

University of Crete
Computer Science Department

FULL-SCALE VISUAL PROGRAMMING IDE:
PROJECTS, COLLABORATION AND DOMAIN PLUGINS

by
YANNIS VALSAMAKIS

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Heraklion, April 2021

University of Crete
Computer Science Department

FULL-SCALE VISUAL PROGRAMMING IDE:
PROJECTS, COLLABORATION AND DOMAIN PLUGINS

by
YANNIS VALSAMAKIS

A thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

Author: 
Yannis Valsamakis, University of Crete

Examination Committee: 

Supervisor
Anthony Savidis, Professor, University of Crete

Member

Maria Papadopouli, Professor, University of Crete

Member
DIMITRIOS-STAVROS GRAMMENOS 
Dimitris Grammenos, Researcher, ICS FORTH

Member

Nikolaos S. Papaspyrou, Professor, National Technical University
of Athens

Member
 Digitally signed by DIOMIDIS
SPINELLIS
Diomidis Spinellis, Professor, Athens University of Economics
and Business

Member
KONSTANTINOS MAGOUTIS 
Kostas Magoutis, Associate Professor, University of Crete

Member
Polyvios Pratikakis 
Polivios Pratikakis, Assistant Professor, University of Crete

Approved by: 
Antonis A. Argyros, Professor, University of Crete
Chairman of the Graduate Studies Committee

Heraklion, April 2021

FULL-SCALE VISUAL PROGRAMMING IDE: PROJECTS, COLLABORATION AND DOMAIN PLUGINS

YANNIS VALSAMAKIS

PhD Thesis

University of Crete
Computer Science Department

Abstract

Today, visual programming languages (VPLs) are the most popular programming system for non-professional developers. Originally, they have been introduced for teaching purposes, as experimental tools encouraging children to program small-scale games. Nowadays, they are increasingly treated as instruments that can give more powerful and flexible configuration, customization and extension features to the end-users of software systems, through controllable programmability relying on some exposed underlying system functionality. Such an approach has already been applied within various large-scale systems via scripting frameworks, but is still targeted to more professional users and is very demanding for the general end-user.

Overall, in the rapidly emerging era of end-user development (EUD) the adoption of VPLs seems more promising and hotter than ever before. In fact, the broad proliferation of Internet of Things (IoT) technologies has set end-user development as the vehicle to accommodate the increased personalization demands for smart automations. In particular, the IoT domain still faces a low commercial acceptance, something attributed to the low popularity of monolithic and all-in-one solutions. It is clear that there is trend towards more flexible and open infrastructures that end-users may directly tailor to their individual requirements, and even functionally combine into new ways with custom-made programmable personal automations.

However, the existing VPLs are supported with very primitive and poor tool chains, missing the notion of a full-scale integrated development environment (IDE) with all the inherently required high-quality production toolset. In this sense, the missing features should be explicitly focused on genuinely optimizing the end-user

programming process, meaning the mirroring of typical IDE functionality of the professional software development domain is insufficient and rather inappropriate.

To this end, as part of this thesis we set one grand challenge: *define, develop and validate in a demanding real domain what an integrated toolset for end-user development should offer*. To this end, the primary technical challenge has been the development of a full-scale IDE for VPLs, capable to accommodate and host virtually any VPL editor. Then, our research has revealed and supported a number of primary disciplines in the context of EUD that we have fully designed, implemented and tested in the context of our IDE: (i) *assisted project management*, (ii) *collaborative editing and debugging*, and (iii) *open interactive domain plugins*.

In particular, the role of the domain plugins is very crucial, far more than mere extension packages, with the purpose of enriching the interactive IDE functionality with extra development features optimally suiting a target domain. This notion is novel to EUD, with no counterpart in traditional IDEs, and aims to address the inherent complexity of domains for EUD due to the custom programming models and libraries that are very hard to manage without extra toolboxes on top of the IDE. To test and validate our proposition we have developed, on top of our IDE, a complete full-scale IoT plugin, including a very rich interactive toolset, for EUD support of personal smart automations.

Ολοκληρωμένο Περιβάλλον Ανάπτυξης
Για Οπτικό Προγραμματισμό:
Εργαλεία Διαχείρισης, Συνεργασίας και Επεκτάσεων

ΙΩΑΝΝΗΣ ΒΑΛΣΑΜΑΚΗΣ

Διδακτορική Διατριβή

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Σήμερα, οι γλώσσες και τα συστήματα οπτικού προγραμματισμού είναι τα πιο δημοφιλή συστήματα για μη επαγγελματίες προγραμματιστές. Αρχικά εμφανίστηκαν για εκπαιδευτικούς σκοπούς, ως πειραματικά εργαλεία που ενθαρρύνουν τα παιδιά να προγραμματίσουν παιχνίδια μικρής κλίμακας. Τα εργαλεία οπτικού προγραμματισμού στις μέρες μας αντιμετωπίζονται όλο και περισσότερο ως προγράμματα που μπορούν να προσφέρουν πιο ισχυρές και ευέλικτες δυνατότητες διαμόρφωσης, προσαρμογής και επέκτασης σε χρήστες των συστημάτων λογισμικού, μέσω της ελέγξιμης δυνατότητας για προγραμματισμό που βασίζεται στην εξαγωγή λειτουργικότητας του εκάστοτε συστήματος. Μια τέτοια προσέγγιση έχει ήδη εφαρμοστεί σε διάφορα συστήματα μεγάλης κλίμακας μέσω πλαισίων δέσμης ενεργειών, αλλά εξακολουθεί να απευθύνεται περισσότερο σε επαγγελματίες χρήστες και είναι πολύ απαιτητική για τον γενικό τελικό χρήστη.

Συνολικά, στην ταχέως αναδυόμενη εποχή μη προγραμματιστών (EUD) η υιοθέτηση γλωσσών οπτικού προγραμματισμού (VPL) φαίνεται πιο ελπιδοφόρα από ποτέ. Στην πραγματικότητα, η ευρεία διάδοση των τεχνολογιών για το διαδίκτυο των πραγμάτων έχει θέσει την ανάπτυξη εφαρμογών από μη προγραμματιστές ως το όχημα για να φιλοξενήσει τις αυξημένες απαιτήσεις για έξυπνους αυτοματισμούς. Συγκεκριμένα, ο τομέας του διαδικτύου των πραγμάτων (IoT) εξακολουθεί να αντιμετωπίζει χαμηλή εμπορική αποδοχή, κάτι που αποδίδεται στη χαμηλή δημοτικότητα των μονολιθικών και όλα σε ένα λύσεων. Είναι σαφές ότι υπάρχει μια τάση προς τις πιο ευέλικτες και ανοιχτές υποδομές που οι τελικοί χρήστες να μπορούν να τις προσαρμόσουν στις

προσωπικές τους απαιτήσεις και ακόμη και να τις συνδυάσουν μέσα από νέους τρόπους με προσαρμοσμένους προγραμματίσιμους προσωπικούς αυτοματισμούς.

Ωστόσο, οι υπάρχουσες γλώσσες οπτικού προγραμματισμού υποστηρίζονται από πολύ πρωτόγονα και φτωχά σύνολα εργαλείων, χωρίς την έννοια ενός ολοκληρωμένου περιβάλλοντος ανάπτυξης (IDE) πλήρους κλίμακας με όλα τα εγγενώς απαιτούμενα σύνολα εργαλείων υψηλής ποιότητας. Υπό αυτήν την έννοια, οι λειτουργίες που λείπουν θα πρέπει να εστιάζουν ιδιαίτερα στην πραγματική βελτιστοποίηση της διαδικασίας προγραμματισμού για μη προγραμματιστές, πράγμα που σημαίνει ότι ο κατοπτρισμός της τυπικής λειτουργικότητας IDE του επαγγελματικού τομέα ανάπτυξης λογισμικού είναι ανεπαρκής και μάλλον ακατάλληλος.

Για τον σκοπό αυτό, ως μέρος αυτής της διατριβής θέτουμε μια μεγάλη πρόκληση: να ορίσουμε, να αναπτύξουμε και να επικυρώσουμε σε έναν πραγματικό και απαιτητικό τομέα τι θα πρέπει να προσφέρει ένα ολοκληρωμένο σύνολο εργαλείων για ανάπτυξη εφαρμογών από μη προγραμματιστές. Στα πλαίσια αυτά, η πρωταρχική πρόκληση ήταν η ανάπτυξη ενός πλήρους IDE για οπτικό προγραμματισμό, ικανό να φιλοξενήσει οποιοδήποτε συντάκτη οπτικού προγραμματισμού. Στην συνέχεια, η έρευνά μας, εμφάνισε και υποστήριξε έναν αριθμό από βασικούς κλάδους στα πλαίσια εργαλείων προγραμματισμού για μη προγραμματιστές τα οποία και σχεδιάστηκαν, υλοποιήθηκαν και ελέγχθηκαν πλήρως στα πλαίσια του ολοκληρωμένου περιβάλλοντος ανάπτυξης για οπτικό προγραμματισμό: (i) *υποβοηθούμενη διαχείριση έργων*, (ii) *εργαλείο συνεργασίας στα πλαίσια της επεξεργασίας και του εντοπισμού σφαλμάτων* και (iii) *ανοιχτοί διαδραστικοί προστιθέμενοι τομείς εφαρμογών*.

Συγκεκριμένα, ο ρόλος των προστιθέμενων τομέων είναι πολύ κρίσιμος, πολύ περισσότερο από απλά πακέτα επέκτασης, με σκοπό τον εμπλουτισμό της διαδραστικής λειτουργικότητας του IDE με επιπλέον δυνατότητες ανάπτυξης που ταιριάζουν βέλτιστα σε έναν συγκεκριμένο τομέα εφαρμογών. Αυτή η έννοια είναι νέα για τα εργαλεία για μη προγραμματιστές, χωρίς αντίστοιχη στα παραδοσιακά ολοκληρωμένα προγραμματιστικά περιβάλλοντα για επαγγελματίες προγραμματιστές, και στοχεύει να αντιμετωπίσει την εγγενή πολυπλοκότητα των τομέων εφαρμογών για τα εργαλεία για μη προγραμματιστές λόγω των εξειδικευμένων μοντέλων

προγραμματισμού και των βιβλιοθηκών που είναι πολύ δύσκολο να διαχειριστούν χωρίς επιπλέον εργαλείοθκες πάνω από το IDE. Για να δοκιμάσουμε και να επικυρώσουμε την πρότασή μας, έχουμε αναπτύξει, πάνω από το ολοκληρωμένο περιβάλλον οπτικού προγραμματισμού, ένα πλήρες πρόσθετο τομέα εφαρμογής για το διαδίκτυο των πραγμάτων, που περιλαμβάνει ένα πολύ πλούσιο διαδραστικό σύνολο εργαλείων, για υποστήριξη των μη προγραμματιστών έτσι ώστε να τους ενθαρρύνει να προγραμματίσουν τους προσωπικούς τους έξυπνους αυτοματισμούς.

Ευχαριστίες (Acknowledgements)

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επόπτη μου καθηγητή του τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης, κ. Αντώνη Σαββίδη, για τη συνεχή καθοδήγηση και υποστήριξη του στο πλαίσιο της συνεργασίας μας όλα τα χρόνια στο Εργαστήριο Αλληλεπίδρασης Ανθρώπου-Υπολογιστή, του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας, ειδικότερα στο πλαίσιο της εκπόνησης της διδακτορικής μου διατριβής.

Θα ήθελα επίσης να ευχαριστήσω τα μέλη της τριμελούς επιτροπής της διδακτορικής μου διατριβής, κ. Μαρία Παπαδοπούλη, καθηγήτρια του τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης και κ. Δημήτρη Γραμμένο, ερευνητή του ινστιτούτου πληροφορικής του ιδρύματος τεχνολογίας και έρευνας, για τις εποικοδομητικές παρατηρήσεις και σχόλια που έκαναν κατά τα στάδια αυτής της εργασίας.

Επιπλέον, ευχαριστώ το Δημήτρη Λιναρίτη για τη συνεργασία μας στο κομμάτι της ανάπτυξης του framework για έξυπνους αυτοματισμούς ως προσθήκη στο ολοκληρωμένο προγραμματιστικό περιβάλλον του οπτικού προγραμματισμού.

Επίσης θα ήθελα να ευχαριστήσω τους φίλους μου για την υποστήριξη τους όλα αυτά τα χρόνια. Τέλος, πιο πολύ από όλους θα ήθελα να ευχαριστήσω τους γονείς μου Φλώρα και Νίκο. Είμαι ευγνώμον για όλη τους την αγάπη και υποστήριξη.

List of Publications

- Visual End-User Programming of Personalized AAL in the Internet of Things. Y. Valsamakis, A. Savidis - European Conference on Ambient Intelligence, 2017.
- Sharable Personal Automations for Ambient Assisted Living. Y. Valsamakis, A. Savidis - PETRA 2017.
- Personal Applications in the Internet of Things Through Visual End-User Programming. Y. Valsamakis, A. Savidis - Book Chapter in Digital Marketplaces Unleashed, 2018.
- Collaborative Visual Programming Workspace for Blockly. Y. Valsamakis, A. Savidis, E. Agapakis, A. Katsarakis – VL/HCC 2020.
- Smart Automations for Everybody: When IoT Meets Visual Programming. Y. Valsamakis, A. Savidis – IOT Companion 2020.

Table of contents

Figures of Chapter 1	xxi
Figures of Chapter 3	xxi
Figures of Chapter 4	xxii
Figures of Chapter 5	xxii
Figures of Chapter 6	xxiii
Figures of Chapter 7	xxiii
Figures of Chapter 8	xxiii
Figures of Chapter 9	xxv
1.1 Background and Motivation.....	35
1.1.1 End-User Programming.....	36
1.1.2 Visual Programming Languages	36
1.1.3 Internet of Things	37
1.2 Definition of the Problem and Objectives.....	39
1.2.1 Full-Scale IDE for Visual Programming.....	39
1.2.2 Collaborative Visual Programming.....	41
1.2.3 Smart Automations for Everybody	41
1.2.3.1 Smart Objects in Daily Life	42
1.2.3.2 Scenarios for Personal Automations	42
1.2.3.3 Scenarios for Ambient Assisted Living	45
1.3 Technical Approach and Contributions.....	51
1.4 Outline	54
2.1 Visual Programming Workspaces	57
2.1.1 Block-Based Languages.....	57
2.1.2 Flow-Based Languages	59
2.1.3 Game Development Visual Programming Editors.....	60

2.1.4	Visual Programming Approaches for the IoT	60
2.1.5	Discussion	61
2.2	Extendable IDEs	61
2.3	Tools for Debugging in End-User Programming	61
2.3.1	Debugging and Testing for IoT automations	63
2.4	Collaborative Programming Workspaces	64
2.4.1	Collaboration in Text-Based Programming	64
2.4.2	Collaboration in Visual Programming	65
2.5	Collaborative Debugging	65
2.6	Teaching and Learning Tools for Debugging	66
2.7	Code Snippets	67
3.1	Software Architecture	69
3.1.1	Shell	71
3.1.1.1	Menu Toolbar	71
3.1.1.2	Workspace Toolbar	72
3.1.2	Configuration Management	72
3.1.2.1	Basic Property Views	74
3.1.2.2	Select Property View	75
3.1.2.3	Aggregate Property View	76
3.1.3	Communication with Third-Party Applications	76
3.1.4	Openness and Extensibility	79
3.2	Extension Mechanism for Application Domain Frameworks	80
3.3	Browsing and Handling Projects of the Application Domains	82
3.4	Sharing and Versioning	84
4.1	General-Purpose Visual Programming Editors	87
4.1.1	Blockly Editor	88
4.1.2	Configuration of Editor Instances	90

4.1.3	Visual Code Snippets	92
4.1.3.1	Administering Snippets.....	93
4.1.3.2	Using Snippets	94
4.2	Domain-Specific Visual Programming Language Elements and Editors ...	94
4.2.1	Supporting Behavior of Domain VPL Elements.....	96
4.2.2	Linked Visual Programming Elements	97
5.1	Project Manager	99
5.1.1	Authoring Project Structure for Application Domains	100
5.1.2	Functionality and Style.....	101
5.1.3	Settings for Project Elements	102
5.1.4	User Action Hooks and Validation for Project Elements	103
5.1.5	Using Alternative Project Manager or None.....	104
5.1.6	Authoring by Using JSON Schemas	104
5.2	Project Elements.....	104
5.2.1	Templates	105
5.2.2	Hosting and Browsing Project Elements.....	107
5.3	Project Dependencies	107
6.1	Hosting the Runtime Environment.....	109
6.1.1	Running Projects of the Application Domains.....	111
6.2	Selective Project Execution.....	113
6.3	Input-Output Console	114
6.4	Hosting User-Interface of Application Domains at Runtime	115
6.5	Exporting Project to an Isolated Application	116
7.1	Initiating the Debugging Process	118
7.2	Debugger's Toolbar.....	119
7.3	Breakpoints.....	119
7.4	Conditional Breakpoints.....	121

7.5	Tracing.....	123
7.6	Watches	124
7.7	Execution Snapshots.....	125
7.8	Explanations	126
7.9	Supporting Debugging for Application Domain Frameworks	127
8.1	Collaborative Editing	129
8.1.1	Peer Roles.....	130
8.1.2	Local Workspace.....	132
8.1.2.1	Personal Project Elements.....	132
8.1.2.2	Toggling Live Syncing.....	133
8.1.2.3	Selective Project Execution	133
8.1.3	Initiating Collaborative Sessions.....	134
8.1.4	Collaboration Toolbar	135
8.1.5	Supported Collaboration Models	137
8.1.6	Evaluation.....	137
8.1.6.1	Aims and Design.....	138
8.1.6.2	Use Case Scenarios	138
8.1.6.3	Participants.....	140
8.1.6.4	Process	140
8.1.6.5	Results.....	141
8.2	Collaborative Debugging	142
8.2.1	Initiating Collaborative Sessions.....	144
8.2.2	Debugging Rooms.....	145
8.2.3	Visual Debugger.....	148
8.2.4	Correction Suggestions	151
8.2.5	Discussion of Supported Applications	154
8.2.6	Empirical Study.....	154

8.2.6.1	Preparing the Environment	155
8.2.6.2	Participants.....	155
8.2.6.3	Procedure	157
8.2.6.4	Results.....	158
9.1	Visual Programming Editor for Smart Objects	162
9.1.1	Communicating with Smart Objects	162
9.1.2	Managing Smart Objects Through Domain Visual Programming Language Elements.....	163
9.1.2.1	Smart Devices	164
9.1.2.2	Smart Device Environments	166
9.1.2.3	Smart Device Groups	167
9.1.3	Loading Shared Automations.....	170
9.2	Visual Programming Blocks for the Behavior of Smart Objects	171
9.3	Visual Programming Blocks for Conditional Automations	175
9.4	Visual Programming Blocks for Scheduled Automations	177
9.5	Authoring Project for IoT Automations	178
9.5.1	Creating IoT Automation Project.....	179
9.5.2	Project Elements.....	180
9.5.2.1	Smart Devices	180
9.5.2.2	Smart Device Groups	182
9.5.2.3	Visual Programming Blocks for Project Elements of Automations	182
9.5.2.4	Automations for Basic Tasks	184
9.5.2.5	Automations for Conditional Tasks	185
9.5.2.6	Automations for Scheduled Tasks	187
9.5.2.7	Handling Dependencies	189
9.6	Running Smart Automations	190
9.6.1	Execution of IoT Automations.....	190

9.6.1.1	Interacting with Smart Objects	191
9.6.1.2	Running Conditional and Scheduled Tasks	192
9.6.2	User-Interface of IoT Automations	193
9.6.2.1	Smart Devices View.....	194
9.6.2.2	Calendar View for Automations of Scheduled Tasks...	196
9.6.2.3	History View	198
9.6.2.4	Explaining Why Automations Occurred.....	200
9.7	Debugging and Testing Facilities for IoT Automations.....	201
9.7.1	Simulating Smart Environment.....	202
9.7.2	Simulating Smart Devices.....	204
9.7.3	Testing Automations	206
9.8	Case Study	208
9.8.1	Discussing of Use Case for Morning Automations.....	208
9.8.2	Initiating of the End-User Development Process	208
9.8.3	Visual Programming of Scheduled and Conditional Tasks	213
9.8.4	Running Morning Automations	214
9.8.5	Morning Automations Testing	216
9.9	Evaluation.....	219
9.9.1	Aims and design	219
9.9.2	Use case scenario	219
9.9.3	Participants	221
9.9.4	Process.....	221
9.9.5	Results	221
10.1	Summary	223
10.2	Conclusions	225
10.3	Future Work	229

List of Figures

Figures of Chapter 1

Figure 1.1. Layered Architecture of IoT.....	38
Figure 1.2. The flow of remote hospitality application and the involved smart objects.	43
Figure 1.3. Morning Automations triggered by environment events.....	44
Figure 1.4. Tina's daily activities, contacts and smart objects.	46
Figure 1.5. Tina's morning automations.	47
Figure 1.6. Tina's daily activities, destinations and smart objects.	48
Figure 1.7. Tina's transportation automations to visit Alice.	49
Figure 1.8. (T1) Tina's peace of mind automation; (T2, T3, T4) her children's peace of mind automations.....	50
Figure 1.9. The notion of professional developers (i.e., application domain authors) and end-user developers & users (i.e. novices, non-programmers) in the visual programming IDE.	51

Figures of Chapter 3

Figure 3.1. The component-based infrastructure of Blockly Studio; IDE's component infrastructure for the UI view and the component functionality is required (top-left).70	
Figure 3.2. IDE's menu toolbar including the logo of the IDE and menu items which are declared by the registered components.	71
Figure 3.3. Dialogue of the Configuration Management to configurate the dialogue parts of itself.	73
Figure 3.4. Dynamic extra number property value appears on selecting the option 'number' for the HTML font size select property value.	75

Figure 3.5. Extension layer for the Blockly Studio communication with third-party applications.	77
Figure 3.6. Communication among third-party applications and the Blockly Studio.	78
Figure 3.7. Making application domain-specific frameworks for visual programming on the top of Blockly Studio.	81
Figure 3.8. Having choose the application domain "Smart Automation in the Internet of Things" at the Start Page of the <i>Blockly Studio IDE</i>	83
Figure 3.9. Configuring the dialogue to create new application based on specific application domain.	84

Figures of Chapter 4

Figure 4.1. Blockly Editor privileges modes; editing mode (tag A), read-only mode (tag B) and not accessible (tag C).	89
Figure 4.2. Default View of Blockly's instance (top); Alternate View of Blockly's instance (bottom).	91
Figure 4.3. Visual Code Snippets Toolbar.	92
Figure 4.4. Pop-up dialogue for <i>Blockly's</i> code snippets creation.	93
Figure 4.5. Extension mechanism for <i>Blockly</i> to automatically manage the behavior handling set of blocks for visual programming language domain elements.	96
Figure 4.6. Linked visual programming language element with other visual sources.	98

Figures of Chapter 5

Figure 5.1. Configurable view parts of the project manager component.	100
Figure 5.2. Authoring settings for project element type.	103
Figure 5.3. Example of a project element template; project element information (tag 1); interactive parts of the template (tag 2); area for visual programming editors (tag 3).	106

Figure 5.4. Splitted in two project element instances area vertically. 107

Figure 5. 5. Visual programming project sources of application and dependencies among the visual programming language elements..... 108

Figures of Chapter 6

Figure 6.1. Authoring runtime of a domain project and runtime environment system of the Blockly Studio IDE. 111

Figure 6.2. Selective execution dialogue for ‘Morning Automations’ project. 113

Figure 6.3. Console input is enabled and the corresponding block is browsed. 115

Figures of Chapter 7

Figure 7.1. Debugger's Toolbar. 118

Figure 7.2. Breakpoint icons for Blockly Editor..... 120

Figure 7.3. Handling breakpoints by right clicking on *Blockly* blocks..... 121

Figure 7.4. Conditional breakpoint's dialogue. 122

Figure 7.5. Automatic variable inspection and the Evaluate operation which works for any kind of block, enabling to re-evaluate on-the fly (during debugging) any code snippet. 123

Figure 7. 6. Adding explanations for the execution of smart automations based on the environment temperature. 126

Figures of Chapter 8

Figure 8.1. Collaborative Project “Morning Automations” with 3 participants (George, Mary and James). George’s view of the collaborative project (see 1) and James (see tag 2). 130

Figure 8.2. Dialogue to create new visual code correction suggestion for a project element. 131

Figure 8.3. Dialogue to view the visual code suggestion in order to accept or deny it.	132
Figure 8.4. Left: Starting share the project; Right: Joining the collaboration.	134
Figure 8.5. Collaboration project settings.....	135
Figure 8.6. Collaboration Toolbar.	136
Figure 8.7. Participants' time to accomplish each of the scenarios.....	142
Figure 8.8. High level of our collaborative debugging approach.	143
Figure 8.9. Starting view (i.e., home page) of the collaborative debugging session.	145
Figure 8.10. Modal to create a new debugging room.	146
Figure 8.11. Viewing ' <i>Debug Room 4</i> '.....	147
Figure 8.12. Using Visual Debugger of Blockly.	149
Figure 8.13. Visual debugger's architecture for classic debugger version (left), collaborative debugger version (right).....	150
Figure 8.14. Debug Control (left); Give floor control dialog (right).	151
Figure 8.15. Creating new Correction Suggestion for " <i>Alarm Clock Rings</i> " project element.....	151
Figure 8.16. Debug the project by choosing the project items will participate and which of the project items will be original and which of them will be correction suggestions.....	152
Figure 8.17. Choosing which of the correction suggestions will be applied to original project and which of them will be saved.	153
Figure 8.18. Teaching application domain for the collaborative debugging environment.	155

Figure 8.19. Exercises asked to debug individually under supervision. (top) Program swaps x and y and adds them. Find the bug.; (bottom) Program calculates the amount of money for wages(w): $w < 1000 = 50$, $1000 \leq w < 1500 = 100$, $1500 \leq w < 2000 = 150$ and $w \geq 2000 = 300$ 156

Figure 8.20. Exercises asked to debug in groups under supervision. (top) Program attempts to output the sum of the input number's digits; (bottom) Program attempts to recognize palindrome..... 157

Figures of Chapter 9

Figure 9.1. The notion of personalized custom automations in the Internet of Things through an End-User Programming framework. 161

Figure 9.2. Importing Smart Device 163

Figure 9.3. The view parts of a registered air-conditioning device. 164

Figure 9.4. Smart device group for air-conditioning. 166

Figure 9.5. The view of air-condition living room. 167

Figure 9.6. The view of air-condition living room. 168

Figure 9. 7. Handling smart object groups for the alarm clock. 169

Figure 9.8. Replacing the 'Air Condition' smart device of the shared application with a compatible smart device..... 170

Figure 9.9. Basic Blockly Blocks for Smart Objects; actions for smart objects (tag A), setters, getters (tag B, C) and input, output for smart object properties in the I/O Console. 172

Figure 9.10. Dynamic change of a Blockly block based on the choice during the end-user development. 173

Figure 9.11. Blockly Blocks for Smart Object Groups..... 173

Figure 9.12. Conditional Event Blockly Blocks for Smart Automations. 174

Figure 9.13. Scheduled Event Blockly Blocks for Smart Automations.	176
Figure 9.14. Configuring the create application dialogue for IoT Automations and the Project Manager view based on the user’s input data.	178
Figure 9.15. Project element template that includes information and hosts one visual programming editor instance.	180
Figure 9.16. Menu options for the Smart Devices Category.	181
Figure 9.17. Creating new smart group device by choosing smart device that will export its functionality interface.	181
Figure 9.18. Choosing if automation will start automatically in the beginning of project execution or later with visual programming block element instruction.	183
Figure 9.19. Authoring <i>Blockly</i> blocks to enable the end-user developers handle manually start and stop of the automations for project elements.	184
Figure 9.20. Automations for ‘ <i>Basic Tasks</i> ’ configuration of <i>Blockly</i> editor's toolbox.	185
Figure 9.21. Automations for ‘ <i>Conditional Tasks</i> ’ configuration of <i>Blockly</i> editor's toolbox.	187
Figure 9.22. Automations for ‘ <i>Scheduled Tasks</i> ’ configuration of <i>Blockly</i> editor's toolbox.	188
Figure 9.23. Dialogues in case the end-user chooses to delete a Smart Device	189
Figure 9.24. Dialogue on connection issues of the smart devices.	192
Figure 9.25. Runtime environment for IoT automations.	194
Figure 9.26. Request to set input in property of a smart device.	194
Figure 9.27. Display of the Smart Devices at runtime environment.	195
Figure 9.28. Enabling control smart devices during the project execution.	196

Figure 9.29. Monitoring scheduled automations in the runtime environment of IoT automations.....	196
Figure 9.30. Browsing project elements that includes the scheduled blocks.....	197
Figure 9.31. Interactive bubble which depicts action of the history panel view.	198
Figure 9.32. Monitoring conditional tasks and browsing respective visual code snippets.	199
Figure 9.33. Filtering executed explanations per scheduled (top) and conditional (bottom) automations by enabling info button that opens dialogue which present them separately.	201
Figure 9.34. Simulation Environment View: tests control panel (left), date & time simulation (right).	203
Figure 9.35. Managing Simulation Behavior and Expected Values Tests.	204
Figure 9.36. Simulating smart device actions for debugging purposes.	205
Figure 9.37. Simulating behavior of smart devices at specific time periods.	206
Figure 9.38. End-user development of tests for expected values in smart devices properties.....	207
Figure 9.39. Warning message in case a test of expected values of smart device properties fails.....	207
Figure 9.40. Morning home automations example.	208
Figure 9.41. Creating morning automations and defining bedroom lighting device.	209
Figure 9.42. Workspace view having define the smart devices for morning automations.	212
Figure 9.43. Visual programming scheduled and conditional tasks for morning automations.	213
Figure 9.44. Smart Devices monitoring values for 'Morning Automations' project..	214

Figure 9.45. Calendar view of the scheduled tasks for 'Morning Automations'.215

Figure 9.46. History actions view of the tasks that will be shown running 'Morning Automations'.215

Figure 9. 47. Preparing state of smart device properties (tag A) and go at specific time in order to trigger scheduled task of 'Morning Automations'.216

Figure 9.48. Implemented actions for smart devices of 'Morning Automations'.217

Figure 9.49. Testing 'Home Safety' conditional task of 'Morning Automations': Adding breakpoint (tag A); Simulating behavior of the smoke sensor (tag B); Stepping in until the simulated fire extinguisher starts and view variables and smart device properties state (tag C); View actions history to verify the fire extinguisher started (tag D);218

List of Tables

Table 1. Project Element Privileges.....	133
Table 2. Efficiency and Usability.	141
Table 3. Fields of Use.	141
Table 4. Questions focusing on learning programming and debugging.	158
Table 5. Questions focusing on the collaborative debugging environment.	159
Table 6. Standard SUS Questionnaire.	160
Table 7. Smart Devices that are used for Morning IoT Automations.....	210
Table 8. SUS Questionnaire for the Smart Automations Workspace Environment. .	222

To my family

“Professional developers have integrated development environments and full-scale tools for programming applications. Non-programmers and learners behoove to be provided with more efficient end-user programming tools in their arsenal for developing and learning purposes”

Chapter 1

Introduction

“The whole of science is nothing more than a refinement of everyday thinking.”

- Albert Einstein

More and more devices are connected in networks resulting exponentially increasing need of development applications. As a result, these needs are not able to be covered by professional developers and the end-user programming research area attempts to address this need by empowering non-programmers to program through appropriate approaches and tools. We strongly believe that such tools could not be less powerful than the existing software tools for professional developers. We base this on two reasons. Firstly, novices or non-programmers needs more support to program an application and secondly provided tools could be the vehicle of teaching and learning programming. In this context, our work focuses on contributing by empowering non-programmers with efficient tools. In this Chapter, we analyze the background and motivation of this PhD thesis. We present the research questions and the objectives of this work. Then, we briefly describe the technical approach and discuss the contributions of our work.

1.1 Background and Motivation

Nowadays, most software programs are written by people who are not professional software engineers [1], but they may have expertise in other domains. This arises from two main reasons. First, the innumerable needs of programming that cannot be covered from professional developers. Second, there are specific requirements that are well known from people who use them and need them rather than software engineers. Using correspondent software tools, people who are not professional programmers acquire the power to develop their programming purposes without significant knowledge of a programming language. For example, a user interface designer could use a user-interface builder to develop user interfaces. This concept is an active research topic called *End User Programming*.

1.1.1 End-User Programming

End User Development (EUD) or End User Programming (EUP) can be defined as a set of methods, techniques and tools that allow users of software systems, who are acting as nonprofessional software developers, at some point to create, modify, or extend a software artifact [2]. There are various techniques and approaches which have been developed in previous decades in EUP. The most used end-user programming approach is the *Spreadsheets* that are used in the industry from professionals in several applications [3]. Some use case examples are teachers that write grading spreadsheets to save time grading, receptionists that use spreadsheets for reservations, accountants that write accounting spreadsheets for their job etc.

Another EUD technique is the use of natural language phrases interpretation. Natural languages are mainly attempt to eliminate the need for language constructs all together, and focuses on the presence of keywords in a command expression that call them as keyword commands [4], [5]. In addition, there is approach in case does not know how to perform step, try to help user with development by predicting and suggesting possible alternative phrase commands [6]. Yet another EUD approach is the scripting languages. Using scripting languages, the end-users are able to extend and adapt an existing application (e.g., Open Office Scripting Framework [7]). On the one hand scripts are the most powerful EUD tools, but on the other hand present users with a considerable learning burden and in addition scripts are prone to errors. Furthermore, there are scripting languages which use in parallel graphical tiles, giving one more friendly way to write scripts and able the possibility to have more complex and expressive scripting languages [8]. This hybrid technique from the view of the graphical drag and drop context seems with the Visual Programming which is discussed in the following section.

1.1.2 Visual Programming Languages

Visual Programming Languages (VPLs) and systems are amongst the most popular tools of end-user development (EUD) thanks to learning programming purposes which are targeted primarily at children (e.g., *Scratch* [9], *Tynker* [10], *MakeCode* [11], *LEGO MINDSTORMS* [12], *LEGO* in *MakeCode* [13], *LearnBlock* [14], etc.). VPLs allow programming with visual expressions. The basic idea is to associate icons to high-level functionalities that are important for the specific domain experts. There

are two main categories of VPLs, the jigsaws and the flow diagrams. Most of the existing approaches are focused on playing-learning purposes and they don't attempt to provide full-scale toolset of programming.

Recently, application domains in visual programming have been appeared which are not targeted at learning. The mobile applications constitute such domain for the end-users. Particularly, the use of smart phones and tablets in people's daily life lead to the explosion of mobile applications. *App Inventor* [15] is a visual programming environment that empowers the end-users with the ability to build fully functional applications for smart phones and tablets. However, this visual programming environment does not support full-scale toolset for end-user programming (e.g. debugging, project management, versioning features are missing). Moreover, *BlocklyDuino* [16] and *ArduBlock* [17] are two visual programming workspaces that focus on the application domain of programming in the context of *Arduino* [18].

In general, based on new arising technologies, new application domains in which visual programming is able to be applied in order to empower novices or non-programmers to program related applications. In this direction, a notable application domain for visual programming which motivated us to begin this PhD Journey is based on the Internet of Things era. In the next section, we discuss about the Internet of Things era. Afterwards, we analyze the research questions and the objectives of this PhD thesis. We then present the technical approach and the contributions of our work.

1.1.3 Internet of Things

The Internet of Things (IoT) is a new paradigm which refers to advanced connectivity of devices, systems, and services. The term IoT has become recently popular to emphasize the vision of a dynamic global network infrastructure of physical objects or "things" which are embedded with electronics, software, sensors and connectivity capabilities. The connection of physical things to the Internet gives them the capability of producing data, collecting information and accessing remote sensor data. Furthermore, this connectivity also allows for the control of the physical world from a distance by users. In addition, the inserted intelligence into physical objects enables them to communicate with each other and even to control each other's functional state, e.g., a thermostat sensor can control the state of an air conditioning unit by

turning it on or off when the room has reached a certain temperature reading, or by activating the window shutters. Alternatively, this kind of everyday physical objects is called Smart Objects (SOs) and is the building block of the IoT.

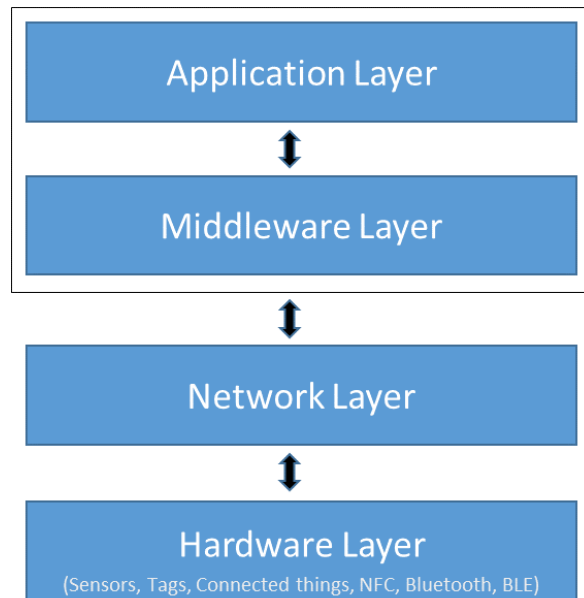


Figure 1.1. Layered Architecture of IoT.

Internet of Things has been based on a layered architecture. In Figure 1.1 is depicted, the architecture of IoT that is consisted of four main layers. The bottom layer of IoT is the hardware layer which consists of RFID tags, sensor networks and generally all kind of connected things. All kinds of information of the physical things in world that participate in IoT era are perceived and collected in this layer. The above layer of hardware is the network layer which includes access and core network, provides transparent data transmission capability. The data from hardware layer can be sent to this layer using existing mobile communication network. The upper layer from network is middleware layer. This is a software layer which facilitates the development of application. In particular, middleware hides the details of different technologies and the heterogeneity of smart objects in order to exempt the developers from issues that are not directly pertinent of their focus. The topmost layer of IoT architecture is the Application layer. This layer responsible for the delivery of a variety of applications which are provided through the middleware layer to different applications and users in IoT based systems.

The novelty of the IoT concept is not in any new disruptive technology, but is the pervasive deployment in the environment of a variety of smart objects around us, such as sensors, actuators, mobile phones, Radio Frequency Identification (RFID) tags, etc. More specifically, it is appreciated that in the next few years, smart objects that will be connected in the Internet will be approximate trillions [19]. A key part of the future Internet will be that through wireless and wired connections and unique addressing schemes are able to interact with each other and cooperate with other smart objects in order to create new services and reach common goals.

This has as a result more and more in the community of researchers and industrials moving their interest in this new trend and trying to address the new challenges, defining and creating the new world of the IoT. The main strength of the IoT idea is the high impact it will have in the behavior of people who will use it and generally in several aspects of their everyday life such as personal, societal, social, businesses, medical, environmental etc.

1.2 Definition of the Problem and Objectives

Our work targets to three main research directions. Starting the PhD journey, our first direction is focused on how could smart automations be developed by everybody exploiting the Internet of Things era and visual programming languages. This research direction led us to the next two directions of providing full-scale IDE in the context of visual programming languages which will include collaborative facilities. In this section we discuss each of them including the research questions, the key missing end-user development facilities and our work objectives.

1.2.1 Full-Scale IDE for Visual Programming

The professional programmers are empowered by integrated development environments (IDEs) which include several advanced and efficient facilities in order to program applications. However, in case of non-programmers and novices, the visual programming workspaces are treated as children of a lesser God. In particular, the existing visual programming frameworks are missing a full-scale end-user development toolset. The existing approaches are mainly targeted to children learning within the context of a game. In addition, several visual programming features are at

an infant level or not mature enough (e.g., project management, remote collaboration, debugging, intelligence, etc.).

Additionally, the visual programming frameworks are limited to specialized requirements resulting in satisfying a narrow set of needs for end-user programming. This set of needs is specialized either in the application domains (e.g. *Scratch* is a visual programming framework only for end-user development of animations) or in the audience knowledge and level of experience (e.g. focusing on end-users that have experience on flow diagrams will not be efficient for end-users that may have only experience on jigsaws). Moreover, taking into account that new applications are arising, existing application domain requirements for visual programming are fluid and third-party technologies are updated continually, constantly changing requirements for developing new IDEs for visual programming languages. For example, in the context of the IoT, communication libraries (e.g. *IoTivity*), smart services and devices are upgraded and each one of them uses different technology based on the circumstances. However, the development of an IDE for visual programming languages from scratch for each new application domain is no trivial process and it is extremely expensive.

Moreover, with the absence of one full-scale IDE for visual programming languages the non-programmers are affected as this would happen in case of developers if they didn't have an IDE. This might have been acceptable in the case of application domains that are targeted primarily on learning programming. However, there are application domains that the end-user would like to be fully empowered such as the case of personalized ambient assisted living automations in the Internet of Things we analyze in section 1.2.3. In addition, learning programming would be more efficient by using an appropriate full-scale IDE which will provide adequate end-user development facilities.

Our objective in this PhD concerns the development of an extendable IDE for visual programming languages, while offering full-scale end-user programming facilities and a mechanism to plug-in application domain frameworks.

1.2.2 Collaborative Visual Programming

One of the key features in the visual end-user programming is the collaborative programming. Visual programming languages users are novice programmers for which collaboration as a learning and support instrument is more important compared to typical experienced programmers. In particular, this feature could be notably useful in the case of using it for teaching and learning programming purposes. Additionally, this feature is able to be used in the context of asking for help from more experienced users, co-working for automations etc. Moreover, errors are able to be corrected through collaborative testing and debugging. The later makes it important for groups of end-user developers to have suitable tools to support their collaborative programming tasks. However, existing works are focusing on co editing of the visual programming process (e.g., App Inventor approach [20]), without caring to sort out the collaborative programming process. In addition, there is no approach that undertakes the testing and debugging collaboratively.

The objective of our work concerns the development of a full-scale toolset for collaborative visual programming which is able to empower novices to cooperate for end-user development process. We also target this toolset efficiently support the novices to test and debug their applications collaboratively. Last but not least objective is to support teaching and learning programming through the provided facilities.

1.2.3 Smart Automations for Everybody

In the IoT context, people's daily lives could benefit from using smart objects, as they can offer an environment of automations for everyday activities. However, in practice, the demands for such automations are highly personalized and fluid, resulting in a respective digital market that is either inexistent or marginal. Consequently, in order to fully benefit from the capabilities of this environment, individuals should be able to interact with smart objects, potentially managing, parameterizing and even programming applications involving them. In this section, we discuss the introduction of smart objects in daily life. To better represent the requirements and the benefits of smart automations in IoT, we describe potential scenarios of personal automations that could be developed.

1.2.3.1 Smart Objects in Daily Life

The IoT concept is the pervasive deployment of a variety of network connected smart objects around us, including physical things, smart devices, applications, etc. in the environment. Furthermore, devices which are commonly used in daily life have been evolved to smart connected devices by offering extra services and automations (e.g. tracking information, remote control, exchanging data with other smart objects etc.). The refrigerator is a representative example of a device used on a daily basis. Its main function is to maintain and store food items and fresh produce. But as a smart object, apart from the above functions, it will also be able to do other more complex functions such as identifying, enumerating, and holding important information about the food items it contains. Smart refrigerator notifies users when a food item is close to expire or if it has already expired. Furthermore, the refrigerator is able to display through an embedded screen, recipes based on the food items that are currently stored. Moreover, the users can remotely view what is stored in their refrigerator.

In addition, apart from the physical connected things and the smart devices, there is a huge number of applications online and day by day this exponentially increases. These applications could be used in the world of IoT and could be considered as smart objects which are connected online and are able to communicate through web-services. Such applications could be available via digital market-places. Examples of applications could be weather forecast, a clock, a chronometer etc. Furthermore, examples of such applications that could be interoperated with the smart refrigerator are a nutrition calendar and online shopping. Using these smart objects, the user will be able to program a weekly meal plan based on which the refrigerator could automatically place online orders in authorized food shops.

Taking into account the aforementioned about regarding smart objects which are available in people's daily life, people may like to have custom automations based on their needs. In the next section, we discuss scenarios of possible personal applications.

1.2.3.2 Scenarios for Personal Automations

Using existing smart objects, we discuss potential scenarios which could be developed by end-users based on the visual end-user environment we develop. However, the scenarios discussed below are just indicative, since by offering end-user programming

features and due to the fact that there is a huge variety of smart objects available, the possibilities are endless.

Remote Hospitality

People would often like to be at home (or office) when their doorbell rings but instead they happen to be somewhere else. This happens either when there is a meeting for which they couldn't be there on time or in case of a surprise visit. Before the existence of IoT concept, visitors could only call the potential hosts in order to communicate with them. Thanks to IoT, people are able to use smart doorbells which are supported by appropriate software applications. The latter notify users when the doorbell rings and help them communicate with the person who rang it. On the one hand, smart doorbell software provides support for all possible services of the device, on the other hand, it is impossible to provide support for other smart objects that users would like to use with the smart doorbell. For example, end-users may like to have an application which uses home smart objects in order to host visitors remotely until they go back at home as depicted in Figure 1.2. The smart door gives access to the visitors, while the smart lights turn on or window blinds open depending on the time of the day. Furthermore, home temperature can be regulated using the air conditioning system and the thermometer. Then, the smart Hi-Fi or TV could take on the visitors' entertainment. In addition, drinks can be prepared by the smart coffee machine or the smart kettle.

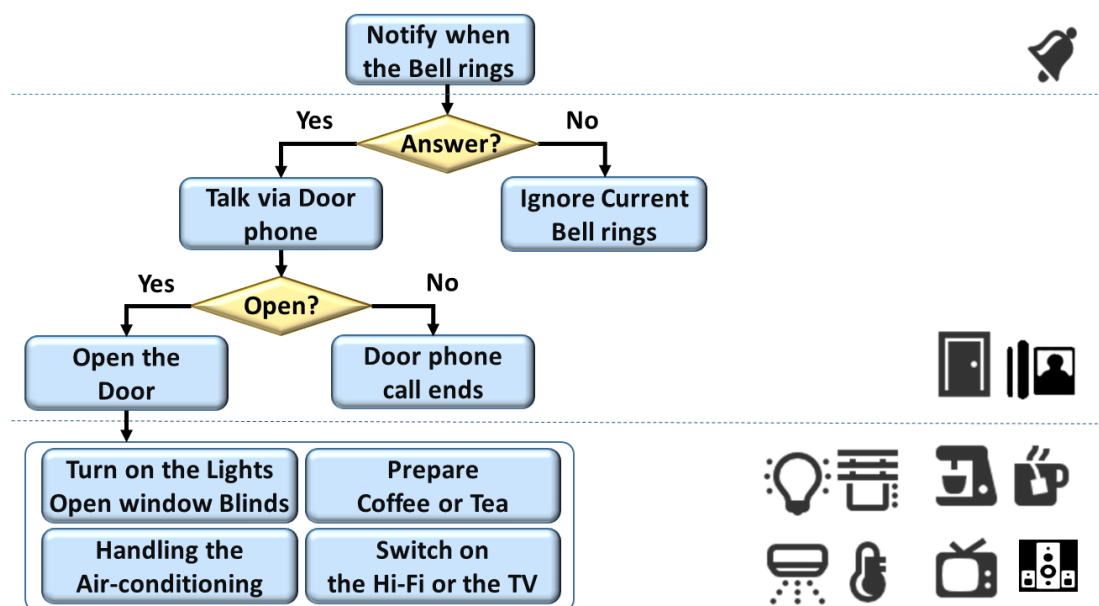


Figure 1.2. The flow of remote hospitality application and the involved smart objects.

Someone would wonder why we have to create a new application using smart objects and not use all the provided applications from our smart objects. The answer is twofold. First, users would like to have custom automations without having to use each of the applications of the smart objects. In addition, running applications for each smart object would be impossible in case of using several smart objects for one task something which would be a common scenario in the concept of the IoT which is based on the pervasive deployment of smart objects around the world. Second, there are several cases that smart objects are based on the events and data of other smart objects. A representative example of such application is discussed on the next section describing morning automations.

Morning Automations

One of the most difficult times of the day for people is wake up doing 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 objects, several processes could be automated and users would gain some more minutes of sleep, find their home temperature regulated, not for-get 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 1.3.

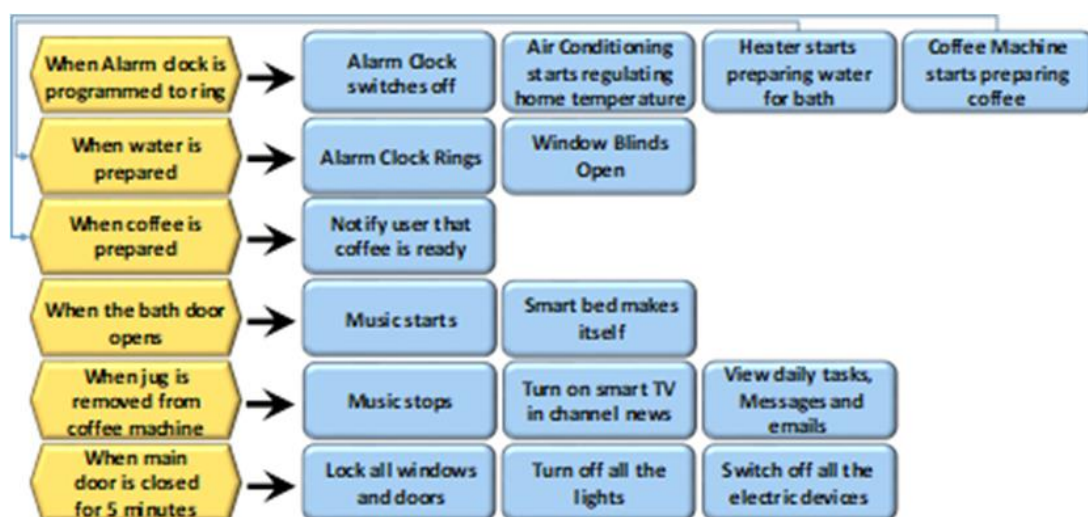


Figure 1.3. Morning Automations triggered by environment events.

The first event of application is based on the time that the alarm clock is programmed to ring. When the event is fired, the alarm clock is switched off before it rings, then the air conditioning regulates the home temperature, while heater starts preparing water for a morning bath and the coffee machine prepares the first coffee of the day. Once the water for the bath is ready, the alarm clock rings and the window blinds open. Also, when coffee is prepared, the coffee machine notifies the user. Afterwards, when the user opens the bath door, the smart Hi-Fi automatically starts playing music and the smart bed makes itself. Afterwards, when the user starts serving coffee (once she has finished with her bath) music stops and it is time to catch up with the news and view the daily tasks she has to do, messages or email she has received. Finally, when leaving home for work, smart objects take on the home safety by locking all windows, window blinds and out-doors which are still open, switching off not used electric devices such as the air conditioning, the TV etc. turning off the lights and activating the alarm system.

1.2.3.3 Scenarios for Ambient Assisted Living

Moreover, in the context of smart personal automations could be developed applications that will focus in the Ambient Assisted Living (AAL). AAL aims to support the elderly and disabled in their daily routine and health care by extending their independent living as far as possible. Particularly, in the case of elderly people, AAL attempts to encourage and maintain their autonomy by increasing their safety in their home environment, improving their daily life activities and reducing the burden on societal economics from the assisted care of elderly people [21]. Main categories of applications of AAL for the elderly are health (e.g. medications, pill reminder), safety (e.g. emergency button, fall detection), peace of mind, social contact, mobility, security etc. Applications of Ambient Assisted Living can be implemented on top of the Internet of Things [22][23][24], the emerging paradigm regarding the deployment of network connected smart objects in the environment, including physical things, smart devices, applications, etc.

In this section, we discuss scenarios that are focused mainly on the elderly and on the way their daily life can benefit from the use of smart objects through custom automations supporting everyday activities. The demands for such AAL automations are very personalized, while the requirements may also change on a regular basis due

to seasons, social life, health conditions or the progress of ageing. We discuss the case of Tina, being 72, lives alone, has diabetes and is overweight.

Tina should carry out specific tasks in her daily life due to diabetes, including daily workout, medical therapy and medical examinations (e.g. track insulin glucose), check her weight and have a strict diet. Furthermore, she has to take bath on a regular basis in order to prevent possible infections. Tina's tasks are split in three parts of the day as depicted in Figure 1.4 (right), while the people she communicates with are family, nurse which gets blood samples once a week, nutritionist and doctors, as depicted in Figure 1.4 (left, top). Tina wakes up every morning at 7 o'clock; using an alarm clock in order to get the required pill for her therapy. She has to track her weight, track glucose in her blood, get breakfast with specific ingredients and take her morning bath. However, Tina's morning tasks will be different every Monday for the next two months during which a nurse will be coming to her home once a week. The nurse will take blood samples, which require from Tina not to have received any medication or breakfast on that particular morning. All these changing tasks are difficult to follow for an elderly patient either because they may forget to do some of the tasks (e.g. forget to check weight, remember not to get a pill on the day of blood sampling etc.) or forget to abide by the rules of a strict diet.

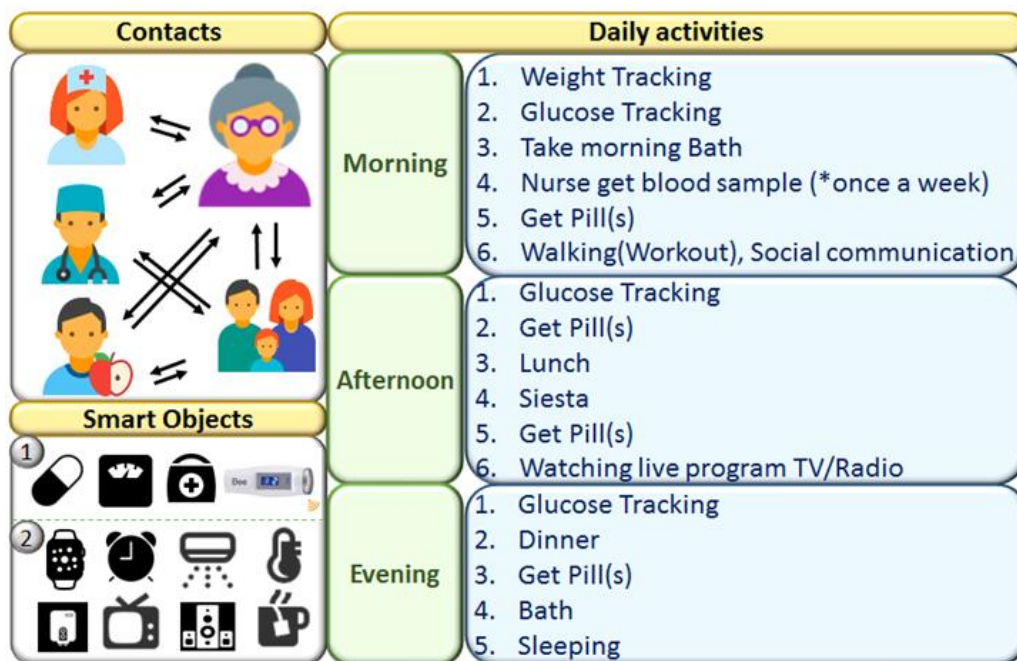


Figure 1.4. Tina's daily activities, contacts and smart objects.

Thanks to IoT, Tina is able to use smart objects such as Bee+ [25] tracking glucose, smart scale tracking weight, and smart heater preparing water for bath. In particular, Bee+ is able to track glucose and send data to the doctor directly for further analysis and alert to do this task at a specific time daily. However, Bee+ does not provide functionality to remind her to track glucose after activities such as tracking weight or finishing the bath. Such customized automations require ways to introduce extra algorithmic logic across smart objects.

In Figure 1.5, such extra automations are shown to remind and guide Tina for all morning tasks, like track weight and glucose levels, get pills in time, prepare heated water for the morning bath, and regulate home temperature wake up. Furthermore, automations which are depicted in Figure 5 care to remind Tina not to receive any medication or breakfast every Monday morning before doing her blood tests.

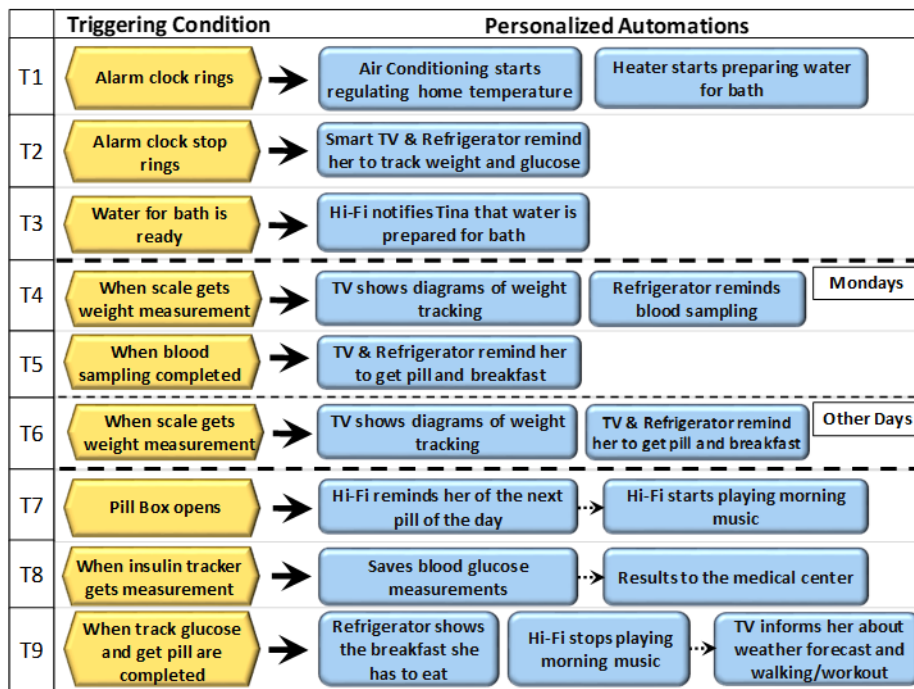


Figure 1.5. Tina's morning automations.

Moreover, using smart devices for automations, Tina is able to be benefited on alternative cases of her mobility requirements. Tina has intense social life and has to move around the city to visit her contacts on a daily basis. In particular, Tina visits her son's family twice a week, goes to the gym three times a week, goes to the nutritionist once a week and she visits her friend Alice twice a month. In order to go to all these

places, it is required from Tina to follow different routes as depicted in Figure 1.6 while on longer distances, she is required to take pill(s) and/or track blood glucose. In addition, Tina takes her emergency bag and personal belongings with her during her journeys. Furthermore, Tina has difficulty using the means of transport due to vision issues which arose two months ago. Tina’s son, Nick is anxious that his mother may neglect her health by forgetting to take her medication or to track glucose levels during travelling. She may also forget her emergency equipment bag in the means of transport. Furthermore, he worries that his mother may be confused and get lost if she gets the wrong bus or gets off the train/metro at the wrong stop. Also, Nick knows how important it is for his mother to continue her social life as earlier.

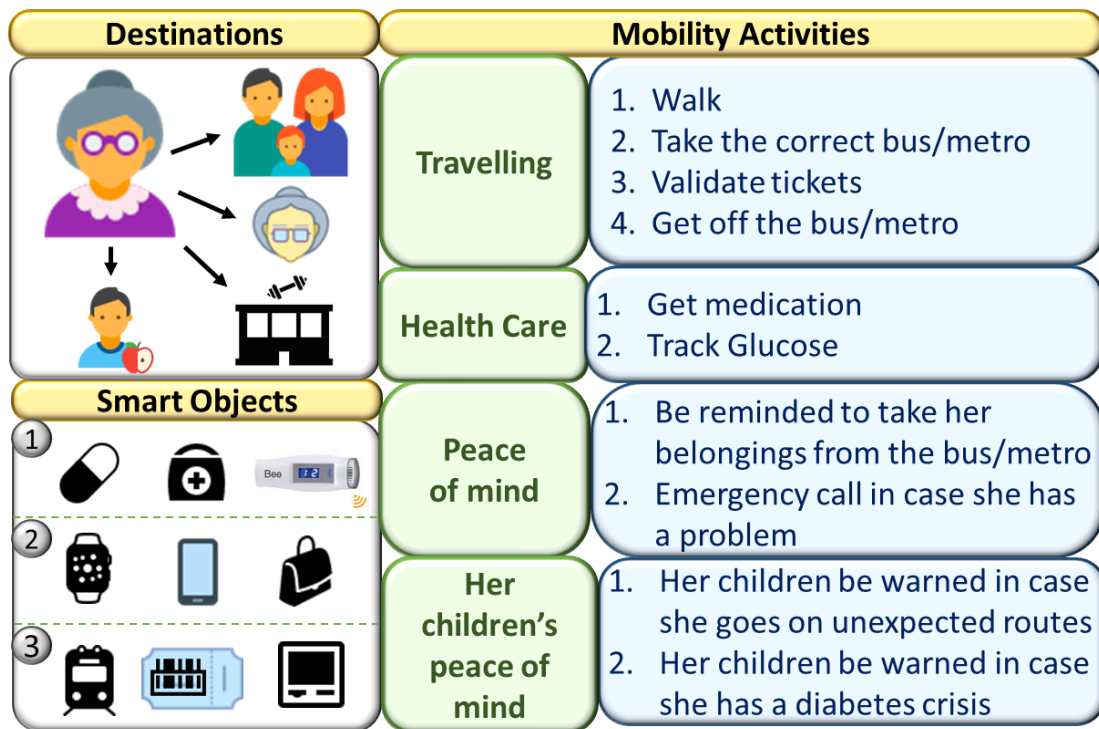


Figure 1.6. Tina’s daily activities, destinations and smart objects.

Thanks to IoT, Tina is able to use smart objects such as Bee+, e-ticket and smart metro assistant. In particular, Bee+ is able to track glucose and send data directly to the doctor for further analysis and alert the user to take a sample at a specific time. However, Bee+ does not provide functionality to remind her to track glucose after different activities such as getting on the train. Furthermore, e-ticket works with NFC technology which requires from her to go close to the ticket reader. But it may be difficult for her to find the device to validate her ticket due to visual impairments.

Such customized automations require ways to introduce extra algorithmic logic across smart objects.

	Triggering Condition	Personalized Automations
T1	At 8 o'clock	Smart watch reminds her to start walking to go to the metro Smart phone reminds her of the things she needs to take with her
T2	When door opens	Smart bag checks if Tina got her belongings. If not smart watch warns her
T3	3minutes before the metro carriage door opens	Smart watch notifies her that she needs to go close to the carriage
T4	Tina gets on the metro	Smart phone validates e-ticket using smart metro assistant
T5	At 9:30	Smart watch & mobile remind her to track glucose
T6	Before metro reaches the appropriate stop	Smart watch & mobile warn to get off Smart watch & mobile Remind her to get her belongings
T7	Tina reaches the bus station	Smart watch notifies her about the ETA of her bus
T8	3 minutes before the bus starts	Smart watch notifies her to get on the bus
T9	At 10 o'clock	Smart watch & mobile notify that she needs to take her pill
T 10	Before the bus reaches the appropriate stop	Smart watch & mobile warn to get off Smart watch & mobile Remind her to get her belongings

Figure 1.7. Tina's transportation automations to visit Alice.

Using smart objects which exist on the market (see in Figure 1.6, label 1 and 2) and smart objects which are provided by the metro/train (see Figure 1.6, label 3) such as e-tickets, route assistant, ticket reader, Nick could develop custom applications for his mother's necessities. In particular, Tina needs one application for each journey due to different requirements per travel (e.g. different means of transport, get medical therapy or not etc.). These applications require different automations that are

categorized as shown in Figure 1.6, i.e. the first category is for Tina’s travelling, the second is for Tina’s health care, the third is for the Tina’s peace of mind and the final category is for her children’s peace of mind.

We choose to discuss about the required automations for the journey from Tina’s home to Alice’s (see Figure 1.7) because of the longest route. In particular, this route demands from Tina to walk to the metro station, get the metro at 9:00 o’clock, get off at a particular stop, get the bus and get off at a stop near Alice’s home. Also, after Tina has got on the metro, she has to take her medication at 9:30. Moreover, during traveling by bus Tina has to track her blood glucose.

In addition, for Tina’s and her children peace of mind, Nick could has developed an extra application with automations as depicted in Figure 1.8.

	Triggering Condition	Personalized Automations
T1	Tina forgets to take a pill or track glucose in time	Nick’s smart watch warns him that his mother didn’t take a pill or didn’t track glucose
T2	Tina’s wearable detects a diabetes crisis	Nick’s smart watch warns him of the diabetes crisis Tina’s doctor is notified for the diabetes crisis
T3	Tina press emergency button	Nick’s smart watch warns him of danger of his mother’s life Police station is warned for danger of Tina’s life.
T4	Tina gets on unexpected metro carriage	Nick’s smart watch warns him that his mother has got on wrong metro carriage

Figure 1.8. (T1) Tina's peace of mind automation; (T2, T3, T4) her children's peace of mind automations.

