2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

# TDeX: A Description Model for Heterogeneous Smart Devices and GUI Generation

Luciano Baresi[1], Mersedeh Sadeghi[1], and Massimo Valla[2]
[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
[2]TIM S.p.A., Services Innovation Department, Joint Open Lab S-Cube, Milano, Italy
(luciano.baresi@polimi.it—mersedeh.sadeghi@polimi.it—massimo.valla@telecomitalia.it)

**Abstract**

The fast-growing availability of smart devices is making our environments smarter than ever, but also more and more heterogeneous. Many diverse devices must be integrated and this paper proposes a unified description model and a supporting middleware. Besides focusing on the functional capabilities of the different devices, and on the contexts of use, TDeX, which extends the W3C's Thing Description model, has been designed to embed descriptions of the graphical user interfaces (GUIs) needed to interact with them. The paper also proposes a model-driven and device-agnostic approach for the automatic generation of these GUIs. *M4HSD*, our RESTful middleware infrastructure, mediates between the GUIs and the devices themselves and provides a generic and platform-independent solution for the interoperability of Internet/Web of Things systems. Preliminary experiments demonstrate the feasibly of the proposed approach and its innovative characteristics.

## I. Introduction

The Internet of Things (IoT) has been promising a superior quality of life through a global network of sensors, actuators, and smart devices. However, the interconnection of billions of diverse devices [1] and the management of their interactions make the classical interoperability problem even more challenging. One of the possible solutions is the so-called Web of Things (WoT) approach, which targets a complete integration of devices into the web by adopting web technologies [2], and specifically it exploits the REST architectural style [3] to associate a uniform interface with any resource/functionality provided by these devices [4].

The adoption of web protocols and REST principles hides the diversity of communication protocols and significantly enhances interoperability at network level, but the semantic interoperability remains unsolved. REST principles support information exchange, but the structure and semantics of exchanged data cannot be standardized universally. The HTTP protocol allows for content negotiation, but it cannot address the underlying data model. In addition, numerous IoT-related ontologies and data and domain models are often domain-specific and the number of IoT/WoT devices is growing. The result is that their standardization is often jeopardized. Each of the many existing APIs, standards, and specifications comes with its own syntax, vocabulary, semantics and consistency rules. This clearly hampers the integration problem and often precludes the interactions among heterogeneous devices. The use of web technologies as integration layer of the IoT shifts interoperability towards the application layer, where the developer must cope with all the cases of interest and harmonize them properly.

This work has been partially funded by TIM S.p.A., Services Innovation Department, Joint Open Lab S-Cube, Milano.

In this context, the paper does not want to propose yet another description model for different IoT/WoT devices. In contrast, we use and extend an existing description model to develop a middleware infrastructure that supplies the application developer with a single, integrated view of the set of IoT/WoT devices of interest. The middleware is then in charge of coping with the idiosyncrasies to simplify the development of applications that exploit heterogeneous devices. The work extends the *Thing Description* (TD) proposed by the W3C's WoT Interest Group[1] to propose TDeX (TD eXtended) and address some of its limitations. The most notable one refers to the interactions between these devices and their users. By categorizing the interactions, that is, by grouping the different actions that can be performed on these devices properly, TDeX provides the developer with the elements to generate basic graphical user interfaces (GUIs). This additional information is then used by a model-driven solution for the automated creation of the GUIs. TDeX also adds the idea of context, to better scope the different devices, takes the dependencies between the properties of a device and the actions executed on it into account, and defines elements to properly identify the current state of a device. TDeX categorizes properties into static and dynamic ones, to highlight those characteristics that are mainly informative and descriptive, and do not change, and those that represent state changes. We have also followed the suggestion provided by the W3C WoT group and organized actions into five main groups. Given that the different devices are organized around contexts, also the GUI generation process can exploit the grouping and provide user interfaces that *follow* the users or that *evolve* according to their needs and requests.

The whole solution proposed in this paper thus comprises three elements:

- A uniform model, called TDeX (TD eXtended), that contributes a uniform description of a wide variety of devices. The model associates each device with a rich set

[1]https://w3c.github.io/wot/current-practices/wot-practices.html

of contextual and functional information, together with the seeds for organizing and rendering the interaction between users and devices.

- A model-based automated solution for the creation of (basic) GUIs for interacting with the different devices. Obtained GUIs are device- and platform- agnostic and can then be rendered into many diverse concrete solutions (from web-based ones to the layouts of Android activities).
- A RESTful middleware, called *M4HSD*[2], that bridges GUIs and devices. *M4HSD* exploits TDeX to abstract devices through standardized REST APIs and to support the context-aware interaction with them. Special-purpose drivers and plug-ins convert the device-specific APIs and data models into our unified one.

The rest of the paper is organized as follows. Section II presents TDeX, our integrated IoT/WoT model. Section III describes the REST services provided by *M4HSD*. Section IV explains the automated creation of GUIs. Section V discusses an experimental deployment of the framework and a first exemplar application. Section VI surveys related approaches and Section VII concludes the paper.

