Kamil SKARŻYŃSKI[1],
Waldemar BARTYNA[1],
Marcin STĘPNIAK[1]

[1] Siedlce University of Natural Sciences and Humanities
Faculty of Exact and Natural Sciences
Institute of Computer Science
ul. 3 Maja 54, 08-110 Siedlce, Poland

Human integration in an ontology-based IoT system

DOI: 10.34739/si.2020.24.04

Abstract. The IoT systems are growing field of automation. In contrast to industrial applications, where the system is custom made for each customer or use case, the home IoT systems can be composed and used in many, sometimes dangerous and unpredictable, ways. This paper presents a system that is based on a common ontology as a unified and universal method of representing the environment including humans. Such approach allows for easy integration of heterogenous devices and declarative definition of services, tasks, and rules ensuring human safety and/or comfort.

Keywords. IoT, Ontology, Multi-robot systems, Smart Environment

1. Introduction

In recent years, we have observed a continuous increase of interest in IoT devices used in houses or single-family apartments. According to Gartner, each apartment will have over 500 smart devices by the end of 2022 [16]. Time has verified that this number is exaggerated, but we still see a large increase in the number of such devices in homes. Each year more devices are becoming "smart", thus joining the IoT environment, and the term "smart living" is becoming popular. Automation using such devices makes life easier, but it carries dangers that are not always immediately visible. Even if the device manufacturer tries to ensure a certain

level of safety for the family of its products, it is not able to predict the environment in which the devices will be used. Thanks to the growing competition of companies such as IKEA [9], Samsung [5], Fibaro [13], Xiaomi [14] and others, the prices of devices are becoming more and more attractive, and thus more easily available. Each of these manufacturers offers its central unit (gateway) and software solutions, often not compatible with devices produced by other companies. Software solutions that allow the integration of various gateways, such as IFTTT [2], Node-RED [11], Home Assistant [12] and others, have been available for a long time. This makes it possible to provide increasingly complex functions that a single device would not be able to provide. During the integration of such heterogeneous systems, we may encounter security problems related to external attacks, starting from the application layer, and ending with direct attacks on devices [15]. There are also possible conflicts in the IoT environment, causing a threat due to conflicting tasks assigned automatically, e.g. too high level of CO_2 detected by the sensor causes an automatic reaction of the system. In this case, the ventilation system is turned on or the windows are opened. At the same time, however, a second home temperature monitoring task may try to close windows or turn off the ventilation system to maintain an optimal temperature. In this case, an action conflict [1, 4] appears, which may potentially affect human safety. In most of such systems, the human condition is not monitored or considered on an ongoing basis, which should be crucial to ensure maximum comfort and safety. In the proposed system, monitoring is carried out by means of computer vision (CV) by recognizing the human figure [17] and then identifying the person based on the face [18, 19, 20].

This introduces the possibility of tracking the current position of a person in the space of a smart home and allows definition of events and corresponding actions to ensure his safety and comfort. As in the case of the CO2 example, the conflict of action itself is not as important, as ensuring the safety of a person in a potentially dangerous environment. Based on common ontology in the Autero system [8], services and tasks are described in a declarative manner, and each task is subject to the planning and arrangement process, which allows to assess the result of the service before its execution, and thus to determine whether its performance would violate the defined safety (comfort) rules.

2. System Architecture

Figure 1 shows the architecture of the Autero system designed according to the SOA paradigm. The system components communicate with each other by generic protocols. Repository stores ontology and provides access to object maps for the other system components. It has also a graphical user interface (GUI) for developing the ontology, and for its management.

Task Manager (TM for short) represents a client and provides a GUI for the client to define tasks, and to monitor their realization. The tasks can also be sent by other system components, i.e. Service Manager or Safeguards. Planner provides abstract plans for TM, that are used to construct a concrete plan (workflow) based on information on available services registered in Service Registry. The workflow is constructed by arranging concrete services. Arrangement is realized by TM (via the Arrangement Module) by sending requests to services (in the form of intentions) and collecting answers as quotes (commitments).

TM controls the plan realization by communicating with the arranged services. Service Registry stores information about services currently available in the system. Each service must be registered in Service Registry via Service Manager (SM). In this case, SM controls the execution of subtasks delegated by TM and reports the success or failures to TM.