The objective of our work is to provide end-users with the necessary tools enabling them to easily and quickly craft, test and change the automations they desire. Now, the latter is not an easy task as it implies end-users to directly manipulate smart objects in a developer perspective, ranging from parameterizing and linking together, to actually programming the control and coordination of a set of smart objects. In this context, we target to address challenges of communicating, managing, programming,

testing and running smart automations in the IoT context by developing all required end-user programming facilities.

1.3 Technical Approach and Contributions

Our approach aims to develop an open IDE for visual end-user programming languages by not limiting it on a specific application domain and aiming on extendibility of new visual programming features (see Figure 1.9). In order to cope with these requirements, the IDE focuses on two directions, the domain application adaptability of the IDE and the extendibility of the IDE.

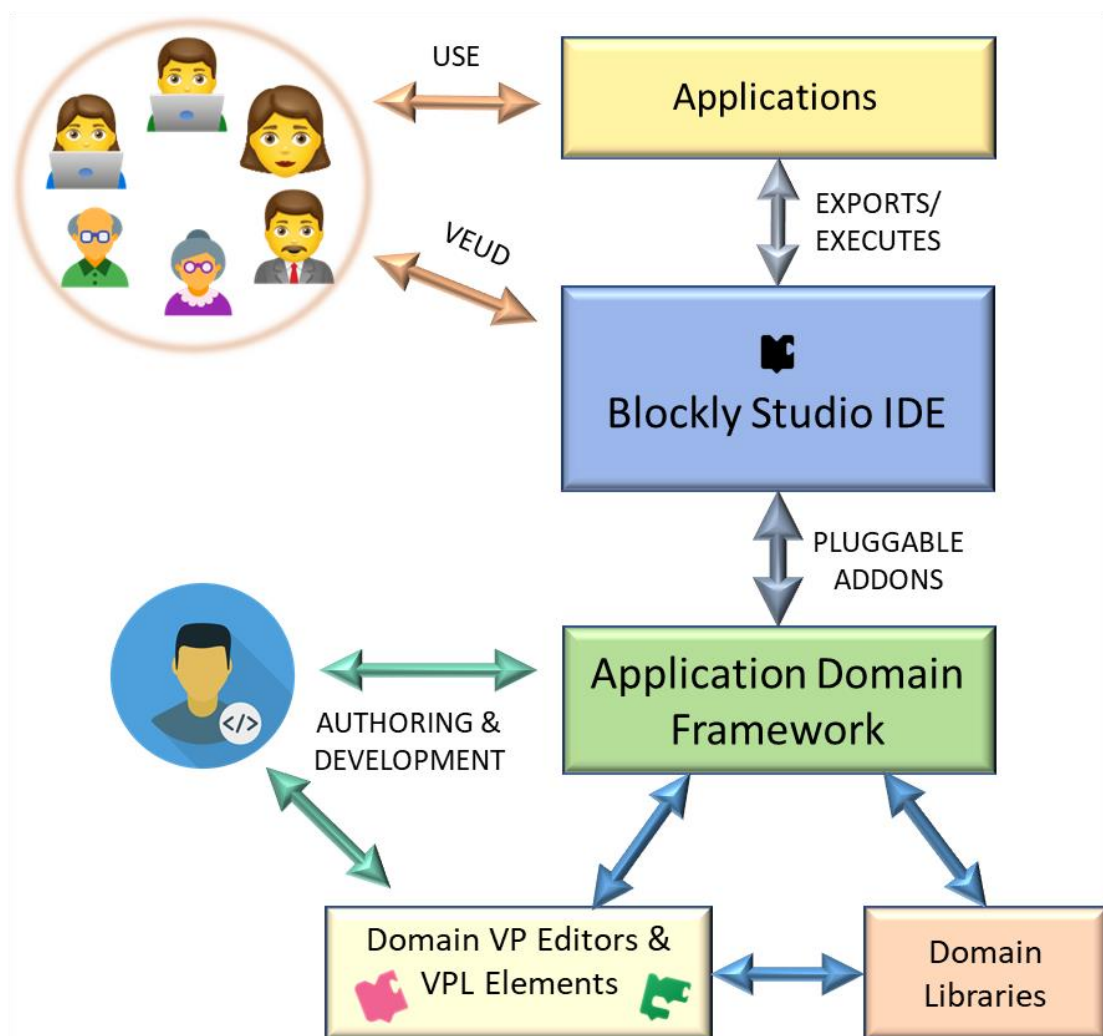


Figure 1.9. The notion of professional developers (i.e., application domain authors) and end-user developers & users (i.e. novices, non-programmers) in the visual programming IDE.

In particular, based on the requirements of visual programming in application domain, the developers (i.e., domain application authors) are able to define new application

domain(s) for visual programming workspaces by authoring the required visual programming language elements, adapting the core components of the IDE through meta-data definitions and developing domain specific components as plugins if required. They are also able to reuse all components and definitions of other application domains. Moreover, they are able to contribute by developing adaptable tools for visual programming that will be used by the application domains. Furthermore, following our approach, the developers are able to use all provided visual programming features of the IDE for each of the authored application domains.

Since the visual programming IDE for VPLs has to be extendable, our approach is following component-based architecture. The core of the IDE includes the component's communication which is based on Blackboard pattern and the Shell Component which is responsible for the user-interface management. Each component is independent from others and communicates through the provided communication of the IDE. We present our IDE approach and extension mechanism for new application domains in Chapter 3. Using this approach, the developers are able to add new visual end-user programming features through the development of new IDE components.

Overall, our work of this thesis is categorized in three main directions. Firstly, the development of a visual programming IDE with a full-scale end-user development toolset. Secondly, we focus on embracing visual programming domain variations as pluggable domain frameworks in the IDE. The last direction of our work is the development of a pluggable application domain framework for smart automations in the internet of Things. Based on these three directions, the contributions of this PhD thesis are following.

- We provide an extendable visual programming IDE that allows programmers to extend it in two perspectives. The first perspective is that of new visual end-user programming features and the second is that of the new application domains. Our approach could be used for existing (e.g., smart automation in the Internet of Things, mobile applications, etc.) and emerging application domains and technologies.
- We introduce the notion of application domain authors which is the role that developers are able to play in order to build new application domains as

frameworks based on the arisen technologies and requirements for visual programming workspaces.

- We develop a full-scale collaborative editing approach that sorts out the process by introducing peer roles and project element privileges. Additionally, our proposed approach supports multiple collaboration models (i.e., Pair Programming in one or more groups, teaching and learning purposes, working in small teams, etc.) by regulating the settings are provided in order to configure the collaboration process when it starts.
- We facilitate debugging and testing for novices by providing collaborative debugging process that can be used for personal and collaborative EUD projects. The collaboration proposed approach guarantees the preservation of the project's visual code by isolating it, creating a local replica for each one of the collaboration members. In this context, the users are able to create correction suggestions per project element. Those correction suggestions are shared among the participants. During the debugging session, one user at a time is able to handle the debugger instructions (i.e., master of the debug session). However, the rest of the members (i.e., observers) are able to navigate the visual code to acquire information independently of other members browsing, without interfering with the experience of any collaboration member.
- We propose an alternative model of collaborative debugging in order to contribute to teaching and learning in the context of debugging and programming. Particularly, the tool can be used by teachers to demonstrate the debugging process to students in real-time. The students are able to perceive the flow of a program and learn the process of debugging. Additionally, our approach introduces multiple *debugging rooms* in a session by enabling the students to live debug programs, individually or collaboratively while allowing the teachers to supervise all the debugging processes.
- We introduce conditional breakpoints for domain-specific visual programming language elements contributing in the debugging process.
- We propose code snippets for visual programming languages by developing infrastructure to manage and use them in the context of general purpose

(common loops, branches, etc.) and specific purposes for the application domains.

- We provide a full-scale management for the smart devices in the context of EUD including user actions to authenticate, organize, customize smart devices in order to enable isolation and handling of the numerous existing smart devices.
- We provide a full-scale visual programming workspace environment for personalized internet of things automations including conditional and scheduled tasks and choice of starts them automatically or manually during the project execution.
- We provide GUI for the runtime environment that cares for monitoring and interacting with smart automations, facilitating the end-user developers by removing the requirement to program user interfaces for their smart automations.
- We provide facilities in the context of testing and debugging the smart automations by developing infrastructure to enable the users to simulate the smart devices, the behavior of the smart devices, the date and the time that the automations will be executed.
- We address the issue of responding to the arising user questions about automations which caused during the execution of the constructed IoT applications.
- We demonstrate how visual programming IDEs can be used to address the highly personalized and fluid requirements of Ambient Assisted Living through custom personalized automations in the context of Internet of Things.

1.4 Outline

This thesis is organized as follows. Chapter 2 discusses the related work, focusing on visual programming workspaces and toolset support for end-user development. Chapter 3 presents the core system of the visual programming IDE, including the software architecture, the extension mechanism for application domain frameworks and the IDE's core components. Chapter 4 describes the visual programming editors presenting the types and hosting of the editors. In addition, we discuss the main visual programming editor which is incorporated in the IDE and basic features are supported

by editors. Moreover, we analyze the domain-specific VPL editors and elements and how their behavior is handled by our approach. Chapter 5 presents the authoring of application domain projects including the Project Manager's functionality, the application structure, the project elements, etc. Chapter 6 explores the runtime environment of the IDE, presenting how application domain projects are supported to be executed. Moreover, we present the I/O console of the visual programming IDE and how hosting user-interface of application domains at runtime is supported. Chapter 7 discusses the debugger of the visual programming IDE including full-scale block-level debugger for *Blockly* editor and appropriate features of debugging for novices. Chapter 8 presents our approach of collaborative visual programming which focuses on two directions, the collaborative editing and the collaborative debugging. Chapter 9 discusses the visual programming framework for IoT automations we have developed, including the smart object editor which manages the smart devices, the visual programming language elements for the behavior of smart objects, authoring of the application structure, the user-interface which is viewed at runtime and the simulator for the runtime which contributes the debugging process. Chapter 10 summarizes the key points of this thesis, draws key conclusions and discusses directions for future research.

Chapter 2

Related Work

“The greatest part of a writer's time is spent in reading, in order to write: a man will turn over half a library to make one book.”

- Samuel Johnson

Our work in this thesis is focused on three directions, the full-scale visual programming IDE that will not be limited on a specific application domain, the collaborative visual programming and the IoT automations through visual programming. The related work is organized in three areas: visual programming workspaces, extension mechanisms of IDEs, visual debuggers in the context of end-user development, collaborative programming and code snippets.

2.1 Visual Programming Workspaces

There are several visual programming approaches the past two decades. In this section we outline the most current or remarkable approaches.

2.1.1 Block-Based Languages

The most popular category of graphic artifacts for the visual programming languages are the *jigsaws*. This technique is based in the traditional *jigsaws* which all people has already experienced during their childhood and beyond. Each *jigsaw* has mapped with correspondent functionality of the visual programming language. Alternatively, this approach is called as block-based languages. There are several approaches of visual programming languages based on this technique. One of the most popular approaches is the *Scratch* [26]. It is a web-based application and online multimedia authoring tool that can be used by end-users to program their own interactive stories, games, animations and simulations. Additionally, Scratch gives the ability creations can be shared [27]. Inspiring from the work of *Scratch*, several research works have been developed such as *Phratch* [28], *Snap!* [29], etc.

Another approach that resembles *Scratch* is the *Blockly* which is a project of Google [30]. *Blockly* uses blocks that link together like a puzzle in order to make writing code easier. It can generate source code in JavaScript, Python and Dart [31]. Using *Blockly*, people learn about coding and logic of programming. In the same context, yet another approach is the App Inventor Blocks Editor which is specialized for the development of the logic of applications for devices running Android [32].

In the context of robotics, the *Lego Mindstorms* is another approach that uses blocks too [33]. *Lego Mindstorms* blocks' has been mapped in a higher level of functionality than *Blockly* and are specialized in the concept of robots that can be programmed by youngsters. This visual programming approach combines block-based programming with flow-based programming we discuss in the following paragraphs. An approach which resembles with the *MindStorms* is the *MODKit* [34]. *MODKit* product provides two versions of products, the first is related by micro controllers and the second is related by robotics. Moreover, an extension of *Scratch* is named as *mBlock* [35] and focuses on the end-user development of *Arduino* and robotics.

Yet another approach uses block parts is *Tynker* [10] and is targeted for children. Using *Tynker*, kids educated on programming web applications, building custom games, interfacing with hardware (e.g. program motors, LEDs, speakers etc.), drawing math art etc. In addition, students which use it, are able to learn fundamental programming concepts. This arises from the included ability coding visually or to write JavaScript source code and viewing in parallel the results of blocks and vice versa.

In addition, there is yet another authoring tool which is based on blocks [36]. This authoring tool focuses on the development of mobile services. The interesting with this approach is that provides to users the ability of the choice between two levels of programming. The first level of programming is for beginners and it is based on programming with questions. The second level is based on blocks. All above approaches using jigsaws for visual programming run either as a desktop application or as a web application. Additionally, in literature there is a framework called *Puzzle* which supports a visual based environment for opportunistically creating mobile applications [37].

In addition, there are two approaches which use their own technique for visual programming. First approach belongs to the Microsoft Research and is called TouchDevelop [38]. Using *TouchDevelop*, end-users can develop in their mobile devices [39]. Through *TouchDevelop*, applications can be created to access data, media, and sensors on smart phone, tablet or PC. End-Users could program without coding technology, but only by touch predefined statements and expressions to express logic. *Touchdevelop* uses tree view of steps that define windows, events, logic source code and in parallel provides advices in order to help user understand which have to be the current and next step of development. Microsoft retired the *Touch Develop* platform in June 2019. However, Microsoft continue research in the world of visual programming by introducing the *MakeCode* which focuses on two different directions, the game development of *Arcade* [40] and the *MakeCode* editors [41] which focus on the educational part through programming via blocks.

An alternative visual programming approach is called *Thyrd* [42]. *Thyrd* is a VPL that both data and code are stored in cells. *Thyrd* is an attempt to reduce the spreadsheet programming model to its minimal aspects by focusing on a small set of central concepts.

2.1.2 Flow-Based Languages

Another category of visual programming languages is the flow diagrams. There are icons with high level functionality as in the aforementioned VPLs based on jigsaws, but there is the concept of design flow diagrams. One of the approaches following the concept of design flow diagrams is the *Microsoft VPL* [43]. This VPL is specialized for building robotics applications. It can be used by both professional and non-professional developers. In this direction, a research work which is related with robotics and IoT is the research work of *VIPPLE* [44]. Another of robotics kit is the *ROBO Pro* [45]. This approach is specialized for robotics as toys for children. The robotic process control is based in the design of flow diagrams. In the concept of flow diagrams there are several approaches which are specialized on education such as *LabVIEW* [46], *Flowgorithm* [47], *LARP* [48], *Raptor* [49], *Visual Logic* [50] etc. All these languages are targeted on learning the concept of programming using designers in order to construct flow diagrams and execute them by correspondent interpreters.

Another approach for flow-based visual programming tools is *Rete* [51]. *Rete* is a JavaScript framework for visual programming by enabling the developers to build flow-based visual programming languages based on their requirements as the *Blockly* library accomplishes in case block-based visual programming languages.

2.1.3 Game Development Visual Programming Editors

Another category for visual programming is in the area of **game development editors**. Using this kind of authoring tools, end-users can design virtual worlds using predefined actors and objects. Such software tools are used for end-user development of custom games. One very well-known approach is *Kodu* [52] from the Microsoft research. *Kodu* provides numerous words and character artifacts that can be used in order to design a game. Furthermore, in this context, there are several approaches such as *Construct 2* [53] developed by *Scirra*, *GODOT* [54] developed by *OKAM Studio*, *GameSalad* [55], etc. Moreover, *AgentCubes* [56] is an educational programming language for children to create 3D and 2D online games and simulations.

2.1.4 Visual Programming Approaches for the IoT

HomeKit [57] is a product from *Apple* allowing control connected home accessories when compatible with *HomeKit*, and supports to a certain degree user-defined automations as combinations of accessory control actions. It is not an end-user programming system as such, and focuses mostly on smart home solutions with emphasis on advanced configurations. *Puzzle* [58] is a visual development system for custom automations with smart objects in IoT adopting the jigsaw metaphor.

Extending the *App-Inventor*, the developed blocks ‘*When*’ for sensors in the context of IoT [59][60]. Another approach is the *Smart Block* [61] which is based on *Blockly* library. The *Smart Block* is a visual block programming language for Smart-Things IoT application development. This approach is based on the *IoTa* calculus by creating new custom blocks for ECA rules, events, conditions, and actions. Additionally, *Node-Red* is a visual tool developed for wiring IoT centric applications. Moreover, *NETLab Toolkit* [63] is a flow-based programming approach in the context of the Internet of Things, providing a simple web interface to connect sensors, actuators, media, and networks associated with smart widgets.

2.1.5 Discussion

Existing visual programming workspaces are missing a full-scale end-user development toolset, while most of them are limited on specific application domains. In case of visual programming approaches for the IoT automations, existing approaches are limited on full-scale workspace environment for visual programming. Particularly, they are limited on programming expressiveness (i.e. provided conditional events are limited on basic expressions, while there is no approach that deals with time and calendar events). Additionally, there is no provided full-scale management of smart objects. Moreover, there are not adequate runtime environments for smart automations in the context of visual programming workspaces. Furthermore, there are not visual end-user development facilities that empowers the novices in order to test and debug their creations.

2.2 Extendable IDEs

To our knowledge there is no visual programming IDE which is extendable in the context of plugins or application domain frameworks enabling configurable and adaptable components in which the developers can develop and author their visual programming workspaces for specific application domains. However, most of classic IDEs are extendable in the context of developing plugins by providing infrastructure which enables the developers to incorporate their plugins such as *IntelliJ* [64], *Eclipse* [65], *Visual Studio Code* [66], etc.

In the context of programming frameworks which are able to be hosted as plugins in the IDEs there two approaches. *Eclipse IDE* supports hosting of application domain frameworks. A representative example of developed frameworks is the *Eclipse Modeling Framework (EMF)* [67] which a modeling framework and code generation facility for building tools and other applications based on a structured data model. Another IDE supports the domain-specific development framework is the *Sparrow IDE* [68]. An example is the *Game Maker 1.0*, being a domain-specific application development environment for cartoon-like games.

2.3 Tools for Debugging in End-User Programming

There are errors which arise during the development process and programmers have to resolve them by using debugging methodologies and strategies. Some of the

techniques used by professional programmers have been adapted in the end-user development tools. In the context of debuggers for end-user programming, research work attempts to elicit the way end-users think in the case of correcting an error [69][70]. Furthermore, another research study analyzes possible gender-based differences that may exist in the debugging strategies that end-users following in order to eliminate errors [71]. Moreover, a study [72] demonstrates that end-user debugging process is more efficient through pair collaboration. In addition, a study [73] investigates the debugging process that early childhood preservice teachers used during the process of block-based programming. This study reports the types of errors commonly made and how teachers debugged them.

In general, most of the approaches to end-user debugging are based on analyzing dependencies. There are several approaches that attempt to help users in the context of finding errors in spreadsheets. *ExceLint* [74] is an approach that uses static analysis for *Microsoft Excel*. *UCheck* [75] is an approach for spreadsheets that applies type checking in order to detect errors automatically. *UCheck* automatically infers the labels associated with cells and uses this information to carry out consistency checking of the formulas. Another approach uses a combination of spatial and semantic label analysis aiming to improve the rate of detected errors [76]. *StratCel* [77] is an Excel add-on that improves the process of finding errors. Participants using it, found twice as many bugs as participants using standard *Excel*, they fixed four times as many bugs, and all this in only a small fraction of the time. Another approach, *GoalDebug* [78] lets the end-user set the correct expected value of a cell, then generates a list with all possible solutions and suggests them to the end-user in order to choose the correct one.

In the case of debugging in visual programming workspaces, the first approach proposes an interrogative debugging interface for the *Alice* programming environment [79]. This approach [80] is a debugging paradigm in which end-users are able to ask why and why not questions about their program's run-time failures. In addition, there are approaches which adapt classic visual debuggers to debuggers for visual programming languages. *MakeCode* [81] incorporates a visual debugger which offers watcher view for variables, step-in, restart and slow-motion step execution actions. Starting the visual debugging process, *MakeCode's* view mode changes by turning

from editing to debugging mode (i.e. the visual code blocks turn in read-only mode and their view changes by adding ‘holes’ to enable adding breakpoints functionality etc.). Also, *Tynker* [10] has developed a debugger tool [82] which includes start, pause and resume actions, while stepping is allowed only by using breakpoints. The breakpoints are inserted in the visual code by adding specific breakpoint blocks that have been defined. When the program runs in a different mode (i.e. release mode), the breakpoint blocks remain present but are ignored.

In the context of *Blockly* Library, there are two approaches of debugging. The first one is a demo approach [83] for *Blockly* which provides only step execution of the program without functionality for watching variables, breakpoints etc. The second approach provides a full-scale visual debugging toolset for *Blockly*, working over blocks, supporting the full-range of debugging features [84].

2.3.1 Debugging and Testing for IoT automations

Moreover, in the context of end-user programming for the Internet of Things, *EUDebug* [85] is a system that enables end-users to debug trigger-action rules that are composed in a web-based application like *IFTTT* [86]. Additionally, *My IoT Puzzle* [88] is a debugging approach for *IF-THEN* rules through the jigsaw metaphor. Yet another approach that supports end-user debugging of trigger-action rules for IoT smart automations is [87], providing answers to why and why not questions considering the execution of the rules.

However, in the context of visual programming there are not approaches to test and debug the automations. In this context, we developed a simulator which is able to simulate smart object actions, simulate the behavior of smart object during the project execution. Additionally, simulates the time, date and enables the end-users to author tests of expected values of the smart object properties during the project execution. Using this, the end-users are able to debug and test their applications without communicating with real devices.

There is no existing tool that provides infrastructure in order to debug smart automations in the context of visual end-user programming. However, there are research approaches and experiences in case of professional developers. Particularly, debugging IoT control system correctness for building automation experience is

presented on [89]. Additionally, a framework for debugging IoT wireless applications [90] has been developed.

Additionally, simulators have been developed for debugging purposes in case of processors such as *Simulics Platform Simulator* [91]. Additionally, there is approach of a versatile emulator for the identification of vulnerabilities of IoT devices [92]. Moreover, an emulator has been developed in the context of debugging service programs in Ad Hoc networks [93].

2.4 Collaborative Programming Workspaces

One of the cases that collaborative debugging is notably useful is the collaborative programming of applications. There are several collaborative programming approaches for software integrated development tools and end-user development tools. Collaborative spreadsheets with concurrent cursors (i.e. one cursor per member), such as *Google Docs* [94], and *Office Online* [95] constitute one of the most popular approaches.

2.4.1 Collaboration in Text-Based Programming

In the case of software developers, the collaboration process is mainly based on version control systems such as *Git* [96] and *SVN* [97]. The developers work locally on different replicas of the project and merge their changes in the repository. However, during the development process, conflicts may appear and the programmers have to resolve them. In addition, each programmer has to set up the workspace in order to participate in the collaborative project. Moreover, visualization tools have been developed for the history of changes and handling them e.g. *Bellevue* [98] as an IDE extension and *Sourcetree* [99] as an independent software tool.

Furthermore, in the case of text-based programming, real-time collaboration features have been developed for IDEs. As previously mentioned, Visual Studio and Visual Studio Code enable developers to collaborate in real-time through Visual Studio Live. Using this tool, the developers have concurrent cursors and are able to edit the project's source code in parallel. *Collabode* [100] is a web-based IDE for Java that supports real-time collaborative editing through concurrent editors, isolating the error report only for their own changes. Moreover, *Codiad* [101] is another web-based IDE that supports real-time collaborative editing via concurrent cursors. *Jimbo* [102] is a

collaborative IDE that attempts to provide better collaboration and communication between designers and developers.

Also, *Saros* [103] is an open-source plugin-in for IDEs which offers distributed collaborative editing and pair programming. This plugin has been incorporated into Eclipse IDE and has recently been added to IntelliJ. In addition, another plug-in that has been developed for Eclipse is *Ripple* [104]. This plug-in enables the students to collaborate for educational purposes, incorporating a chat software tool for communication.

Furthermore, plugins for remote collaboration have been developed for text and source code editors (i.e., *RemoteCollab* [105] for *SublimeText* [106] and *Teletype* [107] for Atom [108]). Moreover, *Codeshare* [109] is an online code editor that is used for sharing code in real-time with developers and incorporates a video call software for communication purposes.

2.4.2 Collaboration in Visual Programming

In the case of visual programming, full-scale collaboration facilities are missing. However, approaches of collaboration for visual programming tools have appeared. An extension of *App Inventor* [110] that supports collaboration using concurrent cursors has been developed. Additionally, there is an approach for collaboration in Scratch that includes a shared stage screen in which each child develops one animated character and then merges it with the other animated characters in the shared stage of the application [111]. An approach of co-located collaborative block-based programming [112] has been developed for exploring block-based programming in a cross-device environment consisting of digital tabletops, mobile tablets and laptops. Furthermore, an approach in *TouchDevelop* [113] focuses on a merge algorithm which is conflict free, thanks to reasoning on changes at the level of *AST* [114]. Moreover, extending Alice Framework, there is work that enables interaction and collaboration among students [115].

2.5 Collaborative Debugging

In the case of developers, there are approaches to integrated development environments (IDEs) for collaborative debugging. *Visual Studio* and *Visual Studio Code* enable developers to collaborate in real-time by using *Visual Studio Live* [116].

In this context, *Visual Studio Live* enables collaborative debugging features by communicating with the debuggers that are provided by Visual Studio. IntelliJ with its plugin, *Code With Me* [117], is another IDE that has recently released collaborative programming features. In addition, *CloudStudio* [118] is a web-based IDE that supports collaborative software development on the web [119].

2.6 Teaching and Learning Tools for Debugging

Debugging is one of the most important tasks of the programming process. However, it is also a challenging task from which novice programmers can learn. In this context, there are research studies that attempt to teach the debugging process and improve the debugging skills of novices. There are several approaches which aim to facilitate them by using game-based applications. *RoboBUG* [120] aims to help students learn effective debugging techniques by playing a puzzle-type game, focusing on students who are learning to debug for the first time. *G4D* [121] is another approach which aims to teach debugging to novice programmers through interactive games. *Laddebug* [122] is an online software tool that aims to help novice programmers to improve their debugging skills. Using *Laddebug*, students follow a structured debugging process to find and fix errors in predefined exercises. Furthermore, *Gidget* [123] is an online debugging game for learning.

Moreover, a study [124] proposes a teaching model for learning debugging by designing worksheets to guide students on how to apply debugging strategies in order to find errors and correct them. *ViDA* [125] is a virtual debugging advisor that supports students' learning. *CMeRun* [126] is a software tool that enables the user to see each statement in a program during execution. *Backstop* [127] is a software tool that provides extra debugging features and attempts to be user-friendly in order to facilitate novices to understand run-time errors and correct them.

Furthermore, *DESUS* [128] is a tool that aims to support beginners of programming by providing them guide for tracing which benefit them to better understand the behavior of their programs. Additionally, *LondonTube* [129] is a visual programming language that blends dataflow and actor-based programming paradigms. An IDE plugin [130] aims to show where in the code the computation breaks down and help the programmer to understand why the code is not working.

2.7 Code Snippets

Code snippets are templates that make easier to enter repeating code patterns, such as loops or conditional-statements. In the context of visual end-user programming there are not approaches. However, in case of software development there are several approaches in different languages and IDEs. For example, in Visual Studio Code there is support of code snippets that appears in IntelliSense [131]. *Wing Python IDE* supports code snippets too [132]. Moreover, IntelliJ supports code snippets mechanism through plugin which is called *TagMyCode* [134]. In addition, there is study of providing better code snippets by exploring how code snippet recall differs with programming experience [133].

Chapter 3

CORE SYSTEM

“The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.”

- Albert Einstein

Using the *Blockly* library, we have developed a full-scale IDE for visual programming languages (VPLs) on the top of it. Our approach is focusing on an open IDE for Visual Programming Languages by not limiting it on a specific application domain and aiming on extendibility of new end-user visual programming features. In this chapter, we present the IDE’s core system; outlining the software architecture, the functionality for extending the IDE’s application domains, the communication with third party applications and the core components of the system.

3.1 Software Architecture

Blockly Studio IDE is a web-based IDE, including login system where someone signs up for the IDE using their credentials, namely their email and password. The projects are retrieved based on the account privileges. The projects are saved, shared and loaded by the back-end of the *Blockly Studio*; written in Node.js and its MongoDB data base. The backbone of the IDE follows a component-based infrastructure enabling components to be added or removed via a centralized components registry. Components can be activated or deactivated on-the-fly while the IDE is running. Each component is independent and communicates with the IDE via an extended custom version of the Blackboard pattern that has been developed as depicted in Figure 3.1.

For each of the component plugins is required to export which is the functionality is provided and which is the required functionality in order to be hosted in the IDE. In addition, each component has to define which are the messages (i.e. signals) that will be sent potentially during the execution, as well as the messages that will be listened. Defining this information, the system validates and warns in case something goes wrong with the communication among the registered components of the IDE. These

validity checks are applied during the build-time of the IDE and concern the static dependency analysis of the components' communication. However, there are cases in which the components' dependencies are changed dynamically during the run-time. As a result, there are components which are not able to define total messages will be exchanged and the whole exchanged functionality that will be required and exported during the execution. In this case, the IDE enables a component to define that will be included communication by exception.

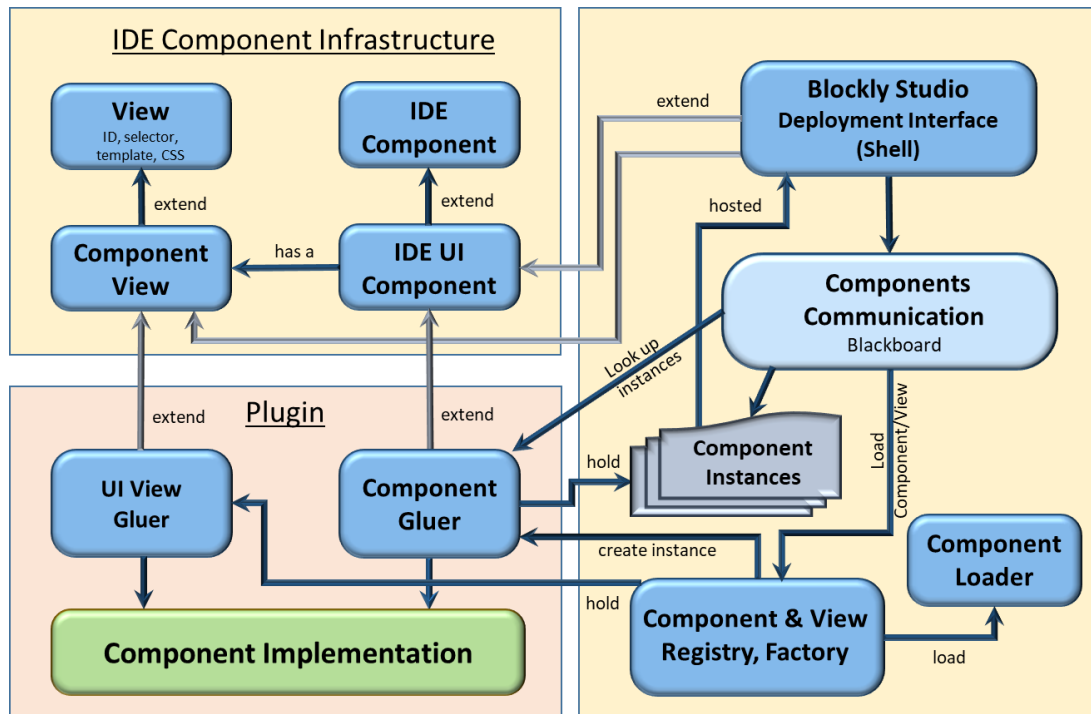


Figure 3.1. The component-based infrastructure of Blockly Studio; IDE's component infrastructure for the UI view and the component functionality is required (top-left).

The infrastructure of each IDE Component (see top-left of the Figure 3.1) is based on two main parts: The *Component Base Class* and the *View*. The *View* handles the user interfaces that are hosted in the system (i.e. rendering HTML from *Lodash* templates, applying the style from the defined CSS and attaching the events). Moreover, the *View* handles everything related to attach and detach the events that are declared for the template on render and on close action respectively. The IDE Component Base Class cares about the export and import of the functionality and the messages that will be exchanged between the IDE and the component.

3.1.1 Shell

The IDE's core component is the *Shell* component that registers in the system and then undertakes the hosting of the rest IDE UI components. It consists of three user-interface parts, the menu toolbar, the workspace toolbar and the main action area of the IDE. In this context, UI components could define their menu items, their area for the tool items (available when components are active) and their configurable settings for allowing adaptation to the end-user's needs. Moreover, in the main action area one or more UI components are hosted depending on the circumstances. Starting the IDE, the *Shell* component installs the "Menu toolbar" component. Afterwards, the "Start Page" component is initiated in the main action area by which users are able to browse and handle their projects (see section 3.3). In the following sub sections, we analyze each of these parts of the IDE.

3.1.1.1 Menu Toolbar

Each component that is registered to the IDE requires declaring menu options. These options will export functionality that will be available to the users. In this context, the top user-interface part of the IDE constitutes the menu toolbar which is registered as a UI component of the IDE. The components have the access control of the menu options of the toolbar by using two methods.

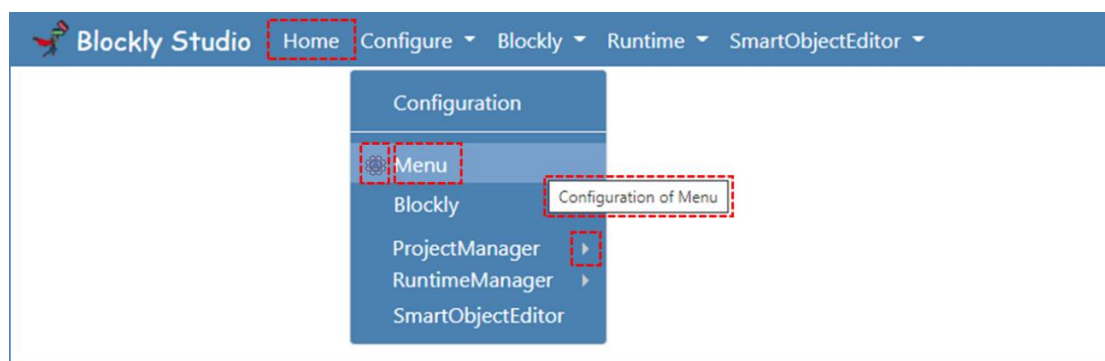


Figure 3.2. IDE's menu toolbar including the logo of the IDE and menu items which are declared by the registered components.

The first method deals with the static definition of the menu items that would like to be inserted on loading time of the IDE (see Figure 3.2). In particular, each component can include a *JSON* file that will define which are the sub menus, menu options and separators would like to be presented on the IDE starting it. In this context, *JSON* file

includes definition of the location (i.e. path) in the menu toolbar for each sub menu and menu item, their labels, their icon images (optional) and their tooltips (optional).

The second method relates to the dynamic editing of the defined menu items, separators by the components. This method is provided by the functionality which is exported by the menu toolbar as an IDE component. Using the exported API, the components could enable, disable, add, edit, remove menu items in which they are the owners based on their functionality.

3.1.1.2 Workspace Toolbar

Moreover, each component that is registered on the IDE, requires to declare their tool items which will be available in the end-user development time. In this context, the workspace toolbar is initiated below the menu toolbar when the visual programming workspace area is initiated. Thereafter, each component which is instantiated on the workspace, defines their tool items by including the icon, the tooltip and the function handler that will be fired on click the tool item. In addition, the components could define the order which tool items will be added among the components. This order number works similar to z-index in CSS, which means that in case there are tool item groups defined by components with same order number, first request gets previous location versus the next request. Additionally, the API enables functionality to handle the tool items by providing actions add new items, edit, remove, disable etc. Closing a component during the end-user development process, the workspace toolbar component is notified in order to detach the event handlers of the tool items and remove them from the toolbar view.

3.1.2 Configuration Management

The IDE provides configuration management in order to enable the end-user developers to personalize their workspace based on their preferences. The configuration preferences are separated into two categories. The first category is the global view preferences of the IDE (e.g., dark/white/colorful view mode, font preferences, etc.) and the second category which includes the configuration parts of specific components. In the first category, the components that are registered on the IDE have to support the style for each of the view modes by defining which UI part(s) of the components correspond to the view choices. Using these declarations, the

configuration management handles the whole IDE presentation (i.e. all IDE UI components).

In the second category, the configuration management provides functionality of configuration control for the personalization of each one of the built-in components and the pluggable components that may be added later. For each one of the registered components, the configuration management inserts a menu option which enables the user to choose in order to configure them. When one of these menu options are chosen, a dialogue opens which consists of three parts: the title which defines the component that will be configured, the configuration contents and the actions (i.e. save, reset, cancel).



Figure 3.3. Dialogue of the Configuration Management to configurate the dialogue parts of itself.

In the context of the component's configuration contents, each component defines the configuration parts that are supported by them by listing them in a JSON file. Each of the list items includes the property title and the property value of the configuration. In this context, the configuration management supports specific UI types for the values of the configuration parts. Based on these types, the configuration management generates a dialogue which includes the user-interface of the configuration for each of the components. Moreover, the configuration management undertakes to save or reset the preferences of the user. Additionally, the configuration management as component of the IDE defines its configuration including the background color, the font of the title and the contents of the configurations dialogue as depicted in Figure 3.3. In the following sub-sections, we analyze the value UI types that are supported for the UI code generation of the component's configuration parts. Each of them inherits *Property View* which inherits the aforementioned *View* infrastructure. The *Property View* undertakes the functionality of collecting and retrieving the values of each property when a configuration dialogue closes and opens respectively.

3.1.2.1 Basic Property Views

The first category of values for the configuration properties are the basic property views, including number, color, percentage, text, text area, date, checkbox, image and file. These basic types are following the input *HTML* tag; introducing default values, placeholders, min. max, step values, etc. depending on the input type. Moreover, in case of images and files, there is an extra view for the values that are selected. In the case of a file, a link with the name is presented, while in the case of an image, a preview of the image is depicted. Using these property types, the developers are able to define the property values of basic configuration parts for their components.

In addition, for each of the basic property types, it is provided to define messages that will be presented in case user selects specific values for the properties. In particular, a list of set with a *Comparator* function and the warn message could be defined for each of the property value types. The *Comparator* function is defined as a handler in the "*onChange*" event of the property value. In case it evaluates to true, the message will be shown, otherwise the message will be hidden.

3.1.2.2 Select Property View

The second category of values for the configuration properties are the *ENUM* property view type or the select property view alternatively in the context of *HTML* tags. There are cases in which the users have to choose one option among a list of values (e.g., IDE view: *dark*, *white* or *colorful* mode). Furthermore, for a better organization of the options in a select property view, our approach supports grouping options. Moreover, options could be images instead of texts.

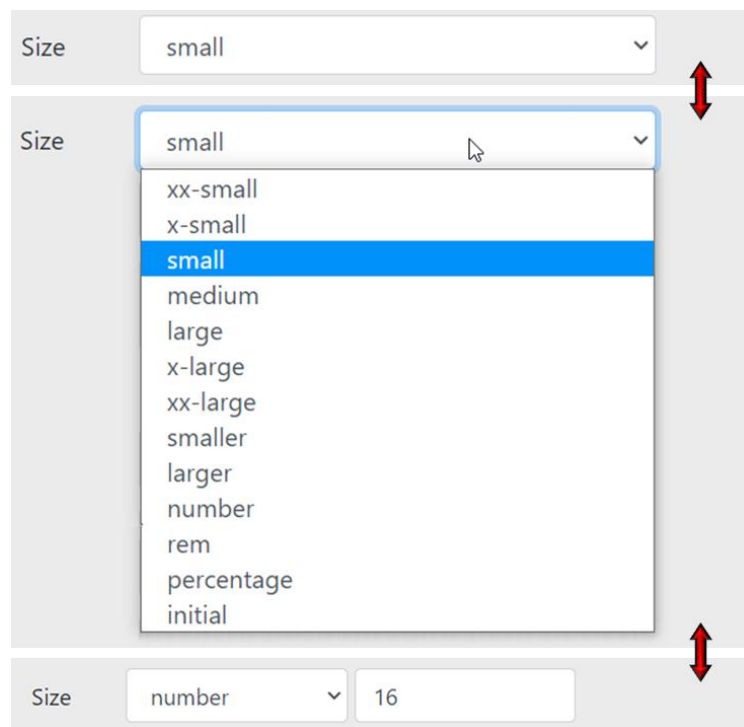


Figure 3.4. Dynamic extra number property value appears on selecting the option 'number' for the HTML font size select property value.

Additionally, we have introduced a dynamic select property view which includes extra property views on selection of specific values. For example, in case of the *CSS* font size there are options which include more property value such as the number (see Figure 3.4), rem and percentage. The extra property value type could be one of the aforementioned basic property views, select property view or dynamic select view as well. Finally, potential notification and warning messages could be defined to be presented in case of specific circumstances as in case of basic property views.

3.1.2.3 Aggregate Property View

The third category of values for the configuration properties is focusing on supporting the definition of more complicated types by grouping a list of property views. An aggregate property view includes the group title and the list of pairs property name, value. The property values could be basic property view, select property view or aggregate property view as well. The ingredients of the aggregate property view are contained on a UI box.

A common deployment of the aggregate property view that we have developed and introduced as an independent property view type is the “*Font Property View*” which supports the *HTML* font style including family, text color, size, weight and style as the list of the inner properties (see its use in Figure 3.3).

Based on the aforementioned property view types are supported and the features are included, the developers are able to define any property value for the configuration parts of the components.

3.1.3 Communication with Third-Party Applications

As we have showed earlier, *Blockly Studio* follows component-based architecture in which components communicate through an extended blackboard pattern including support for “*Function Requests*”, “*Function Responses*”, and exchange messaging through “*Signal Post*”, “*Signal Listen*” actions. In this context, the developers are able to add components as plugins in the IDE. However, this communication is limited on the components which are registered and running in the context of the IDE. There are components that could require to run in an independent context in order to prevent freeze the rest IDE UI functionality during their operations such as the IDE’s run-time environment. Additionally, independent third-party applications may interoperate with the IDE as plugins. Moreover, conflicts could be identified among the components in the context of *CSS* rules. In this context, we extended our approach in order to enable hosting of third-party applications that will run in different runtime context from the IDE. In this direction, we have extended the aforementioned communication of components in order to support communication among components of individual applications.

Third-party applications of a JavaScript application that runs in the same domain are hosted in *IFRAME* tags. The applications communicate by exchanging messages through the provided “*window.postMessage*” function [135]. Using pure exchanging messages approach, the provided functionality of components requires extra development in order to support listen and receive messages, coding and encoding in order to accomplish the requests. However, following this approach, each one of the provided parts has to deal with this requirement in the side of IDE and in the side of the third part applications. Moreover, in case communication between third-party applications which are hosted by the IDE is required, this could not be carried out. In this context, we have built an extra layer for the communication among third-party applications and the IDE.

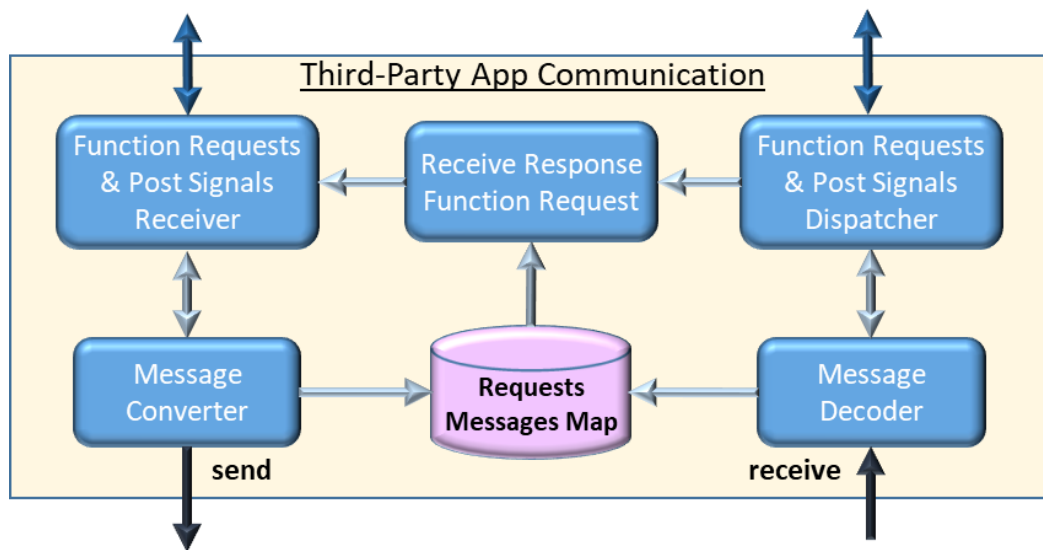


Figure 3.5. Extension layer for the Blockly Studio communication with third-party applications.

Extending the components communication, we have added an extra optional field for the function requests which included the third-party application name in case the request is not addressed for a local IDE component. In this case, the components communication forwards the function request to the third-party applications communication infrastructure. This infrastructure (see Figure 3.5) handles the communication of components by converting requests for a component (i.e. function request/response, post/listen signal) to messages by pinning unique ID and inserting appropriate function callback by using this unique ID to the requests’ communication map. In case, there is response of the function request, this function callback is

utilized in order to communicate with the respective component which sent the function request. Moreover, the infrastructure undertakes message exchanging by decoding received messages from other applications in order to apply the requests for components or handle responses from other applications.

The aforementioned infrastructure is able to be used in both sides (third-party application and the IDE). Using it in the third-party applications, each application is responsible for the development of the dispatcher that will handle their communication locally for their components. Additionally, the infrastructure for communication of third applications supports the signals mechanism that is provided by the components of the IDE. In this context, the third-party applications have to undertake the definition of which signals are listened and their handlers, use the mechanism to post signals (if exist) to the IDE.

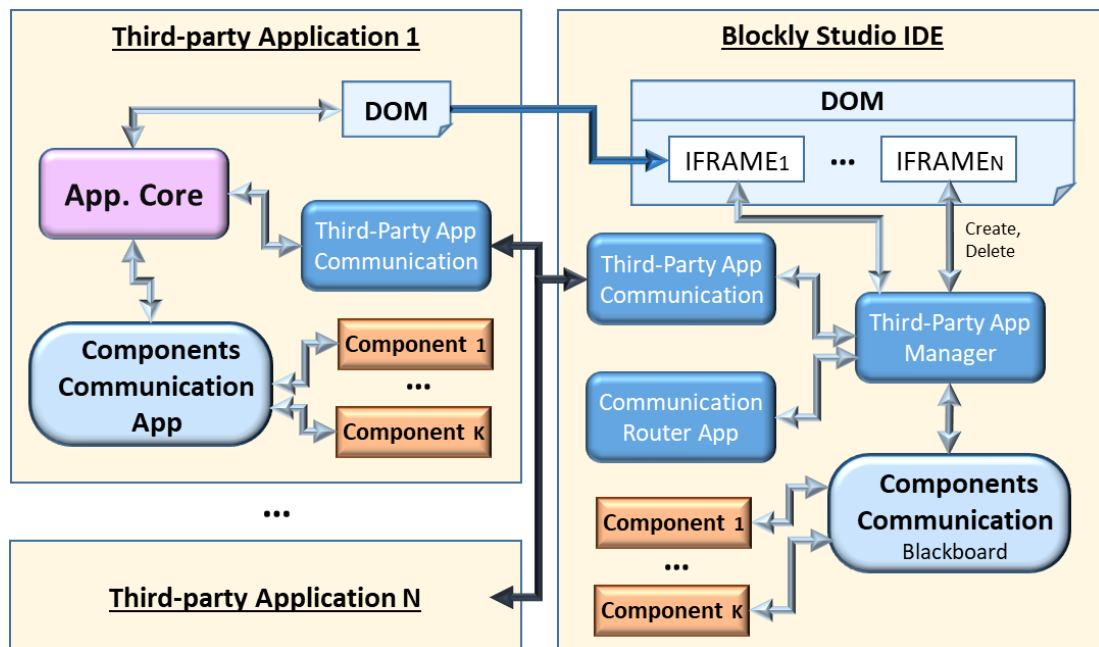


Figure 3.6. Communication among third-party applications and the Blockly Studio.

In addition, the IDE supports serving of the communication among third-party applications by playing the role of router in the function requests that are received and that are not addressed to the IDE. In particular, every message request is received by the IDE is decoded and routed either in the component communication of the IDE or it is forwarded to the respective third-party application by playing the root node of the communication among the applications as depicted in Figure 3.6.

Using our communication approach, the developers are able to extend the IDE's functionality by developing either new components and glue them as plugins (see bottom-left of the Figure 3.1) or new independent applications that will be injected in *IFRAME*. However, this is not adequate to support end-user development for different domains. The components of the existing visual programming workspaces are restricted on one specific application domain. In the following paragraph, we analyze our approach for the infrastructure of the IDE in order to set up and extend it developing new application domains.

3.1.4 Openness and Extensibility

As mentioned earlier, IDE is following component-based architecture by using an extended version of the *blackboard pattern* for the communication of components. Particularly, each component publishes the exported functionality by using precompiled customized annotation tags of the IDE that are developed by using decorators that are provided by *TypeScript*. All the defined functionality is able to be used by all installed IDE components. Moreover, each component defines the functionality that is required and has to be provided by the IDE from other components. During the compilation process, IDE collects defined functionality that is exported and required, then checks if required functionality is provided. Moreover, system checks their validity at runtime (i.e. asserts in case there is call request of functionality which is not defined).

All components communication for the IDE is handled by the *blackboard pattern*. In particular, the *blackboard pattern* has information about all the components provided and required functionality in the context of using it as a plugin. The blackboard pattern provides the functionality which is able to be used by the components in order to request a function (i.e. *ComponentsCommunication.functionRequest (destComponent: string, funcName: string, args: Array<any>)*). In this context, before apply the requests, starts with the validation checks of the communication. Moreover, the components are able to define which is the functionality that they are interested when happens from another components. In this case, the components which perform this functionality are responsible to post signals of the functionality with respective data (i.e. *ComponentsCommunication.postSignal (signal: string, data:*

any)). Components that are interested for the functionality of other components are responsible to define that they listen their signals by developing the respective functionality they would like to be executed (i.e. callback). In this case, when a signal is posted, the blackboard pattern is responsible to broadcast the signal to the interested components that have been registered in the IDE. An example of basic signals which is listened by different components is the *'PROJECT_ELEMENT_CREATED'*, *'PROJECT_ELEMENT_DELETED'* and *'PROJECT_ELEMENT_EDITED'* that are posted by the project manager when respective actions are performed by the end-users. These signals are used by the domains management component in order to care about the update process of the respective visual programming language elements, the collaboration component in order to care about the broadcast updates of the project manager to other peer members of the shared project.

The developed components communication mechanism is extended in order to notify the developers about the provided and required API for each one of the registered components. In particular, the developers are able to request the API for one or more components that are registered in the IDE (i.e. *ComponentsCommunication.consoleLog(compName: string)*). Using this functionality, the developers are able to be informed about the API for each one of the communication components and they are able to test their component's API. In this context, they are able to add new components or even replace existing IDE components with others just by implementing their functionality. Incorporating components to the IDE, developers have to program the glue code which defines the aforementioned requirements of the functionality and the exported API which is provided by the component which has to be written in Typescript in order to use the annotation compile time tags. Moreover, the IDE components which include user-interface has a selector of empty *DIV* that is hosted.

3.2 Extension Mechanism for Application Domain Frameworks

As we have already discussed, the visual programming workspaces are limited on one application domain. Additionally, new application domains are arising, existing application domain (e.g. games, learning, IoT, etc.) requirements for visual programming are fluid and third-party technologies are updated continually,

constantly changing requirements for developing new IDEs for visual programming languages. However, the process of developing an IDE for visual programming languages offering a full-scale end-user development toolset for each new application domain, is not trivial process and is extremely expensive.

In the case of the developers, the setup for an application domain in the IDE (i.e. installing and using third party libraries, editors, models etc.) is handled by them. For example, using user-interfaces for applications requires a GUI library and maybe the use of an *What You See Is What You Get (WYSIWYG)* editor. Developers are able to setup the environment of the IDE by installing appropriate libraries and tools based on their requirements. This task is not able to be done by the non-programmers.

Our approach enables the developers to setup the visual programming environments for application domains on the top of Blockly Studio. In this context, the developers will be able to author application domains based on the requirements of the end-user developers, consolidating domain third party libraries, visual programming editors, etc. (see Figure 3.7). Blockly Studio supports the development of application domain frameworks by providing the following features:

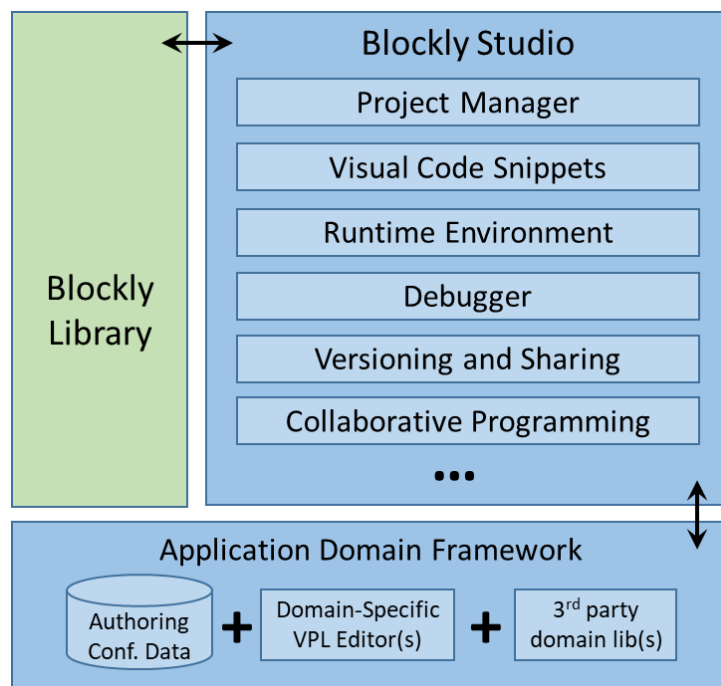


Figure 3.7. Making application domain-specific frameworks for visual programming on the top of Blockly Studio.

- As have already mentioned, extensibility mechanism to add new IDE components (e.g. Domain-Specific Visual Programming Language Editors, etc.).
- Adaptable and/or extendable core components of the IDE (e.g. Project Manager, Runtime environment etc.) that provide full-scale functionality and enable adapting them based on the application domains. Additionally, the ability to overwrite them by developing new components with relevant functionality.
- Authoring of the application domain project structure with automatic loading and handling of the project elements.
- A mechanism that handles the end-user development dependencies between the visual programming language elements automatically by cooperating with domain-specific editors (e.g. WYSIWYG editor) and the general-purpose visual programming editors (e.g. Blockly editor) in order to provide the appropriate visual programming elements (e.g. Blockly blocks) for the end-user development of the domain-specific elements (e.g. UI widgets).
- Reusing whole (or parts) of the developed application domain frameworks. For example, an application domain framework for mobile applications, including GUI library and WYSIWYG editor. The part of GUI can be reused for the application domain of smart automations in the IoT.

Authoring the application domain visual programming frameworks, developers will be benefited from the full-scale end-user development toolset, while they will be able to extend it by developing new features for end-user development. In the next chapters, we analyze the end-user development features and the infrastructure for the development of an application domain framework in Blockly Studio.

3.3 Browsing and Handling Projects of the Application Domains

Loading the *Blockly Studio IDE*, the start page presents the applications which have been developed by the user. This page is separated on three different parts (see in Figure 3.8). In the first part, the user chooses which of the application domain is interested by a drop-down list, while in the second the user views the information of the domain application (i.e. image and description). In third part of the start page, it is

presented the applications have been developed in the application, the user is given the option to create, open, delete, version, share an application, get a replica or join a shared application. On choose the application domain would like to view, the last two parts of the user interface are refreshed automatically. Moreover, when one application domain is added, edited or removed for the *Blockly Studio IDE*, the drop-down list is updated.

Moreover, the domain author is able to configure the dialogue in which the end-user developer uses in order to create new applications based on the requirements of the application domain. Using the automatic user-interface generation which developed for the configuration system (see section 3.1.2), the *Blockly Studio IDE* interprets the domain defined parts. Additionally, the domain author is able to configure the header title of the create project dialogue (see Figure 3.9). The extra defined information that the end-user developers fill-in is able to be used by the workspace components which are configurable based on the application domains such as the *Project Manager*, the

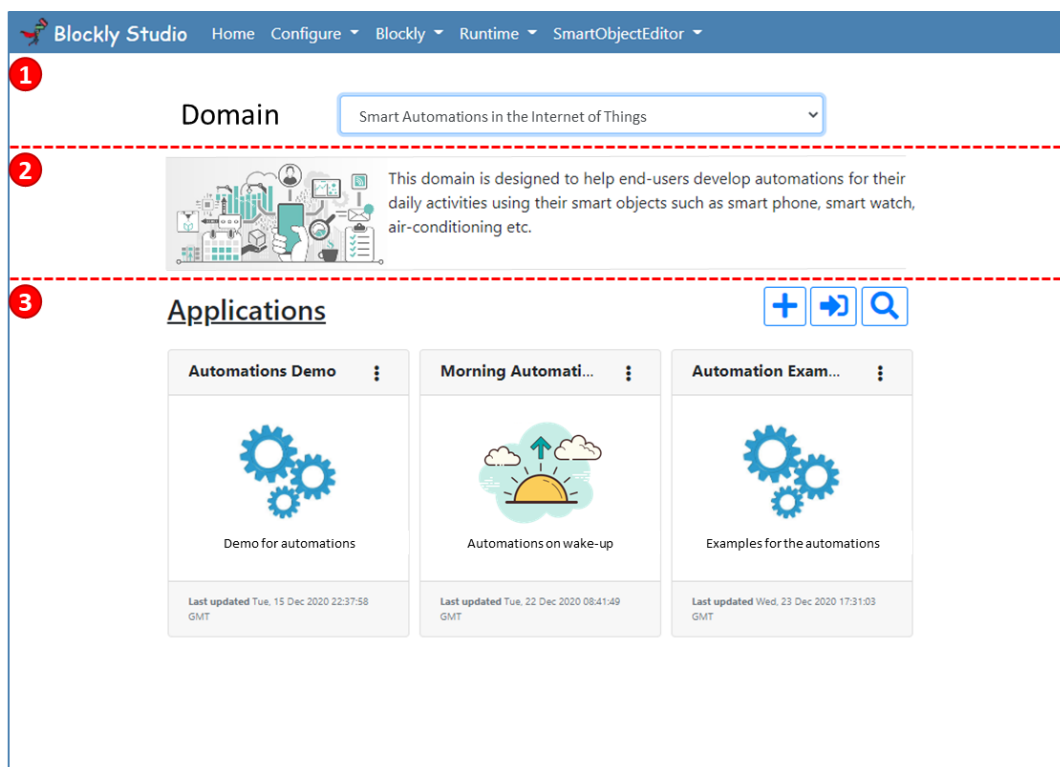


Figure 3.8. Having choose the application domain "Smart Automation in the Internet of Things" at the Start Page of the *Blockly Studio IDE*.

Runtime Environment, etc.

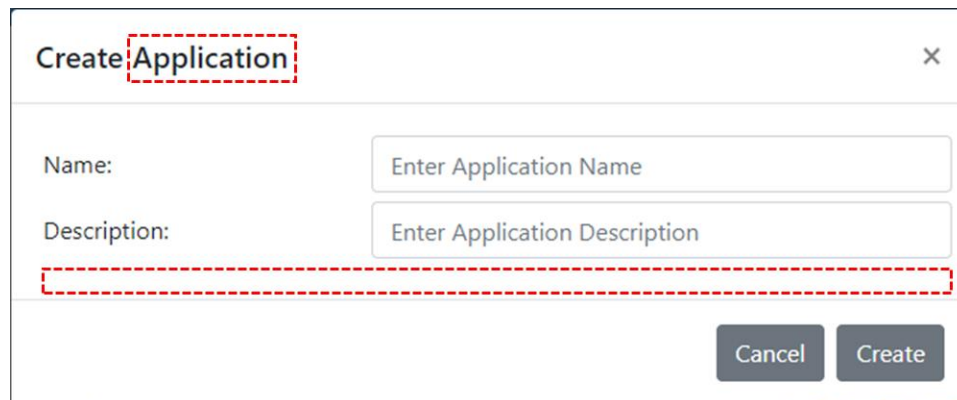


Figure 3.9. Configuring the dialogue to create new application based on specific application domain.

3.4 Sharing and Versioning

As discussed in previous section, the “*Start Page*” component give access in a set of actions. Three of these actions are addressed by the “*Start Page*” component itself, while other actions are requested to be addressed by other components. The first action is addressed is deletion of an application. The other two actions are the creation version of an application and sharing an application. We are going to describe in details these actions in the next two paragraphs.

End-users would be able to re-use existing applications developed by them or by other end-users and may be inspired by them. In addition, various applications need versioning so as to restore previously saved applications, make new ones or even use them interchangeably due to circumstances. Furthermore, each end-user develops several applications for different requirements. For example, the end-users could develop automations for other persons, but also, they could design automations for their personal requirements. Based on these requirements, our approach provides the end-user with the ability to define groups of applications. Also, the end-user is able to create new version(s) of the developed automations in order to apply the required changes and at the same time maintain previously developed version(s).

In addition, our approach supports sharing of applications. Upon starting the use of a shared application, a replica is created in the end-user’s environment. Sharing functionality is supported by several visual programming workspace approaches (e.g. *TouchDevelop* [136], *Scratch* [26], etc.). However, there are application domains

which require extra development steps. For example, in case of the visual programming applications involving smart objects require an extra step. In particular, the first development step of a new application for the end-users is to define which of the registered smart objects will be involved. As a result, the first (i.e. extra) development step for shared applications is the replacement of the smart objects which participate. In Chapter 9, we discuss the handling of loading shared applications in case of Internet of Things automations in section 9.1.3. Moreover, this issue is appeared by other application domains that are focusing on the end-user development of devices in general (e.g. mobile phones, Arduino, sensors, robotics, etc.).

Chapter 4

Editors

"To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support by extending your language into one that does support my programming methods, I don't need your shiny new languages."

-Robert Floyd

Basic weapon for developing source code by using programming languages constitutes the text editors. In this context, important features have been added to the advanced text editors and editors that are incorporated to IDEs, including invalid syntax highlighting, specific color in programming keywords, source code folding-unfolding, source code autocompletion, etc. These features do facilitate the professional programmers to develop their projects. However, critical skills and programming knowledge is required in order to develop applications. In case of non-programmers and programming learners, visual programming editors have been developed in order to encourage programming applications without syntax knowledge. In particular, coding by avoiding text-based programming and promoting visual-based programming which is categorized into icon-based languages, form-based languages, and diagram languages. In the context, of an IDE for visual programming languages, there are two different types of visual programming editors. The first category is first category is the general-purpose editors which supports basic programming operations and the second is the domain-specific editors which are specialized on application domains. In the following two sections, we analyze these two categories that are appearing in our IDE.

4.1 General-Purpose Visual Programming Editors

The first visual programming editor category is the general-purpose visual programming editors. These editors empower the end-user to develop basic programming expressions including variables, assignments, mathematic and logic

operations, branches, loops, data structures, function definitions and calls. The general-purpose visual programming languages could be used in the programming independent of the application domains. There are two basic approaches of general-purpose visual programming editors: the block-based editors or jigsaws (e.g., Blockly) and the diagram-based editors (e.g., *Flowgorithm*). Both of these approaches are able to support programming for the aforementioned basic approaches.

In our approach, we have used the *Blockly* Library which is open source and follows the block-based approach. However, we could use another or more than one general-purpose visual programming editors by deploying to all the editors the respective logic that is described in this chapter.

4.1.1 Blockly Editor

As already referred in this thesis, our IDE is extendable through the development of new components that could communicate to each other. In this context, we have developed a general-purpose visual programming editor component of the IDE by incorporating the *Blockly*. In particular, each visual programming editor has to support a set of functionalities in order to undertake the role of an editor for the IDE. In the next paragraphs, we are going to discuss these functionalities.

The main functionality of visual programming editors is the *opening of visual sources in editor workspace instances*. Loading sources in editor instances includes several settings. One of them is the user privileges (i.e. read-only, editing, not accessible). In case of visual programming editors, there are two main parts. The first part of a visual programming editor is the toolbar(s) which are used to select and handle the visual graphic parts during the end-user development process. The second part is the main workspace area in which the visual sources are be visualized. In this context, the user privileges for the visual programming editor instances are handled by loading the toolbar(s) in case of editing mode and not loading the toolbar(s) in case of read-only mode. In addition, in case that a visual source is not accessible, the visual programming editor instance does not load the source and present an appropriate warning message. In case of the Blockly editor, three privilege modes are supported as depicted in Figure 4.1.

