Smart Home KNX System Simulator

Léo Alvarez & Isis Daudé

School of Computer and Communication Sciences
Master Semester Project Report

Supervisor
Samuel Chassot
EPFL / DSLAB

Supervisor
Prof. George Candea
EPFL / DSLAB

June 2022

Smart Home KNX System Simulator

Léo ALvarez & Isis Daudé
EPFL, Switzerland
leo.alvarez@epfl.ch & isis.daude@epfl.ch

1 Introduction

When designing or debugging a Smart home KNX [1] infrastructure composed of several devices, one must possess all the physical devices and wires to test the system. This implies an important cost in money and time before being able to test a certain functionality, even a simple one. The same problem arises when testing Secure and Verified Smart Home Infrastructure (SVSHI, [2]) applications for correctness, or when verifying that they achieve the desired behavior. To address this problem, we developed a simulator software that represents a KNX system without physical devices, and models its evolution in time in response to user interactions and physical world influence. With the proposed solution, one user can configure a KNX system with visual feedback, interact with it and test applications developed with SVSHI before implementing them in a real physical KNX system.

2 Background: KNX systems and SVSHI

In this section, we provide some background information about KNX systems and SVSHI software. More detailed information can be found on KNX website [1], and in SVSHI paper [2].

2.1 KNX systems

KNX is a communication protocol for smart homes that allows to establish a connection, physical or wireless, between several devices in a building. All devices are connected to a bus on which they send & receive *telegrams* (name given to KNX data packets) to interact with each other.

2.1.1 KNX Certified Devices

Devices that can connect to the bus are certified by KNX Association. They can be split in two main types: *System devices* and *End devices*. For the sake of this project, we focus on the *System device* **IP interface** and *End devices* **Sensors** and **Actuators**.

- **IP Interface**: Device that bridges an internal KNX system to an external module/network using IP protocol (e.g. sending a request from your smartphone to the IP interface to start the heater while on your way home).
- **Sensors**: Devices that perceive physical states and transmit information on the bus (e.g. temperature).
- **Actuators**: Devices that react to information from the bus and act on a physical world state (e.g. heater acts on temperature).

Each device is assigned an **Individual Address** denoting its location on the KNX bus, with the following format: **Area.Line.Device**. Also, devices exposes **communication objects** on the bus, akin to IO ports visible by the system. Each communication object can be linked to one or more *group addresses* (see Section 2.1.2).

2.1.2 KNX Bus

As stated earlier, the KNX Bus allows communication between all KNX devices in the system. It transmits information encapsulated in packets called telegrams.

For instance, an actuator can receive a telegram sent by a sensor on the bus and adjust its state accordingly. They are composed of several fields, the most important ones in our context are:

- **Address Field**: It contains the **individual address** of the sender and the destination address (**group address**).
- **Data field**: It contains the telegram's payload (e.g. temperature value or binary state).

To implement a functionality, the KNX protocol makes use of the **group address** concept: when a telegram is sent to a particular group address, all devices assigned to it accept and process the telegram, and all others ignore the telegram. For this system to work, each group address must implement exactly one functionality.

2.1.3 Configuration of KNX Systems

To configure a KNX system, users need to use the software ETS [3] developed by KNX Association. All manufacturers provide a KNX catalog entry for each of their devices, whose format vary from one manufacturer to another. Users use catalog entries to represent the whole system in ETS, define the KNX topology in the building and configure devices to achieve the desired functionalities. This is done by creating group addresses, linking them to appropriate devices' communication objects and setting their parameters.

This process is cumbersome, very error-prone and time-consuming. SVSHI was created to overcome these limitations by facilitating and verifying KNX system configurations.

2.2 SVSHI

SVSHI stands for Secure and Verified Smart Home Infrastructure acts like a KNX device in the system as it sends and receives telegrams respectively on and from the bus. By assigning it to every group address, it interprets all telegrams from the bus, runs the user application and send back telegrams to appropriate group addresses. SVSHI allows users to develop and run secured and verified Python applications in a KNX system, that respect a set of constraints (invariants) provided by the user (more details in SVSHI paper [2]. Its aim is to reduce the amount of time spent on ETS and the complexity of configurations. This leads to fewer errors committed, with easier and quicker debugging.