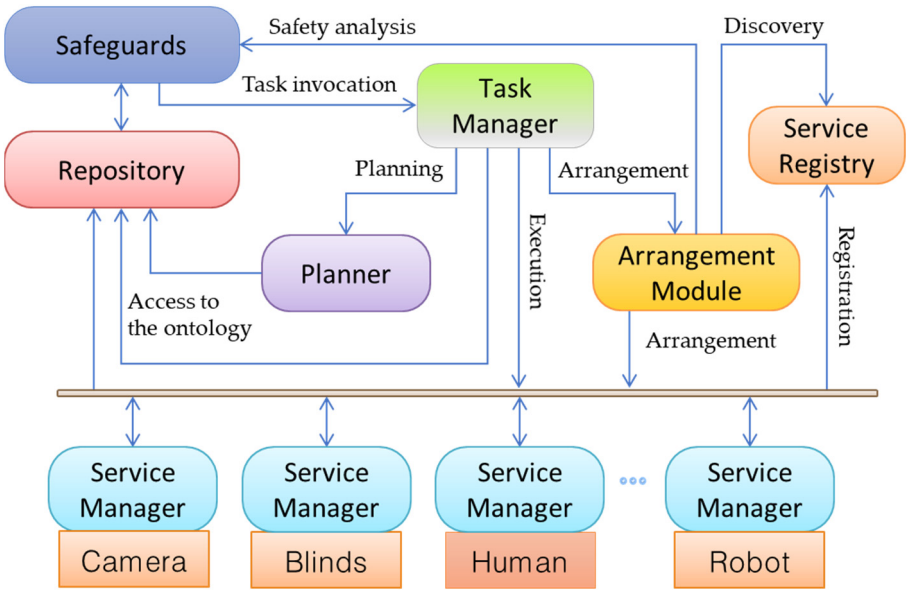

Figure 1. The system architecture. Source: own study.

Safeguards module is proposed in this work as a new addition to the system. It allows for defining of the ontology-based rules. Validation of the rules is performed in two cases:

- during the task arrangement, to ensure the safety of intended task by checking the expected state of the environment after task execution,
- when the object map changes. If an event is triggered, Safeguards acts by sending a task request to TM to change dangerous environment state, if possible.

Task is defined, based on the ontology stored in Repository, as a logical formula that describes the initial situation (optionally) and the required final situation in the environment. For a given task, Planner returns abstract plans that, when arranged and executed, may realize the final situation specified by the given task. More details can be found in [8].

3. Ontology and object maps

The ontology according to Thomas Gruber [7] is: "a formal, explicit specification of a common conceptualization". In the proposed system, it was necessary to create unified and universal method of representing all entities used for describing the environment and the existing or desired situations. Each entity can be treated differently depending on the user perspective. For example, from the point of view of an ordinary user, a car is designed to transport him/her from point A to point B, which is why it is treated only as a means of transportation and the knowledge about such elements as a radiator or pistons is not necessary. On the other hand, from the perspective of a car mechanic, all these elements are important in his daily work when e.g. looking for a fault.

To model the representation of the environment for the Autero system, it was essential to define the description only of a set of selected entities and their properties. A few versions of the ontology for a multi-robot system were created for different iteration of the Autero system. The ontology created for this work considers the aspect of human safety/comfort in the IoT environment. For the purpose of experiments, we also transitioned from previously used simulation environment to the real environment with physical devices (sensors and effectors). Ontology and object maps are stored in a relational database and can be exported in any format including OWL.

3.1. Basic concepts

The ontology consists of a set of definitions of the following concepts:

- attributes – define type properties, e.g., colour, shape, position,
- types – a set of attribute definitions, attribute value constraints specific to objects of a given type,
- constraints – they define the ranges of possible attribute values for a given type and the relations between its attributes,
- relations – define dependencies between objects,
- objects – type instances with specified attribute values and defined relations between sub-objects.

To describe a fragment of the environment, we need to define objects that represent different things in our surroundings. The common structure for similar objects (of the same class) is specified as a *type*. Based on a given type its instance can be created by assigning values to its properties (attributes). A collection of objects and their relations constitute an object map.

A definition of a type consists of:

- Type name,
- Parent type,
- Sub-objects (walls in the room, etc.),
- Attributes (position, rotation),
- Constraints:
    - acceptable attribute values for a given type,
    - required relations between sub-objects of a given type.

When defining a type, we can specify the parent type. All members of the parent type automatically become members of the type being defined (as in inheritance in object-oriented programming).

For example, we will define all the necessary concepts required to model a room. To do this, we define its sub-objects (walls) first, and then create the type representing the room. To begin with, we will define the type *PhysicalObject*; it makes creating types that have their own physical representation (e.g. a wall) much easier.

The *PhysicalObject* type definition includes:

1. The sub-object of type Shape,
2. The Movable, Weight and Texture attributes,
3. Position complex attribute consisting of:
    a. *PositionX,*
    b. *PositionY,*
    c. *PositionZ,*
4. Rotation complex attribute consisting of:
    a. R*otationX,*
    b. R*otationY*,
    c. *RotationZ.*

After we define a base type that can be extended by all physical types, we can proceed to the definition of the type representing the wall shape. The shape hierarchy is defined in the ontology. The shape of the wall expands *CuboidConstructionShape*, which, in turn, expands *CuboidShape*. The *CuboidShape* type has *Width*, *Height*, and *Length* attributes, which describe the dimensions of the cuboid; *CuboidWallShape* inherits these attributes. In this case, it would be possible to describe the dimensions of the wall using the *CuboidShape* type, but because of the possibility of defining constraints (more in the Constraints section) a new type was created to represent the shape of the wall. The constraints specified in the type definition will allow for

the classification of recognized objects. In most cases, objects will be recognized based on the external appearance, that is, their shape (**Figure 2**).

