Simulated IoT Runtime with Virtual Smart Devices: Debugging and Testing End-user Automations

Anthony Savidis[1,2] Yannis Valsamakis[2] and Dimitris Linaritis[2]
[1]Institute of Computer Science, FORTH, Heraklion, Crete, Greece
[2]Department of Computer Science, University of Crete, Greece

Keywords: Internet of Things, Smart Automations, Visual Programming Languages, End-user Development.

Abstract: The notion of end-user programming gains increasing attention in the context of the Internet of Things (IoT) as a promising way to enable users develop personalized automations by deploying visual programming tools. In an IoT ecosystem, devices may be either invisible to users, embedded or hardly locatable, sometimes physically inaccessible. In this sense, testing becomes very challenging and difficult, since bringing physical devices to certain states may be either impractical (e.g. window and door sensors) or overall unsafe (e.g. fire or smoke sensors). It is crucial that trials are carried out in a protected, virtual environment, not the physical one. In this context we discuss a simulated runtime that addresses the challenges of testing end-user automations by entirely virtualizing devices. In this runtime, tests are not confined to a particular location, but may be carried out anywhere and anytime, totally disengaged from the physical ecosystem, with all user tools residing in any typical mobile machine, capable to fully operate standalone in test mode. Finally, when automations involve time and scheduling, for practical reasons, time itself can be simulated so that testing is done on demand, not following or waiting the pace of physical time.

1 INTRODUCTION

The Internet of Things (IoT) is a rapidly-growing domain, constantly evolving in terms of infrastructures, integrated solutions, development tools and best practices. Technically, the IoT domain rents its roots to ubiquitous computing, which in the late 90s envisioned the future as ecosystems of distributed computation and interaction resources. In the context of user-interface technology this idea was at that time abstracted by the concept of *beyond the desktop* interactions. Some interaction paradigms that appeared in this early period included the following features: treating environments as displays, projecting display output on various surfaces, using physical objects for input, putting main emphasis on hand gestures and body postures, and deploying public shared screens and projectors.

Such works tried to preserve computational ubiquity by treating interaction as an activity involving directly the environment. However, the entrance to the smartphone era caused a huge paradigm shift, with the vision of information and computation anywhere and anytime becoming fully instantiated. The user-interface technology for smartphones progressed rapidly, supported with novel interaction styles and advanced software libraries. The latter turned interaction in mobile user machines as the prevalent interaction paradigm in the new era, effectively disrupting past ideas and concepts related to beyond the desktop interactions.

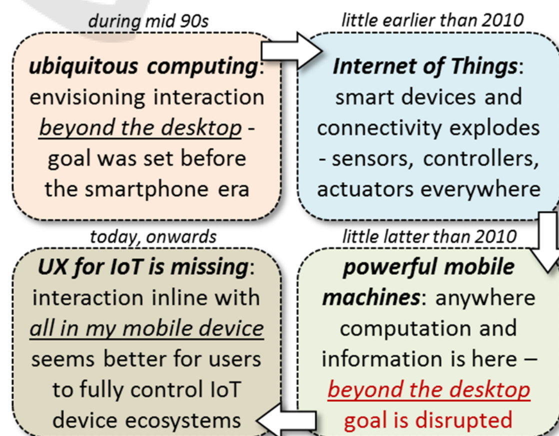

Figure 1: The disruption of the beyond-the-desktop metaphor well after the entrance to the smartphone era.

Savidis, A., Valsamakis, Y. and Linaritis, D.
Simulated IoT Runtime with Virtual Smart Devices: Debugging and Testing End-user Automations.
DOI: 10.5220/0010714400003058
In *Proceedings of the 17th International Conference on Web Information Systems and Technologies (WEBIST 2021)*, pages 145-155
ISBN: 978-989-758-536-4; ISSN: 2184-3252
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Simultaneously, IoT grew with a large variety of mission-specific devices, mostly in the category of sensors, actuators and controllers (Dachyar et al, 2019), while large-scale infrastructures started to proliferate. The previous situation, which is depicted under Figure 1, caused a technological gap: while device ecosystems constantly grow, the real benefits to daily life for individual consumers are lacking.

The latter is explained by the fact that everyday automations are highly personalized in nature, being technically small-scale applications, something that implies a niche-market with a small industrial interest. This also explains why the idea of end-user development quickly received attention and is now considered a very promising solution. Not only it may address this gap, but it is fully aligned to the need of *everything in my mobile*, enabling the management and execution of automations to be entirely handled via a typical smartphone device.

Figure 2: End-user tool layers required to enable crafting of personalized smart automations (from Savidis, 2021).

1.1 Contribution

Supporting the end-user development of smart automations entails a number of challenges that can be only addressed by offering very powerful but also user-friendly toolchains. The required layers of functionality are depicted in Figure 2, with testing being the upper level that today is less explored and examined in the context of end-user development.

Due to the highly distributed nature of IoT device ecosystems, it is crucial that testing can be carried out in a protected, virtual environment, not the physical one, since bringing physical devices to certain states may be either impractical (e.g. window and door sensors) or overall unsafe (e.g. fire or smoke sensors). In this context, out contribution is the *full-scale implementation of a simulated IoT runtime*, enabling end-users carry out isolated testing and debugging of smart-automations, with virtual devices and virtual-time control, independently and physically away of the actual IoT device ecosystem.

