

VISUAL PROGRAMMING FOR SMART DEVICES: UI GENERATOR, SIMULATOR AND RUNTIME

Dimitrios Linaritis

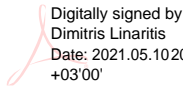
Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science
University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

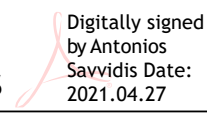
VISUAL PROGRAMMING FOR SMART DEVICES: UI GENERATOR, SIMULATOR AND RUNTIME

Thesis submitted by
Dimitrios Linaritis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: **Dimitris Linaritis**  Digitally signed by
Dimitris Linaritis
Date: 2021.05.10 20:26:04
+03'00'

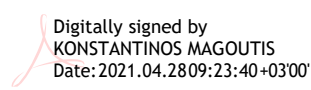
Dimitrios Linaritis, Computer Science Department

Committee approvals: **Antonios Savvidis**  Digitally signed
by Antonios
Savvidis Date:
2021.04.27

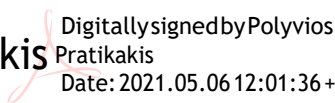
Anthony Savidis
Professor, Computer Science Department, University of Crete,
Thesis Supervisor

DIMITRIOS-STAVROS GRAMMENOS  Digitally signed by
STAVROS GRAMMENOS
Date: 2021.04.27 22:47:31

Dimitrios Grammenos
Principal Researcher, Institute of Computer Science, FORTH

KONSTANTINOS MAGOUTIS  Digitally signed by
KONSTANTINOS MAGOUTIS
Date: 2021.04.28 09:23:40 +03'00'

Kostantinos Magoutis
Associate Professor, Computer Science Department, University
of Crete

Department approval: **Polyvios Pratikakis**  Digitally signed by Polyvios
Pratikakis
Date: 2021.05.06 12:01:36 +03'00'

Polyvios Pratikakis
Assistant Professor, Computer Science Department, University
of Crete

Heraklion, May 2021

Abstract

The Internet of Things (IoT) is the new rapidly-growing domain that is constantly evolving in terms of infrastructures, integrated solutions, development tools and best practices. The availability of so many devices in the environment, for various purposes and missions, entails a critical control challenge, raising issues related not only to security and safety but also to individualization and adaptation. In fact, the main benefit in everyday life is expected by the wide introduction of software automations that can control and coordinate such devices in ways matching individual people needs, preference and requirements. But the demands for such automations are so customized and fluid that the corresponding digital market is currently either non-existent or very limited. Now, one potential solution to this supply-demand gap is enabling users develop directly their own automations. In this context, the adoption of visual programming gained increased attention as a vehicle to enable composition of individualized automations by non-professional developers.

In this thesis, we present a custom toolset, built on top of a recently developed visual programming IDE, which facilitates end-user development, execution and testing of IoT automations. Firstly, an automatic generator is introduced, which produces user-interfaces for smart devices relying on their API specifications. Then, we present a runtime environment for automations that provides advanced monitoring and interaction tools including a device dashboard, a calendar for scheduling automations and a history panel that records and displays device events. Following, we discuss a custom runtime for testing purposes, which offers virtual counterparts of all physical smart devices, so that testing is done locally, in a protected and isolated environment, without requiring operation of the real devices. The latter is possible through our simulator, which enables interactive manipulation of all device properties and operational modes. Additionally, we implemented a time controller (i.e. virtual time) to handle the flow and pace of time during testing, enabling trigger scheduled tasks in a way not interfering with system time. Finally, we outline a case study involving various scenarios of everyday automations.

ΟΠΤΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΓΙΑ ΕΞΥΠΝΕΣ ΣΥΣΚΕΥΕΣ: ΓΕΝΝΗΤΡΙΑ ΔΙΕΠΑΦΗΣ ΧΡΗΣΤΗ, ΠΡΟΣΟΜΟΙΩΤΗΣ ΚΑΙ ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ

Περίληψη

Το Διαδίκτυο των πραγμάτων (IoT) είναι ο νέος ταχέως αναπτυσσόμενος τομέας που εξελίσσεται συνεχώς σε όρους υποδομών, ολοκληρωμένων λύσεων, εργαλείων ανάπτυξης και βέλτιστων πρακτικών. Η διαθεσιμότητα τόσων πολλών συσκευών στο περιβάλλον, για διάφορους σκοπούς και αποστολές, συνεπάγεται μια κρίσιμη πρόκληση ελέγχου, θέτοντας ζητήματα όχι μόνο ως προς την ασφάλεια αλλά επίσης ως προς την εξατομίκευση και την προσαρμογή. Στην πραγματικότητα, το κύριο όφελος στην καθημερινή ζωή αναμένεται να προέλθει από την ευρεία εισαγωγή των αυτοματισμών λογισμικού, οι οποίοι μπορούν να ελέγχουν και να συντονίζουν τις συσκευές με τρόπους έτσι ώστε να αντιστοιχούν στις μεμονωμένες ανάγκες, προτιμήσεις και απαιτήσεις των ανθρώπων. Παρόλα αυτά οι απαιτήσεις για τέτοιου είδους αυτοματισμούς είναι αρκετά εξατομικευμένοι και ρευστοί με αποτέλεσμα η ψηφιακή αγορά να είναι είτε πολύ περιορισμένη είτε εντελώς ανύπαρκτη. Μια πιθανή λύση στο κενό προσφοράς-ζήτησης είναι να δοθεί η δυνατότητα στους χρήστες να αναπτύσσουν τους δικούς τους αυτοματισμούς. Στα πλαίσια αυτά, η υιοθέτηση του οπτικού προγραμματισμού κερδίζει όλο και περισσότερη προσοχή ως μέσο που επιτρέπει την σύνθεση εξατομικευμένων αυτοματισμών από μη επαγγελματίες προγραμματιστές.

Σε αυτή την εργασία, παρουσιάζουμε ένα προσαρμοσμένο σύνολο εργαλείων, που δημιουργήθηκε πάνω σε ένα πρόσφατα ανεπτυγμένο ολοκληρωμένο προγραμματιστικό περιβάλλον (IDE) για οπτικό προγραμματισμό, που διευκολύνει την ανάπτυξη προγραμμάτων από μη προγραμματιστές, την εκτέλεση και τον έλεγχο ορθότητας των IoT αυτοματισμών. Αρχικά, αναπτύχθηκε μία αυτόματη γεννήτρια διεπαφών χρήστη (UI) για έξυπνες συσκευές βασισμένη στις API

προδιαγραφές τους. Στη συνέχεια, παρουσιάζουμε ένα περιβάλλον εκτέλεσης για αυτοματισμούς που παρέχει προηγμένα εργαλεία παρακολούθησης και αλληλεπίδρασης, στα οποία συμπεριλαμβάνονται ένας πίνακας απεικόνισης ιδιοτήτων των έξυπνων συσκευών, ένα ημερολόγιο για προγραμματισμένους αυτοματισμούς καθώς και ένας πίνακας ιστορικού που καταγράφει και εμφανίζει τα εκάστοτε συμβάντα των συσκευών. Έπειτα, παρέχεται ένα προσαρμοσμένο περιβάλλον εκτέλεσης για σκοπούς δοκιμών των αυτοματισμών που προσφέρει εικονικές αντιστοιχίες των φυσικών συσκευών με σκοπό οι δοκιμές να πραγματοποιηθούν τοπικά σε ένα προστατευμένο και απομονωμένο περιβάλλον, χωρίς να απαιτείται η λειτουργία των πραγματικών συσκευών. Το τελευταίο μπορεί να πραγματοποιηθεί μέσω ενός προσομοιωτή που αναπτύχθηκε, ο οποίος επιτρέπει τον διαδραστικό χειρισμό όλων των ιδιοτήτων της συσκευής καθώς και τους τρόπους λειτουργίας της. Επιπλέον, αναπτύχθηκε ένας χειριστής χρόνου (δηλ. εικονικός χρόνος) για τον χειρισμό της ροής και του ρυθμού του χρόνου κατά την διάρκεια των δοκιμών, επιτρέποντας την ενεργοποίηση προγραμματισμένων εργασιών χωρίς να επηρεάζεται από τον χρόνο του συστήματος. Τέλος, περιγράφουμε μια μελέτη περίπτωσης που περιλαμβάνει διάφορα σενάρια καθημερινών αυτοματισμών.

Acknowledgements

First of all, I would like to thank my supervisor, Professor of the University of Crete, Anthony Savidis, first for my trust and then for his continued support. I would also like to thank Yannis Valsamakis for his excellent cooperation, for his continued support and valuable advice. I am also grateful to Principal Researcher Dimitrio Grammeno and Associate Professor Konstantino Magouti for their participation in the Supervisory Committee. I would also like to thank the Department of Computer Science of University of Crete for offering a high level of academic education.

I would also like to thank my second family consisting of all those people whom I love and who have supported me throughout this period. Finally, mainly, I would like to thank my parents Giannis and Athena. I am grateful for all their love and support. Without them, I would not be who I am today.

Στην οικογένεια μου

Contents

VISUAL PROGRAMMING FOR SMART DEVICES: UI GENERATOR, SIMULATOR AND RUNTIME.....	1
Abstract.....	4
ΟΠΤΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΓΙΑ ΕΞΥΠΝΕΣ ΣΥΣΚΕΥΕΣ: ΓΕΝΝΗΤΡΙΑ ΔΙΕΠΑΦΗΣ ΧΡΗΣΤΗ, ΠΡΟΣΟΜΟΙΩΤΗΣ ΚΑΙ ΧΡΟΝΟΣ ΕΚΤΕΛΕΣΗΣ.....	5
Περίληψη.....	5
Acknowledgements.....	7
Contents.....	9
List of Figures.....	14
List of Tables.....	19
1 Introduction.....	20
1.1 Smart Devices in the Internet of Things.....	20
1.2 Automatic User Interface Generation.....	21
1.3 Visual Programming.....	22
1.3.1 Blockly Studio IDE.....	22
1.4 Problem Definition.....	23
1.5 Primary Contributions.....	25
1.6 Thesis Structure.....	25
2 Related Work.....	27

2.1	Middlewares on IoT	27
2.2	Automatic UI Generators.....	30
2.3	Visual Programming for IoT	32
3	System Overview	35
3.1	Architecture	35
3.2	Communication with Devices	36
3.2.1	Simulating Smart Devices.....	38
4	User Interface Generator	43
4.1	Generic Device API.....	43
4.2	MicroUis.....	50
4.2.1	Device Properties	50
4.2.2	Methods and Actions	52
5	Integration with Blockly Studio IDE.....	54
5.1	User Interfaces on device management.....	54
5.1.1	Single Device	54
5.1.2	Device Groups	56
5.2	Visual blocks provided by the IoT domain framework.....	59
5.2.1	Device	59
5.2.2	Device Group	61
5.2.3	Conditional	62
5.2.4	Scheduled	64

5.3	Types of automation provided by the IoT domain framework	65
5.3.1	Automations for Scheduled Tasks	65
5.3.2	Automations for Conditional Tasks	66
5.3.3	Automations for Basic Tasks	67
5.4	Runtime of Automations	68
5.4.1	Device Dashboard	70
5.4.2	Calendar	71
5.4.3	Event History	72
5.5	Automation Testing	73
5.5.1	Device Simulation	75
5.5.2	Tools	76
5.5.3	Tests	78
6	Case Studies	81
6.1	Morning Automations	81
6.1.1	Devices	82
6.1.2	Automations	83
6.1.3	Execution of Automations	84
6.2	Self-Caring Home	86
6.2.1	Devices	87
6.2.2	Automations	88
6.2.3	Execution of Automations	89

6.3	Fire Protection	93
6.3.1	Devices	93
6.3.2	Automations.....	94
6.3.3	Execution of Automations.....	94
7	Conclusions and Future Work	97
	Bibliography	99

List of Figures

Figure 1. The macro-architecture of the system for visual programming support for smart devices.....	36
Figure 2. API implementation for communicating with smart devices	37
Figure 3. Creation of setter and getter methods	38
Figure 4. practically-RESTful API for Air Conditioning smart device	38
Figure 5. Properties of the Air Conditioning smart device	39
Figure 6. Actions of the Air Conditioning smart device	40
Figure 7. IoTivity Simulator included Air Conditioning smart device.....	41
Figure 8. Converting virtual device data to Generic Device API	43
Figure 9. ConvertDevices of Converter library.....	44
Figure 10. Generic Device API definition	45
Figure 11. Property of a smart device definition	47
Figure 12. Method of smart device definition	48
Figure 13. Parameter definition	49
Figure 14. Design of device properties and their automatic rendering with MicroUis.....	51
Figure 15. Update method with its description	52
Figure 16. Scanning devices on the network	55

Figure 17. Air Conditioning device interface which is generated by <i>Automatic UI Generator</i>	56
Figure 18. Air Conditioning device group interface which is generated by <i>Automatic UI Generator</i>	57
Figure 19. Adding Air Condition device to an already defined group	58
Figure 20. Update <i>universal-IDs</i> of the smart device and match it with an existing group	59
Figure 21. Visual programming blocks for device actions	60
Figure 22. Setter and Getter Blockly Blocks for Properties	61
Figure 23. Input/Output for smart device properties in the I/O Console	61
Figure 24. Setter and Output <i>Blockly</i> Blocks for device group	62
Figure 25. <i>When</i> Conditional blocks (A), <i>After</i> Conditional blocks (B)	62
Figure 26. <i>Continue/Break</i> blocks (A), Extra conditional blocks (B)	63
Figure 27. <i>Break/Continue</i> blocks outside of <i>When/After</i> block	64
Figure 28. <i>Break/Continue</i> blocks for <i>Every</i>	64
Figure 29. Blocks for scheduler events	65
Figure 30. An Automation for Scheduled Task	66
Figure 31. An Automation for Conditional Task	67
Figure 32. An example of Automation for Basic Tasks	68
Figure 33. Overview of execution of automations	69
Figure 34. User interfaces for smart devices on runtime generated by <i>User Interface Generator</i>	70

Figure 35. Calendar tool on runtime environment.....	71
Figure 36. (A) “Wait” block with given description (B) Description of block is visualized in organizer with the starting and finishing time.....	72
Figure 37. Event History that includes two “When” conditional events	73
Figure 38. Event History that includes a device action and a property change of smart devices	74
Figure 39. Overview of runtime for automation testing.....	75
Figure 40. Implementation of an action for execution window for automation testing	76
Figure 41. Control virtual devices that participate in execution for automation testing	77
Figure 42. (1) Controls for simulated time. (2) User interface for going to specific time	78
Figure 43. Test Control Panel included in the execution for automation testing .	78
Figure 44. Define changes of smart devices at specific times	79
Figure 45. (1) Blocks for checking device state (2) Warning message generated from value checking test.....	80
Figure 46. <i>Morning Automations</i> triggered by environment events.....	82
Figure 47. Visual programs for <i>Morning Automations</i> scenario using <i>Blockly</i> blocks	85
Figure 48. (1) <i>Event History</i> including bubbles generated when the “Alarm Clock rings” automation is executed. (2) <i>Event History</i> including bubbles which are generated when the “water is ready for bath” automation is executed. (3) <i>Event History</i> including bubbles which are generated when the “window blinds open”	

automation is executed (4) *Event History* including bubbles which are generated when the “coffee is ready” automation is executed86

Figure 49. *Organizer* tool for the scheduled event and the *Event History* including bubbles which are generated when the "main door is locked for 5 minutes" automation is executed87

Figure 50. Home care automations triggered by calendar events88

Figure 51. Visual programs for *Self-Caring Home* scenario using Blockly blocks ..90

Figure 52. (1) Implementations for “*TurnOn*”, “*TurboMode*” and “*Service*” actions of Dehumidifier for simulated execution of automations. (2) Implementations for “*Program*”, “*Temperature*” and “*Start*” actions of Washing Machine for simulated execution of automations. (3) Implementations for “*Mopping*” and “*Sweep*” actions of Robot Vacuum Mop for simulated execution of automations. (4) Implementations for “*StartDefrost*” and “*ConfigureRapidMode*” actions of Refrigerator for simulated execution of automations.....90

Figure 53. (1) *Organizer* tool that includes daily events and the generated bubbles of the *Event History* for dehumidifier and smart robot tasks. (2) *Organizer* tool that includes daily event and the generated bubbles of the *Event History* for washing machine task. (3) *Organizer* tool that includes daily event and the generated bubbles of the *Event History* for refrigerator task.91

Figure 54. Fire protection automation triggered by environment event93

Figure 55. Visual program for *Fire Protection* scenario using Blockly blocks94

Figure 56. *Event History* of *Running Automations* of *Fire Protection*95

Figure 57. (1) Implementations for “*Open*” action of doors. (2) Implementations for “*TurnOff*” action of electric devices. (3) Implementation for “*Start*” action of the fire extinguisher.....95

Figure 58. Test for activating smoke sensor96

List of Tables

Table 1. Smart Devices for <i>Morning Automations</i>	83
Table 2. Smart Devices for <i>Self-Caring Home</i>	88
Table 3. Smart Devices for <i>Fire Protection</i>	94

1 Introduction

1.1 Smart Devices in the Internet of Things

The Internet of Things (IoT) is a domain that, after the Internet, represents the next most exciting technological innovation [1], [2], [3], [4]. IoT would open up a world of possibilities and influence in every corner of the globe. We can build smart cities using IoT, where parking, urban noise, traffic congestion, street lighting, drainage, and waste can all be tracked in real time and handled more efficiently. We can build healthy and energy-efficient smart homes. We can create smart environments that control air and water emissions automatically and allow for early detection of earthquakes, forest fires, and other catastrophic disasters.

Moreover, in the IoT, there is a wide variety of objects or "things," and some of these objects are referred to as "smart devices", "mobile devices", "smart things", or "smart objects" in the literature. From basic sensor nodes to home appliances and smartphones, smart devices are objects capable of communication and computation [6]. Smart devices are considered to be objects in the IoT.

Cisco projected in 2011 that by 2020, 50 billion Things will be connected to the Internet [5]. Another study, on the other hand, suggests that by 2020, 25 billion devices will be connected to the internet, with the goal of enabling the process of autonomous intelligent decision making. Regardless of which prediction is right, the key point is that the number of smart things would be many times greater than the current global population.

Additionally, devices in the Internet of Things should be able to rapidly adapt to evolving situations and take actions based on their operating conditions; and they should be self-configuring and interoperable, with unique identities and the ability to communicate and exchange data with other devices and systems [7]. As a consequence, smart devices should be context-aware and linked to the internet.

1.2 Automatic User Interface Generation

The design of user interfaces for different applications is becoming increasingly difficult. Users demand high-quality user interfaces and user-friendly complex applications. Consumers often expect the same applications to work on a variety of devices, including tablets, PDAs, notebooks, and other computers. It is incredibly difficult to develop an application interface that is scalable across different devices, resulting in the development of multiple user interfaces that are based on expected device capabilities and features. The design of such user interfaces is difficult, resulting in an increase in application development time. As a result, a concept for automatic user interface generation was developed.

By offering a collection of design rules and effectiveness requirements, automatic user interface generation systems promise to make an application programmer's design tasks simpler. To determine these parameters, you must first decide the properties of the data to be visualized are related to user interface design and how they are related. Data characterization is the term for this role. It is possible to build automated presentation systems using versatile data characterization. These, on the other hand, may not permit the development of rich user interfaces. A code characterization is needed to build a rich user interface with the ability to perform various operations on the characterized data.

The automatic user interface generation concept is relevant with the IoT, despite the fact that is often overlooked. Indeed, the new Internet of Things vision focuses primarily on the technical and infrastructure aspects, as well as the management and analysis of the massive amounts of data produced.

So far, only a small amount of research has been conducted on the front-end of user interfaces for IoT devices. However, as has been the case in other fields such as the Web, smartphone, and wearable technology, user interfaces in the IoT ecosystem will play an increasingly important role in end user adoption.

1.3 Visual Programming

Spreadsheets are the industry's most common end-user programming approach [8]. They favor both individuals and companies, and they are used in a variety of applications such as student grading, accounting, and hotel booking. Visual programming environments are also among the most common software tools for end-user development (EUD) [9], leading to the popularity of learning programming (e.g., Scratch [10], MakeCode [11], Tynker [12], Snap! [13]) and gaming (e.g., LEGO MINDSTORMS [14], LEGO in MakeCode [15], Tynker, LearnBlock [16]) for children.

Additionally, in visual programming, there are application domains that are not focused on learning programming. For inexperienced programmers, developing mobile apps is one such application area. The widespread use of smart phones in everyday life has resulted in an explosion of mobile apps. App Inventor [17] is a Google-provided web-based visual programming integrated development environment that allows novice programmers to build fully functional Android and iOS apps.

Furthermore, visual programming is also affected the IoT. Particularly, the use of connected smart devices and services, as well as automations that can be created, could benefit people's daily lives. In this context, there are a range of approaches that concentrate on smart-home automations, using commercially available smart devices and services. These approaches include HomeKit [27], Puzzle [28], Wia [29], Embrio [30], and SmartThings [31]. These apps provide a simple form-based architecture for creating simple automations among a collection of devices that support their standard.

1.3.1 Blockly Studio IDE

Visual programming languages are based on the production of graphical elements that correspond to high-level abstractions of source code expressions, removing the need for text coding. Visual programming languages, on the other hand, are insufficient for novices to build applications. They must be accompanied by

suitable development toolsets, such as text editors that are integrated into IDEs for software developers. On top of the Blockly library [49], it is built a full-featured IDE, the *Blockly Studio*, for visual programming languages in the context of end-user development.

The IDE's backbone is built on a component-based architecture that allows users to add and remove components through a centralized components registry. While the IDE is running, components can be enabled or deactivated on the fly. Each part is self-contained and interacts with the IDE through a specially developed extended Blackboard pattern.

Additionally, the IDE is application domain configurable. This means that the key components for end-user development could be modified based on the specifications of each application domain. Furthermore, the IDE contains an extension mechanism that allows developers to define and construct new application domain frameworks on top of it. These application domain frameworks are built right into the IDE and take advantage of all of its features.

1.4 Problem Definition

People's daily lives are able to benefit from smart devices based on the IoT concept. Particularly devices are able to provide an environment of automations that contribute to everyday activities. However, the needs for each person are different and fluid. As a result, everybody should be able to communicate with smart devices, potentially handling, parameterizing, and even programming applications involving them.

As mentioned in section 1.1, the Internet of Things is made up of a wide variety of connected devices. Different types of smart devices are connected through the network and are used to help people in their daily tasks. There is a need for end users to manage the state of their smart devices using the appropriate tools. Also, users want to be able to develop their personal IoT automations for their daily

activities based on their requirements without having any programming knowledge. In addition, the monitoring and interaction of devices during the execution of IoT automations is an area that suffers from a lack of solutions.

The purpose of this thesis is to provide a suite of tools for supporting the visual programming of Internet of Things. The tools that were implemented on the top of *Blockly Studio IDE* gives to the end users the opportunity to execute and test their defined automations as well as to visualize their smart devices through appropriate user interfaces.