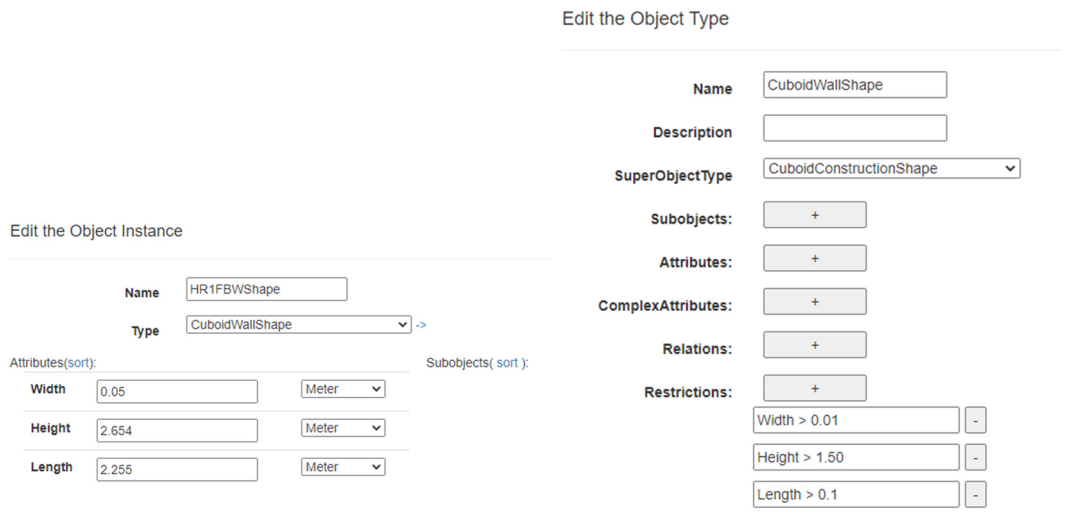

Figure 2. Object of the CuboidWallShape type and CuboidConstructionShape type creation forms. Source: own study.

In the next step, we define a type describing a wall. In the following **Figure 3** the wall type edition and a portion of the type tree are shown.

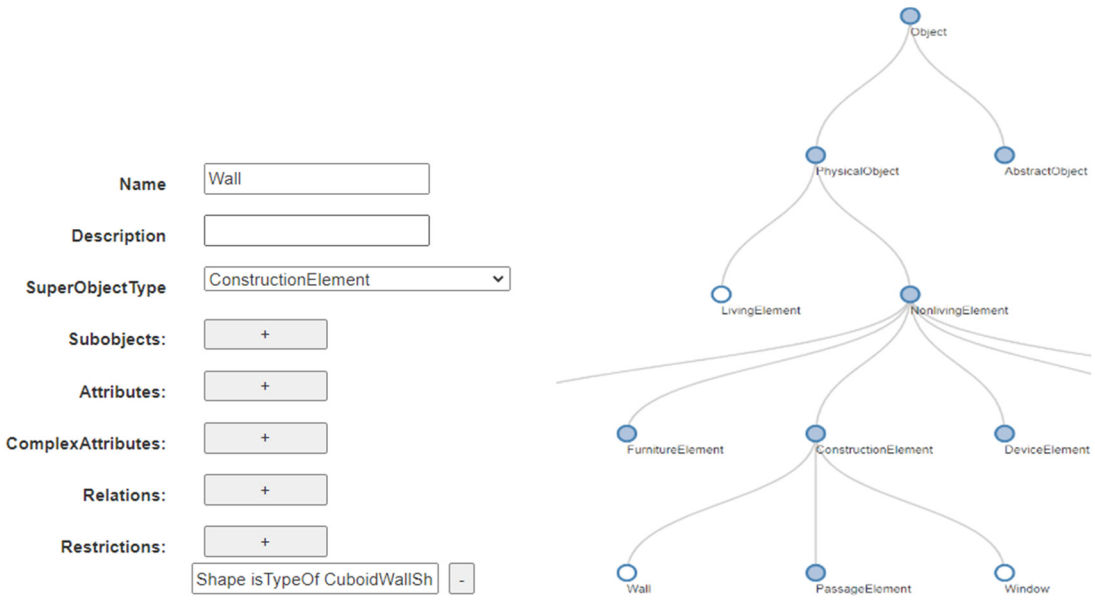

Figure 3. Wall type definition and types tree. Source: own study.

Additionally, the type definition specifies a constraint for the *Shape* sub-object (it is inherited from the *PhysicalObject* type) in order to limit the types of objects that can be assigned as the shape when creating a new *Wall* object. After we have defined all the types necessary to describe the wall, we can define an instance of this type. When defining an object, we select its

type, provide values for all required attributes, and assign sub-objects. The attribute values and relations between the specified sub-objects must not violate the type constraints. To create a wall, we need an object that represents the shape of that wall. In the case of standard rooms, it is often the case that the opposite walls, e.g. ceiling and floor, are of the same size, so we can use the same object representing that shape.

After creating objects for the remaining walls of the room, we need to create another type, i.e. *CuboidRoom,* that allows for grouping of the defined walls. Abstract types are used to group physical objects. The type *CuboidRoom* extends the type *CuboidClosedSpace* from which sub-objects such as *Floor*, *Ceiling*, *Wall1*, *Wall2*, *Wall3*, *Wall4* are inherited. An additional sub-object of type Passage was defined within the *CuboidRoom.* It represents the point of transition between two rooms. The process of defining the appropriate type, shape type, and type instances is not presented because it is similar to the process of creating a wall object. On the **Figure 4** we can see the form for creating an instance of a new *CuboidRoom* object named *LivingRoom01*. The result of the visualization of such a room is shown in **Figure 5** (the ceiling and one of the walls were hidden so they would not cover up most of the scene).