2 RELATED WORK

We briefly review most popular tools for visual programming in the IoT domain, judging their testing facilities.

HomeKit (HomeKit, 2021) from Apple allows control connected home accessories (if compatible with the system), and supports to some degree user-defined automations as combinations of accessory control actions. Not an end-user solution as such, focuses mostly on premade smart home solutions with emphasis on advanced configurations.

Puzzle (Danado and Patterno, 2015) is a visual development system for automations with smart IoT objects adopting the jigsaw metaphor. The system is primitive, without the full-scale capacity of common VPLs, entirely lacking testing or simulation tools.

Wia (Wia, 2021) is a cloud-based IoT development platform for linking devices, services and sensors using Wia Flow Studio. This system is better for service composition, while for testing only the real service elements can be deployed.

Embrio (Embrio, 2021) offers a drag-and-drop visual programming interface for Arduino, requiring connection to the actual circuit and peripherals upon testing, lacking any debugging facilities.

XOD (XOD, 2021) is a microcontroller programming platform with a visual interface. It is based on the node model, which can represent sensors, motors, or a piece of functional code like comparison operations, text operations, and so on. As with all previous systems, testing requires connectivity of the real devices.

Zenodys (Zenodys, 2021) allows developers create IoT apps by organizing dataflow connections. It is an advanced platform for predictive maintenance, real-time control systems and product line automation, rather than typical non-professional end-users and personal automation development. All testing is done in the field by professionals.

Node-Red (Node-Red, 2021) is a visual flow-based system for wiring hardware devices with input and output connections, but relies on JavaScript for more elaborate algorithmic features. Hence, it is more complicated for non-professionals while, as in all previous tools, tests must be performed with the real connected devices.

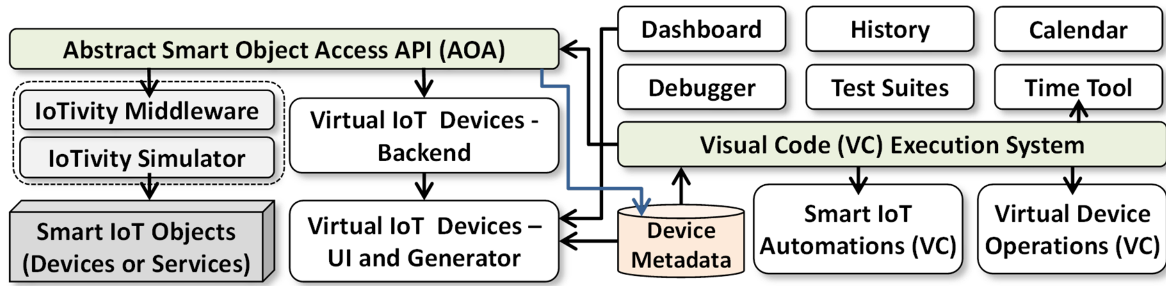

Figure 3: Software architecture of the integrated test runtime.

In summary, there are various tools focusing on visual programming, some of which could be used to build smart IoT automations. However, testing no special care is taken for testing support, treated as a process that is carried out within real infrastructures and device ecosystems. The latter is impractical, unsafe and for certain cases even infeasible.

3 SOFTWARE ARCHITECTURE

Our testing tools are part of large-scale Integrated Development Environment (IDE) for visual-programming, which relies on the (Blockly, 2021) visual programming editor and the (IoTivity, 2021) middleware for smart objects. The software architecture of our integrated test runtime is illustrated under Figure 3 (the rest of the IDE components are skipped for clarity).

In the IoT era, heterogeneity is a fundamental and likely unavoidable characteristic, concerning networking, protocols and device APIs. In this context, diversity is expected to further proliferate, but it can be technically confined to the lower levels, with extra decomposition, better middleware and more service layers. In our architecture, for this purpose there is a specific layer named *abstract object access* (AOA). This layer sits on top of the *IoTivity middleware*, which is already a level of abstraction over device protocols. The entire backbone of our testing tools sits on top of the AOA layer, something that makes testing instruments resilient to scaling and tolerant to change. In particular, to accommodate device virtualization we had to allow switching between physical and virtual device access, at the backend, something that we introduced as a built-in feature of the AOA API.

Overall, device ecosystems are expected to constantly grow, with decentralization becoming a necessity to break or avoid monoliths. However, certain infrastructures are naturally huge by design. In this framework, the notion of *ecosystem federations* appeared (see Figure 4), with cross-federation interoperability enabling the disciplined orchestration and control of all constituent ecosystems. In our work, the AOA of a local ecosystem is the gateway to other ecosystems and encapsulates the cross-federation API. The AOA is already visible to, and deployable by, the entire IoT testing runtime, something that effectively results in the notion of *cross-federation testing*.

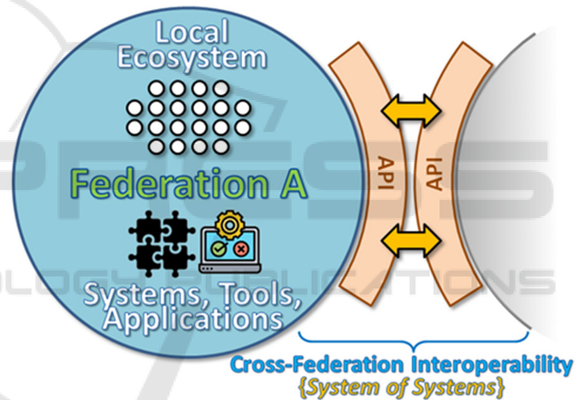

Figure 4: Notion of ecosystem federations and cross-federation interoperability through well-documented APIs, resulting in systems of systems (Savidis, 2021).