In order to solve the visualization problem for smart devices, we have designed and implemented an automatic user interface generator. This tool generates appropriate interfaces for devices that use their data provided by the IoT middleware, IoTivity [41], in our case. Particularly, the device data is converted to the Generic device API that we have designed. In addition, the generator uses the data from the device API and creates the final interfaces in which there are MicroUIs for the device properties and actions.

Moreover, after the development of IoT automations, users want to execute them and have a clear picture of their smart devices that are included. In order to tackle this problem, we implement an execution window for automations. It includes a calendar tool in which there are all scheduled tasks included in automations. Furthermore, we add an event history tool that records every event that is triggered during the execution of automations. Additionally, the UI generator produces appropriate user interfaces which visualize the state of smart devices during the execution.

Finally, as we mentioned in the previous paragraphs, users want to be able to test their automations. In this context, we develop another execution window on the top of *Blockly Studio IDE* which is for the automations testing. Users who run their automations for testing can find and correct their errors and control the behavior of the virtual devices included in this particular execution.

1.5 Primary Contributions

Our main contribution is the creation of a set of tools that supports the visual programming for the IoT domain framework. An important component of this set is the execution of the IoT automations. In this context, we provide a complete set of tools for monitoring smart devices and the events triggered during the execution. In addition, the execution window includes user interfaces for smart devices for tracking changes. Also, we provide a different execution window for automation testing on the top of Blockly Studio. It executes the IoT automations including virtual devices that have same data as real devices. Furthermore, in this execution, the users can control time and date for testing their scheduled tasks. Finally, in this window there are two types of tests, the first to change the state of the devices at a specific time and the second to check their values.

For the visualization of the smart devices, we develop an external library called *Automatic UI Generator*. The first step in creating device interfaces is to design and define a generic device API that is used as input to the library. The device data is converted to the API and then the library receives the conversion data to create specific user interfaces. The user interfaces consist of *MicroUIs* for each property of the device and the actions are visualized with buttons. Finally, every user interface provided in the Blockly Studio for devices is the result of the generator.

1.6 Thesis Structure

The rest of this work is organized as follows; In Chapter 2, we review popular middlewares, tools for automatic UI generation and visual programming tools for IoT. Chapter 3 follows, which has the system overview. It begins with the architecture of our system and then describes the communication with the smart devices. Chapter 4 gives a description of the automatic UI generation tool. It begins with the description of the generic device API and then the micro-UIs that produces. Chapter 5 describes the contribution to the Blockly IDE for IoT. It begins with the user interfaces provided

by automatic UI generator for device management of IoT framework. Then, it describes the visual programming blocks that provided by IoT domain framework. Also, describes the execution of IoT automations. Finally, it gives a description of the environment for automation testing. Chapter 6 gives a description of the Case Studies; we have carried out in order to test our work. Chapter 7 concludes the work and identifies issues for further research work.

2 Related Work

2.1 Middlewares on IoT

Paraimpu

Paraimpu is an IoT middleware [18], [19] that allows users to register, manage, handle and interconnect their RESTful IoT devices or services whether physical or virtual. Things are mapped to either the abstract concept of sensors or actuators in *Paraimpu*. The former characterizes anything capable of producing data of a related type (text, numeric, JSON, XML etc.) and the latter characterizes anything that is able to perform actions by consuming data produced by the sensors. With the *Paraimpu* also users can connect their things. This allows users to compose simple IoT applications via JavaScript. All things in *Paraimpu* represented as RESTful resources and JSON is used for internal interchange of data between devices. The implementation of *Paraimpu* is succeeded using a scalable architecture leveraging a non-blocking Tornado Web server [20], a NGINX [21] load balancer, and a MongoDB [22] which provides persistency, replication and fail-over data management support. In other words, *Paraimpu* aims to provide a scalable cloud infrastructure.

Reusing and sharing the IoT resources in their social networks are the main advantages of *Paraimpu* over other IoT middleware. *Paraimpu* provides a limited set of configurable sensors, actuators and connections that can be reused across applications via filtering and mapping between inputs and outputs among sensors and actuators. *Paraimpu* does not support service discovery. *Paraimpu* does not provide device to device communication and thus entails the usual latency problem of a cloud-based architecture.

Google Fit

Google Fit [23] is a free and open IoT platform. It is a cloud-based IoT middleware that allows users to manage their fitness data and create fitness apps all from one location. A fitness store is included in the scheme, which is a cloud storage service that collects data from various devices and applications. A sensor framework is a collection of APIs that enable third-party IoT devices to link to its store. It offers APIs for subscribing to a specific fitness data form or source (e.g., Fitbit or Smartwatch), as well as APIs for querying historical data and continuous storage of sensor data from a specific source (e.g., a smartwatch). There's also a permission and user controls module that protects data privacy and security by requiring user consent before *Google Fit*'s apps can read or store collected data. *Google Fit* is an Internet of Things middleware designed to make it simple to build a specific form of application, in this case, self-tracking data from wearable fitness devices.

Google Fit has built-in support for IoT devices that use Bluetooth Low Energy (BLE) (Bluetooth Low Energy). A developer must include an implementation of the Fitness Sensor Service class as well as the supported data form if it is not accessible when adding a new fitness sensor type that does not communicate via BLE.

Calvin

Calvin [24] is an open source IoT middleware from Ericsson that aims to provide a single programming model for capability and energy limited IoT devices that is light-weight and portable. It is a hybrid paradigm for composing and handling IoT applications that combines principles from the actor-oriented model and flow-based computing. An actor, which is a reusable software component that can represent a computer, a computation, or a service, is the key abstraction for building IoT applications in *Calvin*. The input and output ports of an actor describe its interface. In contrast to the standard object-oriented model, which responds to method calls by returning values, an actor responds to inputs by generating outputs. The Asynchronous Atomic Callbacks (AAC) pattern is used in this actor model, where

short atomic actions are interleaved with atomic invocation of answer handlers for high-performing real-time interaction. *Calvin's* actor model often hides the low-level communication protocols of things, so actors link and interact via ports, regardless of how physical connectivity is accomplished. *Calvin* comes with its own scripting language to make it easier to program an actor. To enhance the process of creating an IoT application, it supports a prescriptive application development process called Describe, Connect, Deploy, and Manage. *Calvin* is a lightweight IoT middleware that can run on edge devices to reduce latency while still using the full computing power of the cloud when necessary.

Calvin's actor has the ability to switch from one runtime environment to the next, making it a reliable distributed IoT computation platform. The platform often includes a pre-defined set of actors who carry out common but distinct tasks. Actors for popular communication protocols and parallel processing are included. *Calvin's* developer will expand the capabilities of this middleware by using CalvinScript to create a new actor and adding it to the library. CalvinScript can be used to create actors in the game.

Node-RED

IBM's open source IoT middleware platform, *Node-RED* [25], is an open source IoT middleware platform. It is built on node.js, a server-side JavaScript platform that uses a distributed computing environment's event-driven, nonblocking I/O module. It is an IoT middleware that, like *Calvin*, can be run at the network's edge due to its small footprint. The most important abstraction is *Node*, which is a visual representation of a block of JavaScript code that performs a specific function on an IoT computer (e.g., reading a particular value). To put it another way, each node can be thought of as an actor.

The main benefit of *Node-RED* is a visual tool that makes composing IoT devices easier, particularly if the node for the IoT device has already been created and published by others. Users may use Node-visual RED's tool to drag-and-drop blocks

that represent components of a larger system and link them to create an IoT application. As a result, *Node-RED* facilitates the development of IoT applications. The composition engine binds IoT devices that can be abstracted as nodes together.

The APIs for communicating with the system must be available as a node.js library or a module accessible by *Node-RED* for a device or service to operate with *Node-RED*. Password authentication provides a minimal level of protection. The *Node-RED* team believes that by forming a social network of *Node-RED* developers, modules or node.js libraries for heterogeneous IoT devices can be crowdsourced. Service discovery is not available in *Node-RED*. It is made with Node.js [26], a modern framework with few libraries and modules.

2.2 Automatic UI Generators

Some research works of automatic UI generation have been found for appliances, but there is not any relevant work that emphasizes on UI generation for IoT devices. However, some previous works are very useful to our research.

Supple

In [34] Gajos et al. presented a toolkit named *Supple* which can generate UIs for ubiquitous applications. The *Supple* can generate a concrete UI for the target device after the designers specify declarative UI models and target device. Beside the generation of UIs can be customized and its distributed architecture enables devices to show *Supple* UI with less overhead.

Dynamo-AID

In [35] based on the traditional models like task model, environmental model and dialog model, Clerckx et al. extend them to provide a design process and runtime

architecture, *DynaMo-AID*, that enables designers to develop context-sensitive user interfaces which can change during the runtime of the interactive application.

Pebbles

The most relevant work was done by Nichols et al. in *Pebbles* project which aims to generate the high-quality UIs on a hand-held device working as the personal universal controller (PUC) for various appliances [37]. They extended the PUC with a layer named *Uniform* to provide the UIs which are consistent with past used UIs [38]. Moreover, the *Huddle* system uses a model of content flow to generate UIs for controlling connected appliances at high-level and low-level [39]. The simplification of UIs can increase the usability of appliances with complex functionalities.

RBUIS

Akiki et al. present a tool supported approach, *Role-Based UI Simplification (RBUIS)*, that simplifies enterprise application UIs by providing users with a minimal feature-set and an optimal layout based on the context-of-use [40].

Other Approach

In [36] Roscher et al. identify the concept of ubiquitous user interfaces (UIs) including five properties, shapeability, distribution, multimodality, shareability and mergability. Then they proposed an approach of combining UI runtime architecture MASP and runtime UI models to adapt UIs based on automatic adaptation algorithms.

2.3 Visual Programming for IoT

HomeKit

HomeKit [27] is a product from Apple allowing control of connected home accessories when compatible with *HomeKit*, and supports to a certain degree user-defined automation as combinations of accessory control actions. It is not a EUP system as such, and focuses mostly on smart home solutions with emphasis on advanced configurations.

Puzzle

Puzzle [28] is a visual development system for custom automations with smart objects in IoT adopting the jigsaw metaphor. However, the visual system is primitive and lacks the full-scale capacity of common VPLs like all algorithmic elements, procedures and objects, as well as versioning and application management.

Wia

Wia [29] is a cloud platform that makes creating IoT apps easier by linking IoT devices and external services. It is possible to attach IoT development boards, IoT devices, sensors, and external services using Flow Studio. It differs from others in that it employs complex blocks that execute complex operations such as sensor management. It fits with Arduino MKR1000, MKR1200, Espressif, Raspberry Pi, Particle, and other IoT creation boards. It also integrates with third-party applications such as AWS, Twitter, and Twilio. We can use *Wia's* API to communicate with it and exchange data.

Embrio

Embrio [30] is another interesting visual tool to develop IoT apps. It is built for Arduino and works with a range of operating systems, including Windows, OS X, and Linux. *Embrio* is a visual programming interface for Arduino that uses a drag-and-drop approach. It is based on the Agent principle. An Agent is essentially a process with a task to complete. Agents can run concurrently and can trigger or kill other Agents. The data flow and logic of an IoT app are described by the relations between Agents. The *Embrio* app can be translated to Arduino code and run on the platform.

XOD

XOD [32] is a microcontroller programming platform with a visual interface. It is based on the Node model, which can represent a sensor, motors, or a piece of functional code like comparison operations, text operations, and so on. Each node has an input and an output, allowing us to define the IoT app logic by connecting all of the nodes. *XOD* produces native code that can be uploaded to and run on Arduino compatible boards. It primarily supports Arduino. It is an open-source project with an interesting feature: it is extensible, meaning new nodes can be introduced to support new components.

Zenodys

Zenodys [33] makes it easy for developers to create IoT apps. It is possible to collect data from any sensor and easily visualize the values acquired using the *Zenodys* platform without programming. Using Workflow builder makes it possible to build complex backend solutions using visual programming tools. Finally, the UI builder aids the developer in the development of an IoT dashboard for the visualization of data and details. It is a robust platform that offers a range of services that can be linked together with the aid of its software and builders. *Zenodys* can be used in a

range of scenarios, including predictive maintenance, real-time control systems, product line automation, and so on.

3 System Overview

In this chapter we are going to describe the overview of the visual programming support system that is developed for the IoT domain framework at the top of *Blockly Studio*. First, we describe the macro-architecture of the system and the elements included in it. In addition, we describe the communication between our system and smart devices and the need to simulate them.

3.1 Architecture

Figure 1 shows a macro-architecture of components that supports the visual programming for smart devices at the top of *Blockly Studio*. At the bottom of the stack are the smart devices that export their functions. To enable communication between smart devices and other components, we use IoT middleware.

The next component is the *Automatic UI Generator* (section 4). To successfully create user interfaces for smart devices, we first implement a Converter library that converts the device API to a specific API that is called Generic Device API. It includes only the data needed for the visual ion and interaction with smart devices. Then, based on the API, Automatic UI Generator generates more than one user interface type for smart devices from device-management process to the execution of automations.

Moreover, we implement an execution environment for the IoT automations on the top of *Blockly Studio*. In this context, we provide a visual programming toolset for monitoring and interaction with devices. Particularly, in the environment exists a calendar tool for monitoring scheduled tasks and an event history panel that records device actions and conditional-based event. Furthermore, a device dashboard is provided that displays in real-time an updated view of all smart devices involved in running application

Finally, an extra environment is introduced for testing the crafted automations. In this context, we extend the toolset that exists in the aforementioned execution with test and simulation tools. Specifically, we provide a device simulator that emulates all properties and actions of actual smart devices by displaying them with virtual UI implementation. In addition, we create a time simulation enabling to control directly the flow of time, with five basic operations supported, and thus trigger directly all related scheduled events, by communicating internally to the basic calendar component. Lastly, a suite of tests is available for simulating the behavior of smart devices in a specific time period and another type of test enabling the users to check the current state of devices.

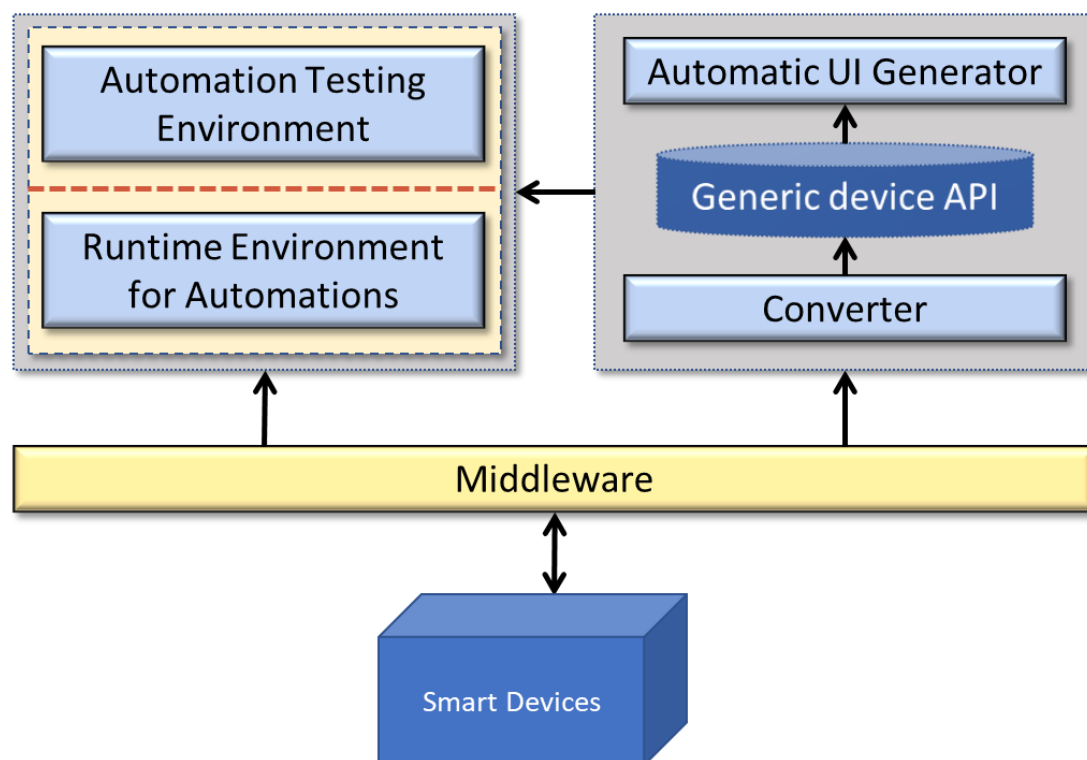


Figure 1. The macro-architecture of the system for visual programming support for smart devices

3.2 Communication with Devices

To make the communication with smart devices possible, we use the IoTivity middleware [41]. It is an open-source software framework, reference implementation of the Open Connectivity Foundation (OCF) standards for the IoT.

Furthermore, IoTivity provides the `iotivity-node` [42] a JavaScript API for OCF functionality and it is implemented as a native addon using IoTivity as its backend. Our work uses both IoTivity and `iotivity-node` to communicate with smart devices, and carries out all the required functionality which is described in the following paragraphs.

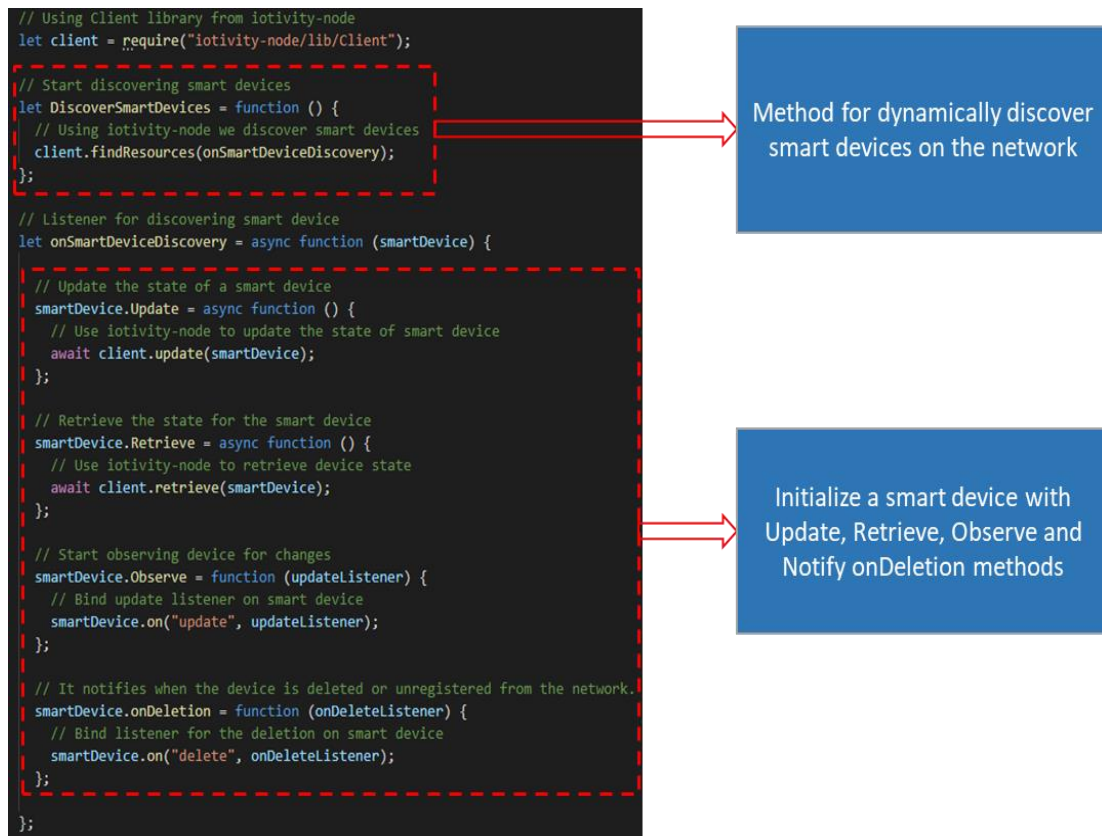


Figure 2. API implementation for communicating with smart devices

Using `iotivity-node`, we have managed to create our *communication API*. It consists of five main methods. Firstly, a method for dynamically discovering smart devices which are connected to the network is implemented. Also, we have implemented methods for updating and retrieving the state of the devices. Furthermore, we create two event-based methods, the first one uses the *update* event of `iotivity-node` to observe any change on the state of a smart device and the second one uses *delete* event to notify when the device is deleted or unregistered from the network. Last but not least, the *communication API* implements two more methods for each property of a smart device, first one is created to set a new value to the property and the second exists for getting the current value of the property. In

their inner body, they are used the Update and Retrieve methods of the API respectively. The API that we have implemented is presented in Figure 2 and Figure 3.

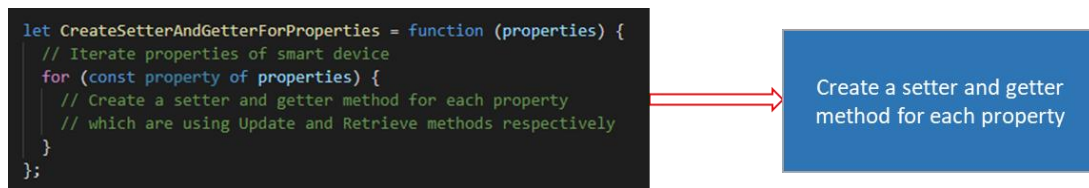


Figure 3. Creation of setter and getter methods



Figure 4. practically-RESTful API for Air Conditioning smart device

3.2.1 Simulating Smart Devices

For the need to test and evaluate our work, we need to get a wide variety of smart devices. The only way to achieve the large number of different smart devices

is to simulate as much as you can. The simulation of the smart device is intended to have the same data as the real ones and the same functionality.

For the simulation of smart devices, we use the IoTivity Simulator [43]. It is a plugin tool over the Eclipse IDE [44]. Using this tool, we can simulate smart devices as OIC (Open Interconnect Consortium) resources. Open Interconnect Consortium (OIC) [45] is a standard and open-source project that delivers “just-works” interconnectivity for developers, manufacturers and end users. The IoTivity Simulator comes with a Service Provider that manages creation, deletion, request handling and notifications of simulated resources. Furthermore, it handles the requests received and sending appropriate responses to clients.