## II. TDeX

TD (Thing Description) defines the elements for describing the capabilities of an object and for accessing it. An object is described by means of `Properties`, `Actions`, and `Events`. A `Property` refers to the readable/writable data that render the internal state of an object. An `Action` is an operation provided by the object, and an `Event` is used to notify certain conditions (e.g., an occurrence of an action or a specific value of a property).

TDeX can represent any smart object: from sensors to home appliances, including WoT/IoT devices, devices with embedded web servers, and devices that expose a web service through a gateway. In addition, unlike most of the existing efforts for modeling IoT/WoT objects, which only concentrate on the functional capabilities of these objects, TDeX also addresses the context of use and the GUIs to interact with them. Figure 1 shows the core components of TDeX.

Since properties and actions characterize objects, but events mainly define the interests of the different observers, TDeX does not support the concept and offers subscriptions/notifications, from/to external components, to let interested parties be notified about fired actions and changed properties.

`Metadata` specifies general information, such as the name and type of the device and its BaseURL, that is, a RESTful endpoint through which client applications can interact with the device. A `Location` indicates the current location of the object and can provide a means to discover objects, based on their position or on the position of other objects nearby. A `Location` is defined by a name, type (*public* or *private*), and zero or more `near` associations with other `Locations`.

[2]Because of the sponsorship, *M4HSD* and the applications we developed to evaluate it are only available upon request.

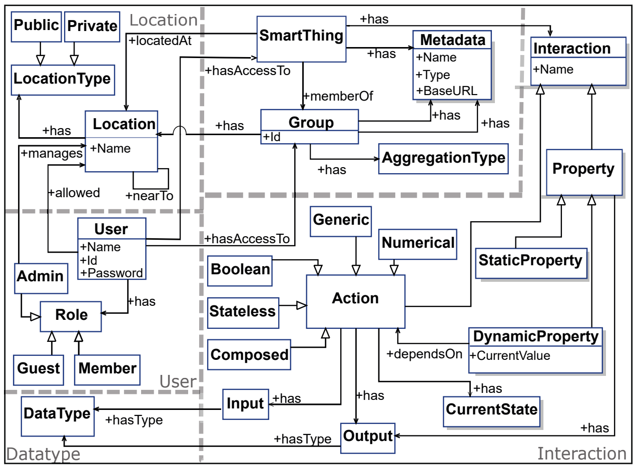


Figure 1: TDeX meta-model.

A `User` identifies a potential utilizer of the object and is specified by means of a name and a role (*guest*, *member*, or *admin*). Users can thus create locations, relate them, and distribute devices properly: different users can consider different descriptions of the same space. The associations between users and locations are based on the following rules:

- **Guests** have only access to public locations, and thus, can only interact with devices in these locations;
- **Members** have access to all public locations and to their private ones, and then to the devices in these locations;
- **Admin** has access to all locations, and then to all devices. It is also the only role that can state that locations are private to some members.

This means that if a device is in one of the *allowedLocation*s of a user, the user *hasAccessTo* to the device.

An `Interaction` requires a `Property` and an `Action`. TDeX specializes TD's `Property` into `DynamicProperty` and `StaticProperty`, where the former identifies a permanent, inherent characteristic of a device. The latter corresponds to values that can change during the object's lifecycle, and capture the internal state of an object (e.g., the water level of a coffee machine). A dynamic property also `dependsOn` actions. For example, property *waterLevel* of a coffee machine depends on action *makeCoffee* —the action modifies the property's value. Another key difference between dynamic and static properties is related to their visualization. A dynamic property requires a binding between a graphical element and the datum it is supposed to render to be able to visualize any change to it. The graphical representation of a static property is only generated once and there is no need to manage changes.

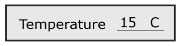
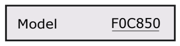
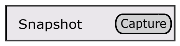
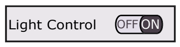
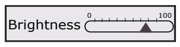
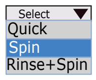
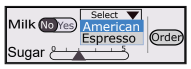
An `Action` captures some functionality provided by a device, and is defined by means of *currentState* and *type*. The type both characterizes the expected behavior and paves the ground to the graphical rendering of the interaction. The GUI element (e.g., radio button, slider, etc.) used to visualize an interaction depends on the nature of the interaction itself.

Although smart devices differ in type, capabilities, and usage, the essence of their interactions falls into a limited set of categories. For example, locking/unlocking a door is "similar" to turning on/off a light. Consequently, as for their GUIs, both actions can be represented via the same element (e.g., a toggle or switch).