4 VIRTUAL DEVICES

In testing mode, virtual GUI counterparts for all smart devices of the local ecosystem are deployed, *on top of the middleware*, which, as earlier mentioned (see Figure 3), are all linked directly to the AOA and thus become inseparable to the physical devices for rest of the runtime.

4.1 Automatic Device User-interfaces

Smart device information is retrieved via the middleware, during every device scanning process, that is regularly initiated on-demand by the end-user

inside a particular local device ecosystem. Such information is gathered by the AOA and populates an up-to-date database of device meta-data (see Figure 3, blue arrow), containing information regarding device properties and operations in the form of typed records and typed function signatures. Based on such device meta-data, we apply an automatic widget generation technique similar to (Dewan, 2010), where the device GUI is composed by mapping device field data-types to corresponding widget classes (see Figure 5).

Besides GUI creation, it is crucial to keep the GUI state always synced to the backend holding a database of the virtual device state records. This ensures that when visual code fragments update any device, the change is instantly mapped to the device GUI. For this purpose, upon creation of the widgets corresponding to device properties, the GUI generator will also install an internal event handler that keeps the two device images (GUI and backend) always in sync to each other (see Figure 6).

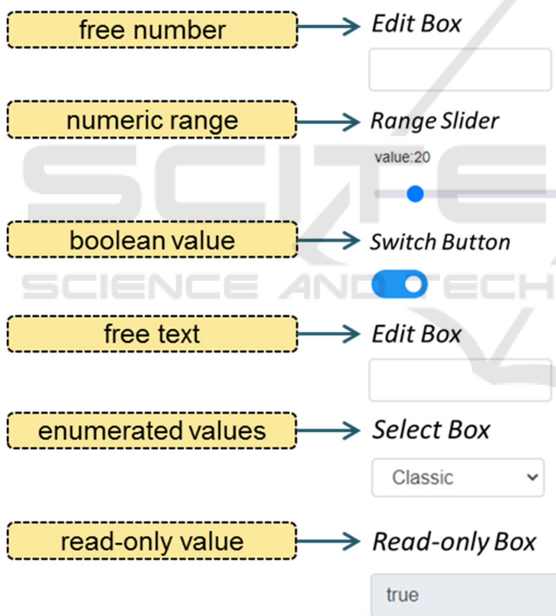

Figure 5: Automatic GUI generation for interactive virtual devices relies on the mapping of device property types (left) to specific widget classes (right).

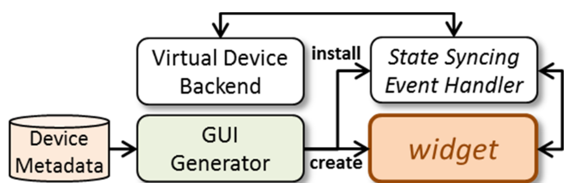

Figure 6: Auto-syncing between the device GUIs and the respective backend device state via a common event handler propagating state updates.

4.2 Visually Programmed Operations

When trying to virtualize smart devices there is one issue that cannot be automatically addressed via GUI generation. More specifically, besides device properties, device operations are also enlisted within device meta-data with typed function signatures. Now, with such information we may directly generate a GUI so that end-users can supply all required argument values (if any) together with a push-button to internally invoke the underlying device operation with the supplied arguments. As we explain in a latter section, this feature is fully supported when physical devices are deployed, as part of a live device dashboard which enables directly affect the physical devices.

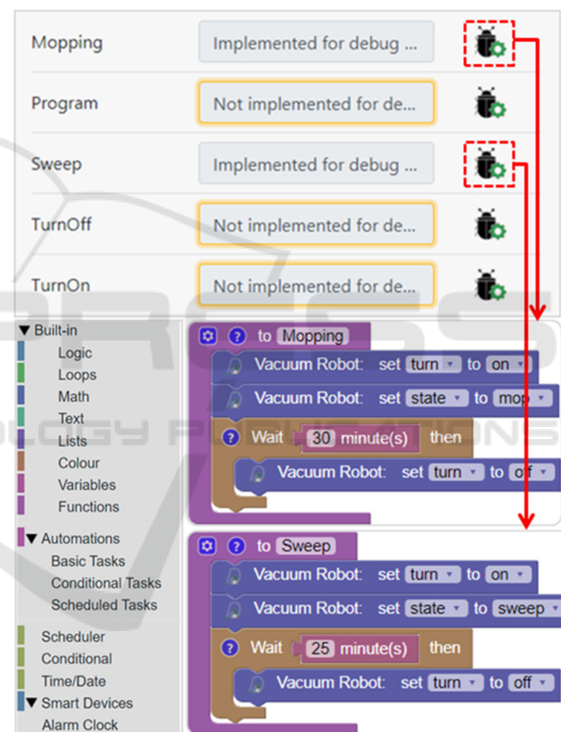

Figure 7: Simulating the logic of device operations for virtual devices to mirror physical operations during test mode through visual programming – here the *Mopping* and *Sweep* operations are programmed to schedule respective state changes.