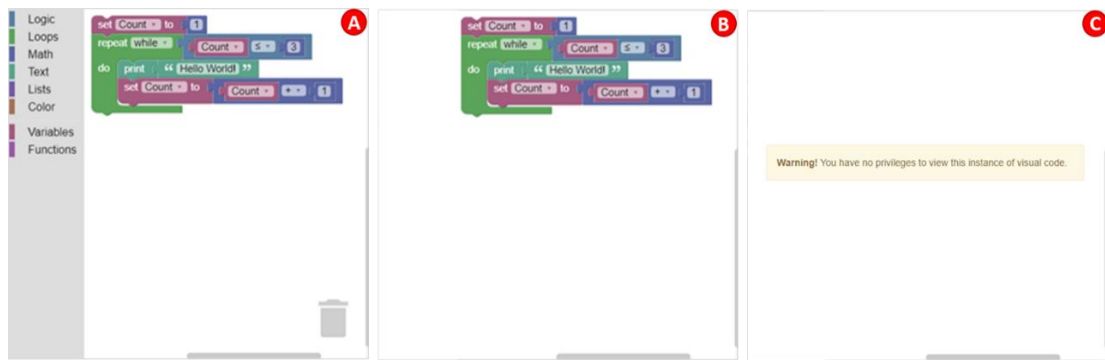


Figure 4.1. Blockly Editor privileges modes; editing mode (tag A), read-only mode (tag B) and not accessible (tag C).

Another two important functionalities that are required for an editor of the IDE are the following: Closing and saving visual sources. Closing an editor workspace instance means that instance used memory will be deleted and the view that is inserted in DOM will be removed in order to prevent memory leaks. In this context, the *Blockly* library provides API to dispose the workspace. Moreover, the Project Manager (see Chapter 5) undertakes the functionality of save the editor workspace instance by communicating with the back-end of the IDE in order to save project data in the data base. However, it requires from each visual programming editor to provide the respective visual source (i.e. model) of the editor instance. In this context, the Blockly editor exports and imports the visual sources in the form of *XML*.

Moreover, in case of the real-time collaborative editing (see section 8.1), functionality of syncing is required. Sending peer to peer message with whole updated visual source and updating the visual source to the other peer could work and be tolerable from the peer users (i.e. without lagging issues). However, *Blockly* editor supports syncing by send only the change event data and applying them in the other peer side. In addition, functionality of browsing specific visual elements is required in order to open and highlight them (e.g. ask which visual element caused an action, request highlight a visual block for the peer users for presentation purposes in collaboration editing). In this context, the project manager provides functionality that requests from the responsible visual programming editor to open a visual source and then, request from the editor to highlight specific visual elements.

Furthermore, each visual programming editor is responsible to export source code or data that will be used for the execution of the project. In this context, *Blockly* library

provides API that generates JavaScript source code from the workspace instance. Additionally, each visual programming editor instance has to support basic actions such as copy, cut, duplicate, delete visual programming language elements and undo-redo functionality as well. *Blockly* library supports these actions and we incorporated them in the *Blockly* editor instances. Moreover, functionality of tracking the visual programming language elements that have been developed for each of the visual programming editor instances is required. We have built an extra layer with API on the top of *Blockly* editor component that can provide information about the use of each visual programming language element and the visual programming instances that are loaded.

Moreover, there are two more directions of functionalities for visual programming editors. The first direction is the configuration of the editor instances in the context of authoring application domains. The second direction is based on the intelligence of visual programming editors including automatic visual code suggestions, visual code assistance, visual code snippets, etc. In the next two sections, we will discuss regarding the configuration of visual programming editors and the visual code snippets.

4.1.2 Configuration of Editor Instances

The visual programming editor instances could be configured either from the end-user developer as previously discussed (see section 3.1.2) or from the developer of the application domain. In the context of the application domain, the developer is able to define the configuration of visual programming editor instances based on the concept, the parts and the style of the project elements of the application domain. The visual programming editor configuration includes two categories as follows.

The first category refers to the customization of the visual programming editor's style for a specific visual source instance. For example, Blockly's visual source instance can be rendered by default (e.g. toolbox is positioned vertically along the leading edge, the positions of the undo/redo buttons are in the bottom-leading edge corner and the trash can button is in the bottom edge corner) as depicted on the top of Figure 4.2 or could be rendered alternatively (i.e. toolbox is positioned horizontally along the bottom edge, undo/redo buttons are in the top-leading edge corner and the trash can is located in the trailing edge corner) as presented on the bottom of Figure 4.2.

Moreover, *Blockly* editor enables to customize the view (e.g., background color, etc.) apart from the layout of the user-interface parts.

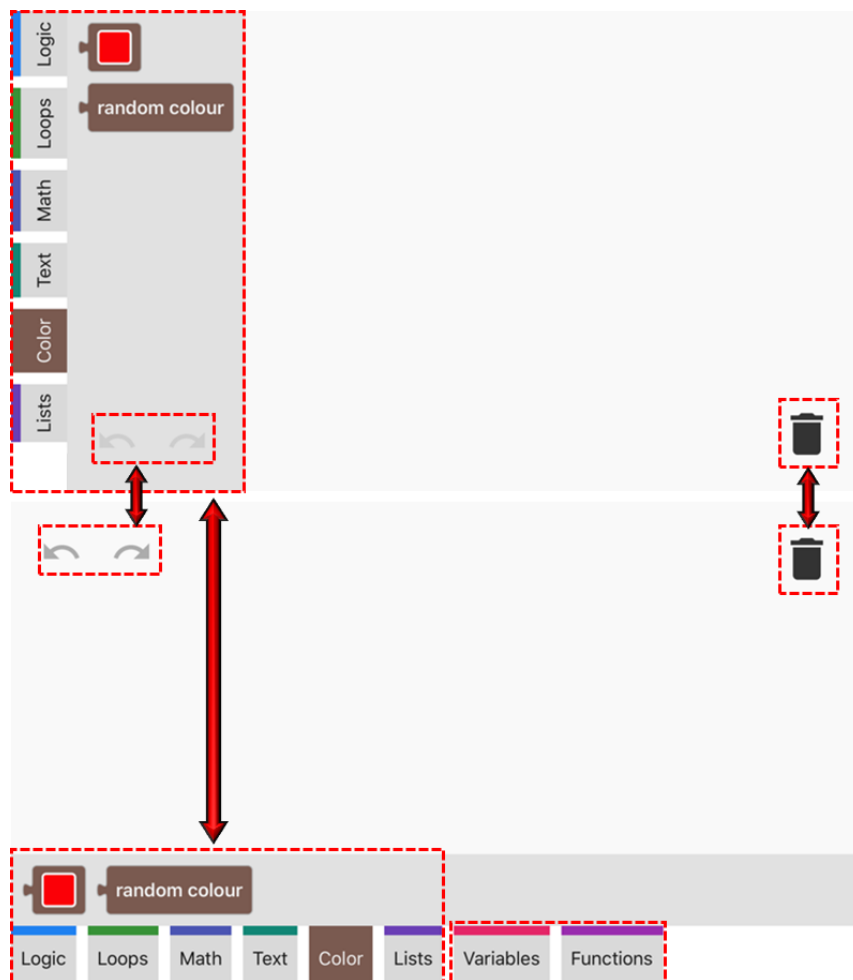


Figure 4.2. Default View of Blockly's instance (top); Alternate View of Blockly's instance (bottom).

The second category refers to the customization of the visual programming editor's filter for the visual domain elements. In particular, the domain author chooses which of the supported VPL domain elements will be enabled in the toolbar of the visual programming editor configuration. Based on the configuration, the IDE undertakes to refresh the toolboxes of the visual programming editor instances (if needs) when a VPL domain element instance is created, edited or deleted (see section 4.2.1). For example, on the top of Figure 4.2, *Blockly's* editor instance is filtered not to include the last two categories of the *Blockly* Blocks in the toolbar (i.e., Variables and Functions).

4.1.3 Visual Code Snippets

Code snippets are small blocks of reusable code that can be inserted in a source code file by using a right-click menu (context menu) command or a combination of hotkeys (i.e. shortcuts). They typically contain commonly used code blocks such as loops or conditional statements, but they also can be used to insert entire classes or methods, etc.

In the case of visual end-user programming, we introduce the visual code snippets. The visual code snippets could be defined either by the domain authors or by the end-users. The domain authors are able to define visual code snippets based on the requirements of their application domain. For example, in case of the user interfaces, visual code snippets could be added that iterate all the designed screen areas and change specific properties (e.g., the background color). Additionally, the end-users can define their visual code snippets or edit existing ones in order to reuse them for their applications. However, this feature could be usable by more experienced end-users only.

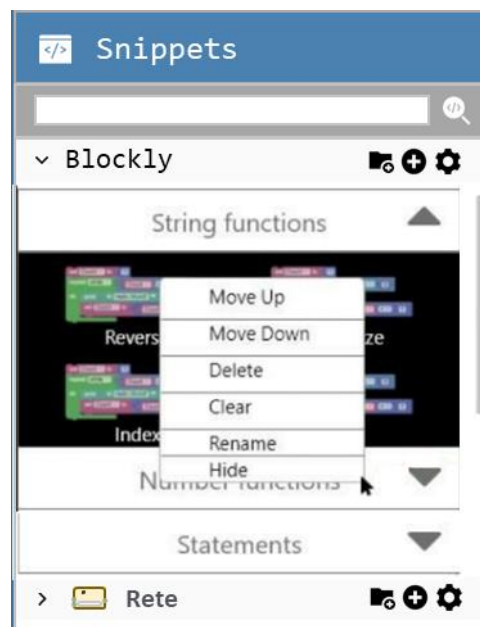


Figure 4.3. Visual Code Snippets Toolbar.

Compared with the code snippets for developers, the visual code snippets require to support extra features for their efficient functionality and usability for the visual programming IDE. In particular, visual code snippets must be presented by the visual programming editors. This requires the toolboxes that will be hosted by each visual programming editor, their minimized

view in the toolboxes, their categorization, etc. As a result of this, the visual code snippets are dependent on the visual programming editors. In this paragraph, we discuss the functionality of the features by using the *Blockly* editor as an example. However, the same logic has to be followed by other general-purpose visual programming editors. Although the visual code snippets are handled by the visual programming editors, there is a main visual code snippets toolbar which is responsible for viewing and managing the visual code snippets presented in Figure 4.3.

4.1.3.1 Administering Snippets

The first feature that has to be supported for visual code snippets is the ability to build new snippets. There are two ways for the end-user to create a new one: The first way is by creation of a new snippet from scratch by providing a menu item choice “*New Snippet*” for each of the visual programming editors as presented in Figure 4.3. The second way is to select the visual code that the end-user would like to define as a new code snippet by the visual programming editor instance, click right mouse button and



Figure 4.4. Pop-up dialogue for *Blockly*'s code snippets creation.

choose from the menu option “*New Snippet*”. Then, a pop-up dialog opens which includes the visual programming editor instance area for the visual code and a form to fill-in the category, the title and the description of the visual code snippet as depicted in Figure 4.4. In addition, this pop-up dialogue opens when editing the visual code snippet. Finally, the end-users are able to delete a visual code snippet by using the menu option “*Delete*”.

The visual code snippets are separated into two main categories, the general purpose and the domain specific visual code snippets. In addition, the domain authors and the end-users are able to define sub categories for the visual code snippets. In this context, they are able to rearrange the order of appearance, as well as to delete, initialize, rename or hide a category. Moreover, the order of appearance could be adapted based on the recently or most used snippets category.

4.1.3.2 Using Snippets

The developers use keyboard shortcuts in order to insert the source code snippets. In case of visual code snippets, the end-users are able to search them through an appropriate search toolbar. Each visual code snippet includes a label and category that the search mechanism uses to find the appropriate information as presented in Figure 4.3.

Each visual programming editor which supports visual code snippets, has to provide a hosting area of the toolbox whose visual code snippets are visualized. The end-user developers are able to instantiate a visual code snippet either by right click or by drag and drop in the visualization main area of the visual programming editor. When a new visual code snippet is instantiated, there are fields which have to be filled-in. The visual programming editor focuses on these fields (i.e. values and variables) and the end-user developer is allowed to handle them. Alternatively, a pop-up could open per each of these fields in order to handle them all.

4.2 Domain-Specific Visual Programming Language Elements and Editors

The domain-specific visual programming language editors are used to develop and/or handle one or more domain visual programming language elements of the application. For example, the graphical elements of the user interfaces are developed using *WYSIWYG* editors. Each graphical element corresponds to a domain visual

programming language element. Additionally, smart objects in the context of personal automations in the Internet of Things domain are developed using a *Visual Smart Object Editor* (see section 9.1). The smart object corresponds to a domain visual programming language element and each smart object which is registered in the application corresponds to one visual programming language element instance which includes its personal data.

The developers of an application domain framework have to incorporate domain-specific visual programming editors for their application domains. These visual programming editors will be used by novices in order to be able to develop the corresponding VPL elements for their applications. Thanks to the component-based architecture which is followed by the *Blockly Studio IDE*, the domain author is able to develop it as new plugin(s). In addition, third party libraries could be used by the visual programming editors for the application domain. For example, in the case of the domain of the personal automations in the Internet of Things, middleware for the communication among the end-user developed applications and the smart objects is required (e.g. *IoTivity*).

Additionally, the domain-specific visual programming editors, apart from developing and handling the domain visual programming language elements, are responsible for the source code generation which corresponds to the visual sources that are created. In particular, the developed VPL elements are saved by the domain-specific visual programming editors in visual sources (i.e. DSL format). Based on these visual sources, the domain-specific visual programming editors generate source code. Source code generation targets either the project execution or the debugging process in the context of the IDE workspace.

Moreover, the domain-specific visual programming editor has to export data from the visual domain element instances which are handled and notify the IDE with appropriate signals for end-user actions (e.g. create, edit, remove etc.). This helps the aforementioned mechanism, developed on the top of *Blockly*, which automatically handles the development dependencies between the project elements. This mechanism is analyzed in the following section.

4.2.1 Supporting Behavior of Domain VPL Elements

Designing instances of domain VPL elements is not adequate for the end-user programming of an application. In particular, their behavior must be developed in the applications. For example, in the case of the graphical elements, the end-users have to develop the logic and the events for an interactive user-interface. In the case of smart objects, the domain VPL editor is specialized in registration and communication between the smart objects and the applications. However, the domain specific visual programming editors do not include the logic and the instructions of the behavior of an application. This requires the definition of a behavior handling set of new visual elements (e.g. blocks in case of *Blockly*) for each domain VPL element. The role of visual programming in this part of the application is handled by general purpose VPL editors (e.g. *Blockly* editor for our approach) that are registered in the IDE.

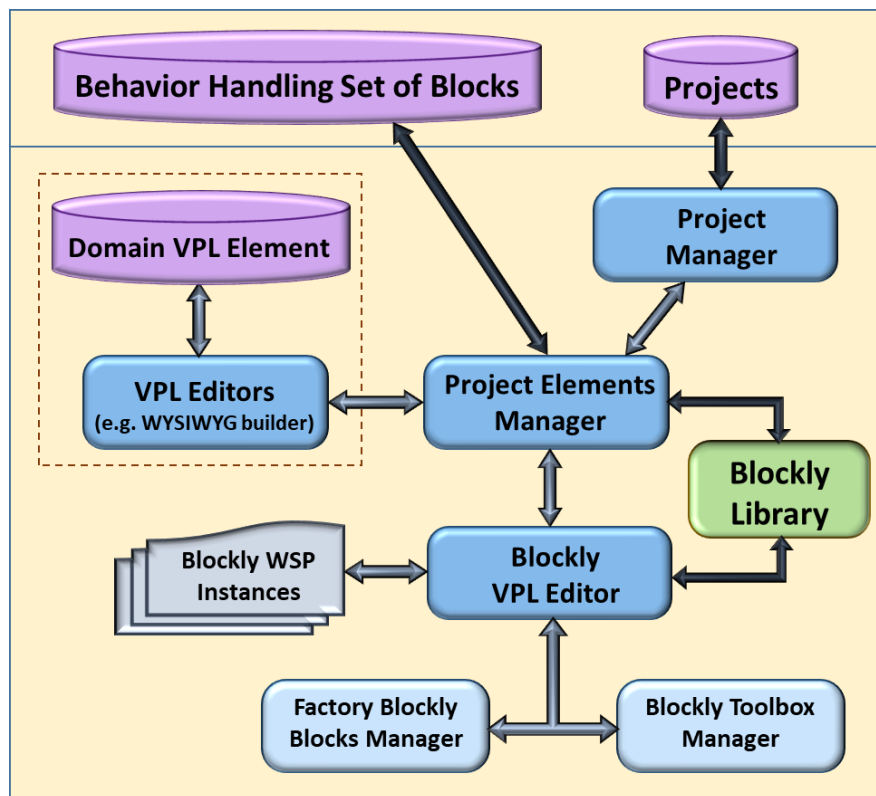


Figure 4.5. Extension mechanism for *Blockly* to automatically manage the behavior handling set of blocks for visual programming language domain elements.

Each *Blockly* editor instance consists of two main parts, the block canvas area in which visual code is designed and the toolbox that is the side menu through which the user may create new blocks. The *Blockly* Library supports creating custom blocks and configuring the toolbox for its *Blockly* editor workspace instances. Based on this, we

are able to define new blocks and toolboxes for the project elements. However, this is not adequate for managing the project element dependencies. Particularly, existing *Blockly* blocks and toolboxes have to be dynamically changed during the development process. For example, in the case of personal smart automations in the Internet of Things, when new smart objects are registered (i.e. added) in the project, new blocks have to be defined and the toolboxes have to be updated by adding these blocks. In the same context, when a smart object is unregistered (i.e. removed) from the project, the corresponding definitions of blocks have to be deleted, the toolboxes have to be updated and possible instances of these blocks in Blockly editor workspace instances have to be removed.

The latter led us to build a layer on top of the *Blockly* library for our IDE (see Figure 4.5). This extension requires to define the behavior handling set of *Blockly* blocks for each one of the visual programming language domain elements. The visual programming language domain elements information is exported by the specific domain visual programming editors through which they are managed (i.e. create, delete, edit etc.) by end-users. Getting this information as input in our mechanism and using the behavior handling set of Blockly block definitions, our tool undertakes to automatically define the required Blockly blocks as well as to generate and update the Blockly editor toolboxes. In this context, the development dependencies between the project elements are handled. When end-users attempt to delete a defined domain element instance from the project (e.g. a registered smart object), they are warned which project elements will be affected in this case.

4.2.2 Linked Visual Programming Elements

The domain authors are able to define links among the visual programming language elements by adding in their definition an extra field of visual source data that corresponds in the link. Moreover, the visual programming editor has to support right click functionality in the visual programming language elements. Afterwards, when the end-user developer chooses to browse the linked visual programming editor instance, the editor manager component undertakes to open the responsible visual programming editor.

For example, using *RETE*, a flow-based programming editor, defining the nodes of the editor, the domain authors are able to define the browsing linkage. In particular, they are able to include an extra field of linking the visual source information. In this case, extra menu items

are added for the specific visual domain element (by right clicking) in the visual programming editor (see Figure 4.6). The visual programming editor communicates with the editor manager which handles the browsing by using the authoring data.

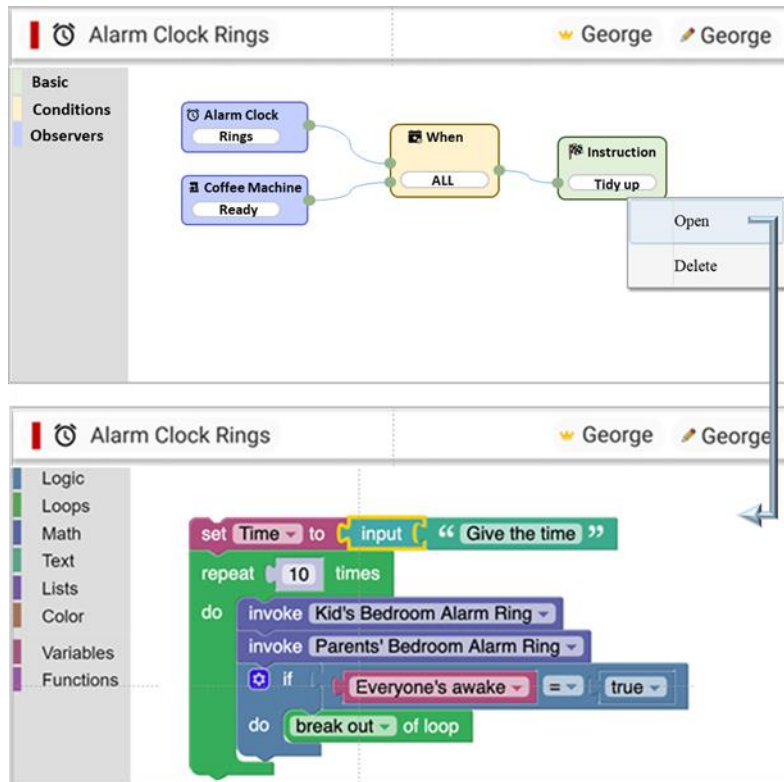


Figure 4.6. Linked visual programming language element with other visual sources.

Chapter 5

Projects

"Man is a tool-using animal. Without tools he is nothing, with tools he is all."

- Thomas Carlyle

The set of sources that are developed for the programming of a software application constitute its project. Since every project consists of many sources with different extensions, it is sometimes hard to find, create and handle them. In this context, the project manager is one of the core components of the IDE by undertaking to manage the application sources. The common view of a project manager is a tree view in which every node represents a project element that relates with a folder or a source of the applications. The project managers facilitate the developers to better organize and structure their projects, however, this is not an easy process for novices or non-programmers. In this context, in case of visual end-user programming workspaces, the project managers have to be more user-friendly and targeted in order to facilitate the application structure and development process in general. In this chapter, it is firstly presented the project manager of our IDE that is configurable based on application domains project manager we have developed, Then, we will discuss about the project elements that can be authored for application domains and the editor manager provided features for the browsing of project elements.

5.1 Project Manager

The project manager is one of the core components of the IDE. This component undertakes the managing of the application sources that the end-users develop. However, the project manager is more demanding in the case of novices than in the case of software developers. This could be easily perceived by considering that novices are not experienced in structuring the sources of the project of their applications. Due to this the project manager has to be more use-friendly and targeted on specific application domain by restricting the structure of the project elements, the project element types would be available to create in the development process, the

available user options, etc. However, each application domain has different requirements for features, different project element types available for the development process, etc.

5.1.1 Authoring Project Structure for Application Domains

In this context, we have developed the project manager in order to be configurable based on the application domain that the visual end-user development focuses on. The developer of the application domain is able to author the project structure and the functionality of the project manager. In particular, the project manager is configurable based on four sections as depicted in Figure 5.1.

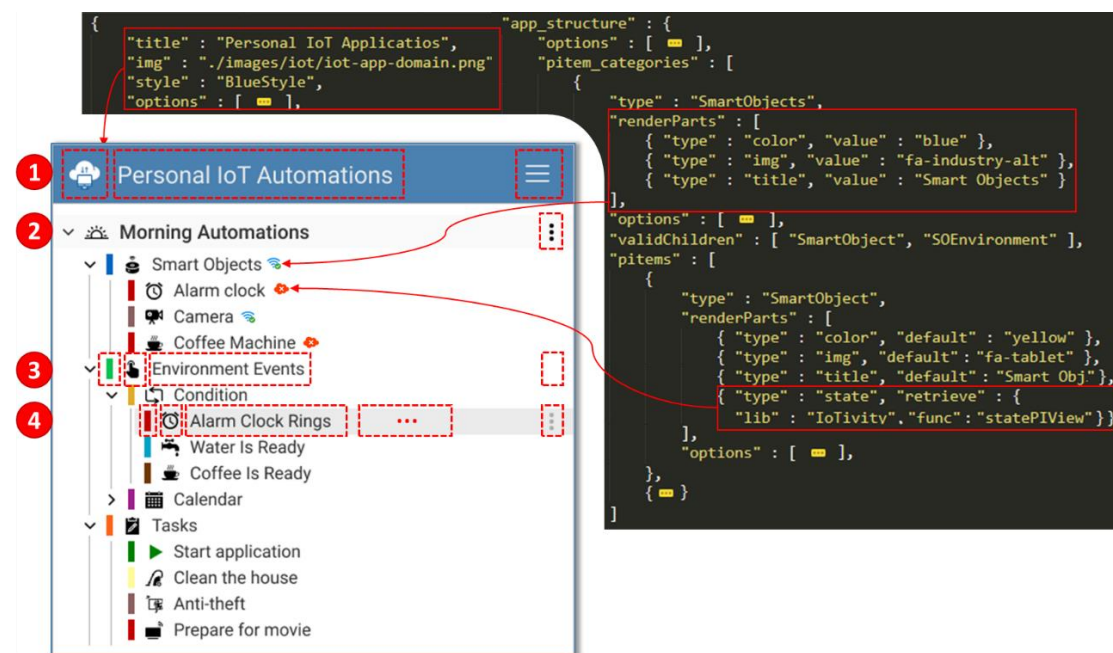


Figure 5.1. Configurable view parts of the project manager component.

The first section is the application domain label consisting of the application domain image, title and menu options. Using the latter, the application domain author is able to restrict which will be the abilities of an application domain (e.g., create new project, open project, open all projects, delete project, exit etc.).

The second configurable section is focused on the application domain projects contents. In particular, the domain author defines the structure of the projects' visual sources, authors the project categories (i.e., folders in case of text-based development) and the project element types that can be created by the end-users. Additionally, the

application domain author is able to select which will be the menu options (e.g., close project, rename project, share project, delete project etc.).

The third section is the definition of each one of the categories of the projects. Each category view includes the color (optional), the image (optional) and the title (see tag 3 of Figure 5.1). Each category optionally includes a list of available sub-category types and project element types. Moreover, predefined sub-categories and project element instances are able to be developed by the domain author for the end-users. Furthermore, predefined visual sources may be authored as read-only with the possibility of being not renamable and not removable. All the aforementioned options for the project structure are addressed by defining the specific options that will be available for the users.

The fourth section of configuration is the project element types. For each project element type, the domain author has to author the view (see tag 4 of Figure 5.1). Particularly, they have to select which of the information will be included (i.e., color, image, title) and optionally define any extra data view. For example, in the case of the smart objects, the application domain author may define the state of the smart devices (e.g., online, offline, etc.). Moreover, extra elements are able to be rendered due to project element properties (e.g., shared elements, read-only elements, etc.).

5.1.2 Functionality and Style

As previously mentioned, the project manager supports user actions (i.e. create, delete, edit, etc.) that are available either by click on three dots button which positioned on the right side of each node of the tree view or by right click in the node. When user triggers the action, an appropriate dialogue opens to serve it. In particular, the developers of the application domain are able to define which will be the ingredients of the dialogue for the specific user action. They are able to choose which of the project element values (title, image, color, etc.) will be available to be handled by the user, which of them will be visible on dialogue, etc. In addition, they are able to define the actions that will be available at the bottom of the dialogue. The domain authors are able to develop extra actions that will be accomplished by their added components.

Also, default functionality of a Project Manager such as search project elements, drag and drop, automatically-sorted project elements etc. exists and the domain author is able to select it or not according to the needs of the application domain. Last but not least, the domain author is able to customize the style of the project manager by defining new styles or by using existing ones from other application domains that have already been authored.

5.1.3 Settings for Project Elements

For each project element type, the domain author could define settings that are relevant to the specific element and/or the whole domain project. There are standard types of settings that are supported, including drop-down list of choices (see example in Figure 5.2). In particular, the application domain author is able to define any value type either basic or aggregate by reusing the user-interface code generation that is developed for the purposes of the “*Configuration Manager*” as mentioned earlier on section 3.1.2.

Furthermore, the application domain author is able to define function name that will be used as callback in order to get the option values in case of select *HTML* view. This could be useful in case the options depend on the end-user development process (i.e., by development actions the list of options is affected). Additionally, the application domain authors are able to define types which will be handled by specific third part tools (e.g., one *Blockly* workspace instance could be defined in order to handle dynamic settings through visual programming) and will be injected in the dialogue with other settings.

The values of these settings are accessible in the visual programming editors, the project element templates (described in the section 5.2.1) and the runtime environment (described in Chapter 6). Moreover, the options are available by choose to view the project element settings. As a result, the end-user developers are able to edit the values of these options during the development. However, the application domain authors may would like to base the construction of the project element in these options and wouldn't like to allow edit in specific option(s). In this context, we enable two more choices in authoring of the options. The first choice enables the application domain author to render the value option as read-only when the project

element has been constructed, while the second choice enables that the option will not be visible.

This implies that the application domain author could express them in the visual end-user programming time by rendering different visual programming view areas or different view parts. For example, the user could choose to develop the project element by using the Blockly Editor or alternative editor (if exists) and then constructing the project element, the chosen editor could be used. The latter means that the project element types could be dynamic driven by the user settings for the project elements. Furthermore, the users define the experience level in the context of programming. Using settings of the project element, the domain author is able to define rendering of different project elements per the user experience. Additionally, settings are able to be utilized during the execution time by interpreting their meaning. For example, the user may choose when or how to execute the project elements during the project execution as depicted in Figure 5.2.

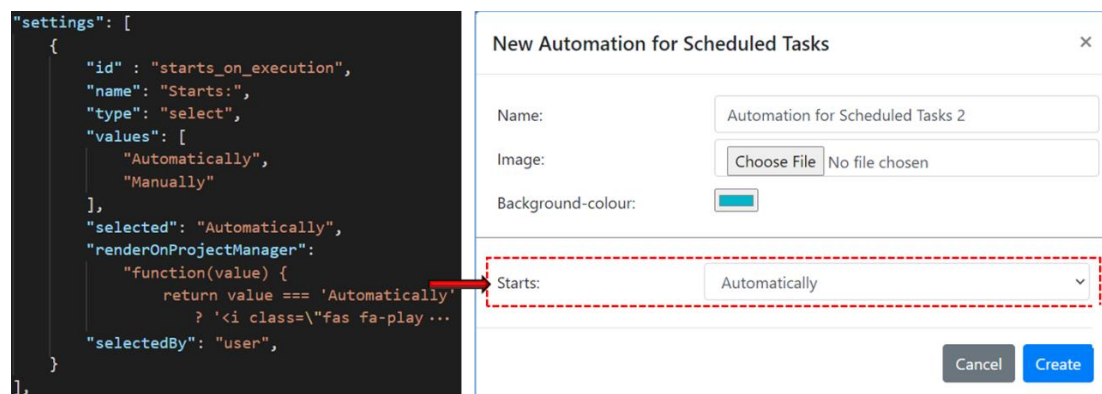


Figure 5.2. Authoring settings for project element type.

5.1.4 User Action Hooks and Validation for Project Elements

Additionally, the project manager supports real-time validity check through rules for user actions (i.e., create, edit, delete) of the project elements. In case users insert invalid input, an appropriate message is rendered on the top of the input field. In the context of tests, standard rules have been developed such as prevent duplicate names, start labels with/without specific names etc. Moreover, based on the application domains, different rules have to be applied. In this context, the application domain author has to define which rule or rules will be applied per project element type. Moreover, they are able to define new rules through scripts.

Furthermore, based on the rules of the application domain project, our approach empowers the domain authors to develop user action hooks. In particular, they are able to develop functionality that will be applied before and after a user action. Using this feature, the application domain authors are able to prevent an action based on the rules that they would like to follow in their application domain framework. In addition, they are able to add functionality to the before and after user action. For example, they are able to add or change the user action dialogues for the specific user action based on the project state, the project element state, etc.

5.1.5 Using Alternative Project Manager or None

Moreover, As discussed earlier, the *Blockly Studio IDE* is extendable following component-based architecture and each of the components is loaded dynamically when the IDE initializes. Based on this, the domain author could configure which components will be loaded. Each component implements a specific interface. So, the domain author is able to replace the provided project manager if it is required for its domain application. Moreover, the domain author could choose not to use the Project Manager for the domain application at all. This means that the domain application project will be limited to one project element. However, this choice does not mean that it will be limited to one visual programming editor instance. This depends on the authoring of the project elements that are described in the following section (see section 5.2).

5.1.6 Authoring by Using JSON Schemas

The application domain authoring for the project manager is written in the form of JSON and has to satisfy the *JSON Schema Validator* [137]. Using this mechanism, the platform will notify the domain author of any possible mistakes regarding the authoring process. The same logic applies to the authoring of the project manager's style. Alternatively, the project manager exports an API through which developers may customize and author the application structure of the project, instead of doing it by defining JSON data.

5.2 Project Elements

Each type of project element includes by default one visual source which is loaded and handled by specific visual programming editor(s). However, project elements

apart from the visual editor data, include information (i.e., color, image, title, author etc.) and values of the settings (see section 5.1.3) that domain authors may would like to interpret them in specific style. Additionally, they may would like to define more complicated project element types including more than one visual programming editor instances which will be injected in a customized user-interface. Moreover, the number of visual programming editor instances that are included in a project element may changes during the end-user development. In this context, we introduced authoring of project element templates in *Blockly Studio IDE*.

5.2.1 Templates

For each project element type, the domain authors are able to describe the contents will be included in project element instances. In particular, they can author the view, the interactivity, the injected visual programming sources and which visual programming editor will handle them. Using templates, the project element information can be rendered (e.g., the file name and/or path of the project element, the date created, the current end-user actions, the author etc.). Furthermore, the functionality and the style can be developed through JavaScript and CSS in the case of more fancy and interactive project elements. Additionally, the domain author could define one or more visual programming sources of visual programming editor instances that will be injected in the designed empty *DIV* elements during instantiation of a project element. The templates are saved in a repository and the domain authors are able to develop new ones or re-use already existing ones. The development of such templates includes the following parts:

1. *Lodash* template [138] (i.e. *HTML* enriched by template tags).
2. Cascading Style Sheets (CSS) used for the presentation.
3. Map of the empty *DIV* elements (i.e., Selectors) and the configuration for visual programming editor instances (see section 4.1.2) will be injected. Additionally, for each of the visual programming editor instance is defined if the instance will be loaded or not when the project element is loading.
4. Required functionality for project elements (e.g., on focus view, on close, etc.).
5. Required functionality of the rendered *HTML* from *Lodash* template.

The IDE handles these templates in order to address the functionality of the project element (e.g., create, open, close etc.). Using this mechanism of templates, the domain authors are empowered to design and develop any project element type that will be required for their application domain frameworks.

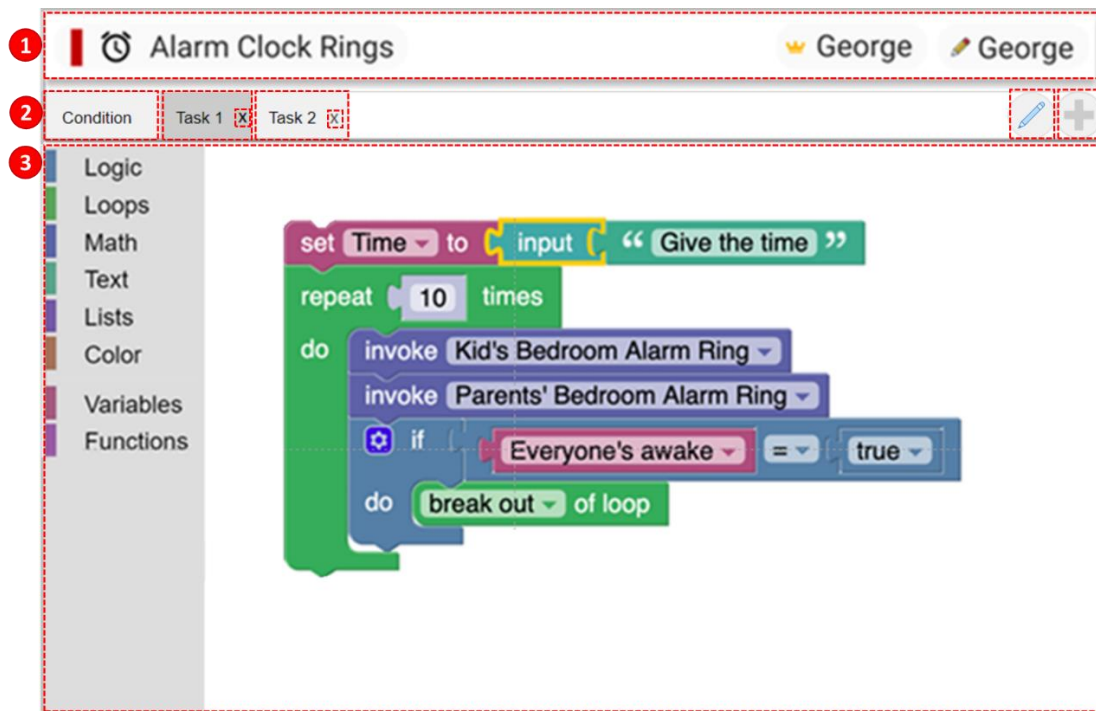


Figure 5.3. Example of a project element template; project element information (tag 1); interactive parts of the template (tag 2); area for visual programming editors (tag 3).

An example could be the following template. We have developed a classic tab view in which each tab area hosts one *Blockly* workspace instance. The template includes three parts (see Figure 5.3). The first part presents the information of a project element (i.e., file name, author and owner). The second part of the project element enables the end-users to browse among the visual programming editor instances, create new tasks (i.e., *Blockly* workspace editor instances), rename the name of a task and delete a task. The last part of the project element template includes the areas in which the visual programming editor instances are hosted (see tag 3 of Figure 5.3). Another example could be the form-based smart object editor that is discussed in Chapter 9. This visual programming editor loads a dynamic number of actions based on the functionality that is provided by each smart device. We provide *Blockly* editor instances for each of these actions in order to simulate the functionality of the action in case of debugging the smart IoT automations. In this case, the only visual

programming editor instance that opens is the smart object editor and the *Blockly* editor instances loads only when event is triggered by the smart object editor.

5.2.2 Hosting and Browsing Project Elements

The “*Editor Manager Component*” of the IDE handles the view of the project elements. The end-user could view one or more project elements in parallel by splitting the editor manager’s area horizontally or vertically as depicted in Figure 5.4. During the end-user development process, the end-users are able to browse the project elements through the *Project Manager*. Additionally, two options are given by *Blockly Studio IDE* to provide browsing of project elements. The first option is by enabling the action of “*GOTO*” previous or next project element that was loaded, using the previous and next tool items appearing in the toolbar (see second red rectangle of Figure 5.4).

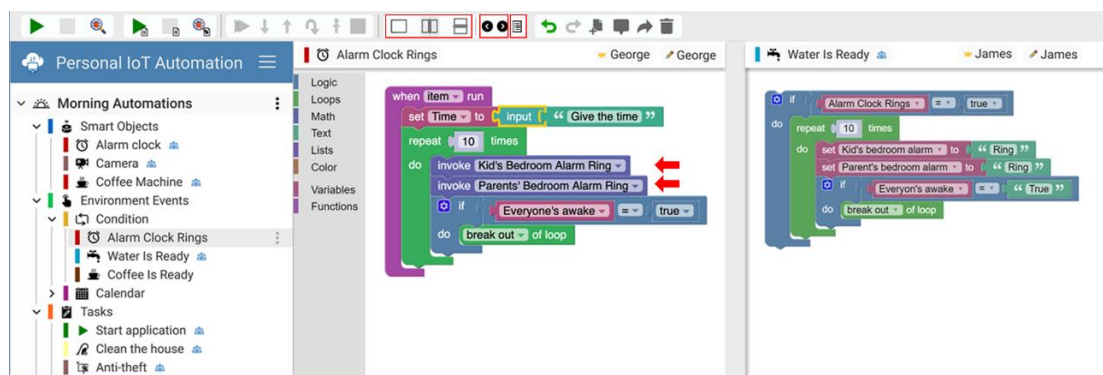


Figure 5.4. Splitting in two project element instances area vertically.

5.3 Project Dependencies

There are dependencies among project elements and visual sources that are included in the context of a project. In the case of software developers, dependencies are handled by them. For example, when developing a GUI application, the design screen parts constitute project elements on which graphic elements are designed. The logic of these elements is developed in other source(s) which depend on the aforementioned project element and the developer has to handle it. Also, the developer has to handle graphical elements which have been created, edited or deleted.

In the case of visual programming languages, the Blockly Studio IDE handles the dependencies automatically. The project dependencies are caused by the visual

programming language elements which have been developed in the visual sources during the end-user development process (see Figure 5. 5). As we have already presented in this thesis, the dependencies of the visual programming language elements are handled by the extra layer mechanism of our approach (see section 4.2.1). This mechanism is based on signals are posted when one of the basic actions (i.e., create, edit and delete) happens during the end-user development time. However, this mechanism is defined for dependencies among general-purpose visual programming editors either from specific domain visual programming editors or from general-purpose visual programming editors. In case of dependencies between domain-specific visual programming editors (e.g., dependency for *WYSIWYG* editor and Smart Object Editor), the domain authors have to handle them by utilizing the posted signals.

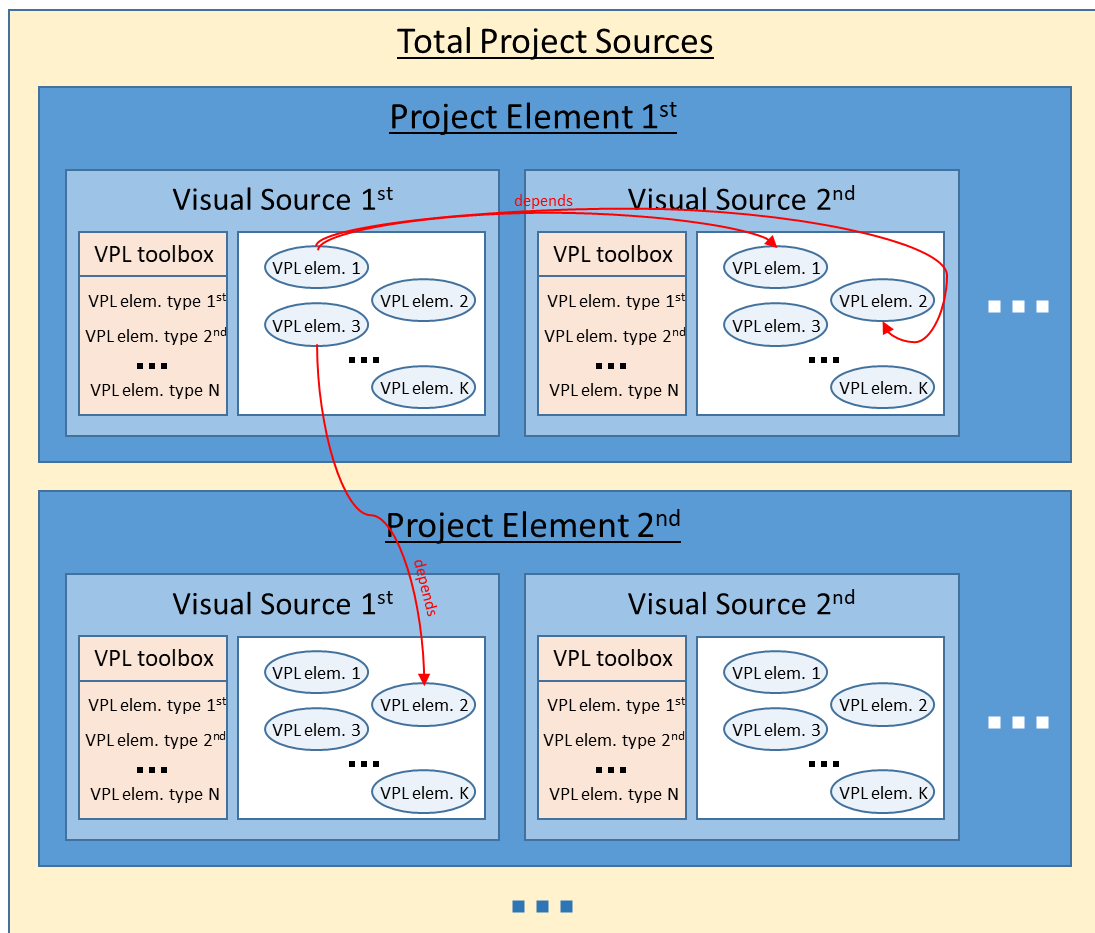


Figure 5. 5. Visual programming project sources of application and dependencies among the visual programming language elements.

Chapter 6

Runtime Environment

“Nevertheless, I consider OOP as an aspect of programming in the large; that is, as an aspect that logically follows programming in the small and requires sound knowledge of procedural programming.”

-Niklaus Wirth

When software developers write programs, they need to execute and test them. Therefore, the integrated development environments include runtime environment components that allows the programmers to execute their programs and interacts with the input-output console, the extra user-interface view based on domains, etc. Based on the programming languages and the libraries are used for the development of an application, the software developers have to set up the environment by installing compiler or interpreter, libraries, etc. In case of novices and non-programmers, the visual programming IDE has to handle the setup of the environment without burdening them. In our approach, the runtime environment has to support the execution for every domain application. In this chapter, we discuss the runtime environment for the *Blockly Studio IDE* and how our approach is envisaged to work for all the potential application domains. Additionally, we discuss the selective execution feature that we introduce, the input-output console and the hosting of extra user-interfaces for the application domains. Concluding, we discuss the potential of isolating a project as an independent application.

6.1 Hosting the Runtime Environment

Based on the component-based architecture which is followed by *Blockly Studio IDE*, we developed the runtime environment as an independent component. Loading the workspace of the IDE, the runtime environment is initiated by registering the tool items in the IDE’s toolbar (see sub section 3.1.1.2). When users choose to run the application, the IDE instantiates the project execution by retrieving the project data of

the application. In this context, the users would like to be able to use the IDE in parallel during the project execution. However, hosting the project execution in the same execution context with the IDE, issues will be appeared. In particular, executing the application's source code, the event loop system of JavaScript could be locked and the IDE could freeze until the end of the execution of the program. There are two different approaches which could be used to solve this issue and host the run-time environment in the IDE:

The first approach could address the issue by code decoration. In particular, the run-script could be executed in the same context with the IDE by using code decoration (or instrumentation). The latter is a technique that applies the insertion of extra special-purpose instructions, either at source or at the binary level, with the intent of introducing additional mission-specific functionality, however, without altering the original observed behavior of the subject program. Based on code decoration and the *JavaScript Generators*, the run-time system executes each visual programming language instruction, then gets the control to satisfy possible IDE's requests and afterwards continues to the next instruction and so on. This technique requires to care for the naming of the variables and the events must not override the IDE. However, this is not a problem, due to the context of visual programming and code generation. This means that the domain authors have to follow specific rules for the names that will be used in the run-script and the code generation by always using a prefix e.g. "runtime_script" for all of them. This technique will be used by the system to address user actions (i.e. stop, pause application) as discussed in the following section. Applying this approach, brings extra requirements for third-party domain-specific visual programming editors.

As result, we adopted the classic approach of runtime environments for IDEs. In this direction, we developed the project execution manager of the runtime environment as a third-party application (see right top of Figure 6.1). This application communicates with the IDE's runtime environment by using the communication infrastructure which discussed previously in section 3.1.3. Following this approach, the issue of IDE freezing is addressed, while the IDE environment is not affected by the project execution environment.

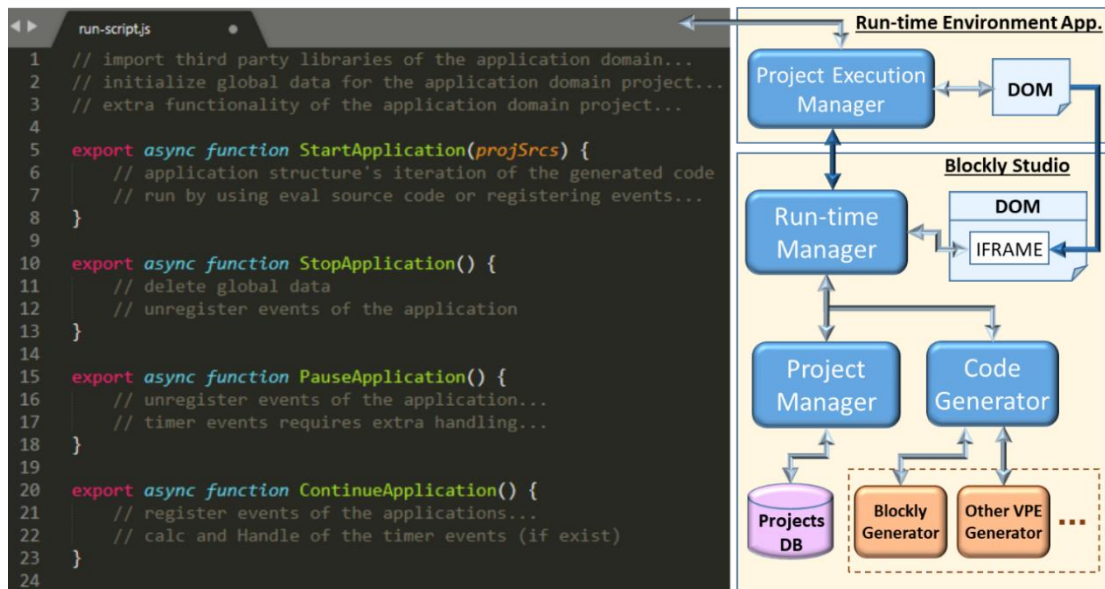


Figure 6.1. Authoring runtime of a domain project and runtime environment system of the Blockly Studio IDE.

6.1.1 Running Projects of the Application Domains

Each project which is constructed using the IDE has a different way to be executed based on the application domain it belongs. This issue arises from the authoring of application domains which differs in the application structure (see section 5.1.1) and the set up (i.e. third-party domain libraries, etc.).

In this context, the execution of the project is based on appropriate scripts that are developed per application domain. These scripts are the entry points of the application and undertake to load the required third-party application domain libraries, initialize the required application domain data and the extra application domain run-time view (if exists). Moreover, the run-script exports functions to handle user actions of the IDE (i.e., *Start Application*, *Stop Application*, *Pause Application* and *Continue Application*) to run the application (see left of Figure 6.1). In the following paragraphs, we describe each of the user actions:

Start Application: When the user chooses to run the application, the IDE instantiates the project execution of the runtime environment. Then, the runtime environment requests the project environment data from the project manager. In this context, respective JavaScript source code and/or execution data are generated for each visual programming source, for each one of the project elements, using the responsible code

generators that are provided by the visual programming editors. The generated source code data is mirrored with the application domain structure which is defined by the domain author. Afterwards, the run-time environment calls the exported function '*StartApplication*' of the script, giving the generated source code data as input. The run-script function uses the EVAL function [139] to execute the generated source code parts.

Stop Application: The action of "*stop*" requires control of the flow of the application execution. The runtime system accomplishes the stop action by using the JavaScript control flow and error handling [140]. In particular, when the end-user chooses to stop the run process, the run-time system has to interrupt the execution of the application by causing internal exception (i.e. throw exception) and then handle it appropriately. However, the visual programming elements are not matched with source code instruction one by one. This would interrupt the execution of the project in an unexpected state of the run-script (i.e. not completed execution of the current visual programming element). In this context, the code generator of each one of the VPL editor's code generator injects an extra instruction in the end of each visual programming element. This instruction checks if there is system state to stop or pause the execution of the application and undertakes throwing the exception in expected state. Moreover, this would not be an adequate approach to solve this issue. There are applications which run asynchronously or applications that include asynchronous and sequential instructions. The run-script will be responsible to notify the run-time system when the sequential instructions have been executed and if other sequential instructions have started from an event. In case there are not sequential instructions, the run-time system throws the internal exception to stop the execution by itself instantly. Then, handling this exception, the run-time system calls the function "*StopApplication*" which will be exported by the run-script. This function is responsible to reset and/or delete the required data included in the run-script. Furthermore, the script is responsible to unregister all the events that are registered in the context of the project execution.

Pause Application: The action of "*pause*" follows the same logic with "*stop*" action. The difference is that instead of throwing exception, in case of pause, the runtime system activates a busy waiting loop that waits until the user chooses to continue or

stop the execution. Before the activation of waiting state, “*PauseApplication*” function is called. This function is responsible to unregister the events which are activated in the context of the project execution.

Continue Application: The action of “*continue*” needs to call the “*ContinueApplication*” function which is exported by the run-script and activates back the existing events which was activated in the context of the project execution. Afterwards, the busy waiting loop is deactivated and the project execution continues with the next visual code instruction.

6.2 Selective Project Execution

Running the project during the development process in order to verify if it is working as expected is one of the main tasks. In this context, we introduced an alternative way of running the project. In particular, the end-users are empowered to run the project selectively. Starting the execution process, the default choice is to run the project including all the project elements. However, the end-user developer could choose which project elements will be included in the execution process (see Figure 6.2).

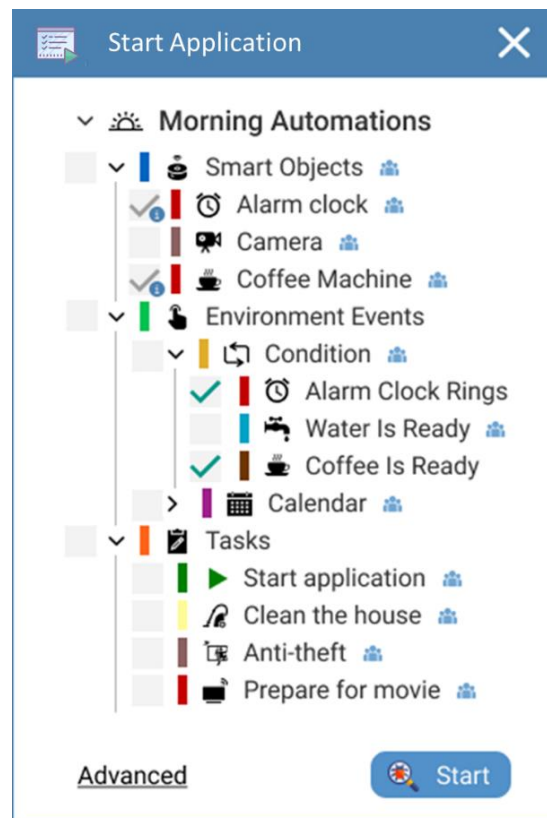


Figure 6.2. Selective execution dialogue for ‘Morning Automations’ project.

Selective execution allows the end-users to run the application partially which means that testing the functionality of their tasks will be easier. In general, separating the execution of source(s), as a feature contributes to testing the project. Moreover, the users will be able to run the project elements as independent applications.

However, selecting which project elements will participate in the project is not that simple. The project elements include dependencies between other project element(s) as discussed earlier. Having the knowledge of these dependencies, the visual programming workspace automatically adds the project elements that are dependent on the selected project elements (e.g., see the dependency of the condition event *'Alarm Clock Rings'* with the smart object *'Alarm clock'* in Figure 6.2). Moreover, there are project elements which are required for the execution of the project (e.g., the main task of the project). For such source(s), the application removes the option of deselection.

6.3 Input-Output Console

End-users are familiar with instant messaging software tools (e.g. Skype, Messenger etc.). Based on this, we simulated the output console for the applications as a chat. In particular, when an output block is executed, the users receive the corresponding messages via the console. The input text area is disabled by default and when the end-user developer has to input text in the application, it alters to enabled as depicted in Figure 6.3. Moreover, the output console interacts with the project manager. In particular, when a Blockly input block is executed the project manager opens the respective project elements of this block. In addition, the bubbles (i.e. text messages) are interactive too. When the end-user developer clicks on each bubble, the project manager opens the respective project element which triggered the message in the run-time output console.

Additionally, based on the authoring of domain visual programming language elements, the domain author can define alternative user interfaces of the messages by replacing the bubbles. For example, input could be a form of element(s) completion. This functionality is possible thanks to the API provided by the Console Output component which enables functionality to adapt input and output messages. Moreover, the domain author is able to define input/output domain visual programming language elements by adding extra I/O devices (e.g., gamepad, Joypad,

microphone, camera etc.) with their respective third-party libraries according to the application domain requirements.

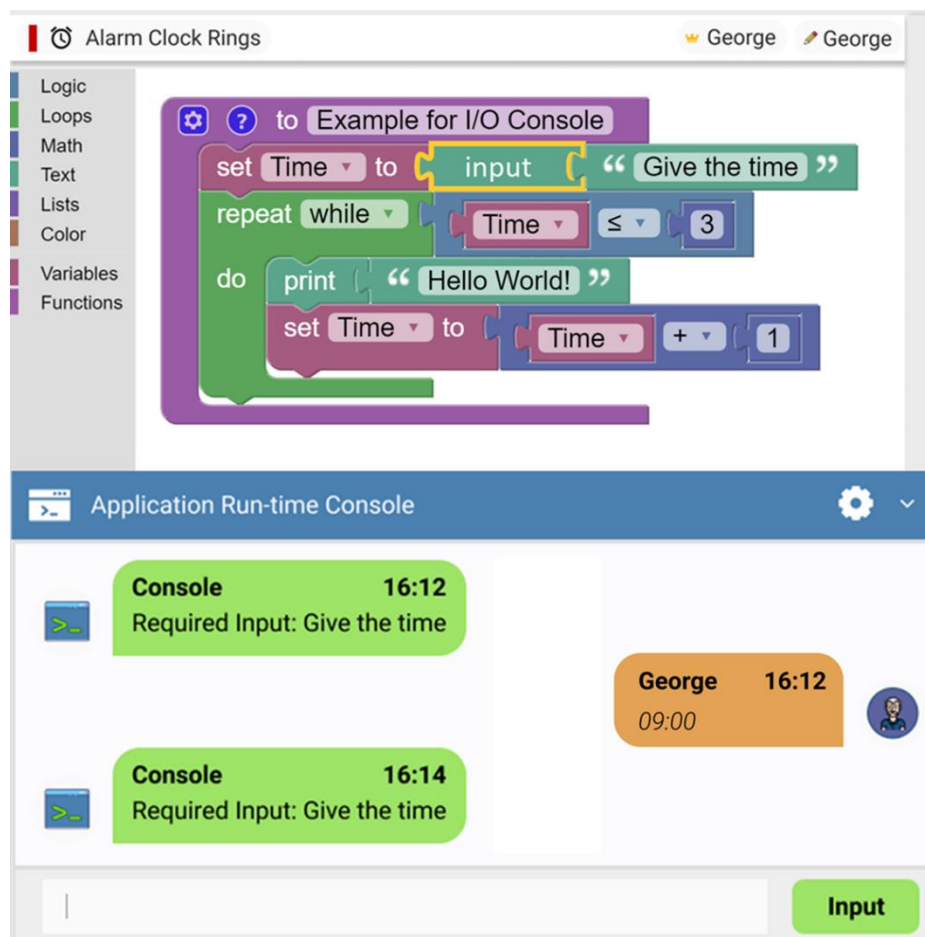


Figure 6.3. Console input is enabled and the corresponding block is browsed.

6.4 Hosting User-Interface of Application Domains at Runtime

The domain author may want to add extra input-output view component(s). For example, in case of GUI Application, the domain author would want to present the GUI of the application during the project execution. In case of a Game Application, the domain author would want to present the game view during the project execution process. Finally, in case of the personal automations in the Internet of Things, the domain author would want to add view components for the state and control of the smart objects which will participate in the application.

The domain author could define one or more domain views as components that will be initiated and hosted during the project execution. The workspace provides the

required empty div elements. The domain author is responsible for handling these components in the run-script by using the provided empty div elements.

6.5 Exporting Project to an Isolated Application

The end-user would like to export the project to an isolated application. This would require from the domain author to develop an additional appropriate script as the entry point of the application. This script would include only function of “*StartApplication*” with input argument the project data (i.e. application structure with code generated parts).

Furthermore, the authored visual programming language elements would require to develop separate definitions of code generation in case there is interaction between the application and the IDE. In this context, there is an optional field in the authoring “*exportGen*” that will be used by the code generation process instead of “*codeGen*” or “*debugGen*” which are defined in the domain visual programming element. Moreover, in case of I/O actions the domain author could use the Console Output that is provided by the IDE by incorporating this component to the application or could develop an alternative approach based on the “*exportGen*”. Finally, the workspace would export the package of the JavaScript sources including the defined entry script without adding the code instrumentation for the project execution in the context of the *Blockly Studio IDE*.

Chapter 7

Debugger

"Programming allows you to think about thinking, and while debugging you learn learning."

- Nicholas Negroponte

Debugging is the systematic process of detecting and fixing bugs within software programs. Visual Debuggers are the core tools for debugging process that are provided by the IDEs. These tools include facilities for tracing source code, viewing memory of the programs (e.g. variables, data structures, etc.), browsing the call stack of the function calls, etc. In the context, of visual end-user programming for our IDE, we have developed a full-scale visual debugger including the facilities of classic visual debuggers and aiming to support novices with extra features in order to boost them for accomplishing the debugging process.

The Visual Debuggers are separated in two main components, the front-end (debugger) and the back-end (debuggee). The front-end of visual debugger encourages the user to test and debug programs by enabling step by step control of execution, handling of breakpoints, and monitoring values of variables. The back-end of visual debugger is computing the application or a process which a debugger acts. We are following this approach in case of the *Blockly Studio IDE*.

In the context of visual debuggers for IDEs, front-end components cooperate with the text editor component is used in order to handle the breakpoints, highlight source code lines and view memory of values on mouse over the respective source code. In the case of *Blockly Studio IDE*, the front-end debugger communicates with each of the visual programming editors that are registered.

In the following subsections, for each of the visual debugger's facilities, we analyze the functionality of the visual debugger's front-end and back-end in order to accomplish them.

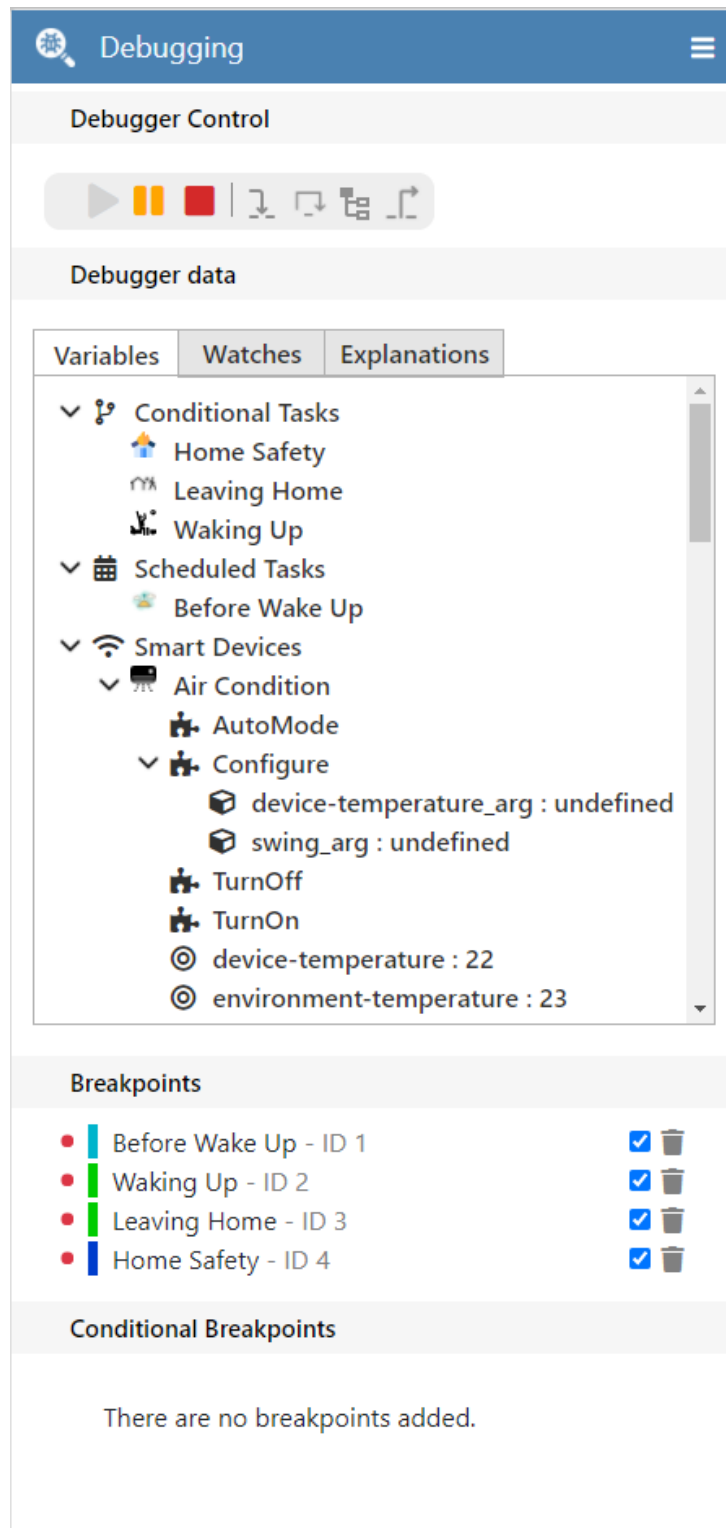


Figure 7.1. Debugger's Toolbar.