3 Related work

In this section, we present the current existing free simulation software for KNX systems called **KNX Virtual**. Other costly software exist but are not considered as they mainly focus on electrical connection and configuration for KNX professionals and trainings.

KNX Virtual [4] is a free and closed-source software able to simulate KNX systems. It is used in combination of ETS to display the configuration and state in real-time. But it is very limited. The virtual devices are too simple and does not exist outside the simulator: a user cannot test its configuration with real devices simulated. The amount of virtual devices available is very small, and their settings are simplified. The Graphical User Interface is not user-friendly and too complex. Finally, there is no representation of the physical world (weather, time...) that could influence the devices in the system and its behavior in time.

The goal of the proposed simulator is to overcome most of these limitations with a more usefull and truthful simulation of a real KNX system and its environment, through a user-friendly interface.

4 Simulation

We now present the simulator we created with the assumptions and choices made to represent the KNX system and the physical world it evolves in. The object-oriented programming capabilities of Python are exploited to represent and manage the system and its interactions.

4.1 Simulating a KNX system

First, we explain our choices to model KNX devices and communication in the simulator.

4.1.1 General implementation

We limited the simulation to a single room environment, because the targeted user of our simulator would typically implement a KNX system for a small space, such as a bedroom.

In the code, a central **Room** object contains all the system components' objects (KNX Bus, devices, World states, SVSHI interface, ...), as the room is shared by all of them in a physical system. A major distinction is made between the simulation of KNX components, which should be as close as possible to real KNX elements, and physical world states, which should represent environment conditions interacting with the KNX system. All elements are represented by Python class instances and their interactions are defined with specific methods respecting the general behavior of a real KNX system.

Indeed, the **KNXBus** object permits the exchange of **Telegram** objects, created using a **Payload** object (**Binary** or **Dimmer**), between **Device** objects. **GroupAddress** objects are assigned to devices defined with an **IndividualAddress** object. As explained in Section 4.2, the **World** object manages evolution of physical states and time with class instances as well (e.g. **Time**, **Temperature**, **Humidity**, **CO2**, **Light**).

4.1.2 Device interactions

As seen on Figure 1, we chose to implement three types of devices: **Actuator**, **Sensor** and **FunctionalModule** (type specific to simulator, considered as sensors by KNX).

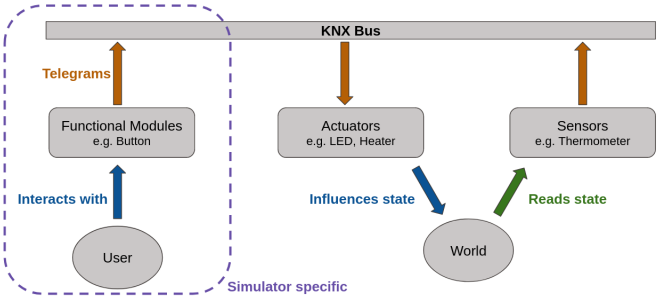

Figure 1: Components' interactions in our simulator

Most of KNX certified devices' communication objects can be represented using one of the three simulator's device classes. The abstraction of KNX devices takes place at a communication object level, making possible to implement complex devices from a combination of the ones defined in the simulator.

Their interactions with the bus, the world and the user are well-defined:

- Actuators receive telegrams from the bus, update their state depending on the information transmitted by the telegram, impact or not the physical world state. For instance, a heater is an actuator: if it receives a telegram ordering to turn on, it will start heating the room and room's temperature will rise.
- Sensors send their telegrams on the bus with information on their current state and value(s) sensed. Their value attribute is set and updated by the World object at every tick of our simulated clock (see Section 4.1.4). For instance, a thermometer will regularly send its value on the bus, which is exactly how real KNX sensors behave. In the simulator implementation, sensors are devices that users cannot interact

physically with. Users can only use sensors' value sent on the bus to implement SVSHI applications.

- FUNCTIONAL MODULES are a type of KNX sensors that we decided to introduce in order to represent user actionable devices, compared to physical states sensors. They are lightly mentioned in KNX documentation, see [5]. From a KNX point-of-view, users are part of the environment and any physical user action is considered part of the world influence on the system, that KNX sensors could capture. The distinction between Functional modules and other sensors is explained by a major behavior differences: users cannot interact with sensors. It permits to handle both of them optimally in the code, and make it also clearer for developers.

