HolonCraft – An Architecture for Dynamic Construction of Smart Home Workflows

Zhuo Wang*, Yehia Elkhatib*, Abdessalam Elhabbash†
*School of Computing Science, University of Glasgow, United Kingdom
†School of Computing and Communications, Lancaster University, United Kingdom
Email addresses: zhuowangwork@gmail.com, yehia.elkhatib@glasgow.ac.uk, a.elhabbash@lancaster.ac.uk

***Abstract*—Smart home systems have developed rapidly and entered the daily lives of many people. However, it is difficult for products from different manufacturers to work together due to the interoperability barriers posed by divergent hardware, communication protocols and APIs. In this paper, we introduce HolonCraft as a tool to help the regular IoT user in building complex systems. HolonCraft builds on the Holon ontology which enables systems to share their descriptions so that devices can recognize and understand each other and their respective functions, and work together through composing a system of systems (SoS) at runtime to fulfill a given user workflow. HolonCraft extends the machine-readable information provided by the Holon ontology and uses open-source editors (namely Google Blockly and Microsoft Monaco) to implement a cross-platform graphical programming editor for smart home automation. HolonCraft automatically transforms device capabilities into visual programming elements, enabling users without programming backgrounds to design smart home automation through simple drag-and-drop operations while also supporting advanced programming features such as branches, loops and functions. It then type-checks the composed workflow and, accordingly, generates code to actualize it. In consequence, HolonCraft dramatically improves the abstraction and expressiveness of automation as indicated by our user experiments.**

***Index Terms*—IoT, Smart Home Automation, Ontology, Visual Programming Language**

I. Introduction

IoT devices have been developing at a swift pace, especially in the smart home market. Smart home systems (SHS) have the potential to improve quality of life through added visibility and control of household resources and services. The value added by SHSs comes mainly from the integration of smart devices and the utilization of automated instrumentation that executes workflows to realize the functionalities of the smart home such as optimizing energy consumption, integrating with healthcare systems and securing the home environment, among others.

To enable reaping such rewards, a range of solutions have been developed to enable users to create custom and sophisticated home automation services. On one end of the spectrum, smart home platforms from big companies (*e.g.,* Amazon Echo, Apple HomeKit, Google Nest Hub) are geared towards end-users without a programming background. These tools are fairly easy to use via mobile applications or voice assistants. However, the automation they provide is limited to basic capabilities of the archetype "*if X happens, do Y*" using only compatible devices. However, they are not designed for automation that is more sophisticated than that. On the other end of the spectrum, open-source smart home platforms require authoring code and/or description files using general-purpose programming languages to implement more complex automation programs. For instance, Home Assistant[1] uses YAML to describe automation. Effectively, such platforms expect users to be knowledgeable in programming and debugging, which casts such solutions out of reach of many users.

Graphical programming is a popular approach to reach a balance between usability and functionality through lowering the level of required knowledge and experience needed for creating programs. In the context of SHS, there is a handful of graphical automation solutions such as Node-RED[2], Scratch for Arduino[3], Smart Block [1] and My IoT Puzzle [2]. However, despite their advantages, they offer limited flexibility as they provide a fixed set of graphical elements for a specific set of properties and functionalities of smart devices. Given the heterogeneity and scale of smart devices, a dynamic approach for discovering device properties and functionalities is required to ensure graphical elements are available to represent each property and functionality of the smart device.

We address this limitation by presenting *HolonCraft* as an open-source solution for graphical modeling of smart home workflows. It provides the capability to dynamically create graphical elements based on an ontological description of smart home devices, specifically the Holon ontology [3], [4]. This solution has two main advantages. First, it guarantees that for each device used in the smart home there are graphical elements to represent its properties and functionalities. Second, it lowers the required programming knowledge and experience for creating smart home workflows. Our contributions are:

1) An extended **ontology** (§III) for encompassing different elements of a SHS environment, from the end-user to various devices and the services they offer.
2) The HolonCraft **software architecture** (§IV) which includes an ontological parser, a deployment orchestrator, and associated code generation units.
3) An Android app and a web interface as **workflow editors** (§V-B), each of which enables the average user to design SHS automation through drag-and-drop operations. The editors allow the writing of relatively complex control flows,

[1]https://www.home-assistant.io/
[2]https://nodered.org/
[3]http://s4a.cat/

and support programming constructs such as variables and functions.

