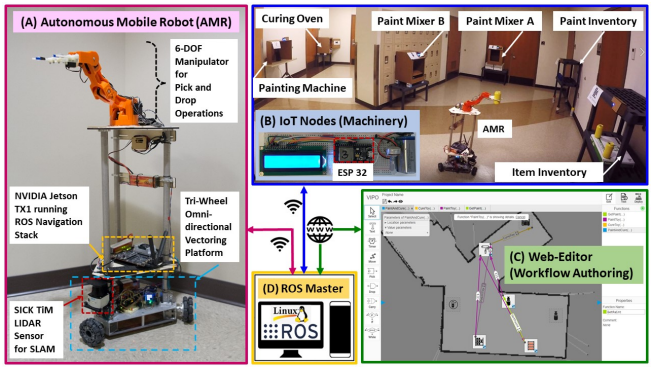

Figure 13. System Setup for the factory use-case (A) Autonomous Mobile Robot (B) IoT Nodes (industrial machinery) (C) The Vipo IDE (D) ROS Master

System setup (Figure 13) for the use case comprised of (a) an omnidirectional robot capable of autonomous navigation using LIDAR (SICK TiM561) and SLAM, (b) a 6-DOF robotic arm mounted on the mobile base for pick and drop operations, (c) Six ESP32 microcontrollers emulating six smart industrial machines, (d) ROS-Master for task flow and execution, and (e) the Vipo IDE for programming the workflow.

Prior to physical deployment, the workflow was programmed and simulated in edit and test modes of Vipo. Then the program was deployed to the robot to complete the programmed sequence in the emulated painting factory.

USER STUDY #1: VIPO VS. BLOCKLY

To understand the strength and limitation of spatial-visual language, we conducted a user study that compared Vipo with a non-spatial visual programming tool called Blockly [8]. Here, “spatial" means the programming constructs that contain spatial information, such as direction and distance.

Blockly is chosen as the non-spatial baseline due to three reasons. First, at the visual programming language level, the Vipo language and the block-based visual language of Blockly are imperative programming language and thus able to specify task for robots as a set of actions. This is unlike other dataflow-based or event-triggered visual languages, such as Node-RED [5]. Second, Blockly can create customized blocks to represent machines, functions, and properties, which can be used to program the same tasks as Vipo. Third, Blocky is a well-established visual programming tool and has been widely adapted for programming educational purposes, which serves as a solid baseline for evaluating Vipo in terms of usability.

In this setting, the key distinction between Vipo and Blockly is that Vipo uses spatial constructs directly on a 2D map, while Blockly uses non-spatial constructs in a separate view.

Experiment

Twelve participants were recruited for the study, of which most are engineering students between the ages of 19-22 years. Of the 12 total participants, 11 participants were novice programmers (0-1 year of experience), and only 1 participant was an experienced programmer (3+ years of experience). 3 users had previous experience of visual programming in Scratch (< 6 months of usage).

A within-subject study was conducted in which each participant was asked to use both Vipo and Blockly to program tasks in counterbalanced order. In other words, six participants used Vipo first and then Blockly, while the other six participants used Blockly first and then Vipo.

For each interface, participants had to watch a video tutorial, finish six tasks (Task 1-Task 6), and finally fill in a questionnaire regarding the user experience and cognitive dimensions for interface evaluation [6]. In the first five tasks, participants were asked to program workflows using basic programming constructs (e.g., move, pick, drop, and If-Else), as well as more advanced programming constructs (e.g., Function calls). For each task, we provided a written document with the goal and general approach, but no hints on how to structure the program. The first two tasks (Task 1 and Task 2) were designed such that each was a standalone workflow (e.g., "PickAndPack" and "StoreItem"), but also could be reused in Tasks 3-5 to form a more complete workflow via function calls. Participants were allowed to use the Test-Mode for assistance (debugging). Once participants finished all five tasks, they were asked to read and comprehend a task programmed by the authors (Task 6). While participants were programming using the interface and filling in the questionnaire, the computer screen was recorded.