7.1 Initiating the Debugging Process

When the user starts the debugging process by selecting the debug tool item from the runtime environment, the front-end debugger is initiated by loading the toolbar.

Afterwards, the front-end debugger component communicates in order to start the debuggee. The runtime environment requests from the project manager component to retrieve the debugging environment data for the project by asking for each of the responsible visual programming editors to generate source code or data for the debugging process.

Similarly, to the project execution (see section 6.1), the debuggee is executed in another execution context from the *Blockly Studio IDE*. In this context, it retrieves the appropriate domain authored *debug-script* and calls the *'StartApplication'* function. In order to accomplish the communication among the debugger and the debuggee, the code generation for debugging injects appropriate code snippets between the generated source code. Executing these code snippets, checks the debugger state, refreshes the debugger toolbar, etc.

7.2 Debugger's Toolbar

When the debugging process starts, the debugger's toolbar appears in the visual programming workspace (see the Figure 7.1) in the right side of the main project elements area. Using this toolbar, the members are able to view memory variables of the application and handle the features that are provided for the debugging process. On the top of the toolbar resides the toolset of handling the debug process (i.e., start, pause, stop, step, collaboration and selective debug). Below this toolset, the toolbar is separated into three different rows. In the first row of the toolbar displays the watches, the variables and the explanations. In the second row the breakpoints are located, while in the third-row conditional breakpoints are shown. Each of these parts are discussed in the following sections.

7.3 Breakpoints

One of the most important concepts that supported by debuggers is handling the source code points in which developers would like to pause in order to monitor the state of the projects which is known as *breakpoints*. In source-level (text-based) debuggers, breakpoints are inserted per line, left to the editor area, usually at a special column reserved for custom icon annotations by the programming tools of the development environment. It is usual that such annotations are inserted by the bookmarker, source manager, IntelliSense, and the debugger frontend.

In the case of the visual programming editors, breakpoints are inserted per visual programming language elements that have to be supported. In this context, typical breakpoint icons have to be injected in visual programming elements. In case of the Blockly Editor we designed a typical breakpoint icon, located on the top-left of each of one of the Blockly blocks as presented in Figure 7.2. In the same logic with breakpoints in text-based visual debuggers, there are different views respective with the state of the breakpoint. The state of breakpoints can be enabled or disabled, while once an enabled breakpoint is hit, it is highlighted.

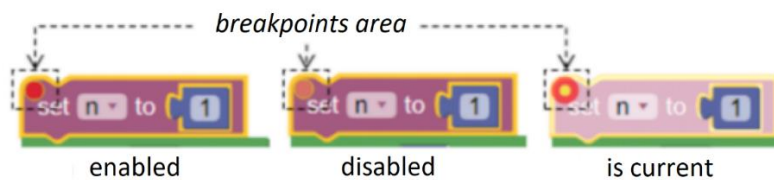


Figure 7.2. Breakpoint icons for Blockly Editor.

Moreover, adding new breakpoints in the visual programming language elements is provided by the visual programming editors. This could be done either using their toolbars or using right click on the visual programming language elements. In case of the Blockly editor, we developed this option by using right click options. The rest of the handling user actions (i.e., enable, disable, delete a breakpoint) are provided by the visual debugger toolbar (see Figure 7.1) and communicating with the respective visual programming editor instances in order to sync the information of the breakpoints. However, the visual programming editors are able to provide them. The visual debugger provides appropriate API that can be used by the visual programming editors to handle the breakpoints.

The association of breakpoints to individual blocks is implemented on top of the *Blockly* as follows: Internally, *Blockly* exposes the actual object reference of every single block. This is actually a well-documented and standard feature of *Blockly* library. We use it to directly associate, as part of the breakpoint manager, the block references to their breakpoint state. Then, as part of the code instrumentation, the code generated per block is decorated to post an event both to: (i) the *Blockly* library, with a request to highlight the block; and (ii) the breakpoint manager, to test if a breakpoint is hit – if the latter is true, meaning a stop point is met, execution will break and a trace command will be expected by the debugger User-Interface so as to proceed.

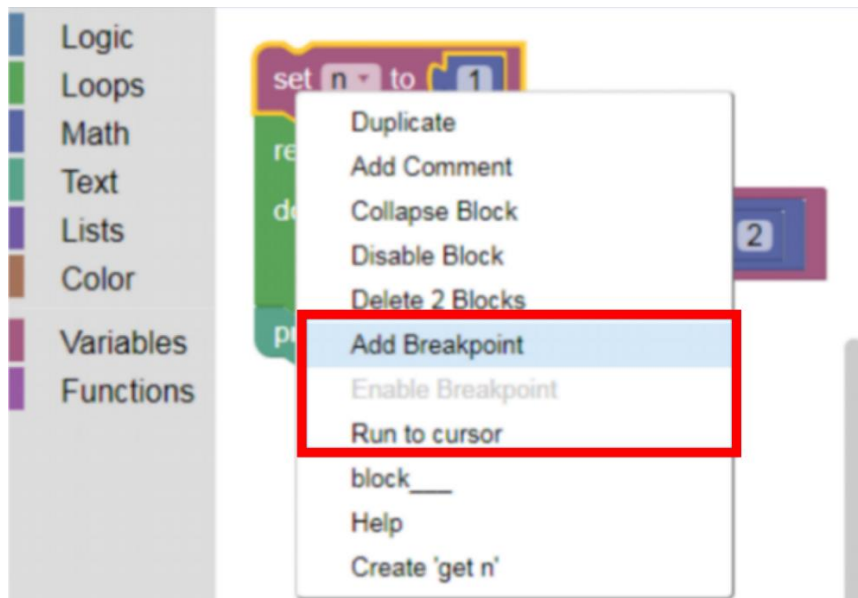


Figure 7.3. Handling breakpoints by right clicking on *Blockly* blocks.

7.4 Conditional Breakpoints

The breakpoints are related to the general-purpose visual programming editors. In the context of specific-domain visual programming editors that are specialized on handling the visual domain elements, there is not source flow in order to control where to stop. However, in this case, there data objects and the end-users may like to be notified when a field of the data object changes. In this context, we have developed conditional breakpoints for the visual programming language elements.

The conditional breakpoints are triggered on change value of visual programming domain element property or on get specific value of a property etc. Moreover, the end-user will be able to choose how many times will be activated the breakpoint observer and/or begin to be activated after N times, pause execution in case something not happens in specific time etc. Using these breakpoints, the end-user developers will be able to view the memory variables and the state of the applications when specific domain element property changes, while they will be notified for the history of the domain element property values.

The conditional breakpoints have been developed thanks to the information of data objects are handled by specific domain visual programming editors. In particular, as we have already mentioned (see section 4.2.1), the domain-specific visual programming editors notify the domain manager system for the data objects that are constructed and handled. In this context, the front-end debugger retrieves total data

objects that are created in the end-user development process. Using this information, the dialogue which handles conditional breakpoints generates the selections of the visual programming elements and their properties (see Figure 7.4).

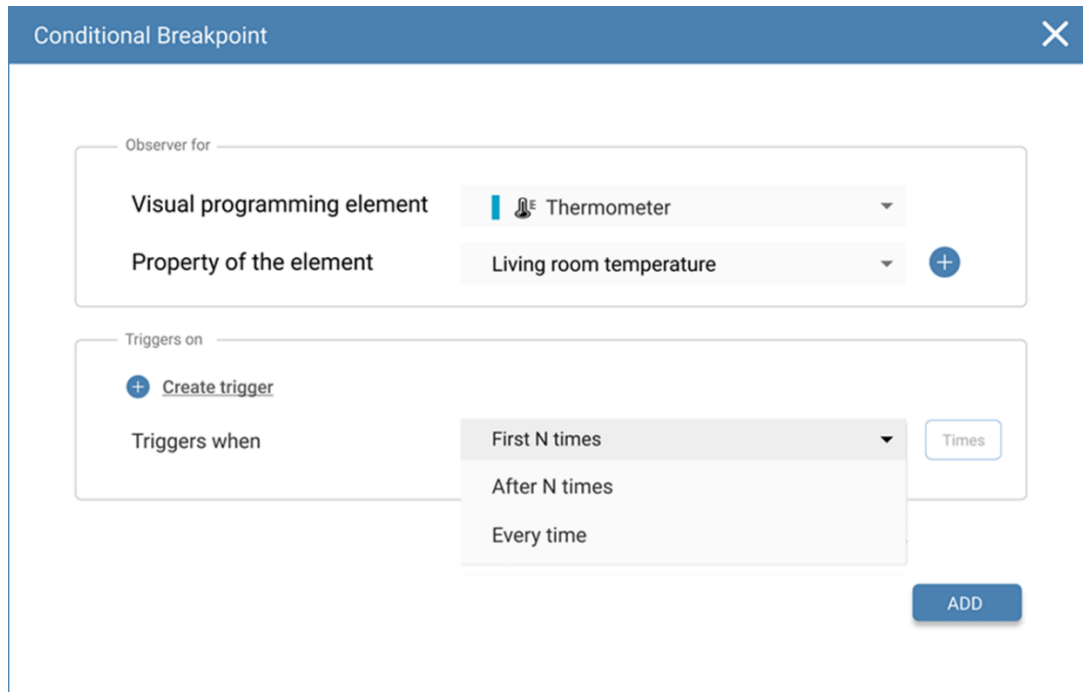


Figure 7.4. Conditional breakpoint's dialogue.

In order to address the conditional breakpoints, we have injected an extra code snippet per visual programming language statement that iterates each of the enable conditional breakpoints and checks if there are triggered by using function calls that are developed in the debug-script. In case there is triggered conditional breakpoint, the debugging execution pauses and opens a dialogue that informs about the previous and the current value.

In addition, the end-user developer is notified if this change happened by the project's visual code execution or by external factors (e.g., in case of the domain of mobile application, mobile sensor changed by the environment, in case of graphical user interfaces, the user pressed a button, etc.). In case change has been caused by the visual code, there is available link to browse and highlight the visual programming language element, opening the respective project element and the visual programming editor instance.

7.5 Tracing

As already pointed out, visual programming elements tracing is functionally similar to source-level tracing, however, with a few important differences. The first variation concerns the basic “*Step In*” and “*Step Over*” commands. These two operations, originating from source-level debuggers, control whether a function call expression is evaluated thoroughly (Step Over), or if the execution progresses by evaluating all actual arguments and then by stopping into the first instruction of the invoked function (Step In). In our case, besides this behavior regarding function invocations, these commands work as follows given a current visual programming element during debugging: “*Step In*” stops in the first inner (child) visual programming element, and “*Step Over*” enters the next sibling visual programming element. Otherwise, if no inner or sibling visual programming elements exists, they stop in the next executing visual programming element, following the control flow. Interestingly, these variations are possible due to the hierarchical structure of code, enabling users skip entire visual programming elements of visual code during tracing, something not possible when using typical source-level debuggers. In particular, in order

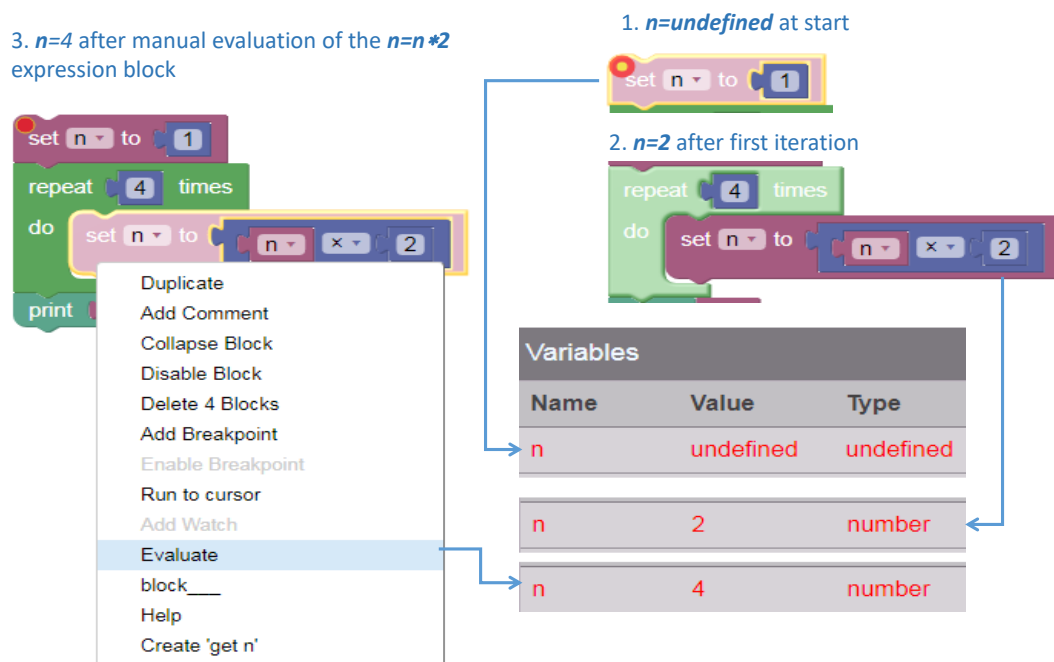


Figure 7.5. Automatic variable inspection and the Evaluate operation which works for any kind of block, enabling to re-evaluate on-the-fly (during debugging) any code snippet.

programmers to skip entire visual programming elements of text code, they would have to either use the “*Run To Cursor*” command or, alternatively, place temporary breakpoints and then use the Continue command. However, for nested expressions this far from straightforward: positioning the cursor in a single line or setting a breakpoint is not precise enough to trace particular subexpressions, unless the source code is manually reworked to place one such subexpression per line.

In case of *Blockly* Figure 7.5, the behavior of “*Step In*” and “*Step Over*” is shown once execution meets a breakpoint in Figure 7.3. In this example, the expression $n=n*2$ is actually split in two blocks: the outer assignment block and the inner multiplication block. The latter allows, as shown in Figure 7.3, to separately evaluate $n*2$ with a “*Step In*” command, something not possible directly with a typical source-level debugger. The same mechanics apply to the “*Run To command*” as well, which works for the currently selected block and will cause execution to stop exactly before evaluating this block.

7.6 Watches

Inspecting program variables, commonly known as watches, is also in two ways. Via the variables pane, showing all variables at the current scope (sometimes designated as autos in various source-level debuggers), and the watches pane, in which inspected variables can be added or removed during debugging by the user through the visual debugger’s toolbar (see first two tabs in tag 2 of Figure 7.1).

As earlier discussed, for each project element is included a list of visual sources that are handled by specific visual programming editors. Each of them may include variables that would like to display them. In this context, the front-end debugger provides API in order to add, edit, disable, etc. the information view of program variables. This API can be used by the code generation process of the visual programming editors when a variable change.

In case of *Blockly*, all variables reside in the global scope, thus used throughout the entire visual program, meaning the presence of the watch pane is somehow redundant. However, it is still possible in *Blockly* to implement a custom block type for the declaration of a local variable, simile to the let specifier of JavaScript. In this case, autos will enumerate only the local variables at the current block scope, and watches

will show the particular user-chosen variables. In our implementation, if no local variables exist, the variables window automatically displays all global program variables. In Figure 7.5, the automatic display of program variables is shown in a debugging session, in a simple example program involving a single n variable.

Besides variable inspection, it is possible to manually evaluate entire *Blockly* blocks, something being more flexible and expressible than typical expression evaluation. For instance, in the example of Figure 7.5, at Step 3, the manual reevaluation of the current block is chosen. This is actually an extra evaluation with respect to the normal program execution. As a result, the expression $n = n*2$ is executed once more, causing n to gain 4 value, meaning it is also allowed to change program variables via watches. Concluding, note that the same logic is able to be followed for all general-purpose visual programming editors.

In the case of watching the domain visual programming language elements, the domain author is responsible to use the provided visual debugger's API in order to display their values during the debugging process.

7.7 Execution Snapshots

Non-programmers are not experienced in the debugging process. As a result, the visual debugger has to empower them with extra features. In this context, we developed the history of variable values.

Debugging the application during the end-user development process in order to verify if it is working as expected, the end-user is able to browse the execution flow history of the visual programming elements. In this context, the end-user developers are able to view the history of the values for each of the visual programming language elements. In particular, by clicking on specific visual programming element, the end-user programmer is able to view the watches values, the visual programming elements had at a specific execution time. In addition, the end-user is able to view all the history of the visual programming element values. However, this is not a straightforward process, due to the classic visual debuggers that enable the current values of the programming elements.

In order to solve the above issue, we have added extra decoration code per visual programming statement. This decoration code requests an execution snapshot for all

the programming elements which are watched. In addition, the decoration code saves information (i.e. visual source and visual programming element ID) for the visual programming element currently executed. Thanks to this technique, the debugger saves the history of the watched values and provides the backwards and forwards browsing of the project execution.

7.8 Explanations

In order to give an extra weapon in the end-user debugging arsenal, we have introduced new visual programming elements that could be helpful in the debugging process. In particular, we have defined a new category of *Blockly* blocks named as “*Explanations*”. These blocks can be used by the end-user developers in order to explain what will happen or happened in the above or below visual code instructions they develop.

Using these blocks, relative messages can be posted in the input-output console of the IDE during the execution. However, there is the option to choose when the block is executing to pop up dialogue, pause the project execution and display the message instead of post it in the input-output console.

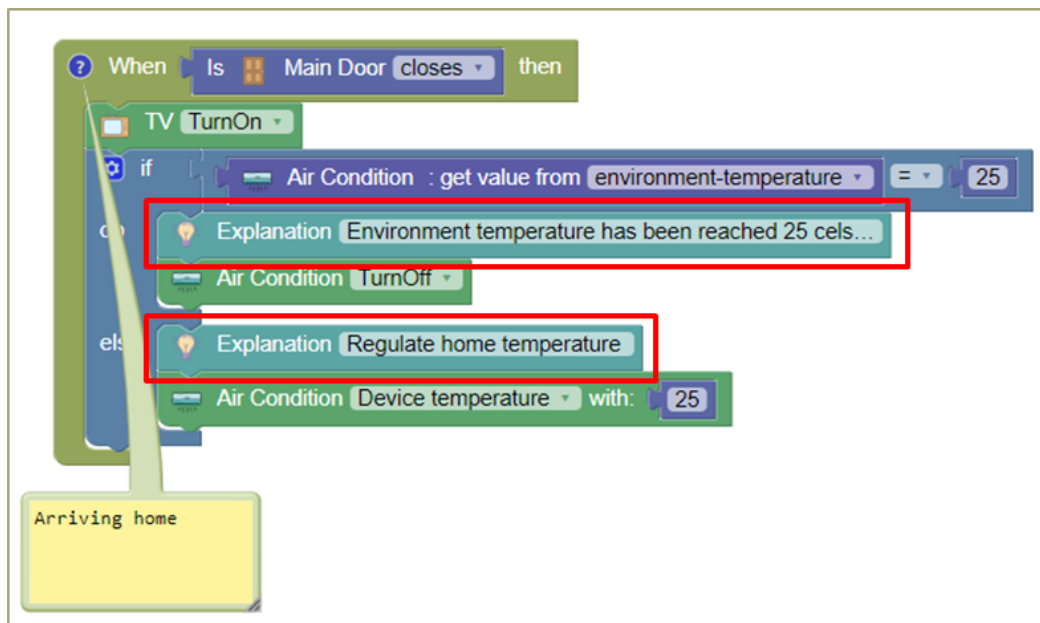


Figure 7. 6. Adding explanations for the execution of smart automations based on the environment temperature.

The handling of these blocks is hosted by the visual debugger’s front-end (see tag 2 of Figure 7.1) in the last tab. In addition, the history of the executed explanation blocks

is displayed during the debugging process. Furthermore, the end-user developer can handle them through the Blockly editor workspaces as classic blocks. Furthermore, there is ability to activate or deactivate them in the release project execution.

7.9 Supporting Debugging for Application Domain Frameworks

As described in this chapter, a debug-script has to be developed for the debugging process of a specific application domain framework. Similarly, with the runtime environment, this script is the entry point of the debug execution by initiating the application domain libraries, the global variables, etc. The source code generation for the visual programming language elements differs in order to inject the extra required information for the debugging process and the extra communication and checks that are required in order to accomplish the visual debugger's functionality.

Additionally, debugging information may have to be initiated. Moreover, there are cases in which the release runtime environment may differ with the debug mode. In particular, in case of the mobile applications, the developers debug and test their applications not in mobile phones but in mobile phone emulators. In this direction, sensors and properties of the mobile phones have to be displayed by the variables view of the visual debugger. In order to accomplish this requirement, the visual debugger provides API to define the information data that will be shown during the debugging process. Another example could be the debugging process of smart automations in the Internet of Things, which is impractical to test the automations in smart devices and sensors (see section 9.7). However, using the debug-script, extra software infrastructure could be developed and used that will be utilized for specific application domains.

Chapter 8

Remote Collaboration

“None of us is as smart as all of us.”

- *Will Harvey*

We consider that collaboration is a key feature in end-user programming and could be notably useful in the case of teaching and learning purposes, asking for help from more experienced users, co-working for automations etc. The later makes it important for groups of end-user developers to have suitable tools to support their collaborative programming tasks. Our motivation to extend the *Blockly Studio IDE* in order to provide a full-scale collaboration toolset in the context of end-user development is based on the aforementioned fact. Our approach focuses on two directions, the collaborative editing and the collaborative debugging.

8.1 Collaborative Editing

In this section, we present the full-scale collaborative editing facilities for end-user development process that are developed for the workspace of the *Blockly Studio IDE* (see Figure 8.1). In our approach, we focus on sorting out of the editing process by introducing peer roles, access and edit privileges for project elements. Additional features include: personal project elements, toggling live syncing during editing, viewing peer action history, and enabling local execution without disrupting the collaboration session. Moreover, through several settings that are exported, our approach enables the domain authors and the end-user developers to configure them in order to accomplish their requirements based on the circumstances of their end-user development process. In the following sub sections, we present each of them, we analyze the collaboration models that can be supported. Finally, use case scenarios and the conduct of an evaluation process are discussed.

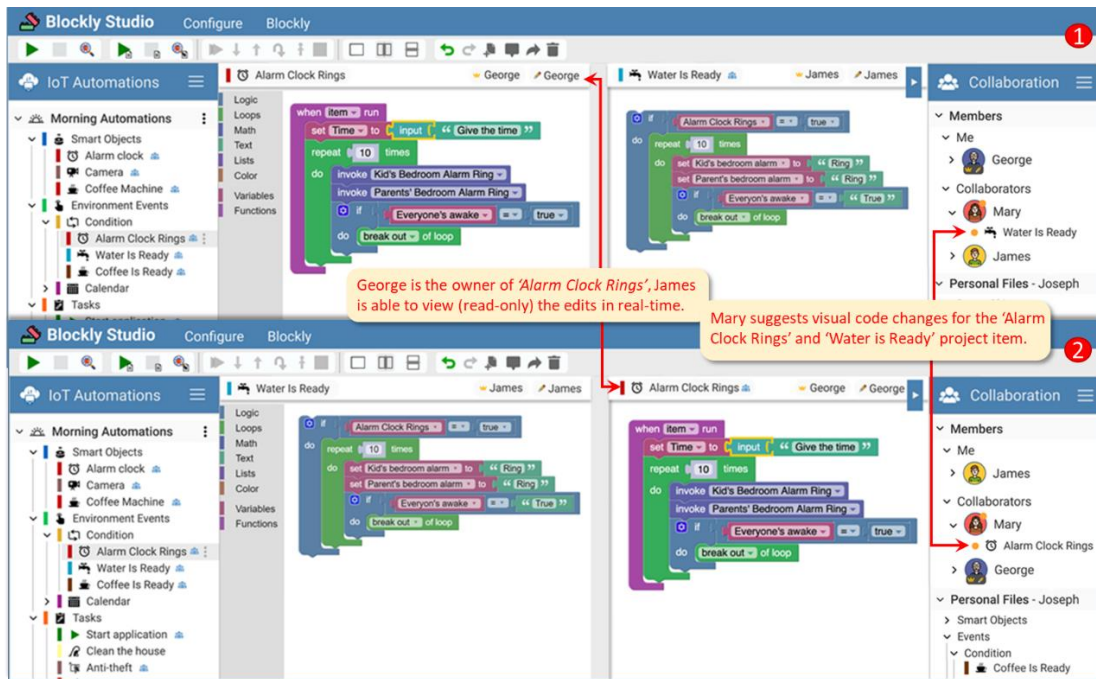


Figure 8.1. Collaborative Project “Morning Automations” with 3 participants (George, Mary and James). George’s view of the collaborative project (see 1) and James (see tag 2).

8.1.1 Peer Roles

The first step of the collaboration process is the agreement among a group of people for co-working in the end-user development project. The main data that are distributed in each collaboration session is the shared project, on which the members are working. In order to sort out the shared data we identified roles for the participants. The first-class subject of our approach focuses on the members that join the collaboration session of the shared project. We are aiming to better organize the collaboration among the members, so we have introduced roles for each of the participating members.

The lead role of the collaboration process belongs to the master of the shared project. In the beginning of the process, this role is given to the end-user that shares the project. The master has full access privileges in all the project elements of the shared project. Furthermore, the master gets decisions for the development process of the project. In particular, other members request to add or delete shared project elements and the master replies to these requests. However, the master has the option to configure the requirements of the requests for actions in the shared project elements (i.e., no request needs, disable the ability to add/delete actions from other members

etc.) according to the circumstances of the collaboration process. Additionally, the master is allowed to delegate his role to another member.

The second role is the owner of each project element. Only one member is qualified to edit a shared project element. In particular, we consider that co-editing of visual sources in parallel using multiple cursors is working well in the documentation and the design tools. However, in the context of end-user development, we strongly believe that it is a first-class subject to organize and structure the projects in small scale project sources, that would be easier for the end-user developers to handle. For this purpose, it is pointless to provide the ability of co-editing. Moreover, supporting parallel co-editing of sources could cause confusion among the members [141]. The author of the shared project element gets the role of the owner. Then, during development, the owner is able to transfer the edit privileges to another member. Furthermore, the master of the project is qualified to get the edit privileges of any shared project element.

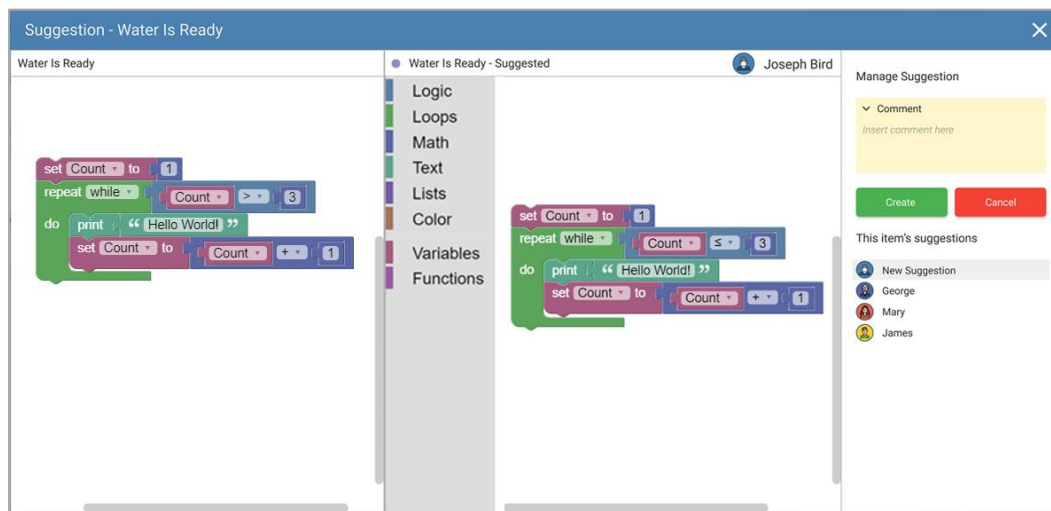


Figure 8.2. Dialogue to create new visual code correction suggestion for a project element.

In addition, the members are able to request authoring new shared project elements or add existing ones from their local project elements. Also, they are allowed to request for edit privileges from the existing shared project elements. Furthermore, in case they don't have edit privileges of one project element, they are able to add notes and correction suggestions for visual code changes (see Figure 8.2). Adding a suggestion, the owners of the project elements are notified and are qualified to accept or deny the changes (see Figure 8.3). However, based on the circumstances, the master of the

project is able to disable the notes and/or the corrections suggestions for all or for specific collaborator members. Moreover, features for communication (i.e., instant chat or video calls) are not provided. We consider that several software tools could be used for communication and it would be pointless to embed communication software technologies in the visual programming workspace.

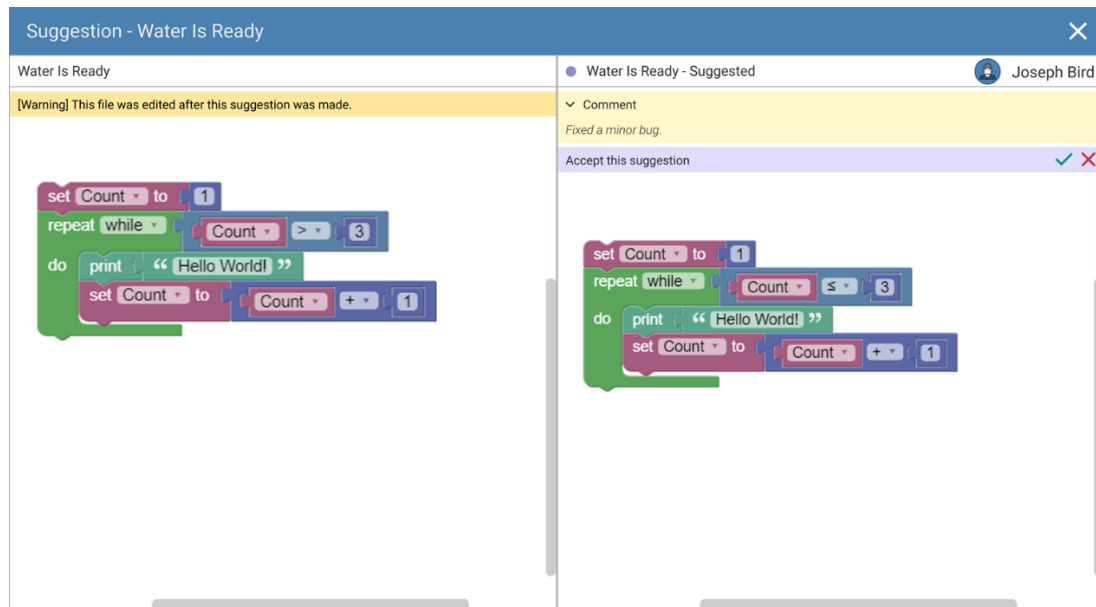


Figure 8.3. Dialogue to view the visual code suggestion in order to accept or deny it.

8.1.2 Local Workspace

One of the most important domains of the collaborative editing is the work that will be able to do each of the collaborator members without affect the productivity of the others by developing and testing their tasks in the shared project. In this context, we developed facilities by focusing on the local workspace.

8.1.2.1 Personal Project Elements

The members are able to create new personal elements in their local workspace which are merged with the shared project. However, these project elements are not parts of the shared project. The personal project elements are used for member's local testing as drafts of their end-user programming tasks. Potentially, they could be added as shared project elements later in the development process. Furthermore, the members are allowed to view personal project elements of other members. However, the master is qualified to choose if these project elements will be viewed or not by other members due to the collaboration circumstances. For example, a teacher may would

like to set personal project elements to be private from students for their assignments. In addition, the master chooses whether other members will be allowed to create personal project elements or not. Moreover, the members are the masters of their personal project elements. They are able to share them to one or more specific members. They are qualified to give privileges of editing and cancel them.

Table 1. Project Element Privileges.

Project Element	Author	Owner	Shared	Hidden
	<i>Member</i>	<i>Member</i>	<i>Enumerated</i>	<i>Boolean</i>

Concluding, the privileges of the project elements are summarized in the above table. The information of the “*Shared*” column is enumerated among the not shared, shared in project and shared personal project elements.

8.1.2.2 Toggling Live Syncing

Our proposed approach supports real-time collaboration which means that members view live changes of other members by default. This is extremely useful for members that are following the process. However, the members have the option to deactivate real-time syncing of the shared project elements they manage and/or the other shared project elements. By disabling real-time syncing, we allow the end-user developers to test their changes locally without any waiting from other members. However, in case the members would like to give editing privileges, they have to sync the specific project element or to revert the changes until the last synced state.

8.1.2.3 Selective Project Execution

Running the project during the development process in order to verify if it is working as expected is one of the main tasks. In real-time collaboration process, testing the shared project could be unmanageable for the members. Specifically, the development progress between the shared project elements may differ. Some members may have finished their tasks, however other members may haven’t. As a result, the members have to wait for other members in the development process.

In order to solve the above issue, we extended the aforementioned feature of selective project execution (see section 6.2) in the context of collaborative editing. In particular, starting the execution process, the default choice is to run the shared

project. However, the end-user could choose which project elements will be included in the execution process as happens with not shared project as well. In addition, as an advanced choice for more experienced users, it is allowed to replace shared project elements with personal project elements or choose to replace an original project element with its corresponding suggested changes of visual code. Selective execution allows the end-users to run the application partially which means that testing the functionality of their tasks will be easier.

8.1.3 Initiating Collaborative Sessions

In order to start a new collaboration session, the end-user needs to share a project and configure the aforementioned settings based on the requirements of the collaboration (see Figure 8.5). Using the modal depicted in the left section of Figure 8.4, the users have to fill-in their personal information which will be viewed by other collaboration members. A unique URL is generated and the users are able to notify the members they would like to join by sending them this URL. When the users join the shared project, they will be asked to fill-in their personal information too (right section of Figure 8.4).

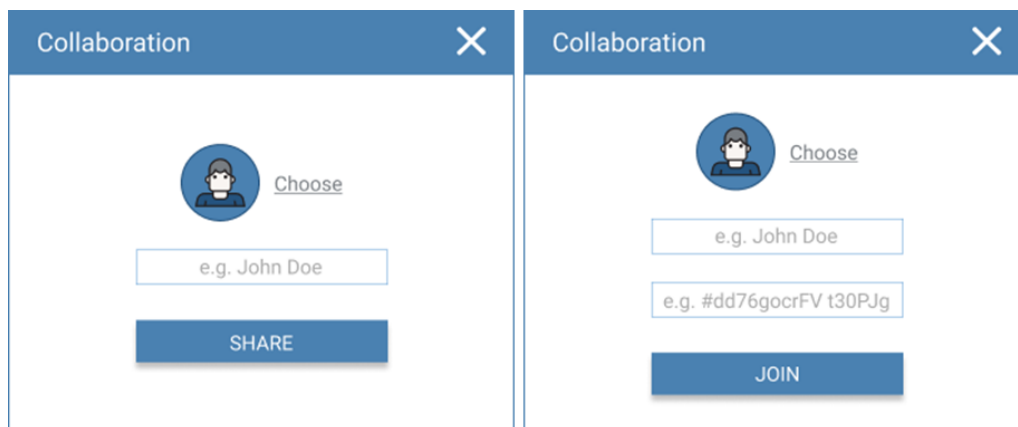


Figure 8.4. Left: Starting share the project; Right: Joining the collaboration.

In the beginning of the collaboration session, the visual programming workspace components are visually and functionally configured. In particular, the project workspace manager user interface is modified in order to visualize and separate the shared project elements from the personal project elements by using appropriate tags next to the titles. In addition, the shared project elements that are updated by other members and the user hasn't read yet are marked with bold style until the user

browses them. Moreover, menu items have been added for each project elements, based on the member's role and the edit & access privileges in each project element. Additionally, thanks to VPL editors' area splitter, the end-users are able to work in their files and view (read-only) the others' files in parallel (see Figure 8.1).

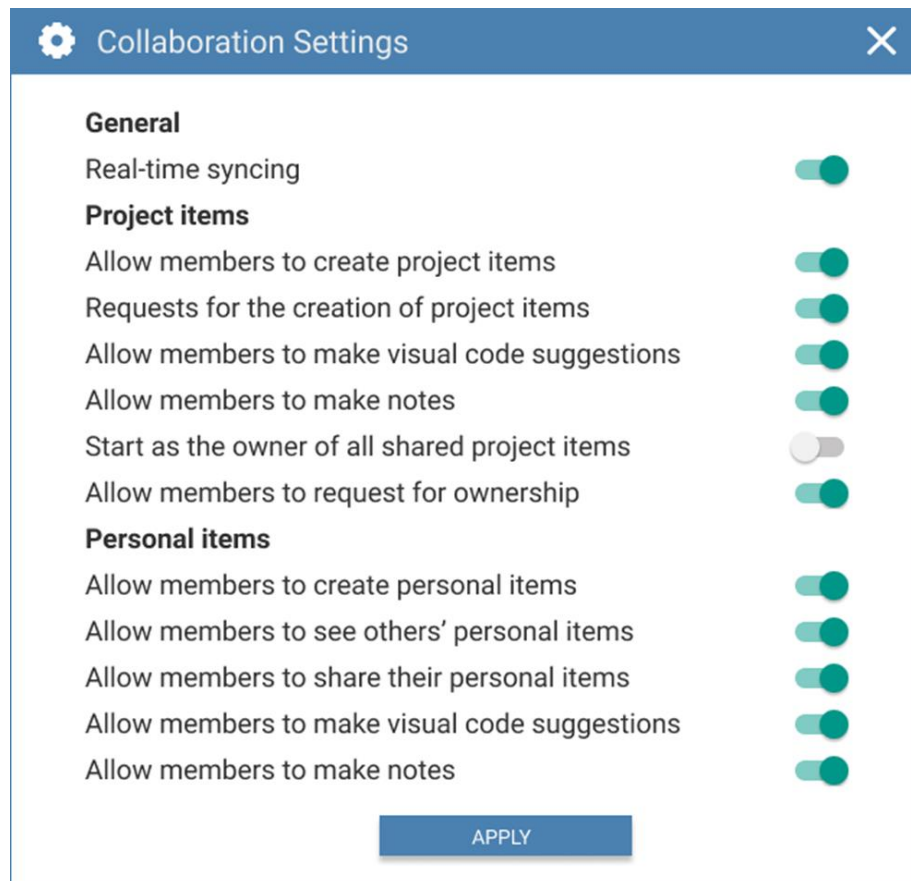


Figure 8.5. Collaboration project settings.

8.1.4 Collaboration Toolbar

The collaboration toolbar appears in the visual programming IDE's workspace (see the Figure 8.6), when the collaboration end-user development session starts. Using this toolbar, the members could view and handle data about the collaboration process. The toolbar is separated into four different rows of information. The first row of data is the main presented information of the toolbar and displays the collaborators of the project. Furthermore, the decisions for requests and correction suggestions of each member are displayed below the personal information of the member (e.g. see the highlighted '*Water is Ready*' element of Figure 8.6). Moreover, the end-users are allowed to transfer the ownership of the viewed project element in case they are the

owners of the specific element. The second row of the toolbar visualizes the personal project elements of a member. When the user selects another member from the above list of members, this specific member's personal project elements are displayed. In case no members are selected, the users are able to view their personal elements isolated from the shared project. However, the master chooses if this information will be visible or hidden for the members. The information regarding handling the shared personal project elements is visualized in the next row of the toolbar. These files are

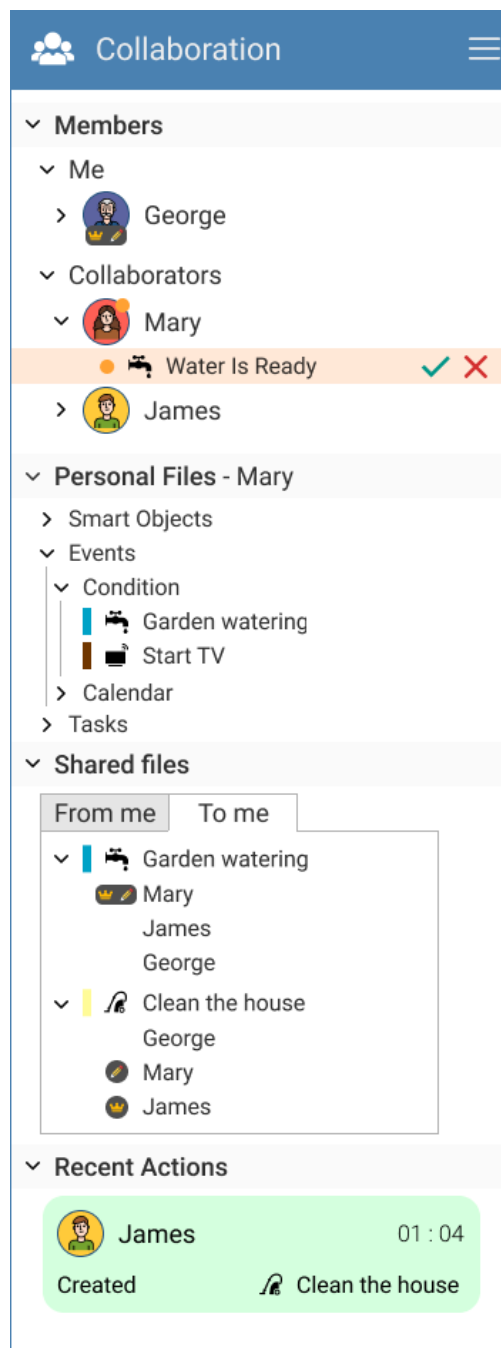


Figure 8.6. Collaboration Toolbar.

separated into two tabs, the first includes the personal elements shared by the user and the second includes the personal project elements that are shared to the user by other members. In the bottom of the toolbar, the actions history of the collaborative end-user development process is visualized as chat message bubbles and each member is able to browse them.

8.1.5 Supported Collaboration Models

Applying the above roles and rules in the shared project, our approach is capable to support Pair programming in one or more groups of members that have joined the shared project. In the beginning of the collaboration session, the master of the project is able to organize members in groups, comprising of the driver and the navigator for specific end-user development tasks.

Moreover, our approach could be used in the context of a teaching classroom. The teacher would be the master of the project and the students would be simple members that could watch the teacher develop in the context of a lesson. Then, the teacher would be qualified to assign development tasks to the students as assignments of the lecture.

In addition, this approach would work well for the collaborative development of applications in groups of small teams (e.g. friends, family etc.). In the context of personal automations in the Internet of Things, such projects could be automations for a family's smart home etc. Furthermore, less experienced end-users could ask for help and share their artifacts with more experienced users or professionals. The applications of collaboration are better represented through the use case scenarios that are described in the following section.

8.1.6 Evaluation

When our work led to well-formed requirements and implemented most of the collaboration facilities, we decided to evaluate our proposed system in the context of collaboration for the end-user development process. In order to assess our approach, we conducted an evaluation study on users. In this section, we discuss the aims and design of our study, present the use case scenarios, outline the evaluation's participants, describe the evaluation process and analyze the results.

8.1.6.1 Aims and Design

The evaluation we conducted aims on observing how users operate and use our system's key features as well as on assessing the system's usability. Particularly, we dedicated our study's focus to evaluating the collaboration toolset. For each collaboration feature that we considered important, we designed a use case scenario that focused on deciding whether the chosen approach was indeed appropriate and well-integrated. For obtaining usability measurements, we used the System Usability Scale (SUS).

8.1.6.2 Use Case Scenarios

We use hypothetical users to discuss the scenarios. Each of the following use case scenarios are separated in two parts, the description and the goal. The description of each scenario refers to the real-world situation that contextualizes the goal. The goal of each scenario refers to the task that should be accomplished. The scenarios' contexts are realistic and the goals are kept simple and short in order to evaluate the usability of specific features of our approach for collaborative editing. The used scenarios are following.

1) *Starting a new collaboration session*

Description: George has bought a new Smart TV and wants to configure it but unfortunately has little to no experience. However, his sister Tina has programmed smart devices in the past and can help him.

Goal: The participants were asked to create a project and start a new collaboration session.

2) *Handling suggestion requests, opening a personal project element*

Description: Bob's grandparents need help for setting up their alarm clock, pill reminder and water heater. For that purpose, Bob has created and shared a project with his family members. His family members have finished working with the project elements "Alarm Clock" and "Pill Reminder" and have suggested them for inclusion. However, his brother hasn't yet made any correction suggestions for the water heater and Bob wants to check on his progress.

Goal: The participants had to accept inclusion requests for the two project elements. Furthermore, the participants were asked to locate a specific user's personal file and open it in the editor.

3) *Creating a new personal project element and asking for inclusion*

Description: Alice has joined her teacher's project, in which she is instructed to create a new personal project element for controlling the class' air condition machine. Once her code is ready, her teacher has instructed her to make a request for the project element to be included in the shared project.

Goal: The participants were asked to create a new personal project element named "Air Condition" and make an inclusion request for the project element to be included in the shared project.

4) *Exchanging the editing rights*

Description: Mike is currently configuring his new smart refrigerator along with his friend, Adam. Mike is facing difficulties and Adam offers to help. For that purpose, Adam asks for the editing rights. After Adam's contribution, Mike retakes the editing rights to continue coding.

Goal: The participants were asked to pass the editing rights of an existing project element to another user. On success, the participants were asked to regain the editing rights.

5) *Suggesting changes for a project element*

Description: Laura notices a certain error in a project element named "Coffee Is Ready". However, the person in charge of the file is currently busy and cannot give her the editing rights. In order to eliminate the error, Laura adds a correction suggestion that contains the correction suggestion for that specific project element.

Goal: The participants were asked to add a correction suggestion for a specific file.

6) *Selective execution of a project element*

Description: John is working on a project with Maria and Peter. John has finished his assigned work and wants to test his code. However, Maria and Peter are still working on their assigned parts, which means that the project is not on a stable state.

Goal: The participants were asked to execute their own project element isolated from the rest of the project.

7) *Configuring the options of the shared session*

Description: Jake is a teacher and wants to setup a test for his students. In order to do that Jake creates a shared project and makes sure students cannot cheat by configuring the shared project's options.

Goal: The participants were asked to create a new project, share it and configure its options so that personal project elements are not visible to simple members.

8) *Sharing personal project elements*

Description: In order to keep his collaborative project in a stable state Joseph is working on a personal project element. However, he is facing issues and asks his collaborator, Mark, for help.

Goal: The participants were asked to share a personal project element to another user and pass the editing rights.

8.1.6.3 Participants

We asked 18 participants (M = 13, F = 5) aged between 14 and 31 to help us. Most of the participants were from our university departments (i.e. Computer Science, Mathematics and Physics). Additionally, 6 of the participants were high school students that have previous experience with Scratch. Moreover, we found 2 individuals that had no previous experience with programming or visual programming.

8.1.6.4 Process

Each participant was evaluated individually. We firstly discussed and presented the classic *Blockly Editor*. Then, we presented our visual programming workspace for *Blockly* and afterwards the collaboration end-user development toolset. Next, each of the aforementioned use case scenarios was described to the users and they were asked to interact with the prototypes in order to accomplish each task. For each task and participant, we measured the time required for completion and we recorded the user behavior. Finally, the users were asked to fill-in the questionnaire which is presented in the *Appendix*.

8.1.6.5 Results

We summarized and further analyzed all the answers given from our participants. The SUS questionnaire was designed in order to export results in two main dimensions. The first was focused on the collaboration end-user development efficiency and usability (see Table 2). Results showed that the vast majority of participants were satisfied with the collaboration toolset. Furthermore, the second dimension was focused on the application fields of use (see 0). Most of the users considered the tool useful for teaching or learning purposes and would use it for their collaborative projects.

Table 2. Efficiency and Usability.

	SD	D	N	A	SA
Q1. The collaboration component is well integrated into the workspace.	0	1	2	8	7
Q2. I find the collaboration process unnecessarily complex.	8	9	1	0	0
Q3. I find the collaboration user interface intuitive and easy to use.	0	0	2	10	6
Q4. I feel confident using the application with guidance.	0	1	0	11	6
Q5. I can use the application in the future without any help.	0	1	4	7	6
Q6. The collaboration toolset offers limited options.	8	7	3	0	0

Furthermore, based on the aforementioned measurements we constructed the following diagram that visualizes the average, the best and the worst time recorded for each scenario. All the users completed the tasks and most of the worst time measurements are not far from the average, while the best are not far from the average too. Moreover, during the evaluation, we realized that after the 3rd scenario, most of the users were more familiar with the tool. The latter is also depicted in the decreased time to complete equally difficult tasks.

Table 3. Fields of Use.

	SD	D	N	A	SA
Q1. I would like to use the tool for my personal projects with my family/friends.	0	1	2	9	6
Q2. I don't see the point of collaborating.	12	5	0	1	0
Q3. I find the application useful for teaching and learning purposes.	0	0	2	7	9

Finally, based on the free form questions (*Appendix*), we summarized that participants were generally satisfied. In addition, some of them commented that the application would be useful to ask more experienced users for help remotely in *Blockly*. Moreover, they were satisfied by the tool’s user interface, however, they spotted some design mismatches that we fixed (e.g., missing *URL* information after sharing process).

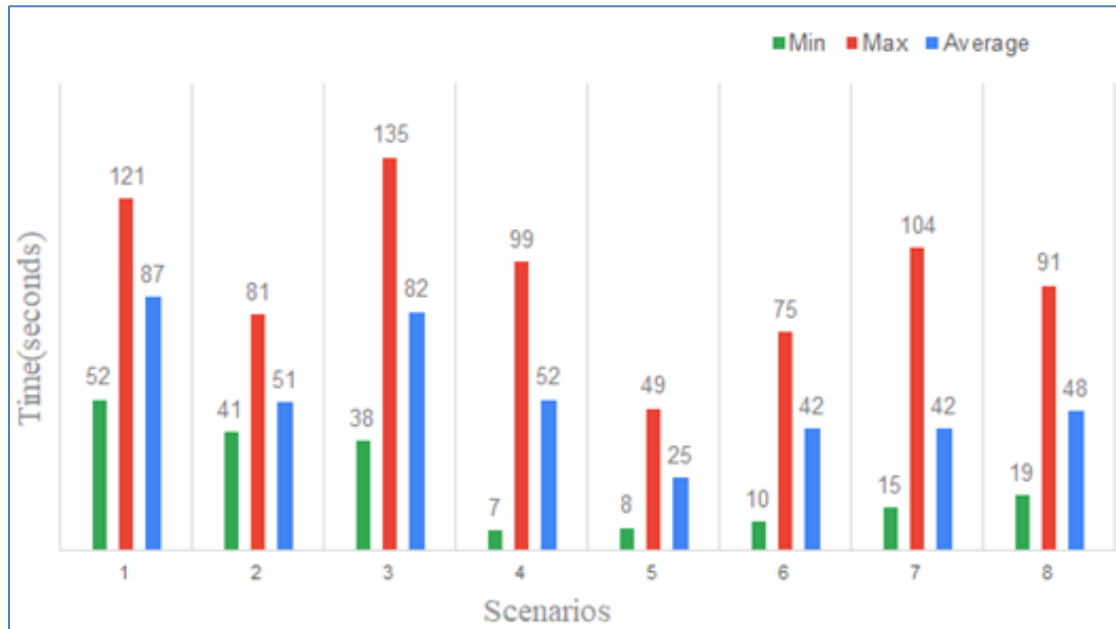


Figure 8.7. Participants' time to accomplish each of the scenarios.

8.2 Collaborative Debugging

Debugging is one of the most demanding activities in the software development process. In the case of novices, the debugging process could be extremely challenging or problematic [142]. Moreover, the debugging process for novices can be more efficient through pair collaboration [143]. Additionally, learning debugging programs is essential for novices. In this context, we consider that collaborative debugging is a key feature in visual end-user programming and could be notably useful in the case of using it for teaching and learning purposes, asking for help from more experienced users, debugging collaborative projects etc. This motivated us to extend the *Blockly Studio IDE* in order to provide a full-scale collaboration debugging toolset. Our approach (see Figure 8.8) focuses on two directions: firstly, on efficiently supporting

the collaborative debugging process among the end-users and secondly, on providing the infrastructure for teaching and learning debugging.

In the first direction, we facilitate debugging and testing for novices in the context of the end-user development process. The tool can be used for personal end-user development projects (i.e. asking for help from other users which are more experienced) or collaborative end-user development projects (i.e. debugging project that will be developed by more than one end-users) hosted by the *Blockly Studio IDE*.

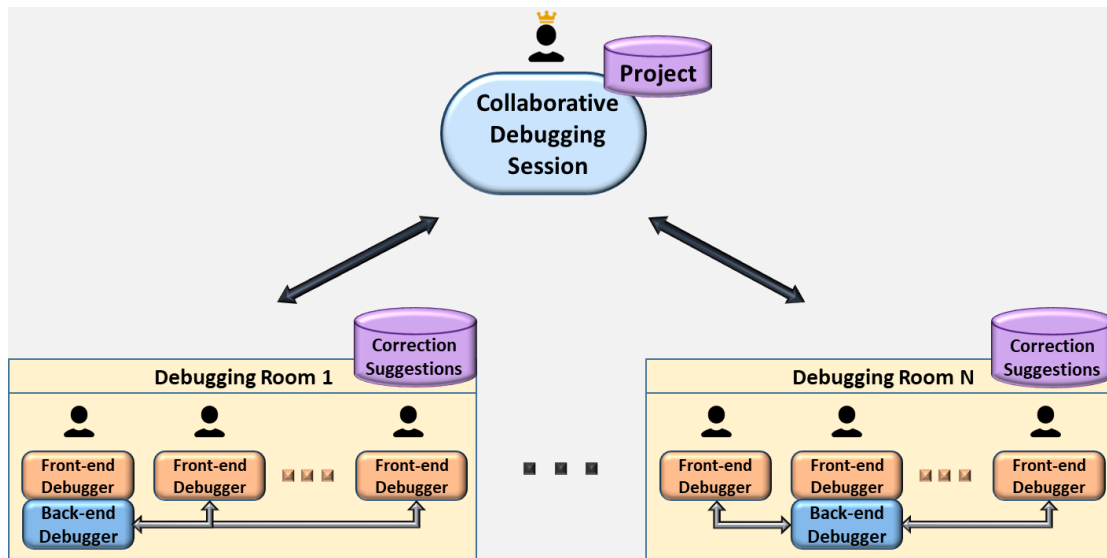


Figure 8.8. High level of our collaborative debugging approach.

The collaboration approach we propose, guarantees the preservation of the project’s visual code by isolating it, creating a local replica for each one of the collaboration members. In this context, the users are able to create correction suggestions per project element. Those correction suggestions are shared among the participants. In this context, at the beginning of each debug session, we allow the end-users to select which version of visual code will participate in the project execution. During the debugging session, one user at a time is able to handle the debugger instructions (i.e. master of the debug session). However, the rest of the members (i.e. observers) are able to navigate the visual code to acquire information independently of other members browsing, without interfering with the experience of any collaboration member. In addition, they are able to add breakpoints to stop the execution at crucial points and use watchers in order to view values independently. Finishing the collaborative debug session, the end-users are given the ability to decide which of the correction suggestions will be applied to the original project and may choose to

include them in the original project as suggestion visual code corrections without applying them. The latter could be useful for new ideas which may arise during the collaborative debugging process.

In the second direction, we aim to contribute to teaching and learning in the context of debugging and programming. The tool can be used by teachers (i.e. masters of the collaboration session) to demonstrate the debugging process to students (i.e. other members of the collaboration session) in real-time. In such demonstrations, the students will be able to perceive the flow of a program and learn the process of debugging. Additionally, the tool enables students to live debug programs, individually or collaboratively while allowing the teachers to supervise all the debugging processes. In particular, the collaboration approach we propose, introduces debugging rooms. The users are able to create debugging rooms in which debugging sessions can be hosted (i.e. one debug session per room at a time). Other members may be permitted or forbidden to join a debugging room and the correction suggestions may be visible to anyone or only to the members of the debugging room. The master of the collaboration session is able to customize the facilities based on their needs. In this section, we present the proposed collaborative debugging approach.

8.2.1 Initiating Collaborative Sessions

In order to start the collaborative debugging session, the end-users have to share the project that they would like to debug. This project can be either a personal project or a collaborative project which is already shared with other users. Using an appropriate modal, the users have to fill-in their personal information which will be viewed by other collaboration members. In addition, they have to configure the settings of the collaboration based on the requirements of the collaborative debugging (i.e. teaching-learning or debugging purposes). A unique URL is generated and the users are able to notify the members they would like to join by sending them this URL. When the users join the shared project, they are asked to fill-in their personal information too via a similar modal. In the case the end-users start a collaborative debugging session for a collaboration project, there is already personal information from the collaboration editing session. This information is used and the end-users are prompted to just confirm the beginning of a collaborative debugging session. In particular, the

collaboration members from the collaborative editing session are notified and asked to choose if they would like to join the collaborative debugging session or not. Moreover, the collaborative debugging session's unique URL is available in case they would like to invite additional members. Members are able to join the session and catch up on the collaborative debugging process at any time, as long as the session is active.

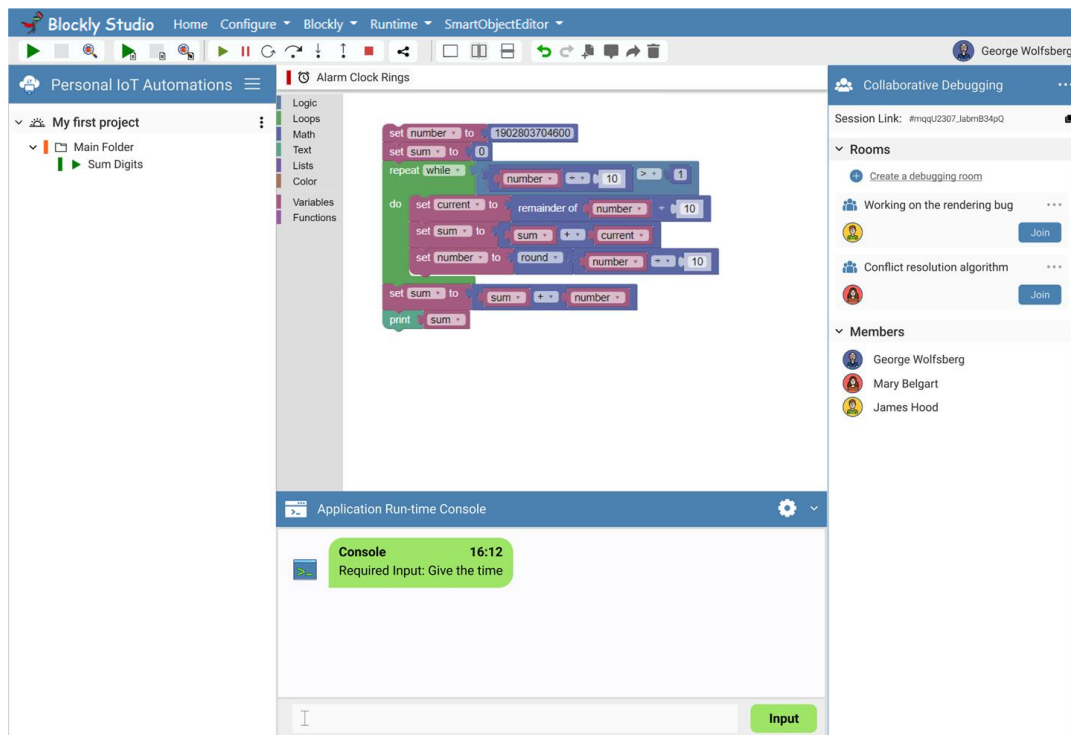


Figure 8.9. Starting view (i.e., home page) of the collaborative debugging session.

After joining the collaborative debugging session, the end-users are able to browse the project using the project manager (see right of Figure 8.9). The collaborative debugging toolbar is located on the right of the IDE. The collaborative debugging toolbar includes the members which participate in the session, the debugging rooms, the correction suggestions and the current member actions logger. In the next paragraphs, we analyze each one of them.

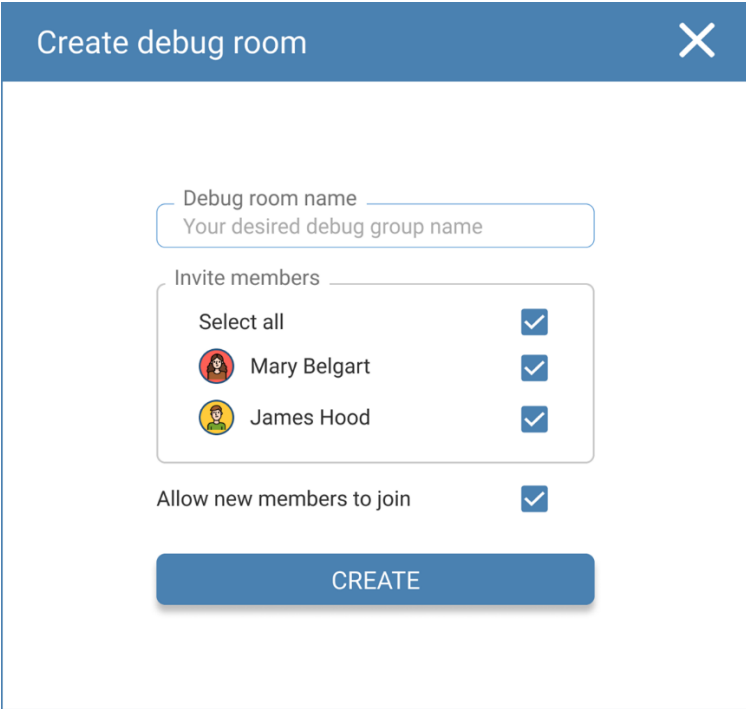
8.2.2 Debugging Rooms

In the collaborative debugging session, the members by default run one debugging process at the time. In particular, a single user is allowed to handle the debugger commands (i.e. master of the debug session). The other members of the team (i.e. observers) are able to navigate the visual code to acquire information independently

of other members browsing, without interfering with the experience of any collaboration member.

In addition, they are able to add breakpoints to stop the execution at crucial points and use watchers in order to view values independently. Moreover, at any time during the session, the master can switch roles with one of the other users. This is often useful when the execution reaches parts of the code with which the master is not familiar; the master may then decide to become an observer and promote a more knowledgeable collaborator, who takes the lead without having to start over with a new debugging session. Furthermore, the end-users are allowed to deactivate the master mode (i.e. deactivates the functionality which restricts that only one member is able to handle the debugger) through settings, in the case they consider that it is tedious to exchange the visual debugger control.

Moreover, the collaborative debugging environment doesn't provide an embedded voice chat. We consider that such tools are not part of the collaborative debugging environment and the members are able to use a third-party communication tool. However, the members are able to view the recent user actions logger (see on the bottom right side of Figure 8.9). Moreover, they are able to add notes for each one of the project items which will be accessible to the other members.



The image shows a modal window titled "Create debug room" with a close button (X) in the top right corner. The modal contains the following elements:

- A text input field for "Debug room name" with the placeholder text "Your desired debug group name".
- A section titled "Invite members" containing a list of members with checkboxes for selection:
 - "Select all" with a checked checkbox.
 - Mary Belgart with a checked checkbox.
 - James Hood with a checked checkbox.
- A checkbox for "Allow new members to join" which is checked.
- A large blue button labeled "CREATE" at the bottom.

Figure 8.10. Modal to create a new debugging room.

However, instead of one team, more teams are required in the case of teaching and learning. In particular, the tool enables the teachers (i.e. masters of the collaborative debugging sessions) to demonstrate the debugging process to students (i.e. observers of the collaborative debugging session) in real-time. In such demonstrations, the students are able to perceive the execution flow of a program. Also, they can be educated by teachers on debugging techniques and practices. The tool enables students to live debug programs, individually or collaboratively while allowing the teachers to supervise all the debugging processes. In this context, we introduce debugging rooms in our collaborative debugging environment.

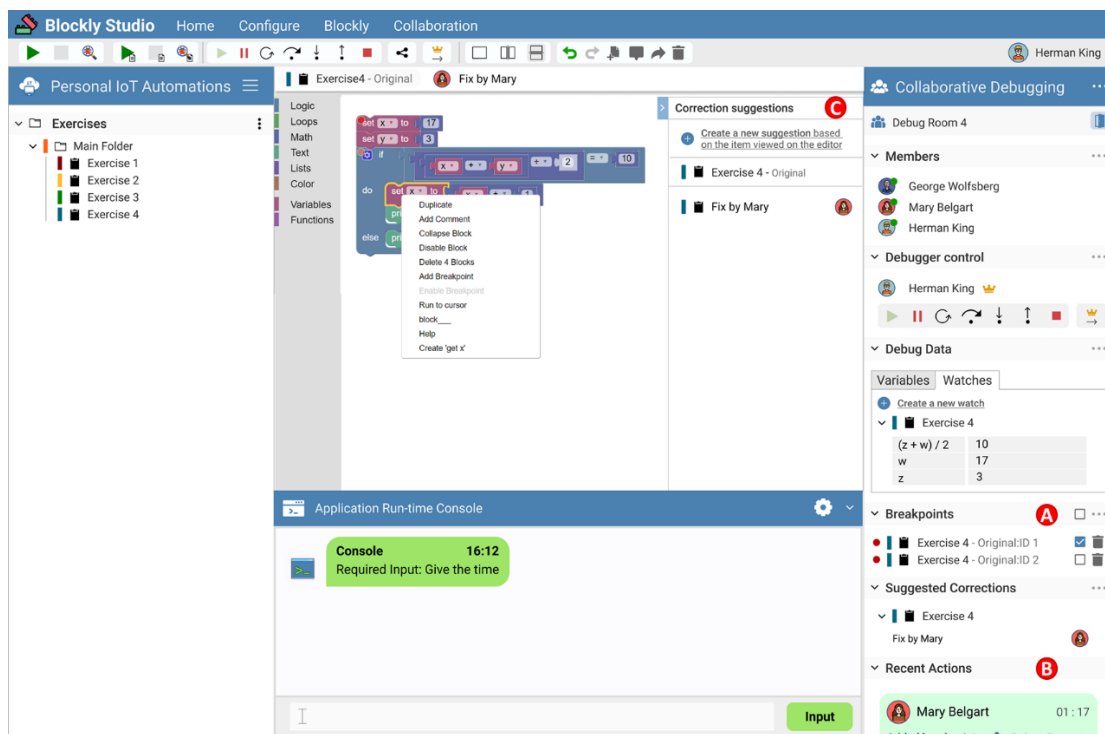


Figure 8.11. Viewing 'Debug Room 4'.