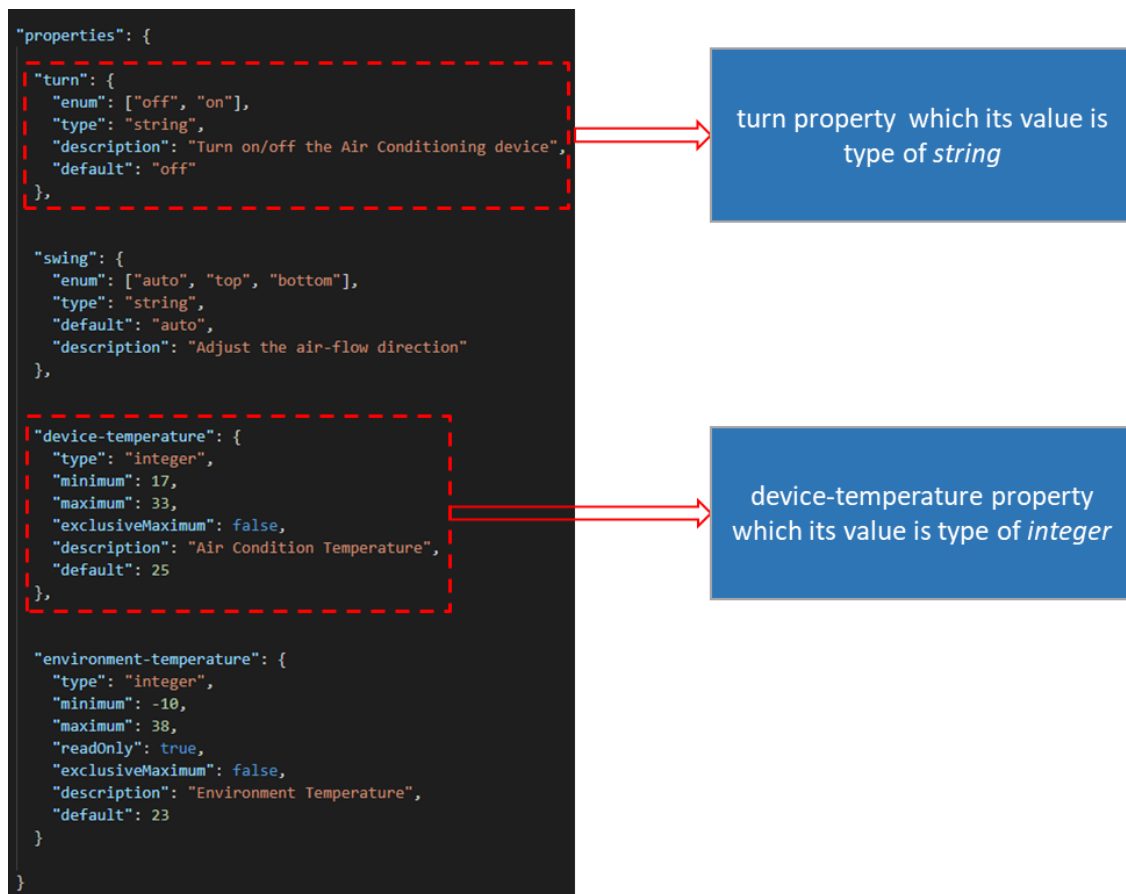


Figure 5. Properties of the Air Conditioning smart device

To successfully simulate a smart device through Simulator, we have to build its REST (Representational State Transfer) API with the help of the RAML (RESTful API Modeling Language). It is a way of describing practically-RESTful APIs in a way that’s

highly readable by both humans and computers. A REST API (also known as RESTful API) [46] is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. Figure 4 shows the practically-RESTful API of OIC resource that we import to Service Provider for simulating a smart Air Conditioning device. We have modeled the GET request for retrieving current state of the device. Also, it has been modeled the POST request for updating the Air Condition with the updated state.

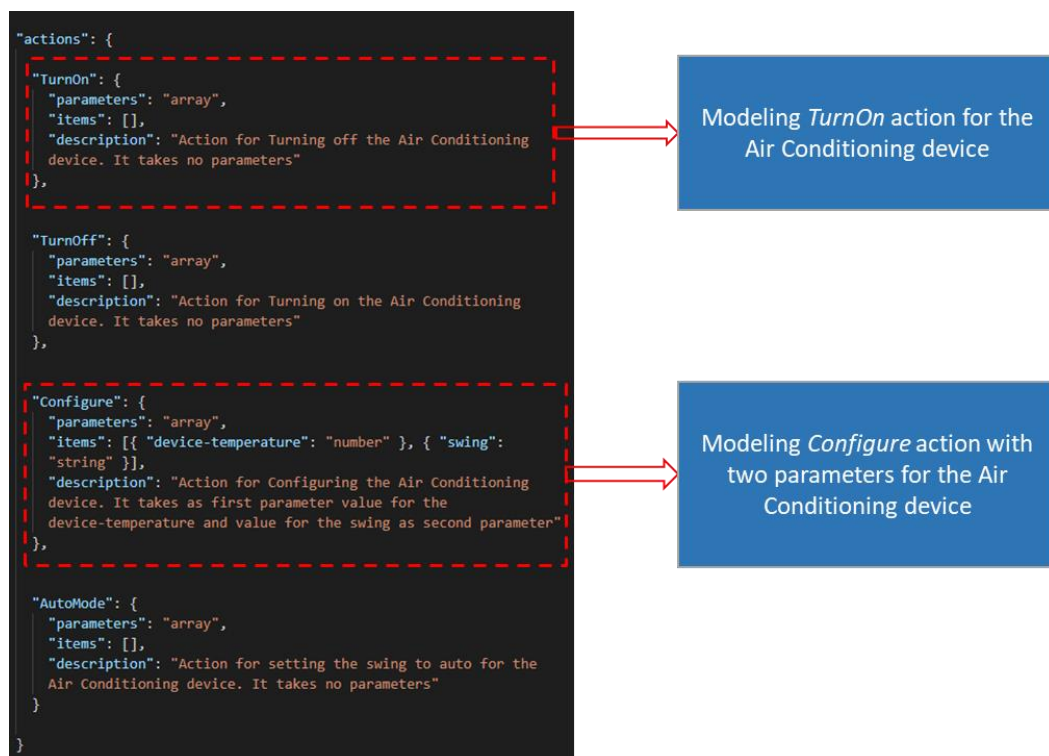


Figure 6. Actions of the Air Conditioning smart device

In order to create functionality for the smart devices we use JSON Schema [47] and more specific the draft 2017-07. It is a vocabulary that describes an existing data format. It also provides clear human- and machine- readable documentation. Every smart device consists of *properties* and *actions*. The first category includes all that items of the device: that they can take a single value. The value types of properties are the following:

- String: This type is used for strings of text and it may contain Unicode characters
- Boolean: This type matches only two special values *true* and *false*.

- Numeric: There are two numeric types *integer* and *number*. The first is used for integral numbers and the latter is used for any numeric type, either integers or floating-point numbers.

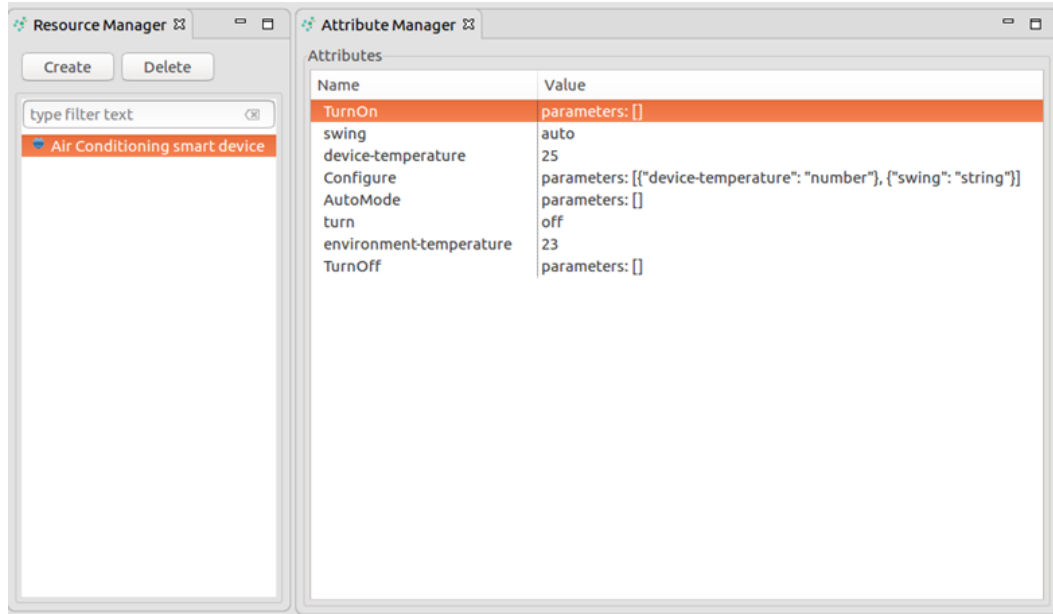


Figure 7. IoTivity Simulator included Air Conditioning smart device

For the properties which take string as value, it can be defined the possible values by the keyword *enum* as we can see for the first two properties turn and swing of Air Conditioning device in Figure 5. Also, we can make a *range* numeric type using *minimum* and *maximum* keywords such as the device and environment temperature properties of the Air Conditioning device. Furthermore, we can define a property as *read-only* which means that user cannot change its value, it can only be changed from the Service Provider. Lastly, using *default* keyword, we can initialize the value of property.

For the purpose of simulation of a smart device, except properties, we define and its actions. They are all these operations that a smart device can perform (e.g., Turn on a Television). We use JSON Schema for modeling actions for a smart device, each action consists of parameters and a function body. The first is an array from items that they have name and type, and the second is added in the communication phase as JSON schema doesn't not support function type. In Figure 6 we show the Air Conditioning device actions. We can see that the action Configure that it has two

items as parameters for setting device-temperature and swing properties respectively. After completing the modeling of a smart device, we import it to the Service Provider to finish the simulation. In Figure 7 we can see the Air Conditioning device from the view of the Simulator tool.

4 User Interface Generator

In this chapter we are going to describe the Automatic User Interface Generator that produces UIs for the smart devices. To generate user interfaces, we have designed and built a generic device API used from the *Automatic UI Generator*. Furthermore, we are going to present the MicroUis for each property type generated automatically. Finally, we describe the user interfaces for actions and methods of device and how they are executed with or without parameters.

4.1 Generic Device API

With the aim of generating User Interfaces for smart devices, we define a Generic Device API. For converting smart device data to the API, we implement a new library which is called *Converter* as we can see in Figure 8. The main function of the *Converter* is the *ConvertDevices* which is called when we want to convert devices data to the Generic API (Figure 9)

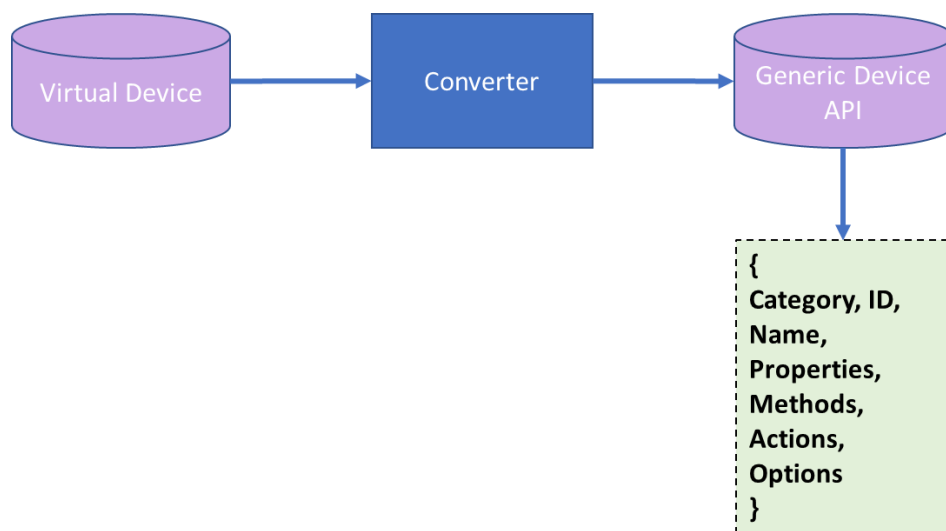


Figure 8. Converting virtual device data to Generic Device API

```
export let ConvertDevices = function (devices) {  
  // Iterate all devices  
  for (const device of devices) {  
    // Convert virtual device data to Generic Device API  
    // for User Interface generation  
  }  
};
```

Figure 9. ConvertDevices of Converter library

In order to validate that the data of each device are successfully converted to the Generic Device API, we use JSON Schema. In Figure 10 we present the schema for the API and according to this a device has the following attributes:

1. *category*

The category attribute declares that the object is a smart device and its default value is "Device"

2. *id*

The id attribute is an identifier for each method and action. Its value is unique for every device

3. *name*

The name attribute contains name of the device that should be presented to user in user interface.

4. *option*

An option attribute for the devices which contain an *image*

A device also contains three more attributes: *properties*, *methods* and *actions*.

In reference to *properties*, the data description is presented in Figure 11 through a JSON Schema and they carry five basic attributes:

```

{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Device",
  "description": "This schema describes a smart device",
  "type": "object",
  "required": ["id", "name", "properties", "methods", "actions"],
  "properties": {
    "category": {
      "default": "Smart Device"
    },
    "id": {
      "type": "string",
      "description": "A unique id for each smart device"
    },
    "name": {
      "type": "string",
      "description": "Display name for the device"
    },
    "properties": {
      "type": "array",
      "description": "Properties of smart device",
      "items": {
        "$ref": "#/definitions/property"
      },
      "uniqueItems": true
    },
    "actions": {
      "type": "array",
      "description": "Actions of smart device",
      "items": {
        "$ref": "#/definitions/method"
      },
      "uniqueItems": true
    },
    "methods": {
      "type": "array",
      "description": "Methods of smart device",
      "items": {
        "$ref": "#/definitions/method"
      },
      "uniqueItems": true
    },
    "options": {
      "type": "object",
      "properties": {
        "image": {
          "type": "string",
          "description": "Path of image for a smart device"
        }
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false,
  "definitions": {...
}

```

Figure 10. Generic Device API definition

1. *category*

An identifier attribute for properties which has “Property” as default value.

2. *name*

The name attribute contains name of the property that should be presented to user in user interface. It is unique for each property.

3. *value*

It is the value of the property and It can be one of the following:

- Number
- Boolean
- String

4. *type*

The type attribute contains type of the property. Converter gives a type to property of a smart device based on its value type. There are five different types:

- *number*: Property value is *number* without minimum or maximum value.
- *intRange*: Property value is *integer* with minimum and/or maximum value.
- *boolean*: Property value is *boolean* and the possible values are *true* or *false*
- *string*: Property value is a *string of text* and it does not have possible values.
- *enumerated*: Property value is as sting of text such as *string*, but its value is selected by a set of strings.

5. *read_only*

The *read_only* attribute is for the property which its value cannot be modified but only be accessed. For the read-only properties a specific Micro UI is presented to the user.

6. *option*

An *option* attribute is for the properties which their type is either *enumerated* or *intRange*. It includes the followings attributes:

- *possible_values*: An array that contains all possible values for an *enumerated* type property.
- *minimum_value*: The minimum value for an *intRange* type property.
- *maximum_value*: The maximum value for an *intRange* type property.

```

"property": {
  "title": "Property",
  "description": "This schema describes a property of IoT device",
  "type": "object",
  "required": ["name", "value", "type", "read_only"],
  "properties": {
    "category": {
      "type": "string",
      "default": "property"
    },
    "name": {
      "type": "string",
      "description": "Display name for the property"
    },
    "value": {
      "anyOf": [
        { "type": "number" },
        { "type": "boolean" },
        { "type": "string" }
      ],
      "description": "The value of the property. It can be scalar: single value"
    },
    "type": {
      "type": "string",
      "enum": ["number", "intRange", "boolean", "string", "enumerated"],
      "description": "The type of the value of property"
    },
    "read_only": {
      "type": "boolean",
      "description": "If the property is read-only"
    },
    "options": {
      "type": "object",
      "properties": {
        "possible_values": {
          "type": "array",
          "items": {
            "anyOf": [{ "type": "string" }]
          },
          "description": "Possible values for the enumerated properties"
        },
        "minimum_value": {
          "type": "number",
          "description": "Minimum value for property that it has number value"
        },
        "maximum_value": {
          "type": "number",
          "description": "Maximum value for property that it has number value"
        }
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false
}

```

Figure 11. Property of a smart device definition

In respect of *actions* and *methods*, we define the same JSON Schema which is presented in Figure 12 and it consists of five attributes.

1. *category*

An identifier attribute for properties which has “Action” and “Method” as default value for the action and method respectively.

2. *Id*

The *id* attribute is an identifier for each method and action. Its value is unique for every method or action

```
"method": {
  "title": "Method",
  "description": "This schema describes a method of IoT device",
  "type": "object",
  "required": ["id", "name"],
  "properties": {
    "category": {
      "type": "string",
      "default": "method"
    },
    "id": {
      "type": "string",
      "description": "A unique id for each method"
    },
    "name": {
      "type": "string",
      "description": "Display name for the method"
    },
    "parameters": { ...
  },
  "_UI": {
    "type": "object",
    "properties": {
      "description": {
        "type": "string",
        "description": "The description of method that gets displayed to user"
      },
      "colour": {
        "type": "string",
        "description": "rendering colour",
        "enum": [
          "blue",
          "grey",
          "green",
          "red",
          "yellow",
          "light",
          "dark"
        ]
      },
      "display": {
        "type": "string",
        "description": "In which area to display the method."
      },
      "dependence": {
        "type": "array",
        "description": "",
        "items": {
          "type": "string",
          "description": "Check dependence to allow execution of the method"
        }
      }
    },
    "additionalProperties": false
  },
  "additionalProperties": false
},
"additionalProperties": false
},
```

Figure 12. Method of smart device definition

3. *name*

The name attribute contains name of the method or action that should be presented to user in user interface.

```
"parameters": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "description": "Describes name of the parameter"
      },
      "type": {
        "type": {
          "anyOf": [
            { "type": "string" },
            { "type": "number" },
            { "type": "boolean" }
          ],
          "description": "Type of parameter"
        }
      },
      "_UI": {
        "type": "object",
        "properties": {
          "description": {
            "type": "string",
            "description": "Description acts as a tip for the parameter."
          },
          "relation": {
            "type": "string",
            "description": "The name of property to take its value as parameter"
          }
        }
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false
}
```

Figure 13. Parameter definition

4. *parameters*

The parameters attribute describes the parameters of the method or action. The JSON Schema for the parameters is presented in Figure 13 and it consists of three attributes:

- *name*

The name attribute contains name of the parameter.

- *type*

The type attribute contains type of the parameter and can be one of the followings:

- i. string

ii. number

iii. boolean

- *_UI*

The *_UI* attribute includes all the information about the User Interface. It consists of the followings:

i. *description*: The description attribute describes parameter.

ii. *relation*: The relation attribute is important for the parameters because it describes relation between parameter and a property of smart device

5. *_UI*

The *_UI* attribute includes all the information about the User Interface. It consists of the followings:

- *description*: Description describes method or action and acts like a tip for the user.
- *color*: It corresponds to the color that the method or action will be colored.
- *display*: It describes the display area that the method will be displayed. It takes as value a property name for displaying in property area, or “generic” to be displayed in generic area of smart device.
- *dependence*: The dependence attribute expresses conditions which have to be satisfied to allow execution of selected method or action. Dependence is very important because without proper dependence checking, user can have access to unavailable methods.

4.2 MicroUis

4.2.1 Device Properties

The first issue that arises, which is also the most apparent and easily confused as the only issue in automatic interface generation, is that of displaying values of data.

The approach that is taken in our work for displaying the values of properties of a device is inspired from the Properties and MicroUis architecture which is introduced by [48].

As we mentioned in 4.1 (Generic Device API) during the conversion of smart device to Device API, a property is taken type based on its value type. For instance, the type of environment-temperature property of Air Conditioning device should be *intRange* because its value type is integer with minimum and maximum value.

The responsibility for displaying properties is then passed to hard-coded, embeddable micro-interfaces. The matching of properties to MicroUis is done simply by type matching: properties of a certain type can only be used by certain MicroUi-rendering methods (Figure 14).

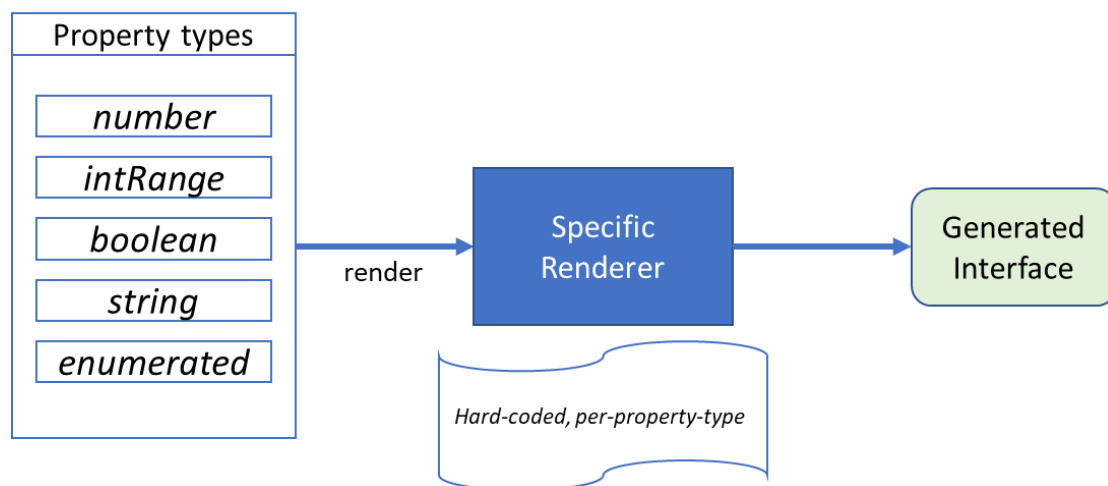
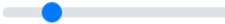






Figure 14. Design of device properties and their automatic rendering with MicroUis

In the following list is presented the MicroUis which are generated based on the property types. However, a read-only property is mapped to a unique MicroUi regardless of the type:

1. *number* → *Edit Box*

2. <i>intRange</i>	→	<p><i>Range Slider</i></p> <p>value:20</p> 
3. <i>boolean</i>	→	<p><i>Switch Button</i></p> 
4. <i>string</i>	→	<p><i>Edit Box</i></p> 
5. <i>enumerated</i>	→	<p><i>Select Box</i></p> 
6. <i>read-only</i>	→	<p><i>Read-only Box</i></p> 

4.2.2 Methods and Actions

Another issue that arises during the automatic generation is that of displaying a method or an action of a device.

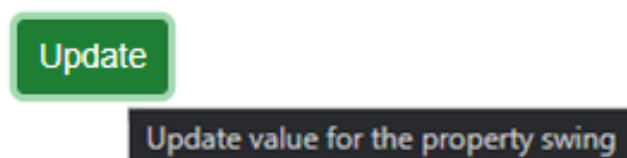


Figure 15. Update method with its description

The approach that we decide to take here is to render every action or method as buttons. Also, to help users to understand which is the use of every method, we create tooltips which include their descriptions (Figure 15)

To successfully complete the interface generation of actions and methods, we have to generate User Interfaces for their parameters. We have managed to re-use some of the MicroUis which are mentioned in 4.2 (*MicroUis*). So, a parameter of number type is mapped to MicroUi for number type, a boolean type parameter is mapped to MicroUi for boolean and so forth.

5 Integration with Blockly Studio IDE

In this chapter we discuss the components for supporting the visual programming for Internet of Things on the top of *Blockly Studio IDE*.

In detail, we present the user interfaces produced by *Automatic UI Generator (4)* for both single devices and device groups. Moreover, the visual programming blocks and elements provided by IoT domain framework of Blockly Studio for the development of automations are described. In addition, the runtime environment for automations and the environment for automation testing are presented.

5.1 User Interfaces on device management

Blockly Studio provides a device management process for defining and managing smart devices for end-user development. For this process we provide user interfaces for smart devices and device groups by *Automatic UI Generator (4)*.