4) Server-side modules (§V-A) for: (i) automatic ontological parsing of device descriptions and translating them into graphical programming elements, and vice versa, as well as type-checking block connections; (ii) code generation to produce executable code to match a given user-designed work ow (we implement Python generators but this could be easily substituted with another module for different programming languages).
5) A mixed-methods evaluation(§VI) of HolonCraft's usefulness and ease-of-use. For this, we employ both software testing and user experiments.

II. Related Work

Current efforts could be categorized under two main approaches:smart home applicationsand graphical programming frameworks.Solutions of the former approach typically offer easy-to-use editors to create automation programs, but only support elementary operations making it challenging to accomplish non-basic tasks. In contrast, open-source smart home platforms commonly use code or graphical editors that requires experience to writing code for creating work ows or extending platform features. Table I summarizes the comparison between existing solutions and our solution, HolonCraft.

A. Smart home applications

Most mainstream SHSs already have built-in editors to create automation programs, which are very easy to use but only support elementary operations. Consider Apple HomeKit[4] as an example. It can be used to compose only IF-THEN-style automation, such as turning on the light when a motion sensor detects any activities. Such a simple work ow takes only ve steps, but it can only customize the trigger events and perform a single action on the actuators. Its sophistication pales in comparison to even the iOS Shortcuts app.

B. Graphical programming frameworks

There have been many graphical programming solutions for smart home systems, of which we only discuss environments for smart homes and GUI rule builders.

Node-Red[5] is a Node.js-based platform for building event-driven applications. It uses JSON to describe work ows and to provide a ow-based programming editor in web browsers. The editor allows users to design ows, de ne JavaScript functions and deploy them with a single click. However, JavaScript is a dynamically and weakly-typed language. In effect, numerous errors could not be detected at edit time, making it hard to use for `error-free' composition of complex applications. In addition, it does not compile work ows into native binary. When deployed on Arduino, for instance, users need to rst installFirmata[6] on the board. The same limitation applies to Scratch for Arduino (S4A), a project based on the MIT Scratch graphical programming platform. It provides custom blocks to control actuators and sensors attached to PicoBoards and Arduino boards.

Project	Type	Auto-gen. Elements	Code Generation	Error Prevention
IFTTT	GRB	é	é	é
Smart Rules	GRB	é	é	é
Node-Red	Flow	é	é	é
S4A	Block	é	é	é
BlocklyDuino	Block	é	Ë (C)	Static typed
My IoT Puzzle	Block	é	é	Con ict Detect
Smart Block	Block	é	Ë (Groovy)	Static typed & Con ict Detect
HolonCraft	Block	Ë	Ë (Python)	Type Check

TABLE I
COMPARISON WITH OTHER GRAPHICAL AUTOMATION EDITORS NOTE: GRBSTANDS FORGUI rule builder.

BlocklyDuino is a Google Blockly-based graphical programming platform for Arduino[7]. It has a web editor that already contains standard and custom blocks for general Arduino input, output, and common peripherals. The platform can generate Arduino code and Blockly XML. BlocklyDurino blocks are statically-typed, which means that users cannot combine mismatched blocks together.

My IoT puzzle is a GUI tool for designing IF-THEN rules [2]. The project focuses on debugging and error prevention. It can detect endless loops, redundancy and inconsistency in user work ows, and then display both the textual and graphical explanation [2] and suggest means of resolution. Currently, it can run in a simulation environment and does not support code generation. Finally, Smart Block [5] is a visual block language for Samsung Smart Things[8]. After the user has designed their program using Blockly, Smart Block can produce an Abstract Syntax Tree (AST) and compile it into Groovy code.

III. Thing Ontology

In this section, we give an overview of the semantic technologies we use to facilitate seamless integration of different devices in a SHS.

A. Background: What are holons?

During their operation, IoT devices constantly need to communicate with each other. As IoT systems do not exist in isolation but instead overlap by sharing physical spaces and operational boundaries, the need also arises for systems to intercommunicate. This requires systems to understand their own properties and functions, and those of other systems. Complicating this further is the fact that the majority of IoT systems do not conform to a standard set of descriptions of device properties, services and other metadata.

From the different efforts made for IoT interoperability, the concept of aholon was proposed [3] to enable the description of device and system properties in a manner that

[4]https://www.apple.com/uk/ios/home/
[5]https://nodered.org/
[6]http:// rmata.org/
[7]https://blocklyduino.github.io/BlocklyDuino/
[8]https://www.smartthings.com/

is comprehensible by other systems. Under this architecture, each constituent system (e.g., smart home device) is a holon that can be described, manipulated, and reasoned about in a program at design- and run-time.