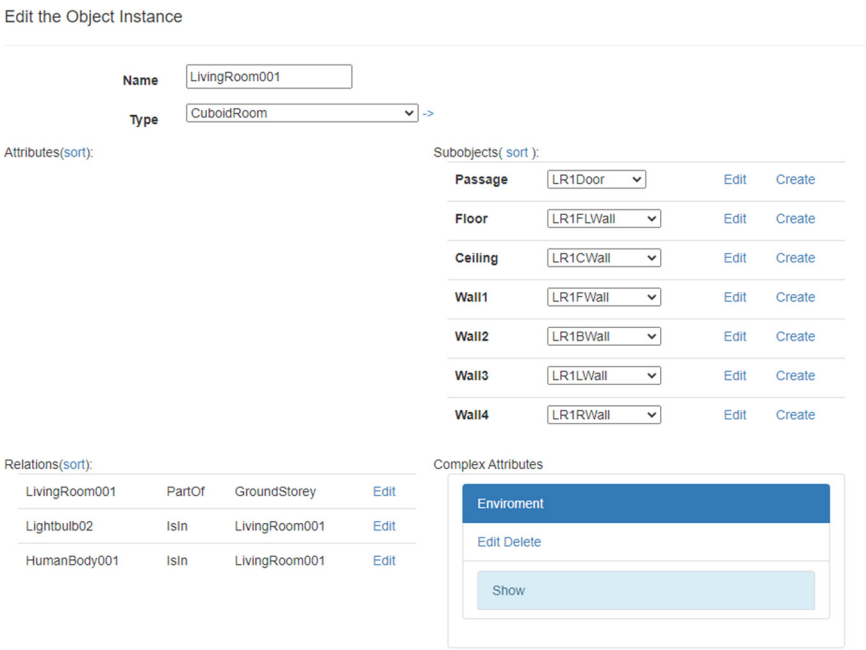

Figure 4. The CuboidRoom type and object definition . Source: own study.

3.2. Relations

In everyday life, people often use phrases such as "turn off the light in the kitchen" or "close the blinds in the living room", which allows us to determine the range of devices required to accomplish these tasks. While everything is understandable when communicating it to another person, based on context and prior information, for most systems the user must precisely define

which device he/she wants to turn on or off. One of the solutions could be grouping of devices of similar purpose and issuing commands to a group instead of to a particular device. The situation is much more complex when dealing with entities that can move within, for example, a house. To solve that problem a concept of relations was introduced.

To allow a more intuitive description of tasks and situations in the environment, we use relation types to determine associations between specific types of objects. The following basic relation types are defined in the ontology:

- *IsIn* – a relation that specifies the inclusion of an object in another object, e.g. a man is in a room,
- *IsFixedTo* – the object is attached to another object, e.g. a cabinet is attached to a wall,
- *IsIntegralPartOf* – an object is an integral part of another object e.g. a doorframe is integral part of the wall which it is placed in,
- *IsPartOf* – an object is an optional sub-object of another object e.g. a storey is part of a building. This allows us to group objects and add connections between them to determine paths.

To be able to automatically evaluate relations (relations created by the Repository), they must be defined based on attributes of specific object types. Repository interface provides a tool supporting the process of defining relation types.

Definition of a relation type consists of:

- Relation name,
- The types of objects between which a relation can occur,
- Human-friendly description,
- Relation specification – the evaluation formula based on values of the attributes of the related objects,
- Whether the relation is dynamic or static.

If the relation type is defined as static, each of its instances must be defined by the user in the object map. For dynamic relation types, each time an object is updated (using, for example, sensors to check one of its attributes), an evaluation operation is performed to evaluate potential new relations and the need to delete those that are no longer up to date. This is necessary for objects that change their position in the environment outside of system control, e.g. humanss.

Figure 5 shows a man in a room. The position and shape of the floor are shown on the right side. When the man changes position, his coordinates are updated and all relation types in which

one of the object types is *HumandBody* are rechecked. In this case, only one relation type is found. Then, all objects with which type *HumanBody* can form a relation, i.e. *CuboidRoom* in our example, will be retrieved. Six rooms were defined for testing, so six instances of rooms will be obtained. In order to assess whether a relation occurs or not, an evaluation of the relation will be made for each of them; if the result is positive – an instance of the relation in the object map will be created, if negative – the existing relation between these objects will be removed (if it exists).