The first five tasks are the same for both conditions, which are used to compare the system under the same context. The last reading task is to test comprehension. However, in order

to prevent participants from repeating their answers from the previous interface, we kept the bulk of the programs the same while used slightly different numbers in If-Else condition. The hypothesis of this experiment is spatial-visual programs created in the Vipo language are more comprehensible than functionally equivalent programs written in a non-spatial visual programming language.

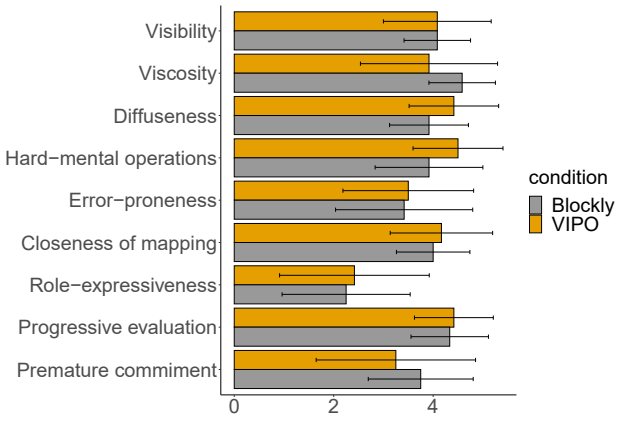

Figure 14. Results of usability test in 9 cognitive dimensions. None of them have significant difference.

Results & Discussion

In this section, we first report the results in comprehension test, then report usability test results.

Spatial constructs make programs more comprehensible

In the comprehension test (Task 6), participants were given a programmed workflow and asked to answer five questions, such as identifying the optimal steps, number of machines involved, and the source and destination of the program based on different if-condition. Each question counts as 1 point. We use the total score of the five questions to represent a participant's understanding of the program. A paired t-test was conducted to compare the comprehension test results in Vipo and Blockly. There was a significant difference in the scores for Vipo (M=3.83, SD=1.03) and Blockly (M=2.83, SD=1.53), t(10)=2.57, $p = 0.03 < 0.05$. The results suggest that participants had a better understanding of the program when the workflow is programmed in the Vipo language, which supports our hypothesis that spatial visual programming language improves user's comprehension of the program. The reason might be that the Vipo language shows the programmed constructs in the same interface, therefore participants can easily infer the results of the workflow without any context switching. On the contrary, participants have to constantly switch between the map and the constructs to understand the execution result of the workflow in Blockly, which increases the chance of making mistakes.

Usability of Vipo is on par with Blockly

Next, we look at the cognitive usability test results reported by the participants. After completing all of the programming tasks (Tasks 1-5), participants were asked to evaluate the system by answering questions in 9 different cognitive dimensions [6]. Results are summarized in Figure 14. No significant differences were found between Blockly and Vipo in each dimension (with all $p > 0.05$), which indicates that both interfaces resulted in similar user experiences.

Correctness and Time spent

Finally, we acknowledge the time spent and correctness in both interfaces. The time spent is defined as the total time spent on finishing Tasks 1-5. Correctness is defined as the number of tasks that were done correctly by the participants in all five tasks. Two paired t-tests were conducted respectively to see whether there were significant differences in time spent and correctness. No significant differences were found in the correctness for Vipo (M=3.25, SD=1.29) and Blockly (M=3.08, SD=1.44); t(10)=0.4, $p = 0.7$. There was a significant difference in the total time spent (seconds) for Vipo (M=1748, SD=559) and Blockly (M=2566, SD=1075); t(10)=0.4, $p = 0.009 < 0.05$. The results suggest that participants spent less time in completing all five tasks with Vipo.

USER STUDY #2: FUNCTION VS. NON-FUNCTION

In this study, we investigated the pros and cons of supporting *functions* in the spatial domain. Participants were asked to program tasks in two conditions: using Vipo with functions (**condition A**) and without functions (**condition B**).

Experiment

Ten (10) participants were recruited. Most were engineering students aged 20-31 years old. Six (6) were novice programmers (0-1 years experience), two (2) were beginners (1-3 years experience), and two (2) were experienced programmers (3+ years experience). Two (2) had previous experience with visual programming in LabView (<6 months of usage).