- TELEGRAMS are a representation of KNX telegrams in our simulated environment. They are composed of the addresses of the source (individual address) and of the destination (group address), but also of the data (payload). The latter is represented by an abstract class **Payload** that can be initialized with two child classes: **BinaryPayload**, used for binary content like a switch with on/off, and **FloatPayload**, used for decimal float content (e.g., temperature in degrees of a thermometer). These two data representation classes are compliant with KNX's data encoding. Indeed, The payload content of KNX telegrams respect some datatypes (called DPT) where the instances **DPTBinary** is used for boolean values, similarly to our BinaryPayload, and **DPTArray** is used for float and integers, similarly to our FloatPayload. Those two can represent most datatypes used in telegram payloads, and cover all the ones supported by SVSHI.

- KNX BUS is the representation of the physical bus used to connect all devices in a real system. In order to simulate this component, that permits communication between devices, we chose to implement a two-way *Observer Design Pattern* with a central class: **KNXBus**. As you would connect the device to the bus in the real world, we make every new actuator device *subscribe* to the KNX bus object, so that it is notified when the bus receives a telegram to an assigned group address. We make the KNX bus object *subscribe* to every new sensor/functional module device, so it can be notified when these devices want to send telegrams on the bus. The bus transmits telegrams it receives to concerned actuators. The implementation described allows for simple and complex communication through the notification concept, basically calling observer methods, without requiring to *emulate* real message exchanges through internal sockets and communication protocols, which would be very cumbersome and potentially not robust.

4.1.3 System configuration

The system configuration can be done either by:

- parsing a JSON file provided by the user,
- calling configuration functions and methods in the source code,
- dynamically interacting with the system through the Graphical User Interface (see Section 6.2),
- combining these approaches.

The easiest way is to configure the JSON file before starting the simulation. As explained in Section 6.2, users can visually set up the devices in the room using the GUI, save the configuration using the SAVE button and launch the simulation with the generated JSON configuration file, that can be completed manually by users.

Regarding KNX components, the configuration of devices includes their class (e.g. LED, Button), their location in the room and on the KNX bus, and their assigned group address(es).

On the other hand, for World components, users can configure the *system_dt*, the *speed_factor* (see Section 4.1.4), the indoor/outdoor physical states (temperature, humidity, co2), the initial date and time, the weather (*clear*, *overcast*, *dark*), the room insulation and the windows (if any). The effect of world components on KNX devices and their behavior is explained in Section 4.2.

4.1.4 Program logic

The end goal of our simulator is to test SVSHI apps on simulated KNX systems. As SVSHI runs in real-time for now, the simulator should be able to also evolve in time. It is also interesting for users to see their system change dynamically. To model the environment impacting the system, there should be scheduled updates of physical states (see Section 4.2). These updates occur every *system_dt* (see Section 4.1.3) real seconds, and correspond to *system_dt* $\times$ *speed_factor* simulated seconds. These two variables allow users to control both the real updates time interval (*system_dt*), and the simulated time gap between them. Simultaneously to the scheduled updates, users should be able to interact with the program through the process' command shell or the GUI.

Because of the interactive nature of the simulator and the dynamic evolution of the system in time, an asynchronous implementation is necessary to interpret the user actions & commands while running the backend logic.

When the GUI is not used, the user interactions take place in the command shell (or via a script parser, see Section 6.3). The Python library **asyncio** [6] is used to manage this asynchronous wait for user input. It allows defining tasks and running them in an infinite loop (e.g. `await user_input()`). At the same time, the program can schedule the world updates with `AsyncIOScheduler` function from **apscheduler** library. Our choice is based on the performances of those libraries, and the important support from the python community. Also, other possible libraries such as multi-threading or multi-processing improve CPU-intensive programs (e.g. ML model training), but not the interactive IO-intensive simulator program, that is limited by the time wasted waiting for the user input.

With the GUI running, the graphical library **pyglet** manages the updates scheduling and user interactions (see Section 6.2). This library is event-based and work by listening to keyboard, mouse or screen actions (e.g. mouse pressed, key released, ...). Specific functions are called when an event is detected, and some code updates the graphical window.