The users are able to create debugging rooms in which debugging sessions can be hosted (i.e., one debug session per room at a time). For the creation of a new debugging room, the author has to specify the name of the room and the members that will have access (see Figure 8.10). Based on the circumstances, the master of the collaborative debugging session decides whether the other members are able to create debugging rooms or not. The users are allowed to be members of more than one rooms but are only able to browse one room at a time.

Joining the debugging room, the end-user developers see the current state of the room as depicted in Figure 8.11. In particular, they are able to catch up on the debugger session state if there is an ongoing collaborative debugging process in the debugger room. In this context, they are able to view the visual debugger information including instructions control (i.e., step in, pause, continue, etc.), debug data and breakpoints (see tag A of Figure 8.11). The user can also view which users are members of the debugging room and which of them are active (see green bubble on George's icon of Figure 8.11). Additionally, they are able to view the current member actions of the debugging room and the debugging session (see tag B of Figure 8.11). Furthermore, they are able to view the list of the correction suggestions and may create new ones (see tag C of Figure 8.11). Moreover, the owner of a debugging room and the master of the collaborative debugging session are able to remove a member or destroy the room.

8.2.3 Visual Debugger

As earlier mentioned, we have developed a full-scale block-level visual debugger for *Blockly* into the *Blockly Studio IDE*. This visual debugger is full-scale block-based featured, offering all of the classic stepping and inspection (watches) features in analogy to source-level debuggers. In particular, control actions such as start, pause, continue, restart, step over, step in, step out and stop are included (see Control Debug on the right of Figure 8.12). Also, inspection of variables and evaluation of expressions are available. Additionally, breakpoints are inserted per individual block, with a typical breakpoint icon, located on the top-left of the associated block as depicted in Figure 8.12. Breakpoints can be enabled or disabled and once an enabled breakpoint is hit, the correspondent block is highlighted. Using the visual debugger toolbar, the user is able to activate or deactivate breakpoints, add or remove watch expressions and use any control action for tracing.

However, in order to support collaborative debugging, the visual debugger of our IDE was extended and customized (see on the right of the Figure 8.13). In particular, the visual debugger consists of two component parts, the front-end and the back-end. The front-end includes the visuals of the debugger including the control commands, the breakpoints, the watcher, the express evaluations etc. The back-end of the debugger includes the execution of the visual source code of the project, the logic for

interacting with the front-end and the required debugger's data that have to be maintained for the debugging process. The back-end runs in a different thread from the IDE using *JavaScript Web Workers* [144] and interacts with the front-end through messages. We extended the back-end of the debugger by exporting its API in order to be able to handle messages that are posted and received via the peer to peer mechanism of the collaborative debugging session. Moreover, we add roles to visual debugger peer instances in order to identify which are the obligations of each one during the debugging process (i.e., view of the debugger control, post messages to collaborative debugging peer and/or apply them to the debugger back-end system etc.).

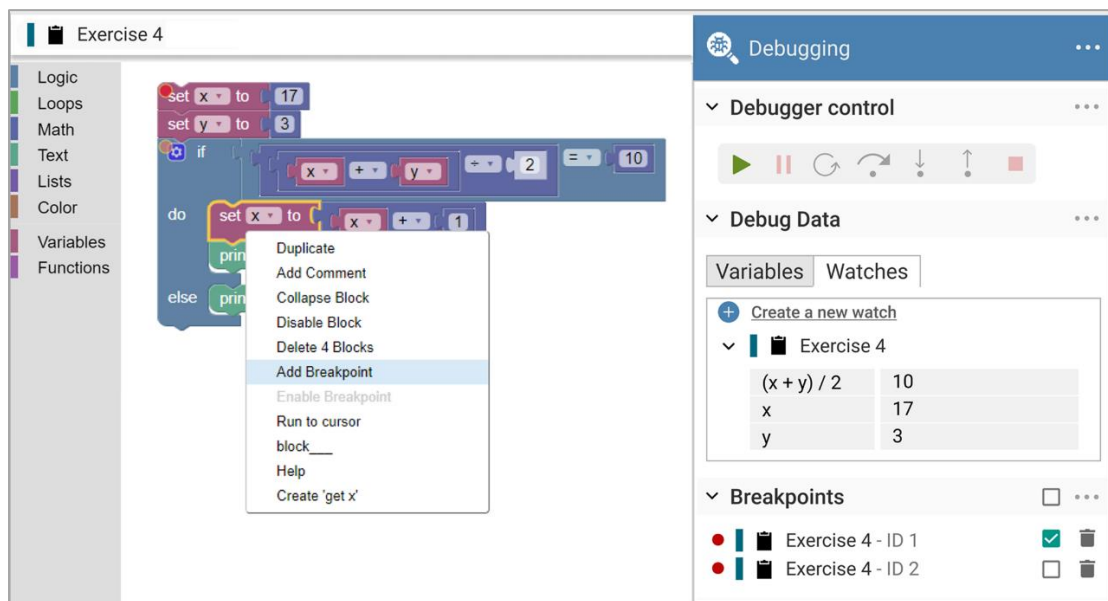


Figure 8.12. Using Visual Debugger of Blockly.

When an end-user starts the debugging process, the back-end (i.e. debuggee) of their debugger is activated by getting the master role of the process (see on the right of Figure 8.13). In order to avoid parallel debugging processes, the master visual debugger peer posts a message to the other debugger peers in order to request the initiation of the debugging process. The other visual debugger peers at first notify their front-end part to disable the debug control user interface and change debugger state (i.e. observer) and afterwards respond that they are ready to participate in the collaborative debugging process as observers. Then, the master visual debugger peer starts the code execution (i.e. only one of the visual debugger peers executes the source code) and notifies the other visual debugger peers via messages that the

debugging process started. Then, the other debugger back-end peers notify their front-end to sync the debuggers data. In this context, the visual debugger data (i.e. debug data, breakpoints etc.) are saved in the master and they are transmitted to all the peers in order to be synced.

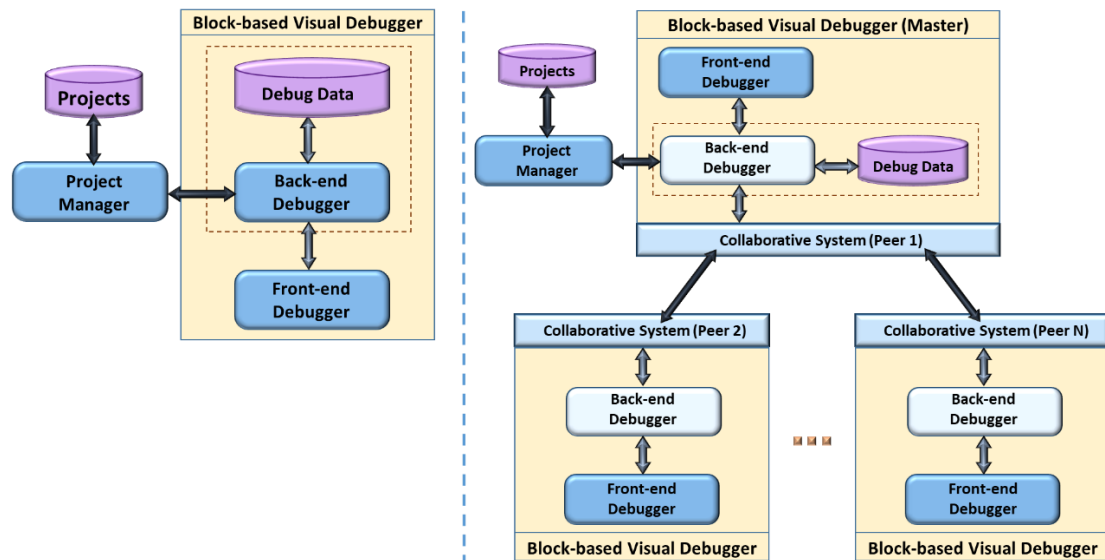


Figure 8.13. Visual debugger's architecture for classic debugger version (left), collaborative debugger version (right).

In the debugging room, the participants are able to run one debug session at a time. The end-user developer that starts the debugging session is the master of the debugger. In particular, this user handles the debug control while the other users are the observers of the process. However, the other users are able to browse the code independently, add breakpoints, watch variables and expressions of the debug process during the debugging session. Moreover, the master of the process is able to pass the role to one of the other members by clicking the extra button on the right side of the debug control toolbar (see on the left of the Figure 8.14) and choosing who of the members will acquire the debugger control. Also, the master of the collaborative debugging session is able to configure the settings of the session in order to have no restrictions on who may control the debugger. Moreover, only the active members participate in the debugging process. Additionally, the members catch up on the current debugging session state watching the current debug member actions (e.g. step-in, step-over, add breakpoints, applied expressions, members joined or left the room, etc.). If the master of the debugging process leaves the debugging room, the current debug session stops.

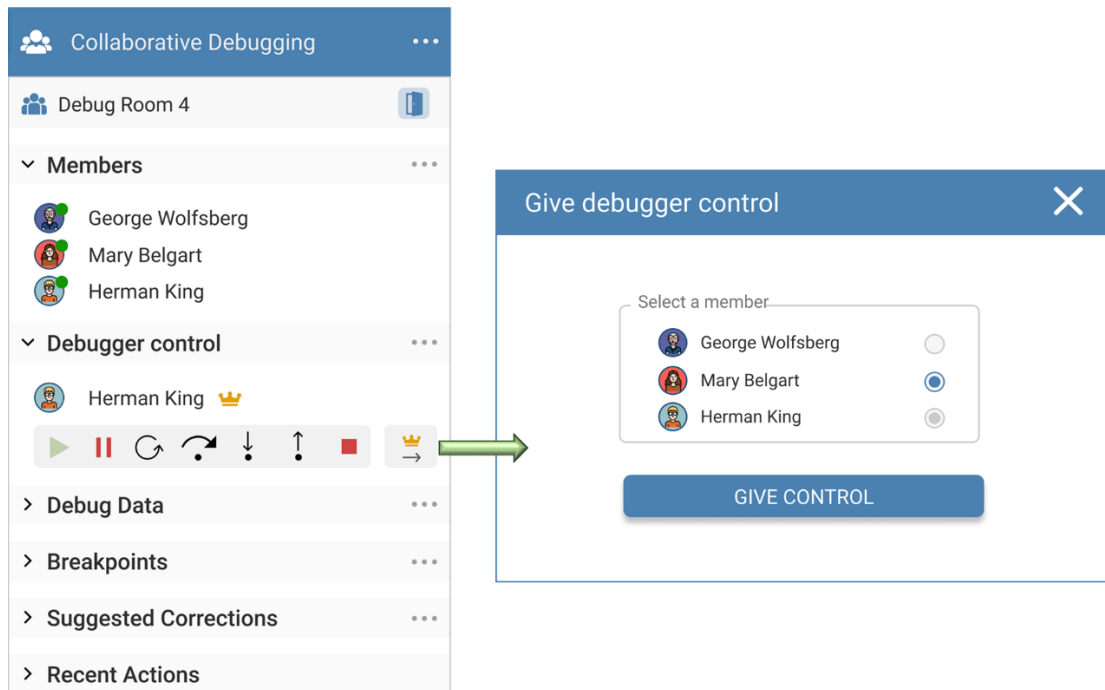


Figure 8.14. Debug Control (left); Give floor control dialog (right).

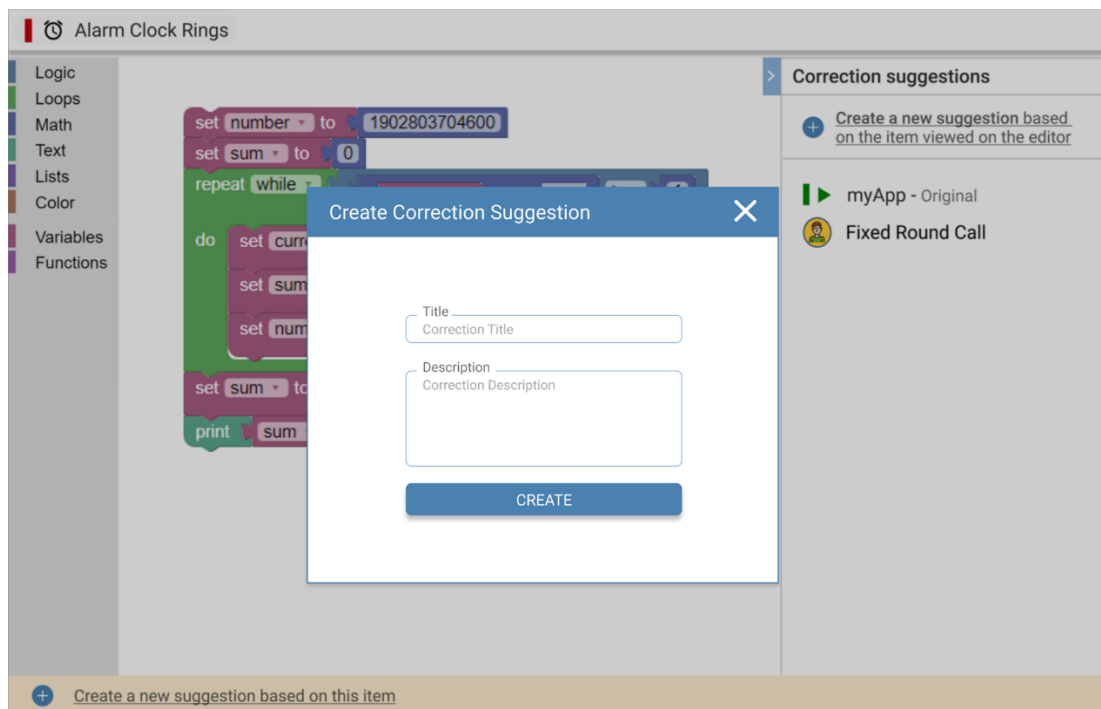


Figure 8.15. Creating new Correction Suggestion for “Alarm Clock Rings” project element.

8.2.4 Correction Suggestions

Starting the collaborative debugging session, a replica of the shared project is created for each of the members that join the session. The members are able to debug the project locally by editing the replica of the project. However, if the members would

like to apply changes to debug the project collaboratively, they have to share the changes of the project element(s). In this context, our approach introduces correction suggestions.

In particular, when the users edit one of the project items of the local project, the system warns them that in order to share the changes, they need to create a new correction suggestion. To create a new correction suggestion, the user has to choose its title and description. The other members of the debugging room are able to view the correction suggestion that is instantly added on the right of the project item, as depicted in the Figure 8.15. However, only the owner of the correction suggestion has privileges of write access. The other members are able to watch live the visual code changes the owner applies. This is necessary for the members in order to be able to browse the correct version of the visual source that is included on a debug session. Furthermore, this could be a useful tool for the teacher to present live development or ask from a student to live give a solution to a problem.

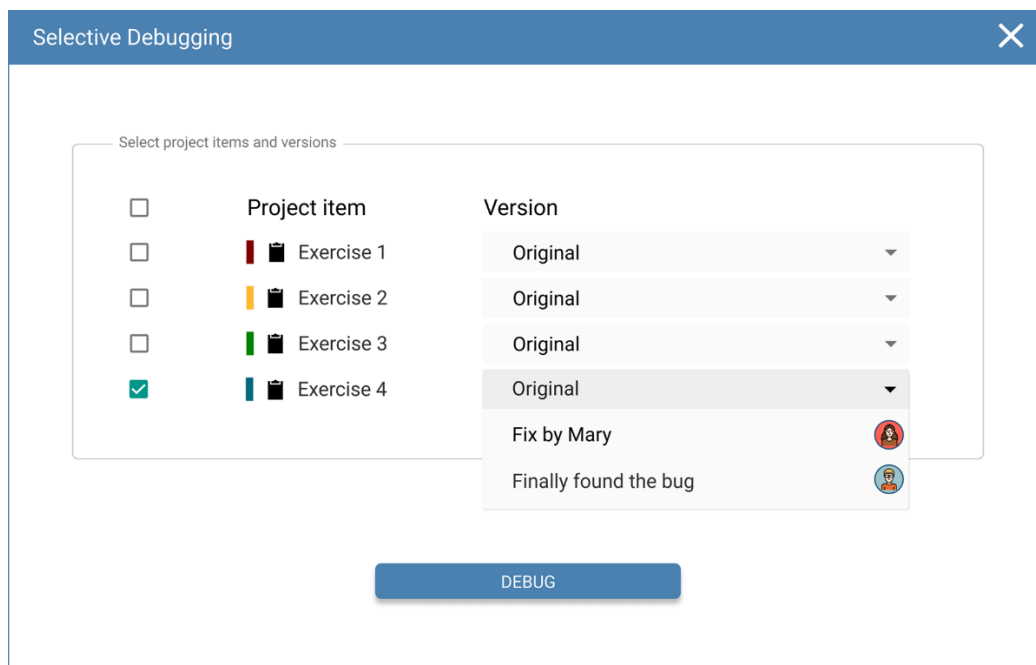


Figure 8.16. Debug the project by choosing the project items will participate and which of the project items will be original and which of them will be correction suggestions.

In addition, users are able to create new correction suggestions based on other correction suggestions independent of who their owner is. Moreover, correction suggestions can be accessible to all the members from the main page of the collaborative debugging session outside of the debugging rooms. This depends on the

settings configured by the master in the beginning of the collaborative debugging session. However, the master is always able to view all the correction suggestions in the main page.

The users are able to start the debugging session and choose which of the project items will participate in the process as earlier mentioned in the selective execution feature of the section 8.1.2.3. Furthermore, in each debugging session, the user who initiated it, is able to choose which of the project items and which of the versions (i.e., the original version or any of the correction suggestions for this project item) will be run (see Figure 8.16).

In the end, when the collaborative debugging process is completed, the master of the session is able to choose which of the correction suggestions will be applied to the original project, replacing the corresponding original project item visual sources as depicted in Figure 8.17. Moreover, they are able to choose which of the correction suggestions will survive as correction suggestions in the original project's visual code.

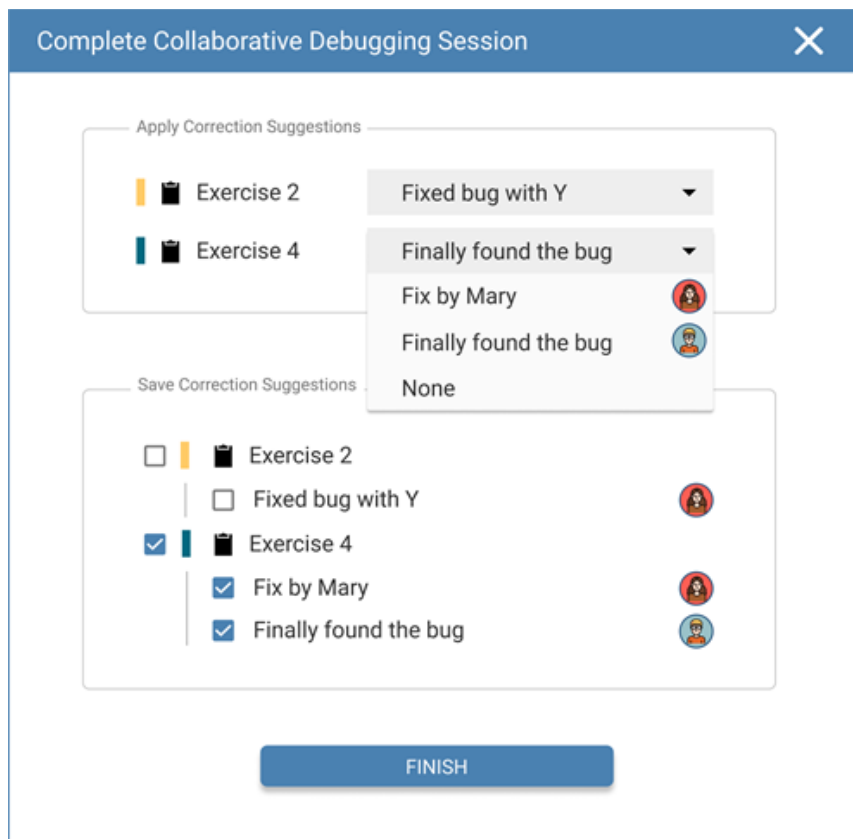


Figure 8.17. Choosing which of the correction suggestions will be applied to original project and which of them will be saved.

8.2.5 Discussion of Supported Applications

Using the above features that are provided by the collaborative debugging environment, our approach is capable of supporting four different types of applications. Firstly, in the case of novice programmers, our approach supports asking for help from more experienced users. The novices can start a collaborative debugging session in which they are able to invite one or more users to help with correcting their project's errors. In the same context, the second supported application contributes to our collaborative programming environment. During real-time remote collaborative editing, the members are able to initiate collaborative debugging sessions and debug their shared project together in real-time. The third supported application contributes to early childhood preservice teachers [145] in order to find their errors easier and more quickly. The teachers are able to set up the collaborative debugging environment in order to supervise and help the children with their programming tasks.

Additionally, an interesting supported application of our collaborative debugging environment focuses on teaching and learning debugging. Learning debugging is crucial for novice programmers. We provide a full-scale toolset for teachers to carry out live debugging sessions in order to present debugging techniques and practices to the students. At the same time the teachers are able to supervise the debugging process whether they ask from students to develop an assignment or find errors in provided visual code. To investigate our toolset's efficiency in such setting, we have carried out a user study, in which we play the role of teachers and the participants play the role of students. We discuss the empirical study in the next session.

8.2.6 Empirical Study

Debugging is a crucial process for learning and understanding programming through comprehending the execution flow of programs. Additionally, through collaborative programming, the learning process can be significantly improved. In this context, one of our main goals is to provide a full-scale toolset, capable of being used in real teaching circumstances. To evaluate our IDE's collaborative debugging environment for teaching, learning and furthermore improve its usability, we have carried out a study with novices.

The study contains a short visual-programming tutorial for the participants and a more analytical tutorial with the goal to teach the basic debugging actions (i.e. step in, step out, step over, continue etc.) and features (i.e. breakpoints, watches etc.) that the IDE's debugger supports. The visuals of the tutorials were organized using the IDE and hosted in the IDE itself. Following, the attendees were asked to individually as well as collaboratively debug block-based programs while being supervised by the tutors.

8.2.6.1 Preparing the Environment

As we have earlier mentioned, our IDE for visual programming languages is configurable and adaptable through the development of application domain frameworks. Based on this, we have developed an application domain for hosting project items which model educational exercises and examples (see Figure 8.18). For the study's purposes, the chosen model for the exercises and the examples consists of two areas: the question area and the visual code area. In particular, the question area allows loading an image which presents the questions in the context of a lesson. Alternatively, they are able to type the question through an embedded text editor. The visual code area is handed by a Blockly workspace through which the teachers and/or the students are able to program-solve the assignments.

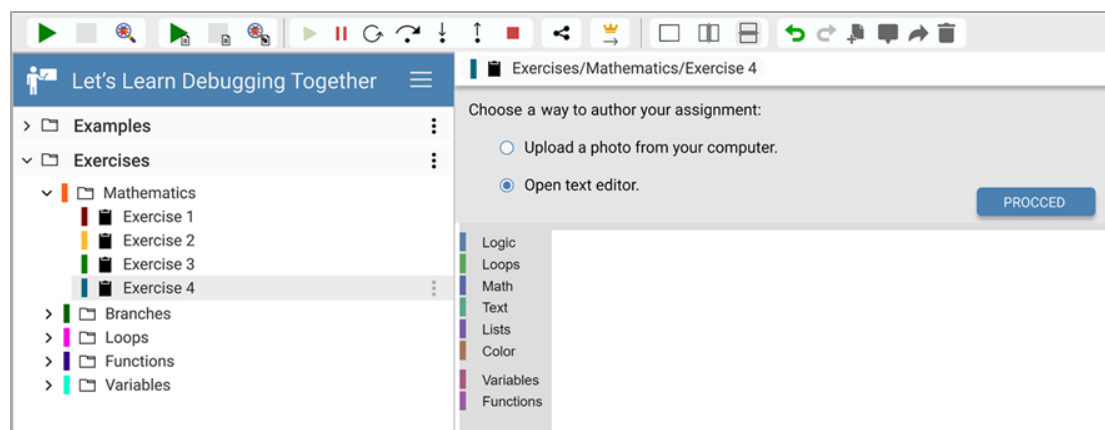


Figure 8.18. Teaching application domain for the collaborative debugging environment.

8.2.6.2 Participants

We undertook the role of the tutors in the study having prior teaching experience and the necessary knowledge of the IDE and its debugging features. However, our study focuses on the students and the efficiency of learning through the collaborative

debugging environment. In this context, 22 first year university students (M=18 F=4, aged from 17 to 19) in the first semester of computer science, taking the introductory course to programming were asked to participate in our study.

In the beginning, the participants were asked to fill in a form with information about their programming experience (i.e. block-based and text-based programming), experience with visual debuggers and their familiarity with collaborative programming. In particular, 12 of them had previous experience with visual programming languages (i.e. *Scratch*, *App Inventor* and *Tynker*) while 7 of them had tried text-based programming previously and all of them had previous experience with pseudocode via high school lessons. Furthermore, none of them had previous experience with debuggers. Moreover, 2 of them had tried to collaborate in programming through sharing screens.

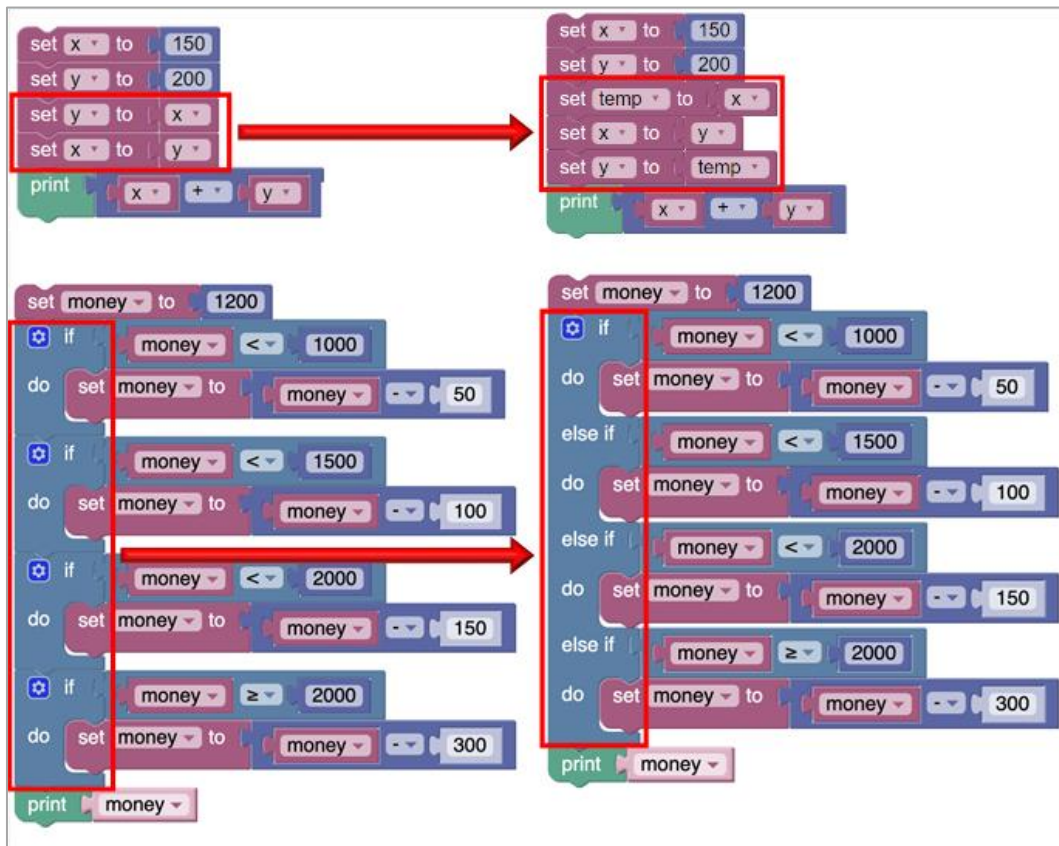


Figure 8.19. Exercises asked to debug individually under supervision. (top) Program swaps x and y and adds them. Find the bug.; (bottom) Program calculates the amount of money for wages(w): $w < 1000 = 50$, $1000 \leq w < 1500 = 100$, $1500 \leq w < 2000 = 150$ and $w \geq 2000 = 300$.

8.2.6.3 Procedure

We prepared 4 exercises which include basic programming parts (i.e. variables, mathematics, branches, loops etc.) for our study. We solved the exercises by using Blockly's workspace and then, we modified them in order to add errors in the visual code. Starting the study process, we created a new collaborative debugging session and invited the students by sending them URL via chat.

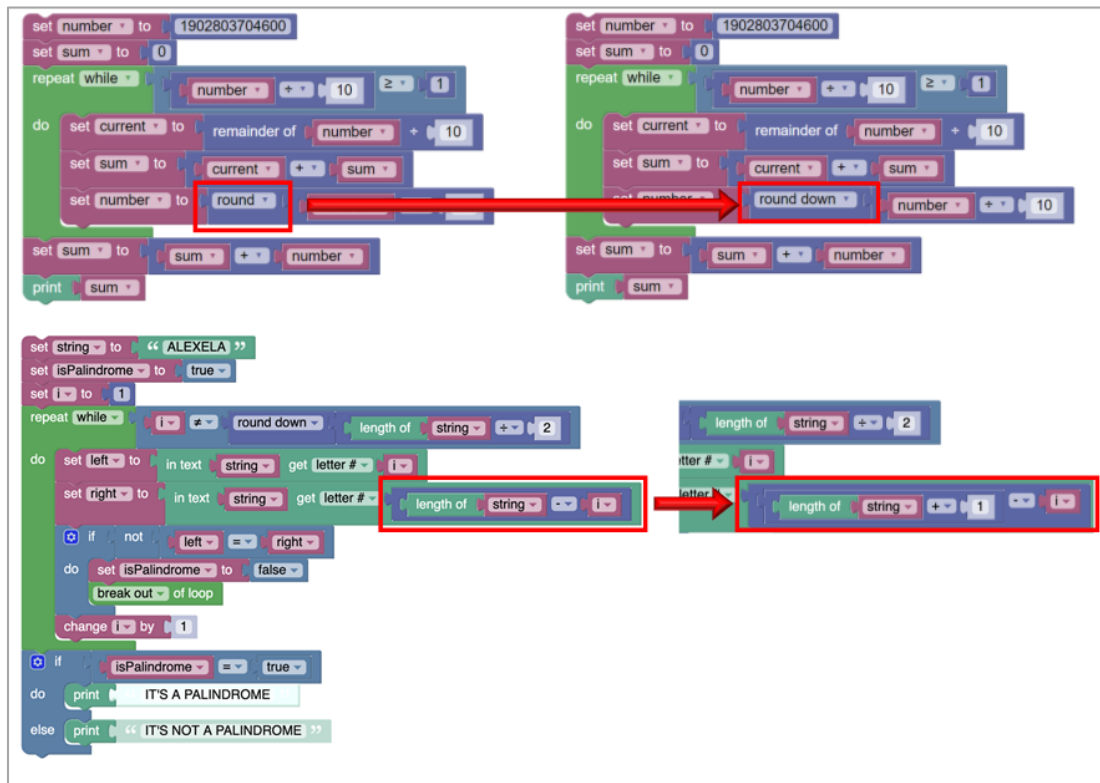


Figure 8.20. Exercises asked to debug in groups under supervision. (top) Program attempts to output the sum of the input number's digits; (bottom) Program attempts to recognize palindrome.

The study procedure was separated into four parts. The first part was a 20-minute block-based programming tutorial in Blockly's workspace. The second part was a 40-minute tutorial in which, we presented the collaborative debugging environment, then, the visual debugger features and we analyzed and debugged two of the aforementioned exercises in real-time. Afterwards, we created one debugging room for each one of the students and one tutor was added as a member to each room. We asked students to join their corresponding debugging rooms and individually find the bugs of the first two exercises (see Figure 8.19) by using the visual debugger for up to 10 minutes. Additionally, the students were asked to think aloud during the debugging process. Each tutor was responsible for monitoring the debugging process of their

corresponding student and tracking their actions for the purposes of the evaluation. Meanwhile, the tutors supervised the students in the debugging process, providing hints if it was required. During the process, tutors kept notes with the total time spent to solve the bug, the number of breakpoints that were added, the number of step-ins and step-overs used, the number of variables watched, the number of the debug sessions initiated and if hints were needed or not.

For the last part of the study procedure, we asked the students to collaborate in groups of two in order to find the bugs of the last two exercises (see Figure 8.20) by using the visual debugger for up to 10 minutes. We created new debugging rooms for this part of the study and invited the students to join them. Furthermore, the tutors followed a similar approach for monitoring and tracking the debugging process. The tutors additionally kept notes for the number of times the students exchange the master privileges. Finally, the students were asked to fill-in online a SUS questionnaire and a second questionnaire with questions that were more specific to our collaborative visual debugging environment.

8.2.6.4 Results

We summarized and further analyzed all the answers given from our participants in two questionnaires. Our primary goal was to evaluate the collaborative debugging environment of our IDE as a teaching tool. As presented in Table 4, the students improve their knowledge for block-based programming through the debugging process. Additionally, the students learnt the basic concepts of a visual debugger and feel confident using it in the future.

Table 4. Questions focusing on learning programming and debugging.

	1	2	3	4	5
Q1. My understanding of block-based programming improved using the debugger.	0	0	2	9	11
Q2. I understand the basic actions (breakpoint, step in, step out, step over, continue) of the debugger.	0	0	0	3	19
Q3. I feel confident that I can use the debugger to inspect values of variables or expressions (watches).	0	1	1	4	16
Q4. I don't think that collaborative debugging can be used as a teaching tool.	14	6	0	2	0

An important part of our evaluation was to test the understandability and the usability of the introduced collaborative debugging features. To the best of our knowledge, the

concept of correction suggestions we presented is new and novices could not have relevant experiences. However, as presented in Table 5, the students felt confident with this feature while the results are relatively similar with the known concept of rooms from communication tools (e.g., Discord, Microsoft Teams, etc.). Moreover, it is important for collaboration tools to offer fluent and natural interactions between users without them feeling restricted by the tool. Based on the first two questions of Table 5, the students are satisfied with the user-to-user interaction via our collaborative debugging environment. In order to thoroughly evaluate our system, we asked the students to fill-in the standard version of the SUS questionnaire (see Moreover, during the evaluation, we perceived two issues of our user-interface in the collaborative debugging environment. The first issue was about the action of creating a correction suggestion. The concept of the correction suggestions is closely related to the concept of project items; thus, the students were expecting to find the option to create a new correction suggestion by right clicking a project item in the project manager. After this observation, we added this option to the right click menu of the project items. The second issue was that the users expected to find the choice of promoting a member to master in the right click menu of the debugging room's members and had some difficulty on spotting the dedicated button next to the debug control. After observing this, we added this option to the right click of each member.

Table 6), resulting in an encouraging outcome.

Based on the data collected by the tutors, we noticed that some of the students which avoided using the visual debugger when debugging individually, started to become more familiar and use it more when asked to debug collaboratively. Furthermore, in general when students debugged collaboratively, they had more targeted and thought out actions. In this context, they achieved approximately the same times to find the bugs, even though, the last two exercises were more complex to solve. However, this may have happened from the experience gained from the previous tasks using the tool and/or from the process.

Table 5. Questions focusing on the collaborative debugging environment.

	1	2	3	4	5
Q1. Collaborative debugging interactions with other students or the teacher felt fluent.	0	1	3	10	8

Q2. Using the debugger (breakpoint, step in, step out, step over, continue) collaboratively with others felt natural.	0	0	2	16	4
Q3. The correction suggestion mechanism seems complicated.	15	5	1	1	0
Q4. The concept of debug rooms is easy to understand.	0	0	0	6	16
Q5. Getting help from a friend for debugging seems easy using this tool.	0	1	2	7	12
Q6. In a future collaborative project with friends, I would use collaborative debugging.	0	1	2	8	11

Moreover, during the evaluation, we perceived two issues of our user-interface in the collaborative debugging environment. The first issue was about the action of creating a correction suggestion. The concept of the correction suggestions is closely related to the concept of project items; thus, the students were expecting to find the option to create a new correction suggestion by right clicking a project item in the project manager. After this observation, we added this option to the right click menu of the project items. The second issue was that the users expected to find the choice of promoting a member to master in the right click menu of the debugging room's members and had some difficulty on spotting the dedicated button next to the debug control. After observing this, we added this option to the right click of each member.

Table 6. Standard SUS Questionnaire.

	1	2	3	4	5
Q1. I think that I would like to use this system frequently.	0	0	6	9	7
Q2. I found the system unnecessarily complex.	16	5	1	0	0
Q3. I thought the system was easy to use.	0	1	1	2	18
Q4. I think that I would need the support of a technical person to be able to use this system.	15	6	1	0	0
Q5. I found the various functions in this system were well integrated.	0	0	0	9	13
Q6. I thought there was too much inconsistency in this system.	18	3	1	0	0
Q7. I would imagine that most people would learn to use this system very quickly.	0	3	0	6	13
Q8. I found the system very cumbersome to use.	15	7	0	0	0
Q9. I felt very confident using the system.	0	2	0	7	13
Q10. I needed to learn a lot of things before I could get going with this system.	12	10	0	0	0

Chapter 9

IoT Automations

“If you think that the internet has changed your life, think again. The Internet of Things is about to change it all over again!”

-Brendan O’Brien

An application domain for visual programming which increasingly gets attention is the smart automations in the Internet of Things. Thanks to IoT era, personal smart devices and services are available in the environment and potentially everybody would like to create micro applications for their daily activities. The vehicle for this goal is the visual programming workspace environment (see Figure 9.1). However, there are several challenges that have to be addressed to achieve this goal. In this context, and in order to better represent our approach, we developed an application

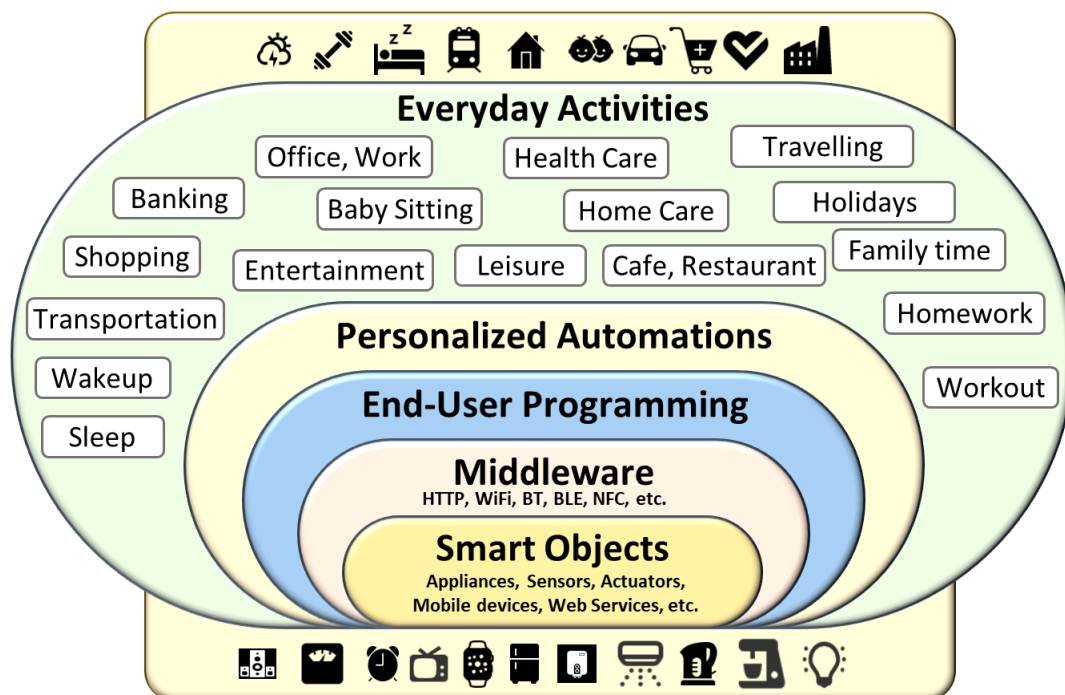


Figure 9.1. The notion of personalized custom automations in the Internet of Things through an End-User Programming framework.

domain framework for smart automations in the IoT by using the infrastructure of the *Blockly Studio IDE*. In this chapter, we analyze each of the challenges of a visual programming workspace environment for IoT automations and discuss how they have been addressed in the application domain framework we built. This work constitutes the main case study of our approach for a domain extendable visual programming IDE.

9.1 Visual Programming Editor for Smart Objects

Connected smart devices and services (also known as smart objects) constitute the first-class element of visual programming in the context of smart automations in the IoT. In particular, the development of applications for smart automations includes actions to communicate with the smart objects, isolate and organize which of them will be involved, filtering their properties and actions, group them, etc. In this context, we developed a specific-domain visual programming editor for the smart objects. We analyze the facilities that have been developed for the editor in the following subsections.

9.1.1 Communicating with Smart Objects

The main functionality of the visual programming editor for smart objects is to make communication possible with the smart objects. IoT middleware is software that serves as an interface between components of the IoT. There are several approaches of IoT Middleware in the bibliography [146]. For our work we use the *IoTivity* framework [147]. It is an open source software framework, reference implementation of the Open Connectivity Foundation (OCF) standards for the IoT. In addition, it provides resource simulation tool that enables us to fully simulate resources (e.g. devices, sensors and services etc.) for testing our platform. Furthermore, the *iotivity-node* [148] provides a JavaScript API for OCF and it is implemented as a native addon using *IoTivity* as its backend. The visual programming editor for smart objects uses the *iotivity-node* to communicate with the smart objects and carries out all the required functionality.

9.1.2 Managing Smart Objects Through Domain Visual Programming Language Elements

In the context of end-user development for smart automations in the IoT, individuals should be able to interact with smart objects, potentially managing, parameterizing and even programming applications involving them. In this context, the visual programming editor for smart objects provides appropriate functionality by introducing and handling domain visual programming language elements. In the next subsections, we describe the functionality and each of the domain visual programming language elements that are identified.

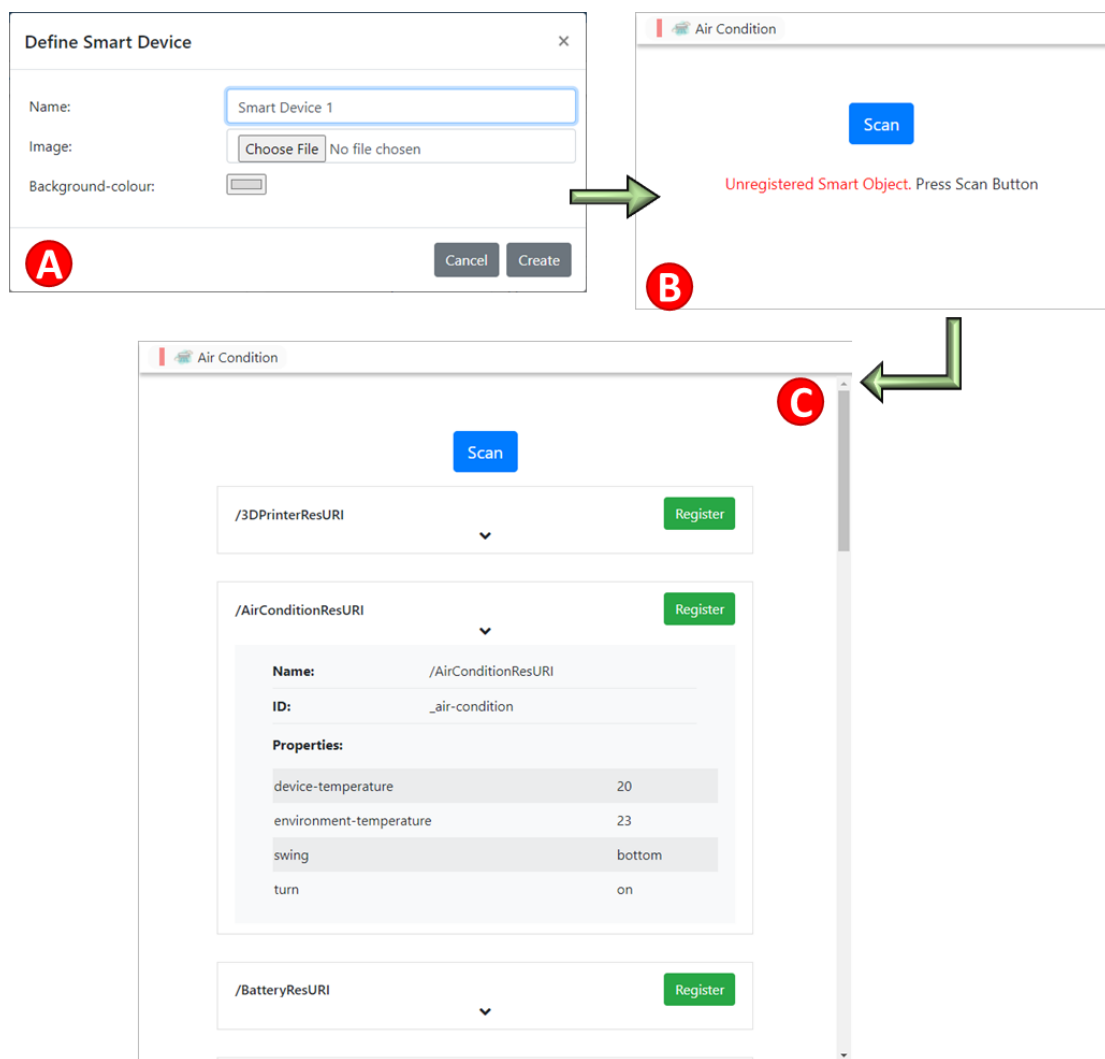


Figure 9.2. Importing Smart Device

9.1.2.1 Smart Devices

In the context of IoT automations, the main elements have to be handled are the smart objects. Hence, the main domain visual programming element that is introduced by the smart object editor is the *'Smart Device'*. In this direction, the first steps of the end-user development process are to scan and register the smart objects that will be involved in a smart automation. In case there are already registered devices from previous created projects, the users are enabled to choose already defined smart devices.

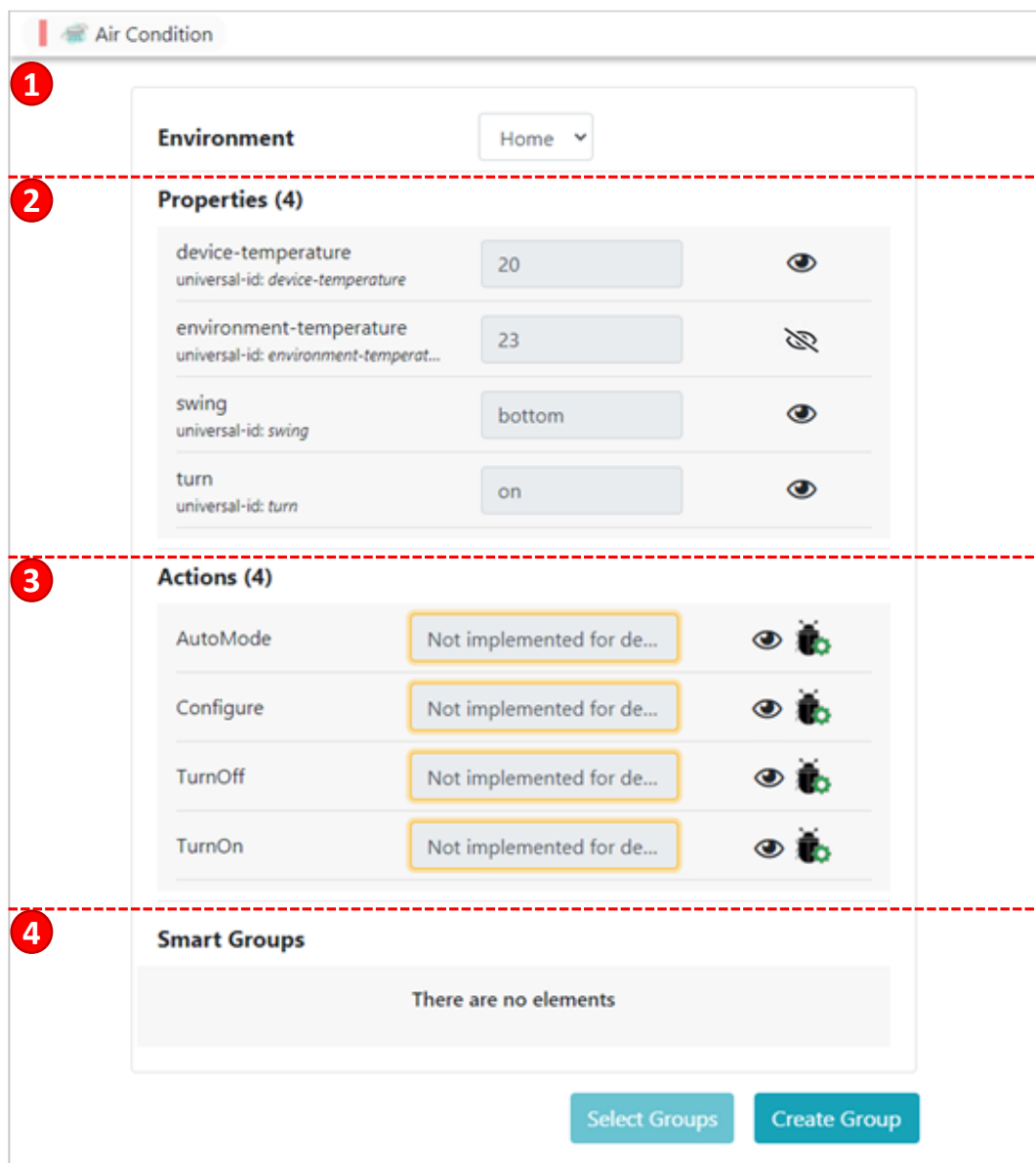


Figure 9.3. The view parts of a registered air-conditioning device.

Defining a new '*Smart Device*' for the IoT automation (see tag A of Figure 9.2), the element is in initial state without selected a connected smart device (see tag B of Figure 9.2). Then, the user chooses to scan for available smart objects and the list of them is shown (see tag C of Figure 9.2). Afterwards, the user select which element will register by clicking the '*Register*' button.

When registration is completed successfully, the '*Smart Device*' visual programming language element instance is created. The visual programming editor for smart objects propagates a signal for the creation of a '*Smart Device*' instance which includes its instance data. This signal will be used by the *Domain Manager* system (see section 4.2.1) in order to define automatically the appropriate *Blockly* blocks that are defined in order to program the automations (see section 9.2).

Afterwards, the '*Smart Device*' instance is established as registered and the smart object editor presents its parts (see Figure 9.3). In particular, there are four different parts. In the top of the smart object view, the user chooses which is the smart environment which is defined (see next section). In the second part, the users can view the properties that are exported by the smart object. In this context, the users are able to handle if one property will be enabled for the development (create or not *Blockly* blocks with the Domain manager system) by clicking the '*eye*' toggle button on the right of each property. Moreover, below of each property name, there is an alias which is used for the smart object groups that we analyze them in the next section. The third part of the smart object view relates with the exported actions of the smart object which includes the '*eye*' toggle button that has similar functionality with the aforementioned properties. Additionally, the end-user developers are able to program the behavior of each action that will be used during the debugging process (see section 9.7). The fourth part of the smart object view shows the smart object groups and it handling which are discussed in section 9.1.2.3.

The user interface for the visualization of the smart objects differs per smart object based on the exported functionality (i.e., properties and actions). In this context, the user-interface is generated automatically based on the *JSON* data response from scan's request to the *IoTivity*.

9.1.2.2 Smart Device Environments

A smart automation in the IoT could include numerous smart objects. In order to better organize them for the end-user development process, the visual programming editor for smart objects introduces an extra domain visual programming element named as *'Smart Environment'* which helps the organization of the smart objects. In particular, the end-user developer is able to define smart environments which include either lists of smart objects or other inner smart environments.

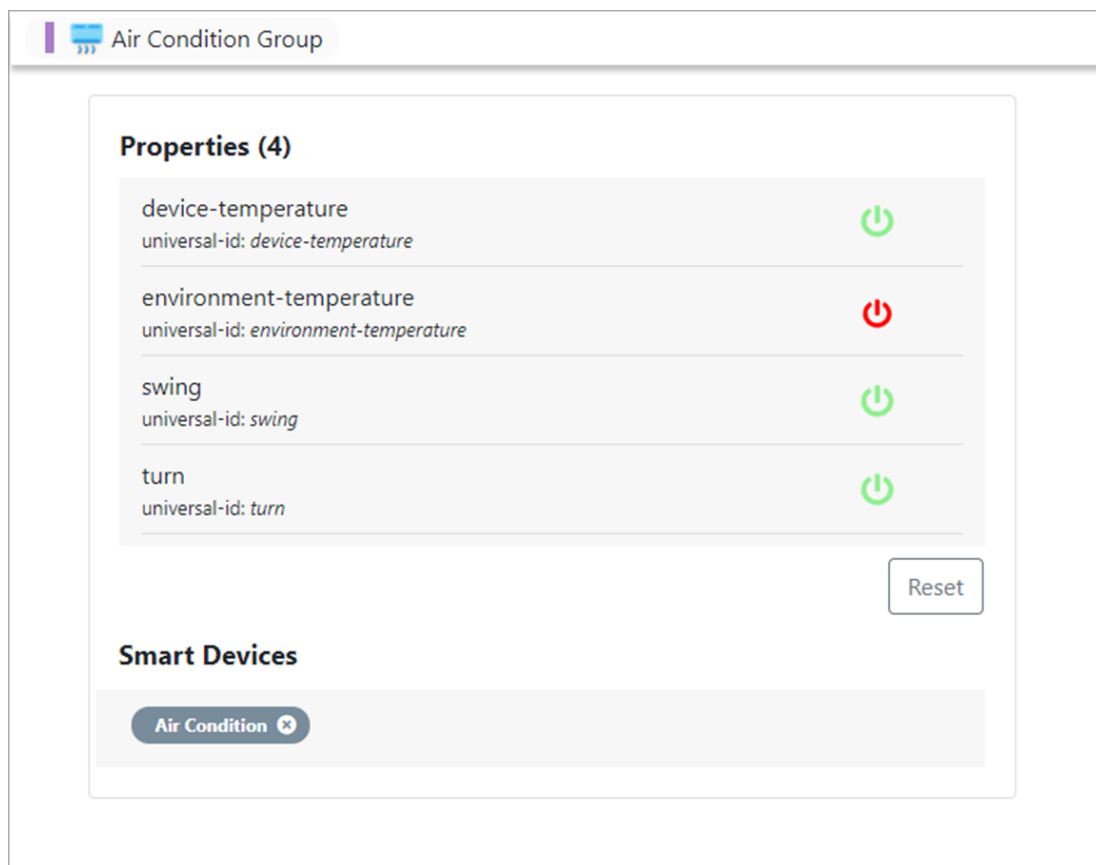


Figure 9.4. Smart device group for air-conditioning.

This feature enables the end-user developers to organize by defining groups (folders) of smart objects. The end-users are able to choose which smart environment will be included each smart object either by creating the smart object instance in the smart environment or by choosing which is the smart environment that will be included through the enable user-interface selection as displayed in the 1st tag of the Figure 9.3. Furthermore, creating smart environments is optional and the end-user developers have the choice of not using smart environments, which is useful for smart automations which include a little number of smart objects.

9.1.2.3 Smart Device Groups

Additionally, several smart objects with common functionality could be involved in IoT automations (e.g. the smart light bulbs of a home). In this case, the end-user developer may like to handle them as a group (e.g. switch on/off the smart light bulbs). In this context, the smart object editor introduces the domain visual programming language element named as *'Smart Device Groups'*.

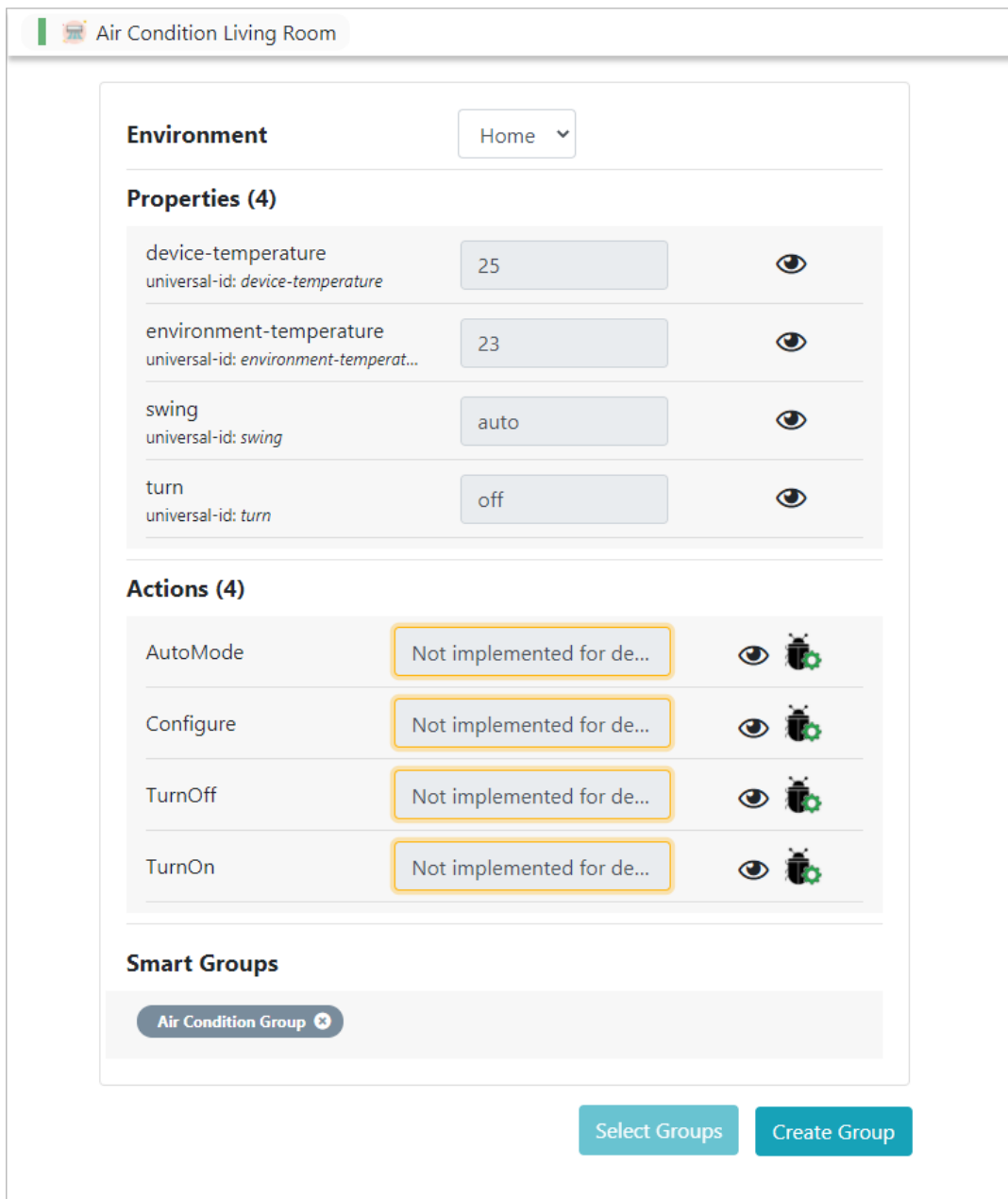


Figure 9.5. The view of air-condition living room.

In particular, the smart object editor attempts to identify which of the registered smart objects of the smart automation have common functionality and organize them in groups. These groups give the ability to develop-handle the smart objects in groups instead of requiring to handle each one of the common smart devices.

In this context, the end-users are able to create new groups with common functionality via the smart objects by exporting the smart object properties (i.e., click the “*Create Group*” button presented in Figure 9.2). The created groups of common functionalities include information of the name, the image and the color of the group. In addition, they include the list of common functionalities and the list of the smart objects that are included (see Figure 9.4). The end-user developers are able to remove a smart object from the list in case they would like to handle it separately.

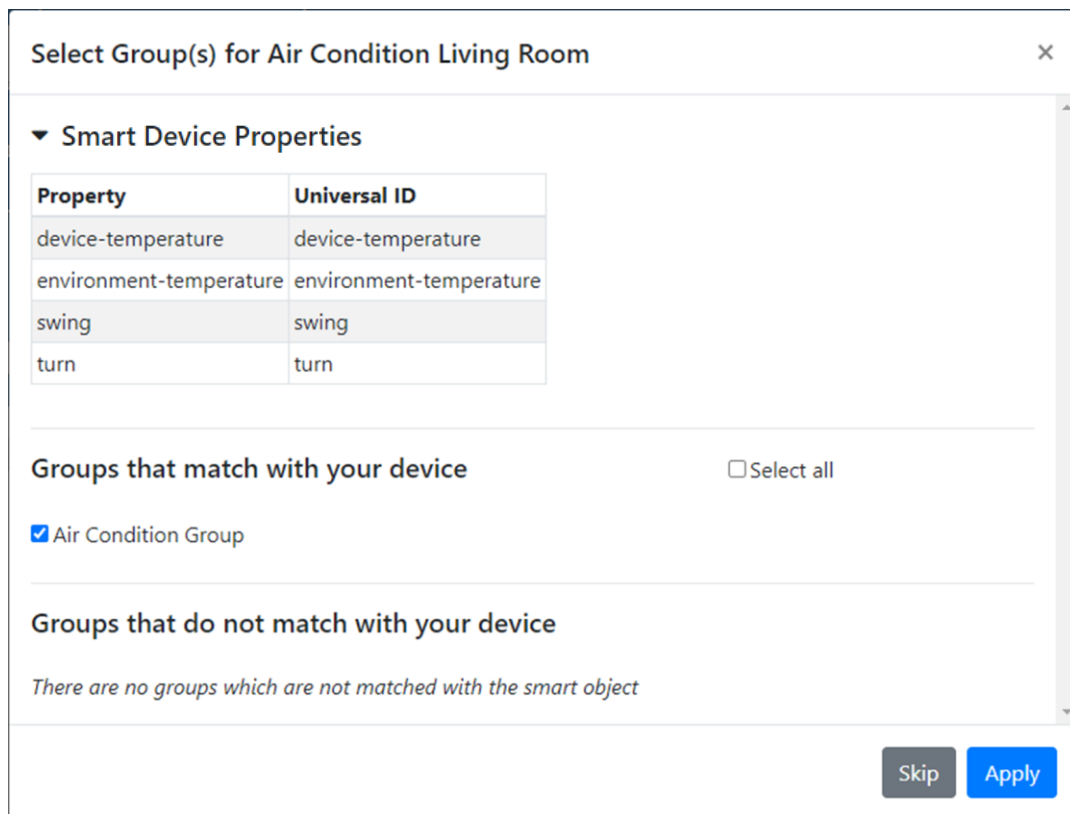


Figure 9.6. The view of air-condition living room.

Furthermore, several smart objects which could be included in a group have a different exported API. The differences could be either some of the functionalities that are not provided by one device or they provide all the functionality but it is exported with different naming(s). In this context, the end-user developers are able to edit the common functionality list (i.e., activate/deactivate items). This is useful in

case they wouldn't like to include a specific common functionality in the group and this functionality is not supported by one smart object that they would like to be included in the group. Using the toggle turn on/off button which is presented on the right of each property, the users can add or remove the functionality. In this case, the smart object editor sends signal for editing with the respective data of the instance. In this context, the generated *Blockly* blocks for the smart object group instance are handled by the Domain Manager as happens in case of smart objects which earlier discussed.

Moreover, in the process of the matching common functionality of smart objects, the end-user developer is able to give for each one of the properties an alias. This is useful in the case that smart objects 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 alias. In this context, a dialogue opens when a smart object is created in order to view the smart object information and manage the smart object groups related with this smart object.