However, when running in test mode, no physical devices are really connected, thus no internal device operations are there to be invoked. While trying to resolve this issue, we observed that many device operations, apart from performing some physical action, also update device state values to indicate the new operational state. In a virtualized testing setup, changes to these device properties

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

suffice to make the virtual device look in the respective operational mode. Based on these remarks we enabled end-users simulate the behavior of such operations by implementing them through visual code (see example of Figure 7). Typically, the visual code for such simulated operations will only have to accordingly change the state of device properties.

4.3 Live Device Access and Control

As mentioned earlier, device meta-data provides enough information to generate a fully-functional GUI through which device property updates and device operation invocations are directly possible. Effectively, such a GUI offers live device access and control, with state synced to the virtual device state, and invocation of operations resulting in the execution of the code for simulated device operations supplied by end-users (as explained in the previous section). An example of such a GUI for the A/C device is shown under Figure 8.

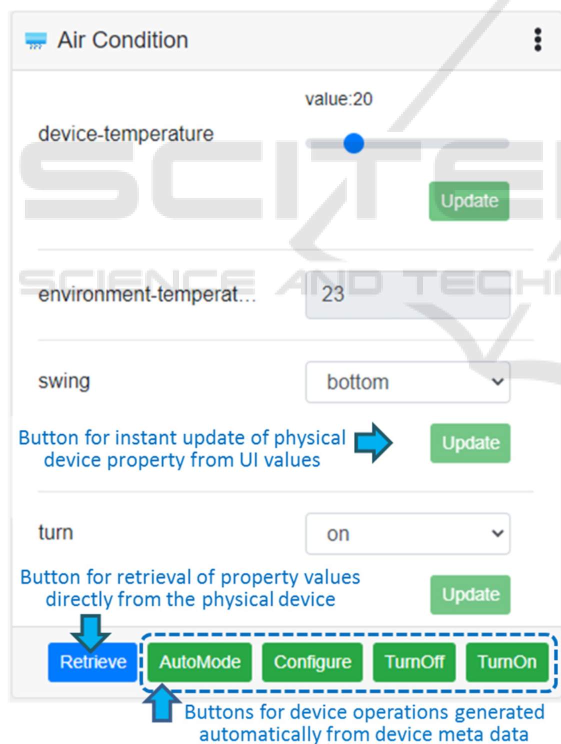

Figure 8: A/C virtual device with a GUI generated automatically from the respective device metadata.

It should be noted that the same GUI *cannot* be used exactly as it is in case of physical device usage. In particular, for most smart devices, all property changes will occur either in response to normal device functioning or as a result of operations requested by the end-user, but never directly as internal system-level requests for explicit property updates. In this sense, when the GUI is embedded inside the global device dashboard (as will be discussed latter) for real physical device deployment, all device properties become read-only, and all respective *Update* buttons are removed.

5 SIMULATED RUNTIME

5.1 Bypassing the Middleware

The middleware is a necessary communication layer above smart device ecosystems and custom networking protocols, enabling to effectively handle heterogeneity. But in testing mode, using all simulation tools, we needed a way to bypass the basic middleware and redirect all requests and notifications to the virtual device backend. This has been accomplished through the abstract object access (AOA) layer in our system architecture, an extra layer for smart object management sitting on top of the real middleware (in our case IoTivity that has been deployed).

Effectively, only when running under simulation mode the AOA internally forwards all respective messages related to device access and operations to the virtual device backend. Otherwise, the *IoTivity* middleware is directly used.

5.2 Simulated Physical-Device Layer

Physical layer simulation implies that smart devices profiles may be registered at the low-level and *look functional for the middleware* without requiring physical presence of the devices in the environment. While offering a GUI counterpart for physical devices is something that can be handled on top of the middleware, physical device simulation itself is only possible below middleware, or at least must be offered as a feature of the middleware itself.

This ability has been critical for the early development phases of our tools, not for end-users, neither it is required during runtime as part of our simulation machinery. In particular, we needed very early a way to test our tools using the middleware, however, without purchasing, installing, scanning, registering and connecting actual physical devices underneath, but only simulated ones.

For this purpose, as indicated in our system architecture, we used the *IoTivity Simulator* (IoTivity Simulator, 2021), an Eclipse plugin that is capable to simulate smart devices as OIC resources

(Open Interconnect Consortium, 2021). The IoTivity Simulator is accompanied with a *service provider* that seamlessly manages creation, deletion, request handling and notifications of all simulated resources. Furthermore, it handles requests received by clients, and sends appropriate responses back to them. To simulate smart devices through the simulator we had to express their REST (Representational State Transfer) APIs in RAML (RESTful API Modeling Language), the latter being a way of describing RESTful APIs so that they can be readable by humans. A REST API (also known as RESTful API) conforms to the constraints of REST architectural style and allows interaction with RESTful web services. In this context, we modeled the GET request for retrieving and the POST request for updating smart-device states respectively.

5.3 Calendar and Virtual-time Tool

The action calendar offers a live view of all scheduled automation actions (see Figure 9, top part). Every scheduled action, specified through the custom visual blocks in our IDE, is internally reported at start-up to the action calendar serving a twofold role for the end-user: (a) it provides an overview of all scheduled activities; and (b) shows which of such activities have been already invoked, with entries shown in green and a tick icon at right.