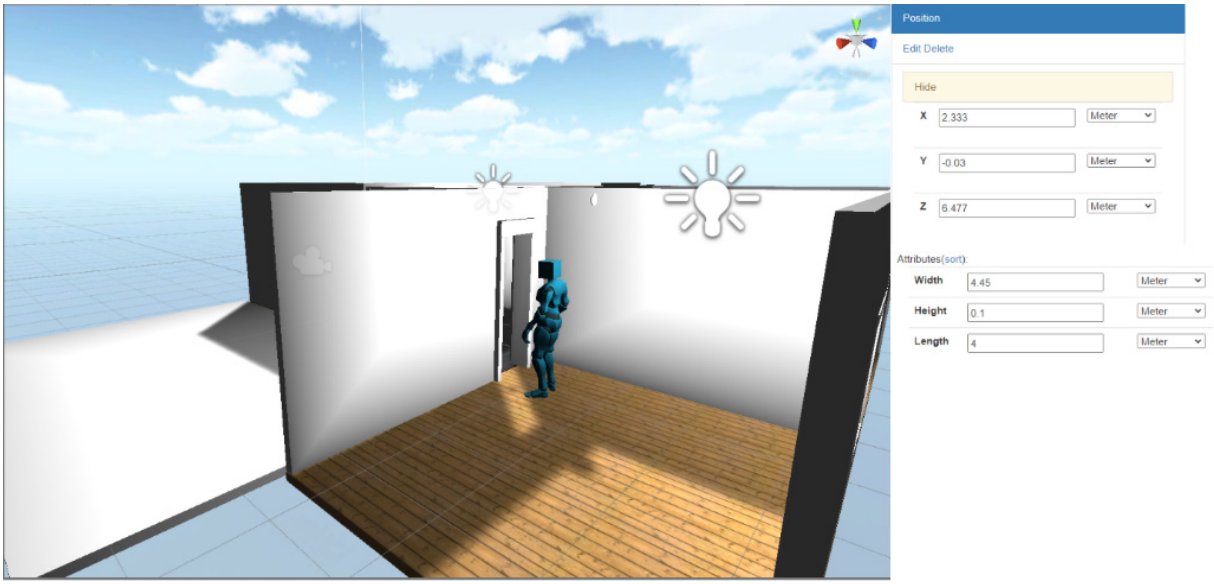

Figure 5. Room visualization with position and dimensions of the floor. Source: own study.

The formula 1 based on the *IsIn* relation will be evaluated for objects of type *HumanBody* and *CuboidRoom*.

$$\begin{aligned} leftObject.X < & \left(rightObject.Floor.X + \frac{rightObject.Floor.Shape.Width}{2} \right) \&\& \; leftObject.X \\ > & \left(rightObject.Floor.X - \frac{rightObject.Floor.Shape.Width}{2} \right) \&\& \; leftObject.Z \\ < & \left(rightObject.Floor.Z + \frac{rightObject.Floor.Shape.Length}{2} \right) \&\& \; leftObject.Z \\ > & \left(rightObject.Floor.Z - \frac{rightObject.Floor.Shape.Length}{2} \right) \end{aligned} \quad (1)$$

where leftObject is of HumanBody type, and rightObject is of CuboidRoom type.

An example of a filled expression for the room with the floor shape: Width: 4.45, Length: 4 and position: X: 2.333, Y: -0.03, Z: 6.477 and the man with position: X: 1.3, Y: 0.03, Z: 5.4 (the situation shown in) will look as follows on formula 2.

$$1.3 < \left(2.333 + \frac{4.45}{2}\right) \&\& \; 1.3 > \left(2.333 - \frac{4.45}{2}\right) \&\& \; 5.4 < \left(6.477 + \frac{4}{2}\right) \&\& \; 5.4 > \left(6.477 - \frac{4}{2}\right) \quad (2)$$

what results in true, thus an instance of the relation between objects *HumanBody01* and *LivingRoom001* is created.

Example of relation evaluation for another room where there is no human (*SmallRoom001*) is presented on formula 3.

$$1.3 < \left(0.19 + \frac{2.78}{2}\right) \&\& \; 1.3 > \left(0.19 - \frac{2.78}{2}\right) \&\& \; 5.4 < \left(-0.27 + \frac{4.485}{2}\right) \&\& \; 5.4 > \left(-0.27 - \frac{4.485}{2}\right) \quad (3)$$

which gives us a false result. No relation will be created between *HumanBody001* and *SmallRoom001,* and a relation of this type between these objects will be removed, if there is one.

3.3. Constraints

The constraints are used for two purposes:

- To specify ranges of allowed values for attributes of a type,
- To specify required/allowed relations between sub-objects of a type.

As mentioned earlier in section devoted to type and object definition, it was necessary to introduce the concept of constraints. It allowed for a clear separation of types that have the same attribute. The distinction can be provided by narrowing the ranges of values of specific attributes in these types.

When defining a relation, it was also necessary to be able to describe the exact dependencies between attributes of the two related objects. Due to the approach used in the ontology, a sub-object named *Shape* of type *Shape* was defined when creating the *PhysicalObject* type. This made creation of new objects more convenient because each new shape type expands the *Shape* type, and each physical object has a *Shape* sub-object. The problem occurs during the relation type definition, where access to the attributes describing the shape of the object is required. Therefore, when defining object types, it is possible to define a constraint for the sub-object.

For example, for the object type *Wall* constraint can be defined as: *Shape IsOfType CuboidWallShape*. This way, when creating new objects of *Wall* type, only shape objects of types inheriting from the one specified in the type constraint will be allowed as *Shape* sub-object.

4. Experiments

The aim of the experiments is to automate the process of managing indoor lights, switch on lights when a person enters any of the rooms and switch off when he/she leaves the room. Three devices were used in the experiment:

1. Wifi Sonoff GK-200MP2-B 1080P camera,
2. TRADFRI LED E27 bulb,
3. TRADFRI Gateway.

The second and third device are available as part of the IoT solutions offered by TRADFRI [9]. Integration with the Autero system was realized with the help of the CSharpTradFriLibrary library [10]. The main difference to the standard solution offered in IoT projects is the use of a camera for human location in 3D space instead of detecting motion using a motion sensor or presence sensor. To implement the scenario in the Autero system, it was necessary to:

1. Define the appropriate types in the ontology, and then define and add objects to the object map,
2. Define the type of service and then add the service that modifies the light intensity,
3. Calibrate the video capture software to define the translation between the position in the captured picture to the position in the object map,
4. Create events and actions in Safeguards module.