5.1.1 Single Device

Through the *communication API* described in section 3.2, users scan the network for available smart devices. Then, a list of smart devices is provided including information for their identity and their properties as depicted in Figure 16. The visualization of smart device is based on the automatic user-interface generation process (4) which gets JSON data response from scan's request to the *IoTivity* as input. The end-user developers are able to choose which of the smart devices from the list will be registered for the development process by clicking the "*Register*" button.

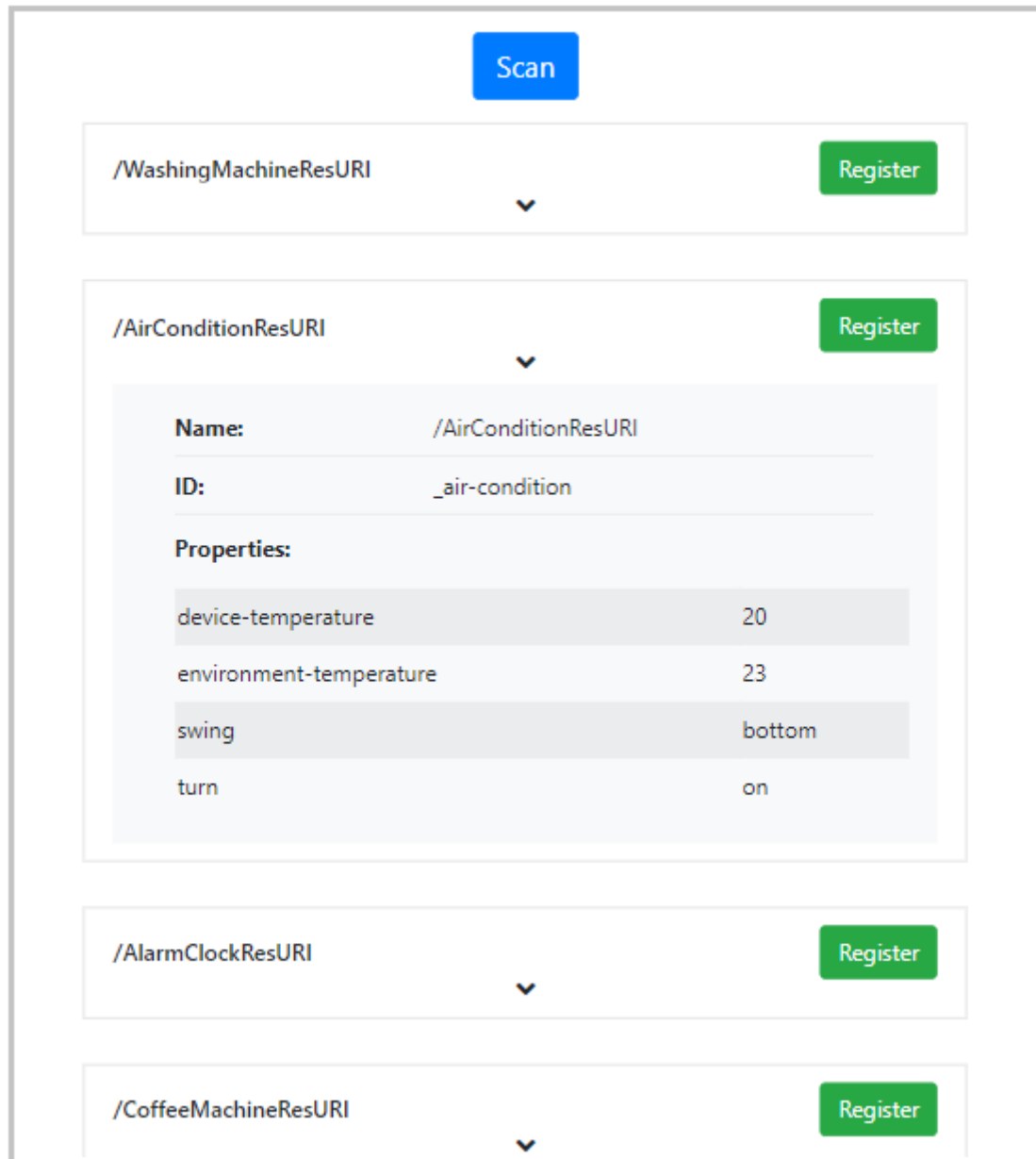


Figure 16. Scanning devices on the network

Registered smart devices are then available to operate during the development process. The generator generates user interfaces for operating devices based on their functionality (see Figure 17). First, it provides a read-only MicroUI for each device property that includes the name, universal-id, value of property and a button that enables the property for the development process. In addition, the UI provides a visualization for each action that includes its name and a button to activate it in the development process. In addition, in the action MicroUI there is a button to implement the action body that is ran in the execution for automation testing.

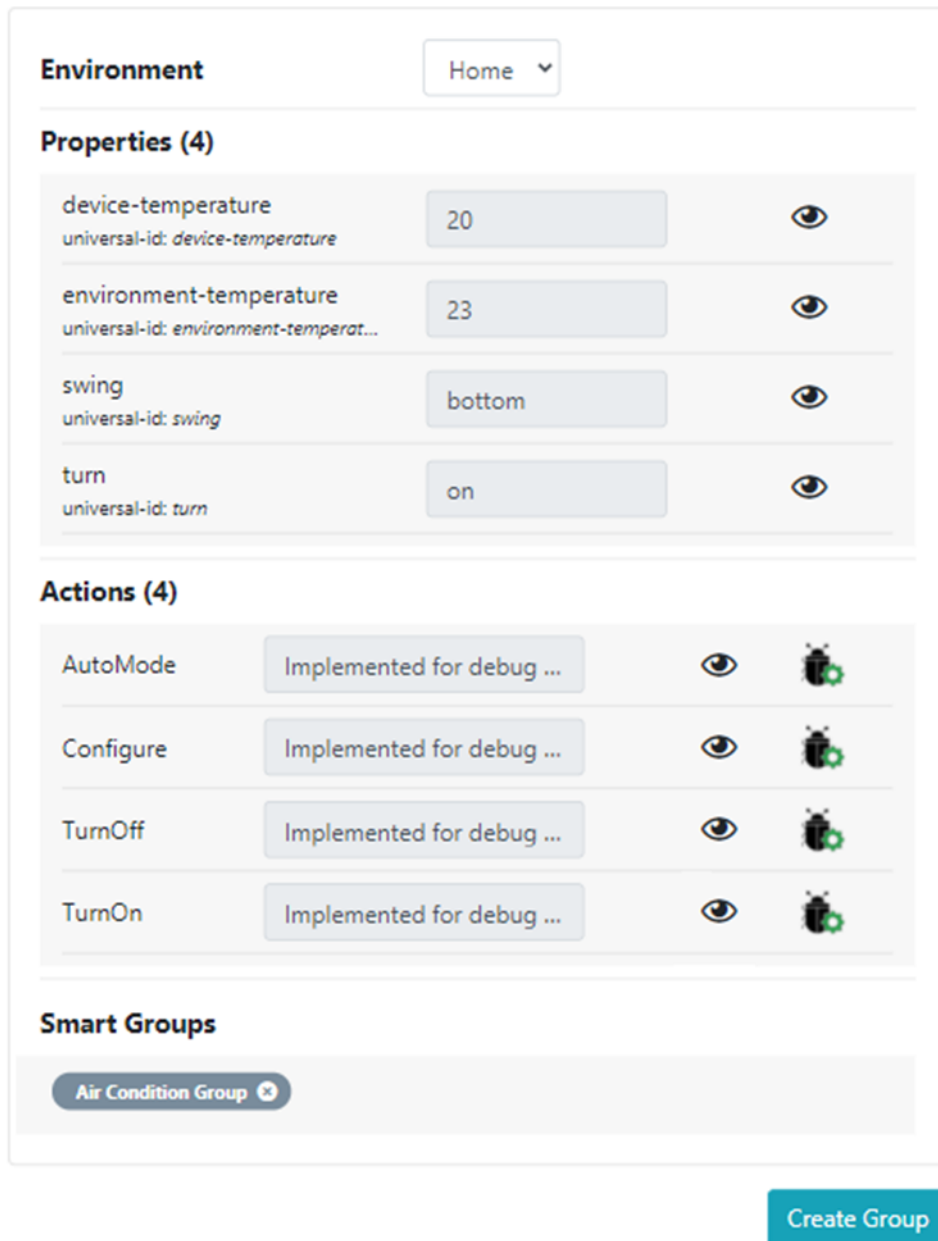


Figure 17. Air Conditioning device interface which is generated by *Automatic UI Generator*

At the bottom of the device user interface is the smart group area that includes all the device groups in which the device participates. The interface of each smart group includes each name and a button for removing the device from this group.

5.1.2 Device Groups

In addition, device management of IoT domain framework attempts to identify which of the registered devices of the smart automation have common functionality

and organize them in groups (e.g., more than one air-conditioning and smart lamps could be registered in a smart group). These groups give the ability to develop-handle the smart devices in groups instead of requiring to handle each one of the common devices (e.g., turn on/off all air conditioning devices in the house).

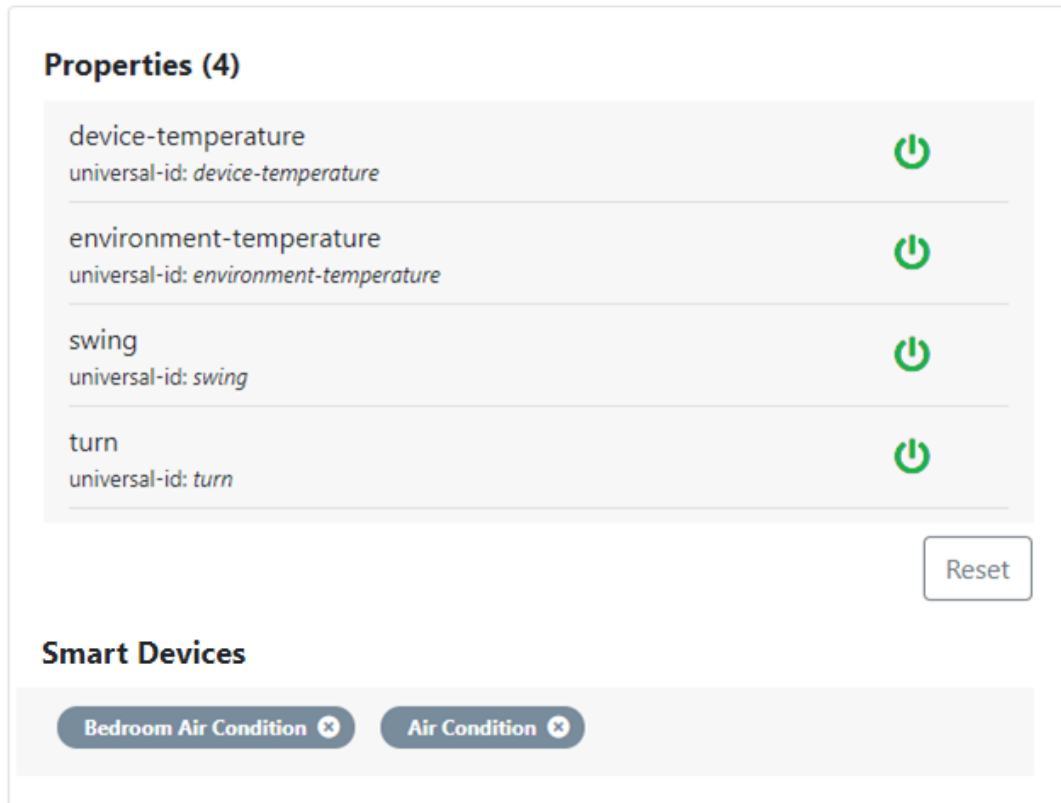


Figure 18. Air Conditioning device group interface which is generated by Automatic UI Generator

The users are able to create new groups with common functionality via the devices by exporting their properties (i.e., click the “Create Group” button presented in Figure 17). The user interface of groups (see Figure 18) includes the common functionality of devices (i.e., same device properties). Particularly, the automatic UI generator provides a read-only MicroUI for each group property. In addition, for each property is provided a button for enabling it in the development process. This is useful in case they would not like to include a specific common functionality in the group and this functionality is not supported by one device that they would like to be included in the group. At the bottom of user interface there is the “Smart Devices” area that includes all devices that belong to this group. In detail, for each device is

visualized its name and a button to remove it from the list in case of the users would like to handle it separately.

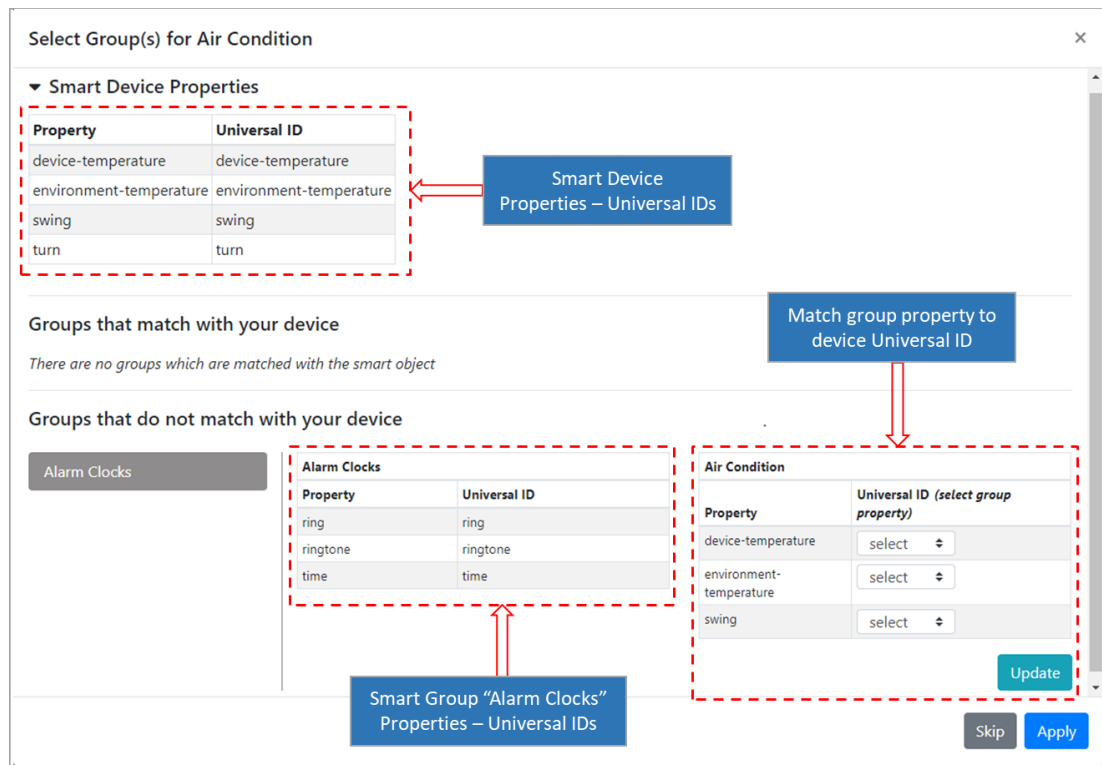


Figure 19. Adding Air Condition device to an already defined group

Moreover, in the process of the matching common functionality of smart devices, the end-user developer is able to give for each one of the properties a *universal-id*. This is useful in the case that devices support common functionality but export different APIs. The matching mechanism attempts to match the original property name and then in the case of failure tries to match with the given universal-id. The matching mechanism to add a smart device in at least one existing group is presented in Figure 19. The UI generator mentioned in section 4 provides interface for the device and group on the top of matching mechanism. For matching a device with a group, a user has to update properties' *universal-IDs* to match with group properties' either name or *universal-IDs*. For instance, in Figure 20 we update universal-ids for an air conditioning device to match with an alarm clock group. Finally, when the device is matching with the group the end-user selects at least one group to add the device.

Air Condition	
Property	Universal ID (select group property)
device-temperature	ring ⇅
environment-temperature	ringtone ⇅
swing	time ⇅

Groups that match with your device Select all

Alarm Clocks

Figure 20. Update *universal-IDs* of the smart device and match it with an existing group

5.2 Visual blocks provided by the IoT domain framework

The visual programming blocks are provided by the Blockly Studio and they have been designed using the *Blockly Developer Tools* [50]. It is a web-based developer tool that automates parts of the *Blockly* library configuration process, including creating custom blocks, building your toolbox, and configuring your web Blockly workspace.

In the following paragraphs we describe the blocks provided by Blockly Studio for end-user development.

5.2.1 Device

The set of Blockly blocks for devices consists of three categories. In the first category of blocks belongs the actions of a smart device. As we have mentioned in 3.2.1 (*Simulating Smart Devices*), in the device functionality belongs and its actions.

So, we have implemented constructors which dynamically generate blocks for each action based on the type and number of their parameters. To make the dynamically generation process of blocks clearer, we present the actions blocks of two smart devices in Figure 21.

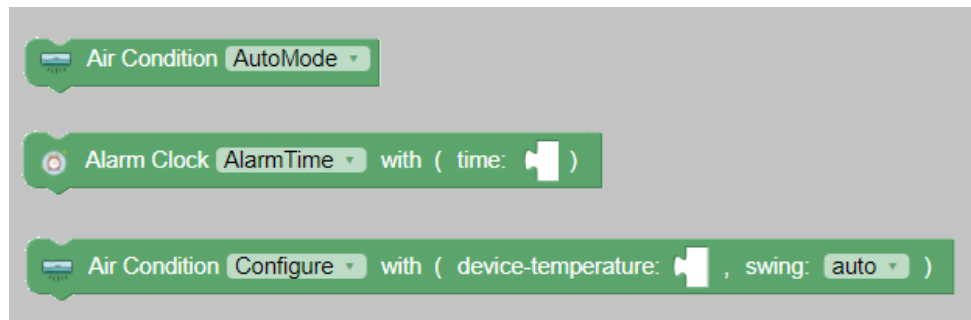


Figure 21. Visual programming blocks for device actions

The second category consists of blocks for setting and getting value for each property. As we have described in previous paragraphs there are five types of properties. Depending on the type of property, there are different blocks. So, the block which sets a value to enumerated property, it takes as value a string block, a block which sets a value to a number property type, it takes as value a number block and so forth. In Figure 22 we present setter and getter blocks for different types of properties.

Lastly, the third category consists of blocks that can take input or print the values of properties using the console tool of *Blockly Studio IDE*. These blocks make more powerful the development process for the end-user as he can change the state of a smart device during the execution of an application. The first block in Figure 23 is for giving input for the *device-temperature* property and the second block is for printing the value in the console.

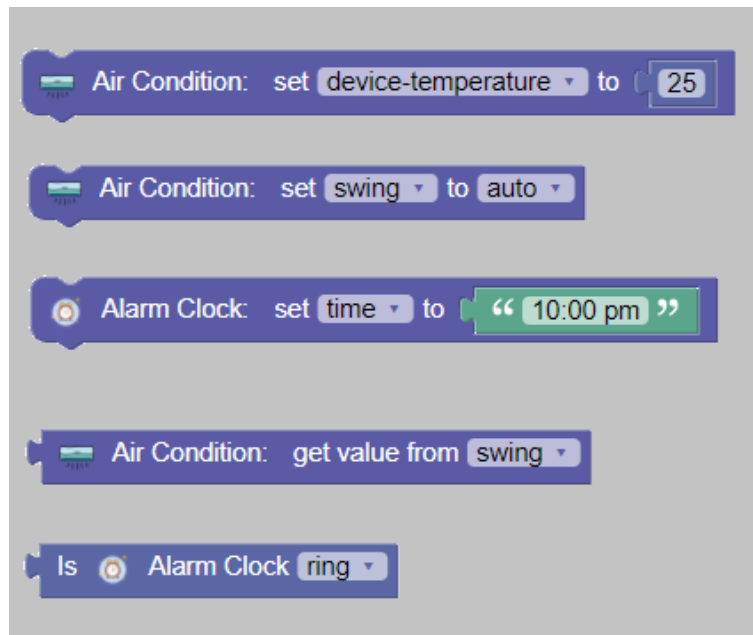


Figure 22. Setter and Getter Blockly Blocks for Properties

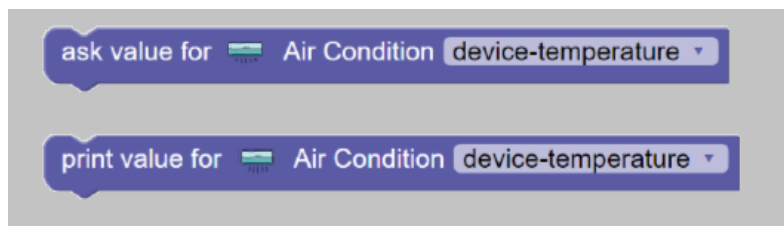


Figure 23. Input/Output for smart device properties in the I/O Console

5.2.2 Device Group

In the case of smart device group, the IDE provides constructors that create only these blocks which are important to help the end-user in the handling of smart devices. However, some blocks that it has been created in smart devices does not have any worth in device group, these are both getter and action blocks. In addition, since there is not getter block for group, there is no reason to exist an output block. However, the setter and input (see Figure 24) are the most important blocks for groups since the end-user has the flexibility to change the state for one or more devices that belongs in a group at the same time.

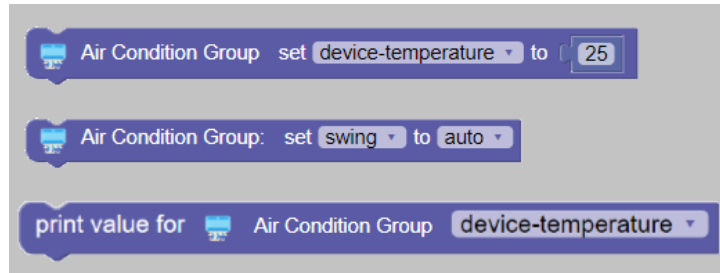


Figure 24. Setter and Output *Blockly* Blocks for device group

5.2.3 Conditional

Another set of blocks provided by Blockly Studio is the conditional blocks that exist to enable end users to define conditions based on the state of the properties of the smart device. There are two types of blocks and some extra that are defined to help the users for creating flexible IoT automations.

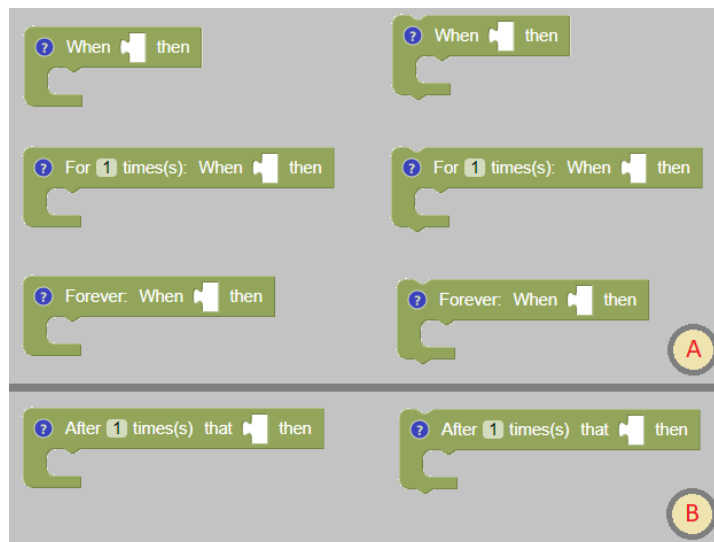


Figure 25. *When* Conditional blocks (A), *After* Conditional blocks (B)

The first type is *When* conditional blocks (see Tag A of Figure 25). The left list can only be used as parents in contrast with the right one that can be executed as statement. In detail, in *Blockly Studio* there is a simple *When* conditional that when it is evaluated to true, its inner blocks (i.e., children) are executed. Furthermore, there are two more complex blocks, the first one defines how many times their children are executed when the evaluation has result of true and in the second the children are executed every time the condition is evaluated to true.