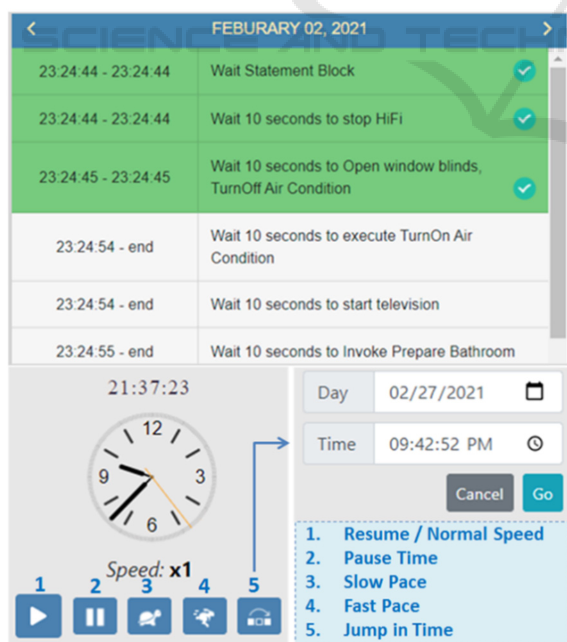

Figure 9: Top part: calendar with all scheduled automations, showing in green those already triggered; Bottom part: the virtual time tool, enabling to control pace of time and jump to a specific time and date.

The brief messages that appear on the calendar (right column, next to time or date) are the actual brief textual descriptions inserted by the developer in the corresponding scheduling visual blocks at development time. Although not elaborated in this paper, it should be mentioned that for all IoT blocks we introduced in Blockly, we support such a user-defined text explanations.

As an add-on component the activity calendar, our simulator also includes a *virtual-time* component, which enables very easily the testing of any scheduled automations, which, otherwise, would have to wait for action triggering following the normal time flow.

The reason this does not interfere with system time is due to the way we have developed scheduling logic in our toolbox: there is a time-access API, used throughout our runtime for querying the current time. In normal execution this is implemented to directly return the system clock time. However, during simulation, this API is implemented by the time simulator in a special way, enabling control the pace of time interactively (see 9, bottom part), and thus apply the current speed factor the user has chosen over the returned value of the current time. This allows far easier and quicker testing of scheduled tasks, especially when sequentially scheduled automation are defined, thus avoiding to wait for the real time to pass for respective actions to be triggered.

5.4 Dashboard and Activity History

The dashboard is a useful tool (see Figure 10) displaying live in real-time an up-to-date view of all smart-devices involved in the currently running set of smart automations, and behaves as follows:

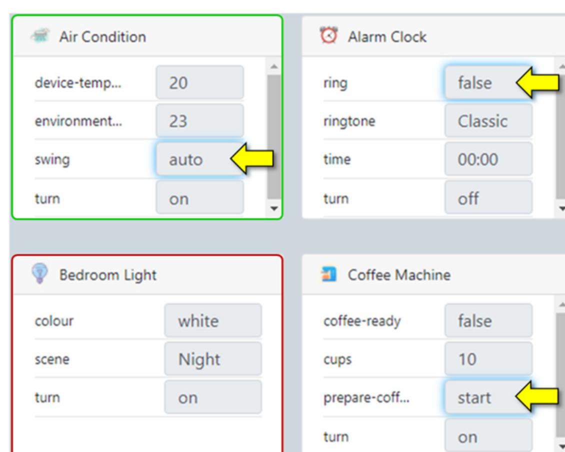

Figure 10: Part of the device dashboard – recently changed properties (see yellow arrows) are highlighted.

SCIENCE AND TECHNOLOGY PUBLICATIONS

- As new devices are discovered, they appeared in front, temporarily shown with a green border (e.g. *Air Condition* device)
- Devices out of range are shown with a red border and after a while they disappear (e.g. *Bedroom Light* device)
- When device properties change, they are highlighted (e.g. *Air Condition*, *swing* state or *Alarm Clock*, *ring* state)

The smart-device dashboard is always synced to actual device sate during runtime, while it is fully interactive, enabling directly select any device and change property values (only in test mode) or invoke operations (test mode or real operation mode). It is also important to note that *all* such changes are committed *instantly* on the smart device itself via the abstract object layer, so that the end-user visual code will indistinguishably treat them as genuine device-level state updates.

Finally, the event history is an interactive live console (see Figure 11, coloured bubbles) providing an informative view of all events triggered. It is essentially a database of annotated events that occur during runtime with the following characteristics:

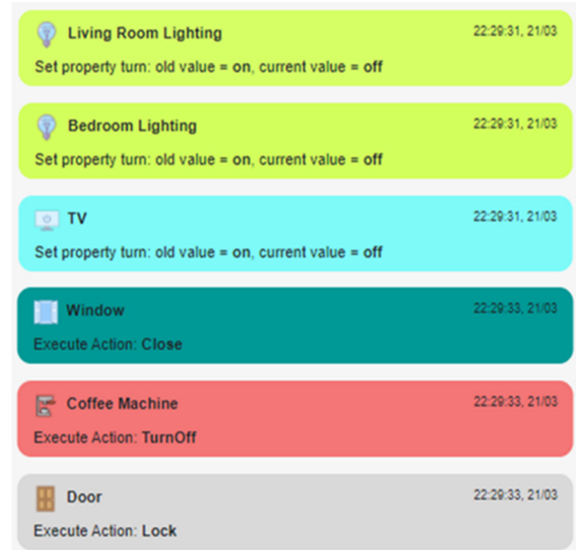

Figure 11: History with descriptions, including both device-level events and automation actions.