The definition of the type representing a room was presented in the Ontology and object maps section. For the scenario, we have defined a *LightBulb* type that represents a light bulb and consists of the basic attributes inherited from *PhysicalObject* and the *LightIntensity* attribute. Based on the type we defined, we can create objects that represent light bulbs in all the rooms. The *LightIntensity* attribute was also added to the *CuboidRoom* type as a part of the *Environment* complex attribute. This attribute symbolizes the computed intensity of the light in the room. We have also defined a *HumanBody* type inheriting from *PhysicalObject.* This type represents a person, his/her position in the object map and his/her identity by adding *PersonFaceId*.

Based on object types, we define a type of services that will be performed on specific physical devices. A declarative description of the service type based on the ontology is required to enable automatic arrangement and execution of appropriate services. For this scenario, we need one type of service that changes the intensity of the light in the room. The definition of precondition is empty and final conditions for a service type is defined in formula 4.

LighBulb<LightBulb>.LightIntensity = LightIntensityValue<Integer> (4)

Since the TRADFRI bulb allows us to control the light intensity, the service changing its state takes one parameter specifying the intensity of light in the form of an integer. Then, based on the type of service, it is possible to create a specific service in the Services Manager. The information about the TRADFRI gateway address, the name of the device (as it was named during the gateway configuration, e.g. "light") and a special token to establish a connection is required in order to properly configure the new service which, from now on, will be provided by the Service Manager.

Correct configuration of the video capture software is necessary so it would be possible to update the position of the human object in the object map. The association between the object map and the recognized person is accomplished by a dedicated P*ersonFaceId* attribute, the value of which is set accordingly when creating a *HumanBody* object, so that the system can monitor multiple people at the same time. When updating information about objects in the object map, existing relations are automatically evaluated by the Repository. This is necessary to represent the current state of the environment and is used when defining events.

The final step is to configure the event based on which the appropriate action will be performed. We can define events in the safeguard module, by formally defining the event and the corresponding action. In this experiment, we detect two events; when a person appears in a room and when a person leaves the room. The rule for the first event is defined in formula 5.

Event: human<HumanBody> IsIn room<CuboidRoom>

Action: room<CuboidRoom>.LightIntensity = 100 (5)

The event definition uses an IsIn relationship, which can be seen in the example in the Relationships subsection. With this record, the user can easily define their own events and actions. This method is similar to applications such as IFTTT[2], but events are not based on information from devices, but on situations in the feature map. At the time of the event, the appropriate action will be performed, which will commission the task to the Task Manager in

order to achieve the desired result, and in the room into which the person entered the light will be turned on. Similarly, we define the rule in formula 6 for extinguishing the light in the room.

Event: !(human<HumanBody> IsIn room<CuboidRoom>)

Action: room<CuboidRoom>.LightIntensity = 0 (6)

We create a negation of the previous event, covering all situations when a person is not in the room and configure the action accordingly to switched off the light. This rule can be used, for example, to save energy.

After setting up the system, the scenario can be realized. There are two ways to visualize the experiment. The first one is through the camera image, after being processed. The second one presents a screen shoot from the simulation environment which visualizes the current state of the object map. The simulation continuously refreshes the positions of objects (in this case, position of the man), so that we can easily monitor the current status of the object map during experiments. At the start of the experiment, the light in the room is switched off and the man is not present (**Figure 6**).