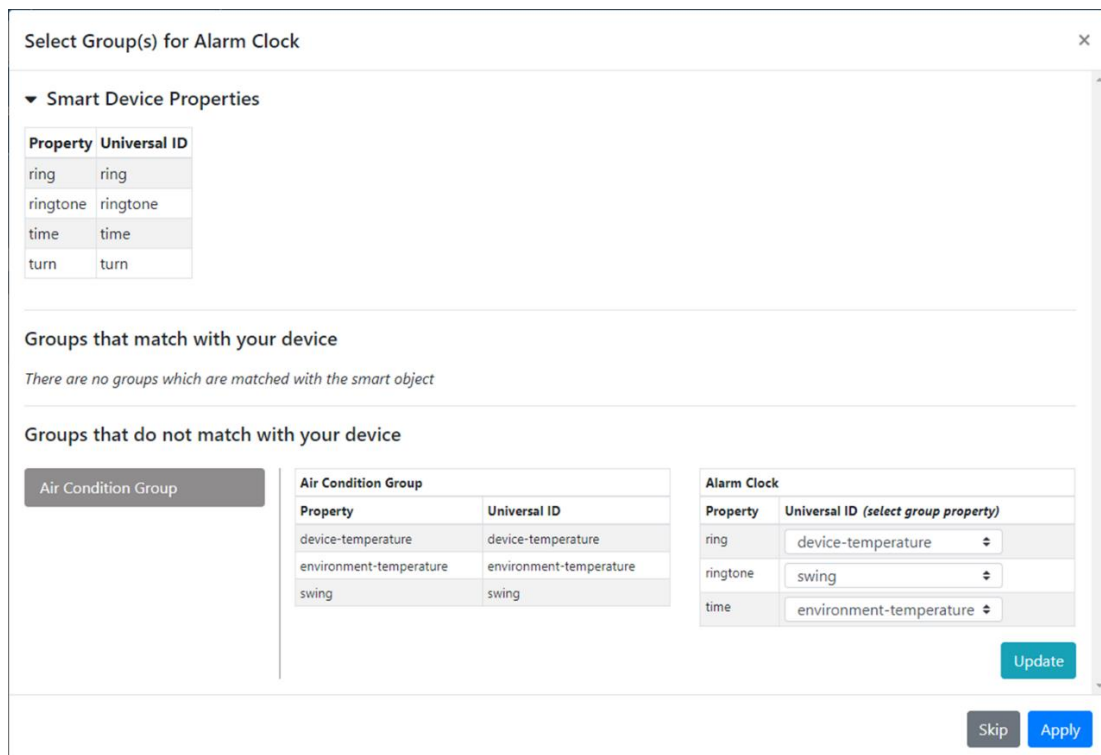


Figure 9. 7. Handling smart object groups for the alarm clock.

For example, considering that we have define the smart object 'Air-Condition' with the exported functionality as it is depicted in Figure 9.2. Then, creating another smart

object *'Air-Condition Living Room'* which includes similar functionality (see Figure 9.5). In this case, the smart object editor suggests to the user to add it in the group when the smart object is created by opening a pop-up dialogue that enables to choose or not include the *'Air Condition Living Room'* in the *'Air Condition Group'*. Moreover, creating an *'Alarm Clock'* as smart object the number of provided functionalities matches (i.e., both *'Air Condition Group'* and *'Alarm Clock'* have 4 properties), however, the API differs. In this context, the dialogue which handles the smart object groups opens and enables the end-user to give appropriate aliases in order to match the APIs (see Figure 9.6) and enable the choice to include it in the group.

9.1.3 Loading Shared Automations

As earlier discussed, *Blockly Studio IDE* enables sharing of the end-user developed applications. The end-users are able to search, download and use the shared applications. In the case of the smart automations in the IoT, there are issues that have to be addressed on loading shared applications of others.

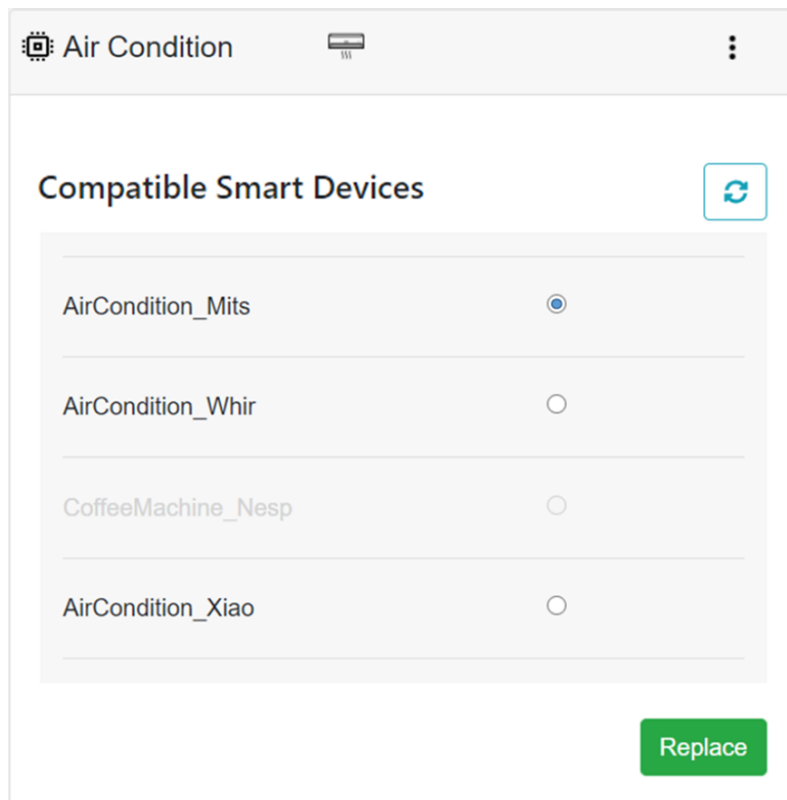


Figure 9.8. Replacing the *'Air Condition'* smart device of the shared application with a compatible smart device.

In particular, before using a shared application, a replica is created in the end-user's environment. However, in the smart automations case, the first development step is to define which of the registered smart objects will be involved. As a result, the first (i.e. extra) development step for shared applications is the replacement of the smart objects which participate. In particular, when loading the shared application for the first time, the visual programming editor for smart objects marks all registered smart objects as invalid and warns that the smart objects have to be replaced. Also, the end-user developers are able to define extra smart objects for these smart automations in order to modify and/or extend them. Furthermore, this process could be repeated during the end-user development process.

The visual programming editor for smart objects firstly identifies the smart objects which provide compatible functionality and are unique in the end-user's environment. In case these smart objects are not unique (e.g. two air-conditioning systems one at home and another one at the office), the smart object visual programming editor asks the end-users to select which of the smart objects will be used as depicted in Figure 9.8. Then, the smart objects which are not compatible with any of the end-user's smart objects have to be handled.

In the case there are smart objects which can replace these smart objects but export different API, the end-users are able to use the alias and this will help in matching the smart objects. Also, the end-users are able to remove the smart objects; whether their functionality isn't useful to them or they don't have them available in their arsenal. Removing defined smart objects prerequires that the author of the shared application hasn't defined these smart objects as mandatory. Furthermore, removing one or more smart objects from a shared application could decrease functionality or even make the application useless. The latter is the responsibility of the end-user.

9.2 Visual Programming Blocks for the Behavior of Smart Objects

As mentioned in the previous section, there are three domain visual programming language elements that are handled by the smart object visual programming editor: the smart objects, the smart object environments and the smart object groups. In this section, we discuss the behavior of the visual programming language elements that

were introduced for each one of the domain elements in order to handle their functionality in smart automations.

As earlier mentioned, the application domain author has to develop constructors that get, as input, the data which are exported by the specific domain visual programming editors during the creation of visual domain elements. These constructors are used by the mechanism (see 4.2.1) and automatically handle the management of new blocks. The first categories of blocks that we will discuss in the following paragraphs are the handling set of blocks for smart objects.

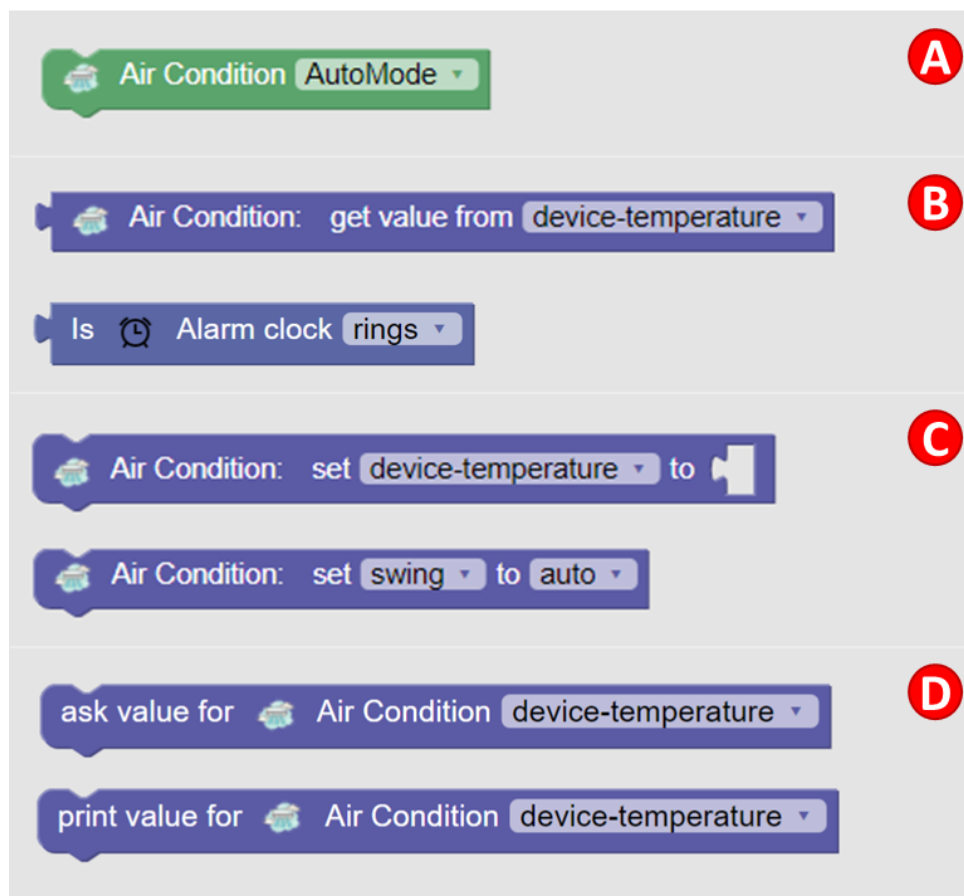


Figure 9.9. Basic Blockly Blocks for Smart Objects; actions for smart objects (tag A), setters, getters (tag B, C) and input, output for smart object properties in the I/O Console.

The behavior handling set of Blockly blocks for smart objects consists of four categories. The first category of blocks is the basic handling of request actions (see tag A of Figure 9.9). Using this block during the development, the end-user developer is able to choose the action name that they would like to use. Based on the signature of the function (i.e., no arguments as input, one or more arguments, type of the

arguments, etc.) the block changes automatically as shown in Figure 9.10. The second category of blocks is the basic handling of getting, setting property values of smart objects (see tag B and C of Figure 9.9). Using these blocks, the end-users are able to request the functionality that is provided by the smart objects. Additionally, the end-user developers are able to print the values of smart object properties in the Input Output Console or ask from the application users to set values in smart object properties through the Input Output Console (see tag D of Figure 9.9).

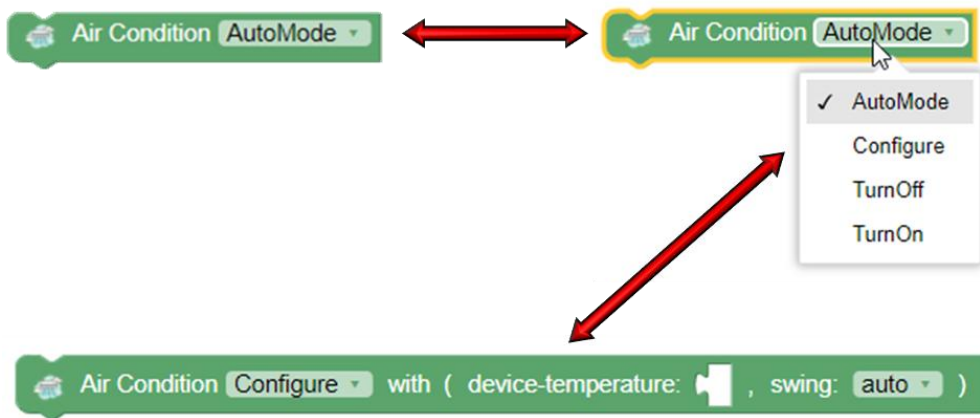


Figure 9.10. Dynamic change of a Blockly block based on the choice during the end-user development.

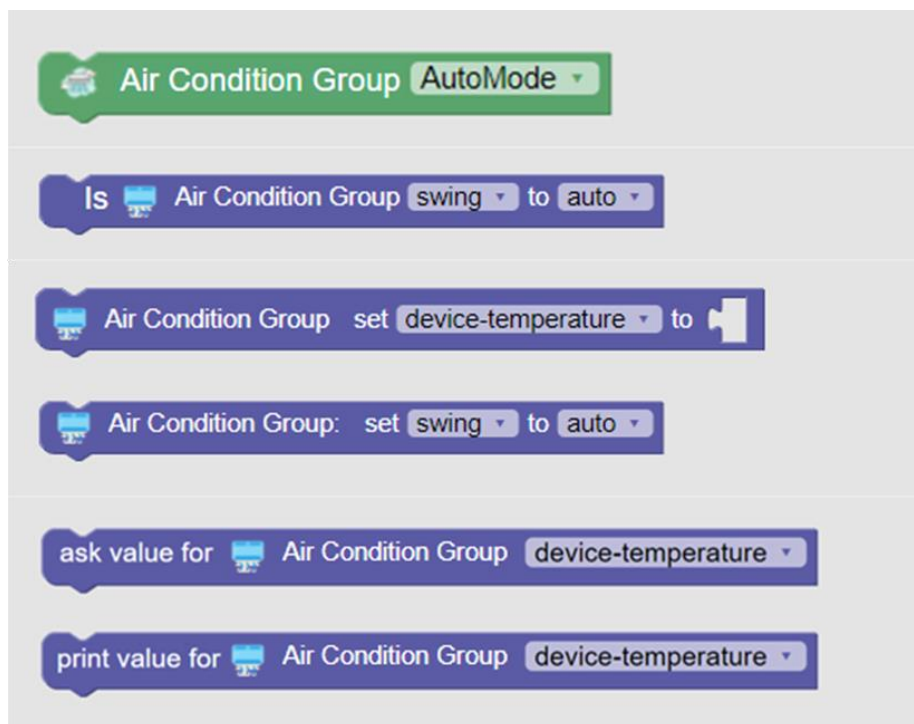


Figure 9.11. Blockly Blocks for Smart Object Groups.

In an analogous way, blocks are defined for the domain visual programming language element for smart groups. All the blocks which are defined for the smart objects are used for the smart object groups except the blocks for getting property values as each of the included devices may have different value (see Figure 9.11). These blocks are handling all the smart objects which are included in the smart object group.

For example, using the block with the action *'AutoMode'* of the *'Air Condition Group'* means that all the air conditions will be set in auto mode. Asking for a value for device-temperature means that all the smart objects will get as device-temperature the input that user will give through the input-output console. The Blockly block *'Is'* checks if the state of a specific property applies for all the smart objects that are included in the group.

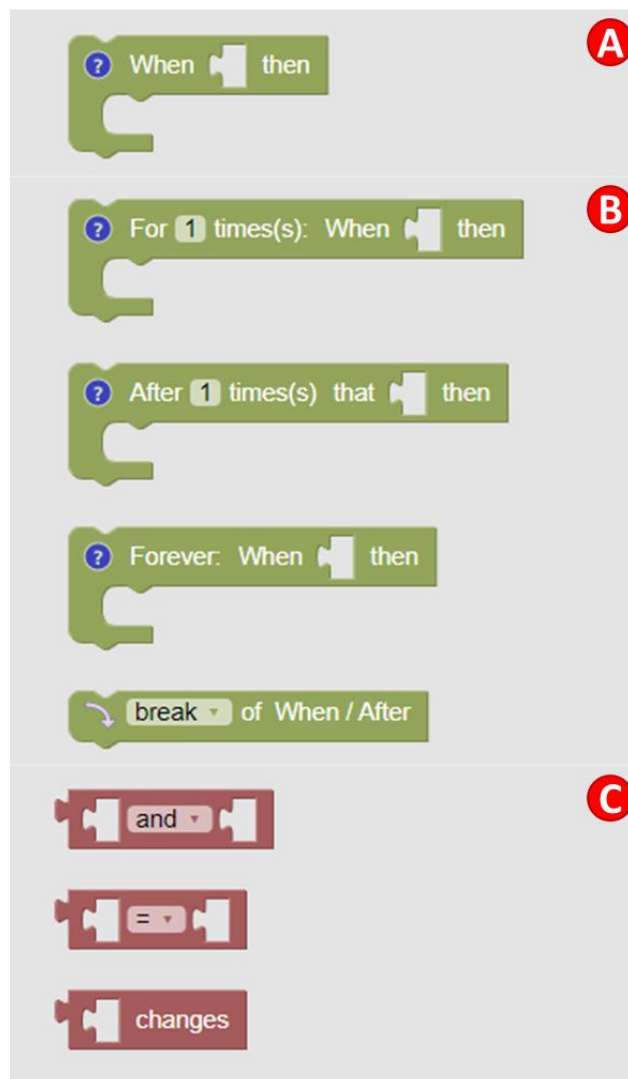


Figure 9.12. Conditional Event Blockly Blocks for Smart Automations.

9.3 Visual Programming Blocks for Conditional Automations

One of the main categories of visual programming language elements is the conditional automations. The conditional automations empower the end-users to define conditions based on the state of smart object properties (see Figure 9.12). When these conditions are evaluated to true, their inner blocks (i.e., children) are executed.

The Blockly blocks for the conditional automations are separated into two categories based on repeatable and not repeatable conditional automations. In particular, the non-repeatable conditional automation is triggered only once, when its defined condition is evaluated as true. This means that the next time that the condition will be evaluated as true nothing will happen. In case the end-user developers would like to develop conditional automations that will happen more than one times, they have to use the conditional Blockly blocks that are displayed on tag B of Figure 9.12. The first and the third block is used to specify how many times conditional automations will be triggered, while the second Blockly block's children will be executed after the condition is evaluated as true for a specific number of times. Finally, the last block can be used as a child of the first or the third block in order to break or continue the execution of its children.

Additionally, the end-users are able to define related descriptions for the functionality of blocks. Based on this, the runtime environment interprets the information respectively (see section 9.6).

Moreover, the end-user is able to use these blocks as either top-level blocks of the conditional automation (i.e., without top-bottom input blocks) or repeatedly as children in order to develop more complicated programming expressions. In this context, the functionality of the inner conditional blocks change. Particularly, for using nested conditional blocks, when the inner block starts, the parent block is deactivated (its condition is not evaluated). When the children conditional blocks are accomplished, the parent condition is activated again (i.e., triggers when its condition is evaluated to true).

Moreover, three blocks contribute to the definition of conditions (see tag C of Figure 9.12). The third block gets, as its input inner block, a getter of a smart object 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 object's value and checks if something changed. The second block is used to compare a value of a smart object property with another value. The first block is used to build more complicated conditions using logic operators *AND*, *OR*. Moreover, when the conditional event is triggered, inner condition blocks are reset in order to be available for a possible next use, in the case this conditional automation will be invoked more than one time (e.g., use it in the body of a loop or function definition).

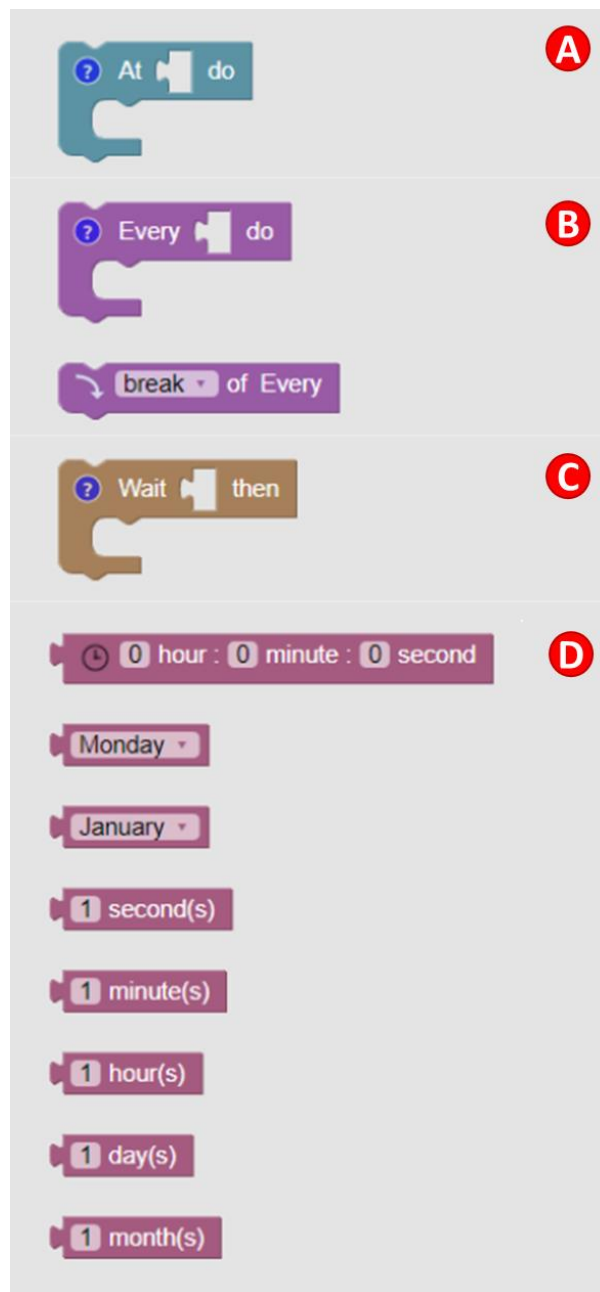


Figure 9.13. Scheduled Event Blockly Blocks for Smart Automations.

9.4 Visual Programming Blocks for Scheduled Automations

Another main category of visual programming language elements is the scheduled automations. The scheduled automations empower the end-users to define schedules based on the state of smart object properties (see Figure 9.13). When the IoT automations project starts, the application calculates the calendar, the time and starts a timer in JavaScript. Based on these values, the scheduled blocks are triggered on the specific date and time which are selected by the end-user developer. When these blocks are triggered, their inner blocks (i.e., children) are executed.

The Blockly blocks for the scheduled automations are separated into three categories based on specific date or time and repeatable or not repeatable automations. In particular, the block *'At'* gets as input the specific date or time that the children blocks will be executed at. This block executes once during the project execution. Another block that is executed once is the *'Wait'* block. This block gets, as input, a specific time period. When this time period will be completed, the children blocks will be executed. The last category of blocks supports repeatable execution based on specific time periods (see tag B of Figure 9.13). Particularly, *'Every'* block gets as input the time period (see tag D of Figure 9.13). Every time the defined time period is completed, the children blocks are executed. Additionally, the second block handles the flow of the repeatable loops by using break in order to stop the loop and continue in order to stop the execution of the below children instructions of the *'Every'* block.

Additionally, the end-users are able to define related description for the functionality of each block. Based on this, the runtime environment interprets the information of these blocks respectively (see section 9.6).

Moreover, the end-user is able to use these blocks as either top-level of the scheduled automation (i.e., without top-bottom input blocks) or repeatedly as children in order to develop more complicated programming expressions. However, in the case of the *'Every'* blocks, restrictions have been added. Particularly, we disable the use of *'Every'* blocks as children of *'Every'* blocks. The reason for this is to prevent the conflicts among the parent and the children *'Every'* blocks.

9.5 Authoring Project for IoT Automations

The basic part of authoring an application domain framework is the project application structure including the project options, the project element types, the configurations of the visual programming editors, the project manager functionality, etc. In the following subsections we analyze each one of them.

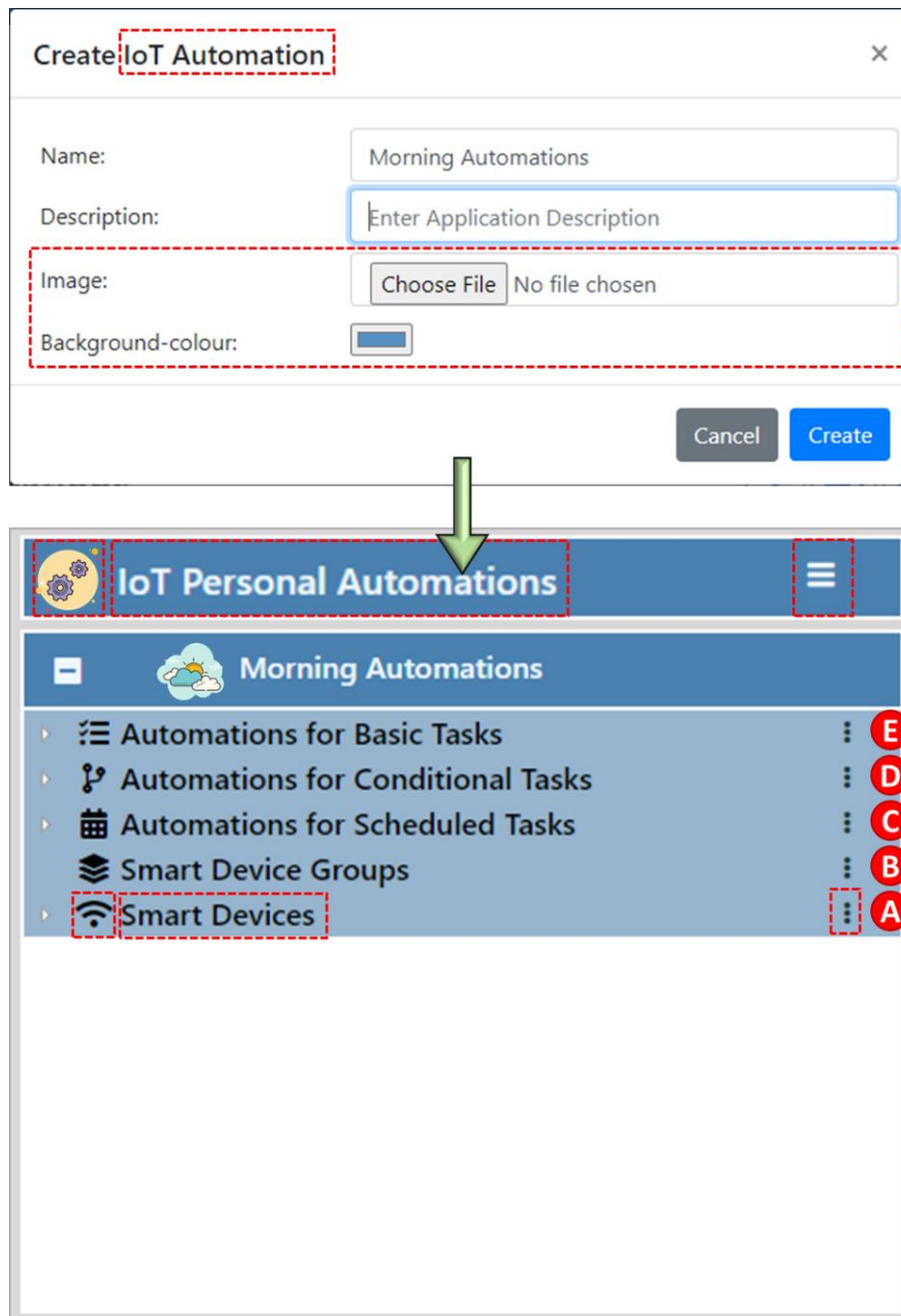


Figure 9.14. Configuring the create application dialogue for IoT Automations and the Project Manager view based on the user's input data.

9.5.1 Creating IoT Automation Project

The first step for the end-user development of an application is to enable the user to fill-in the information of the project. The *Blockly Studio IDE* provides a default dialogue to create a new application. It also provides a view of the projects in *Start Page* component (see section 3.3). We authored two more items in the dialogue when the user creates new IoT automations. The first extra element is an image for the automation and the second is the basic background color of the Project Manager component. Based on the selected color, the categories of project elements and the project elements get lighter color automatically.

When the user selects the ‘*Create*’ button, the visual programming workspace opens by initiating the *Project Manager* component. As earlier mentioned, the *Project Manager* component is fully configurable as to the style, the content and the functionality. In this context, we defined the title and the image that will be rendered in case of the IoT automations (see Figure 9.14).

Additionally, the *Project Manager* component is configured based on the facilities that will be enabled in the context of the application domain. The application domain author is able to filter which of the *Blockly Studio IDE* facilities will be available for the end-user development process based on the requirements of the domains and which of the facilities have been authored. For example, the runtime environment and the debugger are not able to be used if the domain author hasn’t developed the appropriate run and debug scripts which are the entry points of the applications which are executed, the collaborative editing component cannot be used in case the visual programming editors which have been developed for the application domain do not support functionality of syncing, etc.

A filtering of the provided *Blockly Studio IDE* facilities is accomplished by authoring the menu options that are available through the “*burger*” which is provided by the Project Manager component (see on the top of Figure 9.14). In addition, menu options are defined for each of the projects. These project menu options open when users right click the project label or by clicking on the three-dot button which is presented when the user mouse over the label of each project.

9.5.2 Project Elements

Having created the project for IoT smart automations (see at the bottom of Figure 9.14), there are five categories available to the end-user developer for the visual programming process: *'Smart Devices'*, *'Smart Device Groups'*, *'Automations for Basic Tasks'*, *'Automations for Conditional Tasks'* and *'Automations for Scheduled Tasks'* (see tags A-E of the Figure 9.14). The first two categories of project elements targets at the management of smart objects and the last three categories of project elements focus on the end-user development of automations. In the following subsections, we analyze each of these categories.

9.5.2.1 Smart Devices

The main category of project elements is the *'Smart Devices'* (see tag A of Figure 9.14). Using this category, the end-user developers are able to import the smart objects and organize them through the use of smart environments which play the role of subfolders. As earlier mentioned, the smart objects and the smart environment are handled by the Smart Object Editor (see section 9.1). However, in order to render the information of the project element instance, we developed a project element template (see Figure 9.15) in which the visual programming smart object editor instance view is hosted, including information that is related to the project element.



Figure 9.15. Project element template that includes information and hosts one visual programming editor instance.

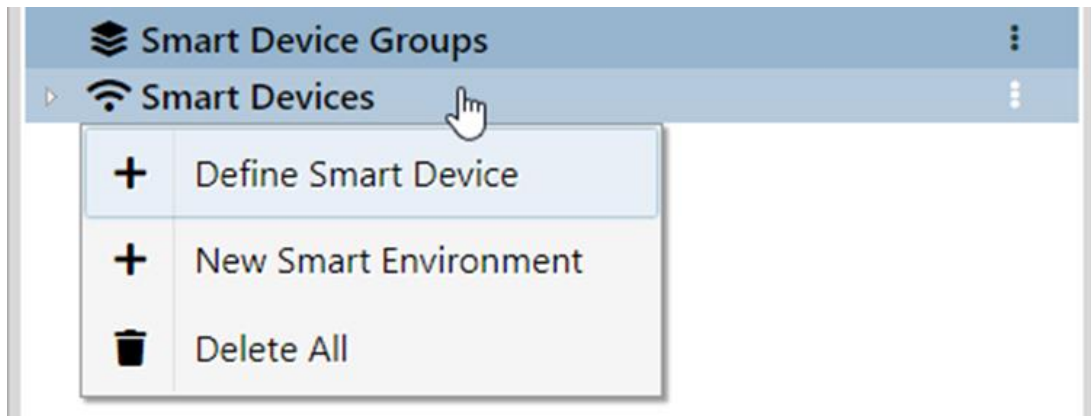


Figure 9.16. Menu options for the Smart Devices Category.

In order to enable the creation of a new *Smart Device* or *Smart Environment*, we authored the menu options of this category. By clicking on the three dots which are positioned on the right of the *Smart Devices* label (see tag A of Figure 9.14) or using right click, the menu options open (see Figure 9.16). In case the user chooses to import smart devices, or to create a new environment, a respective dialogue opens in order to fill-in the smart device information (i.e., name, image and color). Similar options are presented in case of the smart environment menu options in order to give

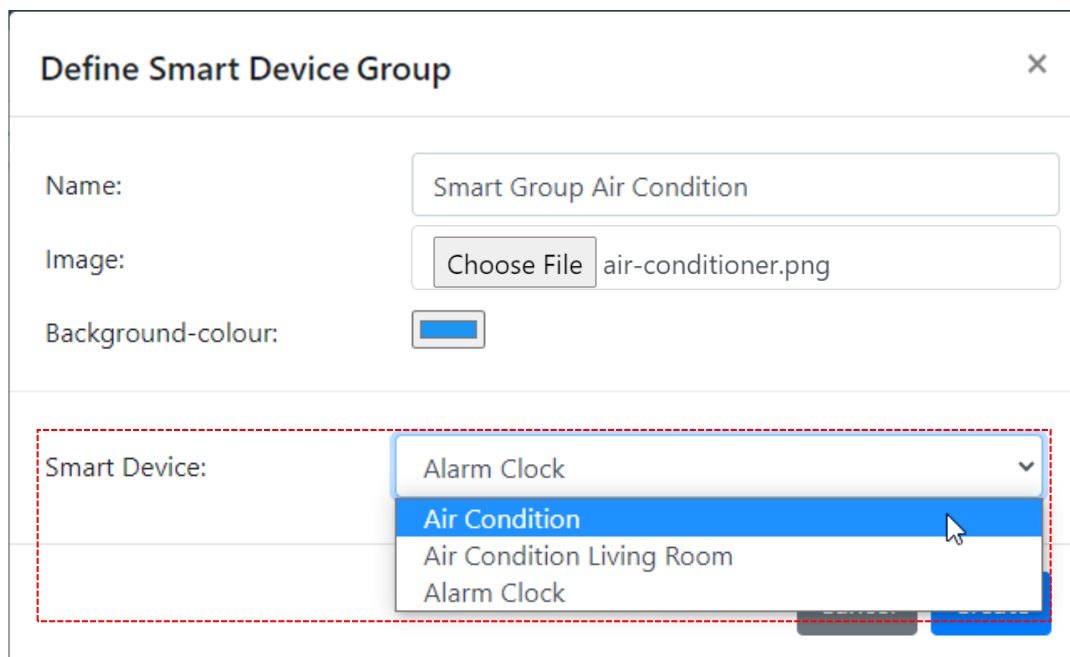


Figure 9.17. Creating new smart group device by choosing smart device that will export its functionality interface.

ability of defining inner smart devices or creating inner smart environments.

9.5.2.2 Smart Device Groups

The second category of project elements is the '*Smart Group Devices*' (see tag B of Figure 9.14). Using this category, the end-user developers are able to group the development of smart devices in case they have common functionality. In this context, we have defined an extra option for the creation of smart groups (see Figure 9.17). Using this option, the end-user will be able to choose the smart device. The functionality of this smart object will be used by the system in order to export it as the smart group interface

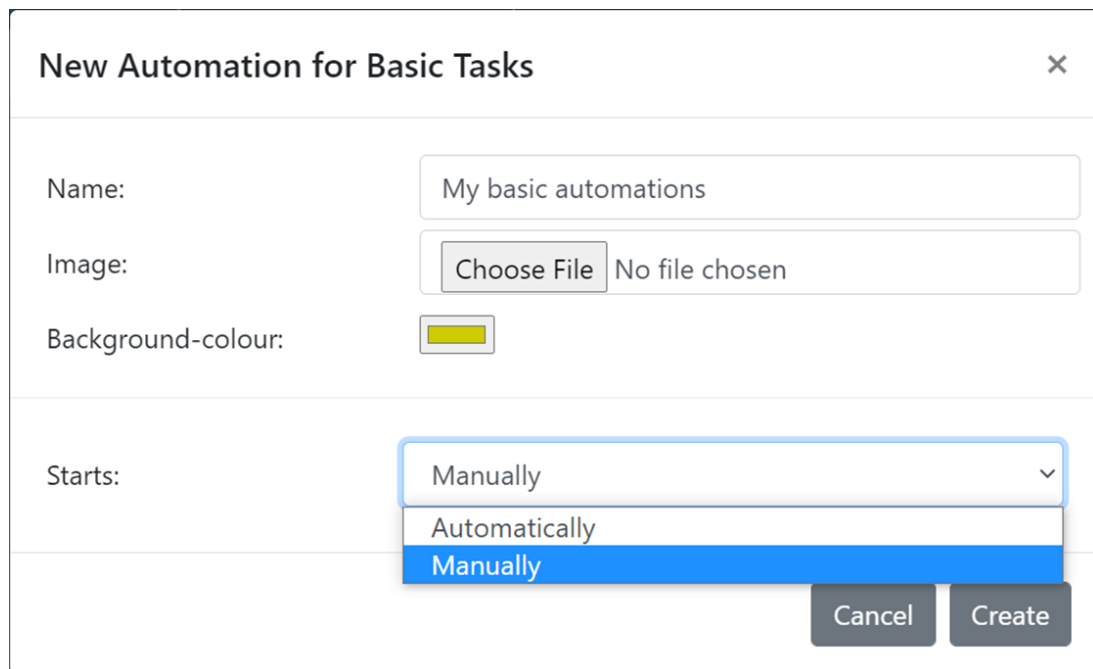
Additionally, we defined this '*Smart Device*' option to not be presented in the list of options after the construction of the smart device group. Following this approach, we prevent dependencies between the smart device and the smart device group. This means that by removing the smart device, the smart device group is not required to be removed. However, the smart device group has to be updated by removing from its list the specific smart device. Moreover, as earlier mentioned, the end-user developers are able to create new smart device groups through the smart device view by pressing the button '*Create Group*' which is available at the bottom (see Figure 9.5).

9.5.2.3 Visual Programming Blocks for Project Elements of Automations

The next three categories of project elements are targeting at the end-user development of automations. In order to enable the end-user developers with more visual programming language expressiveness, we authored an extra option for these three project element categories. This option enables the end-user developers to choose if the automation will start automatically when the project execution starts or manually through visual programming instructions during the project execution (see Figure 9.18).

By authoring this option, we are focusing on enriching the end-user development expressiveness. In particular we provide the ability of manual handling for when automations need to start based on specific circumstances during the execution. This happens through the run-script which gets, as input, the project environment data that includes this information and chooses which of the project elements will be executed

on the beginning of the project execution. For example, when the user leaves home (i.e. opens the door of the car), start automations for securing the house and automations to tidy up by using the smart devices.



The image shows a dialog box titled "New Automation for Basic Tasks" with a close button (X) in the top right corner. The dialog is divided into several sections. The first section has a "Name:" label and a text input field containing "My basic automations". The second section has an "Image:" label and a file selection area with a "Choose File" button and the text "No file chosen". The third section has a "Background-colour:" label and a color swatch showing a yellow color. The fourth section has a "Starts:" label and a dropdown menu. The dropdown menu is open, showing three options: "Manually" (which is selected and highlighted in blue), "Automatically", and "Manually". At the bottom right of the dialog, there are two buttons: "Cancel" and "Create".

Figure 9.18. Choosing if automation will start automatically in the beginning of project execution or later with visual programming block element instruction.

In order to start automations manually, appropriate *Blockly* blocks have to be defined and handled during the end-user development process based on the project elements of automations. These blocks handle starting or stopping an automation. As earlier mentioned, the visual programming editors have to export signals when a domain visual programming language element instance is handled (i.e. create, edit and delete). Based on the provided functionality of the *Blockly* editor which posts appropriate signals when the workspace instance changes (i.e., create, edit, delete), we authored the domain visual programming language element with the respective *Blockly* blocks (i.e., start automation, stop automation). In this context, the *Domain Manager* handles the signals and automatically provides the updated *Blockly* blocks to the appropriate *Blockly* editor instance toolboxes based on the defined configuration (see section 4.1.2).

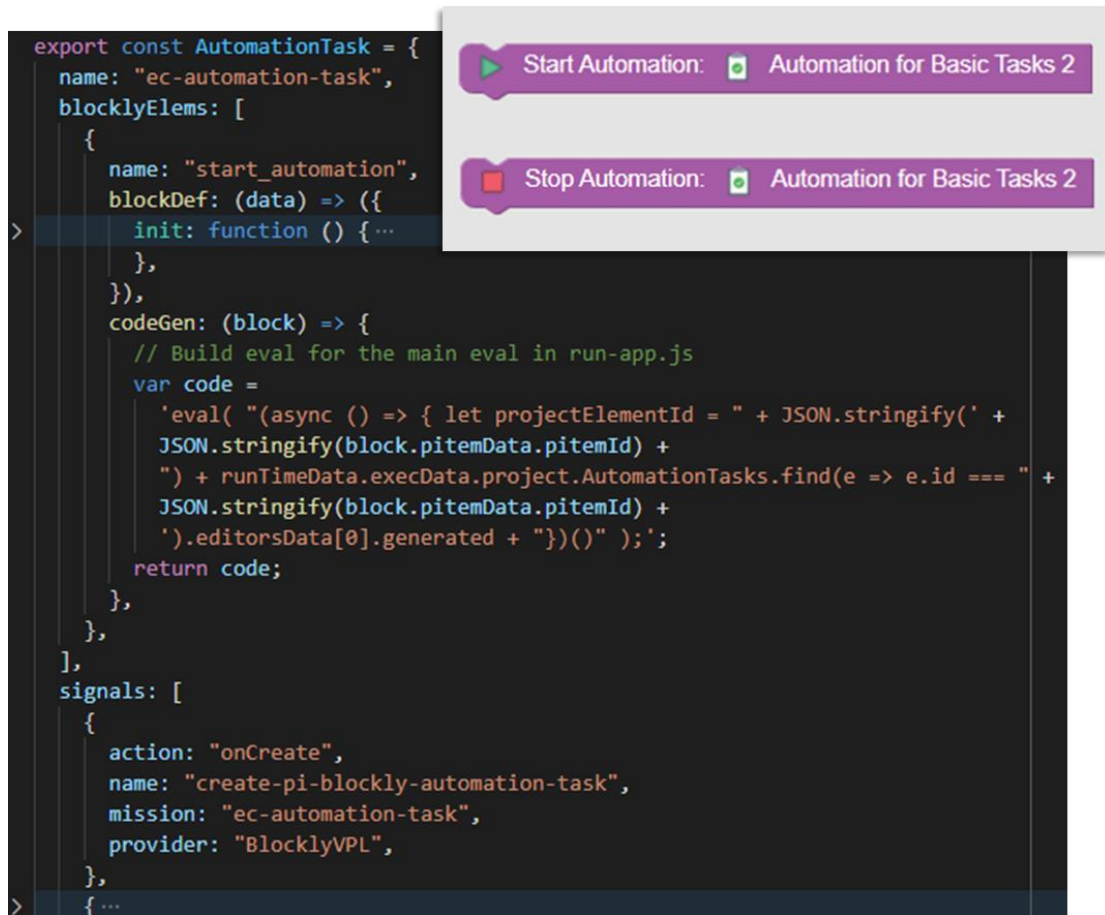


Figure 9.19. Authoring *Blockly* blocks to enable the end-user developers handle manually start and stop of the automations for project elements.

9.5.2.4 Automations for Basic Tasks

Having defined and having organized the smart objects, the end-user developers are able to develop automations. There are three different categories of automations that can be developed in a project for IoT automations. The first category of automations is 'Basic Tasks' (see tag E of Figure 9.14). Creating basic automations, the end-user developer is able to define if the automation will start automatically or manually by using appropriate visual programming block elements.

The project element of 'Automations for Basic Tasks' has developed a pure template (see Figure 9.15). In addition, the *Blockly* editor instances configuration for the basic tasks includes the general-purpose predefined blocks and the dynamic blocks which are generated based on the smart object and smart object group instances that will be developed during the end-user development through the smart object visual programming editor (see Figure 9.20).

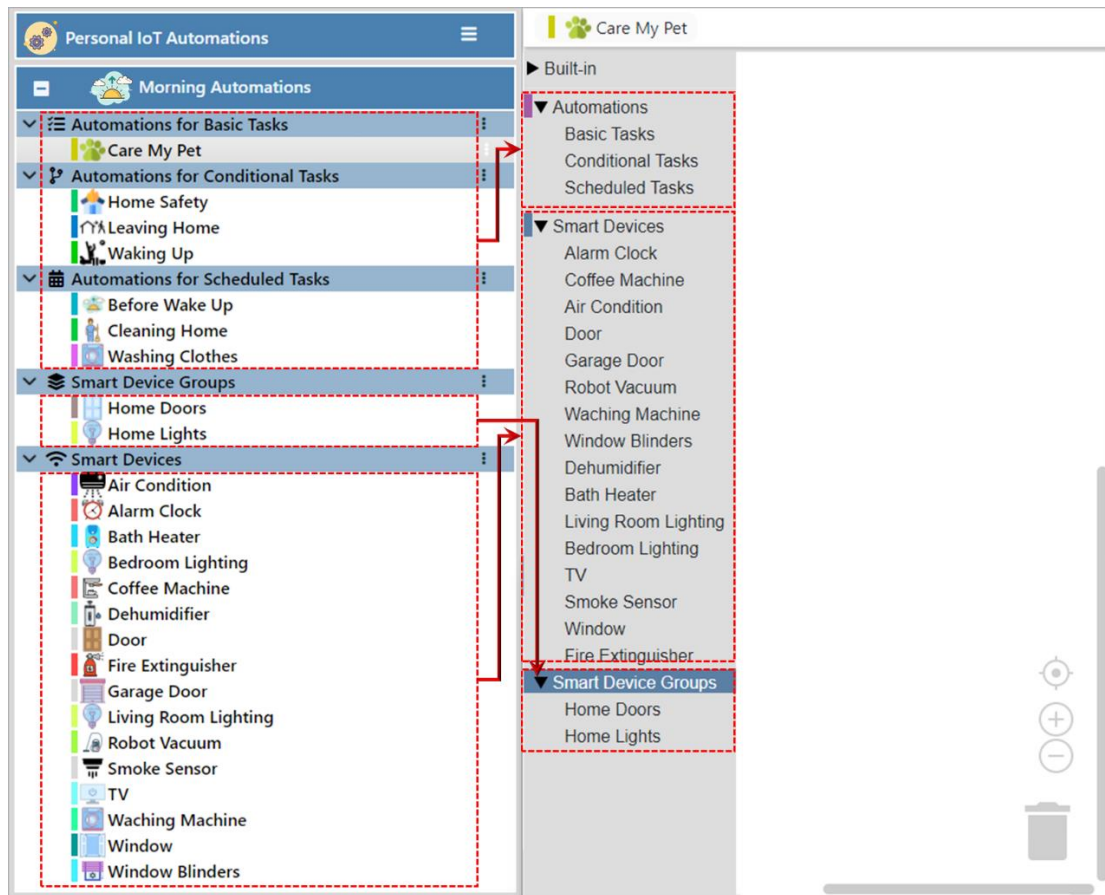


Figure 9.20. Automations for 'Basic Tasks' configuration of *Blockly* editor's toolbox.

This category of project elements can be used for pure automations in order to set a number of instructions for the smart objects. For example, a program can be that after running the project of automations, the blinders will close, the smart lights will turn to the club mode and the smart hi-fi will start the user's favorite club music. However, thanks to the choice of starting the basic automations manually, the end-user developers are able to use these automations combined with other project elements by using the visual programming blocks in order to start the automations.

9.5.2.5 Automations for Conditional Tasks

The second category of project elements for automations is targeted at the automations which are executed based on conditions of the smart object values. This is one of the main categories for the end-user development of IoT automations. When creating conditional automations, the end-user developer is able to define if this project element will start automatically when the execution of the project will start, or

if the automation will start manually through the visual programming block element, as it happens with the basic automations.

The project element of the '*Automations for Conditional Tasks*' has developed a pure template (see Figure 9.15). In addition, the Blockly editor instances configuration for the conditional tasks includes all the Blockly blocks that are defined for the basic tasks (i.e., built-in blocks, blocks for automations handling, blocks for smart objects and smart object groups handling).

Additionally, the *Blockly* editor's toolbox for conditional tasks includes two more categories of blocks (see Figure 9.21). The first category is the conditional blocks which was earlier discussed (see section 9.3) and is the main category for the end-user development of the conditional tasks. In this context, the toolbox includes the conditional blocks in two versions. In the first version, the conditional blocks are defined as root blocks of the development (i.e. no siblings are able to be added). However, in order to enable the development of more complicated conditional automations, the second version of blocks is defined by including top-down input for the conditional blocks. In this direction, the end-user developers are empowered to develop conditional tasks which include inner conditional tasks. Moreover, in conditional tasks, there are included blocks for scheduled tasks which can be inserted as children of the conditional blocks without including their top-level blocks in the '*Scheduler*' category (see on the right of Figure 9.21). In this context, we enable the end-user developers to program more complicated conditional automations by combining conditional tasks that are able to start scheduled tasks during the runtime of the project, when they are triggered.

This category of project elements can be used for automations that will be based on conditional events which happen in a smart environment by using smart objects. Using the available conditional blocks, the end-user developers are able to build simple conditions (e.g. temperature environment changes) and more complicated conditions by using the provided *AND*, *OR* operators. Moreover, they are able to handle when and how many times the conditional automation will be executed based on the blocks (e.g. for N times when the condition is true, after N times when the condition is true). In addition, their execution can be handled through the start/stop conditional automations, which are available too.

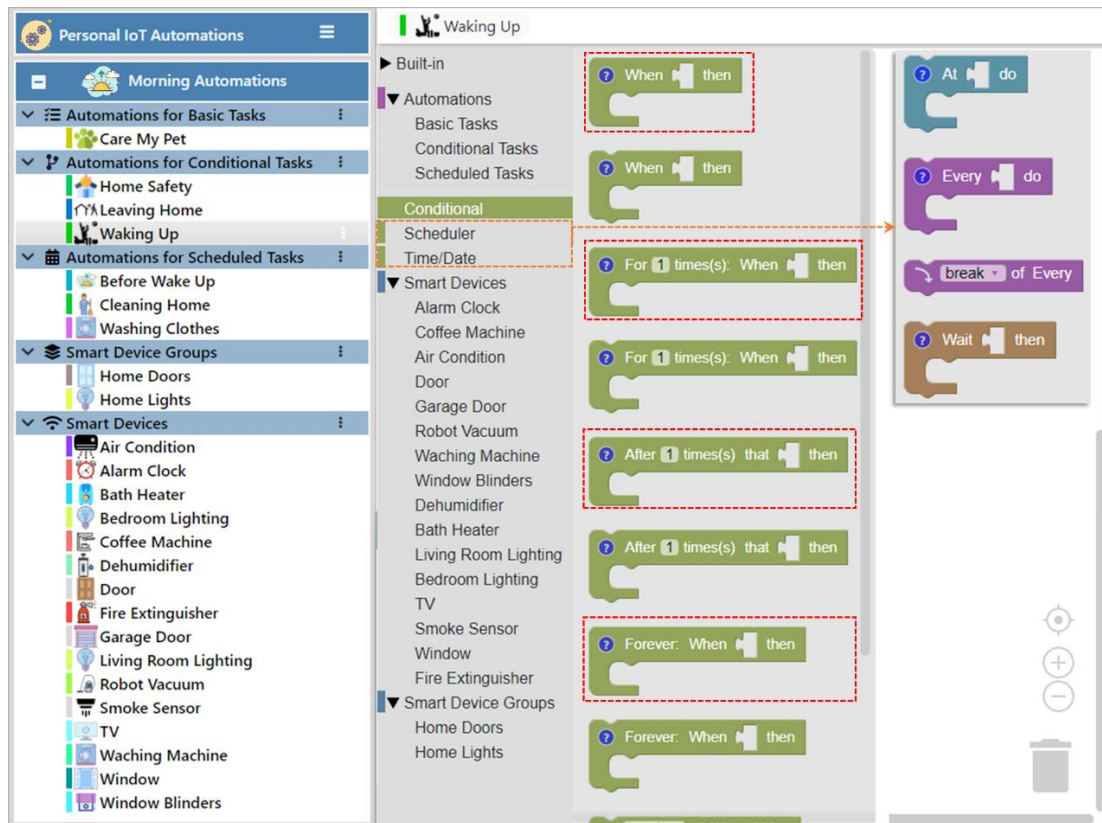


Figure 9.21. Automations for 'Conditional Tasks' configuration of *Blockly* editor's toolbox.

9.5.2.6 Automations for Scheduled Tasks

The last but not least category of project elements for automations is targeted at the automations which are executed based on scheduled events. By creating scheduled tasks using a specific time and date, the end-user developer is able to define automations in the form of a calendar. In addition, waiting for a specific time period in order to apply the automations is another available scheduled task. The end-user developer is able to choose if this type of project element will start automatically when the execution of the project will start, or if the automation will start manually through the visual programming block element, as it happens with the conditional tasks and the basic automations.

The project element of 'Automations for Scheduled Tasks' has developed a pure template (see Figure 9.15). In addition, the *Blockly* editor instances configuration for the scheduled tasks includes all the *Blockly* blocks that are defined for the basic automations (i.e., built-in blocks, blocks for automation handling, blocks for smart objects and smart object group handling).

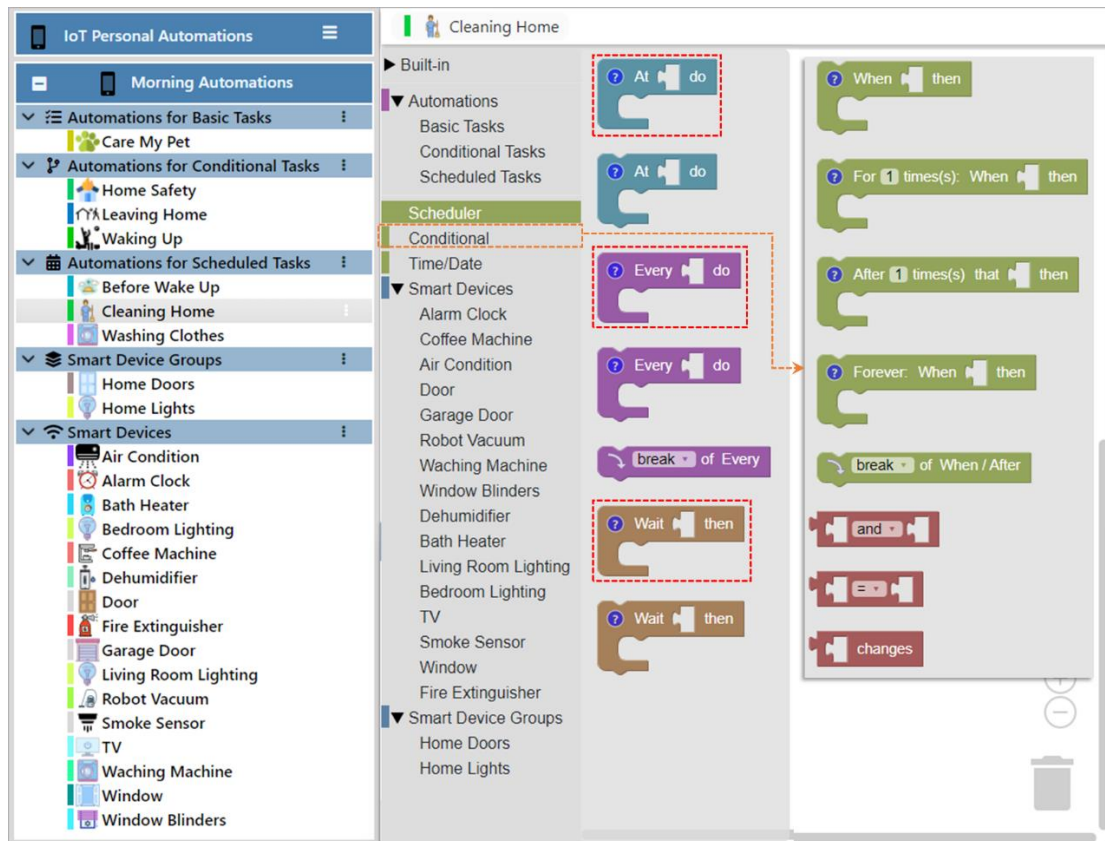


Figure 9.22. Automations for 'Scheduled Tasks' configuration of Blockly editor's toolbox.

Additionally, the Blockly editor's toolbox for scheduled tasks includes three more categories of blocks (see Figure 9.22). The first two categories of blocks are focused on the end-user development of the scheduled tasks which were earlier discussed (see Figure 9.13). In this context, the toolbox includes the scheduled blocks in two versions. In the first version, the scheduled blocks are defined as root blocks of the development (i.e., not siblings are able to be added). These blocks are playing the dominant role in the development of scheduled automations. The second version includes top-down input for the blocks. Using these blocks, the end-user developers are able to define more complicated scheduled automations. Moreover, in this context, there are available conditional blocks (see Figure 9.22) with top-down inputs in order to be used as inner blocks of the main scheduled blocks.

This category of project elements is able to be used for automations that will be based on scheduled events which contributes in smart IoT automations by using the calendar, the time, time periods and the smart objects. Using the available scheduled blocks, the end-user developers are able to build simple conditions (e.g., every two days clean up the house) and more complicated scheduled automations by using the

provided blocks. Moreover, they are able to handle when and how many times the scheduled automation will be executed based on the blocks (i.e. break, continue, branches, etc.). In addition, their execution can be handled through the start/stop conditional automations, which are available too.

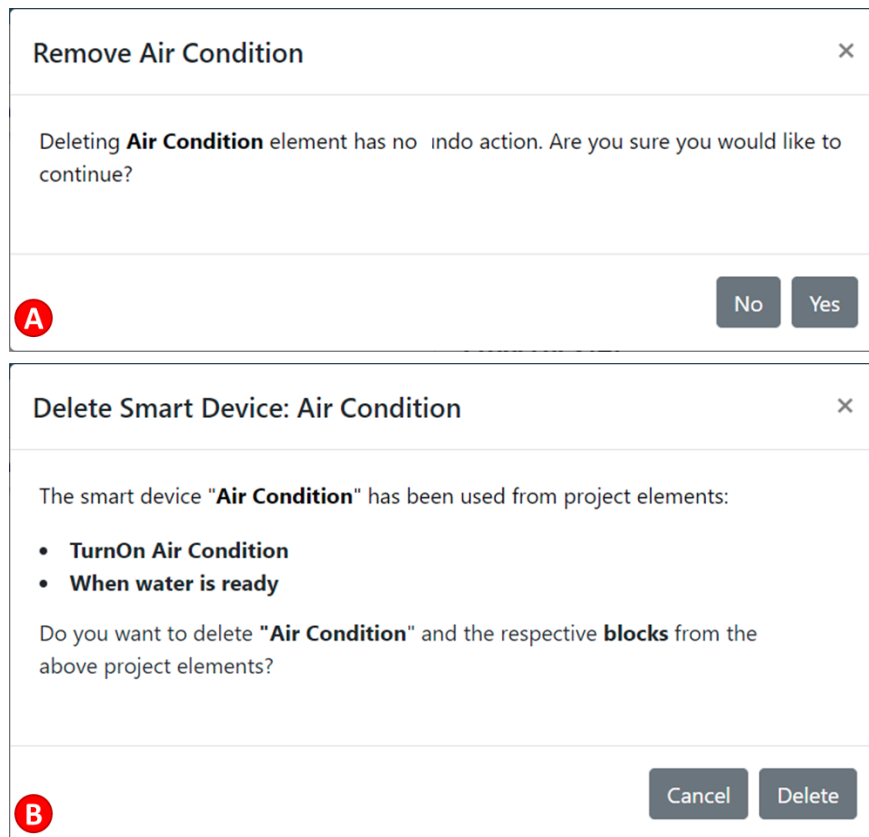


Figure 9.23. Dialogues in case the end-user chooses to delete a Smart Device.

9.5.2.7 Handling Dependencies

During the end-user development process, the end-users create, edit and maybe delete project elements. As earlier mentioned, there are dependencies between the project elements. Each application domain might like to follow different rules for the deletions of project elements, the editing, etc. In this context, the *Blockly Studio IDE* enables the application domain author to develop the behavior before and after the actions of creation, editing and deletion. Moreover, based on the API which is provided by the *Project Manager*, the *Blockly Editor* and other visual programming editors, we are able to retrieve which are the project element dependencies, the responsible visual programming language elements, etc.

In the case of the IoT Automations domain, we decided to open a dialogue in case of deleting a smart device or a smart device group, which will inform the end-user developer of the project elements that this project element has dependencies with (see tag B of Figure 9.23) and the respective visual programming language elements that will be removed from these project elements. In case there aren't dependencies with other project elements, the end-user developer is asked to confirm their decision to remove the smart device or the smart device group.

In both cases, the end-user developers are able to confirm or cancel the action. Using the function which is called by the *Project Manager* after the action (authored by the domain author), we developed the respective functionality of removing the respective blocks from the dependent project elements.

In addition, in this function, having the knowledge of the completed delete action, the smart object editor is able to post the respective delete signal. Based on this signal, the *Domain Manager* is able to handle the deletion of the visual programming language element. In this context, all the respective project elements have updated toolboxes in their *Blockly* editor instances.

9.6 Running Smart Automations

As earlier mentioned, the runtime environment of the *Blockly Studio IDE* requires the development of the entry point script (i.e. run-script) in order to execute a project of a specific application domain. In addition, it is required by each visual programming editor instance to generate the respective JavaScript source code or the run time environment data from the visual code which has been created by the end-user developers. In this context, we developed the respective generator function for the smart object visual programming editor and developed the respective JavaScript source code for the execution of each of the *Blockly* blocks.

9.6.1 Execution of IoT Automations

When the user starts the project execution, the runtime environment requests to get the project's runtime environment data. The project manager iterates each of the constructed project elements. Each of them includes a list of visual programming editor instances. For each one, it is requested to generate runtime source code or data.

In the context of IoT automations, there are two different visual programming editors, the smart object editor and the *Blockly* editor.

9.6.1.1 Interacting with Smart Objects

In the case of the smart object editor, the generator for runtime produces data is constructed for each one of the visual programming language elements (i.e., ‘Smart Device’ and ‘Smart Device Group’ instance). These data are used when the execution starts in order to communicate with the respective smart objects. By communicating with the smart objects, the application gets the current properties’ values of the smart devices (pre-caching data) and adds observers for them updates in case there are changes in their values. The communication mechanism and the smart objects’ data are available in the global runtime environment of the application (developed on the top of the run-script).

Based on this implementation, respective source code is developed to be generated by the *Blockly* blocks which handle the behavior of the smart devices and smart device groups (see Figure 9.9, Figure 9.11). In particular, getters are handled by using the pre-cached data instantly instead of requesting the *IoTivity* middleware. This improves the performance of the getter functions which is critical for the execution of conditional blocks (see next paragraph). In the context of setters and actions of the smart devices and smart device groups, we have developed them using promises in order to have sequential execution (i.e. using `await`) of these instructions and avoid async function calls.

During the project execution, the communication between the application and the smart devices may be lost. In this case, the execution of the application is not able to continue and a dialogue warns the user about the issue and the ending the of execution process. Additionally, a specific smart device may be disconnected during the project execution. In this case the application notifies the users and they are able to decide if they will stop the execution or they will continue in case this smart device not affects their automations (see Figure 9.24). Similarly, warning message is popped-up in case a request in smart device could not be accomplished. For example, water is empty in coffee machine and it is not able to prepare coffee.

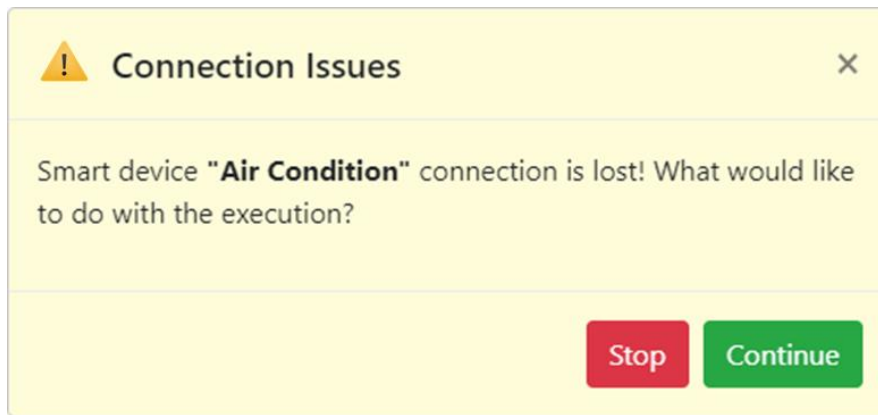


Figure 9.24. Dialogue on connection issues of the smart devices.

9.6.1.2 Running Conditional and Scheduled Tasks

Starting the execution of the project, there is a list of conditional automations which are developed through the earlier mentioned *Blockly* blocks (see Figure 9.12). When the defined conditions of the blocks are evaluated to true, their inner blocks (i.e., children) will be executed. Conditional blocks in IoT automations project could be numerous in a project as they are basic in the context of smart automations. As a result, the performance of their execution has to be efficient. Our approach is based on the *setTimeout* [149] and *setInterval* [150] functions, which are provided by JavaScript. In particular, based on the fact that a change of a state can be delayed of observing it from the people (e.g., less than 0.5 second), we developed a repeatable function call. In addition, there is a global list of conditional defined functions. When a conditional block starts, its respective function is added to the global list, while when a conditional block is deactivated, it is removed by this list. The repeatable function call, iterates and invokes all the conditional functions. If the condition is evaluated to true, the inner body of the conditional block is executed. Based on this technique, we have used only one *setInterval* for all the conditional blocks. In this context, the earlier mentioned pre-caching of values for getters is extremely important for the implementation because the request of middleware is incomparably more expensive.

In the case of scheduled blocks, real time and date is used. When the project execution starts, we calculate the specific time for all the activated scheduled blocks that have to be triggered and we use *setTimeout* in order to start the execution of their inner body. In addition, global data are saved including the *setTimeout ID* in order to

use them in case there is an instruction of stop or pause for the specific scheduled automation or the whole project.

9.6.2 User-Interface of IoT Automations

In running the applications of IoT automations, the sole interaction of the user with their automations, is the input-output console which is provided by the *Blockly Studio IDE*. Through the input-output console, the end-user developer is able to ask input for smart device properties and print the values of smart device properties. However, this is not adequate in order to have a live monitoring of the IoT automations, which are developed through the visual programming environment we described above.

In the case of software developers, one of the main development tasks would be the user-interface programming of the IoT automations. In the case of visual end-user development, existing approaches for building user interfaces for the applications are the *WYSIWYG* editors (or screen designers). In this context, we could incorporate one *WYSIWYG* editor for the application domain of IoT automations in order to enable the end-user developers to program user interfaces for their applications. However, building user interfaces for IoT automations and connecting with the visual code of automations could be unmanageable or extremely difficult for novices. Moreover, developing such user interfaces could be extremely time costly for users that would like to develop simple automations.

In this context, we developed a full-scale graphical user-interface runtime environment for monitoring and interacting with the smart devices, the conditional and scheduled automations, etc. (see Figure 9.25). Also, this environment can be configured by the user. As discussed on section 6.4, the *Blockly Studio IDE* is able to host external domain graphical user-interfaces for their project execution. In particular, the application domain authors are able to initiate and handle these views through the run-script and the code generation of the domain visual programming language elements which are developed using the visual programming editors. In the following subsections, we describe each of the graphical user-interface parts of the IoT automations runtime environment.

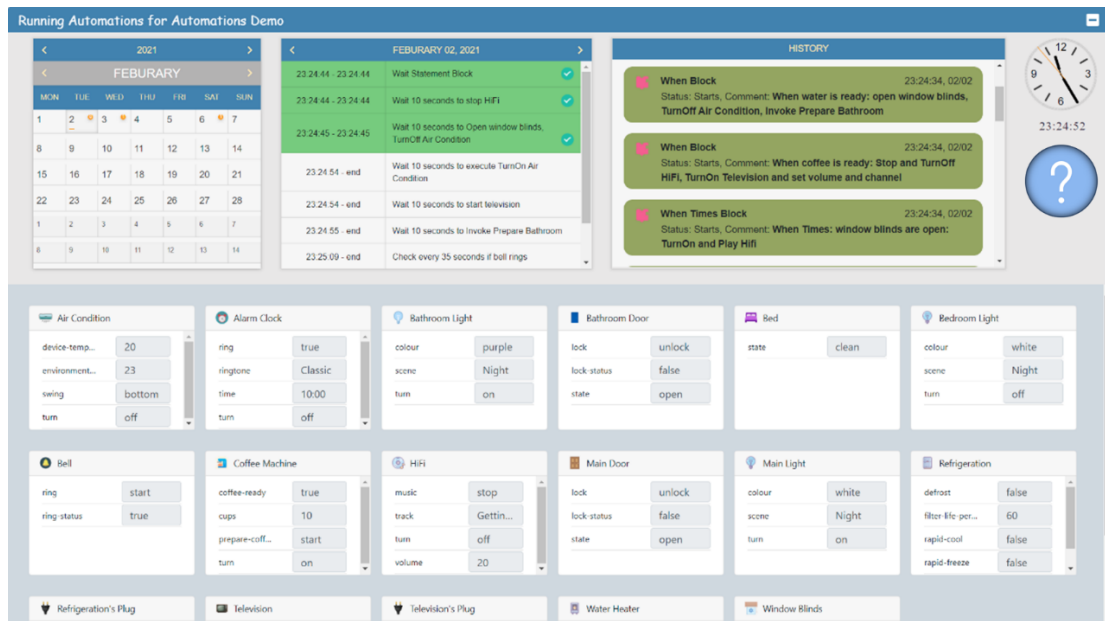


Figure 9.25. Runtime environment for IoT automations.

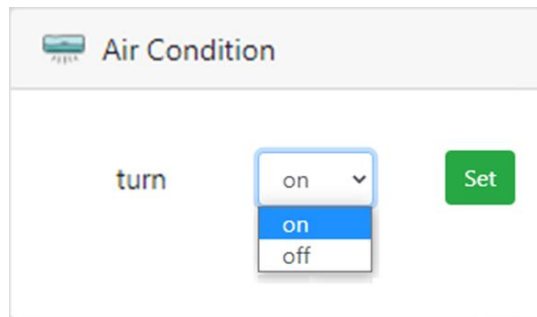


Figure 9.26. Request to set input in property of a smart device.

9.6.2.1 Smart Devices View

One of the main views for the runtime environment of IoT automations is the smart devices monitoring view. The user-interface for smart objects displays their state for each of their properties live. In case a smart object's property value changes, the specific property view is highlighted with blue color (see red arrows of Figure 9.27) for some seconds. The property values' changes are able to arise via the execution of the visual code developed by the end-user or via the device functionality. In this context, we depict the source code and device mode changes by using different icons next to their property values. The users are able to view the history of the values for the properties by hovering on specific value boxes (see environment temperature of the Air Conditioning in Figure 9.27).

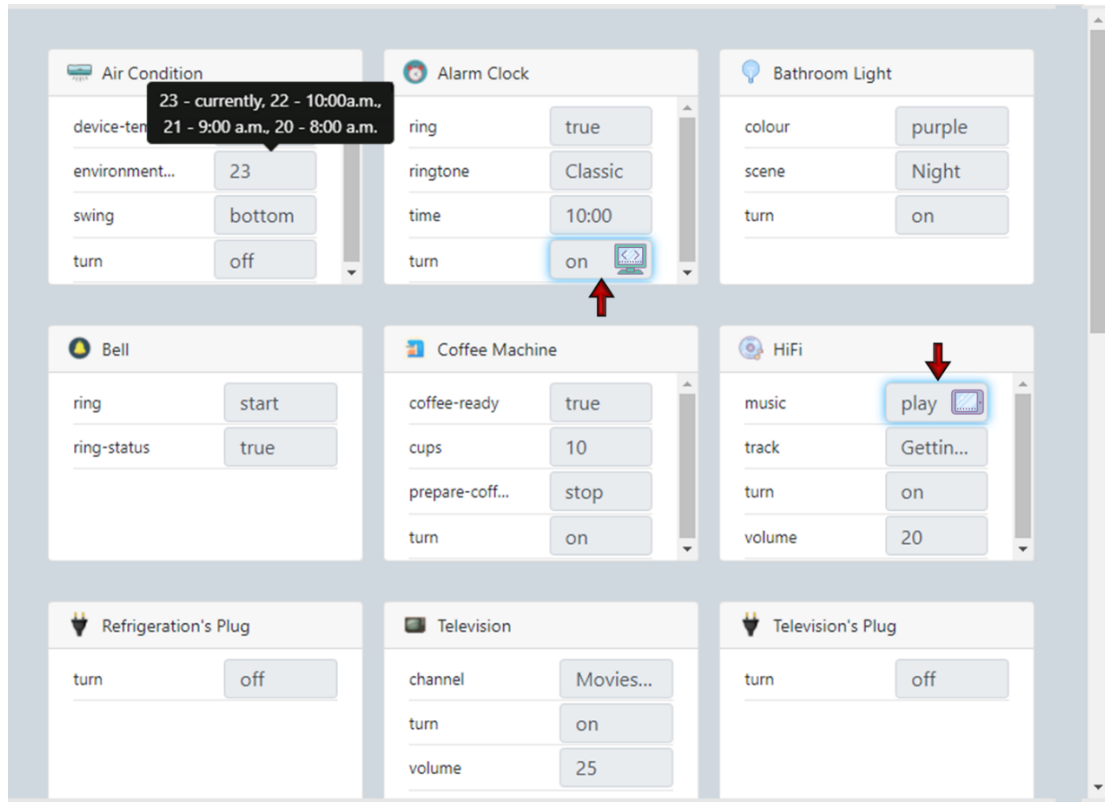


Figure 9.27. Display of the Smart Devices at runtime environment.

In addition, when the blocks which ask property value as input from the user (see tag D of Figure 9.9) are executed, a dialogue opens in order to insert a value for the specific property of the smart device. The user-interface for the input is related with the type of the property. For example, in case the input is of enumerated type, a select input will be depicted (see Figure 9.26), in case of text, a text input will be depicted, etc.

Furthermore, the end-user developer is able to choose if the runtime environment will display an additional button on the header of each smart object through which the users will be able to click them and change values for the properties of the smart objects manually during the project execution. In particular, when the user clicks the control button of a smart device, a dialogue opens with editable view of the smart device's properties (only if they are not read-only, see Figure 9.28).

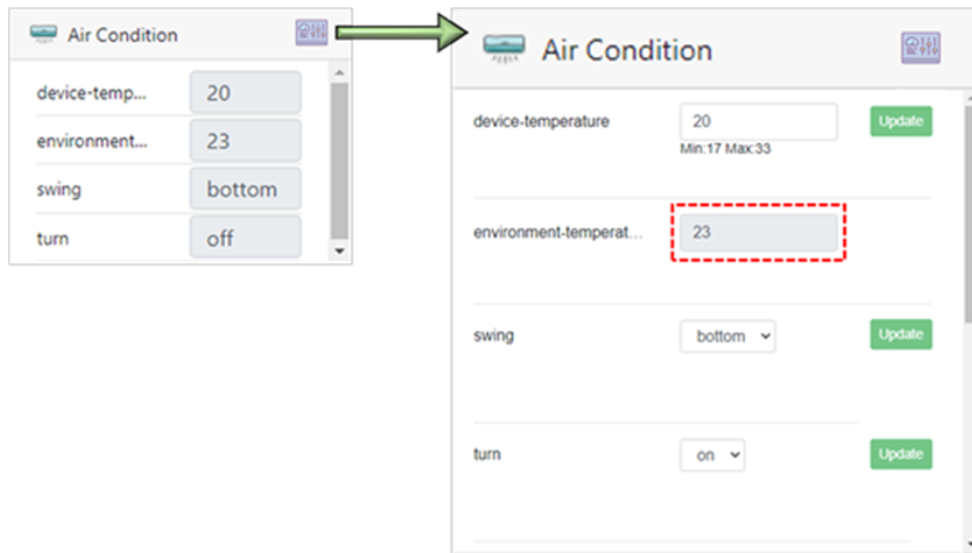


Figure 9.28. Enabling control smart devices during the project execution.

9.6.2.2 Calendar View for Automations of Scheduled Tasks

As earlier discussed one of the main categories of blocks, in the context of IoT automations, are the scheduled blocks. In a project of IoT automations, the end-user developers could use numerous scheduled blocks. However, when the application runs, there is no feedback if these blocks have been developed correctly or not. In addition, it is difficult to understand by just waiting for them to be triggered during the project execution. Moreover, when a project runs the user has no feedback for which scheduled automations are going to happen and when.

2021						
FEBRUARY						
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

FEBURARY 02, 2021	
23:24:44 - 23:24:44	Wait Statement Block ✓
23:24:44 - 23:24:44	Wait 10 seconds to stop HiFi ✓
23:24:45 - 23:24:45	Wait 10 seconds to Open window blinds, TurnOff Air Condition ✓
23:24:54 - end	Wait 10 seconds to execute TurnOn Air Condition
23:24:54 - end	Wait 10 seconds to start television
23:24:55 - end	Wait 10 seconds to Invoke Prepare Bathroom
23:25:09 - end	Check every 35 seconds if bell rings

Figure 9.29. Monitoring scheduled automations in the runtime environment of IoT automations.

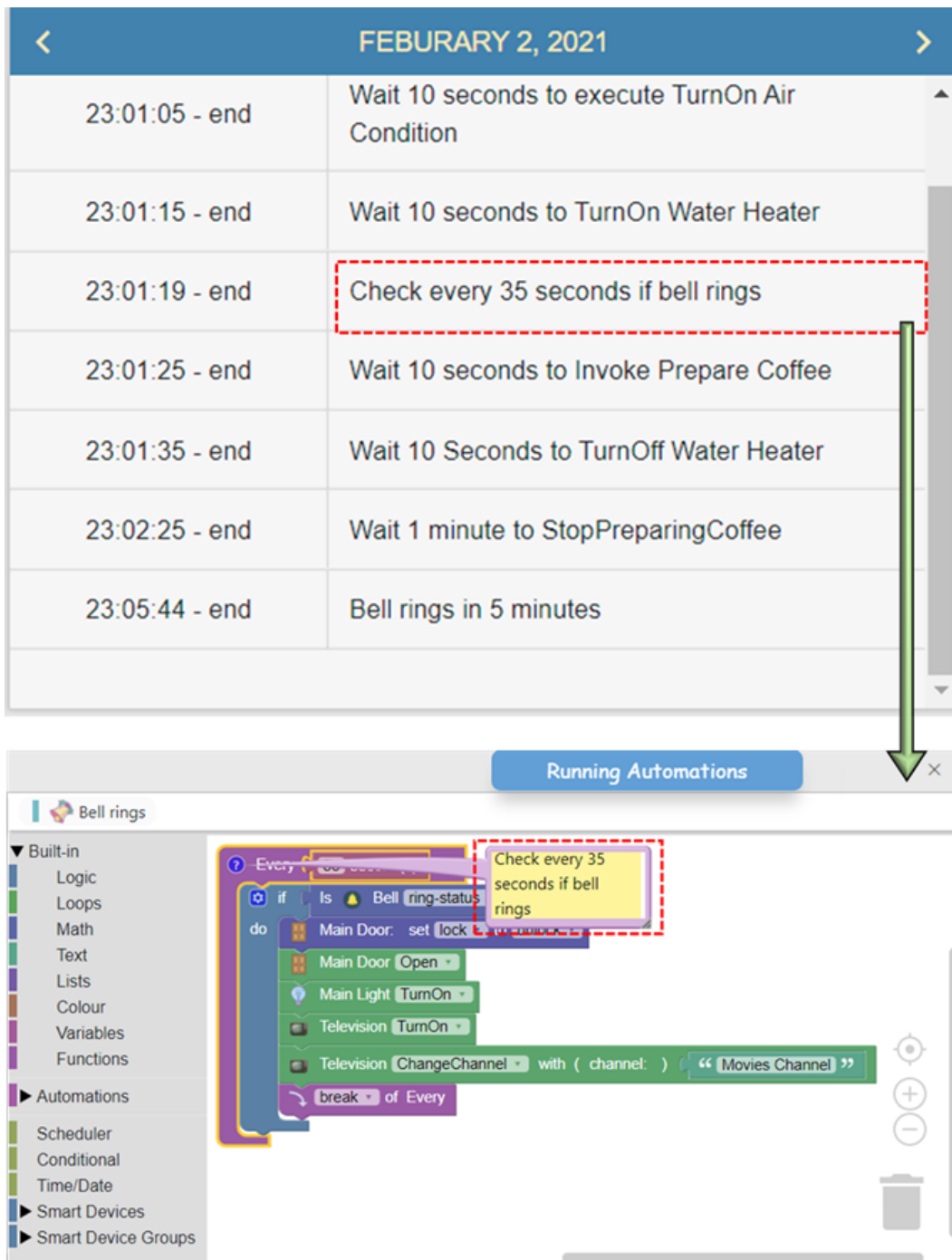


Figure 9.30. Browsing project elements that includes the scheduled blocks.

In this context, based on the execution of the scheduled blocks (see Figure 9.13) we developed an appropriate view for the scheduled tasks which has been incorporated in the runtime environment of IoT automations. In particular, we have developed a calendar view which is separated in two views: the calendar and the view of scheduled tasks (see Figure 9.29). As earlier mentioned, when the execution of an application starts, the date and time are calculated for each of the scheduled blocks

that are executed. In this context, we added extra source code in the blocks' code generation, in order to add the scheduled events in the calendar, using the API of the runtime calendar we have developed.

However, it would be pointless to add scheduled events in a calendar without appropriate description or notes. In this direction, we enable the end-user developer to add description and/or notes for each scheduled block by using the *Blockly* block comments (see Figure 9.13). When the scheduled events are added in the calendar, we also insert the corresponding block comments as their description. Additionally, the users are able to click on the table elements with the scheduled events and the runtime is folded while the respective project element of the scheduled task automation opens and the scheduled block is highlighted with its comment open (see Figure 9.30). When the scheduled tasks are completed, they are marked with green color and checked as completed by filling the time and date that they finished. During the project execution, the scheduled automations and scheduled blocks are able to start/stop. In this context, the respective time and date that they will be triggered is calculated and the calendar is refreshed.

9.6.2.3 History View

As earlier discussed, in the context of IoT automations, one of the most important categories of automations is the conditional tasks which are based on the conditional blocks (see Figure 9.12). Another one of the main categories of blocks is the scheduled blocks. During the execution, the users are not able have feedback if conditional blocks have been triggered or not by only monitoring the changes of values of the smart device properties.

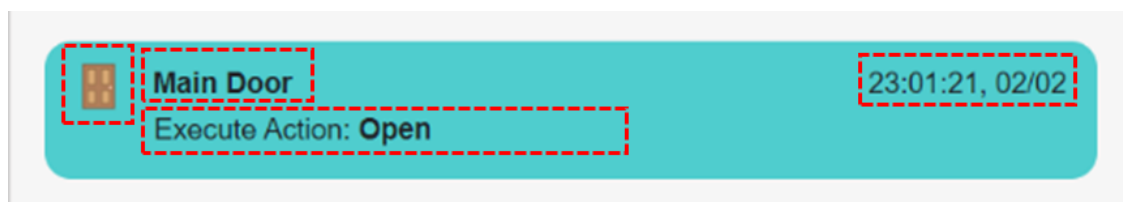


Figure 9.31. Interactive bubble which depicts action of the history panel view.

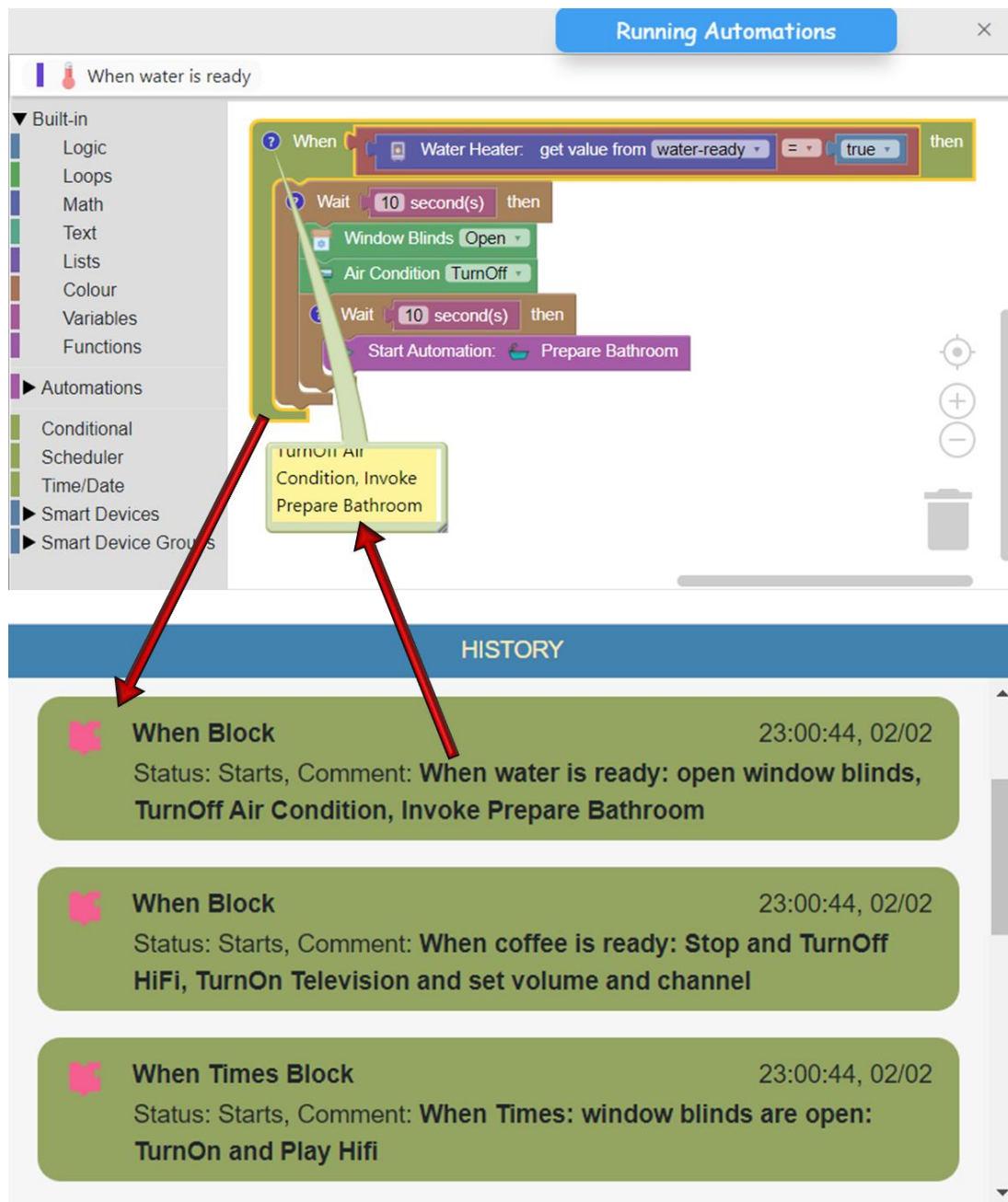


Figure 9.32. Monitoring conditional tasks and browsing respective visual code snippets.

In this context, the users are not able to know what exactly has happened to the IoT automation and what could happen during the execution through the conditional tasks. Moreover, users are watching changes of the smart devices through the smart devices display (see red arrows of Figure 9.27) and they are informed if this change has arisen through the IoT automation code or not, but they are not able to know through which visual code part was executed.

In this context, based on the execution of the scheduled blocks (see Figure 9.12) we developed an appropriate view for the conditional tasks which has been incorporated in the runtime environment of IoT automations. In particular, we have developed a history view which includes facts that have happened in the IoT automation during the execution (see bottom part of Figure 9.32).

These facts are presented in the form of bubbles which includes title, icon, description, date and time and background color (see Figure 9.31). Each of these bubbles enables the users to browse the respective visual code which caused this action by clicking on it (see Figure 9.32). The user is able to add comments in the conditional blocks and the history panel uses them in order to display them when the block is executed. Moreover, the background color is defined respectively. Particularly, in the case of an action bubble which is rendered for a smart device, the background color is the given color on defining this smart device.

9.6.2.4 Explaining Why Automations Occurred

During the execution of IoT automations, the users could wonder how an automation has been arisen. For example, in the case of the Ambient-Assisted Living, the family and caregivers can develop automations that will be used by elderlies. In this context, elderlies may wonder why an automation happened. As result, it would be extremely useful for IoT Automations to be able to answer questions that may arise from the users. However, the system is not able to explain what happened based on the visual code which has been executed.

Using the above user-interface of the IoT automations, they are able to monitor what happened and why it happened by using the comments that have been added in conditional and scheduled Blockly blocks, in order to present them in the calendar and history view. However, in the body of these blocks (i.e., children) different actions could be executed through branches, loops and function calls. As result, comments of the scheduled and conditional blocks are not able to answer exactly what happened in a specific automation.

In this context, we use the earlier mentioned approach (see section 7.8) of defining extra helpful blocks as annotations which are used for explanations during the project execution. Using explanation blocks, the end-user developers are able to annotate

which actions are going to happen in the following visual code snippets. During the project execution, the users are able to use the help button (positioned under the clock in the left of Figure 9.25). A pop-up dialogue opens which resembles the history view (see Figure 9.32) and presents the messages of the explanation blocks which have been executed currently in the form of bubbles, informing the users why the automations happened.

Additionally, the list of explanation blocks identities is pinned in the parent calendar and scheduled blocks. During the execution, when an explanation block is executed, the explanations data (i.e. end-user developer messages) includes their identity, in order to render information, in case the user asks for what happened. In this context, we are able to identify the parent blocks, and we add an extra help button in each of the parent blocks in case they have to present more information about the executed explanation blocks which are related (see Figure 9.33).



Figure 9.33. Filtering executed explanations per scheduled (top) and conditional (bottom) automations by enabling info button that opens dialogue which present them separately.

9.7 Debugging and Testing Facilities for IoT Automations

As earlier discussed, the *Blockly Studio IDE* includes debugging through its visual debugger which supports a full-scale toolset (i.e., stepping, tracing, watching, breakpoints, conditional breakpoints, etc.). However, in the case of IoT automations there are several arising issues that make the debugging process impractical or even impossible.

Particularly, IoT automations are included by smart devices' behavior handling, conditional and scheduled automation tasks. Scheduled automations could be triggered for long periods of time. In this context, the end-user developers can't wait for these time periods in order to identify that automations work correctly. A solution

could be the editing of the time periods in the corresponding visual code in order to shorten the waiting time. However, this requires extra effort from the end-user developer and may cause errors in time periods, when completing the development process. Additionally, conditional automations are based on the smart devices' values state. For example, a conditional automation could be “*When smoke sensors warning Then alarm starts*” or “*When environment temperature changes Then air-conditioning starts*”. In this case, the end-user developers are not able to debug their automations. Additionally, during the project execution the smart devices are affected by the program (i.e. change their properties, requests for actions), while a debug process may include several starts and stops of the visual debugger's execution.

In this context, we developed facilities to simulate the smart devices, their behavior, the date and the time that the automations will be executed. In particular, we replace the real smart devices with simulated in the context of the debugging process. Additionally, the end-user developers are able to create tests of expected values of the properties of the smart devices at specific date & time or at specific conditions during the project execution. As mentioned in section 7.9, Blockly Studio IDE supports the extra domain-specific user-interface runtime view through the debug-script which has to be developed by the application domain authors and the independent development of applications which communicate with the IDE. In particular, using the debug-script we initialize and handle the simulation facilities, while we have authored the code generation of the domain Blockly blocks in order to cooperate with the facilities. In the following subsections, we present each of the simulation features we developed in order to contribute to the debugging process in the context of IoT automations.

9.7.1 Simulating Smart Environment

Starting the debugging process, the user interface of the runtime environment which is displayed on release mode (see Figure 9.25) has been modified. In particular, there are extra elements for the simulation facilities (see Figure 9.34). Next to the history view (see Figure 9.32) there is a test control panel in which the user is able to view and handle the simulation tests for the debugging process. They are able to edit specific tests by clicking on them (bubbles), view if they have been executed (see green check mark in the grey bubbles of Figure 9.34). Additionally, they are able to view all authored tests and manage them by clicking on the folder button located on the

bottom of the test control panel. When you click this button, a dialogue opens (see Figure 9.35) presenting the list of simulations for the smart objects; behavior and the tests of expected values of properties of smart objects at specific time & date or condition. The end-user developers are able to view, add or remove a test.

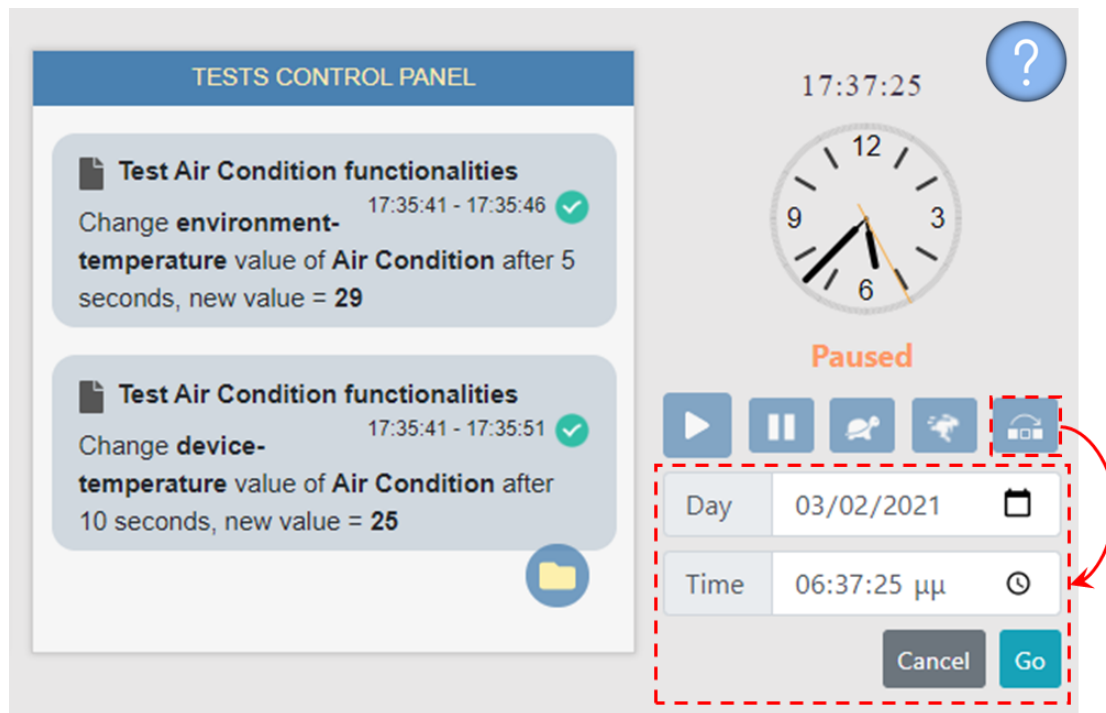


Figure 9.34. Simulation Environment View: tests control panel (left), date & time simulation (right).

Moreover, except of the simulation of the smart objects' behavior, the time and date of the application runtime are also simulated (see right of Figure 9.34). The end-user developers are able to stop, start the time, go slower by using the turtle button or go faster by using the rabbit button. Additionally, they are able to go at a specific date and time. Particularly, when the end-user developers click the button on the right of the rabbit button, it opens a user interface that allows them to select specific date and time. This action allows the end-user developers to see what will have happened by executing all the simulated actions and the visual code of the automation. The end-user developers are able to pause the time and view all the actions in history view, the scheduled tasks in calendar view and the current values (and the history values) of the smart objects' properties.

As earlier mentioned, we have two different types that contribute to testing the application: the simulation of smart devices and the tests of expected values. In the following two subsections, we present them.

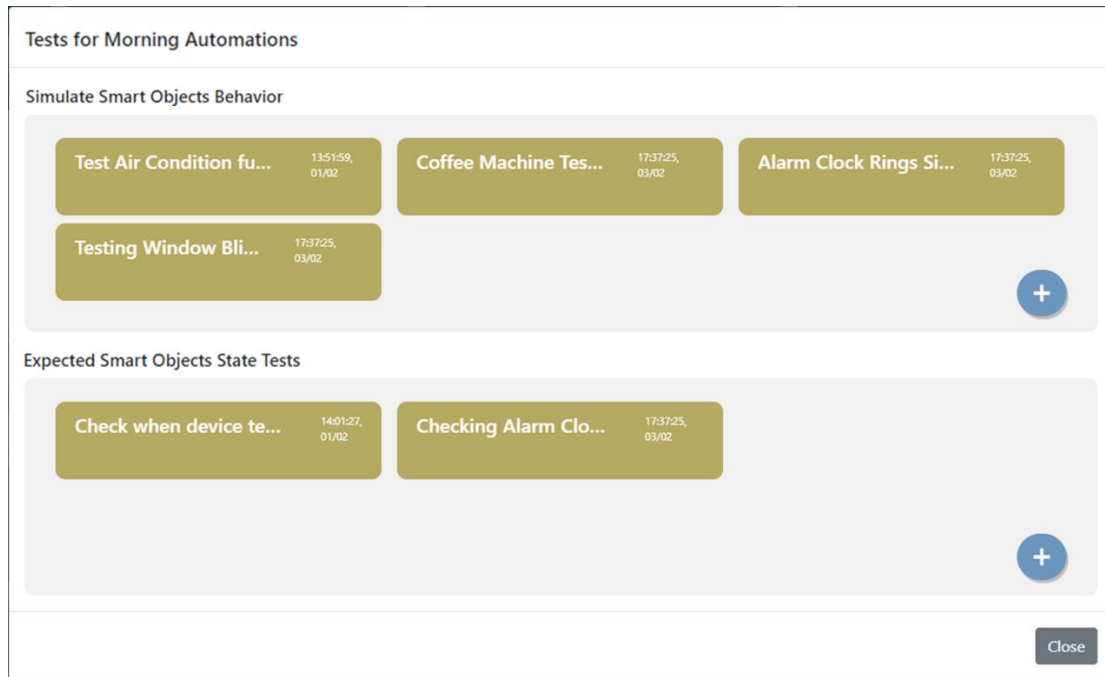


Figure 9.35. Managing Simulation Behavior and Expected Values Tests.

9.7.2 Simulating Smart Devices

When replacing the smart devices with simulated devices, there are issues that have to be addressed. Firstly, the functionality of the actions of the smart devices (e.g., coffee machine includes a *prepareCoffee* action) are not known to the simulation system. As a result, during the runtime, actions are not able to be executed. In this context, we developed the appropriate infrastructure for the end-user developers to be able to program the actions of the smart devices.

In particular, the end-user developers are able to program each of the actions that are provided by the smart devices by browsing the smart device and choose the debug options button which is located in the right of each action (see Figure 9.36). When the user clicks on the button, a dialogue opens with a *Blockly* editor workspace instance. This instance is configured by isolating the specific smart device *Blockly* blocks in the *Blockly* toolbox (see Figure 9.36).

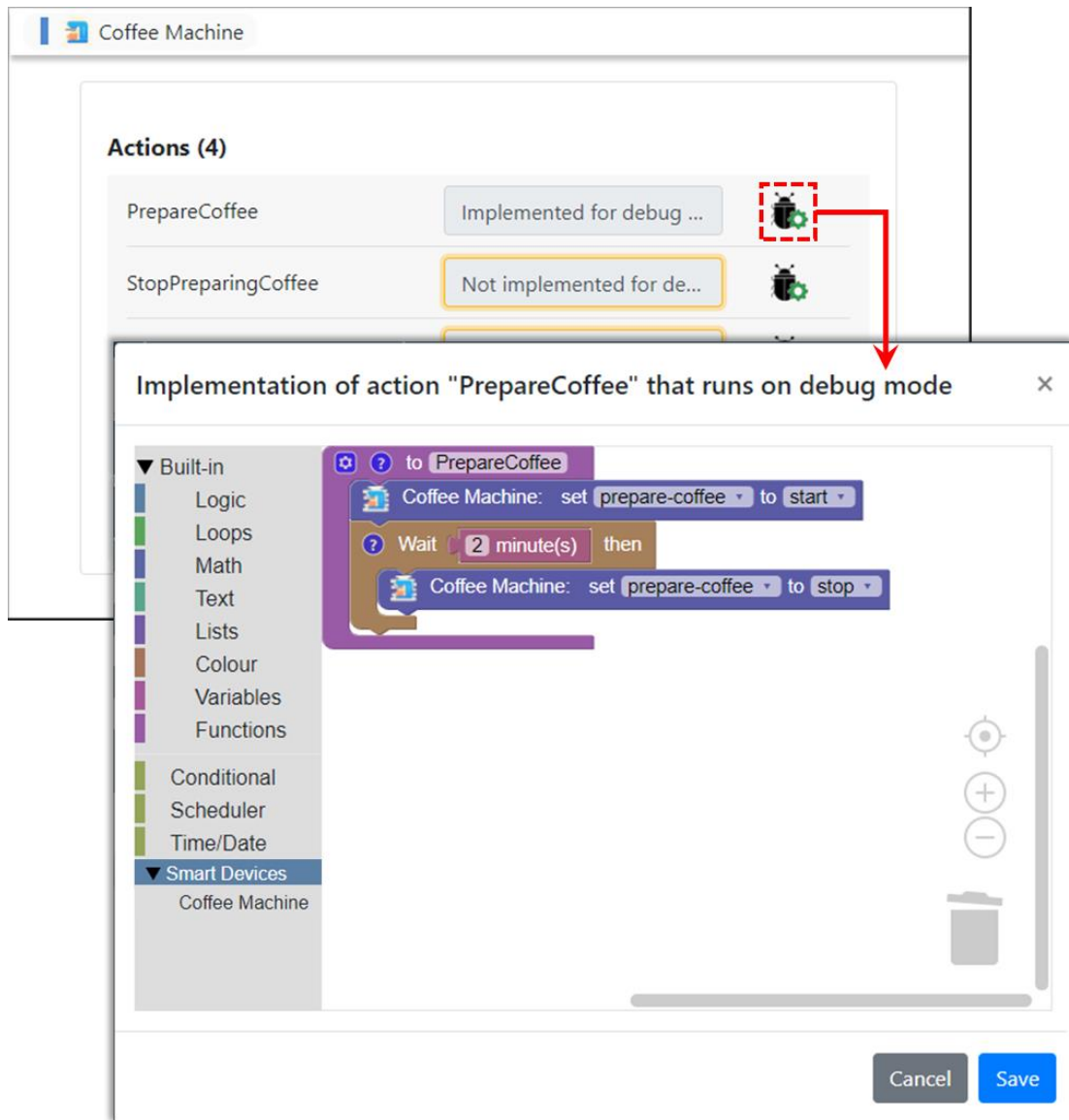


Figure 9.36. Simulating smart device actions for debugging purposes.

Additionally, a function is predefined with a given name, same with that of the action's name. In this context, in case there are input arguments in the action of the smart object, the predefined function is constructed with these input arguments and there are created variables in the Blockly toolbox which refer to these arguments. Moreover, the end-user developers are able to simulate the action by using scheduled blocks as it is shown in the example of Figure 9.36. During the debugging process, when a smart device action is requested, the respective end-user developed function will be called. The end-user developers are able to add breakpoints and in case a breakpoint is triggered, the dialogue of the specific Blockly editor workspace opens and highlights the block which has currently paused the execution, as it happens in classic Blockly editor workspaces of the project. In case there is no implementation

for some smart device's action, the end-user developers are warned when they try to start the debugging process. Moreover, when they browse a smart device, the actions which are not simulated are highlighted, as displayed for the *“StopPreparingCoffee”* action in Figure 9.36.

Having replaced the smart devices with simulated devices, the system has to enable simulation and handling their properties and actions at time periods. In this context, we enabled the end-user developers to program simulation smart device functionality tests (see top of Figure 9.35). When the end-user developer chooses to add a new test, or edit an existing one, they are able to choose specific time periods that they are able to set changes for each of the smart devices, regarding one or more properties as it is presented in Figure 9.37. Moreover, they have to set a name for the test which is presented on the tests management page and a color which is rendered in the execution, in the test control panel. Based on the time periods of these simulated tests and the simulated time & date of the debugging process, the behavior of the smart devices changes in order to enable the end-user developers to test their automations.

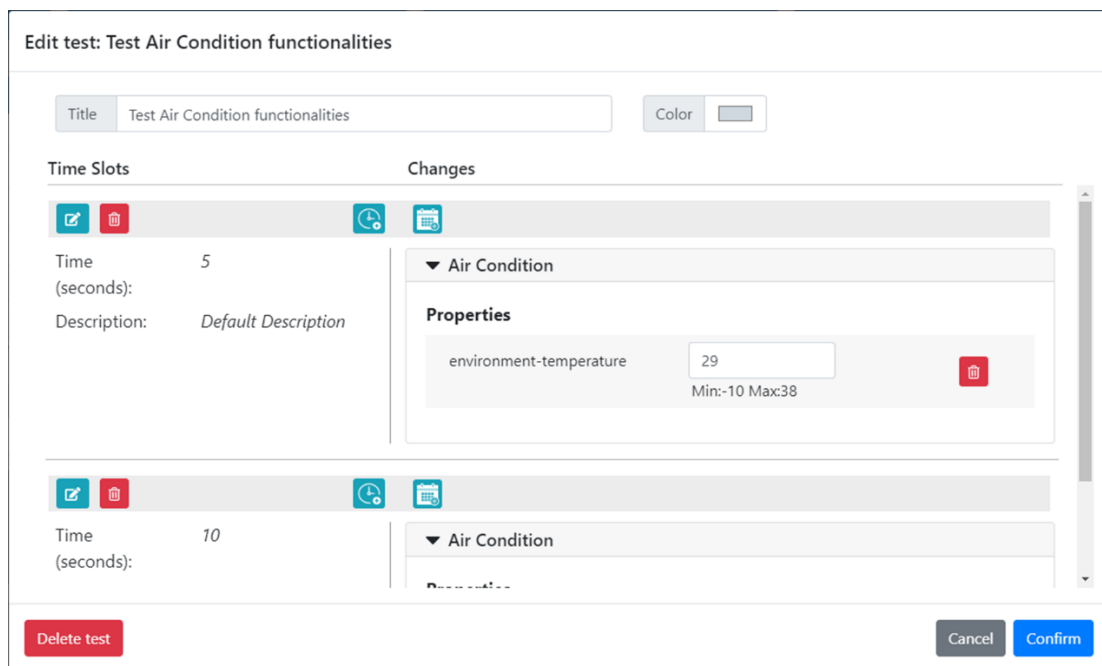


Figure 9.37. Simulating behavior of smart devices at specific time periods.

9.7.3 Testing Automations

We have developed an extra category of tests that the end-user developers are able to use in order to help their debugging and testing process for their applications. In

particular, they defined two new *Blockly* blocks (see grey box of Figure 9.38). Using them, end-user developers are able to build tests which run either as simple instructions, or included in conditional blocks. The authoring of these tests, helps the end-user developers to detect unexpected values of smart device properties either during the whole debugging process or in specific circumstances by using conditional blocks.

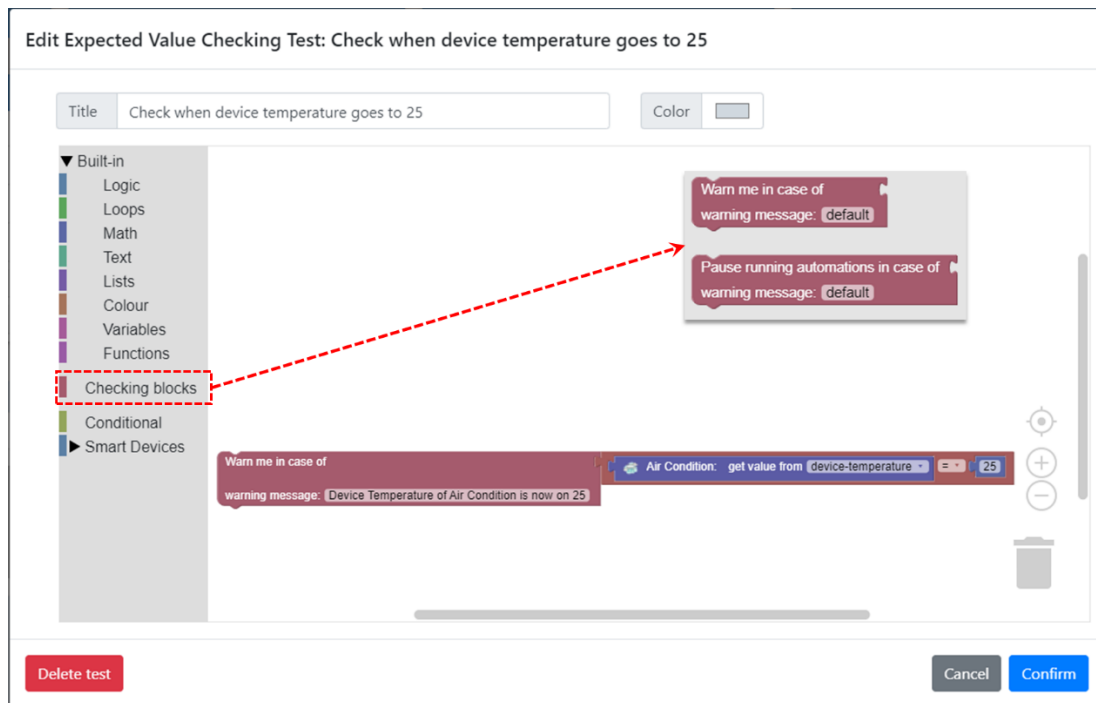


Figure 9.38. End-user development of tests for expected values in smart devices properties.

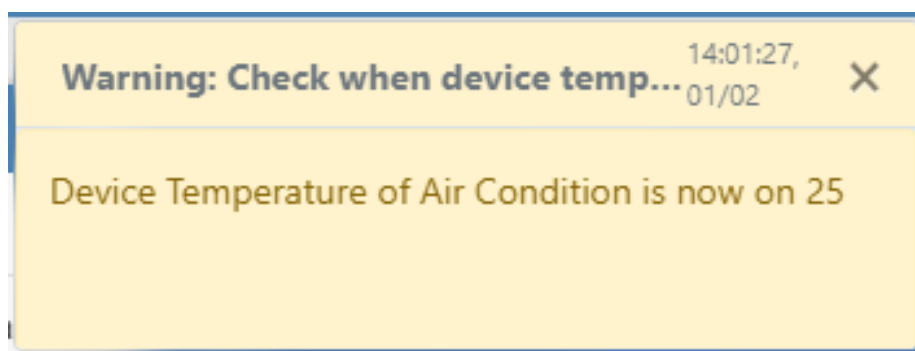


Figure 9.39. Warning message in case a test of expected values of smart device properties fails.

When a checking block is triggered by an unexpected variable value, a dialogue (see Figure 9.39) opens in order to warn the end-user developer and the project execution pauses until the user chooses to continue using the visual debugger toolbar. This is helpful in order to enable the end-user developers to check the values of their

programs in general by using the watches and variables which are provided by the visual debugger's toolbar.

9.8 Case Study

When we developed the integrated domain framework, we decided to carry out a case study to better present the end-user development process, validate and test our visual programming approach for personalized automations in the context of the IoT.

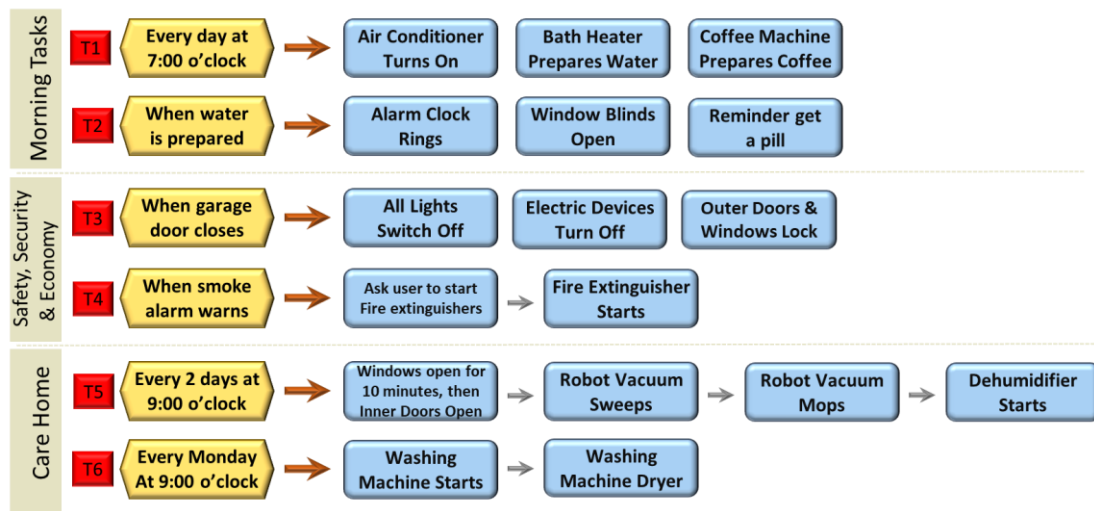


Figure 9.40. Morning home automations example.

9.8.1 Discussing of Use Case for Morning Automations

In this context, we describe a hypothetical scenario of automations in the morning as it is one the most difficult time of the day for people is when they're waking up. People would like to earn more sleep in the morning while their water for their bath and their coffee will be prepared. Additionally, they may like to be reminded to get a pill, while they would like to have peace of their mind when they leave their home to go for work by ensuring home safety, security and economy. Moreover, people would like to clean their home, however, their free time is limited and they would like to automate this task by using smart devices. All the aforementioned automation tasks are able to be served based on their daily life and needs. An example of these tasks is represented in the Figure 9.40Figure 9.2.

9.8.2 Initiating of the End-User Development Process

Starting the process, we chose the applications domain category of '*Personal IoT Automations*' between the application domain frameworks and create new project (see

on the left of Figure 9.41). The constructed project *'Morning Automations'* is empty of project elements (see on the top-right of Figure 9.41).

The first end-user development steps is to define and choose which of the smart devices will be used for the automation. By using the right click of the project manager's element category *'Smart Devices'* there are two option of creating smart device or smart environment. When the end-user chooses to define smart device, the project manager opens a dialogue to define the required data for the smart device (i.e. name, image and color). Afterwards, the end-user is enabled to choose which of the devices will be registered either picking one from the already registered or by scanning to find new ones. The end-user is able to define or undefine smart devices during the process. The smart devices that have been used are displayed on Table 7 in order to represent their properties and actions.

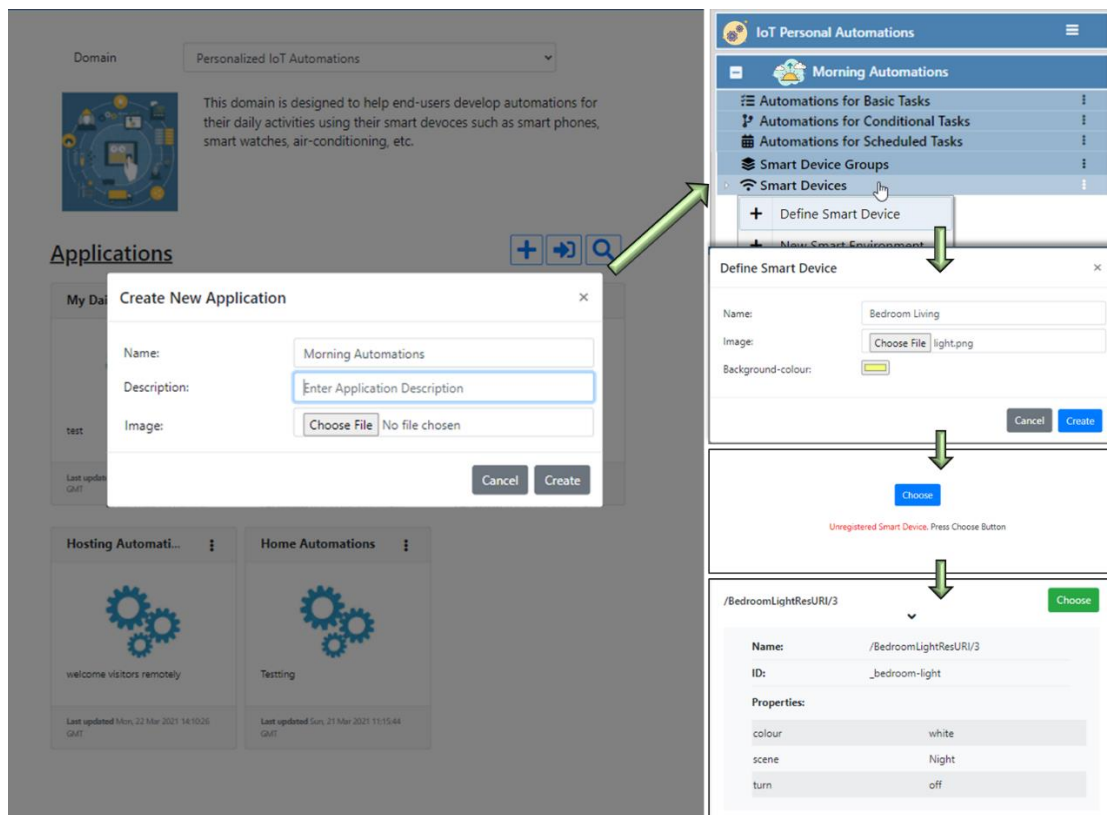


Figure 9.41. Creating morning automations and defining bedroom lighting device.

Table 7. Smart Devices that are used for Morning IoT Automations.

Smart Devices	Properties	Actions
Air Condition	<i>device-temperature: number</i>	<i>AutoMode ()</i>
	<i>environment-temperature: number</i>	<i>Configure (device-temperature, swing)</i>
	<i>swing: enum</i>	<i>TurnOff ()</i>
	<i>turn: on/off</i>	<i>TurnOn ()</i>
Alarm Clock	<i>ring: boolean</i>	<i>SetAlarmTime (time: Time)</i>
	<i>ringtone: enum</i>	<i>TurnOff ()</i>
	<i>time: Time</i>	<i>TurnOn ()</i>
	<i>turn: on/off</i>	
Bath Heater	<i>turn: on/off</i>	<i>TurnOn ()</i>
	<i>water-ready: boolean</i>	<i>TurnOff ()</i>
Bedroom Lighting	<i>colour: string</i>	<i>ChangeColour (colour: string)</i>
	<i>scene: enum</i>	<i>ChangeScene (scene: enum)</i>
	<i>turn: on/off</i>	<i>TurnOff ()</i>
		<i>TurnOn ()</i>
Coffee Machine	<i>coffee-ready: boolean</i>	<i>PrepareCoffee ()</i>
	<i>cups: number</i>	<i>StopPreparingCoffee ()</i>
	<i>prepare-coffee: start/stop</i>	<i>TurnOn ()</i>
	<i>turn: on/off</i>	<i>TurnOff ()</i>
Dehumidifier	<i>humidity: number</i>	<i>Service (service: enum)</i>
	<i>mode: enum</i>	<i>SilentMode ()</i>
	<i>service: enum</i>	<i>TurboMode ()</i>
	<i>turn: on/off</i>	<i>TurnOn ()</i>
		<i>TurnOff ()</i>
Main Door	<i>lock: enum</i>	<i>Close ()</i>
	<i>lock-status: boolean</i>	<i>Lock ()</i>

	<i>state: open/close</i>	<i>Open ()</i>
		<i>Unlock ()</i>
Fire Extinguisher	<i>measurement: number</i>	<i>Start ()</i>
	<i>state: enum</i>	<i>Stop ()</i>
Garage Door	<i>lock: enum</i>	<i>Close ()</i>
	<i>lock-status: boolean</i>	<i>Lock ()</i>
	<i>state: open/close</i>	<i>Open ()</i>
		<i>Unlock ()</i>
Living Room Lighting	<i>colour: string</i>	<i>ChangeColour (colour: string)</i>
	<i>scene: enum</i>	<i>ChangeScene (scene: enum)</i>
	<i>turn: on/off</i>	<i>TurnOff ()</i>
		<i>TurnOn ()</i>
Robot Vacuum	<i>clean-program: enum</i>	<i>Mopping ()</i>
	<i>state: enum</i>	<i>Program (clean: enum)</i>
	<i>Turn: on/off</i>	<i>Sweep ()</i>
		<i>TurnOff ()</i>
		<i>TurnOn ()</i>
Smoke Sensor	<i>measurement: number</i>	
	<i>value: enum</i>	
TV	<i>channel: string</i>	<i>ChangeChannel (channel: string)</i>
	<i>turn: on/off</i>	<i>TurnOff ()</i>
	<i>volume: number</i>	<i>TurnOn ()</i>
		<i>Volume (value: number)</i>
Washing Machine	<i>child-lock: boolean</i>	<i>Program (washing-program: number)</i>
	<i>speed: number</i>	<i>Start ()</i>
	<i>state: enum</i>	<i>Stop ()</i>
	<i>temperature: number</i>	<i>Temperature (temperature: number)</i>

	<i>time-period: number</i>	<i>TurnOn ()</i>
	<i>washing-program</i>	<i>TurnOff ()</i>
Window	<i>lock: enum</i>	<i>Close ()</i>
	<i>lock-state: boolean</i>	<i>Lock ()</i>
	<i>state: enum</i>	<i>Open ()</i>
		<i>Unlock</i>
Window Blinders	<i>state: open/close</i>	<i>Open ()</i>
		<i>Close ()</i>

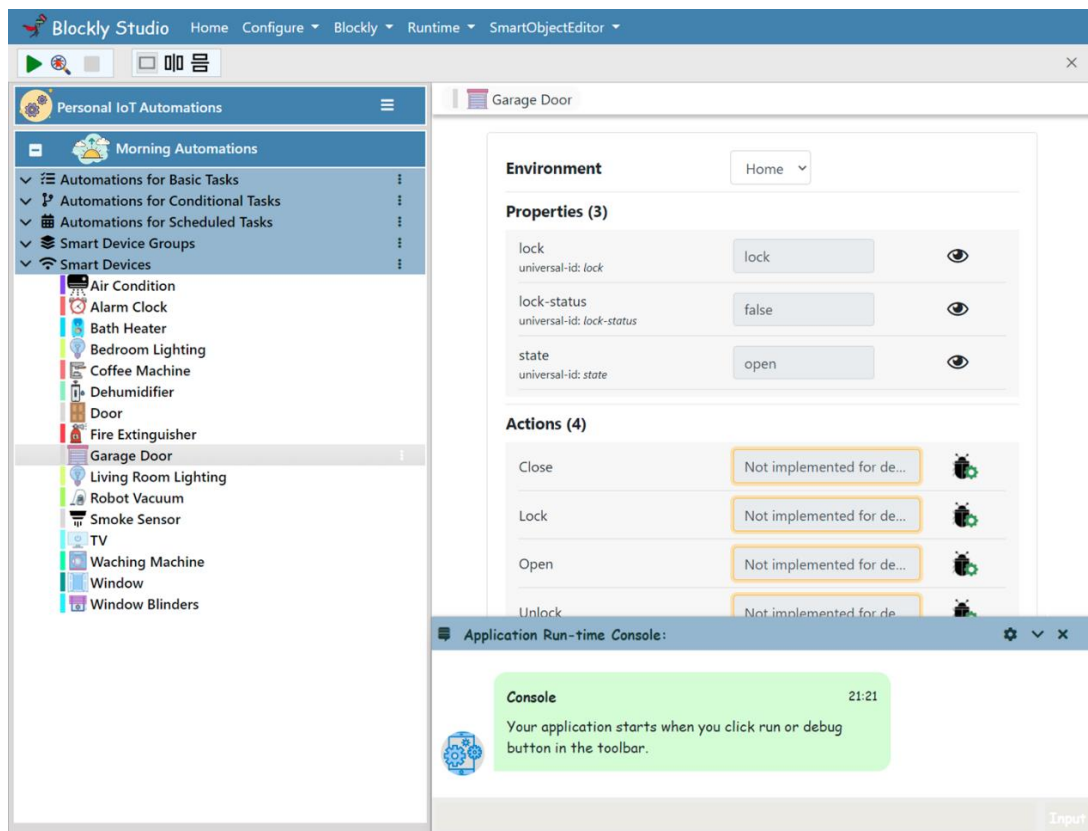


Figure 9.42. Workspace view having define the smart devices for morning automations.

Defining the above smart devices in the 'Morning Automations', the end-user developers are able to browse them and handled by using the project manager as it is depicted on Figure 9.42.

9.8.3 Visual Programming of Scheduled and Conditional Tasks

Having complete the definition of the smart devices for the morning automations, the next step is to define for each of the tasks (*T1-T6* in Figure 9.40) one project element either scheduled or conditional as it is shown in Figure 9.43.

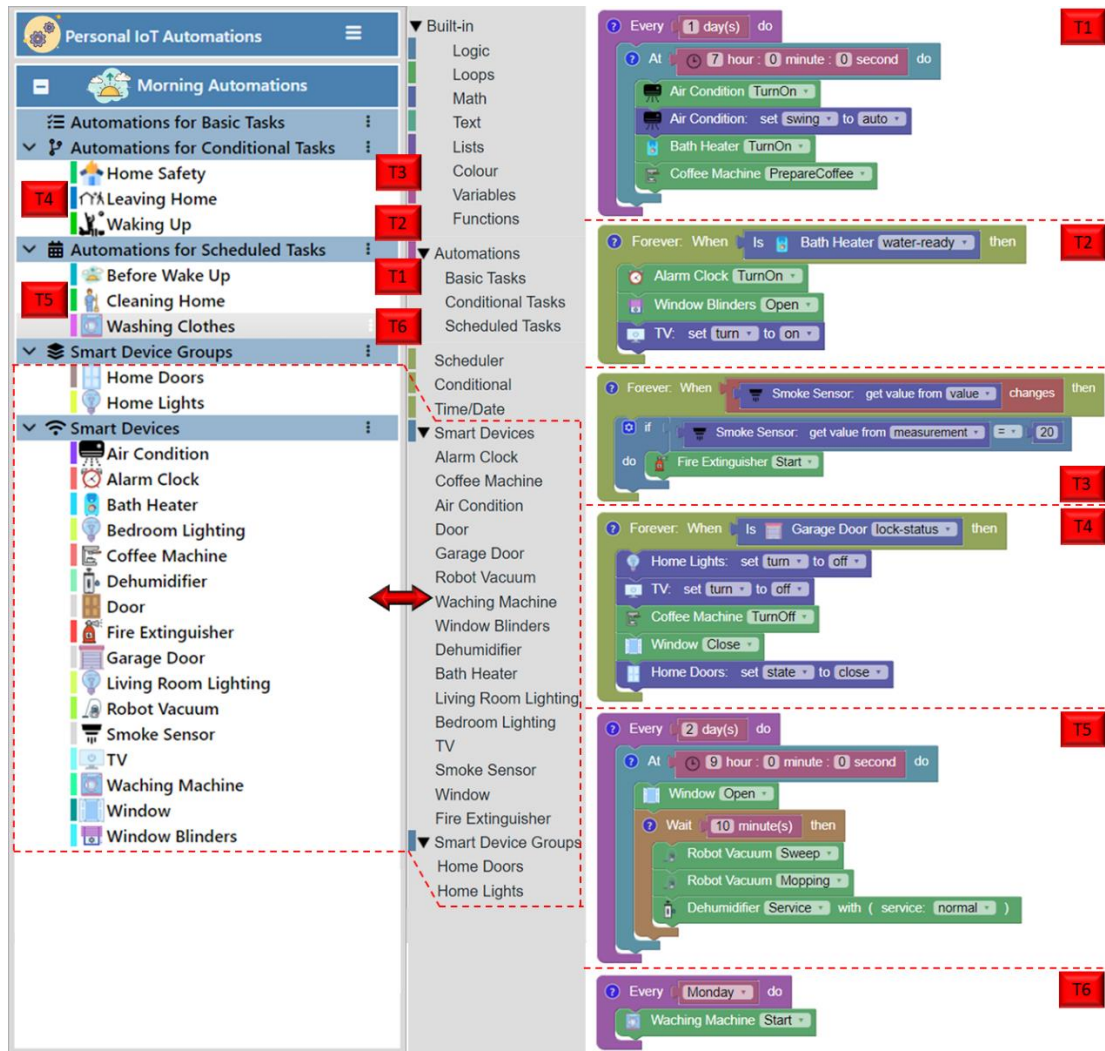


Figure 9.43. Visual programming scheduled and conditional tasks for morning automations.

Defining the smart devices in the project, respective Blockly blocks have been defined in order to handle their behavior. These Blockly blocks are available in each of the tasks (i.e., ‘*Automations for Basic Tasks*’, ‘*Automations for Conditional Tasks*’ and ‘*Automations for Scheduled Tasks*’) as it is shown in Figure 9.43. Using these Blockly blocks, visual code has been developed for each of the defined project elements as it is displayed in *T1-T6* tags of Figure 9.43.

9.8.4 Running Morning Automations

Having completed the end-user development of the scheduled and conditional tasks as it is presented in previous section, we have run the project of *'Morning Automations'* and in this section, we display the parts of the runtime view. The main runtime view part is the visualization of the smart devices during the execution as they are shown in Figure 9.44.