Figure 12: Defining, running and debugging automations with scheduled automation tests.

- All events are sequentially sorted in time, while they can be logically grouped following the smart automation they concern
- Color encoding (with chosen colors being interactively configurable) is deployed to differentiate between scheduled actions, device state changes and shifting of operational modes

Finally, the respective visual code block that actually handled an event appearing in the history can be directly tracked in the IDE by just clicking on the respective device icon appearing at the top left of every event bubble.

6 TESTING TOOLS

Our additional testing tools include the debugger, the interactive definition of automatic test suites and the support for custom checking blocks. The combined testing process using these tools is outlined under Figure 12 and commonly entails the designated steps:

1. Setting breakpoints on blocks
2. Defining a test suite
3. Running tests and stopping in breakpoints
4. Observing changes due to the test suite
5. Tracing with the debugger like step-in
6. Observing execution events in the history

6.1 Block-based Debugger

In our IDE we have incorporated and appropriately adapted the user-interface of an open source block-level debugger for Blockly from the public repository of (Savidis and Savaki, 2019). In particular, as part of the traditional variable watches tab of the original debugger, we have inserted all smart devices, while we have grouped all variables under their respective smart automations code block. Finally, we also grouped breakpoints under smart automations, so that end-users can more easily and intuitively browse and manage breakpoints.

6.2 Test Suites

Test suites are automated tests that enable users easily test the visually programmed automations, with two types of tests currently supported. The first one schedules changes in device states and the second one allows users define warnings for specific device state modifications, by optionally suspending running automations. Every test can be either set as active directly after its creation, or at the beginning of the next execution session. For the first test type (scheduled) we provide a user-interface through which the user may define the elapsed time after which a device state will be triggered (see Figure 12, label 2), or alternatively define repeated device changes at regular time intervals. Multiple device properties may be also modified as part of a single test. It should be noted that in all such cases the virtual devices are only involved, something that gives end-users the opportunity to update even read-only device fields so as to test the respective associated automations.

6.3 Check Blocks

Check blocks are a new category of visual programming blocks that we introduced in Blockly to allow more elaborate and easy testing of smart automations. They generally look similar to conditional breakpoints in debuggers or to *assertions* in programming languages, but are more close to *data breakpoints* which capture data changes. More specifically, they allow users define conditions involving device state fields, which are evaluated by every respective state change, and once becoming satisfied (i.e. true), will issue a warning or pause execution and open the debugger (see Figure 13).

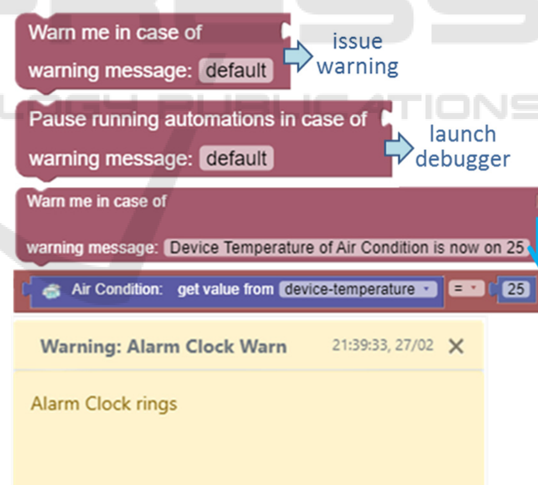

Figure 13: Check blocks and how they allow tracking invalid states or property values / ranges for smart devices.

6.4 Case Scenarios

We briefly mention a few test scenarios for everyday automations, all visually programmed through our IDE, just to give an idea of how easy and straightforward it is to test such automations with our integrated testing toolset. In particular, to make all automations of Figure 14 ready for testing, it

suffices to provide *just once* a simulated implementation with visual programming of the following operations involved in the code blocks:

- Window Blinds: *Open*, *Close*
- Air Condition: *TurnOn*, *TurnOff*
- Bedroom / Main Light: *TurnOn*, *TurnOff*
- Bathroom Light: *TurnOn*, *TurnOff*
- Coffee Machine: *TurnOn*, *TurnOff*
- Window: *Open*, *Close*

Once this step is performed, end-users are able to test all types of automations with virtual devices, in isolation, locally in their smartphone, by defining test suites, opening the calendar and dashboard, interacting directly with virtual device GUIs, viewing history logs, playing with virtual time, and opening the debugger on-demand as needed. All these activities are possible without ever connecting to real devices. Moreover, the exact same tools are usable and available when the real device ecosystem is involved, when testing is carried out in the field.