Participants started with a video tutorial that explained all features of the Vipo language except functions, and then completed three tasks (Task1-Task3) as warm-ups. The authors verified the accuracy of warm-up tasks and explained any issues that occurred. Next, each participant was asked to use both condition A and condition B to program tasks in counterbalanced order. They had to complete three tasks in each condition (Tasks 4a-6a for condition A and Tasks 4b-6b for condition B), then fill in a questionnaire, and proceed to the other condition. For condition A, participants needed to watch a second video tutorial that included functions. Once they have done both conditions, an exit questionnaire was given to compare both. While participants were programming, the computer screen was recorded.

Tasks in both conditions were the same, except that condition A required the use of function calls while condition B required the use of basic notations directly. For example, in Task 5, participants were told that they act as a maintenance worker trying to fix a broken machine. Therefore, they need to program robots to collect tools as well as three types of replacement parts from inventories, use a cutting machine to cut to specific shapes, and then carry to the broken machine. In condition A, participants can define a function called "getPart" and call it three times with different numbers and locations.

Results & Discussion

This section reports results from the questionnaire, and measures of report efficiency and accuracy based on screen recording and analysis of programs.

Functions had less viscosity

The questionnaire asked participants to answer questions in the 9 cognitive dimensions. A paired t-test was conducted for each dimension. We did not find a significant difference in eight dimensions. However, there was a significant difference in viscosity between condition A (M=4.50, SD=0.53) and condition B (M=3.70, SD=1.16), $p = 0.04 < 0.05$. The reason might be that participants in condition A only need to update inside the function being called, while participants in condition B need to update all occurrences.

Functions made programming faster

The total time of completing Tasks 4-6 was measured. A paired t-test was conducted to compare the total time. No significant difference was found between condition A (M=17.1) and condition B (M=20.9), t(9)=-2, $p = 0.1$. However, if we look at Task 5 where the participants were asked to program robots to do the same operation three times, the usage of function saves time for doing the same operation. As Figure 15 shows, participants in condition A spent about one and a half times longer on the first function call (2.63), compared to condition B (1.96). However, for the remaining two sub-tasks, condition A took about half of the time as that of condition B. This shows that the learning curve of function is higher at first but it helps participants be more efficient once mastered. Participants in condition A can reuse the workflow while those in condition B have to repeat the same steps. The time in condition A of making a function call is constant while the time in condition B is proportional to the number of steps.

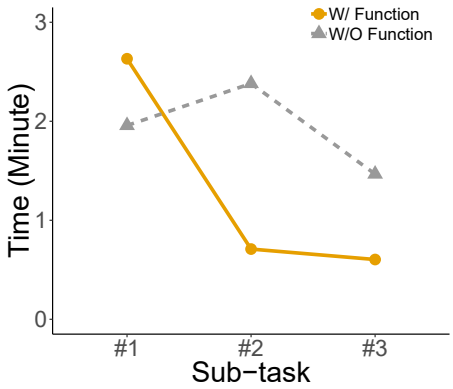

Figure 15. The time spent on three subtasks of Task 5. Learning to use function takes time, but the efforts pay off when the same operation is being used several times.

Functions were more error-prone

The accuracy of the programs was measured. A paired t-test was conducted to compare the number of correct tasks in two conditions. There was a significant difference in the accuracy between condition A (M=2.20, SD=0.63) and condition B (M=2.70, SD=0.48), t(9)=-2, $p = 0.05$. The result suggests that participants made more errors when using functions, compared to using just basic constructs. This is surprising but understandable. Participants need to learn how to accurately use function in a short time which includes: defining a parameter for the function, replacing the constant number in the function with the parameter, and finally passing a different value to the parameter. These extra steps were challenging for novice programmers and increased the error proneness. After checking the created programs, we found one bug that was particularly common and resulted in the significant difference: five participants in condition A forgot to replace the constant number in the function with the defined parameter. This suggests improving the editor by highlighting unused variables.