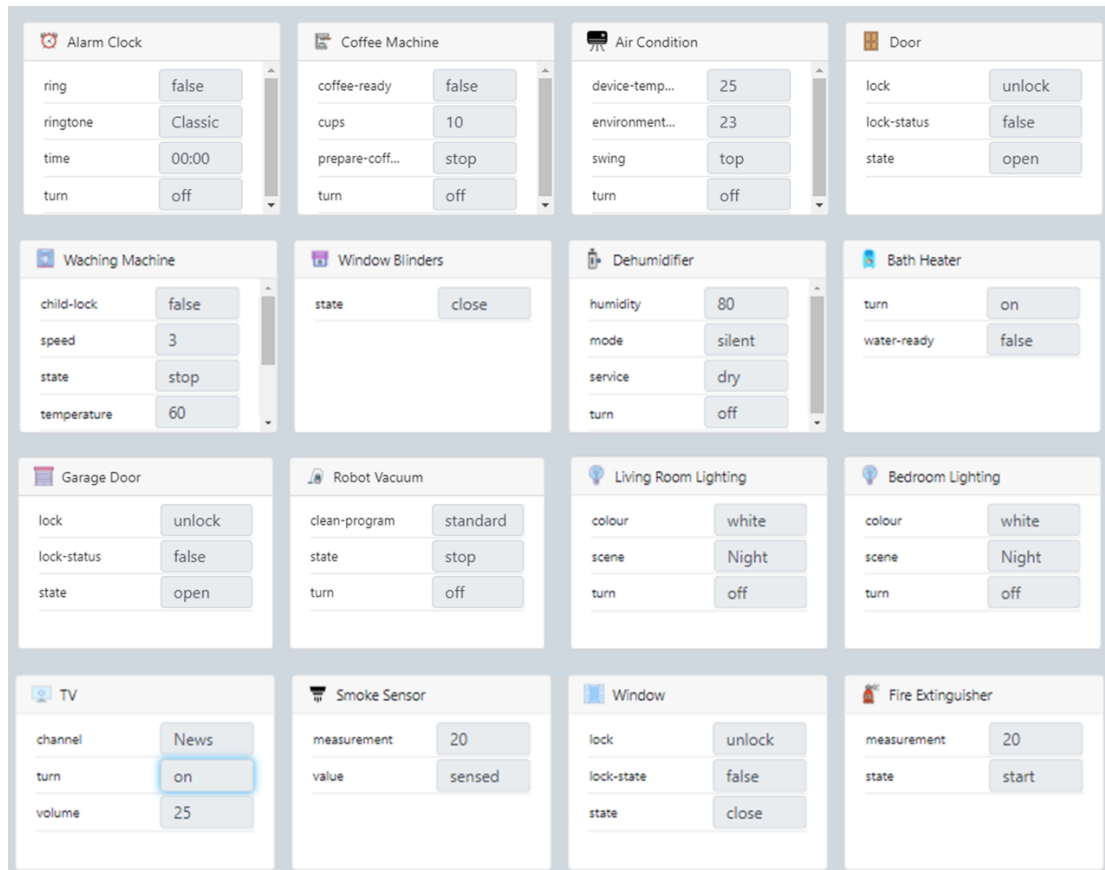


Figure 9.44. Smart Devices monitoring values for 'Morning Automations' project.

Moreover, each of the scheduled tasks have been added in the calendar view as it is shown in Figure 9.45. When a scheduled task is completed the calendar, view underlines it and checks it as finished. In addition, when a scheduled task starts, the calendar view updates the events by adding new event in calendar.

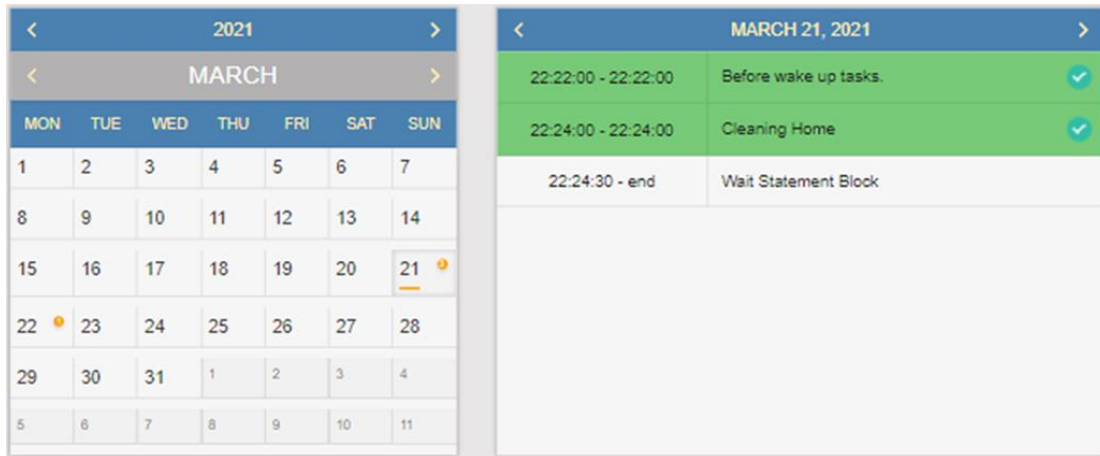


Figure 9.45. Calendar view of the scheduled tasks for 'Morning Automations'.



Figure 9.46. History actions view of the tasks that will be shown running 'Morning Automations'.

Additionally, as earlier mentioned, for each of the tasks that are caused during the runtime execution, a history actions logger with bubbles is displayed. In this context,

running the project of *'Morning Automations'* actions bubbles are depicted in history logger for T1, T2, T4 and T5 tasks as it is shown in Figure 9.46. However, T3 task will only happen in case of fire and T6 tasks will happen one week later. In this context, we have to ensure that visual code by using the simulator and the debugger that are provided. In the next section, we present use of these tools in the context of the *'Morning Automations'* project.

9.8.5 Morning Automations Testing

Starting the debugging process, we have to simulate the behavior of the actions of each of the smart devices as simulator replaces the real devices and enables the end-user developers to implement by using Blockly instances as earlier discussed. Smart device actions of simulated devices have been developed for each of the actions that are used in the *'Morning Automations'*. An example of the implemented smart device actions is the coffee machine as it is presented on Figure 9.48.

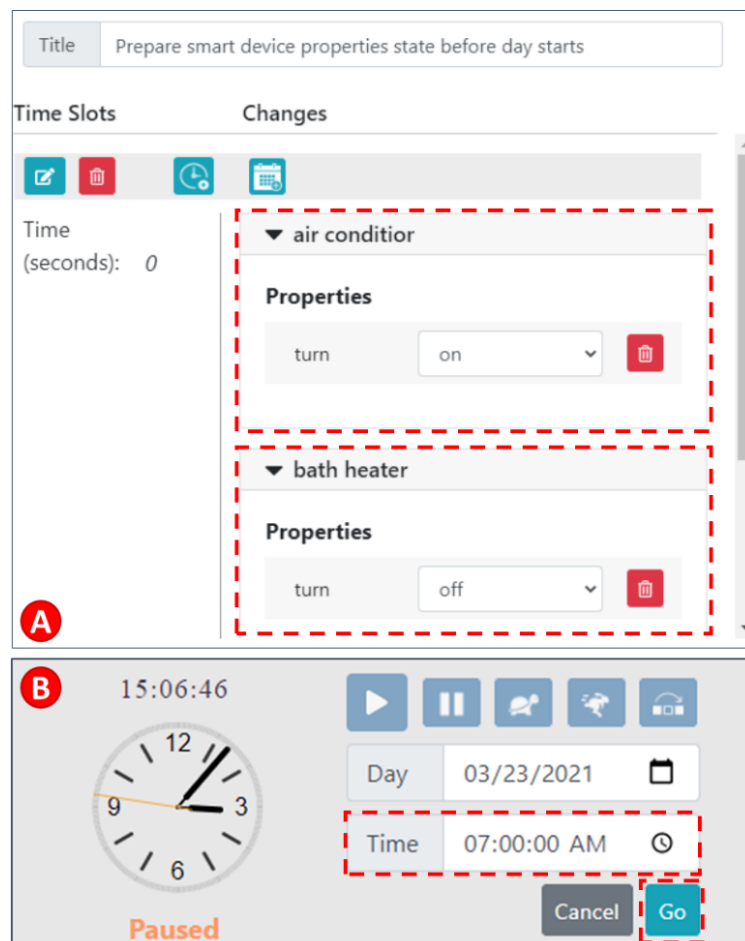


Figure 9. 47. Preparing state of smart device properties (tag A) and go at specific time in order to trigger scheduled task of *'Morning Automations'*.

The second step of using the simulator in order to test the 'Morning Automations' is to build the behavior of the smart device property at specific time. In particular, smart device properties are simulated by the user in order to design their expected state and trigger the conditional tasks. In case of 'Morning Automations', 'air condition', 'coffee machine' and 'bath heater' has to be simulated as turned off (see tag A of Figure 9. 47). Afterwards, the end-user developer has to set specific time to go in order to start the scheduled automation (T1 of Figure 9.43) as it is presented in tag B of Figure 9. 47.

The image displays three sections of a smart home automation simulator, each showing a list of actions on the left and their corresponding code blocks on the right. Red arrows indicate the mapping between the actions and the code.

- Vacuum Robot:**
 - Mopping:** Implemented for debug. Code: Vacuum Robot: set turn to on; Vacuum Robot: set state to mop; Wait 30 minute(s); Vacuum Robot: set turn to off.
 - Sweep:** Implemented for debug. Code: Vacuum Robot: set turn to on; Vacuum Robot: set state to sweep; Wait 25 minute(s); Vacuum Robot: set turn to off.
 - TurnOn:** Not implemented for de... (highlighted in yellow).
 - TurnOff:** Not implemented for de... (highlighted in yellow).
 - Program:** Not implemented for de... (highlighted in yellow).
- Coffee Machine:**
 - PrepareCoffee:** Implemented for debug. Code: Coffee Machine: set turn to on; Wait 1 minute(s); Coffee Machine: set turn to off.
 - StopPreparingCoffee:** Implemented for debug. Code: Wait 5 second(s); Coffee Machine: set prepare-coffee to stop.
 - TurnOn:** Not implemented for de... (highlighted in yellow).
 - TurnOff:** Not implemented for de... (highlighted in yellow).
- bedroom lighting:**
 - ChangeColour:** Implemented for debug. Code: bedroom lighting: set colour to colour_arg.
 - ChangeScene:** Implemented for debug. Code: bedroom lighting: set scene to scene_arg.
 - TurnOn:** Not implemented for de... (highlighted in yellow).
 - TurnOff:** Not implemented for de... (highlighted in yellow).

Figure 9.48. Implemented actions for smart devices of 'Morning Automations'.

Afterwards, we have to test the visual code for the conditional task of the home safety in case the smoke sensor will change its value. The first step is to browse the 'Home Safety' project element and add a breakpoint in *IF* block as it is shown in tag A of Figure 9.49. Then, start debug process, create simulate tests of the smart devices behavior and simulate the smoke sensor is activated with measurement 20 as it is presented in tag B of Figure 9.49. This simulation will trigger the conditional task of 'Home Safety' and the execution will stop in the breakpoint. Afterwards, by using step-in action of the debugger we are able to trace the visual code execution flow and view values of the smart device properties in debugger's data (see tag C of Figure 9.49). Finally, we are able to view the history actions in order to verify that conditional task activated and the fire extinguisher started.

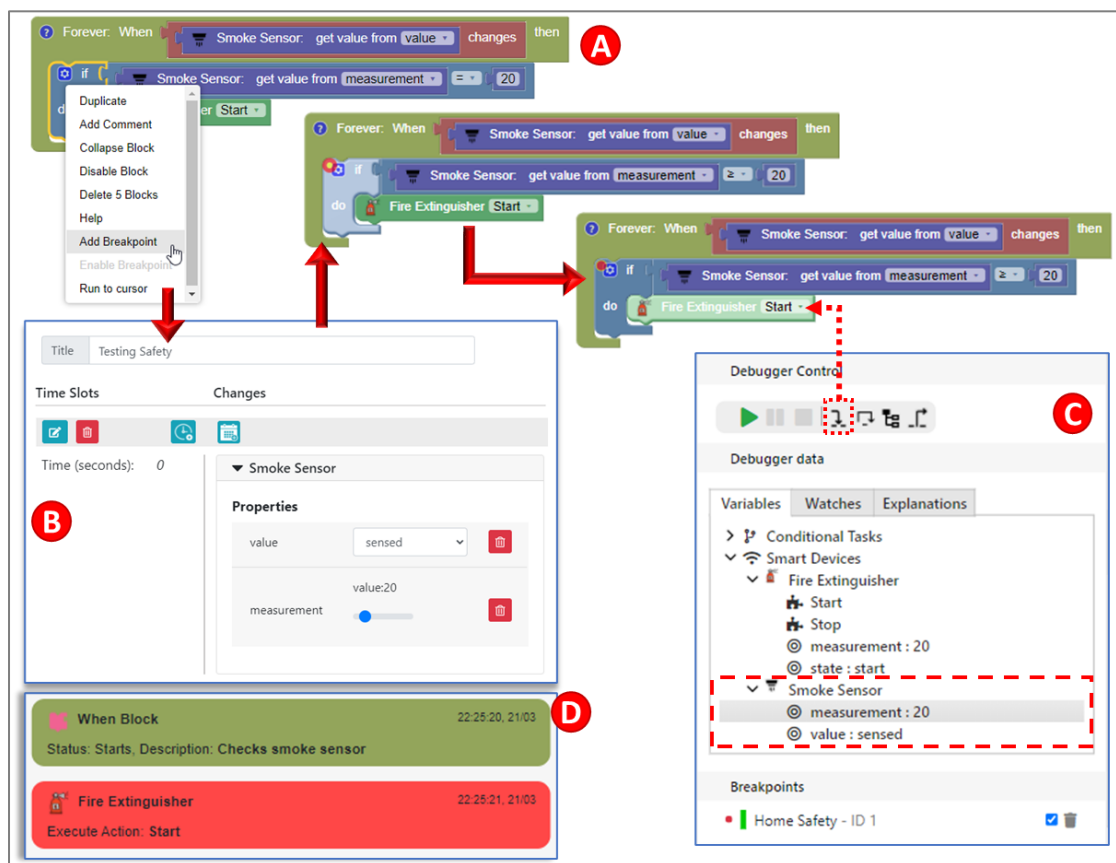


Figure 9.49. Testing 'Home Safety' conditional task of 'Morning Automations': Adding breakpoint (tag A); Simulating behavior of the smoke sensor (tag B); Stepping in until the simulated fire extinguisher starts and view variables and smart device properties state (tag C); View actions history to verify the fire extinguisher started (tag D);

9.9 Evaluation

Having finished the case study of the integrated domain framework, we decided to evaluate our proposed workspace in the context of the smart automation development process. In this section, we discuss the aims and design of our study, present the use case scenarios, outline the evaluation's participants, describe the evaluation process and analyze the results.

9.9.1 Aims and design

The evaluation we conducted aims on observing how users operate and use our system's key features as well as on assessing the system's usability. Particularly, we dedicated our study's focus to evaluating the use of the project manager, the handling of smart objects through the domain VPL editor and the development of automations using the blocks we developed. For each one of them that we considered important, we designed a use case scenario that focused on deciding whether the chosen approach was indeed appropriate and well-integrated. For obtaining usability measurements, we used the System Usability Scale (SUS).

9.9.2 Use case scenario

We use hypothetical user to discuss the use case scenario. In the use case scenario, we introduce the hypothetical user Tina who bought new smart devices and wants to develop smart automations. We have segmented the use case scenario in development mini tasks. Each of the following tasks are separated in two parts, the description and the goal. The description of task refers to the real-world situation that contextualizes the goal. The goal of each development step refers to the task that should be accomplished. The tasks' contexts are realistic and the goals are kept simple and short in order to evaluate the usability of specific features of our approach. The tasks of the use case scenario are following.

1) Creating new smart environment and registering smart objects

Description: Tina has bought new smart alarm clock, smart coffee machine and smart air-condition. She wants to create new environment, create new smart objects and then, register them in order to develop smart automations.

Goal: The participants were asked to create new smart environment, create new smart objects and register the smart objects.

2) *Creating smart group for smart objects*

Description: Tina has two more smart air-conditions in her home and wants to handle them together in a new group. However, two other devices API differs in the property of ‘device-temperature’ which are provided as ‘thermometer’.

Goal: The participants were asked to export smart group from the air-condition and handle aliases in order to include in the group all air-conditions.

3) *Developing conditional events*

Description: Tina would like to create a smart automation in order when the alarm clock rings to automatically prepare coffee, prepare warm water for her bath and regulate the home temperature.

Goal: The participants were asked to create new project element in the category of conditional events and using the available blocks to develop the automation.

4) *Developing calendar events*

Description: Tina leaves her home to go at work at 8:00 o’clock daily except the week-ends. She would like to create a smart automation in order to turn off forgotten devices and lock the door when she has left.

Goal: The participants were asked to create new project element in the category of calendar events and using the available blocks to develop the automation.

5) *Developing combined (conditional and time) events*

Description: Tina would like to sleep some more minutes when alarm clock rings while the coffee and the water will be prepared. In order to do this, she has to edit the previous developed automation.

Goal: The participants were asked to edit the automation and add instruction to stop the alarm clock rings, wait for 8 minutes (i.e. water and coffee will be ready) and ring again the alarm clock.

9.9.3 Participants

We asked 15 participants (M = 10, F = 5) aged between 13 and 32 to help us. Most of the participants were from our university departments (i.e. Computer Science, Mathematics and Physics). Additionally, 6 of the participants were high school students that have previous experience with *Scratch*. Moreover, we found 3 individuals that had no previous experience with programming or visual programming.

9.9.4 Process

Each participant was evaluated individually. We firstly discussed and presented the classic *Blockly* editor. Then, we presented our visual programming workspace for *Blockly* including the project manager, the smart object visual programming editor and the new *Blockly* blocks that are generated based on the smart objects. Next, each of the aforementioned tasks of the use case scenario was described to the users and they were asked to use the workspace in order to accomplish each of them. For each task and participant, we measured the time required for completion and we recorded the user behavior. Finally, the users were asked to fill-in the SUS questionnaire.

9.9.5 Results

We summarized and further analyzed all the answers given from our participants. The SUS questionnaire was designed in order to export results in two main dimensions. The first was focused on the integration and usability of the workspace, the second was focused on the efficiency of handling smart objects and groups through the Smart Object visual programming editor and the third dimension was the use of the *Blockly* blocks for smart automations. Results showed that the vast majority of participants were satisfied with the workspace environment for smart automations. In general, they are satisfied with the use of the Smart Object visual programming editor. However, some users found difficult the concept of the smart groups. In this context, we realize that extra helpful functionality and user interface has to be added. In particular, when the user browses a Smart Group in order to choose from list of possible smart objects and the view of what are the properties which don't match were missing. Based on this feedback, we fixed this design mismatches. Moreover, the users were satisfied with the defined *Blockly* blocks for the development of smart automations.

Table 8. SUS Questionnaire for the Smart Automations Workspace Environment.

	SD	D	N	A	SA
Q1. The smart automations framework is well integrated into the workspace.	0	1	2	6	6
Q2. I find the smart automations workspace environment unnecessarily complex.	6	8	1	0	0
Q3. I find the smart object editor user interface intuitive and easy to use.	0	0	2	8	5
Q4. I don't feel confident using the application without guidance.	5	9	0	1	0
Q5. I feel confident using the project manager.	0	1	3	5	6
Q6. The <i>Blockly</i> blocks for smart automations offer limited options for development.	7	6	2	0	0
Q7. I find <i>Blockly</i> blocks for smart automations complex to use them.	5	7	2	1	0
Q8. I would like to use the tool for my personal projects with my family/friends.	0	1	2	7	5
Q9. I found easy to use the smart object editor for smart groups	0	2	6	5	2
Q10. I found difficult to use the smart object VPL editor to handle the smart objects	8	6	1	0	0

Furthermore, based on the aforementioned measurements we focused on the average, the best and the worst time recorded for each development step. All the users completed the tasks and most of the worst time measurements are not far from the average, while the best are not far from the average too. Moreover, during the evaluation, we realized that after the 3rd task, most of the users were more familiar with the workspace.

Chapter 10

Conclusions and Future Work

“Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.”

-Roger Bacon

10.1 Summary

In this thesis we discuss the full-scale development of an extensible IDE for visual programming languages, including project manager, visual debugger, collaborative programming and pluggable domain frameworks. Our primary motivation was the need for a full-scale visual programming environment for end-user development of personalized IoT automations in order to empower non-programmers craft, modify or extend their automations. Existing visual programming approaches are facilitating by introducing sophisticated editors; however, no IDEs are provided. In particular, the existing approaches are mainly targeted to children learning within the context of a game. Regarding several visual programming features, they are at an infant level or not mature enough (e.g., collaboration, debugging, etc.), as well. We consider that non-programmers and learners behoove to be provided with more efficient end-user programming tools in their arsenal for developing and learning programming purposes.

Moreover, the visual programming frameworks are limited to specialized requirements of one application domain (e.g., Scratch is focused on development of games and animations). However, visual programming radically differs across domains (games, learning, IoT, etc.). Considering that new application domains are arising, existing application domain requirements for visual programming are fluid and third-party technologies are updated continually, constantly changing requirements for developing new visual programming IDEs. In this context, we embrace visual programming domain variations as domain frameworks in our IDE. More specifically, we allow installing domains by introducing custom visual

programming language elements across IDE components (e.g. editor, project manager, runtime environment). Namely, our approach provides application domain plug-in mechanism in the IDE in order to support them.

We consider that collaboration is a key feature in EUD and could be notably useful in the case of using it for teaching and learning purposes, asking for help from more experienced users and co-working for automations. This thesis is focused on full-scale collaborative visual programming facilities. These facilities are separated in two categories, the collaborative editing and the collaborative debugging. Regarding collaborative editing, we developed a full-scale collaborative editing approach that sorts out the process by introducing peer roles and project element privileges. Our proposed approach supports multiple collaboration models (i.e. Pair Programming in one or more groups, teaching and learning purposes, working in small teams) by regulating the settings are provided in order to configure collaboration process when it starts.

In the case of collaborative debugging, our approach addresses two different directions. First direction deals with facilitation of debugging and testing for novices by providing collaborative debugging process that can be used for personal and collaborative EUD projects. Collaboration proposed approach guarantees the preservation of the project's visual code by isolating it, creating a local replica for each one of collaboration members. In this context, the users are able to create correction suggestions per project element. During the debugging session, handling the debugger instructions can be done by one user at a time. However, the rest members are able to navigate the visual code to acquire information independently of other members browsing, without interfering with the experience of any collaboration member. The second direction of our approach includes an alternative model of collaborative debugging that contributes in teaching and learning in the context of debugging and programming. Particularly, this tool can be used by teachers to demonstrate debugging process to students in real-time. Students are able to perceive the flow of a program and learn the process of debugging through multiple debugging rooms in a session that encourage the students to live debug programs, individually or collaboratively while allowing the teachers to supervise each debugging process.

Finally, we discuss the development of a full-scale application domain framework in the context of IoT automations using the IDE. We provide a full-scale management for the smart devices in the context of EUD including user actions to organize and customize smart devices in order to enable isolation and handling numerous existing smart devices. Additionally, we provide a full-scale VPL workspace environment for personalized IoT automations including conditional and scheduled tasks and enabling them automatically or manually during the project execution. We provide GUI for runtime environment for monitoring and interacting with smart automations, facilitating the end-user developers by removing the requirement to program UIs for their automations. We also introduce facilities in the context of testing and debugging the smart automations by developing infrastructure to enable users to simulate smart devices, their behavior, date and time that the automations will be executed. Moreover, we address the issue of responding to the arising user questions regarding automations that caused during the execution of the constructed IoT applications.

10.2 Conclusions

Throughout the entire thesis we have emphasized to three primary arguments driving our research work: (i) novices deserve efficient full-scale end-user development tools in order to develop their applications and learn programming (ii) embracing visual programming domain variations as domain frameworks in IDE (iii) efficient visual programming facilities for end-user development of IoT personalized automations.

During the initial phases we focused on the efficient visual programming environment for personalized IoT automations. However, during the first steps of our work we perceived that existing visual programming environments are narrowed in specific targets by providing sophisticated editors, without providing full-scale toolset in end-user development concept. Moreover, several key end-user development features are in an infant level, or not mature enough, or even not provided. Therefore, we focused on the development of an extendable full-scale IDE for visual programming languages including key features such as project manager, visual debugger and collaborative programming.

Developing the main components of the IDE, we quickly observed that components have to provide additional support and adapt in the context of novices or non-programmers.

Regarding project manager, it has to be restricted and drive end-user developers to structure their applications by providing them specific options of their actions, friendlier user-interface for the project elements view, helpful information messages during end-user development process. In addition, in the direction where novice users handle small scale project elements, the first-class subject focuses on organizing and structuring project in small scale project elements.

Visual programming editors are vehicle for end-user developers to program their visual code. In this context, we focused on facilitating editors' usability by providing filtered VPL elements in their toolboxes based on concept that they have to be accomplished in specific workspace. In addition to that, we consider that it is important to empower users with respective intelligence. For this reason, we introduced visual code snippets through which end-user developers will be able to use them instead of repeating building common small blocks of visual code.

Regarding debugging facilities for visual programming, a complete level debugger has been developed for Blockly. Using tracing, watches and breakpoints is really helpful for a novice user to understand the execution flow of a program, however, finding bugs is not always a trivial process. In the context of the aforementioned consideration, we focused on giving more weapons to end-users for the debugging process. In particular, we introduced execution snapshots to provide browsing of history of values for program variables, selection of project elements that will participate from the project execution and collaborative debugging.

Concerning collaborative programming, we consider from the early phase of this thesis that is a key feature in end-user development context and we targeted to provide a full-scale toolset approach. Our approach was focused on two directions, the collaborative editing and the collaborative debugging. In the context of collaborative editing, the end-users have to cooperate on a shared project. The first-class subject of our approach is targeted to organize and structure project in small project elements. Therefore, we have introduced peer roles and project element privileges for the

participants. Moreover, concerning the productivity for member in the collaborative editing process, we focused on the local workspace of members by introducing personal project elements, toggling live syncing and selective execution (i.e., replace shared project elements with personal or visual code suggestions). Finally, we introduce several settings to enable the master of the process and to configure it based on the circumstances.

Regarding collaborative debugging, we have introduced two different models. The first collaborative debugging model focuses on the collaborative debugging for a team to solve bugs for a shared or not project. In order to avoid the project element privileges and guarantee preservation of the project's visual code by isolating it, creating a read-only replica for each one of the collaborative members. We enable end-user developers to add corrections fixes for a specific project element and thanks to extension of selective project elements' execution, they are able to test their corrections. The second collaborative debugging model focuses on teaching and learning debugging and programming process. In particular, extending the first model, we have introduced debugging rooms in order to enable teachers define teams of the students (or alone). In the debugging rooms, independent debugging sessions run and correction suggestions are local in the room. Based on this, the teachers are able to supervise and help students independently by visiting the rooms and joining current state of the debugging session of the room.

Concerning support of different application domains, we designed and developed an extension mechanism that allows to embrace visual programming domain variations as domain frameworks in the IDE. In this context, we encourage developers to configure all the main components of the IDE based on the requirements of the domain. We enable the developers to define the application structure of the project manager, choose functionality, intervene the process of user's actions about project elements, define respective rules. In this direction, from the early phase of the development, we perceived that in the context of domain, there are cases in which more complicated project elements could be required instead of just displaying a visual programming editor. We have also introduced templates that empowers the developers to design and develop any project element ingredients. Moreover, the main part of project elements are the visual programming editors are injected. Developers

are able to configure the view of these editors and the VPL elements that will be included in their toolboxes.

Furthermore, we have identified two types of editors: a) general purpose editors which cares about basic programming expressions and b) domain-specific editors which contribute in the design or handling of domain objects. However, the behavior of domain objects is handled by a set of VPL elements that has to be provided automatically based on the domain objects that are developed by end-user developers. We have developed a mechanism which cooperates with domain-specific editors in order to bridge the general-purpose editor (i.e., Blockly) required updates of the toolboxes with the appropriate Blockly blocks to handle domain objects. In the context of running and debugging process, based on the different authored application structures and domain libraries developers have to define the entry point script that will bridge all required parts in order to execute domain projects.

Finally, we focus on the initial goal of this PhD thesis, which is the development of a visual programming workspace for IoT automations. Using the earlier mentioned mechanism for domain frameworks, we developed IoT automations framework. In this context, we provide a full-scale management for smart devices in the context of EUD including user actions to organize and customize smart devices in order to enable isolation and handling of numerous existing smart devices. Additionally, we provide a full-scale VPL workspace environment for personalized IoT automations including conditional and scheduled tasks and choice of starts them automatically or manually during the project execution.

In the context of runtime for IoT automations, users would like to build appropriate user-interface for their automations. However, building user-interfaces for automations costs extremely and it is impractical in the concept of creating micro automations. Additionally, *WYSIWYG* editors improve user-interface development process, but needs to have experience in the context of events. In this direction, we were driven to provide user interface that cares for monitoring and interacting with smart automations, facilitating end-user developers by removing the requirement to program UIs for their automations.

In the context of debugging IoT automations, there are several issues that arise by just using the visual debugger is provided. Firstly, developed automations are based on scheduled automations that may include tasks that will be executed much later. {How the end-user developers will be able to test such automations.} Additionally, smart devices and sensors include values that change based on the environment (e.g., environment temperature, smoke sensor). End-user developers are not able to debug and test their IoT automations. In this context, we introduced facilities for testing and debugging smart automations by developing infrastructure to enable the users to simulate smart devices, the behavior of smart devices, date and time that the automations will be executed. Moreover, we address the issue of responding to the arising user questions about automations which caused during the execution of the constructed IoT applications.

Overall, this thesis focused on providing efficient visual end-user programming toolsets through an IDE for visual programming languages. We consider that non-programmers and learners behoove to be provided with more efficient end-user programming tools in their arsenal for developing and learning purposes. Additionally, we emphasize on the collaborative programming as we consider it as a key feature for novices to cooperate and learn programming. Furthermore, supporting extendibility in the context of application domains, guarantees that our work will be able to be applied for new challenges and requirements of visual programming purposes. Finally, contributing in visual programming development for IoT automations, we empower the novices to create, modify, debug, test and use their personal automations in order to benefit their daily life and activities taking advantage of the smart devices.

10.3 Future Work

In this thesis, we focused on the most prominent of the identified requirements, while some of the areas remain open and require additional research work. Below, we briefly discuss key topics for future work.

One of the identified issues is the research on facilitating the debugging process. Non-programmers and novices are not able to use efficiently the visual debugger that is provided in order to detect bugs in their projects. In this context, we introduced the

collaborative debugging. However, another dimension to facilitate end-user developers without cooperate with other users could be the development of a debugger assistant that will drive users on the debugging process.

Another identified issue is the general-purpose visual programming editors where more work is needed to be done in the context of intelligence in order to facilitate their use. For example, auto-complete visual code suggestions, suggestions of editing visual code that is repeatable, warnings in cases of missing body of visual programming language elements, etc.

Moreover, based on the audience could be familiar with different visual programming languages (jigsaws, diagrams, etc.). It would be interesting to explore of defining a top visual code model that through this, visual programming editors will be able to load and save their visual code. Using this mechanism, they will be able to view and handle visual code in different visual programming editors based on their preferences. However, general-purpose visual programming editors may differ in the context of supported elements (variables, branches, loops, etc.) with another concept of messages and objects. This means that the conversion is not a straight forward process and has to be identified if it is feasible. Moreover, the development of alternative approaches of visual programming languages seems to be a good idea.

Another interesting approach in the context of collaborative programming for smart automations could be the development of smart devices that are used by different users in different connected networks. Collaborative execution of smart automations could allow smart devices to interact each other by identifying who is their owner and each of the smart devices instructions of the shared project will be executed in the specific peer user side respectively.

Another perspective of a future work could be the development of other application domains using the Blockly Studio IDE including games for learning. In the context of a new application domain, an interesting application domain could be the development of application domain that will be able to modify or create new application domains from end-user developers. Based on the circular architecture of the IDE in which the components export their functionality, it will be able create appropriate visual programming language elements that will be used in order to

develop and modify functionality. In addition, extra ingredients will be required (e.g., build IDE components and their user-interfaces). In this context, this could be used only by experienced users. It looks to be more feasible empowering end-user developers to modify the existing application domain frameworks or configure the settings of the visual programming IDE through general-purpose visual programming editors (i.e., Blockly editor in our case).

Furthermore, having finished the development of the IDE for visual programming languages including framework for smart automations, our future work focuses on conducting a full-scale evaluation of our visual programming workspace in a high school class.

Bibliography

- [1]. Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothmel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. DOI=10.1145/1922649.1922658 <http://doi.acm.org/10.1145/1922649.1922658>.
- [2]. H. Lieberman, F. Paternò, and V. Wulf, Eds., *End-User Development, Human Computer Interaction Series*, Springer, New York, NY, USA, 2006.
- [3]. Alexandre Santos, Joaquim Macedo, António Costa, M. João Nicolau, *Internet of Things and Smart Objects for M-health Monitoring and Control*, *Procedia Technology*, Volume 16, 2014, Pages 1351-1360, ISSN 2212-0173 (2014).
- [4]. A Greg Little and Robert C. Miller. 2006. Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06)*. ACM, New York, NY, USA, 135-144. DOI=10.1145/1166253.1166275 <http://doi.acm.org/10.1145/1166253.1166275>.
- [5]. Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 943-946. DOI=10.1145/1240624.1240767 <http://doi.acm.org/10.1145/1240624.1240767>
- [6]. Scaffidi, C.; Bogart, C.; Burnett, M.; Cypher, A.; Myers, Brad; Shaw, M., "Predicting reuse of end-user web macro scripts," *Visual Languages and Human-Centric Computing*, 2009. VL/HCC 2009. IEEE Symposium on , vol., no., pp.93,100, 20-24 Sept. 2009 doi: 10.1109/VLHCC.2009.5295290.
- [7]. Open Office Scripting Framework. Open Office feature allows users to write and run macros for Apache OpenOffice. Official Website: https://wiki.openoffice.org/wiki/Documentation/DevGuide/Scripting/Scripting_Framework Accessed Online: 02/2021.
- [8]. Warth, A.; Yamamiya, T.; Ohshima, Y.; Wallace, S., "Toward A More Scalable End-User Scripting Language," *Creating, Connecting and Collaborating through Computing*, 2008. C5 2008. Sixth International Conference on , vol., no., pp.172,178, 14-16 Jan. 2008 doi: 10.1109/C5.2008.33.
- [9]. 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.
- [10]. Tynker web IDE. Educational programming platform aimed at teaching children how to make games and programs. First appeared 2012. Official website: <https://www.tynker.com/> Accessed online 2/2021.
- [11]. MakeCode: Hands on computing education. Producer: Microsoft. Released on: 08/2019. Official website: <https://www.microsoft.com/en-us/makecode> Accessed online 2/2021.

- [12]. 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.
- [13]. 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.
- [14]. 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.
- [15]. MIT App Inventor. Producer: Google, MIT Media Lab. Initial release on 12/2010. Official site: <http://appinventor.mit.edu/> Accessed online 02/2021.
- [16]. BlocklyDuino: The web-based, graphical programming editor based on Blockly. Official Website: <https://github.com/BlocklyDuino/BlocklyDuino> Accessed Online 02/2021
- [17]. ArduBlock: Visual Programming Environment for Arduino. Official Website: <http://blog.ardublock.com/> Accessed Online: 02/2021.
- [18]. Arduino: An open-source hardware and software company, project and user community that designs and manufacturers single-board microcontrollers and microcontroller kits for building digital devices. Official Website: <https://www.arduino.cc/> Accessed Online: 02/2021.
- [19]. Haller, S., Karnouskos, S., & Schroth, C. (2009). The Internet of Things in an enterprise context. In J. Domingue, F. Dieter, & T. Paolo (Eds.), Future internet – FIS 2008, lecture notes in computer science (Vol. 5468, pp. 14–28). Berlin: Springer.
- [20]. Xinyue Deng. Group Collaboration with App Inventor. Thesis: M. Eng., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2017.
- [21]. P. Rashidi and A. Mihailidis, "A Survey on Ambient-Assisted Living Tools for Older Adults," in IEEE Journal of Biomedical and Health Informatics, vol. 17, no. 3, pp. 579-590, May 2013. doi: 10.1109/JBHI.2012.2234129.
- [22]. Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. 2009. The Internet of Things in an Enterprise Context. In Future Internet --- FIS 2008, John Domingue, Dieter Fensel, and Paolo Traverso (Eds.). Lecture Notes In Computer Science, Vol. 5468. Springer-Verlag, Berlin, Heidelberg 14-28. DOI=http://dx.doi.org/10.1007/978-3-642-00985-3_2.
- [23]. A. Dohr, R. Modre-Opsrian, M. Drobics, D. Hayn, and G. Schreier. 2010. The Internet of Things for Ambient Assisted Living. In Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations (ITNG '10). IEEE Computer Society, Washington, DC, USA, 804-809. DOI=<http://dx.doi.org/10.1109/ITNG.2010.104>.
- [24]. Alexandre Santos, Joaquim Macedo, António Costa, M. João Nicolau, Internet of Things and Smart Objects for M-health Monitoring and Control, Procedia Technology, Volume 16, 2014, Pages 1351-1360, ISSN 2212-0173, <http://dx.doi.org/10.1016/j.protcy.2014.10.152>.
- [25]. Bee+, developed by Vigilant. Official site: <https://www.arm.com/innovation/products/bee-smart-diabetes-tracker.php> Accessed Online: 12/2016.

- [26]. Scratch: Block-based visual programming language and website targeted primarily at children 8-16 as an educational tool for coding. First appeared on 2003. Official Website: <https://scratch.mit.edu/>.
- [27]. Scratch Studio – Sharing is caring. Platform enables sharing the creations. Official Website: <https://scratch.mit.edu/studios/4164419/>.
- [28]. Phratch. VPL based on a jigsaw puzzle on top of Phraro. Official Website: <https://github.com/janniklaval/phratch> Accessed online 02/2021.
- [29]. Snap! Berkeley, extension of Scratch. Build Your Own Blocks. Open-source written by Jens Mönig and Brian Harvey, Berkeley. Official Website: <http://snap.berkeley.edu/> Accessed online 02/2021.
- [30]. Pasternak, E., Fenichel, R., Marshall, A. N. Tips for creating a block language with blockly. IEEE Blocks and Beyond Workshop (B&B), Raleigh, NC, USA, pp. 21-24 (2017).
- [31]. Dart, Web programming language developed by Google. Official Website: <https://www.dartlang.org/> Accessed online 02/2021.
- [32]. App Inventor 2: Create your own Android Apps. Second Edition 2014 Book by David Wolber, Hal Abelson, Ellen Spertus, Liz Looney. Official Website: <http://www.appinventor.org/book> O'Reilly ISBN-13: 978-1491906842.
- [33]. Lego Mindstorms. Official Website: <http://www.lego.com/en-us/mindstorms> Accessed online 02/2021.
- [34]. MODKit Micro. Official Website <http://www.modkit.com/> Accessed online 02/2021.
- [35]. Makeblock | mBlock: Extension of Scratch for Arduino and robotics. Official Website: <https://mblock.makeblock.com/en-us/> Accessed online: 02/2021.
- [36]. Danado, Marcin Davies, Paulo Ricca, and Anna Fensel. 2010. An authoring tool for user generated mobile services. In Proceedings of the Third future internet conference on Future internet (FIS'10), Arne J. Berre, Asunci & Gmez-Perez, Kurt Tutschku, and Dieter Fensel (Eds.). Springer-Verlag, Berlin, Heidelberg, 118-127.
- [37]. J. Danado and F. Paternò, “Puzzle: a visual-based environment for end user development in touch-based mobile phones,” in Human-Centered Software Engineering, vol. 7623 of Lecture Notes in Computer Science, pp. 199–216, 2012.
- [38]. TouchDevelop, Microsoft Research. Established 07/2011. Official Website: <https://www.microsoft.com/en-us/research/project/touchdevelop/> Accessed online 02/2021.
- [39]. Tillmann, N., Moskal, M., Halleux, J., Fahndrich, M., and Burckhardt, S. 2012. Touch-Develop: app development on mobile devices. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, , Article 39 , 2 pages.
- [40]. Microsoft MakeCode Arcade. Official Website: <https://arcade.makecode.com/#editor> Accessed online: 02/2021.
- [41]. Microsoft MakeCode editors: Computing education. Brings computer science to life for all students with fun projects, Official Website: <https://www.microsoft.com/en-us/makecode?rtc=1> Accessed online: 02/2021.
- [42]. Thyrd: An Experimental Reflective Visual Programming Language, Mercurio, Philip J. Talk presented at OSCON Emerging Languages Camp, Portland, Oregon, July 2010.

- [43]. Microsoft VPL. Official Website: <https://msdn.microsoft.com/en-us/library/bb483088.aspx> Accessed online 02/2021.
- [44]. Y. Chen and G. De Luca, "VIPL: Visual IoT/Robotics Programming Language Environment for Computer Science Education," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 963-971, doi: 10.1109/IPDPSW.2016.55.
- [45]. ROBO Pro, FischerTechnik. Official Website: <http://www.fischertechnik.de/en/Home.aspx> Accessed online 02/2021.
- [46]. LabView. Official Website: <http://www.ni.com/labview/> Accessed online 02/2021.
- [47]. Flowgorithm 2015. Official Website: <http://flowgorithm.org/> Accessed online 02/2021.
- [48]. LARP, Logic of Algorithms for Resolution of Problems. Official Website: <http://www.marcolavoie.ca/larp/en/default.htm> Accessed online 02/2021.
- [49]. Raptor. Official Website: <http://raptor.martincarlisle.com/> Accessed online 02/2021.
- [50]. Visual Logic. Official Website: <http://www.visuallogic.org/download/> Accessed online 02/2021.
- [51]. Rete: The JavaScript framework for flow-based visual programming. Official Website: <https://rete.js.org/#/#lang=en&tosearch=The%20JavaScript%20framework%20for%20visual%20programming> Accessed online 02/2021.
- [52]. Kodu, Microsoft Research. Official Website: <http://www.kodugamelab.com/> Accessed online 02/2021.
- [53]. Construct 2, Scirra. Official Website: <https://www.scirra.com/> Accessed online 02/2021.
- [54]. GODOT, OKAM Studio. Official Website: <http://www.godotengine.org/> Accessed online 02/2021.
- [55]. GameSalad. From Game Player to Game Maker. Official Website: <http://gamesalad.com/> Accessed online 02/2021.
- [56]. AgentCubes: An educational programming language for children in the context of creating 3D and 2D games. Authored by Alexander Repenning. First Appeared on 2006. Official Website: <https://agentsheets.com/> Accessed online: 02/2021.
- [57]. HomeKit developed by Apple. Official Website: <http://www.apple.com/ios/homekit/> Accessed online 02/2021.
- [58]. Josè Danado and Fabio Paternò. 2015. A Mobile End-User Development Environment for IoT Applications Exploiting the Puzzle Metaphor. ERCIM News 101. Link: <http://ercim-news.ercim.eu/en101/special/a-mobile-end-user-development-environment-for-iot-applications-exploiting-the-puzzle-metaphor> Accessed online: 02/2021.
- [59]. Ruiz-Rube, I.; Mota, J.M.; Person, T.; Corral, J.M.R.; Dodero, J.M. Block-Based Development of Mobile Learning Experiences for the Internet of Things. Sensors 2019, 19, 5467. <https://doi.org/10.3390/s19245467>.
- [60]. K. E. Hendrickson, "Writing and Connecting IoT and Mobile Applications in MIT App Inventor," Master dissertation, Department of Electrical Engineering and Computer Science, MIT, May 2018.

- [61]. Nayeon Bak, Byeong-Mo Chang, Kwanghoon Choi, Smart Block: A visual block language and its programming environment for IoT, *Journal of Computer Languages*, Volume 60, 2020, 100999, ISSN 2590-1184, <https://doi.org/10.1016/j.cola.2020.100999>.
- [62]. Node-RED: programming tool for wiring together hardware devices, APIs and online services. Authored by IBM Emerging Technology. Official Website: <https://nodered.org/> Accessed online: 02/2021.
- [63]. NetLab Toolkit: An authoring system empowers the users to design and build tangible Internet of Things projects. Official Website: <https://www.netlabtoolkit.org/> Accessed online: 02/2021.
- [64]. IntelliJ Platform Plugin SDK. Authored by JetBrains. Initial Released on 01/2001. Official Website: <https://plugins.jetbrains.com/docs/intellij/creating-plugin-project.html> Accessed online: 02/2021.
- [65]. Eclipse Plugin Architecture. Official Website: https://wiki.eclipse.org/Plugin_Architecture Accessed online: 02/2021.
- [66]. Visual Studio extension API. Official Website: <https://code.visualstudio.com/api> Accessed online: 02/2021.
- [67]. Eclipse Modeling Framework: A modeling framework and code generation facility for building tools and other applications based on a structured data model. Official Website: <https://www.eclipse.org/modeling/emf/> Accessed online: 02/2021.
- [68]. Savidis A, Bourdenas T, Georgalis J. An adaptable circular meta-IDE for a dynamic programming language. In *Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007)*, Luxemburg, 2007; 99–114. Available at: <http://www.ics.forth.gr/hci/files/plang/sparrow.pdf>.
- [69]. Neeraja Subrahmaniyan, Laura Beckwith, Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Vaishnavi Narayanan, Karin Bucht, Russell Drummond, and Xiaoli Fern. 2008. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 617-626.
- [70]. Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. 2012. End-user Debugging Strategies: A Sensemaking Perspective. *ACM Transaction on Computer-Human Interaction* 19, 1, Article 5 (May 2012), 28 pages. <https://doi.org/10.1145/2147783.2147788>.
- [71]. Grigoreanu, V.; Beckwith, L.; Fern, X.; Yang, S.; Komireddy, C.; Narayanan, V.; Cook, C.; Burnett, M., "Gender Differences in End-User Debugging, Revisited: What the Miners Found," *Visual Languages and Human-Centric Computing*, 2006. VL/HCC 2006. IEEE Symposium on , vol., no., pp.19,26, 4-8 Sept. 2006 doi: 10.1109/VLHCC.2006.24.
- [72]. Chintakovid, T.; Wiedenbeck, S.; Burnett, M.; Grigoreanu, V., "Pair Collaboration in End-User Debugging," *Visual Languages and Human-Centric Computing*, 2006. VL/HCC 2006. IEEE Symposium on , vol., no., pp.3,10, 4-8 Sept. 2006 doi: 10.1109/VLHCC.2006.36.
- [73]. Kim, C., Yuan, J., Vasconcelos, L. et al. Debugging during block-based programming. *Instr Sci* 46, 767–787 (2018). <https://doi.org/10.1007/s11251-018-9453-5>.

- [74]. Daniel W. Barowy, Emery D. Berger, and Benjamin Zorn. 2018. ExceLint: automatically finding spreadsheet formula errors. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 148 (November 2018), 26 pages. DOI: <https://doi.org/10.1145/3276518>.
- [75]. R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.
- [76]. Chris Chambers and Martin Erwig. Combining Spatial and Semantic Label Analysis. In *VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 225–232, Washington, DC, USA, 2009. IEEE Computer Society.
- [77]. Valentina I. Grigoreanu, Margaret M. Burnett, and George G. Robertson. 2010. A strategy-centric approach to the design of end-user debugging tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 713-722. DOI=10.1145/1753326.1753431 <http://doi.acm.org/10.1145/1753326.1753431>.
- [78]. R. Abraham and M. Erwig, "GoalDebug: A Spreadsheet Debugger for End Users," 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, 2007, pp. 251-260, doi: 10.1109/ICSE.2007.39.
- [79]. Linda Werner, Shannon Campe, and Jill Denner. 2012. Children learning computer science concepts via Alice game-programming. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 427–432. DOI: <https://doi.org/10.1145/2157136.2157263>.
- [80]. A. J. Ko and B. A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'04)*. ACM, New York, NY, USA, 151–158. <https://doi.org/10.1145/985692.985712>.
- [81]. James Devine, Joe Finney, Peli de Halleux, Michał Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *SIGPLAN Not.* 53, 6 (June 2018), 19–30. DOI: <https://doi.org/10.1145/3299710.3211335>.
- [82]. Tynker web IDE: The Debugger Tool. Official Website: <https://www.tynker.com/blog/articles/ideas-and-tips/debugger/> Accessed online: 02/2021.
- [83]. Blockly Step Execution with JS Interpreter. Official Website: <https://blockly-demo.appspot.com/static/demos/interpreter/step-execution.html> Accessed online: 02/2021.
- [84]. Savidis A., Savaki C. (2020) Complete Block-Level Visual Debugger for Blockly. In: Ahram T., Karwowski W., Pickl S., Tajar R. (eds) *Human Systems Engineering and Design II. IHSED 2019. Advances in Intelligent Systems and Computing*, vol 1026. Springer, Cham. https://doi.org/10.1007/978-3-030-27928-8_43.
- [85]. Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering End Users in Debugging Trigger-Action Rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, Paper 388, 1–13. DOI: <https://doi.org/10.1145/3290605.3300618>.

- [86]. Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16). Association for Computing Machinery, New York, NY, USA, 3227–3231. DOI: <https://doi.org/10.1145/2858036.2858556>.
- [87]. Marco Manca, Fabio, Paternò, Carmen Santoro, Luca Corcella, Supporting end-user debugging of trigger-action rules for IoT applications, International Journal of Human-Computer Studies, Volume 123, 2019, Pages 56-69, ISSN 1071-5819, <https://doi.org/10.1016/j.ijhcs.2018.11.005>.
- [88]. Corno F., De Russis L., Monge Roffarello A. (2019) My IoT Puzzle: Debugging IF-THEN Rules Through the Jigsaw Metaphor. In: Malizia A., Valtolina S., Morch A., Serrano A., Stratton A. (eds) End-User Development. IS-EUD 2019. Lecture Notes in Computer Science, vol 11553. Springer, Cham. https://doi.org/10.1007/978-3-030-24781-2_2.
- [89]. Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys '16). Association for Computing Machinery, New York, NY, USA, 133–142. DOI:<https://doi.org/10.1145/2993422.2993426>.
- [90]. F. Gringoli, N. Ali, F. Guerrini and P. Patras, "A Flexible Framework for Debugging IoT Wireless Applications," 2018 Workshop on Metrology for Industry 4.0 and IoT, Brescia, 2018, pp. 230-235, doi: 10.1109/METROI4.2018.8428337.
- [91]. Simulics Platform Simulator: A Deeper Insight into Your System. Official Website: http://www.simulics.com/index_en.php#simulator Accessed online: 02/2021.
- [92]. Khelif, Mohamed Amine & Lorandel, Jordane & Romain, Olivier & Regnery, Matthieu & Baheux, Denis. (2019). A Versatile Emulator of MitM for the identification of vulnerabilities of IoT devices, a case of study: smartphones. 1-6. 10.1145/3341325.3342019.
- [93]. K. Kawada and T. Ohta, "An Emulator for Debugging Service Programs in Ad Hoc Networks," 2009 Fourth International Conference on Software Engineering Advances, Porto, 2009, pp. 326-330, doi: 10.1109/ICSEA.2009.54.
- [94]. Google Docs, web-based software office suite offered by Google within Google Drive. Developed in JavaScript. Released on 2006. Official website: <https://www.google.com/docs/about/> Accessed online 02/2021.
- [95]. Office Online, online office suite offered by Microsoft. Released on 2010. Official site: <https://products.office.com/en/free-office-online-for-the-web> Accessed online 02/2021.
- [96]. Git: a distributed version-control system for tracking changes in source code during software development. Initial released 2005. Author: Linus Torvalds. Official Website: <https://git-scm.com/> Accessed online 02/2021.
- [97]. SVN: a software versioning and revision control system distributed as open source under the Apache License. Apache Software Foundation. Initial Release 2000. Official Website: <https://subversion.apache.org/> Accessed online 02/2021.

- [98]. Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting Developers' Coordination in the IDE. In Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '15). Association for Computing Machinery, New York, NY, USA, 518–532.
- [99]. Sourcetree: a software tool that visualizes and manages repositories. Offered by Atlassian. Official site: <https://www.sourcetreeapp.com/> Accessed online 02/2021.
- [100]. Max Goldman, Greg Little, and Robert C. Miller. 2011. Real-time collaborative coding in a web IDE. In Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11). ACM, New York, NY, USA, 155-164. DOI=10.1145/2047196.2047215 DOI: <http://doi.acm.org/10.1145/2047196.2047215>.
- [101]. Codiad Web-based IDE framework. Started on 2012 from Fluidbyte. Official Website: <http://codiad.com/> Accessed online 02/2021.
- [102]. Soroush Ghorashi and Carlos Jensen. 2016. Jimbo: a collaborative IDE with live preview. In Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '16). Association for Computing Machinery, New York, NY, USA, 104–107. DOI: <https://doi.org/10.1145/2897586.2897613>.
- [103]. Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. 2010. Saros: an eclipse plug-in for distributed party programming. In Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '10). Association for Computing Machinery, New York, NY, USA, 48–55. DOI: <https://doi.org/10.1145/1833310.1833319>.
- [104]. Kristy Elizabeth Boyer, August A. Dwight, R. Taylor Fondren, Mladen A. Vouk, and James C. Lester. 2008. A development environment for distributed synchronous collaborative programming. SIGCSE Bull. 40, 3 (June 2008), 158–162. DOI: <https://doi.org/10.1145/1597849.1384315>.
- [105]. Remote Collab: open-source SublimeText plugin for remote pair programming. Developed by TeamRemote. Started on 2014. Official Website: <http://teamremote.github.io/remote-sublime/> Accessed online 02/2021.
- [106]. Sublime Text: A sophisticated text editor for code, markup and prose. Developed by Sublime HQ, Author: Jon Skinner. Official Website: <https://www.sublimetext.com/> Accessed online: 02/2021.
- [107]. Teletype: Collaborate in real time in Atom. Started on 2017. Official Website: <https://teletype.atom.io/> Accessed online 02/2021.
- [108]. Atom: A hackable text editor for 21st century. Developed by GitHub (subsidiary of Microsoft). Official Website: <https://atom.io/> Accessed online: 02/2021.
- [109]. Codeshare: Share Code in Real-time with Developers. Created by Lee Munroe and Tejesh Mehta. Official Website: <https://codeshare.io/> Accessed online 02/2021.
- [110]. Xinyue Deng. Group Collaboration with App Inventor. Thesis: M. Eng., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2017.
- [111]. Fukuma Y., Tsutsui K., Takada H., Piumarta I. (2017) A Scratch-Based Collaborative Learning System with a Shared Stage Screen. In: Yoshino T.,

- Yuizono T., Zurita G., Vassileva J. (eds) Collaboration Technologies and Social Computing. CollabTech 2017. Lecture Notes in Computer Science, vol 10397. Springer, Cham.
- [112]. B. Selwyn-Smith, C. Anslow, M. Homer and J. R. Wallace, "Co-located Collaborative Block-Based Programming," 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Memphis, TN, USA, 2019, pp. 107-116.
- [113]. N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt, "TouchDevelop: app development on mobile devices," in FSE, Demo, 2012, pp. 39:1–39:2.
- [114]. Jonathan Protzenko, Sebastian Burckhardt, Michał Moskal, and Jedidiah McClurg. 2015. Implementing real-time collaboration in TouchDevelop using AST merges. In Proceedings of the 3rd International Workshop on Mobile Development Lifecycle (MobileDeLi 2015). Association for Computing Machinery, New York, NY, USA, 25–27. DOI:<https://doi.org/10.1145/2846661.2846672>.
- [115]. Al-Jarrah, Ahmad & Pontelli, Enrico. (2015). AliCe-ViLlagE Alice as a Collaborative Virtual Learning Environment. Proceedings - Frontiers in Education Conference, FIE. 2015. 10.1109/FIE.2014.7044089.
- [116]. Microsoft Visual Studio Live Share: enables developers to collaborate in real-time. Developed by Mixrosoft. Initial Released on 2017. Official Website: <https://visualstudio.microsoft.com/services/live-share/> Accessed online 02/2021.
- [117]. Code With Me: Plugin of the IntelliJ IDEA. First released: 09/2020. Official Website: <https://www.jetbrains.com/help/idea/code-with-me.html> Accessed online: 02/2021.
- [118]. Nordio, M., Meyer, B., & Estler, H. (2011). Collaborative Software Development on the Web. ArXiv, abs/1105.0768.
- [119]. H. C. Estler, M. Nordio, C. A. Furia and B. Meyer, "Collaborative Debugging," 2013 IEEE 8th International Conference on Global Software Engineering, Bari, 2013, pp. 110-119, doi: 10.1109/ICGSE.2013.21.
- [120]. Michael A. Miljanovic and Jeremy S. Bradbury. 2017. RoboBUG: A Serious Game for Learning Debugging Techniques. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17). Association for Computing Machinery, New York, NY, USA, 93–100. DOI: <https://doi.org/10.1145/3105726.3106173>.
- [121]. Venigalla, A., Chimalakonda, S. G4D - a treasure hunt game for novice programmers to learn debugging. Smart Learn. Environ. 7, 21 (2020). <https://doi.org/10.1186/s40561-020-00129-4>.
- [122]. Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: an online tool to help novice programmers improve their debugging skills. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018). Association for Computing Machinery, New York, NY, USA, 159–164. DOI: <https://doi.org/10.1145/3197091.3197098>.
- [123]. M. J. Lee, "Gidget: An online debugging game for learning and engagement in computing education," 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Melbourne, VIC, 2014, pp. 193-194, doi: 10.1109/VLHCC.2014.6883051.

- [124]. Chung-Fang Chiu and Hsing-Yi Huang, "Guided Debugging Practices of Game Based Programming for Novice Programmers," *International Journal of Information and Education Technology* vol. 5, no. 5, pp. 343-347, 2015.
- [125]. Lee, VCS, Yu, YT, Tang, CM, Wong, TL, Poon, CK. ViDA: A virtual debugging advisor for supporting learning in computer programming courses. *J Comput Assist Learn.* 2018; 34: 243– 258. <https://doi.org/10.1111/jcal.12238>.
- [126]. Jim Etheredge. 2004. CMeRun: program logic debugging courseware for CS1/CS2 students. *SIGCSE Bull.* 36, 1 (March 2004), 22–25. DOI: <https://doi.org/10.1145/1028174.971311>.
- [127]. Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. 2008. Backstop: a tool for debugging runtime errors. *SIGCSE Bull.* 40, 1 (March 2008), 173–177. DOI: <https://doi.org/10.1145/1352322.1352193>.
- [128]. Tsuruko Egi and Akira Takeuchi. 2007. An Analysis on a Learning Support System for Tracing in Beginner's Debugging. In *Proceedings of the 2007 conference on Supporting Learning Flow through Integrative Technologies*. IOS Press, NLD, 509–516.
- [129]. Christopher Scaffidi, Andrew Dove, and Tahmid Nabi. 2016. *LondonTube: Overcoming Hidden Dependencies in Cloud-Mobile-Web Programming*. Association for Computing Machinery, New York, NY, USA, 3498–3508. DOI: <https://doi.org/10.1145/2858036.2858076>.
- [130]. L. Ganesan, C. Scaffidi and A. Dove, "Support for learning while debugging in a distributed visual programming language," 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Raleigh, NC, 2017, pp. 267-271, doi: 10.1109/VLHCC.2017.8103477.
- [131]. Code Snippets in Visual Studio Code. Official Website: <https://code.visualstudio.com/docs/editor/userdefinedsnippets> Accessed online: 02/2021.
- [132]. Wingware Python Code Snippets. Oficial Website: <https://wingware.com/doc/edit/snippets> Accessed online: 02/2021.
- [133]. M. Ichinco and C. Kelleher, "Towards better code snippets: Exploring how code snippet recall differs with programming experience," 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Raleigh, NC, 2017, pp. 37-41, doi: 10.1109/VLHCC.2017.8103448.
- [134]. TagMyCode: IntelliJ pluggin supports mechanism of code snippets Official Website: <https://plugins.jetbrains.com/plugin/7540-tagmycode> Accessed online 02/2021.
- [135]. Window.postMessage method safely enables cross-origin communication between Window objects. Official Website: <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage> Accessed online: 02/2021.
- [136]. Tillmann, Nikolai & Moskal, Michaa & Halleux, Jonathan & Fähndrich, Manuel. (2011). *TouchDevelop–Programming Cloud-Connected Mobile Devices via Touchscreen*. 10.1145/2048237.2048245.
- [137]. JSON Schema validator: JavaScript Library. Authored by: Tom de Grunt tom@degrunt.nl, Released on 2012-2015. Official Web-page: <https://github.com/tdegrunt/jsonschema#readme> Accessed online: 02/2021.
- [138]. Lodash template: interpolate data properties in “interpolate” delimiters, HTML-escape interpolated data properties in “escape” delimiters. Lodash

- JavaScript Library released on 2009. Last updated on 9th May of 2020. Official Website: <https://lodash.com/docs/4.17.15#template> Accessed online: 02/2021.
- [139]. Eval function evaluates JavaScript code represented as a string. Official Website: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval Accessed online: 02/2021.
- [140]. Control flow and error handling in JavaScript. Official Website: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#throw_state_ment Accessed online 02/2021.
- [141]. Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2019. Interleaving anomalies in collaborative text editors. In Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19). Association for Computing Machinery, New York, NY, USA, Article 6, 1–7. DOI:<https://doi.org/10.1145/3301419.3323972>.
- [142]. Basma S. Alqadi and Jonathan I. Maletic. 2017. An Empirical Study of Debugging Patterns Among Novices Programmers. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). Association for Computing Machinery, New York, NY, USA, 15–20. DOI: <https://doi.org/10.1145/3017680.3017761>.
- [143]. Chintakovid, T.; Wiedenbeck, S.; Burnett, M.; Grigoreanu, V., "Pair Collaboration in End-User Debugging," Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on , vol., no., pp.3,10, 4-8 Sept. 2006 doi: 10.1109/VLHCC.2006.36.
- [144]. JavaScript WebWorkers: run scripts in background threads. The worker thread can perform tasks without interfering with the UI. Official Website: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers Accessed online: 02/2021.
- [145]. Kim, C., Yuan, J., Vasconcelos, L. et al. Debugging during block-based programming. Instr Sci 46, 767–787 (2018). <https://doi.org/10.1007/s11251-018-9453-5>.
- [146]. Palade, A., Cabrera, C., Li, F., White, G., Razzaque, M. A., & Clarke, S. (2018). Middleware for internet of things: an evaluation in a small-scale IoT environment. Journal of Reliable Intelligent Environments.
- [147]. IoTivity. An open source software framework enabling seamless device-to-device connectivity to address the emerging needs of the Internet of Things. Retrieved from <https://iotivity.org/> Accessed online: 02/2021.
- [148]. iotivity-node: Provides a JavaScript API for OCF functionality. Using IoTivity as its backend. Developed by Gabriel Schulhof. Official Website: <https://www.npmjs.com/package/iotivity-node> Accessed Online: 02/2021.
- [149]. WindowOrWorkerGlobalScope.setTimeout: method sets a timer which executes a function or specified piece of code once the timer expires. Official Website: <https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout> Accessed online: 02/2021.
- [150]. WindowOrWorkerGlobalScope.setInterval: method which repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

Official Website: <https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval> Accessed online: 02/2021.

Appendix A

In this appendix we provide information for the evaluation of the collaborative programming of our work.

Collaborative Editing Evaluation

A. Background Information

- 1) What is your age?
- 2) What is your gender?
- 3) What's your occupation?
- 4) How many hours per week do you spend in front of a computer on average?
- 5) How much experience do you have with programming?
- 6) Do you have any experience with visual programming?

B. System Usability Survey

The following SUS questionnaire was aimed to assess the usability of our system's collaboration component. The questions were answered on a scale from 1 to 5, 1 being "Strongly Disagree" and 5 being "Strongly Agree" (i.e. 5-point Likert scale).

- 1) I find the transition from Blockly editor to the Blockly workspace easy.
- 2) The collaboration component is well integrated into the Blockly workspace.
- 3) I find the collaboration process unnecessarily complex.
- 4) I find the collaboration User Interface intuitive and easy to use.
- 5) I feel confident using the application with guidance.
- 6) I can use the application in the future without any help.
- 7) The collaboration toolset offers limited options.
- 8) I would like to use the collaboration tool for my personal projects with my family or friends.
- 9) I don't see the point of collaborating.
- 10) I find the application useful for teaching and learning purposes.

C. Freeform Questions

- 1) As you see it, what are the advantages and disadvantages of using the collaborative visual programming workspace for Blockly over using classic Blockly Editor?
- 2) Do you find the application useful? If yes, what uses do you have in mind? Do you think it could be used for teaching and learning purposes? Explain your thoughts briefly.
- 3) Do you have any suggestions for possible improvements on existing features? Any features would like to be added? Explain your suggestions briefly.

Appendix B

In this appendix we provide a list of demos that constructed in order to demonstrate our work.

List of Demos

A. Building smart automations

- Description: Using visual programming workspace components, we craft automations for daily tasks at home.
- Link: <https://www.youtube.com/watch?v=ltZKqMlnEIE>

B. Running and testing automations

- Description: In the first part, we run automations that are developed in previous demo. In the second part, we use the debugger and simulator in order to test the automations.
- Link: <https://www.youtube.com/watch?v=KQ1j3uRPZ-w>

C. Let's code together

- Description: Three collaborators work together in order to develop their office automations.
- Link: <https://www.youtube.com/watch?v=Gg7fnA34RF4>