To take into account of any possible GUI representation of interactions, Table I summarizes our taxonomy. It currently specializes actions in five different groups by considering how they can change the state of the device. **Stateless** actions can be fired as many times as one wants and they do not depend on any previous state of the device: for example, a security camera can always take pictures. The simplest GUI element to render these actions is a button (to push it). **Boolean** actions assume a boolean state and can only complement it. As said before, this is the case of switching lights or locking doors: when the action is triggered, the opposite value is taken. The GUI element associated with these actions is a switch or toggle. Both **Numerical** and **Generic** actions manage multi-valued states. **Numerical** actions predicate on (limited) sets of numerical values: for example, the set of temperatures offered by a thermostat, the brightness levels of a television, or the sugar quantities of a coffee vending machine. TDeX characterizes these operations by means of the min and max values of the interval of interest, and optionally by a standard increment. The GUI element associated with these actions is a seek bar. **Generic** actions generalize the concept and allow one to decide among the values of general enumerations. This is the case when selecting the working mode of a washing machine or the defrost options of a microwave (e.g., vegetables, meat, fish, and bread). Finally, besides the aforementioned atomic actions, TDeX also supports **Composed** actions, that is, any combination of the previous ones. For example, to instruct a vending machine to pour coffee, one should decide about yes/no milk (boolean action), sugar level (numerical action) and coffee type (generic action): the representation of *makeCoffee* is then the composition of three atomic actions. Note that although the representation is a combination of atomic actions, the associated operation is triggered as a single transaction composed of atomic actions. To achieve this, each **Composed** action has a single atomic action that activates all the others together. For example, a user can select sugar level and type of coffee in the GUI, then press `Order` to collect all the data and send the request to the coffee machine.

This categorization is embedded in our model and allows for the automated creation of the GUI for different smart objects. TDeX also supports Groups of smart devices. A group can be used to control multiple devices of the same type through a single GUI element. Groups can simplify the management of related devices and help organize complex smart environments with a large number of devices. A Group has also a state, which is the aggregation of the state of its members. The aggregation function is a configurable parameter that must be set at the time of group registration. For example, if a user groups several thermostats, she/he could set the aggregation function as the minimum, maximum, or average temperature among those sensed by the members. A device can be a member of more than one Group and a user can only group those devices that are in her/his authorized locations. A Group is discoverable at any location in which it has at least one member device, and by all the users who are authorized to be in that location.

Table I: Interaction types and their corresponding GUI elements.

| Type | Description | Example | Default |
|---|---|---|---|
| Dynamic Property | Property with dynamic value, typically,an internal state of an object. | Current temperature of thermal sensor. |  |
| Static Property | Property with static value, typically the physical characteristics of an object. | Model number of a device. |  |
| Stateless Action | Stateless action. | To take a snapshot in security camera. |  |
| Boolean Action | Action with only two states. | To turn on/off a light. |  |
| Numerical Action | Action with more than two numerically-ordered states. | To set the Brightness of TV screen |  |
| Generic Action | Action with more than two enumeration states. | To select a washing mode in a washing machine. |  |
| Composed Action | Action with any combinations other types. | To order a coffee in a coffee maker. |  |

## III. *M4HSD*

Our middleware framework, called *M4HSD* (Middleware for Heterogeneous Smart Devices), exploits TDeX to smooth the peculiarities of the different devices and allows users to interact with them through suitable instances of TDeX concepts and a homogeneous API. The different characteristics are casted into the unified model and harmonized through RESTful interfaces. The user can exploit dedicated applications (GUIs) to manipulate the model, and then the middleware forwards the commands to the different devices. Dedicated drivers interact with specific hardware bundles to both keep the separation of concerns and allow for the extendibility of the proposed middleware.

Figure 2 presents the high-level architecture of *M4HSD*. The current prototype of the middleware infrastructure is implemented in Java, uses Tomcat and Spring to supply the REST interfaces, and exploits MongoDB as for persistence. As for plug-ins, we currently support Netatmo weather stations, Philips Hue lamps, Garadget door, Zehnder ComfoAir air

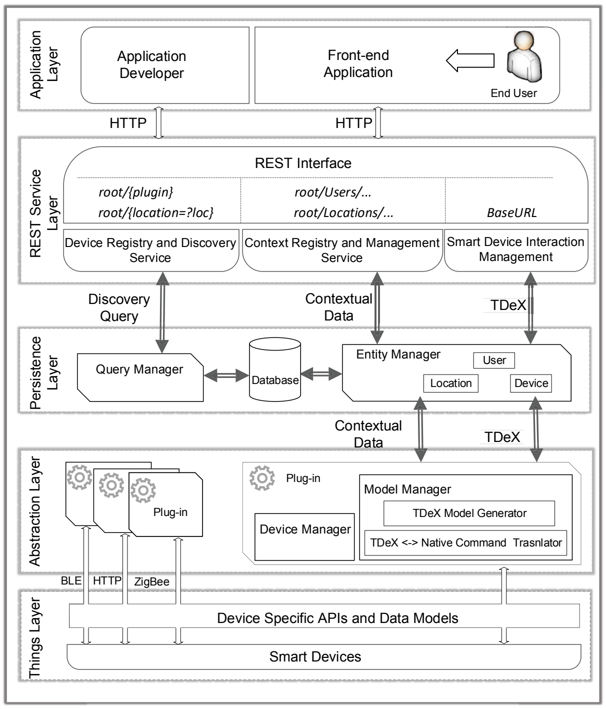


Figure 2: General architecture of *M4HSD*.

conditioning systems, Kasa Smart Plugs, and Wi-Fi motor controllers to control projection screens and window shutters.[3]

The *REST Service layer* supplies all the interfaces required to register and manage devices and to let user applications search for the devices in the context of interest and interact with them. The services provided by *M4HSD* are divided in two groups: those in charge of the domain knowledge, which take care of users and locations, and those in charge of the registration and discovery of devices, and then of the interactions with them.