As mentioned in previous sections, the simulator components are defined as class instances, that interact between each other with specific methods to emulate the behavior of a KNX system. An Observer Design Pattern is implemented between the KNX bus and the devices to model the internal behavior of the system and transmit telegrams between devices. Scheduled updates simulate the evolution of the surrounding environment states (e.g. Temperature). As explain in Scetion 5.2.2, when using SVSHI program to run applications, an additional thread is run in background to exchange telegrams with SVSHI.

4.2 Simulating the physical world

To expand functionalities of KNX Virtual (and most of existing KNX simulators), we implemented a modeling of the physical world states' evolution in time. This is managed by the object **World** that contains the objects **Time**, **AmbientTemperature**, **AmbientLight**, **AmbientHumidity**, **AmbientCO2**, **SoilMoisture** and **Presence**. The last two are specific for sensors of humidity in soil (for plants), and presence sensor. Their state changes only after a particular user action. The **Time** instance manages the scheduling of world updates, and of the simulation time and date.

The others are the most important ones and their state changes almost at each world updates. They are executed with the function `World.update()` that is scheduled every *system_dt* seconds. The logic of these states' evolution is based on a few online references, personal physics knowledge and intuition, but it can be in no case considered as a perfect modeling of a real world environment. There are in reality too many factors and unknown, and the understanding of physical states dynamics is not the purpose of this project. For this reason, technical explanations remains general in this report, without many details because it is half arbitrary. However, our simple models are somewhat realistic and give a real world effect in the simulation. They also allow a user to test applications depending on external factors.

4.2.1 Temperature

Both the indoor and outdoor temperature are considered in Celsius degrees. The outdoor temperature can be set by the user in the JSON configuration file, or by an API command in script mode (see Section 6.3). We consider that it won't change during the simulation (except in Script mode).

The indoor temperature evolution in time depends on:

- Room's insulation (*perfect*, *good*, *average* or *bad*)
- Outdoor temperature
- Temperature actuators' state (heater and ac can respectively rise and lower room's temperature)

The insulation level is mapped to an arbitrary factor (0, 0.1, 0.2 or 0.4), used to compute the update temperature delta (> 0 or < 0). Without any temperature actuator ON, indoor temperature tends to outdoor temperature value. In the other case, the temperature is first updated with the effective power of the actuator, and then updated with outdoor and insulation influence.

4.2.2 Brightness

Both the indoor and outdoor brightness are considered. Outdoor light depends on the time of day (sunrise, sunset, midday, twilight, ...), and on the weather (*clear*, *overcast* or *dark*). Outdoor brightness value in lux is deduced from outdoor conditions [7], and its influence on indoor brightness depends on the room's windows location and size/area. Indoor brightness is also influenced by light actuators. Users can set outdoor weather using the JSON configuration file, or using script API (see Section 6.3).

Light intensity can be measured in several units, but we only consider *lux* and *lumen*. Lumen is the *luminous flux* and measures the total quantity of visible light emitted by a light source per unit of time [8]. On the other side, lux measure the lumen per square meter (area unit). Brightness sensors measure light intensity in lux, and light source's intensity are defined in lumens. By considering the fact that light intensity decrease exponentially with the distance from source, the light received at a sensor location can be precisely computed, depending on light sources locations (light actuators and windows), and their light intensity (*effective_lumen* of light actuators, weather and time of day for windows). It is also possible to compute the global room's brightness, by considering the light reaching room's floor [7]. This computation considers the beam angle of sources, and room specific factors [9].

4.2.3 Humidity & CO2

Both indoor and outdoor humidity and co2 levels are considered, humidity in percentage, co2 in ppm. Users can set outdoor CO2 level and humidity in JSON configuration file, or using script API (see Section 6.3). Outdoor levels are considered stable during simulation, except in Script mode. CO2 indoor level tends to outdoor level, with the room's insulation impacting the speed of evolution. No device can act on indoor levels yet (HVAC systems are not implemented in the simulator because of the complex feedback loop to control air states, and windows remain closed).