With the holon as a programmatic concept, developers do not need to know the underlying API and hardware details when thinking about complex systems, or systems of systems. There is also no need for cataloging services (e.g., [6]) as they are discovered at runtime. Instead, system of systems developers only need to specify an abstract application workflow consisting of tasks, each of which represents an abstract service. Then, at runtime, the holonic architecture will realize the workflow by looking to identify deployed services to implement each abstract service.

We developed such architecture for interoperability and reasoning [4], and for discovery and adaptive composition [7]. This offers strong abstracting power for distributed programming, and enables systems to understand their own hierarchical constitution and capabilities as well as those of other systems and then use this understanding to reason about composition to form a larger system of systems.

We note that some other works have also defined the notion of a holon (e.g., [8], [9]) but focused on goal-driven service composition without means of allowing holons to reason and self-compose.

B. Smart home ontology

We now present the ontology and how we extended it by domain knowledge for SHS settings. This ontology will be utilized by HolonCraft to automate the generation of the ontological description of smart home devices.

The holon ontology defines classes to specify system functionalities and properties. It also specifies relationships to link the defined classes and described systems through parent-children relationships. A holon also details the functionalities supported by the system. A holon's general properties are:

- Timestamp: to identify the snapshot of the system.
- HolonId: a unique identifier.
- Name: a non-unique name.
- Address,Port: the IP address and communication port of the device representing the holon.
- Mobility: a class defining the mobility profile of the device e.g., random way point.
- Reliability: a class defining the reliability profile of the device/system, including best-effort, at least once, at most once and exactly once guarantee levels.
- Messaging protocol: to be used for data exchanged with other systems e.g., MQTT and CoAP.
- Services: define the functionalities that are supported by each device in the system.

In addition to the above generic holon structure, we added the following properties for smart home implementations:

- Type: specifying the device type, e.g., sensor or actuator.
- Events: a class representing certain reactions a device may trigger in response to some events; for instance, an “activity detected” event for a motion sensor.
- Model type: defines variant models of a certain a device/system type, e.g., different models of a smart light.

The Serviceclass contains the following properties:

- Name: the name of the service.
- Parameters: input parameters to the service.
- URL: a reference to access the exposed service.
- Cost: the cost of accessing the service, which can be defined as a simple attribute (e.g., price, delay) or a composite attribute using a set of simple ones. This is used to reason about selecting a service in cases when similar services exist.
- Return: a Boolean indicating whether the function has a return value.
- Return Type: if Return is true, Return Type represents what is returned which is used in HolonCraft's type checking system.
- Message: represents the textual content of the Blockly block that represents the device.
- CustomCodeGen: a reference to a custom code generator, if any; otherwise, HolonCraft will create a general one.

IV. Design

This section details the design principles of the HolonCraft system, then describes its building blocks.

A. Design Principles

Abstraction. HolonCraft aims to raise the level of abstraction provided to designers of SHS workflows. HolonCraft is to enable users to design a workflow then generate executable Python code of the designed workflow. In addition, HolonCraft should programmatically create a holon description of a compatible device and parse OWL files and convert them into block definitions. This would allow manufacturers to design holon ontologies to make their products compatible with HolonCraft.

Interactivity. The literature indicates that user interface complexity is one of the most significant problems of early efforts on SHS automation [10]. Thus, the HolonCraft user interface should provide an easy-to-use graphical programming interface. The interface needs to allow the user to freely add and combine blocks through drag-and-drop actions, as well as enter values to adjust parameters of the SHS workflow elements. Additionally, the user interface could give feedback (e.g. haptic) based on user actions, such as automatic attraction of blocks and immediate indication of connecting to other blocks. Finally, the interface should then generate code to realize the workflow, relieving the user from the complexities of such task.

Easy of use. As the target users of HolonCraft are mainly non-professionals with no programming experience, all upload, export, and edit operations are designed with the fundamental principle of ease of use. The interface is to be intuitive so that users are able to master the basic usage in a short period of time. Categorizing elements according to their functionalities would make it easier for users to find what they want. The interface could also provide flexibility so that users are able

Fig. 1. The architecture of the application

to move contents within the interface, and zoom in and out of the workspace.

Extensibility. HolonCraft should allow any devices or systems with Holon-compliant OWL documents to be loaded to the interface. The application should also be able to generate code in any programming language through minor modi cation through swapping the code generator.