*POST* requests are used to register new users, locations, and devices. Upon these requests, *M4HSD* generates a new resource and returns the link to it. *GET* requests to the proper URIs are used to retrieve information about the different resources. If the target is a device, provided information is the specific TDeX model, while a *GET* on a location returns the name and type of the location itself along with all the nearby locations. For a user, it returns its name, role, and allowed locations. *PUT* and *DELETE* requests allow one to update and delete resources. User actions are then translated into *PUT* requests onto involved devices.

Dedicated plug-ins and drivers define the *abstraction layer*, that is, those elements in charge of abstracting the peculiarities of the different devices and providing a unified, integrated model. The adopted plug-in-based solution increases the flexibility of *M4HSD* and allows one to add new standards, or devices, by simply developing the required plug-in. Each plug-in contains two parts: *DeviceManager*, which creates and manages the network connections needed to interact with the device, and *ModelManager*, which generates the TDeX model from a native one.

[3]Netatmo weather stations and Philip Hue lights are physically available in our experimental environment, while the other devices have been simulated through special-purpose Java components.

The *persistence layer* takes care of storing the data related to users, locations, and devices. As soon as a new device is registered in *M4HSD*, we generate a new TDeX resource (model) to represent the device and an entity in the registry to keep the associations between the TDeX representation, that is, the BaseURL, and the actual device. The specific TDeX model is generated by a special purpose model-to-model transformation implemented in the associated plug-in. The model represents the current state of the device; *persistence layer* must keep this representation updated with respect to both commands from the upper-level applications and state changes from the devices themselves.

Groups are managed through single BaseURLs, which are associated with more than one device. Accordingly, a command to a Group targets all member devices: if all the members can execute the action successfully, *M4HSD* issues a single positive reply, otherwise it warns the application about problematic devices.

The current location of a device allows *M4HSD* to identify authorized users and to add this information to the device's representation. User roles define the set of locations users can access; a user can only interact with devices in allowed locations (see the policies defined above). This is key to enable device discovery given either a location or a user. If it is location-based, *M4HSD* retrieves all the devices in a specific location the particular user is allowed to interact with. If it is user-centered, then the middleware retrieves all the devices the user can exploit in any location. A user can only perform location-based discovery for allowed locations.

Discovered devices are characterized by their names, locations, and links to their TDeX models. This model is the only resource external applications need to read/write to interact with the device in a seamless and homogeneous way. *M4HSD* is always aware of the current state of any registered device. As soon as a new command is issued, the middleware compares the desired and actual states, identifies the action that should be triggered, translates it into the device's proprietary language, and fires it.

Since multiple users can issue contradicting commands to the same device simultaneously, *M4HSD* does not adopt any predefined policy, but it forwards all commands to interested devices, which must solve them locally. In case of race conditions, *M4HSD* does not override the native behavior of devices and it simply mirrors their strategies. For example, some devices just implement one of the simultaneous requests and drop the others. In this case, *M4HSD* would return an error message to the applications that triggered the command unsuccessfully.

## IV. Graphical User Interfaces

Generated TDeX models are platform-independent models that also contain sufficient information for automatically creating the GUIs for controlling associated devices. As for GUI creation, one must only read the aforementioned description and translate it into the concrete elements of the GUI platform of interest (e.g., Web, Android, or iOS). The GUI renders the properties of devices and the actions users can execute on them. As soon as users execute actions, the GUI is modified accordingly. For example, if one wanted to create an application to control some smart lights, the automatically-generated GUI would render each light through a name, its current state (on/off), and a switch to control it. When the user switches the light, the GUI uses some listeners to change the TDeX model, that is, the device's state, and the middleware issues a command to the specific plug-in (the software component specific to the particular light model) to apply the change onto the physical device. As soon as the plug-in notifies the middleware that the device has completed the action, the updated model is stored. If the device cannot execute the action —e.g., because it is out of order or because there is a communication error— and *M4HSD* infers there is a problem[4], it immediately notifies it to the GUI.