LIMITATIONS AND FUTURE WORK

The system is designed to program tasks for one robot. In fact, a task may need different types of robots or a collaboration of multiple robots. Such complexity is abstracted away from the task planning layer (Vipo), but a more intelligent ROS Master is required to manage the execution. Furthermore, we assume a robot is executing one task at a time. In a real factory, the robot may be shared among different tasks, in which efficient scheduling and optimization algorithms are required [28].

Vipo was not tested with workers within real factories, because some issues should be addressed first, such as adapting existing machines to use RDF messages, and handling exceptions during execution.

The current interface has limited visualization on the real-time status of machines (only the location and success/failure). In the future, it would be more informative to allow users to customize the visualization of more kinds of status.

In the future, we envision bringing humans into the workflow more explicitly. Workers would broadcast their location and other available status information via RDF messages. In such case, users will be able to program tasks to control the collaboration between humans, mobile robots, and machines.

CONCLUSION

Vipo supports modular visual programming of robot-IoT workflows in the spatial context of the operating environment. Using the Vipo IDE, users can create, test/simulate, and deploy/monitor automations visually, spatially, and interactively. We implemented two use cases to demonstrate that 1) the Vipo language can support complex programs involving recursive functions and 2) the tasks programmed in Vipo can be executed by robots and machines in physical environments. The user study (n=22) found that 1) spatial constructs improve program comprehensibility, and 2) functions speed up programming.

ACKNOWLEDGMENTS

We would like to thank Senthil Chandrasegaran for valuable feedback, and Manav Wadhawan for assistance fabricating and assembling the prototype for use cases. This work was partially supported by NSF FW-HTF 1839971 and OIA 1937036, as well as Purdue Research Foundation (College of Engineering EFC) and the Donald W. Feddersen endowed chair.

REFERENCES

[1] Colin Archibald and Emil Petriu. 1993. Model for skills-oriented robot programming (SKORP). In *Applications of Artificial Intelligence 1993: Machine Vision and Robotics*, Vol. 1964. International Society for Optics and Photonics, World Scientific, Orlando, FL, 392–403.

[2] Maria Bermudez-Edo, Tarek Elsaleh, Payam Barnaghi, and Kerry Taylor. 2016. IoT-Lite: a lightweight semantic model for the Internet of Things. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, IEEE Computer Society, Washington, D.C., 90–97.

[3] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. 2008. Robot Programming by Demonstration. In *Springer Handbook of Robotics*, Bruno Siciliano and Oussama Khatib (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1371–1394. DOI: http://dx.doi.org/10.1007/978-3-540-30301-5_60

[4] Michael Blackstock and Rodger Lea. 2012. WoTKit: A Lightweight Toolkit for the Web of Things. In *Proceedings of the Third International Workshop on the Web of Things (WOT '12)*. ACM, New York, NY, USA, 3:1–3:6. DOI: http://dx.doi.org/10.1145/2379756.2379759

[5] Michael Blackstock and Rodger Lea. 2014. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *Proceedings of the 5th International Workshop on Web of Things (WoT '14)*. ACM, New York, NY, USA, 34–39. DOI: http://dx.doi.org/10.1145/2684432.2684439

[6] Alan F. Blackwell and Thomas R. G. Green. 2000. A Cognitive Dimensions questionnaire optimised for users. (2000). http://ppig.org/library/paper/cognitive-dimensions-questionnaire-optimised-users

[7] Thorsten Blecker and Nizar Abdelkafi. 2006. Mass Customization: State-of-the-Art and Challenges. In *Mass Customization: Challenges and Solutions*, Thorsten Blecker and Gerhard Friedrich (Eds.). Springer US, Boston, MA, 1–25. DOI: http://dx.doi.org/10.1007/0-387-32224-8_1

[8] Blockly. Retrieved 2019. Blockly. (Retrieved 2019). https://developers.google.com/blockly/

[9] Simon Bøgh, Oluf Skov Nielsen, Mikkel Rath Pedersen, Volker Krüger, and Ole Madsen. 2012. Does your robot have skills?. In *Proceedings of the 43rd International Symposium on Robotics*, Vol. 6. Verlag, VDE Verlag GMBH, Berlin, Germany, 1–6.