Moreover, one more type of block exists this is the *After* (see Tag B of Figure 25) scheduled block. The children (i.e., statements) of this block are executed only when the condition has been evaluated to true so many times as the end-user has given in the input field.

Furthermore, as we said at the start of the paragraph, the need for more blocks (see Figure 26) arises for giving flexibility for the end-users to build any conditional scenario with smart devices.

Some of extra blocks are the *break* and *continue* (see Tag A of Figure 26). The first one terminates the execution of parent and the program control resumes at the next statement following parent block. The latter works somewhat like the *break*. Instead of forcing termination, it forces the next execution of parent block, skipping any child that is under of it. Parent can be only *When* or *After*, otherwise blocks are inactive as we can present in Figure 27.

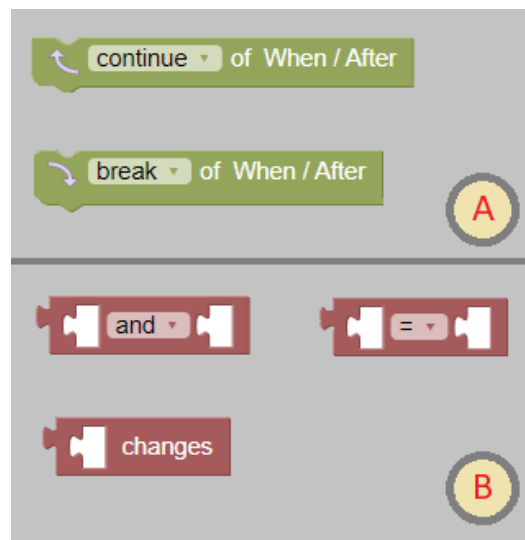


Figure 26. *Continue/Break* blocks (A), *Extra conditional* blocks (B)

Finally, there are some other blocks for evaluating conditions (see Tag B of Figure 26). The first one is for the logical operators (and, or, not). Also, for the evaluation of properties of smart devices a block with relational operators has been defined. The last block gets as its input inner block a getter of a smart device property, to check if this property's value changed. This block is executed repeatedly. The first time it

initializes the value and for every next time it is executed, it retrieves the smart device's value and checks if something changed.



Figure 27. Break/Continue blocks outside of When/After block

5.2.4 Scheduled

The next category of blocks that are provided is focused to the calendar and time events. Particularly, using this category, the end-users are able to define events which will be triggered based on time or date in repeatable basis or once. We have identified three blocks for calendar and time events (see Figure 29).



Figure 28. Break/Continue blocks for Every

The first is the “At” block which is executed once at a specific time or date. The second is the block *Every* which is executed repeatedly every specific time or date. The last is the block *Wait* which is executed once after a specific time. Also, blocks on the right list can be used as children on body for either conditional or calendar events. The blocks which are remaining (see Tag B in Figure 29), are used as inputs for the blocks that we described.

Except from the blocks that we described in the previous paragraph; there are the *break/continue* blocks for the “*Every*” (see Figure 28) block which are used in the same way as for the conditional blocks.

5.3 Types of automation provided by the IoT domain framework

In this section we present the types of automation that end users can develop using the visual programming blocks provided by the Blockly Studio IoT domain framework.

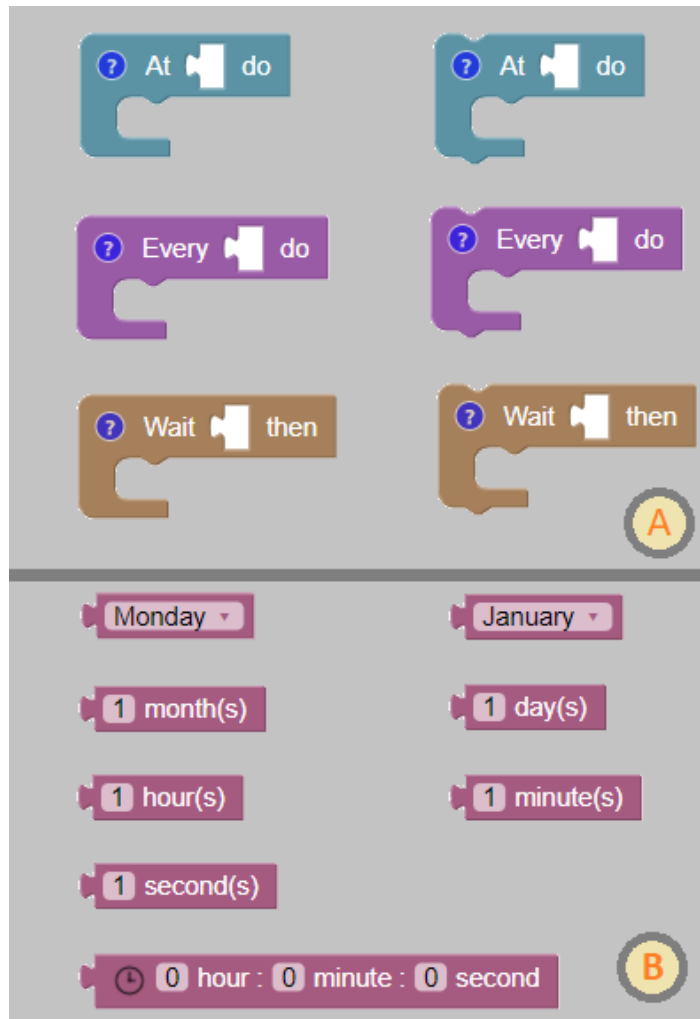


Figure 29. Blocks for scheduler events

5.3.1 Automations for Scheduled Tasks

In the category of “Automations for Scheduled Tasks” users are able to create automations that are focused to the calendar and time events. Also, when the users create their own automations, a new category is created in the *Blockly* toolbox which

includes all three types of *Automations* (i.e., *Scheduled Tasks*, *Conditional Tasks* and *Basic Tasks*).

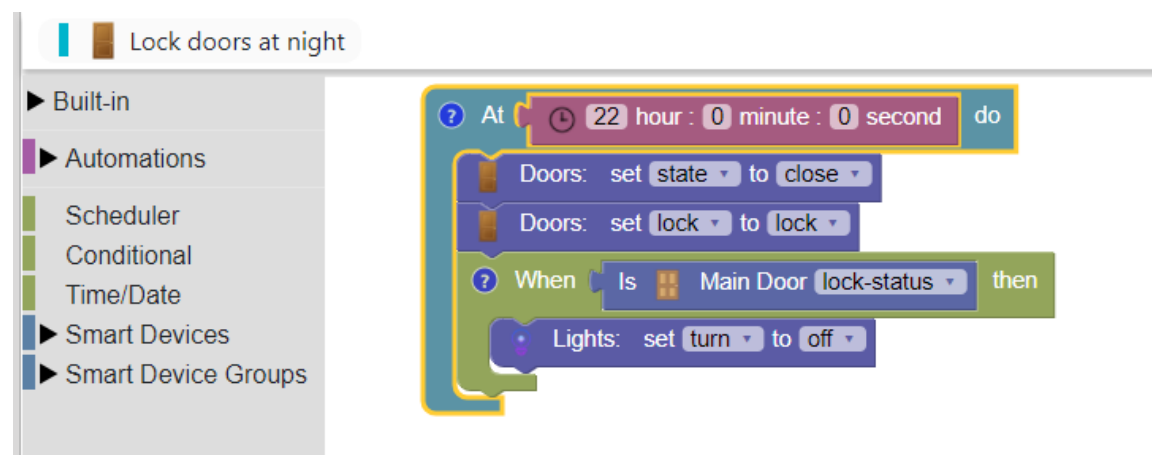


Figure 30. An Automation for Scheduled Task

Moreover, the toolbox for the automations for scheduled tasks includes all blocks of *Scheduled* category which are described in 5.2.4. Also, user is able to use and the Conditional blocks which can be used as statements (5.2.3). Finally, it includes all blocks that are corresponded to functionality of smart devices and device groups.

In Figure 30, we present an example of an automation for scheduled task. In detail, it is used the “At” block to determine that in 22:00 o’clock doors will be locked. Also, we used the conditional block “When” as statement to turn of the devices of group *Lights* when the door *main-door* will be locked.

5.3.2 Automations for Conditional Tasks

The next type of automations is the “*Automations for Conditional Tasks*”, the user is able to create automations that are related to the current state of smart devices or devices groups. The generated blocks for starting manually this type of automations which are mentioned in 5.2.3, are included in the sub-category “*Conditional Tasks*”.

The end-user is able to create automations that control and inspect the properties of smart devices or devices groups. Also, the *Blockly* toolbox of these automations includes the *Conditional* blocks which are mentioned in 5.2.3.

Furthermore, it includes blocks that are focused to calendar and time events (5.2.4) but only these which are used as statements. Lastly, toolbox includes blocks for devices and groups functionality.

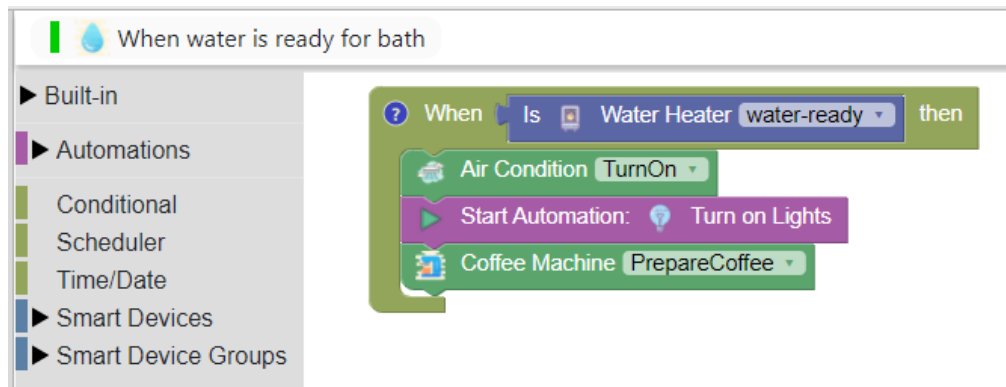


Figure 31. An Automation for Conditional Task

In Figure 31, we present an automation for conditional task. Four blocks are used. The first one is the “*When*” conditional block which checks when the water is ready for bath. When the water is ready, the Air Conditioning device turns on. Also, the automation for basic task “*Turn on Lights*” starts to turn the devices of group *Lights*. Finally, the last statement of the conditional automation is the execution of action “*PrepareCoffee*” of Coffee Machine.

5.3.3 Automations for Basic Tasks

The last type of automations is the simplest one. It includes “*Smart Devices*” and “*Smart Device Groups*” categories which are described previously.

Finally, using this type of automation the end-users can create automations that consists of blocks with the functionality of a smart device or a smart device group. In Figure 32 we present an example of automation that consists of four blocks. The first two blocks are for turning on Water Heater and Coffee Machine devices. Also, a block for starting prepare coffee for the Coffee Machine is used. Moreover, the last block is used for starting manually a defined automation for basic task which is used for turning on all the Light devices. As we can see from the example there is not the categories for conditional or scheduled blocks.

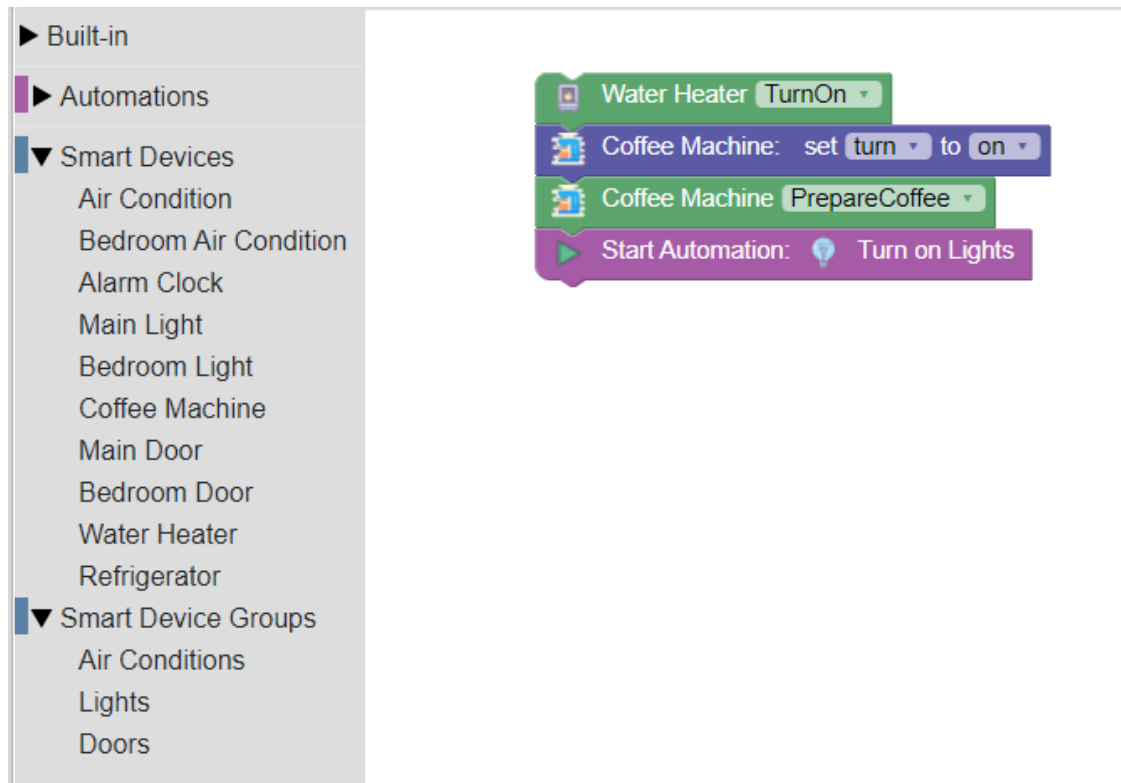


Figure 32. An example of Automation for Basic Tasks

5.4 Runtime of Automations

Using the elements that we described in the previous sections, the end-users can develop and execute their automations. In this section we describe the runtime of IoT automations. Moreover, we describe the execution process and the tools used during it, as well as the interaction between tools and smart devices (see Figure 33).

Firstly, the Blockly Studio collects and provides the required data of project elements. This data consists of all the project elements that the users have defined during the development process. Particularly, the data for smart devices that participate in the development and the source code of the automations. The source code is generated from visual programming blocks included in automations.

In addition, after we take the data of EUD (End-User Development) process, we have to initialize the communication with the smart devices that have been used in the automations. Using the API of the middleware (i.e., IoTivity) we retrieve the state of the devices. From that point on, the execution window communicates with devices for possible changes in their state.

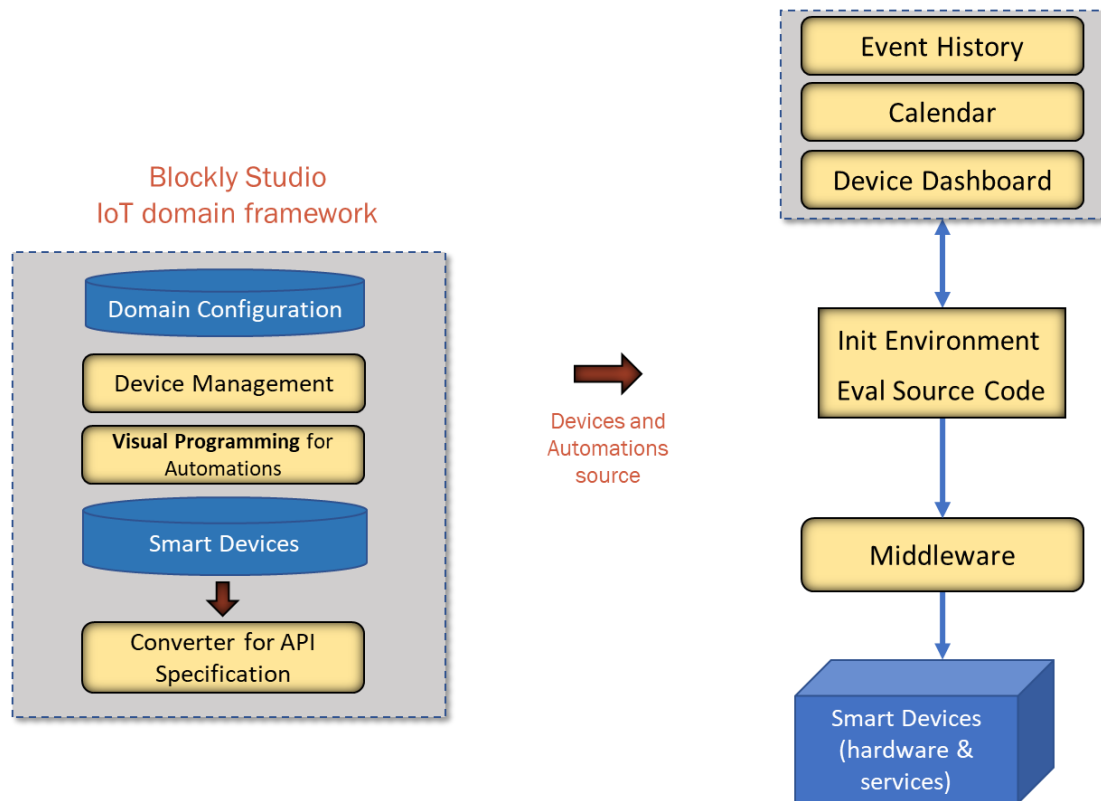


Figure 33. Overview of execution of automations

After successfully communicating with the devices, we need to initialize the tools used during the execution. First, we use the automatic UI generator to display smart devices. We also create the calendar that is useful for keeping track of scheduled tasks, and then initialize the event history that records all the events that were triggered during execution. All aforementioned tools are described in the following sections.

Lastly, to successfully execute the automations, we have to execute the source code generated from them. The source code of automations generated from the visual programming blocks used in automations. Every type of blocks generates

specific source code that interacts with the existing tools (i.e., calendar, event history) and smart devices.

5.4.1 Device Dashboard

As we mentioned in the previous paragraphs at the start of execution process, we initialize the communication with devices. Particularly, this communication is established through the middleware and in our case IoTivity. In the initialization phase we ask the state of each device. In addition, using IoTivity we bind observers to the devices for tracking changes on their state. After a change on the value of property, the MicroUI of this property is highlighted (see Figure 34).

The visual programming blocks that change the state of devices (i.e., setter and action blocks, section 5.2.1) generate code that, when executed, sends requests to the devices. Additionally, during execution, the getter blocks source code requests and receives the value of the device property.

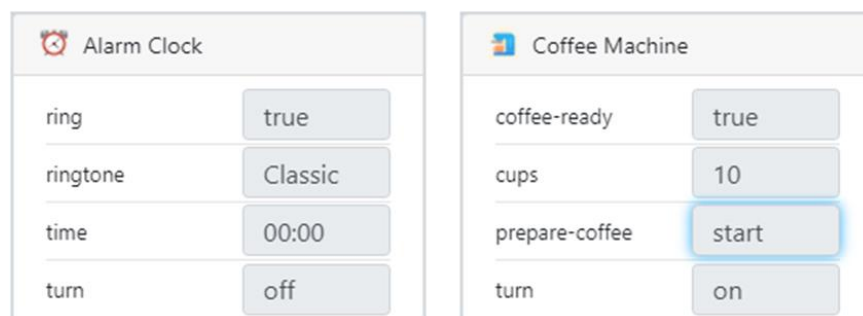


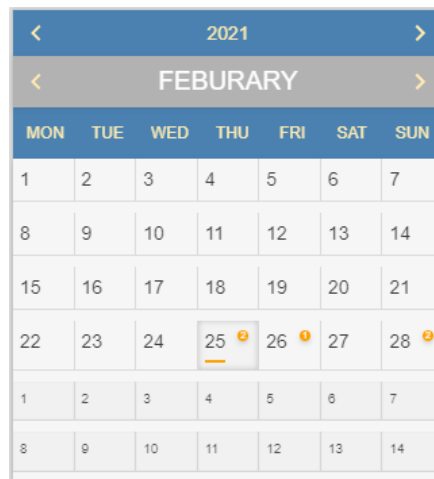
Figure 34. User interfaces for smart devices on runtime generated by *User Interface Generator*

For the visualization of the smart devices (see Figure 34), we use the *User Interface Generator* presented in chapter 4 . In detail, the properties of smart devices are rendered with read-only MicroUIs (section 4.2.1). Smart devices are visualized in read-only mode, because users are not allowed to change their state directly, the state of devices changes only by executing the automations. Additionally, the user interfaces that are created include the name and image selected by the end user when defining it in Blockly Studio.

5.4.2 Calendar

A key tool that was created and used in the automation execution window is the calendar. It is used for tracking the scheduled tasks defined in automations. The blocks that are used for scheduled tasks (section 5.2.4) generate code that interact with the calendar tool. Specifically, we use the JavaScript "*setTimeout*" function to specify the specific time that tasks must wait for their execution. Every task that is created from scheduled block is recorded to a day of calendar (see Figure 35).

For the creation of calendar, we use the *Javascript Calendar & Organizer* library [51]. It is a library for normal calendar use and events scheduling. It fits with our need for displaying scheduled tasks. A user is able to view the calendar and time events that have been used in automations through the blocks that we mentioned in 5.2.4.



2021						
FEBURARY						
MON	TUE	WED	THU	FRI	SAT	SUN
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
1	2	3	4	5	6	7
8	9	10	11	12	13	14

Figure 35. Calendar tool on runtime environment

Moreover, the end-users using the calendar are able to detect when the *Scheduled Tasks* will be executed (statements blocks) and their finishing time (see Tag A in Figure 36). Finally, with the timings of the tasks, we provide a default message to the user for understanding which of the scheduled blocks is executed. However, users are able to change the default message and write a description which is visualized in organizer (see Tag B in Figure 36).

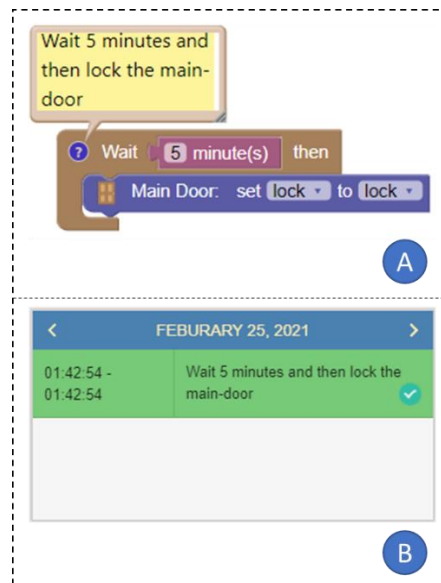


Figure 36. (A) “Wait” block with given description (B) Description of block is visualized in organizer with the starting and finishing time

5.4.3 Event History

The last tool created and used during the execution of automations is the event history. This tool logs any automation-triggered events other than the calendar-based events included in the calendar. There are two types of events in the event history: the events from conditional tasks and events from device actions. The events of the first type are generated from the execution of source code of the conditional blocks mentioned in section 5.2.3. The other events are generated from the execution of source code of device blocks that change its state. In addition, to check if the condition contained in the conditional blocks is satisfied, we use the "setInterval" JavaScript function every 200 milliseconds.