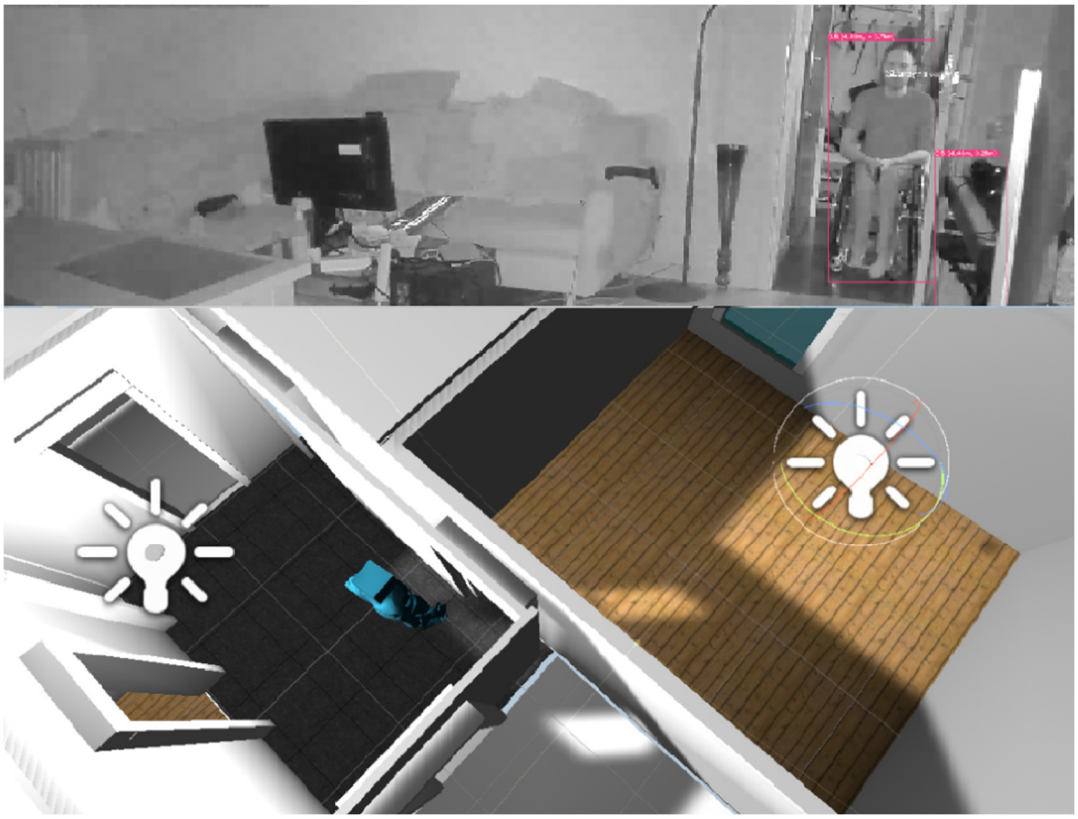

Figure 6. Experiments, corridor CV and visualization. Source: own study.

The person is already visible in the camera image, but the light in the room is still switched off due to the fact that the event describing the presence of a person in the room has still not been triggered. In the above figure (**Figure 6**), we can also see the simulation which shows that in the current state of the object map the person is still in the hallway and not in the room.

After entering the room, the human position in the object map is updated, the IsIn relation is evaluated, and the events defined in the safeguard module are checked. When one or more events are triggered by the change in the object map, the task to switch on the light in the living room is sent to the Task Manager. In Figure 7 we see that the man entered the room, the object map was updated, and the light is switched on. Due to hardware limitations, the whole process takes about 1 to 3 seconds, mostly by the speed of the image processing being done on CPU.

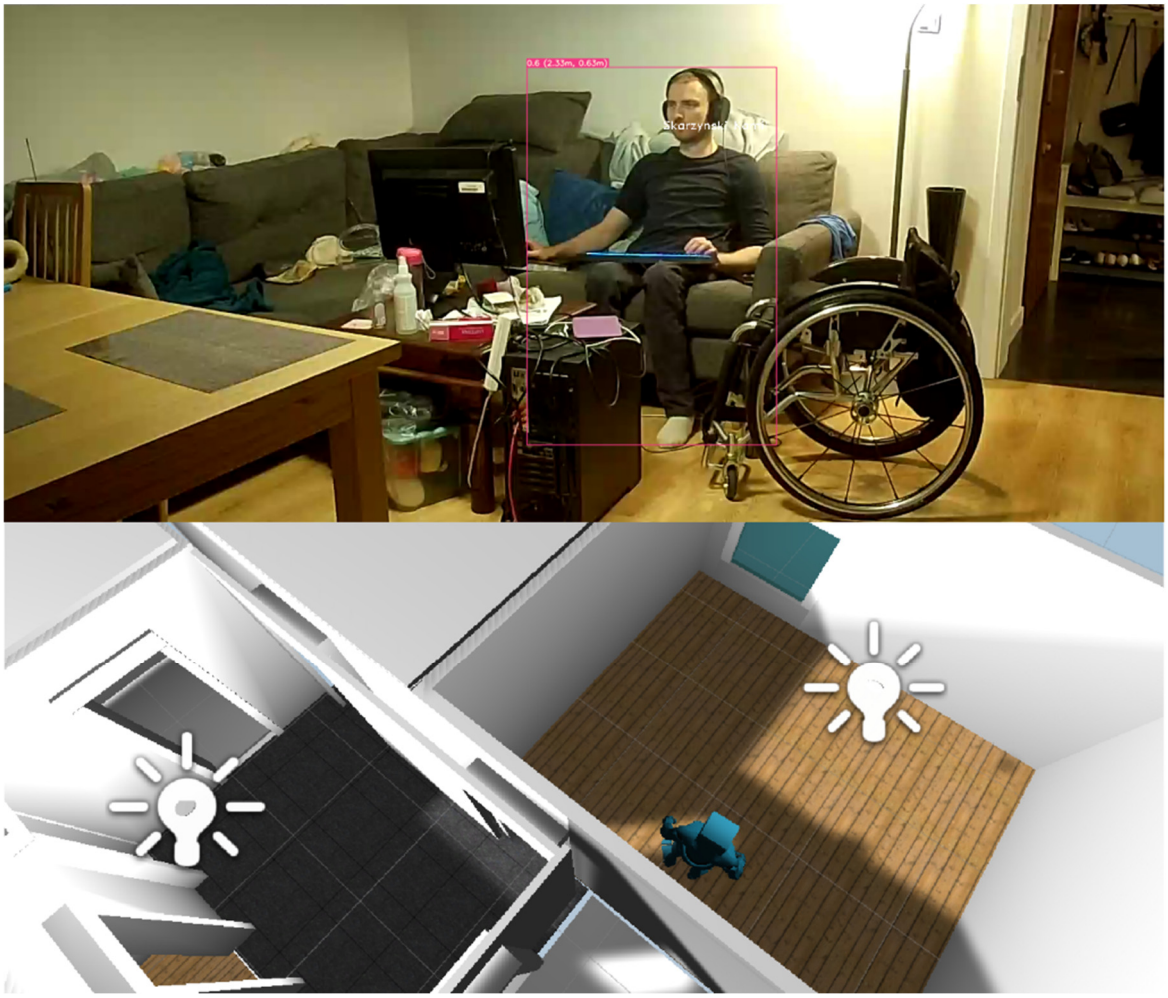

Figure 7. Experiments, room CV and visualization. Source: own study.