Regarding humidity, to get closer to reality, we use the indoor and outdoor temperature levels to compute indoor and outdoor saturation and actual vapor pressure of water [10]. They respectively represent the maximum proportion possible of water particles in air at a certain temperature, and the actual proportion of water particles in air. The ratio of them is a percentage corresponding to the relative humidity (simply called humidity in this report). To model the evolution of humidity, we consider the indoor and outdoor temperatures to compute the vapor pressures,

we update the indoor vapor pressure with the outdoor one and the room insulation, and the relative humidity is computed as the ratio:

$$\frac{100 \times vapor\_pressure\_of\_water}{saturation\_vapor\_pressure\_of\_water}$$

5 Simulation of SVSHI applications

We now detail the integration of SVSHI apps to control the simulated system.

5.1 Compliance with SVSHI

As of the current state of SVSHI, the supported devices are:

- Binary sensors
- Temperature sensors
- CO2 sensors
- Humidity sensors
- Switch actuators

To couple SVSHI program and the simulator, all these devices are implemented and supported by our simulator too. Additional devices that SVSHI does not support are also implemented in the program, such as a heater, a dimmer button, a LED,... They all can be represented by one of SVSHI-supported device types. For instance, the SVSHI switch can be linked to the simulator heater, led or AC, the SVSHI binary sensor can be linked to the simulator button or dimmer button (thus losing its dimming functionality). As mentioned in Section 4.1.2, those simulated devices are in fact representing communication objects of real KNX certified devices. SVSHI considers also devices as communication objects, and this shows that the simulator is compliant with SVSHI.

5.2 Communication with SVSHI

For its communication with a real KNX system, SVSHI uses an external Python library: **xknx**[11]. The advantage of this library is its simplicity and well-defined representation of KNX objects (e.g. GroupAddress, Telegram, ...). To interact with SVSHI, modules of this library must be used in our simulator, and a new device that acts as the communication module for the simulator must be added to the system. This device corresponds to the **KNX IP Interface** device mentioned in previous sections. In the simulator, the IP Interface contains a module that gathers communication sockets (send and receive) and functions used for sending and receiving information through the IP network (locally) between the simulator and SVSHI.

The communication protocol is divided in two main parts: setting up the connection (similarly to a TCP handshake) and transmitting information (similar to a TCP Tunneling).

5.2.1 Connection

Before being able to handle information exchange, i.e. telegram transmission between our simulator and SVSHI, a *tunnel connection* with SVSHI must be established.

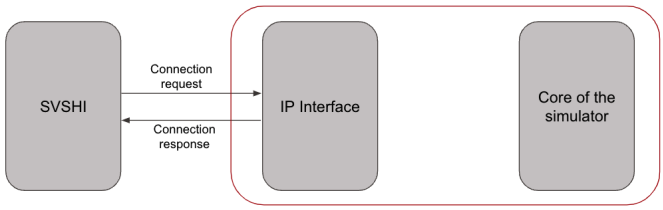

Figure 2: Connection set up with SVSHI

In the figure above, we can observe the different modules involved in the connection: the red box represents the whole simulator which contains the communication interface, as well as the core of the simulation (room, devices, world representation...) The aim is to make SVSHI think that the simulator is a real system with physical devices. A deep analysis of exchanged packet on a real network allowed to create messages that enable this imposture:

1. receive a connection request from SVSHI,
2. reply to the connection request with a connection response,
3. receive the ACK from SVSHI to confirm that the connection was established.

This procedure is done twice, because SVSHI starts a first connection to read the states of the devices and then starts a second connection to launch its applications.

5.2.2 Transmission

Once the tunneled connection is set up, the exchange of telegrams with SVSHI can start! To this end, we decided to start a new background thread, with the **threading** [12] library for Python, that would constantly listen for incoming packets from SVSHI and process packets to be sent from the simulator. This implementation allows the simulator to remain fully independent of SVSHI and respects the isolation of the simulated KNX system from the network. It also lets the simulation be non-blocking for the receiving of SVSHI telegrams. The data structure used to store the telegrams that need to be sent to SVSHI is a `Queue` which is thread-safe in the Python language.

Even though the IP Interface handles the transmission of telegrams, it is not enough because we need to *translate* our simulated telegrams to real KNX telegrams (and oppositely). Indeed, there is no real use to simulate perfectly the XKNX telegrams, and the current implementation is very clear for developer. To answer this problem, a parser module is added to the IP Interface class that parses telegrams in the two ways to send the correctly encoded packets to SVSHI or the simulator, see Figure 3.