Conditional events that are visible in the event history (see Figure 37) are colored the same color as the conditional blocks from which they are created (i.e., *When*, *Forever* blocks). Additionally, conditional event bubbles include the time and date they were activated. Also, in the bubble there is a status with values: "Starts" or "Ends", the first indicates when the event starts and the last when it ends. Finally, users are able to write a description in conditional blocks, such as programmed blocks, and included in the bubble in the history table.

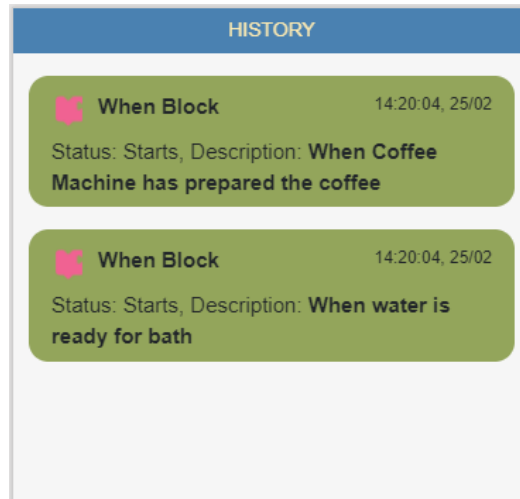


Figure 37. Event History that includes two "When" conditional events

In addition, any change in the state of smart devices from device blocks (section 5.2.1) is recorded in the event history. In detail, the actions performed and the properties changes of the smart devices are displayed with bubbles in the history, such as conditional events (see Figure 38). Each bubble takes on the color of the user-defined smart device during the development process. The bubble also includes the image of the smart device and the time and date of the event. In the case of device actions, the bubbles include the values of the arguments, and for changing device state the old and current values are displayed.

Finally, users can browse the automation that creates an event in the event history. The corresponding bubble for the event can be clicked and using the communication with the IDE data (see Figure 33) the Blockly Studio IDE minimizes the execution window and maximizes the automation workspace by marking the specific block for the created event.

5.5 Automation Testing

As mentioned in section 1.4 there is a need for end-users to test their automations to check if they are running correctly. So, we run the automations in another execution window for automation testing to introduce new automation control tools. An overview of the new execution window is shown in Figure 39.

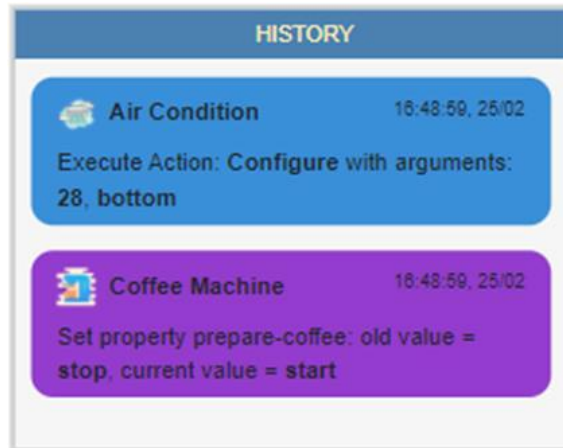


Figure 38. Event History that includes a device action and a property change of smart devices

First, like the previous chapter, the IDE provides data consisting of smart devices and automations source code. Communication with the other components of Blockly Studio is also established.

Second, there is a different approach to smart devices compared to the previous execution. In particular, to help users test their automation, we need to provide them with a set of virtual devices with the current state of their devices. This is very important because when execute automations for testing purposes we do not want to affect the condition of the actual devices.

In addition, we initialize and use an extension of the tools we create in previous execution. Moreover, we initialize a calendar tool to deal with scheduled tasks and an event history that records each event triggered during execution. Also, using the interface generator we provide device visualization during the execution. As we use a set of virtual devices, the generator uses the data of these devices. Furthermore, we create a time simulation, which the end user can control with the provided time controls. Lastly, we initialize a set of device tests to check and change the condition of the device.

Finally, the automations source code is generated by the visual programming blocks involved as in automations. The main difference is that the source code affects the virtual devices and not the actual end-user devices. Executing code for

scheduled tasks again interacts with the calendar tool, but in this execution the tasks can be activated by time simulation.

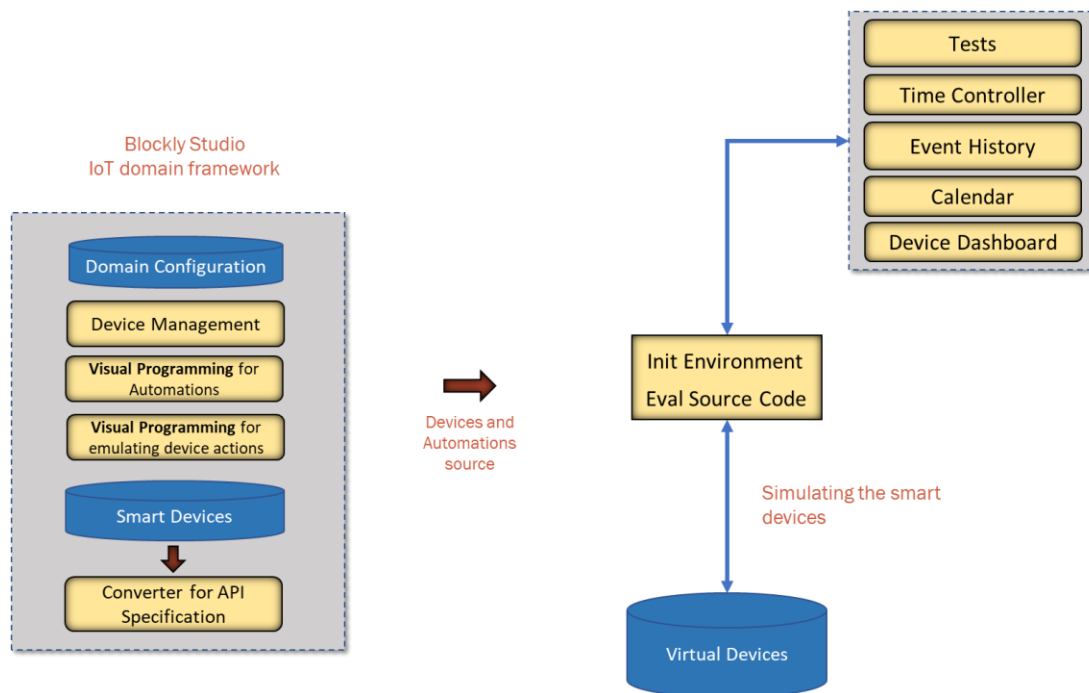


Figure 39. Overview of runtime for automation testing

5.5.1 Device Simulation

For the testing purposes of automations, the end-users want to immediately change the state and test their devices. Therefore, running IoT automations that affect real devices is not practical because assuming a user wants to find an error in a particular automation that uses smart light, he has to turn the light on and off every second for testing. Moreover, an end-user maybe wants to test a smoke sensor device, it is impossible to test it unless he lights a fire. All these gave birth to the need to make virtual devices which have the same functionality with devices used in automations. Thus, in the initialization of this execution of automations we create a set of virtual devices with the current state of the real devices.

A main problem of copying the functionality of the real devices for creating virtual is the execution of their actions. As the user does not know what operations are executed on call of every action, we have to provide a tool to simulate the actions of the virtual devices. Figure 17 shows that we provide a simulate button (in

the third list) for each device action in which end users can simulate them. Blockly Studio provides us a workspace when the button is pressed and the user has the opportunity to implement the action body that will execute (see Figure 40). Finally, for each action parameter a block is created that takes the value of the parameter at runtime.

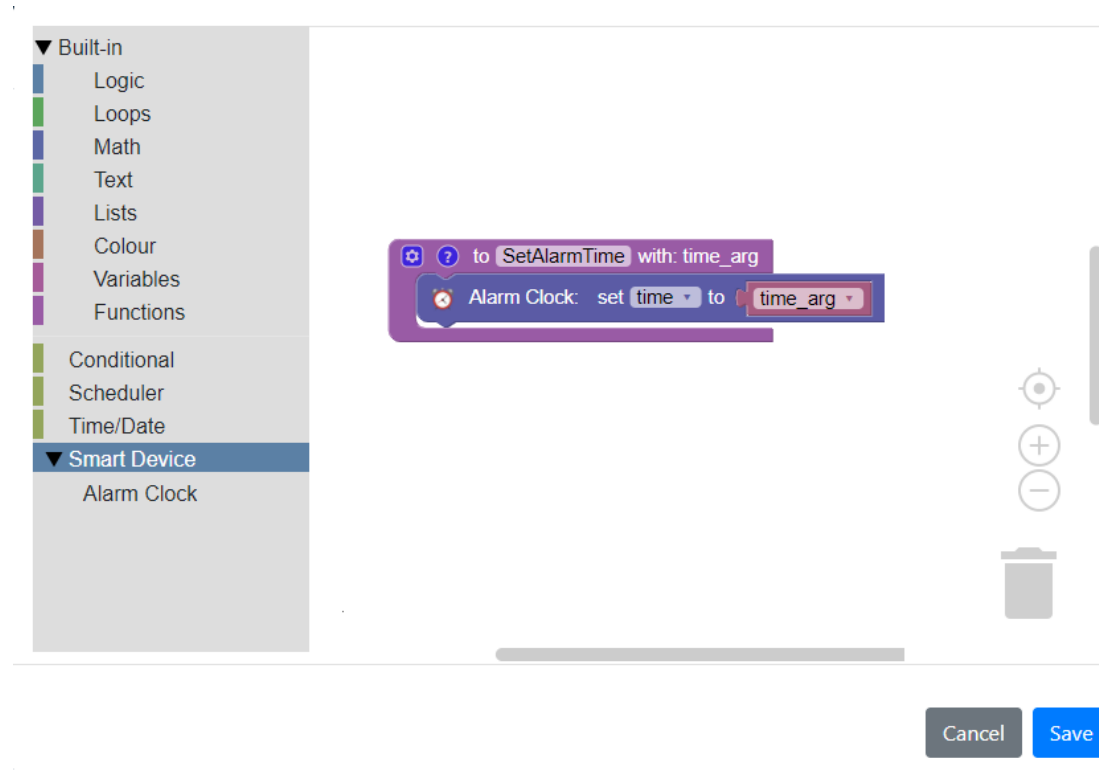


Figure 40. Implementation of an action for execution window for automation testing

Another tool that we provide on this context is the direct control of virtual smart devices. When the automations are running, the end-users are able to control the device properties (see Figure 41). The user interfaces that we provide for controlling devices are generated by *Automatic UI Generator* that we mentioned in chapter 4 . Particularly, it generates a MicroUI for each device property based on its type. Additionally, it produces MicroUI for each device action which is represented with buttons that execute the corresponding action.

5.5.2 Tools

As mentioned in section 5.5, for automation testing we expand the set of existing tools used in the normal execution. The calendar also exists in this execution window

the difference is the scheduled events now can be triggered from the simulated time. Furthermore, the event history is provided and has the same usage as the normal execution of automations.

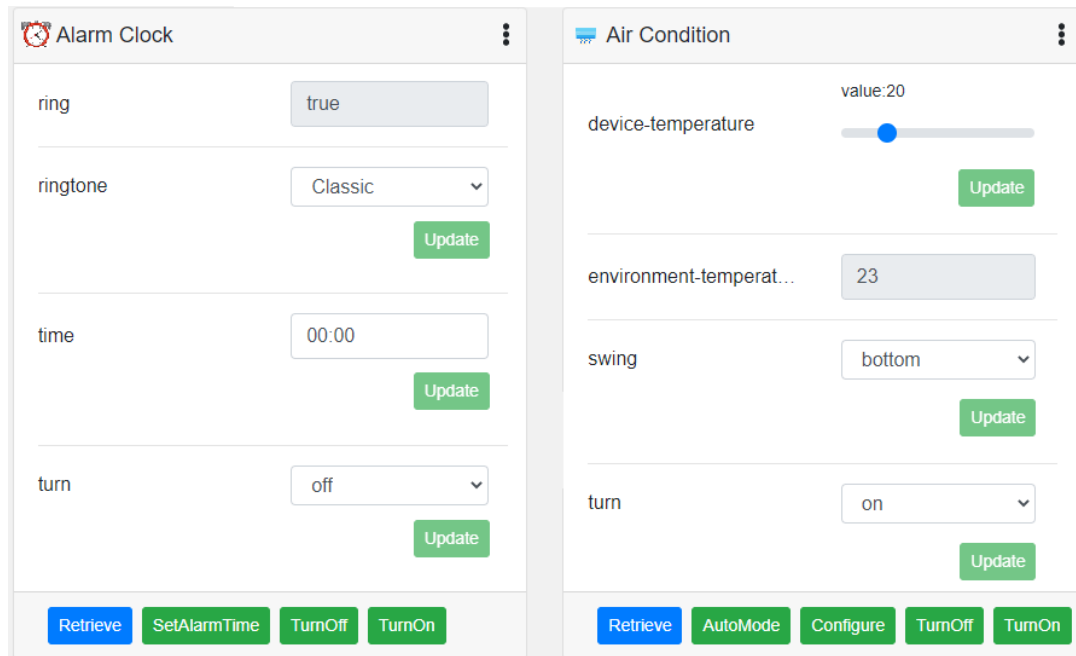


Figure 41. Control virtual devices that participate in execution for automation testing

One of the main tools we provide to users during execution is the control of simulated time. For controlling the time, we use *Day.js* [52]. It is a minimalist JavaScript library that parses, validates, manipulates, and displays dates and times for modern browsers. We provide a set of functionalities to the end-users to control simulated time to test their scheduled tasks (see tag 1 in Figure 42). In detail, the end-users can pause and continue the time. Also, they are able to make time pass slower or faster. Finally, a user can go to a specific time using the corresponding user interface (see tag 2 in Figure 42). The important thing of the latter is that when we go to the specific time in the future, all the events that need to be activated are executed sequentially as in normal execution. Also, events that were to be created by another execution of events are created and executed as in normal execution.

Last but not least, we provide an additional control panel in execution for automation testing which is the test control panel (see Figure 43). This panel records every device test created by the end-user during the execution. The device test can

be either to change the device state at a specific time or to check its state and when it is activated a message is previewed. Both types of tests are described in section 5.5.3.

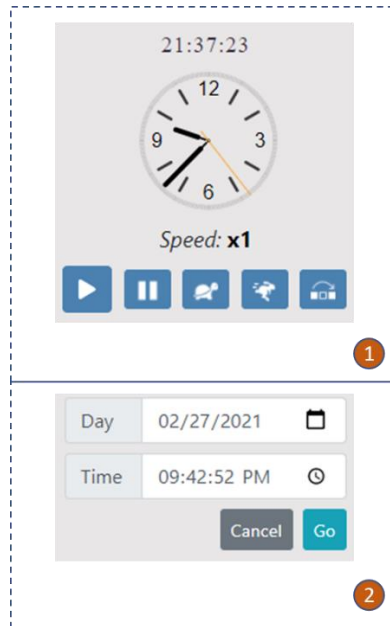


Figure 42. (1) Controls for simulated time. (2) User interface for going to specific time

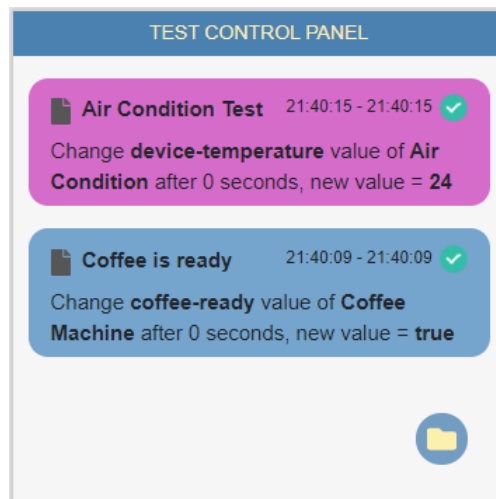


Figure 43. Test Control Panel included in the execution for automation testing

5.5.3 Tests

As mentioned in the previous section, we provide a set of tests. We create two types of test, the first one is for defining changes in the device state and the other

for checking the state of devices. Every test is executed after its creation or at the start of execution.

For the first type, we provide a user interface through which the user can define after how many seconds a change on the device state will be executed (see Figure 44). In detail, users can create more than one time that a change will be executed. Also, in each time slot the user is able to define more than one operation (i.e., property change or action execution) of one or more virtual devices. With this test we give to end users the opportunity to test the read-only device properties by changing their values and therefore to test their automations. Finally, end users can browse to the implementation of action that we mentioned in section 5.5.1 since the execution window communicates with Blockly Studio IDE.

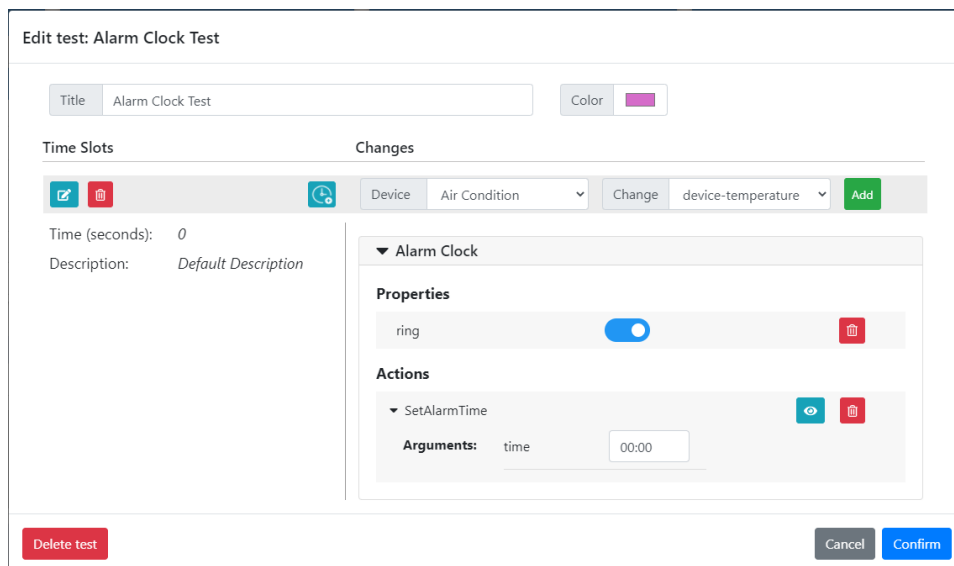


Figure 44. Define changes of smart devices at specific times

In addition, the next type of test is to check the values of the device properties. For creating this type of test, we provide a specific *Blockly* workspace through the Blockly Studio IDE. In addition, we extend the constructors of blocks of *Blockly Studio IDE* to generate two more blocks (see tag 1 in Figure 45). The first is to check the value of the property and notify the user during the execution. The latter is to notify the end user and also stops the simulated time. Both of the blocks receive a warning message that appears in the notification area during the execution of IoT automations (see tag 2 in Figure 45).

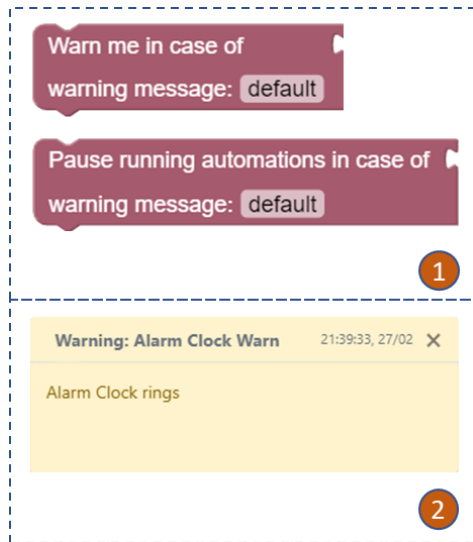


Figure 45. (1) Blocks for checking device state (2) Warning message generated from value checking test

6 Case Studies

Using the visual programming tools, we have carried out three case studies in order to validate and better present our work. Initially, we designed three scenarios of automations: *Morning Automations*, *Self-Caring Home* and *Fire Protection*. For each one we simulate smart devices using *the IoTivity* library. Afterwards, we developed the automations for every scenario and we execute them.

6.1 Morning Automations

One of the most difficult times of the day for people is waking up and their morning habitual tasks. There are several things that people have to do when they wake up such as, have a bath, prepare their breakfast, be informed about the news and their messages, prepare for their work, leave home for work etc. Using the existing smart devices, several processes could be automated and users would gain some more minutes of sleep, find the temperature of their home regulated, not forget to be informed about the news, leave home without worrying if they forgot to lock the windows or turn off lights, electric devices etc. All these automations can be accomplished when related events are triggered as depicted on the Figure 46.

The first event of application is based on the time that the alarm clock rings. When the event is fired, the alarm clock is switched off, then the air conditioning system regulates the home temperature, while water heater starts preparing water for a morning bath and the coffee machine prepares the first coffee of the day. Then, when water for the bath is ready, the window blinds open and the air conditioning turns off. Also, the bathroom door opens and the light turns on. In addition, when the windows blinds are open, Hi-Fi turns on and the "Getting Better" track starts playing. Furthermore, when coffee is prepared, Hi-Fi stops playing music and TV starts playing News. Finally, when leaving the home for work, smart devices take on the safety of the home by locking all doors and lights.

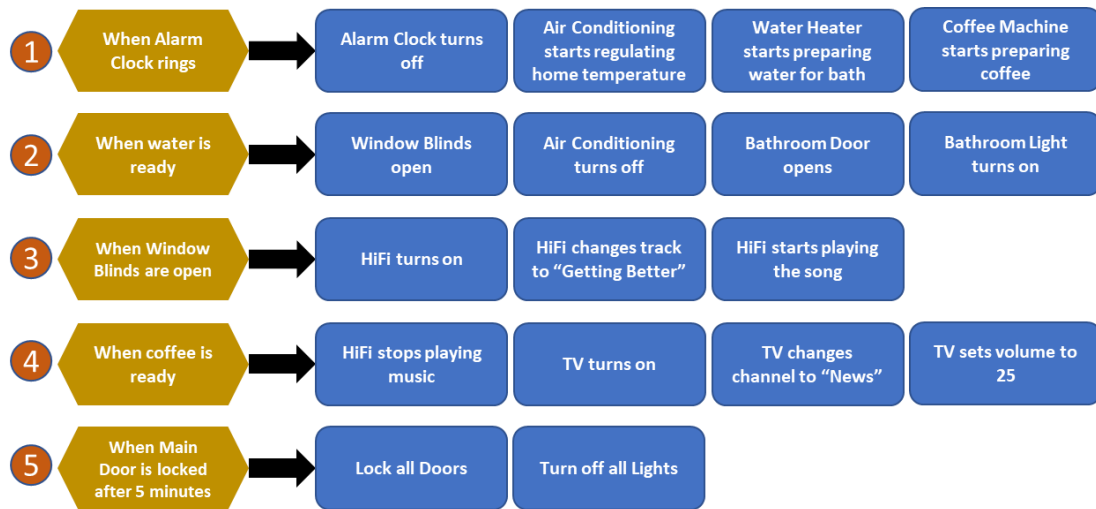


Figure 46. Morning Automations triggered by environment events

6.1.1 Devices

The smart devices included in *Morning Automations* are presented in Table 1.