5. Conclusion

In IoT environments consisting of, e.g. Samsung SmartThings [5] or IKEA TRADFRI [9], the test scenario can be achieved using a presence sensor or motion sensor. The advantage of the proposed solution is the lack of cases in which the detection of movement outside the room may cause the light to be switched on. Also, when using a presence sensor, the person must carry an additional device. The use of object map and human position in 3D space allows for a much wider range of applications, and the only thing that limits users is the number and types of owned devices.

On the other hand, the human detection module requires further work. The positioning mechanism uses one camera and identifies people by their faces. This allows for unambiguous identification, but the human face must be visible from the position of the camera. In environments where multiple cameras are present, the problem can be solved by implementing the capability to process and merge data from all the cameras in a given room. That, by itself, constitutes a challenge from the speed and efficiency point of view.

Further works will focus on expanding the ontology and range of devices that could provide services in the system. The next scenarios will introduce more complex tasks, new events and actions improving not only human comfort but, more importantly, his/her safety in a smart environment.

References

1. Celik Z. B., Tan G., McDaniel P.: IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In Proceedings Network and Distributed System Security Symposium. Internet Society, San Diego, CA, 2019.

2. IFTTT (if this, then that): Helps your apps and devices work together. (https://ifttt.com/, access date: 01.10.2020).

3. Ferry N., Nguyen P., Song H., Novac P. E., Lavirotte S., Tigli J. Y., Solberg A.: GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems. In IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), pp. 870–875. IEEE, Milwaukee, USA, 2019 (https://ieeexplore.ieee.org/document/8753981/).

4. Liu R., Wang Z., Garcia L., Srivastava M.: RemedioT: Remedial Actions for Internet-of-Things Conflicts. In Proceedings of the 6th ACM International Conference on Systems for

Energy-Efficient Buildings, Cities, and Transportation, pp. 101–110. ACM, New York, USA, 2019.

5. Samsung Smart things, Smart home. Intelligent living. (https://www.smartthings.com/, access date: 01.10.2020).

6. Samsung Smart things. Developer documentation. (http://docs.smartthings.com/en/latest, access date: 01.10.2020).

7. IDEAL (Projet), ScienceDirect (Service en ligne): Knowledge acquisition. London: Academic Press, 1993 (http://www.sciencedirect.com/science/journal/10428143, access date: 01.10.2020).

8. Skarzynski K., Stepniak M., Bartyna W., Ambroszkiewicz S.: SO-MRS: A Multi-robot System Architecture Based on the SOA Paradigm and Ontology. In M. Giuliani, T. Assaf, M. E. Giannaccini (editors), Towards Autonomous Robotic Systems, pp. 330–342. Springer International Publishing, Cham, 2018.

9. TRADFRI - Smart Home Products - Lighting, Wi-Fi Speakers, Blinds. (https://www.ikea.com/us/en/cat/smart-home-hs001/, access date: 01.10.2020).

10. TRADFRI - C# integration. (https://github.com/tomidix/CSharpTradFriLibrary, access date: 01.10.2020).

11. Node-RED. (https://nodered.org/, access date: 01.10.2020).

12. Home Assistant. (https://www.home-assistant.io/, access date: 01.10.2020).

13. FIBARO | Smart Home. (https://www.fibaro.com/pl/, access date: 01.10.2020).

14. Xiaomi Mi Smart Home - category of Xiaomi smart devices and accessories for it. (https://xiaomi-mi.com/mi-smart-home/, access date: 01.10.2020).

15. Hassija V., Chamola V., Saxena V., Jain D., Goyal P., Sikdar B.: A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures. IEEE Access, vol. 7, 2019 (https://ieeexplore.ieee.org/document/8742551).

16. Gartner Special Report. (https://www.gartner.com/en/newsroom/press-releases/2014-09-08-gartner-says-a-typical-family-home-could-contain-more-than-500-smart-devices-by-2022, access date: 01.10.2020)

17. Redmon J., Divvala S., Girshick R., Farhadi A.: You Only Look Once: Unified, Real-Time Object Detection. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779–788. Las Vegas, USA, 2016.

18. Parkhi O. M., Vedaldi A., Zisserman A.: Deep Face Recognition. In Procedings of the British Machine Vision Conference 2015, pp. 41.1–41.12. British Machine Vision Association, Swansea, 2015.

19. Ng H. W., Winkler S.: A data-driven approach to cleaning large face datasets. In IEEE International Conference on Image Processing (ICIP), pp. 343–347. Paris, France, 2014.

20. Face recognition library. (https://github.com/ageitgey/face_recognition, access date: 01.10.2020).