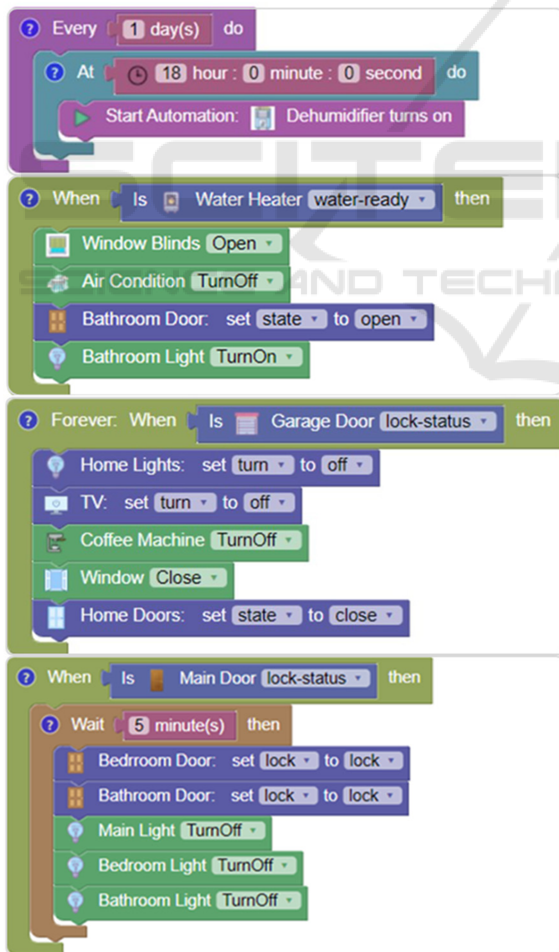

Figure 14: Home automation case scenarios for helping in everyday life and healthy living.

7 DISCUSSION

The reported work relies on end-user visual programming of smart automations as a promising solution bridging the gap between IoT device ecosystems and the present lack of high-quality personalized user experiences for everyday life. In this context, the role of an end-user programmer is very broad and may concern the consumer, friends and family, ecosystem administrators, third-party service suppliers, carers if applicable, volunteers, and so on. Although programming as such is a very specific profession, the scale and complexity of typical smart automations is very small compared to the expertise of professional developers.

In particular, as outlined under Figure 15, learning, amateur and hobby programming are all programming subdomains, however, with very custom tools and broad target population, not implying or requiring the technical skills of a typical professional software developer. Overall, we consider that the basic level of programming knowledge typical in these subdomains is generally sufficient to enable end-users craft personal smart IoT applications.

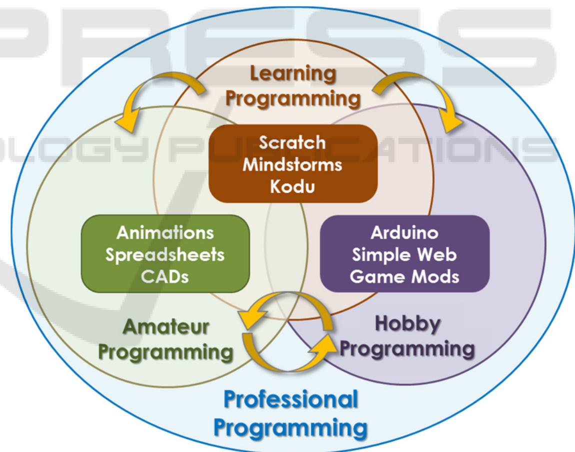

Figure 15: Different levels of programming knowledge with possible associations – all except the outer one are effectively casual or non-professional programming.

Clearly, for such *casual* programming activities there are fields and related applications that must be excluded, such as mission or safety critical automations and those involving proprietary data or personal information. Initially, the primary focus for personally crafted automations should be on home ecosystems, and likely on workplaces and leisure. Putting such creational freedom on the hands of end-users, with easy-to-use and powerful tools, is

promising, challenging and may be a real game changer for the progress of the IoT domain.

The vision of IoT device ecosystems as open, extensible and configurable infrastructures, managed by end-users via tools available in their mobile machines, is based on the prediction that IoT can more rapidly enter daily life once the following conditions are met:

- Smart infrastructures are integrated computing systems of open federated ecosystems, beyond existing monolithic installations of a single manufacturer or contractor
- IoT technology becomes commodity hardware, standalone or embedded in other equipment
- Modular IoT components become affordably available with many varying market options
- Installations may require the help of technicians, but overall should be easy for consumers to handle the process themselves
- Configuring and creating automations is treated as an assembly process managed and configured directly by the end-users

8 CONCLUSIONS

The Internet of Things proliferates as a dynamic and constantly evolving domain, constituting a primary technological backbone of distributed computing resources. Although small-scale IoT hardware becomes rapidly available, the chances for open and easier end-user development, enabling flexible manipulation and composition of such cross-vendor IoT resources, within varying hosting environments and device ecosystems, are still very limited. The present lack of user experiences for the IoT domain is also attributed to the disruption of past research in ubiquitous computing, which emphasized beyond the desktop interactions.

The recent adoption of end-user programming for smart IoT automations is better aligned to the future trend for local control from a mobile device of IoT functionality and resources through small-scale automations. In this work, we focused on the required testing instruments and we developed an integrated toolset enabling end-users test and debug automations in a protected simulated runtime.

We consider that more research work is needed in the field of end-user tools, while part of our future plans includes the design and development of: (a) an explanation wizard that can meaningfully respond to "*why did this happen*" for any event, and (b) a reverse tracer for the simulated runtime, enabling to roll back and forth in time, during debugging or when testing of smart automations.

REFERENCES

Savidis, A. (2021). *Back to the Internet of Things Future: When Everybody Crafts Personal Smart Automations*. Keynote speech. IISA 2021 Conference. DOI = http://dx.doi.org/10.13140/RG.2.2.20423.83365

Blockly (2021). Google Inc. *A JavaScript library for building visual programming editors*. https://developers.google.com/blockly Accessed online: 07/2021.