The GUI associated with a device comprises the rendering of two parts: its state and the commands/actions the user can issue to control it. The first part starts from visualizing the properties that define a device. Each property has always a name and a value. While names, which are strings, can easily be rendered by means of textual labels, the visualization of values is more complex and depends on their types. TDeX and *M4HSD* support the same types as TD: boolean, integer, number, and string. Arrays and objects can be used to obtain complex types. For example, if we had a temperature stored as an integer, its value could easily be rendered through a label and its unit of measurement has no impact on its visualization. Similarly, the cooking mode of a microwave oven can be a string and then a label. In contrast, to render a color, we need a complex type, for example and array of integers, but the unit of measurement (e.g., RGB) is key to render the value. This is to say that if the value were `[255, 255, 0]`, and RGB the encoding, then the GUI would visualize a *yellow* square. Objects can be used to encode any complex type. Again, a color could be an object whose fields are integers, and then `["R": 255, "G": 255, "B" :0]` would correspond to *yellow*.

As for rendering actions, we must recall that TDeX describes each action by means of name, type, required inputs, generated outputs, and current value. TDeX comprises five action types that play a key role in how actions are rendered. Also the representation of their inputs, outputs, and current values depends on the actual type. For example, the current value of a **boolean** action can only be true or false (boolean variable), while if we considered a **numerical** action, its current value could be any integer number. As for produced outputs, we can think of a washing machine that calculates the time needed to complete the washing cycle, given the program selected and the weight of clothes. *M4HSD* can also manage scanners or cameras whose output is stored in a file, whose location is managed through a proper URL.

[4]The middleware may understand there is a problem because of dedicated messages sent by the devices or because sent commands are not acknowledged.

Also the way we inflate the action itself, that is, the graphical elements we use to render it, depends on its type. We use a `switch`, and two labels for the two states, for a **boolean** action. For example, `on/off` for a light and `locked/unlocked` for a lock. We use a `seekBar` to materialize a **numerical** action, but we also need a minimum and maximum value, and allowed increment as inputs. We use snippers for **generic** actions and show the multiple options. For example the speed levels of an air conditioner would be represented in a drop-down menu. **Composed** actions are rendered by unfolding them and identifying their atomic components, along with a single button to trigger all the atomic actions together.

*M4HSD* fosters the common *Model-View-Controller* organization of an application. As said, TDeX provides the *Model* the application works with, the *Controller* is as usual in charge of the developer, but again *M4HSD* provides simple RESTful interfaces to interact with the different devices in a standardized and homogeneous way. The *View* is generated automatically from the TDeX model as explained above. Note that a fully automated process can only generate basic GUIs, while specific applications can exploit the information provided by the model to generate advanced and special-purpose interfaces by means of external libraries. It is also true that GUIs are generated at runtime, and thus the application can adjust its GUIs with respect to the current context of interest. One can change devices (technologies) and contexts (locations), but the user interfaces are always generated for all, and only, the devices in the current scope. For example, if one wanted to control the lights of a big building, there is no need to encode the topology of the building and the way lights are distributed in the building. The actual GUIs are generated at runtime while the user moves around and by only considering the lights that can be switched on and off in the different rooms.

## V. Exemplar Application

This section presents an exemplar Android application we developed to access the effectiveness of our solution. First of all, we deployed *M4HSD* in our lab, which comprises two office rooms, one meeting room, a kitchen and a working space. Each location has a privacy level and is equipped with smart devices. For example, the meeting room is a public location and offices are private ones. The meeting room has six simple lights which are grouped together, a ComfoAir air conditioner, two window shutters and a projection screen.

In parallel, we have implemented an Android application that exploits *M4HSD* and TDeX, to interact with the different devices. Figure 3 sketches the processes related to automatic

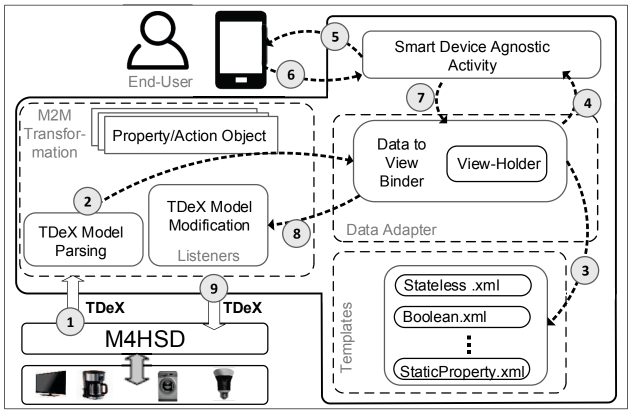


Figure 3: Stepwise process of GUI generation and event handling within the Android application.

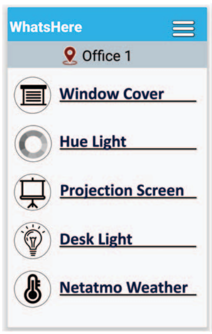
GUI generation and event handling within the application. The developer only needs to decide how to render the two property and five action types (Table I) supported by TDeX to let then the application create the actual GUIs. This means that if Android is the target platform, one only needs to identify the `View` elements for rendering the seven concepts above, and then the actual GUIs can be created automatically. Similarly, the interactions are always carried out through the REST interfaces, and *M4HSD* manages all the peculiar aspects.