[10] Yuanzhi Cao, Zhuangying Xu, Fan Li, Wentao Zhong, Ke Huo, and Karthik Ramani. 2019. V.Ra: An In-Situ Visual Authoring System for Robot-IoT Task Planning with Augmented Reality. In *Proceedings of the 2019 on Designing Interactive Systems Conference (DIS '19)*. ACM, New York, NY, USA, 1059–1070. DOI: http://dx.doi.org/10.1145/3322276.3322278

[11] Stéphane Conversy, Jérémie Garcia, Guilhem Buisan, Mathieu Cousy, Mathieu Poirier, Nicolas Saporito, Damiano Taurino, Giuseppe Frau, and Johan Debattista. 2018. Vizir: A Domain-Specific Graphical Language for Authoring and Operating Airport Automations. In *UIST 2018, 31st ACM Symposium on User Interface Software and Technology* (2018-10) *(UIST '18 The 31st Annual ACM Symposium on User Interface Software and Technology)*. ACM SIGCHI, New York, NY, USA, Pages 261–273/ ISBN: 978–1–4503–5948–1. DOI: http://dx.doi.org/10.1145/3242587.3242623

[12] C. Datta, C. Jayawardena, I. H. Kuo, and B. A. MacDonald. 2012. RoboStudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012-10). IEEE, Vilamoura, Algarve, Portugal, 2352–2357. DOI: http://dx.doi.org/10.1109/IROS.2012.6386105

[13] J. P. Diprose, B. A. MacDonald, and J. G. Hosking. 2011. Ruru: A spatial and interactive visual programming language for novice robot programming. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2011-09). IEEE, New York, NY, USA, 25–32. DOI: http://dx.doi.org/10.1109/VLHCC.2011.6070374

[14] Barrett Ens, Fraser Anderson, Tovi Grossman, Michelle Annett, Pourang Irani, and George Fitzmaurice. 2017. Ivy: Exploring Spatially Situated Visual Programming for Authoring and Understanding Intelligent Environments. In *Proceedings of the 43rd Graphics Interface Conference* (2017) *(GI '17)*. Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 156–162. DOI: http://dx.doi.org/10.20380/GI2017.20 event-place: Edmonton, Alberta, Canada.

[15] T. Fong, C. Thorpe, and C. Baur. 2003. Multi-Robot Remote Driving with Collaborative Control. *IEEE Transactions on Industrial Electronics* 50, 4 (Aug. 2003), 699–704. DOI: http://dx.doi.org/10.1109/TIE.2003.814768

[16] L.A. Grieco, A. Rizzo, S. Colucci, S. Sicari, G. Piro, D. Di Paola, and G. Boggia. 2014. IoT-Aided Robotics Applications. *Comput. Commun.* 54, C (Dec. 2014), 32–47. DOI: http://dx.doi.org/10.1016/j.comcom.2014.07.013

[17] Justin Huang and Maya Cakmak. 2017. Code3: A system for end-to-end programming of mobile manipulator robots for novices and experts. In *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI*. IEEE, IEEE, New York, NY, USA, 453–462.

[18] Justin Huang, Tessa Lau, and Maya Cakmak. 2016. Design and Evaluation of a Rapid Programming System for Service Robots. In *The Eleventh ACM/IEEE International Conference on Human Robot Interaction (HRI '16)*. IEEE Press, Piscataway, NJ, USA, 295–302. http://dl.acm.org/citation.cfm?id=2906831.2906883

[19] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-S-A publish/subscribe protocol for GWireless Sensor Networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*. IEEE, IEEE, New York, NY, USA, 791–798.

[20] Ke Huo, Yuanzhi Cao, Sang Ho Yoon, Zhuangying Xu, Guiming Chen, and Karthik Ramani. 2018. Scenariot: Spatially Mapping Smart Things Within Augmented Reality Scenes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 219, 13 pages. DOI:http://dx.doi.org/10.1145/3173574.3173793

[21] ifttt. Retrieved 2019. IFTTT. (Retrieved 2019). https://ifttt.com

[22] J. Jackson. 2007. Microsoft robotics studio: A technical introduction. *IEEE Robotics & Automation Magazine* 14, 4 (2007), 82–87. DOI: http://dx.doi.org/10.1109/M-RA.2007.905745

[23] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: Sketching Dynamic and Interactive Illustrations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 395–405. DOI: http://dx.doi.org/10.1145/2642918.2647375

[24] James Floyd Kelly. 2010. *Lego Mindstorms NXT-G Programming Guide*. Apress, New York City, New York, USA.