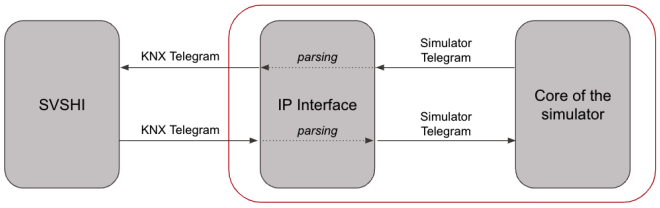

Figure 3: Telegram parsing and exchange

For coherence and correct functionality, users need to set the group address of the devices according to the generated ones by SVSHI (assignments file). This can easily be done through the configuration files or on the Graphical User Interface after the simulation started.

Finally, the core of the simulator is able to send telegrams through the IP Interface as it is considered as an actuator that is automatically assigned to every group address assigned to at least one device. Thus, it receives all telegrams transmitted on the bus and can easily forward them to the communication module, which then parses them and sends them to SVSHI. When it comes to receiving telegrams from SVSHI, the IP Interface module has direct access to the object representing the KNX Bus, so it can directly call the function that transmits telegrams on the bus `knxbustransmit_telegrams()` to the corresponding group address after the telegrams have been parsed from *KNX telegrams* to *simulator telegrams*.

6 User Experience

In this section, we explain the front-end of our simulator, the possible user interactions and the reasons behind our choices.

6.1 Simulator modes

There are multiple possible modes to launch the simulator:

- **CLI** mode: no visual feedback, users can get information and interact with the system through the command-line shell with a defined set of commands (see README).
- **GUI** mode: visualize the system in a graphical window with which users can interact with, the CLI commands are still accessible by the user through a GUI command box.
- **Script** mode: no visual feedback, no dynamic user interaction, a set of API commands is parsed from a .txt file and executed in sequence by the simulator.
- **SVSHI** mode: all the previous modes can be used when running SVSHI applications. However, some delay can occur due to the exchange of telegrams to and from SVSHI program, but there should be no loss of packets (telegrams). SVSHI must be run in a separated process, there is no automated launching of SVSHI program along with the simulator' process.

6.2 Graphical User Interface

The best way to understand what is happening during a KNX system simulation is to visualize it. With the simulator, we provide a Graphical User Interface, which you can see an example of in Figure 4. It is important to note that the GUI is coded around the KNX simulation, and does not interfere with it. Telegrams are exchanged as in a real KNX system and the GUI simply represent the system state. It is thus possible to run SVSHI applications in GUI mode.

6.2.1 pyglet library to represent the GUI

The GUI is based on **pyglet** library. In order to let the possibility of improving the simulator to visualize rooms in fancy 3D representations similar to the *Sim's* game, a library suited for games programming is necessary. It must provide a lot of flexibility and a large range of choices for visual features. Basic GUI libraries such as pyQT or Tkinter are very simple and do not expand well for our project. Our choice was motivated by mainly three reasons: performance, visual possibilities and code simplicity. Pygame was considered, but despite the great community support, pyglet's simplicity and render are more suited for our project [13]. It is one of the most powerful graphical Python libraries, and its object-oriented implementation makes it even better for the simulator. Furthermore, **pyglet** can implement a scheduler to regularly update the physical world states through the method `World.update()`. This allows the system to be fully implement using a single library for asynchronous actions and interactions when using the GUI.

6.2.2 GUI features

For a complete explanation of our GUI functionalities, we redirect interested readers to the README of our GitHub repository. Here is a presentation of main features.

Figure 4: GUI Window example

The first aim of the GUI is to provide a visual feedback of the simulator's behavior. The GUI displays the current configuration with devices' (and windows) locations.

Users can activate Functional Modules, and see the Actuators connected to the same group address having their state change (e.g. on Figure 4, the button2 turned the heater1 ON). Sensors values are automatically updated at each world update and displayed on the side of the GUI window. Some sensors' state are displayed directly on the device representation (e.g. the presencesensor1 and humiditysoil1 on Figure 4). Room devices are also displayed with their individual and group addresses on the side of the GUI, to give feedback on the current configuration. Finally, the simulation time, date, weather and outside physical states are shown to users.