HomeKit (2021). *A software framework to configure, communicate with, and control smart-home appliances using Apple devices*. Apple Inc. https://www.apple.com/shop/accessories/all/homekit Accessed online: 07/2021

Wia (2021). *A cloud platform that makes creating IoT apps easier by linking IoT devices and external services*. https://www.wia.io/ Accessed: 07/2021

Embrio (2021). *Visual, real-time, agent-based programming for Arduino*. https://www.embrio.io/ Accessed online: 07/2021

XOD (2021). *An open-source visual programming language for microcontrollers*. https://xod.io/ Accessed online: 07/2021

Zenodys (2021). *A fully visual IoT platform for Industry* https://www.zenodys.com/ Accessed online: 07/2021

IoTivity (2021). *An open-source software framework enabling seamless device-to-device connectivity to address the emerging needs of the Internet of Things*. https://iotivity.org/ Accessed online: 07/2021

IoTivity Simulator (2021). *Simulating devices which communicate with IoTivity middleware*. https://wiki.iotivity.org/iotivity_tool_guide Accessed online: 07/2021.

Danado, J., Paternò, F. (2015). *A Mobile End-User Development Environment for IoT Applications Exploiting the Puzzle Metaphor*. ERCIM News, Issue 101, http://ercim-news.ercim.eu/en101

Open Interconnect Consortium – OIC (2021). *Standards for the development of the Internet of Things*. https://openconnectivity.org Accessed online: 07/2021

Node-Red (2021). *Low-level programming for even-driven applications.* https://nodered.org/ Accessed: 07/2021

Myers, B., Ko, A., Scaffidi, C., Oney, S., Yoon, Y., Chang, L. S-P., Kery, M.B., Jia-Jun, T. (2017). *Making End User Development More Natural*. New Perspectives in End-User Development 2017: 1-22, DOI= https://doi.org/10.1007/978-3-319-60291-2_1

Dewan, P. (2010). *A demonstration of the flexibility of widget generation*. In Proceedings EICS 2010, ACM, 315-320, DOI= DOI:10.1145/1822018.1822069

Savidis, A., Savaki, C. (2019). *Complete Block-Level Visual Debugger for Blockly*. In Proceedings of IHSED 2019, 286-292, DOI=https://doi.org/10.1007/978-3-030-27928-8_43

Dachyar, M., Zagloel, T., Saragih, L. (2019). *Knowledge growth and development: internet of things (IoT) research, 2006–2018*. Heliyon, Volume 5, Issue 8, DOI = https://doi.org/10.1016/j.heliyon.2019.e02264

APPENDIX

Sample Device Profiles

```
"properties": {
  "turn": {
    "enum": ["off", "on"],
    "type": "string",
    "description": "Turn on/off the Air Conditioning device",
    "default": "off"
  },

  "swing": {
    "enum": ["auto", "top", "bottom"],
    "type": "string",
    "default": "auto",
    "description": "Adjust the air-flow direction"
  },

  "device-temperature": {
    "type": "integer",
    "minimum": 17,
    "maximum": 33,
    "exclusiveMaximum": false,
    "description": "Air Condition Temperature",
    "default": 25
  },
```

Part of the ***device properties profile*** for the *Air Conditioning* smart object as stored in our system, used to generate automatically a GUI for the virtual device.

```
"actions": {
  "TurnOn": {
    "parameters": "array",
    "items": [],
    "description": "Action for Turning off the Air Conditioning
    device. It takes no parameters"
  },

  "TurnOff": {
    "parameters": "array",
    "items": [],
    "description": "Action for Turning on the Air Conditioning
    device. It takes no parameters"
  },

  "Configure": {
    "parameters": "array",
    "items": [{ "device-temperature": "number" }, { "swing":
    "string" }],
    "description": "Action for Configuring the Air Conditioning
    device. It takes as first parameter value for the
    device-temperature and value for the swing as second parameter"
  },
```

Part of the ***device actions profile*** for the *Air Conditioning* smart object as stored in our system, used to generate the respective operation invocation expressions that are necessary in the event handlers of the virtual devices.

```
1   #%RAML 0.8
2   title: OICAirCondition
3   version: v0.0
4   documentation:
5     - title: AirCondition
6       content: |
7         AirCondition
8   schemas:
9     - AirCondition: !include oic.r.aircondition.json
10
22
23    get:
24      description: |
25        Retrieves the current Air Conditioning device state.
26      responses:
27        200:
28          body:
29            application/json:
30              schema: AirCondition
31              example: |
32                {
33                  "turn":  "on",
34                  "device-temperature": 25,
35                  "environment-temperature": 20
36                  "swing": "auto"
37                }
```

```
39
40    post:
41      description: |
42        Sets the desired state for the Air Conditioning device.
43      body:
44        application/json:
45          schema: AirCondition
46          example: |
47            {
48              "turn":  "off",
49              "device-temperature": 25,
50              "environment-temperature": 20,
51              "swing": "top"
52            }
53      responses:
54        200:
55          body:
56            application/json:
57              schema: AirCondition
58              example: |
59                {
60                  "turn":  "on",
61                  "device-temperature": 22,
62                  "environment-temperature": 27,
63                  "swing": "bottom"
64                }
65
```

RAML specification for the *Air Conditioning* smart object, compliant to the OIC standard, with the implementation of its ***get*** and ***post*** methods.