More concretely, given that Android stores the concrete layouts of the GUIs associated with activities (roughly, application screens) in XML files, and the link (file-activity) is static, we used an Android `RecyclerView`, to render generic lists of elements, and used pattern *ViewHolder* to define a generic GUI template. Then we fed it with the specific content, that is, the TDeX of the particular devices of interest. To do this, we defined the templates for the seven elements above as XML documents to define their concrete visualization. A template is thus an independent and atomic portion of the final GUI. For example, the template of **boolean** action contains a `TextView` to render the action's name, and a `Switch` to visualize the device's state and change it. We also created a dedicated *Model* layer within the application to interact with the middleware, that is, to retrieve the TDeX models —as JSON files— and create the corresponding objects. This way, any application element can easily, and quickly, retrieve information about available devices and these objects can be shared among the application activities.

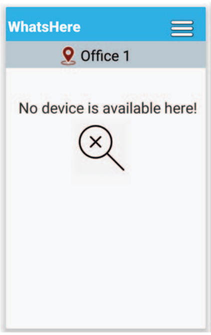
Each Android activity can then create the specific GUI, through a `RecyclerView`, and inflate it while creating the activity itself. For example, the application can parse the TDeX of our air conditioner and creates a GUI to visualize the current temperature, set the target one, put the air conditioner into sleep mode, turn it on/off and adjust the fan speed. This way, applications can render any combination of these templates at runtime to allow users to interact with particular devices.

A user of our exemplar application should first register and log into the system. The, devices are discovered, and becomes available through automatically generated GUIs, based on their locations. When *M4HSD* receives a request, it examines the user's credential and if s/he is authorized to view devices in that location, it sends the list back to the application. If credentials are wrong or the user is not permitted to interact with any device, *M4HSD* returns an *unauthorized* message to the application.

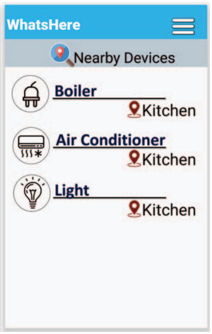
This behavior can be exemplified by considering two possible users of the lab: Bill is a *member* and is the owner of the first office, while Sara is only a *guest*. Accordingly, if Bill entered his office, *M4HSD* would send the list of authorized devices and the application would generate the GUI of Figure 4(a). If Sara entered the same room, the GUI on her phone would be the one of Figure 4(b). If Sara were interested not only in the devices in the first office, but also in those nearby, then she would get the GUI of Figure 4(c), where displayed devices are in the kitchen, which is *near* the first office.


WhatsHere
Office 1
Window Cover
Hue Light
Projection Screen
Desk Light
Netatmo Weather

(a) Bill's view at office 1


WhatsHere
Office 1
No device is available here!

(b) Sara's view at office 1


WhatsHere
Nearby Devices
Boiler
Kitchen
Air Conditioner
Kitchen
Light
Kitchen

(c) Sara's view for nearby discovery

Figure 4: Different GUIs generated according to user identity and location.

As soon as the user selects one of devices (one item of the `ListView` element), a GET request is sent to the middleware to retrieve TDeX of that device. Upon receiving the model (Step 1 in Figure 3), the application parses the model and associates an Action object and a Property object with each device (Step 2 in Figure 3). At runtime, the Activity reads the model, inflates the right templates into the GUI, and populates the different elements through these objects (Steps 3 to 5 in Figure 3). For example, Figure 5(a) shows the automatically created GUIs for interacting with our air conditioner, Figure 5(b) with Hue lights, and Figure 5(c) with a window shutter.

Finally, we have to deal with event handling, that is, the propagation of user actions onto real devices. For example, if a user touches the screen and changes a switch from on to off (Step 6 in Figure 3), the corresponding light must be turned off. To do this, the application creates an `EventListener` for each action type. These listeners are in charge of translating user actions —through the GUI elements— into modifications of the TDeX models. As soon as the graphical switch is slid, its listener is called, which in turn invokes the action-specific listener (Step 8 in Figure 3). This means it changes the TDeX model and issues a PUT to pass it back to the middleware (Step 9 in Figure 3). When *M4HSD* receives the modified version of the model, it triggers the plug-in that translates the change(s) into the right command(s) for the device: that is, the proper light is turned off, and the updated version of the model is persisted. The Android application manages UI events as usual through handlers and callback methods, while *M4HSD* offers means to propagate them and turn them into commands for the different devices. It also manages the current state to allow different applications (users) to interact with the same devices.

Note that nothing is hard-coded in the application. The TDeX models of the different devices are used to generate the GUIs and handlers to interact with them. Hence, the GUI creation process is device-agnostic and only depends on the actual actions and properties provided by the devices. The only requirement for an application to be able to interact with a device is to have access to its TDeX model. The application blindly passes the TDeX model of a device back and forth to *M4HSD*, and ignores what the underlying device is supposed to do. This means that an application could instantly interact with any device in any environment augmented with *M4HSD*. Sara and Bill, for example, could use the exemplar application also at home to control their appliances.

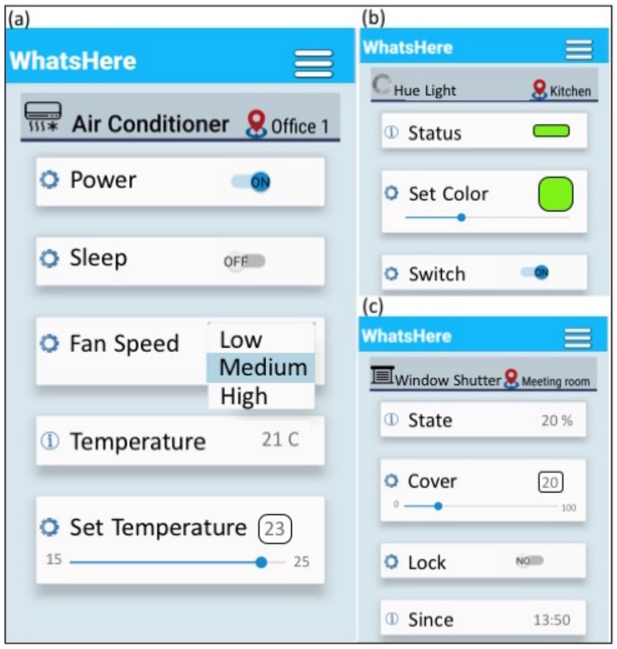


Figure 5: Automatically created GUIs for (a) ComfoAir air conditioner, (b) Hue light and (c) window shutter.

## VI. Related Work

Among the different, competing ontology-based models and vocabularies proposed for modeling IoT/WoT devices, we can cite SSN [5], the ontology developed by the W3C's Semantic Sensor Network Incubator Group in 2011. It is mainly focused on the functional description of sensors and does not support specification of complex or composite devices and other related aspects like the current position of a device. SAREF[5] (Smart Appliances REFerence) covers more generic concepts, like home appliances, and location and similar concepts can be defined, but it does not support different classes of users and there is nothing related to GUIs and user interactions.

The advent of the WoT has also motivated TD, which has inspired this work, and solutions that combine web concepts with data models and ontologies [6]–[10]. Examples of web oriented approaches include IPSO[6] (IP for Smart Objects) and the OCF (Open Connectivity Foundation) resource model[7]. The former is only appropriate for describing the sensing capabilities of smart devices with respect to eighteen predefined types (e.g., temperature, humidity, and presence) and does not consider more complex devices. The latter addresses a wider class of objects, and proposes sixty different *resource types*, does not consider contextual data and user interactions (GUIs), but proposes an interaction model based on RAML (RESTful API Modeling Language [11]). TDeX does not provide any predefined categorization of devices and takes a wider approach. It envisions an extensible, multi-faceted data model for accommodating the functional specifications of smart devices, their contextual information, and also their interaction capabilities. In addition, *M4HSD* supplies a single means to govern the different aspects seamlessly.

As for REST-based IoT and ubiquitous applications, we can mention DigHome [12], which proposes a REST-based publish/subscribe broker that allows devices to produce, distribute, and consume events. DigHome is only interested in fostering the communication among devices, while HomeWeb [13] also provides web-only GUIs to the users. The GUIs are not generated automatically, but are static and predefined for a given set of devices.

Our work can also be related to middleware infrastructures for smart devices, and Home Assistant[8] and OpenHAB[9] are among the most popular ones. Differently from *M4HSD*, these infrastructures focus on network interoperability. They provide means to interact with devices through the same communication interfaces, but the data-models and the commands to interact with devices are device-dependent. This means that the developer should know a priory —and hard-code in the application—- how to interact with a particular device. *M4HSD* fosters syntactic interoperability, and because of TDeX, the interactions with any device are generic and device-agnostic.

In addition, if applications are device-specific, they cannot be used in different environments with conceptually the same

[5]https://sites.google.com/site/smartappliancesproject/ontologies/reference-ontology
[6]https://www.ipso-alliance.org/
[7]https://openconnectivity.org/
[8]https://www.home-assistant.io/
[9]https://www.openhab.org/

devices, but produced by different vendors. Our solution allows one to decouple applications from the environments in which they are used: applications can instantly interact with any environment where there is a running instance of *M4HSD* and exploit it to create required GUIs on demand. This is similar to what the DOG Gateway [14] offers, which is based on an OWL-based ontology (DogOnt [15]) to accomplish semantic interoperability. The big ontology does not consider the GUI-generation problem and is predefined, which could result in a too complex and static solution. For example, DogOnt explicitly considers humidity, temperature, voltage, $CO_2$, pressure, and current, while TDeX would simply consider them as dynamic properties.

The last area we want to touch refers to the model-based GUI generation for ubiquitous and smart spaces. For example, if one considered the Lightweight User Interface Description Language (LUIDS) [16], and the modeling language and tool described in [17], they only focus on GUI descriptions and provided specifications are not enough to render a controller. In addition, semantically identical interactions are described differently (e.g., adjusting light intensity or room temperature are considered to be different functionality, then modeled and visualized differently), and further elements are also needed to take the modifications induced by user actions into account. While probably more sophisticated, these languages are more complex and narrower than TDeX. The Cameleon conceptual framework [18] has inspired a lot of works (e.g., [19], [20]). All these solutions simplify GUI creation at design time, but they do not consider their runtime generation and utilization.