Smart Device	Functionality
Alarm Clock	Turns on/off Start/Stop rings Set alarm time Change ringtone
Air Condition	Turns on/off Sets/Gets Temperature Environment Temperature Swing (auto, top, bottom)
Water Heater	Turns on/off Is water ready
Coffee Machine	Turns on/off Starts/Stops preparing coffee Rest coffee cups Is coffee ready
Window Blinds	Opens/Closes
Bathroom Door	Opens/Closes Locks/Unlocks Is locked

Main Door	Opens/Closes Locks/Unlocks Is locked
Bedroom Door	Opens/Closes Locks/Unlocks Is locked
TV	Turns on/off Sets Channel Sets volume
Hi-Fi	Turns on/off Starts/Stops music Sets track Sets volume
Main Light	Turns on/off Changes scene Sets color
Bedroom Light	Turns on/off Changes scene Sets color
Bathroom Light	Turns on/off Changes scene Sets color

Table 1. Smart Devices for *Morning Automations*

6.1.2 Automations

After we defined the required devices for the *Morning Automations*, we have to implement the automations for the scenario using visual programming blocks. In Figure 47 we present each visual program for each *Morning Automations* event.

Firstly, for the event that the alarm clock rings we use a conditional “*When*” block (see tag 1 in Figure 47). Then we fill its body with the statements that will happen in case of triggering, the action “*TunOff*” of alarm clock will be executed. Moreover, we define a basic automation that turns on the air condition and sets the temperature to 25, the *Regulate Temperature* automation. In addition, for the preparation of water for bath is called the action “*TurnOn*” and for the preparation of coffee we use the action block “*PrepareCoffee*” of the coffee machine.

For the second event of *Morning Automations*, we define a conditional automation (see tag 2 in Figure 47). In detail, we use the “*When*” block to observe when the water is prepared for bath. After the event will be triggered the “*Open*” action of window blinds will be executed. Then we call the “*TurnOff*” of air condition using the *Blockly* block. Finally, we use the visual programming block to change the property “*state*” to open for bathroom door and the action “*TurnOn*” is called.

Next, we have defined an automation for the event that is triggered when the windows blinds open (see tag 3 in Figure 47). We define a basic automation thought which the hi-fi turns on and starts playing the “*Getting Better*” track.

Moreover, we define another conditional automation for the preparation of coffee (see tag 4 in Figure 47). Using the “*When*” block we observe the “*coffee-ready*” property of coffee machine. Then when the coffee is ready, the action “*Stop*” of Hi-Fi is executed and we use the start automation block for turning on and play the *News* channel on TV.

Finally, a last automation for locking the main door is defined (see tag 5 in Figure 47). We use a combination of the “*When*” and “*Wait*” blocks for observing the state of main door and execute inner blocks after 5 minutes. We use blocks that change the value of the “*lock*” of the bathroom and bedroom door. Then, for turning off all lights of home, we call the corresponding actions using the visual programming blocks of actions.

6.1.3 Execution of Automations

After the definition of devices and automations for the *Morning Automations*, we have to run the project and present the tools of runtime environment. As the scenario based on conditional events the main tool that we are interested in is the *Event History* (5.4.3). However, there is an event which is based on calendar, so we will present and the calendar-organizer tool as well.

For the first automation (see tag 1 in Figure 47) we have added a description on the block that detects when the alarm clock rings. This description is displayed in the

generated bubble in the *Event History*. Next, all internal blocks of the program are executed and the corresponding bubbles are generated in *Event History* (see tag 1 in Figure 48).

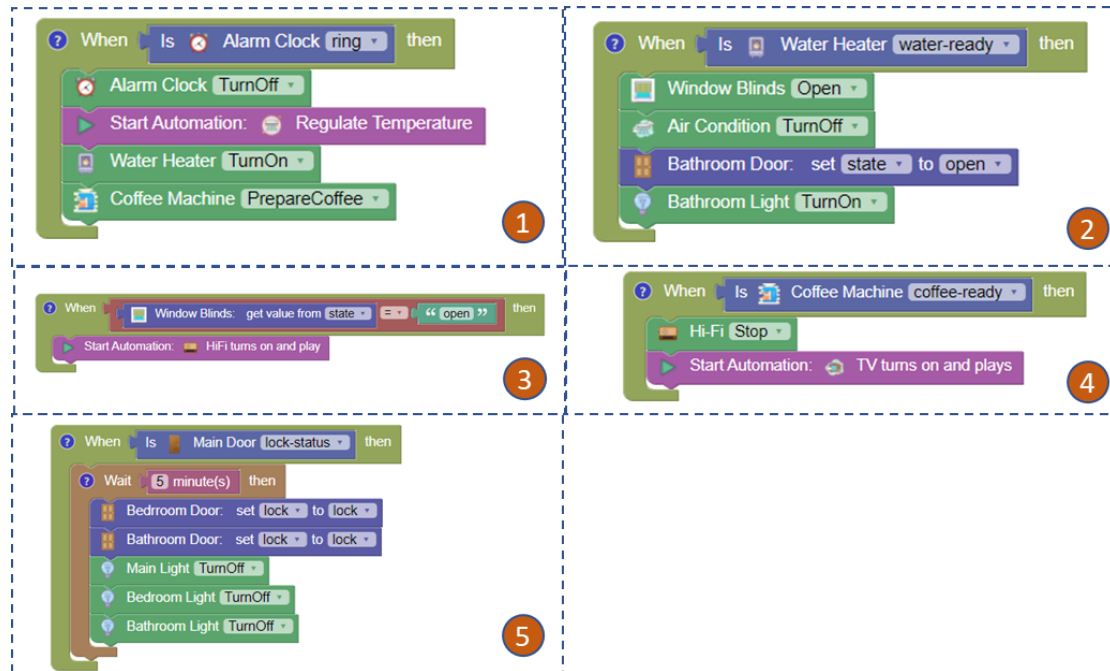


Figure 47. Visual programs for *Morning Automations* scenario using *Blockly* blocks

In addition, in the second box of Figure 48 we present the history of the events from the second automation of the project. There is a description in the "When" block as well as in the previous automation. Once the water is ready for bathing, the blocks for opening the door, opening the blinds, closing the air conditioning and the bathroom light are included in the *Event History*.

Moreover, when the window blinds are opened, the internal automation blocks create the corresponding bubbles in the *Event History* tool (see tag 3 in Figure 48). The inner block (i.e. the basic automation for Hi-Fi) creates three bubbles that correspond to the actions of the device that is turned on, changes track and starts playing.

Furthermore, the fourth box of Figure 48 shows the history of events that occur when coffee is ready by a coffee machine. After the coffee is over, Hi-Fi stops playing music and the TV starts playing News channel.

The last automation (see tag 5 in Figure 47) includes a scheduled task (i.e. the "Wait" block) to lock all the doors and turn off all the lights after 5 minutes of locking the main door. Figure 49 shows the Organizer tool, including the event based on the 5-minute log. When 5 minutes have passed, the doors of the house are locked and the lights go out as we can see from the corresponding blocks which are generated in *Event History*.

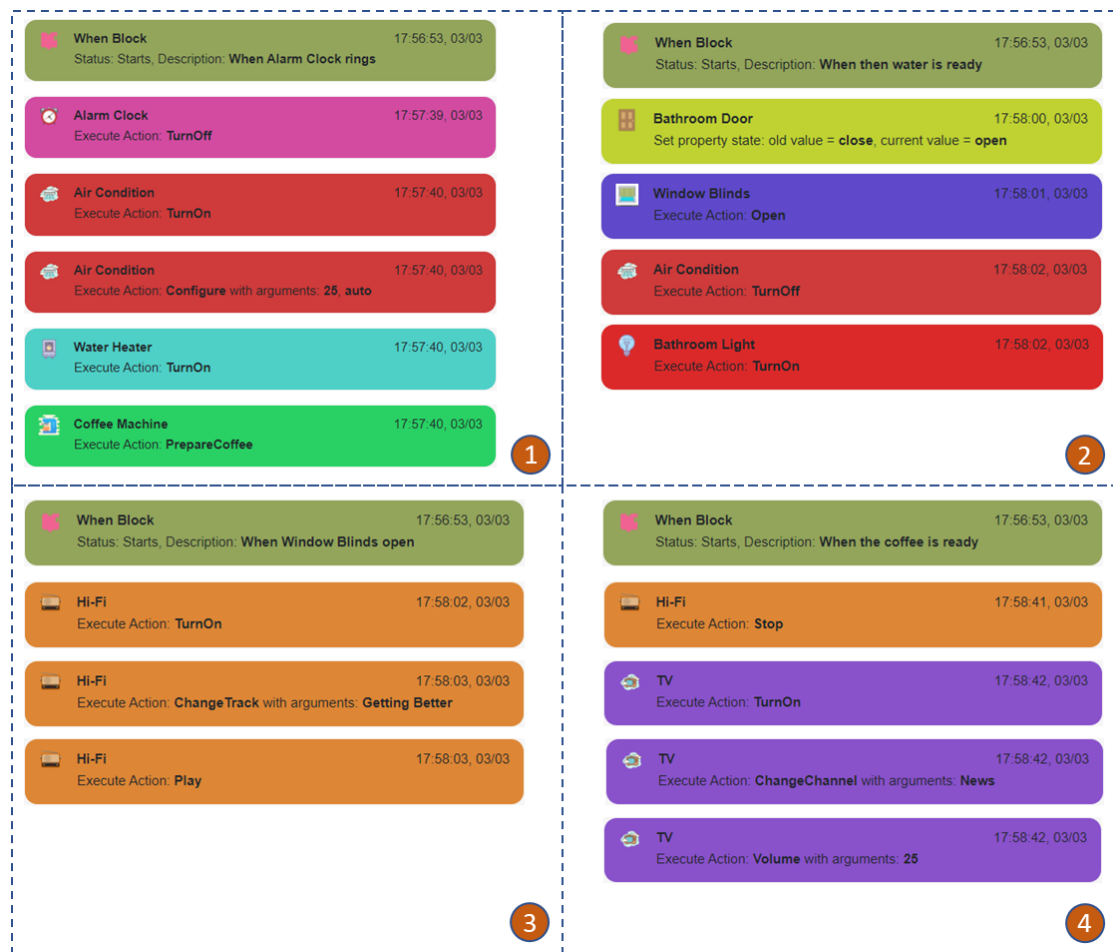


Figure 48. (1) *Event History* including bubbles generated when the "Alarm Clock rings" automation is executed. (2) *Event History* including bubbles which are generated when the "water is ready for bath" automation is executed. (3) *Event History* including bubbles which are generated when the "window blinds open" automation is executed (4) *Event History* including bubbles which are generated when the "coffee is ready" automation is executed

6.2 Self-Caring Home

Continuing the previous scenario of *Morning Automations*, end-users could design automations for tasks required for their home such as cleaning using

appropriate smart devices. However, these mainly based on calendar tasks of the home that are executed repeatedly either with the specific frequency or not. The events defined for the application of home self-care automations are presented in Figure 50.

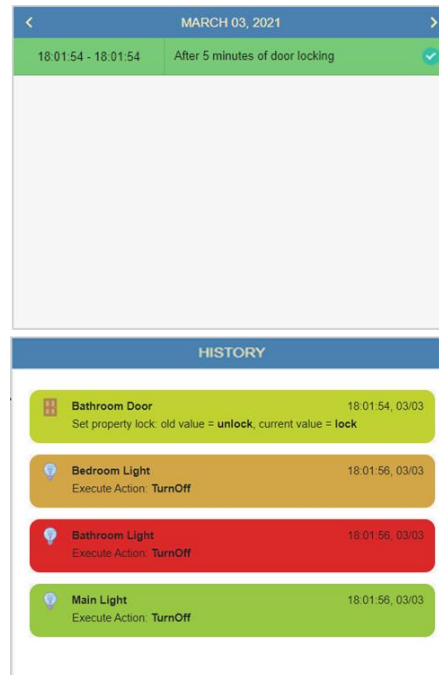


Figure 49. Organizer tool for the scheduled event and the Event History including bubbles which are generated when the "main door is locked for 5 minutes" automation is executed

The first task is programmed to be executed every day, the dehumidifier turns on in the Turbo mode and starts absorbing humidity. Then, the second task is executed every 4 days (when there are enough clothes to wash), the washing machine is set to program 2 and its temperature at 70 Celsius and then it starts washing the clothes. In addition, the smart robot starts vacuuming and mopping the house every day. Finally, there is a defrost task for the refrigerator that is executed one time per month.

6.2.1 Devices

The smart devices included in *Self-Caring Home* are presented in Table 2.

<i>Smart Device</i>	<i>Functionality</i>
---------------------	----------------------

Dehumidifier	Turns on/off Sets silent/turbo mode Sets the service to normal/dry Humidity level
Refrigerator	Turns on/off Starts/Stops defrost Filter life percent Sets rapid-cool Sets rapid-freeze
Vacuum-Mop Robot	Turns on/off Starts/Stops sweeping Starts/Stops mopping Sets clean program
Washing Machine	Turns on/off Starts/Stops washing Sets temperature Sets time period Sets washing speed

Table 2. Smart Devices for *Self-Caring Home*

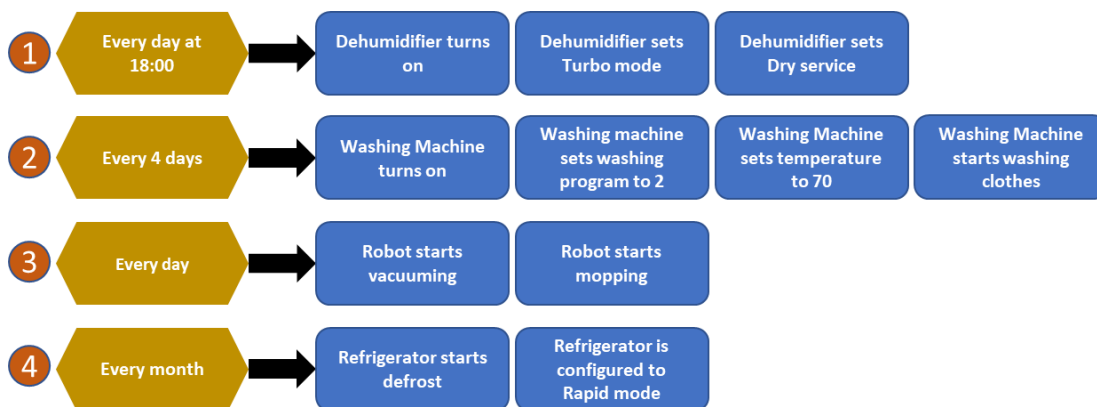


Figure 50. Home care automations triggered by calendar events

6.2.2 Automations

Using the existing device, we define and implement the automations for the *Self-Caring Home* scenario. In Figure 51 we present the visual programs for each event of *Self-Caring Home* scenario.

Firstly, we define and implement an automation for scheduled task for implementing the operations of dehumidifier (see tag 1 in Figure 51). Using the “Every” and “At” blocks, we create a scheduled task for every day at 6 o’clock. The

inner block is a basic automation that turns on and sets the dehumidifier to turbo mode. Also, the dehumidifier service is set to dry mode.

The next scheduled task that we implement with the visual programming blocks is the washing clothes event (see tag 2 in Figure 51). We use the “*Every*” block again to create a scheduled task for washing the clothes every 4 days. Also, we create a basic automation “*Washing Clothes*” that consists of turning of washing machine, setting the program 2 and the temperature to 70 Celsius. In addition, it starts the washing machine to wash the clothes.

For the next event of *Self-Caring Home* (see tag 3 in Figure 51), we use again the “*Every*” block to create a scheduled task for sweeping and mopping using the smart robot. So, every day the automation executes a basic automation that is called “*Sweeping and mopping*”. The executed automation consists of the visual programming blocks of the actions “*Sweep*” and “*Mopping*” of the robot.

Finally, for the last event of scenario we define another one scheduled automation. It consists of refrigerator blocks and they are used for defrosting task every month (see tag 4 in Figure 51). In detail, we use a “*Every*” block for the calendar task and the inner blocks is to start the defrost program and turn on both the rapid freeze and rapid cool of refrigerator.

6.2.3 Execution of Automations

After defining the smart devices of the *Self-Caring Home* scenario and implementing the required automations, we run the project and present the events which will be activated during the execution. Because the scenario is based on scheduled tasks, we use the execute the automations on the window for testing purposes to control the time.

As mentioned in section 5.5 the virtual devices are needed for the execution of the automations in this execution. Thus, each action of the devices used for the *Self-Caring Home* scenario is simulated to perform the corresponding functions of the actual action of the actual smart devices.

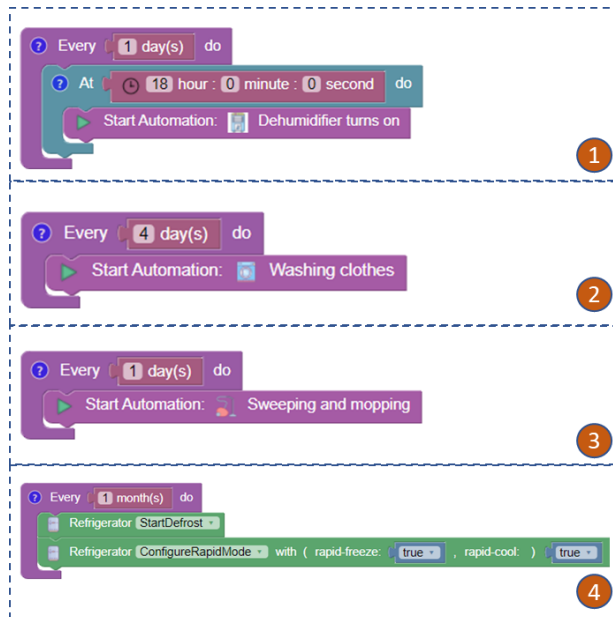


Figure 51. Visual programs for *Self-Caring Home* scenario using Blockly blocks

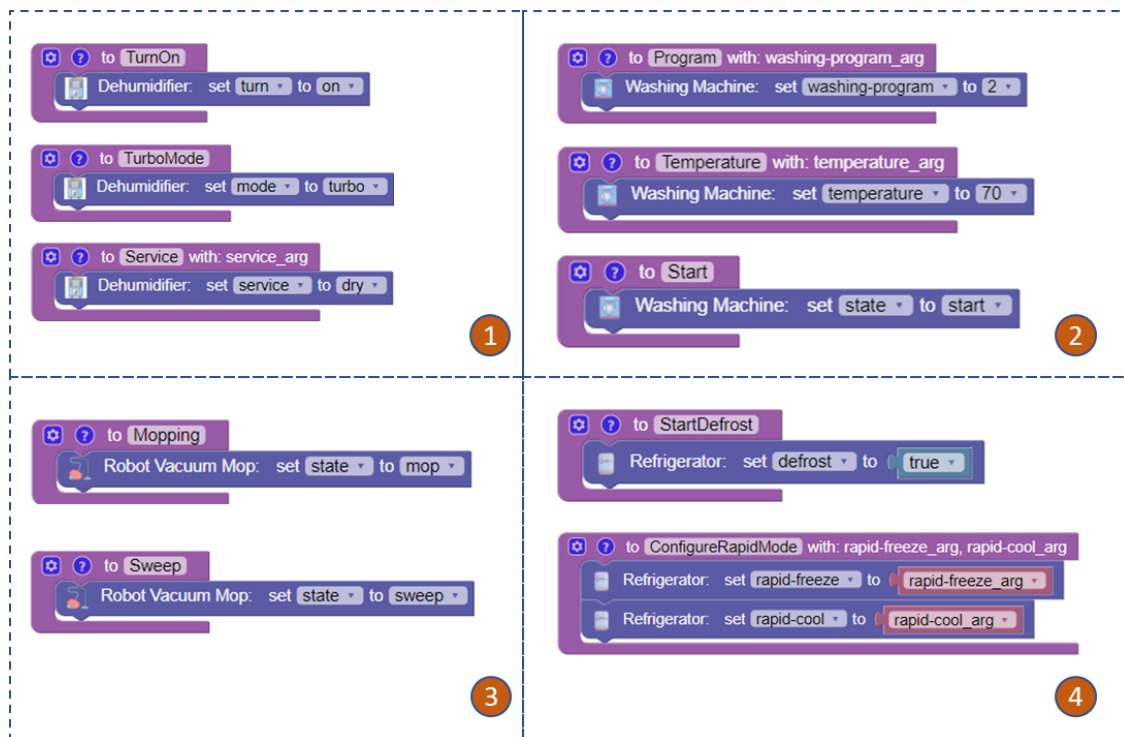


Figure 52. (1) Implementations for “TurnOn”, “TurboMode” and “Service” actions of Dehumidifier for simulated execution of automations. (2) Implementations for “Program”, “Temperature” and “Start” actions of Washing Machine for simulated execution of automations. (3) Implementations for “Mopping” and “Sweep” actions of Robot Vacuum Mop for simulated execution of automations. (4) Implementations for “StartDefrost” and “ConfigureRapidMode” actions of Refrigerator for simulated execution of automations.

Moreover, Figure 52 shows the implementation of each action which is used in *Self-Caring Home* project. In detail, the actions that we simulate for the first event (see tag 1 in Figure 51) of the scenario are the “*TurnOn*”, “*TurboMode*” and “*Service*” of the dehumidifier (see tag 1 in Figure 52). Furthermore, for the second scheduled task of the scenario we simulate three actions of washing machine (see tag 2 in Figure 52). The first action is “*Program*” that sets the washing machine in the second program. Also, the “*Temperature*” action is simulated to regulate the temperature to 70 Celsius. The last action applied to the washing machine is the “*Start*” which is responsible for starting the washing machine. Additionally, we simulate the “*Mopping*” and “*Sweep*” actions of robot for mopping and sweeping respectively (see tag 3 in Figure 52). Lastly, the actions that need to be implemented is the “*StartDefrost*” and “*ConfigureRapidMode*” of refrigerator (see tag 4 in Figure 52). The first adjusts the defrost mode in the refrigerator and the second activates the quick freeze and cooling function.



Figure 53. (1) *Organizer* tool that includes daily events and the generated bubbles of the *Event History* for dehumidifier and smart robot tasks. (2) *Organizer* tool that includes daily event and the generated bubbles of the *Event History* for washing machine task. (3) *Organizer* tool that includes daily event and the generated bubbles of the *Event History* for refrigerator task.

As shown in Figure 51, all automations of scenario consist of scheduled blocks. Therefore, we use the time simulation to advance the date and time to control the

execution of the project. The execution of the *Self-Caring Home* starts on Sunday 07/03 of 2021 and the time is 14:35. Using the *Go-To* function of the simulation tool that mentioned in section 5.5.2, we set the project day to Thursday 08/04 and the time to 14:38. We are going to present the calendar events which are executed as well as the event history.

First, there are two automations (see Labels 1.3 in Figure 51), the scheduled tasks for the dehumidifier and the smart robot for mopping and sweeping are both activated daily. Every day from the beginning of the project until the date we set, the robot first sweeping and then mopping the house. Furthermore, the dehumidifier is set to turn on every day at 18:00. The device is then set to turbo mode and the service is set to dry to start absorbing moisture. The execution of aforementioned events is presented in the first tag of Figure 53

Moreover, a scheduled task which is activated every four days has been defined. The aforementioned task is implemented by the second automation of Figure 51 and activate the washing machine for washing the clothes. As mentioned in the previous paragraph, this event is also activated from the beginning of the run until the day we set with the simulation tool. In detail, we detect via the *Organizer* tool when the event is completed and the bubbles which are created in the *Event History* (see tag 2 in Figure 53). The automation sets the washing machine to the second program, then adjusts the temperature to 70 Celsius and finally starts the device to wash the clothes.

Last but not least, another scheduled task has been set for the completion of the *Self-Caring Home* scenario. This scheduled task is activated once a month. As we started the execution of the project on 07/03, the event will be activated on Wednesday 07/04. In detail, the automation for this task puts the refrigerator in defrost mode and then executes the action for activating “*rapid-cool*” and “*rapid-freeze*”, as we can see from the bubbles created in the *Event History* (see tag 3 in Figure 53).

6.3 Fire Protection

The last scenario is for the home protection by fire. The end-users using the existing devices can design automation through which the house can put out the fire by itself. The event defined for the fire protection is presented in Figure 54.

There is a task in this automation that is performed when the smoke sensor senses smoke in the house. Then all the doors of the house open, the electrical appliances go out and the fire extinguisher starts to put out the fire in the house.



Figure 54. Fire protection automation triggered by environment event

6.3.1 Devices

The smart devices included in *Fire Protection* are presented in Table 3.

Smart Device	Functionality
Smoke Sensor	Is sensed smoke Measurement level
Fire Extinguisher	Starts/Stops
Main Door	Opens/Closes Locks/Unlocks Is locked
Bedroom Door	Opens/Closes Locks/Unlocks Is locked
Bathroom Door	Opens/Closes Locks/Unlocks Is locked
Main Light	Turns on/off Changes scene Sets color

Bedroom Light	Turns on/off Changes scene Sets color
Bathroom Light	Turns on/off Changes scene Sets color
Coffee Machine	Turns on/off Starts/Stops preparing coffee Rest coffee cups Is coffee ready
TV	Turns on/off Sets Channel Sets Volume

Table 3. Smart Devices for *Fire Protection*

6.3.2 Automations

After defining the required devices for *Fire Protection*, we create and apply two automations for basic tasks and one for conditional tasks. Figure 55 shows the conditional task defined for the implementation of the scenario. We use a "When" block to observe the state of the smoke sensor. Then, when the smoke sensor detects smoke, the internal blocks will be executed. We call two basic tasks, the first is to open all the doors of the house (i.e., the main door, the bathroom door and the bedroom door). The next automation is applied to turn off all electric devices in the house. Finally, we call the action "Start" using the corresponding block of the fire extinguisher to extinguish the fire.

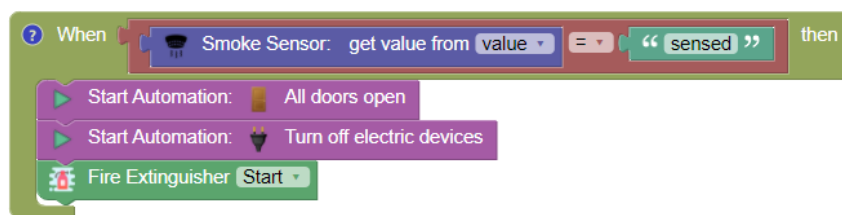


Figure 55. Visual program for *Fire Protection* scenario using Blockly blocks

6.3.3 Execution of Automations

The last phase of the *Fire Protection* is the execution of the created automations. The execution of the automations that we described in the previous section is based

on the smoke sensor “value” property. It is a *read-only* property whose value affected by changes in the environment. For this reason, we execute the automations on the window for testing purposes and we create a test to activate the smoke sensor event.

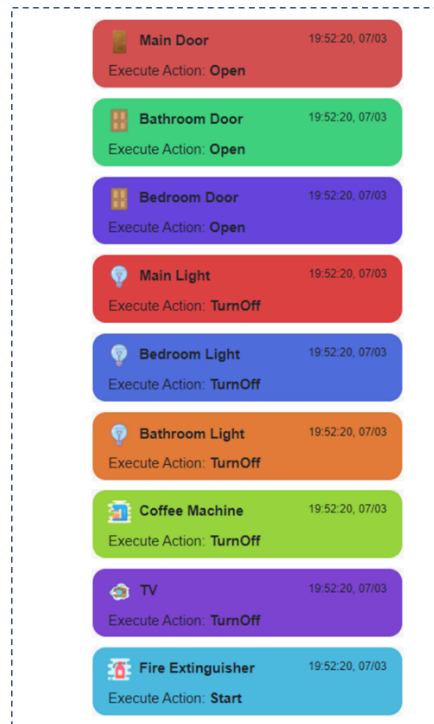


Figure 56. Event History of Running Automations of Fire Protection

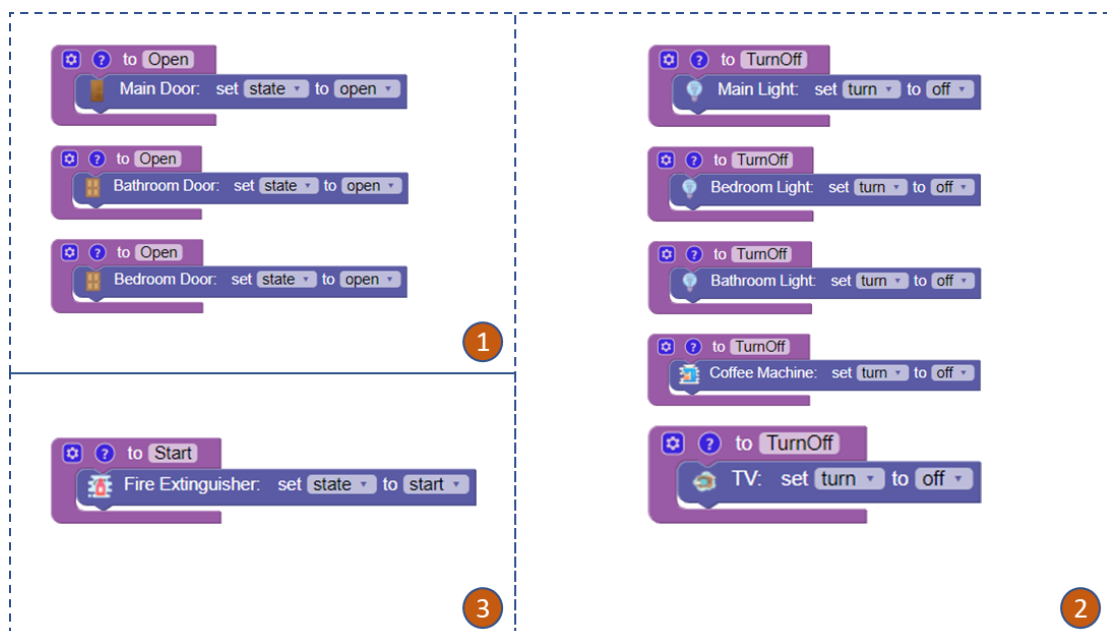


Figure 57. (1) Implementations for “Open” action of doors. (2) Implementations for “TurnOff” action of electric devices. (3) Implementation for “Start” action of the fire extinguisher.

As mentioned in the *Self-Caring Home* scenario (see 6.2), we have to simulate all device actions which are used in the automations. First, we implement for all doors of the home the “*Open*” action for opening them (see tag 1 in Figure 57). Then, for turning off all the electric devices of the house, we implement the “*TurnOff*” action (see tag 2 in Figure 57). Finally, the “*Start*” action of fire extinguisher is simulated to extinguish the fire (see tag 3 in Figure 57).

The screenshot shows a 'New Simulate Behavior Test' window. At the top, there is a 'Title' field containing 'Activate smoke sensor' and a 'Color' field with a grey swatch. Below this, the window is divided into two main sections: 'Time Slots' and 'Changes'. The 'Time Slots' section contains a single slot with a blue edit icon and a red delete icon. The slot's 'Time (seconds):' is set to '0' and its 'Description:' is 'Activating smoke sensor'. The 'Changes' section shows a dropdown menu for 'Smoke Sensor' with a downward arrow. Below the dropdown, the 'Properties' section shows a 'value' field with a dropdown menu set to 'sensed' and a red delete icon. At the bottom right of the window, there are 'Cancel' and 'Confirm' buttons.

Figure 58. Test for activating smoke sensor

For activating the smoke sensor and consequently run the automation of the *Fire Protection* we create a Simulate Behavior Test. We define a new test to change the value of the *read-only* property of smoke sensor (see Figure 58). In detail, we create a time slot and set its time to 0 seconds. This means that the test will run at the start of the project. In the unique time slot, we define a change in the smoke sensor that changes the property “*value*” to *sensed*.

After activating the smoke sensor event, the automation starts executing all the actions of the devices (see Figure 56). First, all the doors of the house open (i.e., the main door, the bathroom door and the bedroom door). Then all the electric devices go out and finally the fire extinguisher starts to put out the fire.

7 Conclusions and Future Work

Currently, The Internet of Things (IoT) is a domain that, after the Internet, represents the next most exciting technological innovation. The smart devices introduced through the IoT will help people's lives in everyday tasks. However, for devices to truly contribute to people's lives to facilitate them, they need to be included in IoT automations. The creation and execution of automations are not easy tasks as they required a minimum programming knowledge. Furthermore, the market does not provide the appropriate tools for aforementioned tasks as well as there is a lack of tools for helping them during the execution of IoT automations.

In this thesis we propose a system that consists of three components for supporting the visual programming for smart devices. First, we provide an automatic UI generator that visualizes smart devices using a generic device API that we have designed. Particularly, we have created a library that converts the device data to this API. The generator uses the device API specifications and produces interfaces for the smart devices. Secondly, a runtime environment for automations is presented that includes monitoring and interaction tools included calendar and event history to help the end-users to track changes for their smart devices. Thirdly, we provide a custom runtime environment for automation testing purposes. It includes a simulator tool that simulates real smart devices and emulates all their properties and operations. We also provide a time simulation (virtual time) for users to activate scheduled tasks. Finally, we present a suite of tests to simulate the behavior of virtual devices and check the expected values of device properties.

We have conducted three case studies to test and evaluate our system. Each case study has been created to present the capabilities of the system. We are really impressed with the use of our tools as they really help in the user experience for visualization, execution and testing of IoT automations.

In conclusion, working to build the tool for the UI generation, we realized that there is an extension that could be added to this approach. First, we can add

annotations to the API specifications for easier configuration by the developer who uses it. Through annotations, the user interfaces produced can be made more personalized. Furthermore, as future work we want to introduce a form-based mechanism used by end-users. In particular, with this mechanism users should choose in real time the appropriate Micro-UI from a set of Micro-UIs for the properties of devices. We want to give the opportunity to developers to add their Micro-UIs in the set to be used by users. Additionally, during the design and development of thesis, we needed to use our toolset in real smart devices. Finally, while using the tools included in the runtime environment for automation, we want to be able to hide either the calendar or the history table at a specific time. For the above, we want to add more functionality to our tools to allow users to hide or show the user interface information they want during runtime. We also want to introduce the aforementioned functionality into the custom runtime environment for automation testing purposes.

Bibliography

- [1] D. Raggett, "The Web of Things: Challenges and Opportunities," in *Computer*, vol. 48, no. 5, pp. 26-32, May 2015, doi: 10.1109/MC.2015.149.
- [2] R. Want, B. N. Schilit and S. Jenson, "Enabling the Internet of Things," in *Computer*, vol. 48, no. 1, pp. 28-35, Jan. 2015, doi: 10.1109/MC.2015.12.
- [3] L. Baresi, L. Mottola and S. Dustdar, "Building Software for the Internet of Things" in *IEEE Internet Computing*, vol. 19, no. 02, pp. 6-8, 2015. doi: 10.1109/MIC.2015.31
- [4] Atzori, Luigi & Iera, Antonio & Morabito, Giacomo. (2010). The Internet of Things: A Survey. *Computer Networks*. 2787-2805. 10.1016/j.comnet.2010.05.010.
- [5] Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011), 1-11.
- [6] Stojkoska, BLR, & Trivodaliev, KV. (2017). A review of internet of things for smart home: Challenges and solutions. *Journal of Cleaner Production*, 140, 1454–1464.
- [7] Ray, PP. (2016). A survey on internet of things architectures. *Journal of King Saud University-Computer and Information Sciences*.
- [8] Abraham, R., Burnett, M. and Erwig, M. (2009). Spreadsheet Programming. In *Wiley Encyclopedia of Computer Science and Engineering*, B.W. Wah (Ed.). doi:10.1002/9780470050118.ecse415.

- [9] Fabio Paternò, "End User Development: Survey of an Emerging Field for Empowering People", International Scholarly Research Notices, vol. 2013, Article ID 532659, 11 pages, 2013. <https://doi.org/10.1155/2013/532659>.
- [10] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. Communications of the ACM, 52(11):60–67, November 2009.
- [11] James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, Steve Hodges, MakeCode and CODAL: Intuitive and efficient embedded systems programming for education, Journal of Systems Architecture, Volume 98, 2019, Pages 468-483, ISSN 1383-7621, DOI: <https://doi.org/10.1016/j.sysarc.2019.05.005>.
- [12] Tynker web IDE: Educational programming platform aimed at teaching children how to make games and programs. Released on: 01/2012. Official website: <https://www.tynker.com/> Accessed online: 03/2021
- [13] Kahn, K., Rani Megasari, E. Piantari and E. Junaeti. "AI Programming by Children using Snap! Block Programming in a Developing Country." EC-TEL (2018).
- [14] Seung Han Kim and Jae Wook Jeon, "Programming LEGO mindstorms NXT with visual programming," 2007 International Conference on Control, Automation and Systems, Seoul, 2007, pp. 2468-2472.
- [15] P. Voštinár, "Programming LEGO EV3 in Microsoft MakeCode," 2020 IEEE Global Engineering Education Conference (EDUCON), Porto, Portugal, 2020, pp. 1868-1872, doi: 10.1109/EDUCON45650.2020.9125170.
- [16] P. Bachiller-Burgos, I. Barbecho, L. V. Calderita, P. Bustos and L. J. Manso, "LearnBlock: A Robot-Agnostic Educational Programming Tool," in IEEE Access, vol. 8, pp. 30012-30026, 2020.

- [17] MIT App Inventor: A web application integrated development environment provided by Google, Development Team: MIT. Released on: 12/2010. Official website: <https://appinventor.mit.edu/> Accessed online 03/2021.
- [18] Antonio Pintus, Davide Carboni, and Andrea Piras. 2012. Paraimpu: a platform for a social web of things. In Proceedings of the 21st International Conference on World Wide Web (WWW '12 Companion). Association for Computing Machinery, New York, NY, USA, 401–404. DOI:<https://doi.org/10.1145/2187980.2188059>
- [19] Paraimpu: An IoT middleware that allows users to register, manage, handle and interconnect their RESTful IoT devices or services whether physical or virtual. Released on: 01/2014. Development Team: Paraimpu. Official Website: <https://web.archive.org/web/20201201043939/http://paraimpu.com/> Accessed online: 03/2021.
- [20] Tornado is a Python web framework and asynchronous networking library. Released on: 07/2010. Development Team: Facebook. Official Website: <https://www.tornadoweb.org/en/stable/> Accessed online: 03/2021.
- [21] Nginx: A web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. Released on: 04/2010. Development Team: Nginx, Inc. Official Website: <https://www.nginx.com/> Accessed online: 03/2021.
- [22] MongoDB: A source-available cross-platform document-oriented database program. Released on: 11/2009. Development Team: MongoDB Inc. Official Website: <https://www.mongodb.com/> Accessed online: 03/2021.
- [23] Google Fit: A health-tracking platform developed by Google for the Android operating system. Released on: 10/2014. Development Team: Google. Official Website: <https://developers.google.com/fit/> Accessed online: 03/2021.
- [24] Persson, Per & Angelsmark, Ola. (2015). Calvin – Merging Cloud and IoT. *Procedia Computer Science*. 52. 10.1016/j.procs.2015.05.059.

- [25] Node-RED: A programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. Released on 10/2015. Development Team: IBM Emerging Technology. Official Website: <https://nodered.org/> Accessed online: 03/2021.
- [26] Node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine. Released on: 05/2009. Development Team: OpenJS Foundation. Official Website: <https://nodejs.org/en/> Accessed online: 03/2021.
- [27] HomeKit: A software framework by Apple, made available in iOS/iPad OS that lets users configure, communicate with, and control smart-home appliances using Apple devices. Released on: 09/2014. Development Team: Apple Inc. Official Website: <https://www.apple.com/shop/accessories/all/homekit> Accessed online: 03/2021.
- [28] Danado, José & Paternò, Fabio. (2015). A Mobile End-User Development Environment for IoT Applications Exploiting the Puzzle Metaphor. ERCIM News. 26.
- [29] Wia: A cloud platform that makes creating IoT apps easier by linking IoT devices and external services. Released on 01/2016. Development Team: Wia Inc. Official Website: <https://www.wia.io/> Accessed online: 03/2021.
- [30] Embrio: Visual, real-time, agent-based programming for Arduino. Released on 01/2010. Development Team: Embrio.io. Official Website: <https://www.embrio.io/> Accessed online: 03/2021.
- [31] SmartThings: An IoT platform for developing home automations. Released on 01/2012. Development Team: Samsung Electronics. Official Website: <https://www.smarthings.com/> Accessed online: 03/2021.
- [32] XOD: An open-source visual programming language for microcontrollers. Released on: 01/2016. Development Team: XOD. Official Website: <https://xod.io/> Accessed online: 03/2021.

- [33] Zenodys: A fully visual IoT platform for Industry 4.0. Released on: 01/2015. Development Team: Zenodys. Official Website: <https://www.zenodys.com/> Accessed online: 03/2021.
- [34] Krzysztof Gajos, David Christianson, Raphael Hoffmann, Tal Shaked, Kiera Henning, Jing Jing Long, and Daniel S. Weld. 2005. Fast and robust interface generation for ubiquitous applications. In Proceedings of the 7th international conference on Ubiquitous Computing (UbiComp'05). Springer-Verlag, Berlin, Heidelberg, 37–55. DOI:https://doi.org/10.1007/11551201_3
- [35] Clerckx, Tim & Luyten, Kris & Coninx, Karin. (2004). DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. 77-95. 10.1007/11431879_5.
- [36] Roscher, D., Lehmann, G., Schwartz, V., Blumendorf, M., & Albayrak, S. (2011). Dynamic Distribution and Layouting of Model-Based User Interfaces in Smart Environments. Model-Driven Development of Advanced User Interfaces.
- [37] Nichols, Jeffrey & Myers, Brad & Higgins, Michael & Hughes, Joseph & Harris, Thomas & Rosenfeld, Roni & Pignol, Mathilde. (2002). Generating remote control interfaces for complex appliances. UIST (User Interface Software and Technology): Proceedings of the ACM Symposium. 161-170. 10.1145/571985.572008.
- [38] Nichols, Jeffrey & Myers, Brad & Rothrock, Brandon. (2006). UNIFORM: Automatically generating consistent remote control user interfaces. Conference on Human Factors in Computing Systems - Proceedings. 1. 611-620. 10.1145/1124772.1124865.
- [39] Jeffrey Nichols, Brandon Rothrock, Duen Horng Chau, and Brad A. Myers. 2006. Huddle: automatically generating interfaces for systems of multiple connected appliances. In Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06). Association for Computing Machinery, New York, NY, USA, 279–288. DOI:<https://doi.org/10.1145/1166253.1166298>

- [40] Akiki, Pierre & Bandara, Arosha & Yu, Yijun. (2013). RBUIS: Simplifying Enterprise Application User Interfaces through Engineering Role-Based Adaptive Behavior. EICS 2013 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems. 3-12. 10.1145/2494603.2480297.
- [41] IoTivity: An open-source software framework enabling seamless device-to-device connectivity to address the emerging needs of the Internet of Things. Released on 10/2015. Development Team: Open Connectivity Foundation. Official Website: <https://iotivity.org/> Accessed online: 02/2021.
- [42] iotivity-node: A JavaScript API for OCF functionality. Released on 10/2015. Development Team: Intel Corporation. Official Website: <https://github.com/intel/iotivity-node> Accessed online: 02/2021.
- [43] IoTivity Simulator: Simulating devices which communicate with IoTivity middleware. Released on 12/2015. Development Team: Open Connectivity Foundation. Official Website: https://web.archive.org/web/20160603180432/https://wiki.iotivity.org/iotivity_tool_guide Accessed online: 02/2021.
- [44] Eclipse IDE: An integrated development environment (IDE) used in computer programming. Released on 11/2001. Developer Team Eclipse Foundation. Official Website <https://www.eclipse.org/ide> Accessed online: 02/2021.
- [45] Open Interconnect Consortium (OIC): Delivers standards for the development of the Internet of Things. Released on 02/2016. Developer Team: Open Connectivity Foundation. Official Website: <https://openconnectivity.org/open-interconnect-consortium-helps-developers-tackle-internet-of-things-with-new-developer-toolkit-2/> Accessed online 02/2021.
- [46] Roy Thomas Fielding and Richard N. Taylor. 2000. Architectural styles and the design of network-based software architectures. Ph.D. Dissertation. University of California, Irvine. Order Number: AAI9980887.

- [47] JSON Schema: A vocabulary that allows you to annotate and validate JSON documents. Released on 12/2009. Developer Team: JSON Schema. Official Website: <https://json-schema.org/> Accessed online 02/2021.
- [48] A. Savidis, "Interactive Configuration Tools and Scripts." [Online]. Official Website: <http://www.csd.uoc.gr/~hy454> Accessed online 02/2021.
- [49] Blockly: a client-side library for the programming language JavaScript for creating block-based visual programming languages (VPLs) and editors. Developer Team: Google, MIT. Official Website: <https://developers.google.com/blockly> Accessed online 02/2021.
- [50] Blockly Developer Tools: Tools for Blockly app developers to help build custom blocks. Released on 05/2012. Developer Team: Google, MIT. Official Website: <https://developers.google.com/blockly/guides/create-custom-blocks/blockly-developer-tools> Accessed online 02/2021.
- [51] JavaScript Calendar & Organizer: Library for calendar in JavaScript. Released on 07/2016. Developer: nizarmah. Official Website: <https://github.com/nizarmah/calendar-javascript-lib> Accessed online 02/2021.
- [52] Day.js: A minimalist open-source library for dates and times. Released on 04/2018. Developer: iamkun. Official Website: <https://day.js.org/> Accessed online 02/2021.