The other most useful functionality is the ability to configure a system using the GUI. Available devices on the side can be dragged and dropped on the precise location wanted by users. It is then possible to assign a group address to it by using the command box on the top right of the window (see the README for more details). By pressing the SAVE button, users can generate a JSON configuration file based on the initial configuration file, with new devices and their group addresses added to it.

6.3 API for automated scripts

One of the most interesting features of our simulator is the possibility to execute automated scripts to test the good functioning of the system created. The program can take a .txt script file containing well-define API commands, and parse it to executes the commands sequentially. For now, scripts cannot be run with the GUI visualization, and users cannot interact with the system in script mode.

Commands can be:

- *wait* for a certain amount of time (real time or simulated time),
- *set* a certain device's state or value,
- *store* one system value if a variable,
- *assert* a comparison between two stored variables, values, attributes or states,
- *show* displays a certain system value
- *end* terminates the script execution.

We redirect interested readers to the README of our project for more details.

The program terminates when all commands are executed successfully, when an *end* command is reached, or if one *assert* command failed. A summary of all stored variables and assertion results is provided to users when the script stops.

7 Future work

In this section, we discuss potential future work and improvement possibilities for this project.

First, concerning the back-end of the project, there are a few potential improvements:

- Handling multiple rooms: For the moment, users can configure a single room and its environment. It could be interesting to be able to have different rooms at once and make their devices interact together as well.
- Devices: Even though the simulator presented supports more devices than KNX Virtual, we could think about adding more devices to be used for new functionalities.
- Physical World modelling: To be even more truthful to our real world, the simulator could fetch information from real weather forecast and integrate it in its calculations for brightness, humidity or temperature levels.

Then, concerning the front-end for users, we could think of transforming our 2D representation of the room to a 3D representation. This could give the opportunity to users to visualize better the model they created and potentially implement multiple 3D rooms.

8 Conclusion

We developed and created a solution for developers and users to test their KNX systems' functionality and configurations without the need to invest time and money in real devices. They are now able to observe the behavior of their devices and track their impact on the surrounding environment, and vice-versa. This solution can be used as a tool, for instance for a SVSHI user who wants to test their apps, or as a main development interface for their KNX system. It could also be linked to a real KNX system and interact with it as an IP Interface device.

Here is a link to the code: `https://github.com/dslab-epfl/svshi_private/tree/simulator`.

References

[1] "Knx association." https://www.knx.org/knx-en/for-professionals/.

[2] S. Chassot and A. Veneziano, "Svshi: Secure and verified smart home infrastructure," , EPFL - DSLAB, 2022.

[3] "Ets software." https://www.knx.org/knx-en/for-professionals/software/ets-professional/.

[4] "Knx virtual." https://www.knx.org/knx-en/for-professionals/get-started/knx-virtual/.

[5] "Knx basics." https://www.knx.org/wAssets/docs/downloads/Marketing/Flyers/KNX-Basics/KNX-Basics_en.pdf.

[6] "asyncio library." https://realpython.com/async-io-python/#the-10000-foot-view-of-async-io.

[7] "Outdoor light lux levels and indoor light computation." https://www.engineeringtoolbox.com/light-level-rooms-d_708.html.

[8] "Lumen wiki." https://en.wikipedia.org/wiki/Lumen.

[9] "Room factors for global brightness computation." https://www.fuzionlighting.com.au/technical/room-index.

[10] "Formula for saturation vapor pressure of water." https://journals.ametsoc.org/view/journals/apme/57/6/jamc-d-17-0334.1.xml#:~:text=New%20formulas%20for%20saturation%20vapor%20pressure%20of%20water%20and%20ice.

[11] "Xknx library." https://xknx.io/.

[12] "Threading library." https://docs.python.org/3/library/threading.html.

[13] "Comparison pygame-pyglet." https://www.pythonpool.com/pyglet-vs-pygame/#:~:text=Pygame%20uses%20SDL%20libraries%20and,to%20subclass%20to%20do%20anything.&text=It%20has%203D%20support.,create%20a%20simple%202D%20game.