[25] Kexi Liu, Daisuke Sakamoto, Masahiko Inami, and Takeo Igarashi. 2011. Roboshop: Multi-Layered Sketching Interface for Robot Housework Assignment and Management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 647–656. DOI: http://dx.doi.org/10.1145/1978942.1979035

[26] Joaquin López, Diego Pérez, and Eduardo Zalama. 2011. A framework for building mobile single and multi-robot applications. *Robotics and Autonomous Systems* 59, 3-4 (2011), 151–162.

[27] Eric Miller. 1998. An Introduction to the Resource Description Framework. *Bulletin of the American Society for Information Science and Technology* 25, 1 (1998), 15–19. DOI: http://dx.doi.org/10.1002/bult.105

[28] Gérard Morel, Carlos Eduardo Pereira, and SY Nof. 2019. Historical survey and emerging challenges of manufacturing automation modeling and control: A systems architecting perspective. *Annual Reviews in Control* 47 (2019), 21–34.

[29] Mikkel Rath Pedersen, Dennis Levin Herzog, and Volker Krüger. 2014. Intuitive skill-level programming of industrial handling tasks on a mobile manipulator. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, IEEE, New York, NY, USA, 4523–4530.

[30] Emmanuel Pot, Jérôme Monceaux, Rodolphe Gelin, and Bruno Maisonnier. 2009. Choregraphe: a graphical tool for humanoid robot programming. In *RO-MAN 2009-The 18th IEEE International Symposium on Robot and Human Interactive Communication*. IEEE, IEEE, New York, NY, USA, 46–51.

[31] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software in robotics* (2009), Vol. 3. IEEE, New York, NY, USA, 5.

[32] Partha P. Ray. 2016. Internet of Robotic Things: Concept, Technologies, and Challenges. *IEEE Access* 4 (2016), 9489–9500. DOI: http://dx.doi.org/10.1109/ACCESS.2017.2647747

[33] Michaela R Reisinger, Johann Schrammel, and Peter Fröhlich. 2017. Visual languages for smart spaces: End-user programming between data-flow and form-filling. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, IEEE, New York, NY, USA, 165–169.

[34] Leonard Richardson and Sam Ruby. 2008. *RESTful web services*. " O'Reilly Media, Inc.", Sebastopol, CA.

[35] Fetch Robotics. Retrieved 2019. Cloud Robotics and Automation: FetchCore from Fetch Robotics. (Retrieved 2019). https://fetchrobotics.com/

[36] Peter Saint-Andre. 2011. *Extensible messaging and presence protocol (XMPP): Core*. Technical Report. Cisco.

[37] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. *The constrained application protocol (CoAP)*. Technical Report. Universitaet Bremen TZI.

[38] Franz Steinmetz, Annika Wollschläger, and Roman Weitschat. 2018. RAZER—A HRI for Visual Task-Level Programming and Intuitive Skill Parameterization. *IEEE Robotics and Automation Letters* 3, 3 (2018), 1362–1369.

[39] Daniel Szafir, Bilge Mutlu, and Terrence Fong. 2017. Designing planning and control interfaces to support user collaboration with flying robots. *The International Journal of Robotics Research* 36, 5-7 (2017), 514–542. DOI:http://dx.doi.org/10.1177/0278364916688256

[40] Steve Vinoski. 2006. Advanced message queuing protocol. *IEEE Internet Computing* 10, 6 (2006), 87–89.

[41] Igor Zubrycki, Marcin Kolesiński, and Grzegorz Granosik. 2017. Graphical programming interface for enabling non-technical professionals to program robots and internet-of-things devices. In *International Work-Conference on Artificial Neural Networks*. Springer, Springer, Cham, New York, NY, USA, 620–631.