Interoperability. As SHS users tend to use various devices, HolonCraft components need to be interoperable and usable across different platforms. Components that require high computing performance can be extracted separately and placed on the server-side. UI components and code generators should be manageable through different platforms.

Error handling. HolonCraft should avoid type errors by type-annotating the blocks, and type-checking them as the user edits them. For instance, the editor should prohibit the user from interlinking blocks that do not match the type. For error handling, the application should restrict user input when editing work ows by only allowing users to import valid OWL les to load blocks.

B. System architecture

The system architecture is depicted in Figure 1, where components are arranged between the client- and server-side. The server-side parses the ontology and automatically generates blocks and code generators to populate the user interface. The client-side includes the automation editor and the code generator. The editor allows users to build a graphical model of the smart home work ow. This results in an ontological description of the composed work ow. After validation, this is used to generate the code that realizes the user's work ow. To make our solution easier to transport to more platforms, we moved the part that requires high computing performance to the server-side, and kept the client-side mainly responsible for actions speci c to the user's work ow. Figure 2 illustrates the order of interactions between the user and the client- and server-side parts of HolonCraft.

The model built by the client will be passed to the server-side for processing. The model After the device description, information is collected from the IoT and sent to the server. The server will parse the ontology and automatically generate blocks and code generators and send them back to the application. To improve extensibility, the visual code editor also

Fig. 2. A time sequence diagram of user interactions and the subsequent system processes.

uses the JavaScript version of the Blockly open-source project, which is a pure client-side JavaScript framework that can be easily extended to more platforms in the future [11].

V. Implementation

This section details the implementation of both the client and server sides of the system. Both the server-side and the Android app of the client-side were developed using the Kotlin language. The web client was developed using JavaScript. The code bases are released as open-source at https://github.com/WangZhuo2015/HolonCraft_Server and https://github.com/WangZhuo2015/HolonCraft_Android.

A. Server side

The server is responsible for the website services and APIs. It is developed using Ktor[9], an asynchronous non-blocking web framework written in Kotlin. Due to Kotlin's excellent compatibility with Java, we are able to use mature OWL libraries written in Java to parse OWL les and manipulate ontologies.

Holon creation. Users can create Holon and services programmatically in a few lines of Kotlin code, such as the example shown in listing 1.

```
val light1 = Holon()
val switchService = Service()
initService(switchService,
"switch",
"switch %1 %2",
parameters = "dropdown(on/off),device" )
initHolonDevice(light1,
"Living Room Light",
type = HolonType.ACTUATOR,
service = arrayListOf(switchService),
events = ArrayList() )
OntologyCreator.creatOntology(light1)
```

Listing 1: An example of programmatic creation of a holon

[9] https://ktor.io/

OWL files are processed on the server using the core classes OntologyCreator and OntologyParser. The former provides methods to transform Holon objects into OWL Ontology objects through the OWL API, and can automatically add Holon related properties and relationships into the ontology. The latter is responsible for parsing and converting OWL files into Holon objects.

Restful APIs. The server exposes the following endpoints:

1) POST: /api/description
   Allows clients to upload Holon compatible device descriptions in OWL2 format to the server for parsing.
2) GET: /api/load-blocks
   Analyzes device description, transforms it into Blocks through the templates, and sends the corresponding Blocks definition and code generator back to the client.
3) POST: /api/result-report, parameters: task, name
   Uploads the results completed by the user to the server. The request body includes the task number, the time spent to complete the task, the XML tree generated by the task and the final generated Python code. Upon receipt, the server will look for the corresponding file based on the name, and if not, create a file and then append the information about the user completion.

For each physical device, there is a virtual object of type Holon that represents it. In this virtual representation, the device type, message, and name are mapped to the object attributes. The Holon objects then automatically generate the device blocks.

B. Client side

We have developed a native Android client for tablets/mobile devices, and a web client for general usage.

Web application. Most of the interactions with the user are implemented in the Web interface. All communication to server is sent using XMLHttpRequest.

The left half of the page is the editor, which is essentially a graphical programming interface based on the Blockly project. It is divided into the toolbox and workspace. The upper four sections in the toolbox are custom blocks we created. When the user switches to a different task, it will automatically clear the workspace, load the corresponding block from the server and put the blocks into the appropriate section by block name prefix.

The right half of the page contains the information and code area based on the Monaco editor, the core component of Microsoft VS Code. The information area is for users to enter names, switch tasks, start the timer and upload the results, in addition to an introduction to the experiment, a tutorial on the HolonCraft editor and a list of tasks.