Other solutions (e.g., [21]–[23]) introduce extended and supportive GUI models to envision their runtime utilization. For example, Dynamo-AID [22] uses extended task models to drive the generation of and adapt the GUIs. These models are mainly created by developers, as first manual inputs to the generation process. In contrast, we collect the characteristics of the devices at runtime and generate required GUIs automatically.

## VII. Conclusions and Future Work

This paper presents TDeX, a uniform model for describing and managing IoT/WoT devices, its supporting middleware, *M4HSD*, and a model-based solution for the automated generation of basic GUIs to interact with these elements. Some first experiments witness interesting results and motivate further work. Besides developing additional plug-ins and drivers for interacting with other devices, we are also working on a more sophistical access control model, to govern the interactions with the different devices more in detail, and on shaping the cooperation of multiple instances of *M4HSD*, to conceive more complex and distributed scenarios.

## References

[1] L. Ericsson, "More than 50 billion connected devices," *White Paper*, 2011.

[2] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource-oriented architecture and best practices," in *Architecting the Internet of things*, pp. 97–129, Springer, 2011.

[3] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[4] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and internet of things: a survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[5] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, *et al.*, "The ssn ontology of the w3c semantic sensor network incubator group," *Web semantics: science, services and agents on the World Wide Web*, vol. 17, pp. 25–32, 2012.

[6] S. De, T. Elsaleh, P. Barnaghi, and S. Meissner, "An internet of things platform for real-world and digital objects," *Scalable Computing: Practice and Experience*, vol. 13, no. 1, pp. 45–58, 2012.

[7] S. Hachem, T. Teixeira, and V. Issarny, "Ontologies for the internet of things," in *Proceedings of the 8th Middleware Doctoral Symposium*, p. 3, ACM, 2011.

[8] S. Chun, S. Seo, B. Oh, and K.-H. Lee, "Semantic description, discovery and integration for the internet of things," in *IEEE International Conference on Semantic Computing (ICSC)*, pp. 272–275, IEEE, 2015.

[9] V. Charpenay, S. Käbisch, and H. Kosch, "Introducing thing descriptions and interactions: An ontology for the web of things," in *Proceedings of the 1st Workshop on SemanticWeb technologies for the Internet of Things (SWIT) at the 15th International Semantic Web Conference (ISWC)*, 2016.

[10] W. Xu, C. Marsala, and B. Christophe, "Matching objects to user's queries in web of things' applications," in *IEEE Symposium on Computational Intelligence for Communication Systems and Networks (CIComms)*, pp. 31–38, IEEE, 2013.

[11] R. Workgroup, "Raml-restful api modeling language," 2015.

[12] D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen, "Restful integration of heterogeneous devices in pervasive environments," in *Distributed Applications and Interoperable Systems*, pp. 1–14, Springer, 2010.

[13] A. Kamilaris, V. Trifa, and A. Pitsillides, "Homeweb: An application framework for web-based smart homes," in *18th International Conference on Telecommunications (ICT)*, pp. 134–139, IEEE, 2011.

[14] D. Bonino, E. Castellina, and F. Corno, "The dog gateway: enabling ontology-based intelligent domotic environments," *IEEE transactions on consumer electronics*, vol. 54, no. 4, 2008.

[15] D. Bonino and F. Corno, "Dogont-ontology modeling for intelligent domotic environments," in *International Semantic Web Conference*, pp. 790–803, Springer, 2008.

[16] J. Nichols and B. A. Myers, "Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, p. 17, 2009.

[17] E. Umuhoza, "Domain-specific modeling and code generation for cross-platform mobile and iot-based applications," 2017.

[18] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paterno, C. Santoro, N. Souchon, D. Thevenin, *et al.*, "The cameleon reference framework," *Deliverable D1*, vol. 1, 2002.

[19] G. Mori, F. Paterno, and C. Santoro, "Design and development of multidevice user interfaces through multiple logical descriptions," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 507–520, 2004.

[20] F. Paterno, C. Santoro, and L. D. Spano, "Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, p. 19, 2009.

[21] G. Varela, A. Paz-Lopez, J. A. Becerra, and R. Duro, "A framework for the development of context-adaptable user interfaces for ubiquitous computing systems," *Sensors*, vol. 16, no. 7, p. 1049, 2016.

[22] T. Clerckx, K. Luyten, and K. Coninx, "Dynamo-aid: A design process and a runtime architecture for dynamic model-based user interface development," in *International Workshop on Design, Specification, and Verification of Interactive Systems*, pp. 77–95, Springer, 2004.

[23] M. Blumendorf, G. Lehmann, and S. Albayrak, "Bridging models and systems at runtime to build adaptive user interfaces," in *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pp. 9–18, ACM, 2010.