By listening to Blockly events, HolonCraft can immediately generate the Python code when block_move or block_change events are detected by the event listener, giving real-time feedback. Compared with the regular text area widget, the code area can highlight keywords, variable names, function names, and values by a different color, making the Python code more readable.

Upon clicking the finish button, the application sends the user's name, XML tree and the generated Python code file to the server.

Android App. We intended to place the OWL file parsing module on the client-side application in the initial plan. However, in the development process, we found that the Android platform currently only has two popular ontology libraries, Apache Android Jena and OWL API, of which Jena-Android has not been updated for seven years and only supports some of the OWL API1.1 features. OWL API 4 and 5 use javax, and this part of Java features are not provided by Android, so the new version of OWL API has compatibility problems with the Dalvik virtual machine on Android. As such, the Android client cannot be used without an internet connection in this version. The client uploads the description information of the device to the server, which responds with the definition of graphic programming blocks, the code generator part for code generation, and other necessary contents to the client.

The Android app contains three main activities:

- SettingsActivity – for users to enter their names and select the task they want. When using edit the name or task selected, it will notify Task Manager to change its record.
- EditorActivity – here, editor.html and related Javascript and CSS files are loaded into a webView, the Kotlin part communicates with JavaScript through the interface class. The timer is also implemented by native Kotlin code. After the user submits, the result are stored in Task Manager.
- ResultViewerActivity – as Android devices have a smaller screen, a separate code viewer was created for users to check XML and Python code.

Components. Both the web application and the Android app provide the following components to model smart home workflows:

- Blocks design – The HolonCraft editor uses JSON format to define the blocks, in order to be compatible with the Blockly framework. We abstract things involved in smart home automation into four categories of programming blocks.
- Sensors & Actuators – We divided the devices into the following two categories based on user feedback during the initial evaluation phase to better differentiate the devices.
  1) Sensors represent the various sensors and switches in a smart home system, providing services such as triggering events and sensed values.
  2) Actuators represent the parts of a smart home system that can be controlled to do some action, e.g., lamps, curtains, and sound systems.

The Holon class generates both types of sensor and actuator device blocks. A device block is prefixed with the type of the block (i.e., sensor or actuator), and is defined in Listing 2. If the user does not assign a value for the model type property, the value will be the same as the device's name. However, a type of service is commonly supported by a certain kind of device, so we can set the model type

Fig. 3. The interface of the web-based HolonCraft client application.

of these devices with the same value so that these service blocks can be combined with any instance of this type of device.

```
{
  "holonId" : "ACTUATOR_Living_Room_Light",
  "type" : "actuator_Living_Room_Light",
  "message0": "%1",
  "args0" : [{
      "type" : "field_label_serializable",
      "name": "NAME",
      "text" : "Living_Room_Light" }],
  "output" : "Living_Room_Light",
  "colour" : 30,
  "tooltip" : "",
  "helpUrl" : "to be added later" }
```

Listing 2: Example of defining a block representation of a smart device

Events. HolonCraft enables defining events to trigger the execution of the workflow. The supported events types are sensor reading- and time-triggered events.

Services. Services are operations performed by smart home hardware or systems, such as opening and closing of a smart curtain. Operations that depend on specific hardware need at least one device as their parameters, while services that do not depend on hardware execution, such as waiting for n seconds, can be executed without a specific device. According to the experimental results of the first version of HolonCraft, parameters are generally limited to 2 or fewer to facilitate user understanding. Service is the core part of smart home automation. Here we focus on the blocks generation of services.

- Parameters property is used to describe the parameters needed for the service, in which we made some special conventions, such as pointing to the parameters that own this device. Fixed named device. This parameter location will automatically bind the type of the device and will be type-checked during editing. The program will automatically generate the dropdown menu parameter box, the content inside the brackets to / split into dropdown menu options.
- Message property generates a custom blocks template, where %1 to %n is used to represent the first n parameters. This property directly affects the appearance of the graphical programming blocks, e.g., message0: ``%1 is %2" .
- Service Type is one of two categories: action type – services that perform an operation with no return value (e.g., open/close such action); and value type – operations that will have a numerical return value (e.g., read brightness).
- Return Type of value type services.

Block definition generation – To implement the block generation, we define the Kotlin interface BlockJSONExportable which export supported classes to a set of blocks supported by Blockly. The Holon, Service, and Event classes implement this interface. The application will gain data from the holon ontology or user-created holon objects, and then fill them into a pre-defined JSON template to achieve this goal.

Code generator – The same design has been followed for the code generator, using the CodeGeneratorExportable interface that exports JavaScript definitions for a Holon and its services and events, to then be fed to the appropriate code generator. To support different programming languages with minor modifications, the parameter LanguageType specifies the programming language type, and users can define their code generator template. Currently, only Python is supported, but this can be extended with suitable code generators.

VI. Evaluation

We now describe our experiments to examine HolonCraft's effectiveness and usability.

A. Strategy

We designed a set of user experiments to study how non-specialist users use HolonCraft. Specifically, we want to measure HolonCraft's (i) ability in enabling users to create complex workflows, (ii) productivity, (iii) intuitiveness, (iv) and effectiveness of the type-checking system.

To allow more participants to join our experiments, we bundled the Kotlin code (using Gradle Shadow) then deployed the server and web application on cloud servers (through Digital Ocean). Participants were required to fill out a background questionnaire, complete the tasks (see below), and finally complete an exit questionnaire.

Task completion was tracked in 2 ways. First, its duration was measured as the time elapsed between pressing the `Start' and `Finish' buttons. Second, its success was measured by manually verifying each submitted implementation.

B. Tasks

Participants are first provided with a tutorial on how to use HolonCraft, which explains the UI and its 4 main types of elements and then gives a guided example. They are then given 3 automation tasks of increasing complexity, as follows:

T1: A straightforward task of automating the control of lights under a certain condition. It tests whether the user can master basic operations after receiving the tutorial.
Sensors:1 Actuators: 1 Events: Level

T2: A slightly more complex task of automating daily patterns using 1 sensor and different actuators. This requires the use of `IF' statements and numeric blocks to compare sensor levels.
Sensors:1 Actuators: 3 Events: Time

T3: The third task requests the changing of multiple actuator states based on the levels of 2 sensors and on a particular looping logic. This task requires some programming knowledge like functions and loops.
Sensors:1 Actuators: 3
Events: Level, Time, Loop

C. Recruitment

Inline with our goal of supporting inexperienced end users, the main criteria of recruiting participants is the lack of programming experience. For this purpose 17 participants were recruited. The majority (70%) had no programming experience as evident by the use of at least one programming language or knowledge of basic control flows. Ten participants (58.8%) have used a smart home system before.

D. Findings

We summarize our results in Table II. Task completion time is calculated from the user's interaction with the `Start' and `End' buttons. User difficulty is computed using the task success rate and completion time.

Fig. 4. Participant age and previous programming experience, which in most cases matched their prior experience with smart home automation systems.

TABLE II
Summary of results from the user experiments.

Task	Success Rate	Time (s) Min	Time (s) Avg	Time (s) Max	Avg. User Difficulty
T1	100.00%	20.09	164.49	1027.66	9.412
T2	88.24%	187.09	652.17	1974.97	23.530
T3	5.88%	367.01	367.01	367.01	N/A

As T1 did not involve variables or functions, the success rate was the highest at 100%. Users only need to drag 4 blocks and combine them together. Most participants took less than 100 seconds, although a few mentioned that they took a long time to get familiar with HolonCraft. Overall, though, the majority of users could master the basic operations of HolonCraft in a relatively short period of time.

Despite its increased complexity, the success rate of T2 is relatively high at 88.24%, illustrating HolonCraft suitability (for most users) in handling more challenging tasks. Here, we observed some participants making mistakes, especially in corner cases, e.g., choosing “lower than or equal” instead of “lower than”, or forgetting to choose the “everyday” option in the event block. Over half of the successful participants used two `IF' statements instead of one `IF-ELSE' statement because they did not find how to configure the `IF' block (by clicking on the gear icon). Many participants took a while to put the `IF' and compare blocks together, suggesting that there is some room for further simplification of the blocks.

The success rate of T3, which requires a basic understanding of variables and functions, is only 5.88% as only one participant was able to accomplish it in just over 6 minutes. Their solution was found to be bug-free. The participant had prior programming experience. This outcome demonstrates that even a fairly straightforward graphical editor requires some basic knowledge to use for composing complex automation tasks. We also surmise some possible improvements such as reducing the menu hierarchy to help users find what they want effortlessly, and providing some pre-assembled block combinations to simplify operation.

E. Feedback

Upon completing the experiment, participants were asked to answer the following questions using a 5-point Likert scale: