# Sense, Transform & Send for the Internet of Things (STS4IoT): UML Profile for Data-Centric IoT Applications

Julian Eduardo Plazas*
Universidad del Cauca
Popayán, Colombia
Université Clermont
AuvergneAubière, France
jeplazas@unicauca.edu.co

Sandro Bimonte
TSCF, INRAE Clermont-Ferrand
Aubière, France
sandro.bimonte@inrae.fr

Michel Schneider LIMOS,
Campus des Cézeaux, Université
Clermont Auvergne
Aubière, France
michel.schneider@isima.fr

Christophe de Vaulx
Université Clermont Auvergne,
CNRS, Mines de Saint-Étienne,
Clermont-Auvergne-INP, LIMOS
Clermont-Ferrand, France
christophe.de_vaulx@uca.fr

Pietro Battistoni
Università degli Studi di
SalernoFisciano, Italy
pbattistoni@unisa.it

Monica Sebillo
Università degli Studi di
SalernoFisciano, Italy
msebillo@unisa.it

Juan Carlos
CorralesUniversidad
del Cauca Popayán,
Colombia
jcorral@unicauca.edu.co

Accepted version published in Data & Knowledge Engineering,
DOI: 10.1016/j.datak.2021.101971

A B S T R A C T

The Internet of Things is currently one of the most representative sources of Big Data. It can acquire real-time data from multiple spatially distributed points, allowing for the extraction of valuable insights. However, an appropriate integration, processing, and analysis of these data depends on several factors starting from the correct definition of the information systems. This paper introduces *STS4IoT*, a UML profile and automatic code-generation tool for model-driven IoT, to address this issue. STS4IoT allows designing and implementing an IoT application from the required data only, bridging the gaps between the IoT and database design worlds. The IoT data design includes both different in-network transformations and the join of streams from multiple sources. Besides, it follows the Model-Driven Architecture (MDA) guidelines to provide abstraction levels oriented to the different roles participating in the application design. The STS4IoT validation shows it has an excellent structure and is highly understandable. Its instance models are well-formed, highly abstract and readable. And the automatic implementation tool can generate complete code for complex real-world applications involving multiple IoT devices. Then, STS4IoT simplifies the definition and development of IoT applications and their integration into other information systems, such as stream data warehouses.

## 1. Introduction

Generally speaking, the Internet of Things (IoT) refers to a global, distributed network of physical objects. These objects may sense and compute data and communicate amongst them or with other computers or humans using different network protocols such as ZigBee, Wi-Fi, LoRaWAN. These capabilities differ from object to object since the IoT comprises several devices, ranging from simple sensors to advanced industrial robots. Nevertheless, most basic IoT devices can collect and communicate data.

Recently, IoT has penetrated almost every application domain, from healthcare [1] to agriculture [2], thanks to insights raised from analysing its data. Indeed, their pervasive spatial and temporal coverage has positioned IoT as one of the most representative sources of Big Data [3–5].

Technical IoT solutions and architectures have reached the maturity level [4]. However, they have a high degree of complexity derived from the association of several features:

- Spatio-temporal, historical and streaming data [6]. IoT data are *complex Big Data* characterised by the 3Vs features, namely *Volume, Variety* and *Velocity*. Moreover, IoT data are usually geo-referenced and associated with temporal attributes since they usually define data streams with particular temporal windows for acquisition and delivery. These data streams are also commonly associated with classical (transactional or historical) data.

- Continuous network architectures (*i.e.* fog-edge-cloud) [7]. IoT data evolve over network architectures where objects exchange data in a progressive physical organisation involving different types of devices that join, transform and re-send data over the Internet.

- Distributed computation [8]. Objects can only compute data they handle at a local level. However, centralised data computation is usually costly and disregards the capacities of individual devices. The IoT should then coordinate data operations throughout the different layers of the network architecture to achieve a coherent overall computation.

Applications based on such complex data require a rigorous conceptual design phase. Indeed, as widely proved for classical information systems [9], conceptual modelling is mandatory since it provides several benefits. First, end-users can ignore the implementation details of the application and focus on its functional requirements. Second, it provides a standard and non-ambiguous formalism to validate the application requirements. Third, it guides the implementation process (sometimes also by an automatic process).

In the context of conceptual modelling, much attention has been devoted to UML profiles and Model-Driven Architecture (MDA) since they can fully satisfy such benefits [10].

A UML profile offers a generic extension mechanism for customising UML models in particular domains and platforms. Its definition uses stereotypes, tagged values, and constraints applied to specific model elements, such as Classes, Attributes and Operations. A UML profile also provides a graphical and formal notation for the functional requirements (in terms of data) expressed by the end-users (decision-makers and domain experts) of an IoT application.

MDA is a software design approach for the development of software systems. It provides a set of guidelines for structuring specifications expressed as models organised in different levels of abstraction that define various aspects of the system. The highabstraction level is the Platform-Independent Model (PIM). It represents the functional requirements of the application without any implementation detail. The low-abstraction is the Platform-Specific Model (PSM) that extends the PIM with characteristics from the implementation platform. Finally, the lowest abstraction level is the implementable code automatically generated from the PSM.

It is important to note that conceptual models represent the schema of the data used by the application and the cardinalities of their associations. Besides, UML also includes a declarative language called Object Constraint Language (OCL), which allows defining constraints over the instances of the classes. In this way, designers can define particular rules for the modelled data (*i.e.* schema instances).

Several UML profiles and MDA approaches have been proposed and successfully applied for transactional information systems, Data Warehouses, and stream applications [11]. This scenario has motivated the adoption of a mandatory conceptual design phase for successful IoT projects [5,12,13]. Along this line, some recent works propose formalisms for IoT systems, mainly based on UML and MDA, such as [14–16]. However, they are not appropriate for modelling IoT applications where data play a fundamental role [17,18]. Indeed, the challenge in conceptual modelling IoT data is an abstract, complete and data-centric representation of:

- ***Complex data features*** (Spatio-temporal, historical and stream) from the IoT definition and implementation. IoT data must become easy to understand and integrate into the subsequent systems for analysis. We address this challenge by allowing decision-makers and domain experts (*i.e.* business users) to focus on IoT Functional Requirements (FR), *i.e.* issued data and their computation, ignoring technical implementation features (such as efficiency or network topology). Besides, giving business users a significant role in the application design increases the success of a project [17].
- ***Computational*** and ***communication*** **capabilities** [19]. IoT objects can (and usually do) compute the sensed data before sending them to the subsequent data system. These computation tasks are strongly associated with the needs of business users regarding data and its further analysis. In this way, the design framework must also provide a transparent view of these computation tasks besides the data view. Furthermore, IoT applications generally rely on multiple IoT objects with different roles and tasks that send data. Therefore, the design framework must allow IoT experts to define and distribute such communications and tasks according to contextual needs.

Consequently, to address this challenge, we propose an extension of our previous work [16] to model IoT data and its ***operations*** and ***communications*** throughout the network in this paper. Classical *Extract-Transformation-Loading* (ETL) methods used for Business Intelligence (BI) systems [20] inspired our approach: *Sense-Transform-Send (STS)*. This new modelling paradigm for IoT applications offers a complete description of the computation and

communications provided by multiple IoT devices. Including the definition of transformations over sensed or received data (*e.g.* with aggregation, formatting, join, or filtering) before sending them to the subsequent data-management system (*e.g.* a BI).

We also extend our previous work [16] with new PIM, PSM, DM and automatic implementation (STS4IoT). In particular, PIM and PSM of STS4IoT provide a highly abstract representation of IoT data, their computations and communications (sensing, transformation and sending operators) (*i.e.* FR). Consequently, they provide a deeper understanding of the data received from the IoT for their analysis and use.

This paper is organised as follows. Section 2 introduces a reporting application in smart farming as a case study. Section 3 discusses the most relevant related works on conceptual modelling for IoT data and summarises the main outcomes of [16,21]. Section 4 presents the updated UML profile of STS4IoT, including the new computational (Section 4.2) and communications (Section 4.4) capabilities. Section 5 describes the transformation process to implement the models. Section 6 validates the STS4IoT profiles by using two theoretical approaches. Section 7 describes the observations issued from the use of STS4IoT in a real case study. Finally, conclusions and future works are drawn in Section 8.

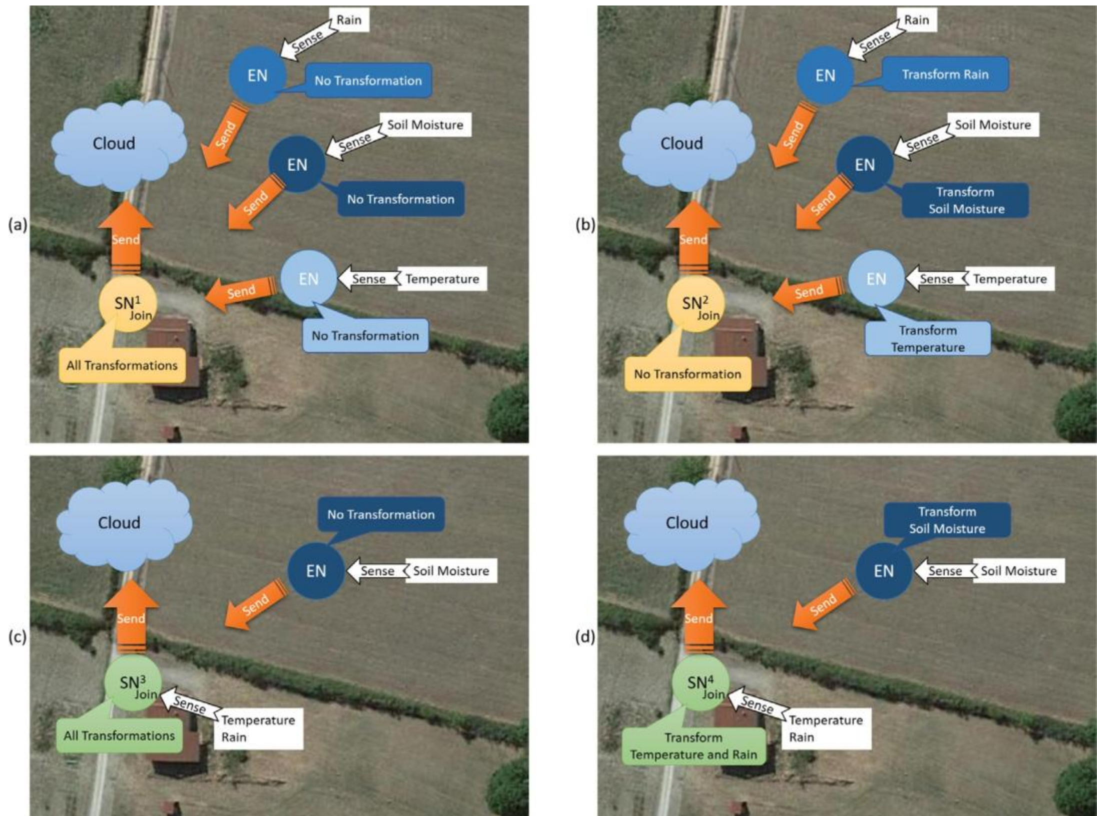


**Fig. 1.** Some deployment options for the running example application.

## 2. Running example

Amongst the multiple IoT applications in smart farming [2], the present paper focuses on smart irrigation to illustrate the use of the proposed profile.

Smart irrigation is a complex task that has to consider different meteorological and phenological parameters, such as (i) daily median soil moisture measured at 30 cm and 60 cm depth, (ii) daily Growth Degree Unit (GDU), which is

daily heat accumulation, and (iii) daily accumulated rainfall. Then, if a crop has accumulated enough heat, the soil is not moist enough, and there is no rain, it requires irrigation [22].

IoT devices can easily sense these variables to avoid problems, costs, and limitations of manual collection. A stream reporting system would allow farmers to visualise the irrigation needs by crop and plot in real-time from these data. Such a reporting system is an IoT-based application.

Implementing such stream reporting systems is complex since it concerns data, sensors, and network modelling issues. Fig. 1 displays four implementation options for this IoT application considering some hardware and contextual constraints, collected data, and operations on data. Those configurations are pretty standard in the agricultural context, where sensors are deployed in fields to communicate their data over a decision-making application deployed over the cloud. Moreover, since most sensors rarely have an Internet connection, they must be connected to the cloud via another communication and computation layer deployed at the farm level, according to the well-known edge-fog-cloud architecture adopted for IoT. Let us note that different kinds of sensors can be deployed:

- **Sink nodes** (SN) (such as a meteorological station and a laptop), which have electrical power, the Internet connection, and significant computation and storage tools;
- **End nodes** (EN) (such as soil moisture sensors), which have battery power, no Internet connection (but support some wireless protocols), and low computation and storage tools.

In all configuration options presented in Fig. 1, the Internet access point is in the farmer's house, and thus an SN is positioned there to upload data into the cloud. Also, temperature, rain, and soil moisture data are collected every 20 min, one hour, and 24 h (respectively); and uploaded to the cloud every 24 h. Nevertheless, there are relevant differences between each option.

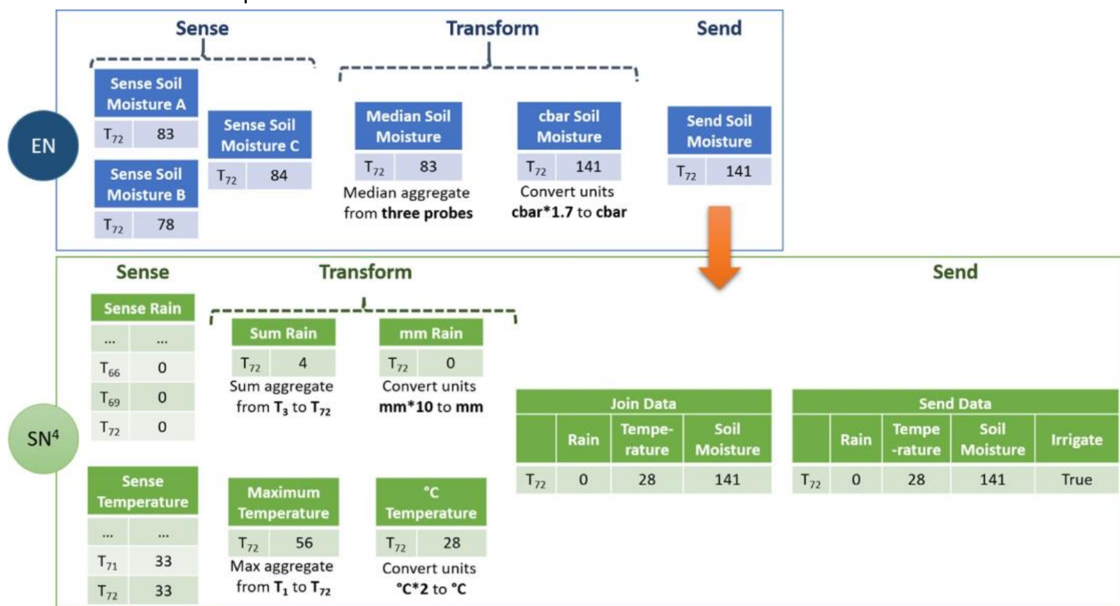


**Fig. 2.** An example of simplified data and operations flows for the deployment in Fig. 1-D following STS.

The upper configurations (Figs. 1-A and 1-B) use four devices, namely three end-nodes (EN) and one sink-node (SN). This distribution provides more reliable data (sensed directly on the plot). Rain, soil moisture, and temperature measures have high spatial precision.

As for option A, having more devices makes the implementation more complex and expensive since its ENs send raw data, consuming more battery to communicate.

As for option B, the ENs aggregate (*i.e.* data transformation) the last 24-h of data, thus sending data exclusively once a day. In this way, option B reduces the number of communications and energy consumption by directly transforming data into the ENs [23].

The lower configurations (Figs. 1-C and 1-D) use two devices, namely one EN and one SN. The EN reads the soil moisture directly in the plot, while the SN reads both temperature and rain. In option C, the EN does not aggregate the data, while the EN in option D makes some computations as in option B.

Although these configurations may reduce the measures reliability (*i.e.* precision), since they use only two sensing points, they also reduce the complexity and costs of implementation.

Fig. 1-D represents the most efficient option. Unlike option C, which sends raw soil-moisture data to the SN, it reduces the number of communications by directly transforming data inside the EN.

Fig. 2 shows a simplified example of data and operations flows in the EN and SN of Fig. 1-D. Both nodes collect their data in regular intervals denoted by $T_x$. The End Node senses and stores soil moisture data from three probes in time-instant $T_{72}$. Then, at the same time instant, it aggregates the stored data by using the Median (producing the 83 value) and converts the measurement units from 1.7*cbar to cbar (141 value). Finally, it sends this value to the sink node during the same $T_{72}$. Likewise, SN senses and stores temperature and rain values until $T_{72}$ and aggregates them in this time instant by using sum and maximum, respectively. Then, it converts their units to mm and °C. During the same time-instant $T_{72}$, also it receives the soil-moisture value from the EN, joins all these data with the same timestamp ($T_{72}$), calculates whether irrigation is necessary, and sends them to the cloud.

Finally, a web-based application in the cloud receives the data, calculates the irrigation need, and displays the information.

Fig. 3 shows a web dashboard with real-time data visualisation. The current sensor data are displayed numerically, and a window of the latest ten readings is displayed graphically for each sensor (on the left side). Since all sensors are geo-localised, they are displayed on a live map (on the right).

The web application receives sensor data through WebSockets events. This solution allows the connected browser to receive sensor data whenever they are ready, without polling or multiple HTTP requests. The data exchange between IoT and dashboard follows the Publish–Subscribe paradigm. Every time the sink node (*i.e.* data producer) is ready to send its data, it must publish it to a *Topic*, which generates an event. In this way, any application subscribed to a *Topic* will receive a notification event when new data is published. Utilising Messaging Queuing Telemetry Transport (MQTT) technology, the sensors publish their data to a known Broker under a specific Topic (event). Registered applications will receive the Broker data as soon as possible, although asynchronously. In our case, the Broker also acts as a web-sockets server, so any browser processing a lightweight JavaScript MQTT client code and connected to that Broker will receive the event notification and related data without an HTTP request. Publishers and Subscribers do not need to know each other. They only need the Broker address and an account to access it when a security policy is applied.

Moreover, this implementation moves the data visualisation and geographical elaboration to the client (browser machine). Thus, tiny Edge-Device can act as web servers, avoiding the Internet connection to Cloud resources.

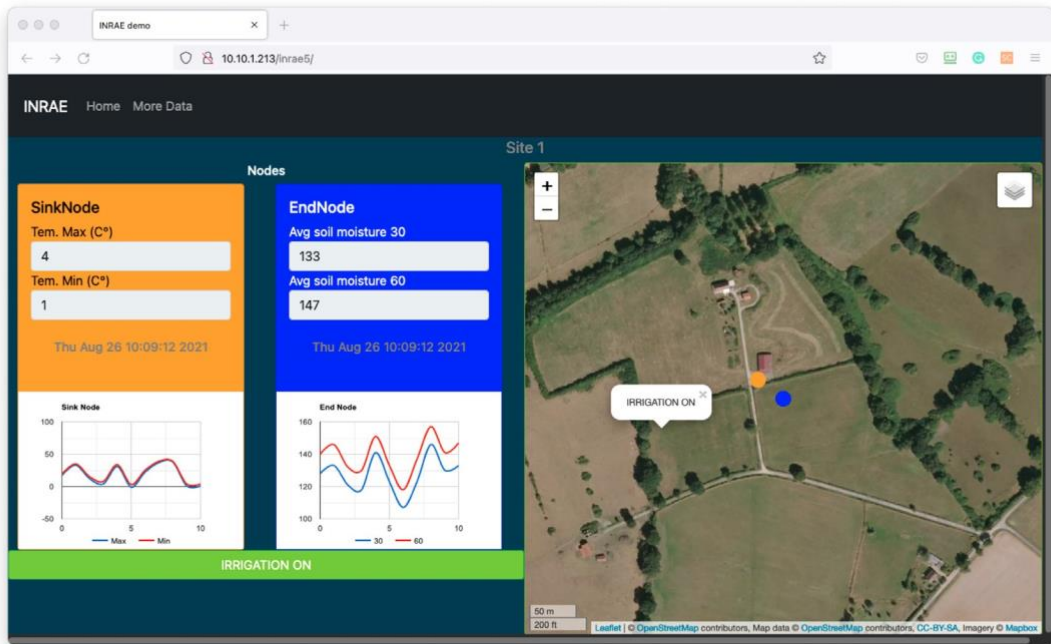


**Fig. 3.** Web-based dashboard for irrigation.

The example described in this section shows the complexity and the diversity of the possible configurations of the same IoT application, which directly impacts the data and costs of these applications.

Therefore, a highly abstract design is crucial for farmers and agronomy experts to exclusively focus on the applications' FR (*i.e.* their data and analysis needs). Besides, lower-level modelling and model-to-code transformation are essential for IoT experts to define the most relevant technical issues (*e.g.* the role and assignment for each node) and seamlessly achieve a successful implementation.

## 3. Related work

We analyse in this section the most recent research works on model-driven and data-centric IoT. Moreover, we provide a brief description of our previous approach [16], stating its advantages and limitations, which we address in STS4IoT.

Recently, some works have focused on modelling IoT-data information systems in the edge and cloud, such as [24–27]. Some of these works propose interesting concepts (such as Data as a Service), conceptual data models, or even allow for multiple transformations and operations over IoT streamed data. Nevertheless, since we focus on modelling and implementing data *inside* the IoT devices and network before uploading, such approaches are out of the scope of this research.

Similarly, the academic literature on databases presents several works about Extraction Transformation Loading (ETL). Most of the existing works propose a conceptual model based on UML, BPMN, and other formalisms representing relational operators used by ETL [28]. However, these approaches generally focus on static data (table files and relational databases). They do not address the continuous sensing and sending operations that characterise IoT nodes. Moreover, their transformations can be utterly complex, contrasting with the limited resources and simple computations of IoT devices.

Therefore, we detail only the most pertinent works to our IoT data-centric representation goal. Table 1 provides a categorisation of such works considering four criteria:

(1) Proposal of a **Conceptual Data Model**. A formal and abstract representation of a particular kind of data (*e.g.* spatial, temporal, historical, static, graph) as a meta-model, where each element instance defines a specific graphical notation.
(2) Modelling and implementation of **Data Transformations inside the IoT**. Here we consider at least three primary preprocessing transformations [37]: *Aggregation*, which allows summarising a set of data to increase their value; this is a crucial transformation in IoT systems that requires the creation of temporal windows [38]. *Conversion* allows changing the data types and format (*e.g.* different units) without altering the intrinsic meaning of data. And *Filtering*, which removes undesired or potentially erroneous measurements. Moreover, we consider the possibility of *additional transformations* and whether the models are *easily extendable*.
(3) Modelling and implementation of **Data Communications between IoT objects**. Here we consider three main features that allow for a correct representation of data transmission and integration: The model explicitly represents the *Join* of data streams from external sources. The model enables the *Composition Association of data* from different sources. And the model allows representing a *network*.

**Table 1**
Classification of the most recent related works.

| Paper | Conceptual Data Model | IoT Data Transformations | | | | | IoT Data Communication | | | Implementation | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Aggregation | Conversion | Filtering | Additional Transformations | Easy Extension | Join | Composition Association | Network | Multiple Options | Automatic Generation |
| [29] | No | No | No | No | None | No | No | No | Yes | Yes | Yes |
| [30] | No | No | Yes | No | Calculations | Yes | No | Yes | Yes | Yes | Yes |
| [31] | No | No | No | No | None | No | No | No | Yes | Yes | Yes |
| [16] | Yes | Yes | No | No | None | No | No | No | No | No | Yes |
| [32] | No | Yes | No | No | None | No | No | Yes | Yes | No | Yes |
| [33] | No | No | No | No | None | No | No | No | No | No | No |
| [34] | No | No | Yes | No | None | No | No | No | Yes | No | Yes |
| [35] | No | No | No | No | None | No | No | No | No | No | Yes |
| [36] | No | No | No | No | None | No | No | No | No | Yes | Yes |

(4) The **Implementation** of the models considers two parameters: whether the model allows for *multiple implementation options* of the same data; and whether the approach *automatically generates* the required code.

[29] proposes a model-driven process to develop Digital Twins of IoT systems, *i.e.* complete simulations of the IoT that evaluate several aspects before the actual implementation. This approach considers the data streams generated and transmitted amongst IoT objects. However, it focuses on the identification of possible failures at different levels.

[30] defines a conceptual (system) model for the Industrial Internet of Things following the Service-Oriented Architecture (SOA). This work is centred on the interoperability of IoT devices and applications, specifying them as services of different granularity through the concept of Sensing as a Service. This approach allows designing conversions and calculations on a single datum and is easily extendable for other (single-datum) transformations. Besides, it enables the data composition from different sources in their network representation. Nevertheless, it fails to represent mechanisms for transforming or merging stream data (*e.g.* temporal windows and timed join). Thus, it cannot represent data aggregation or data received from external sources.

[31] presents a meta-model to integrate IoT data into cloud-based RESTful services. They emphasise the interoperability amongst heterogeneous IoT objects and their connection with centralised systems. In this way, they represent and implement multiple network options. Yet, they do not consider the data transformation or composition.

[32] proposes a method for the seamless integration of IoT data into Digital Twins of complex information systems. It provides a conceptual model that also allows for the aggregation and composition of data. Even though

it is an abstract model, it focuses on multiple system details and thus hinders the definition of additional transformations and the representation of different implementation options.

[33] provides multiple meta-models to represent IoT objects, Microservices, Ansible, Docker, and Kubernetes technologies. Then, it defines different integration mechanisms amongst the meta-models, which leads to the conceptual integration of IoT and Docker for easy implementation. Although promising, this proposal does not consider any operation on IoT data or the representation of the network.

[34] proposes the integration of semantics into the WSN for easier data use and integration with different kinds of data. For that purpose, it defines a meta-model for semantic-enhanced WSN and architecture for complete applications (from the devices to the cloud). The meta-model enables the conversion of IoT data through semantics, though the definition of other transformations is not clear. Besides, it allows representing the network and data communications, but the composition and integration of sensed data execute only at the cloud level.

[35,36] propose easy-to-use approaches for the simple definition and implementation of IoT. [35] defines AutoIoT, a useroriented framework that simplifies the building of complete IoT applications (from the devices to the could). It uses an integration meta-model that represents the whole system and its communications. [36] presents Midgar, a graphic-programming platform for Arduino© IoT devices. It provides a simple drag and drop interface to define the hardware device and sensors used in the implementation; then, it automatically generates the corresponding code. However, these works do not consider any transformation or composition of the sensed data inside the IoT.

As shown above, no work in the context of IoT covers the design and implementation requirements we have found out. Nevertheless, a large amount of research in the context of Business Intelligence (BI) use MDA to design and implement ETL and Data Warehousing (DW) systems for classical data. For instance, [39] proposes a UML profile for modelling complex DW through a multidimensional data model. [40] provides a complete MDA and UML profile for the automatic implementation of relational DBMS. Other works such as [41] have also contributed to this issue. Besides, [42] defines a model-driven approach for the design and implementation of ETL processes in multiple platforms. [43,44] propose MDA to improve the definition and maintenance of ETL and DW systems. While [43] records how the initial user requirements change in the MDA transformation process, [44] tracks how the user requirements and DW model evolve in the different versions of the DW. These works evidence the importance of modeldriven, data-centric approaches for modelling and implementing complex systems such as DW or ETL; yet, they remain focused on classical data.

Therefore, in our previous work [16], we define a simple conceptual data model for single-object IoT applications and their implementation and integration into complex BI systems. [16] specify their UML profile according to the MDA principles in different abstraction levels and concerns: application Platform-Independent (PIM) and Platform-Specific (PSM) models, and platform Device model (DM). The PIM represents the sensed data used by the BI application. In particular, it provides a conceptual representation of IoT data without any reference to the hardware sensors. Then, the PSM describes the same data providing details about the physical sensors available in the DM. The DM represents the physical sensors with measurable data (*e.g.* temperature or pressure) and their physical implementation details, such as sensing probes or storage memory size. This work allows transparency and easy integration with Data Warehouse conceptual models thanks to its data-centric representation of sensor data [21].

Nevertheless, [16] presents two main limitations:

- Firstly, the processing capabilities of IoT devices have increased considerably, and thus current applications execute various transformations on the data before sending it [15]. Therefore, a conceptual representation of IoT data should describe such transformation operations since business users must know how data is acquired and modified before they receive it. [16] *allows only to represent a unique aggregation function* that is not representative of the current processing capabilities of sensors.
- Secondly, common characteristics of IoT and sensors applications are the direct communication amongst devices and the distributive computing over different layers according to the continuum communication

paradigm offered by the edgefog-cloud architectures [45]. However, [16] focuses *exclusively on representing isolated sensors* that do not form a real IoT network.

From the study of the related work, we evidence the importance of model-driven approaches in reducing the efforts for building complex IoT applications. Contributions on this topic go in different directions, from the definition and integration with digital twins to highly useable graphic interfaces. Various approaches represent the devices' network since it increases the range of applications. However, few proposals consider the data composition from multiple devices, while none of them makes an explicit representation of joined streams.

Furthermore, even the most recent approaches cannot explode the enhanced processing and communication capabilities that IoT devices acquire with every upgrade. Indeed, even though existing literature widely acknowledge data aggregation as a prime feature for properly managing data in IoT systems [38], only two approaches included it (one of which is our previous approach). In the same way, conversion and filter are less demanding than aggregation and common on manually-developed IoT, yet only two proposals considered data conversion.

#### 4. STS4IoT UML profile

In this section, we present the STS4IoT UML profile. In particular, we divide each meta-model (PIM in 4.2 and PSM in 4.4) into the data model and the STS operations.

### *4.1. STS4IoT overview and methodology*

STS4IoT extends and updates the UML profile of [16] with new stereotypes, tagged values, and constraints that allow for a complete design of real-world IoT applications. STS4IoT still follows the MDA guidelines, having three abstraction levels: PIM for the data design, PSM for the implementation design, and DM for the platform information. Besides, it also provides an automatic-implementation system: *STS4IoT model-to-code* tool.

Fig. 4 shows how different actors can use the STS4IoT components to define IoT data-centric applications. In particular,

- Firstly, business users (*i.e.* decision-makers and domain experts) define their functional (data) requirements (FR) for the application in natural language: what data to sense, how to acquire them, how to transform them, and what data to send and how to send them.
- Then, the IoT experts translate these requirements into a PIM model. If the available DM models do not offer the required measures, they must add a new DM model for a device that can provide the data needed. This step is an iterative process that demands constant communication with the business users.
- Once the PIM respects the functional requirements, IoT experts can address the implementation issues. They list the technical features that the implementation must support, such as hardware devices and sensors, topology, and network communications. According to these technical issues, IoT experts define some (one or more) PSM models that map the PIM into particular implementations, respecting both the functional requirements and technical issues. This step is also an iterative process.
- The *STS4IoT model-to-code tool* automatically implements these PSM models. The code is used within a simulator that simulates data sensing for test purposes. IoT experts must redefine the faulty PSMs whenever the test simulations do not match the implementation choices. Finally, the business users receive the resulting systems and test data.
- There is always a gap from the conceptual definition to the implementation. This gap is not easy to remove, even following good communication processes. Therefore, business users must evaluate the received outcomes against their original FR. If the implementation does not fit the business users' needs, a new design phase for the PIM design is required, and the process restarts. For example, the PSM models corresponding to Figs. 1-C

and 1-D may not satisfy the reliability needed by the business users. Therefore, they should define a new PIM associating rain and temperature data to a class representing plots. In this way, IoT experts and business users can precisely state what data they acquire from each field at the conceptual level.

- Finally, business users select the simulated implementation that better meets their requirements. Then, IoT experts use the corresponding PSM and code for IoT hardware configuration and deployment.

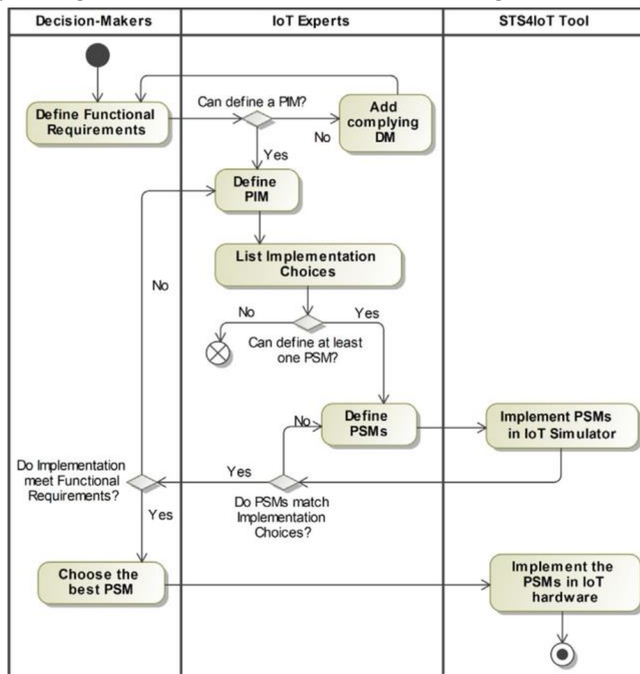


**Fig. 4.** STS4IoT methodology.

### 4.2. *STS4IoT PIM*

This section presents the PIM, which allows formalising the business users' data needs in UML without providing any physical or implementation details. Fig. 5 highlights the stereotypes that model the sensing, transformation, and sending parts with dashed blue lines, dash-dotted grey lines, and dashed double-dotted orange lines, respectively.

The high-abstraction level of STS4IoT (the PIM) describes the data-related functional requirements of IoT applications. Specifically, the PIM defines the variables to sample, sampling rate, required transformations, transformation (time) windows, transmission rate, and transmitted data. This level aims to provide a highly readable representation of the IoT application for business users.

For example, Fig. 6 represents the IoT application indicator providing daily median soil moisture, daily maximum and minimum temperatures (to calculate GDU), and daily accumulated rainfall data for our running example (Section 2). To better understand the PIM, we describe it in two components: data structure 4.2.1 and STS operations 4.2.2.

#### 4.2.1. *PIM data structure for STS*

The PIM data structure (Fig. 5-A) allows defining the variables required from the IoT application. In this way, it constitutes a **conceptual data model for IoT** data.

This data representation has its basis on two main classes: A_PIM_Source_Measure and A_PIM_Measure. They represent simple and composed sensed data, respectively. A_PIM_Source_Measures contain simple sensed and transformed variables. A_PIM_Measures present a join operator that abstracts the logic behind the composition of those simple variables. In our case study, individual A_PIM_Source_Measures represent each sensed type of data

(*e.g.* rain, temperature and soil moisture), while a single A_PIM_Measure represents the composition of all these sensed data.

In particular, from the previous profile [16], we keep three stereotypes: A_PIM_Data, A_PIM_Measure and A_PIM_Variable. These elements provided a clear and functional representation of IoT data and only required slight modifications to comply with STS (*e.g.* in their tagged values).

A_PIM_Measure represents IoT data composed of multiple sensed data. The A_PIM_Variable property models these composing variables. Moreover, the type of these properties should state the generic type of each sensed variable (*e.g.* air temperature or soil moisture). Nevertheless, when the IoT application requires a clock to measure time, the model should declare an
A_PIM_ExplicitTime property instead of a regular A_PIM_Variable.

Contrary to [16], STS4IoT allows representing composed data using the A_PIM_Measure stereotype, not only simple sensed data.

The A_PIM_Source_Measure stereotype provides further details about the sensed data. The chief A_PIM_Measure can relate with multiple A_PIM_Source_Measures through associations stereotyped as A_PIM_GatheredAs. Besides, we defined the Source tag as a link between each A_PIM_Variable in the main (composed) class to their particular source measure.

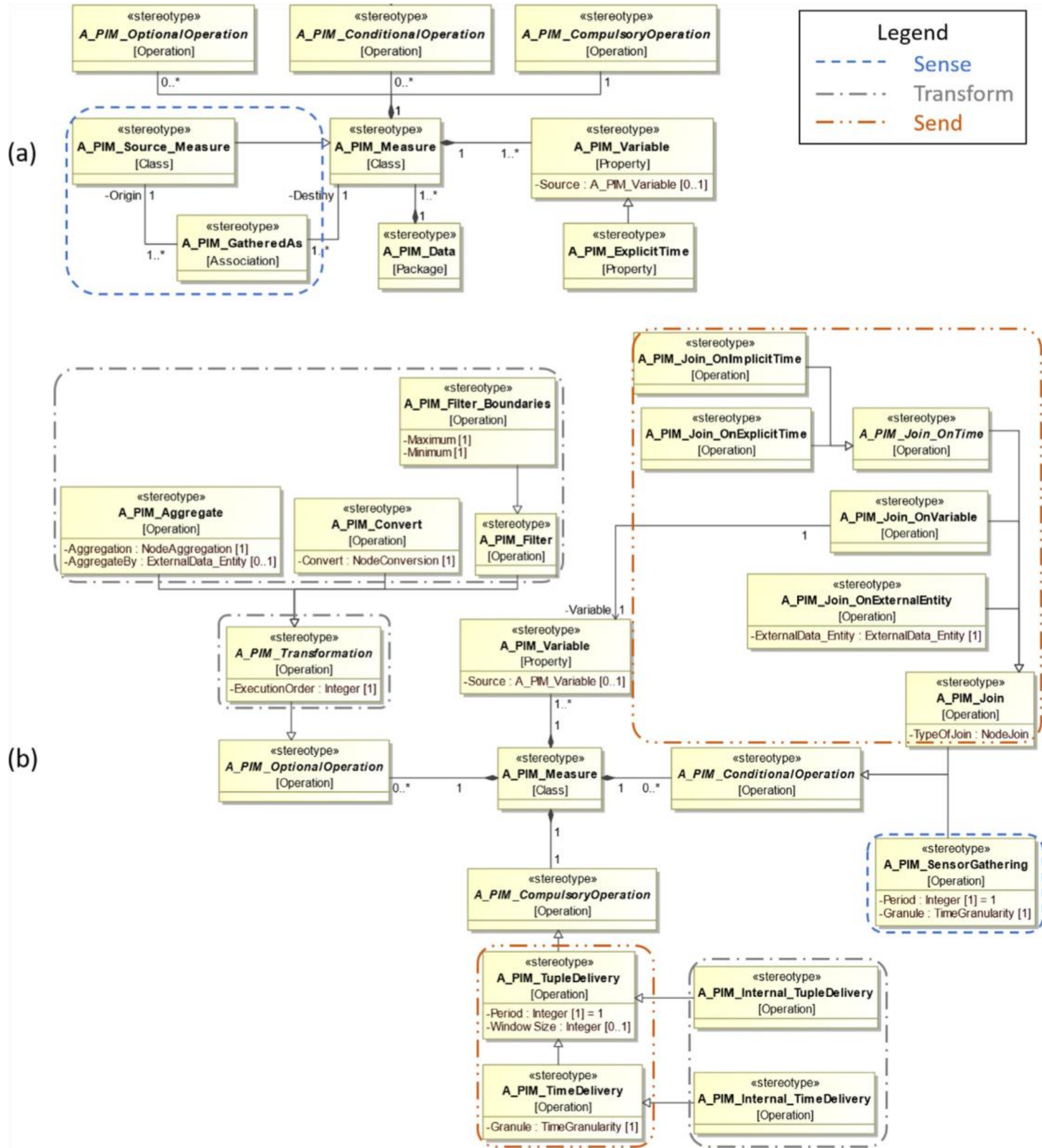


**Fig. 5.** STS4IoT PIM profile data part (a) and STS part (b). Dashed blue lines are for the *Sense* stereotypes, dash-dotted grey lines for the *Transform* stereotypes, and dashed double-dotted orange lines for the *Send* stereotypes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Furthermore, considering that A_PIM_Measure represents a composition of data streams arriving with different frequencies and attributes from various end-nodes, these classes need an explicit mechanism for joining such stream data [46]. Therefore, we define a new operation stereotype that is A_PIM_Join (Fig. 5-B). It allows for joining data from different sources. This Operation is only required (and allowed) in A_PIM_Measures and A_PIM_Source_Measures with two or more Associations. The tag TypeOfJoin defines whether the join considers only matching values (*Inner*) or all the values in the window (*FullOuter*). This stereotype and its specifications define what data the IoT finally sends.

A_PIM_Join has four different specifications: Firstly, A_PIMJoin_OnExternalEntity allows joining data associated with the same external entity, *e.g.* plot. Secondly, A_PIM_Join_OnVariable allows merging data when the value of a

particular A_PIM_Variable is the same. Finally, A_PIM_Join_OnTime joins all the valid data (validity defined as the Period in the Source) inside the window (defined by WindowSize in the corresponding delivery Operation). The join on time can consider the explicit time (A_PIM_Join_OnExplicitTime) as the time gathered and delivered by the source, or the implicit time

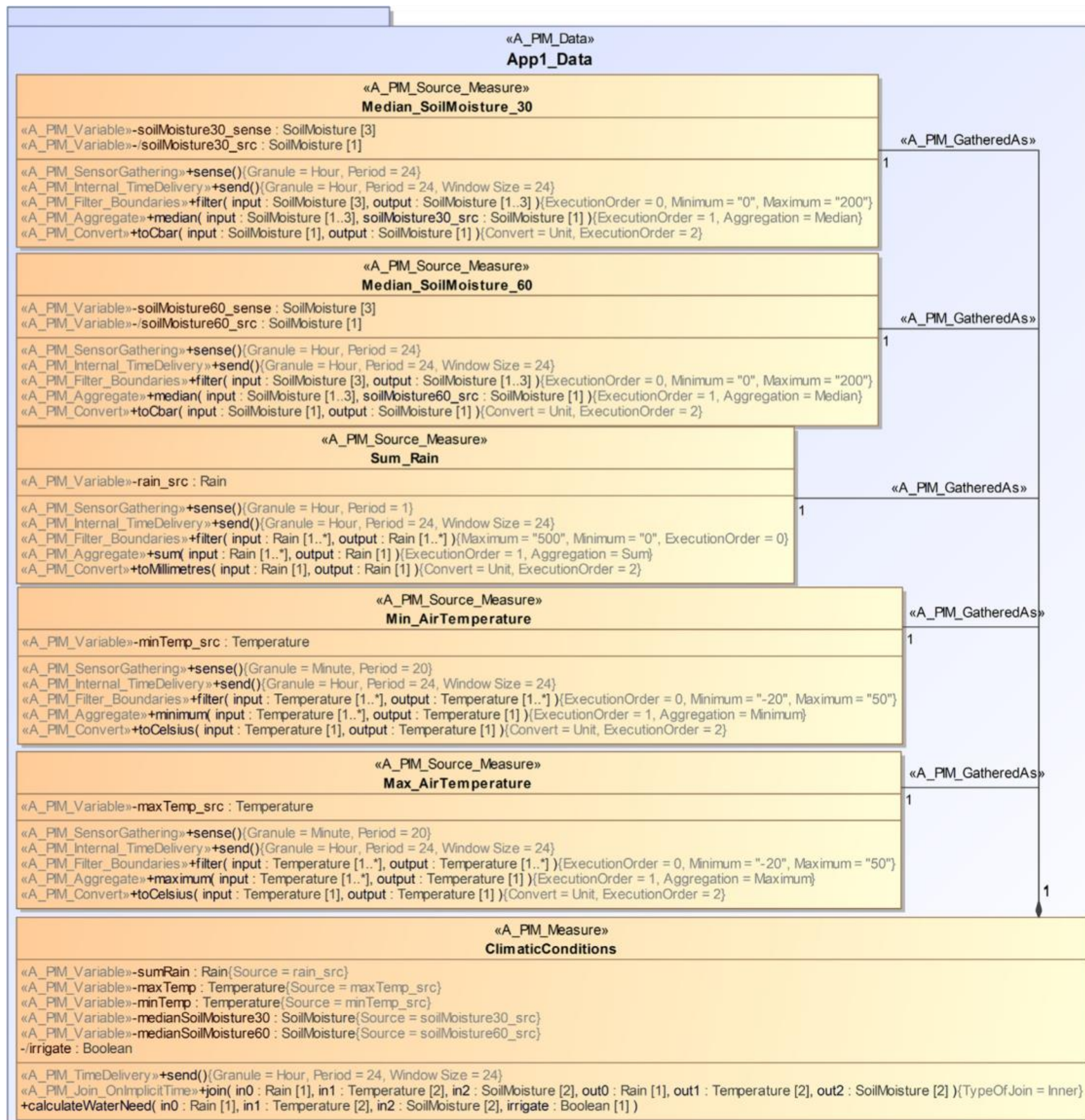


**Fig. 6.** Possible PIM for the running example making all the transformations inside the IoT.

(A_PIM_Join_OnImplicitTime) simply as the time at which the data was available. A_PIM_Join_OnImplicitTime is the simplest join type since it considers all the data available in the window.

Finally, A_PIM_Data represents the package containing all IoT data: A_PIM_Measures and A_PIM_Source_Measures.

**Example.** In the following, we present the PIM of our running IoT example (Fig. 6). *ClimaticConditions* represent all data used by irrigation rules described in Section 2. It is a composition of four variables (which have three types):

*sumRain* (daily accumulated rainfall), *maxTemp* and *minTemp* (daily maximum and minimum temperatures), and *medianSoilMoisture30* and *medianSoilMoisture60* (daily median soil moisture measured at 30 and 60 cm in the ground). *ClimaticConditions* also presents a derived Boolean attribute (*irrigate*) that states whether to irrigate considering the rainfall, temperatures, and soil moisture values [22]. *ClimaticConditions* defines the *join* operation to join all the data received from the five different source measures during the last 24 h (*i.e.* operation window): *Sum_Rain*, *Min_AirTemperature*, *Max_AirTemperature*, *Median_SoilMoisture_30*, and *Median_SoilMoisture_60*. For instance, *Sum_Rain* provides all the relevant details to deliver rain data in the required format. It relates to *ClimaticConditions* through an A_PIMGatheredAs association, and the *sumRain* property declares *rain_src* as its Source. □

*4.2.2. PIM STS operations*

Nevertheless, the IoT data structure only defines *what* variables the IoT application must provide, not *how* should it deliver them. Therefore, the PIM STS Operations (Fig. 5-B) allow representing **data transformations inside the IoT**.

IoT applications can generally define three main classes of operations: *sense*, *transform* and *send*. *Transform* operations diverge into three main classifications: aggregation, filter and conversion. *Sense* and *send* operations describe how the IoT objects gather and deliver data, respectively. These operations can be based on temporal and numerical policies to define frequency and windows. Our UML profile represents these methods as operations (Fig. 5-B) of the classes previously described. In particular, A_PIM_Aggregate, A_PIM_Convert and A_PIM_Filter are used for *transformation*; A_PIM_TupleDelivery and A_PIM_TimeDelivery for *send*; and A_PIM_SensorGathering for *sense*. In our case study, the variables are not only sensed and sent. They require specific transformations such as the median aggregation (A_PIM_Aggregate) or a change of units operation (A_PIM_Convert).

In detail, from the previous profile [16], we keep the stereotypes for aggregating and delivering the IoT data: A_PIM_Aggregate, A_PIM_TupleDelivery and A_PIM_TimeDelivery. These stereotypes allowed for simple models with a clear representation of sent data and some transformations. However, only the delivery operations remain unchanged in STS4IoT since it has a different way to compute data (including aggregation). In the following, we describe these transformation operations.

***Transform.*** A_PIM_Aggregate provides a single summary statistic from a set of samples of one variable using an aggregation operation. The Aggregation tag defines the function that summarises the variable (*e.g.* maximum, average). The AggregateBy tag defines a ‘‘group by’’ to aggregate the samples in time and space. This Operation transforms the original sensed data, and thus it is an A_PIM_Transformation.

Other transformations in our profile are A_PIM_Convert and A_PIM_Filter. The first one changes the original data format (*e.g.* the units), while the second one selects a relevant part of the samples, discarding the rest. For instance, A_PIM_Filter Boundaries drops the data samples below or above the limits (Minimum and Maximum).

To define a different kind of transformation, IoT experts should extend A_PIM_Transformation with a new operation stereotype.

Transforming the data is optional since the application may also require raw data. Besides, a single variable may have multiple transformations applied. Nevertheless, all A_PIM_Transformations must declare the ExecutionOrder tag since the order might affect the result. In this way, STS4IoT allows defining the different computations of each variable to express tailored operations according to the business users’ needs.

In this way, STS4IoT allows representing IoT data computation. However, the execution of these operations requires a periodicity and a window. Besides, the IoT data sensing and sending must follow a particular logic. We thus present the transformation-execution, delivery, and sensing operations.

***Send.*** The operation stereotypes A_PIM_TupleDelivery and A_PIM_TimeDelivery allow defining how often the IoT should provide the required data as a stream. Period states the periodicity of such streams in terms of tuples or time. To precise the time granularity, A_PIM_TimeDelivery uses the Granule tag. Moreover, the IoT should execute the

transformations with the same periodicity using the WindowSize tag to define the total data samples to process. WindowSize has the same granularity as Period, whether tuples or time. These stereotypes are compulsory; yet, only the main class (A_PIM_Measure) should define them. To express the execution of transformations in source measures (*i.e.* inside the IoT), we provide the new A_PIM_Internal_TupleDelivery and A_PIM_Internal_TimeDelivery stereotypes. Therefore, models can define when the IoT must transform the data and how and when it must send the (transformed) data.

***Sense.*** STS4IoT represents the process of sensing data with the stereotype A_PIM_SensorGathering, which defines each variable's sampling rate. It uses the Period and Granule tags to express the periodicity of each sample in terms of time. This Operation is conditional since only source measures can use it, not the main class.

**Example.** In the PIM of our running IoT example (Fig. 6), we present some examples of operations. Every source measure defines one *sense* Operation using A_PIM_SensorGathering to *sense* the required data. Nevertheless, they have different sensing rates. While it samples rain (*Sum_Rain*) every hour, it must sample temperature (*Min_AirTemperature* and *Max_AirTemperature*) every 20 min. These classes also define data *transformations*. For instance, *Median_SoilMoisture_30* defines two: Firstly, it aggregates the data with the *median* operation. Secondly, it converts the median value to centibar with the *toCbar* operation. Besides, *Sum_Rain* uses *filter* to keep only the sensed data values between 0 and 500. Finally, to *send* the sensed and transformed data, *ClimaticConditions* uses the *send* operation to deliver a stream of data every 24 h, operating the collected data of the last 24 h. Similarly, *Max_AirTemperature* defines an internal *send* operation to transform the sampled data of the previous 24 h every 24 h. □

To finalise our PIM profile, we added a set of rules in Object Constraint Language (OCL) to secure the definition of well-formed and semantically-coherent instance models. These constraints avoid empty classes and determine which operations are compulsory, conditional, optional or restricted for each class. For example, Fig. 7 shows two rules that define which delivery Operation belongs in each class (main or source). Similarly, Fig. 8 shows the OCL that restrict the sensing in A_PIM_Measure and control its use in
A_PIM_Source_Measure.

Finally, as stated in Section 1, designers can also define OCL constraints over the instance models to set some data quality characteristics [47]. For instance, a first simple rule can state that the node must provide at least one value of air temperature (for completeness criteria) expressed in OCL (first rule in Fig. 9). Also, a second rule for consistency can state that the maximum air temperature must be above 0 degrees Celsius (second rule in Fig. 9).

```
context A_PIM_Measure inv deliveryOperationRequired:
(self.ownedOperation->select(o|o.oclIsKindOf(A_PIM_TupleDelivery))->size()=1)
```

```
context A_PIM_Source_Measure inv internalDeliveryOperationOnly:
(((self.ownedOperation->select(o|o.oclIsKindOf(A_PIM_Internal_TupleDelivery))->size()=1)
or
(self.ownedOperation->select(o|o.oclIsKindOf(A_PIM_Internal_TimeDelivery))->size()=1))
and
(self.ownedOperation->select(o|o.oclIsTypeOf(A_PIM_TupleDelivery))->size()=0)
and
(self.ownedOperation->select(o|o.oclIsTypeOf(A_PIM_TimeDelivery))->size()=0))
```

**Fig. 7.** OCL rules for the delivery operation in A_PIM_Measures and A_PIM_Source_Measures.

```
context A_PIM_Measure inv sensingBanned:
(self.ownedOperation->select(o|o.oclIsKindOf(A_PIM_SensorGathering))->size()=0)
```

```
context A_PIM_Source_Measure inv sensingRestriction:
(self.ownedOperation->select(o|o.oclIsKindOf(A_PIM_SensorGathering))->size()<=1)
```

**Fig. 8.** OCL rules for A_PIM_SensorGathering.

```
context maxTemp_src inv sensedMaxTempRequired:
Not (self.oclIsUndefined())
```

```
context maxTemp_src inv minimumMaxTempValue:
(self > 0)
```

**Fig. 9.** Example OCL rules for data quality.

Although these quality constraints on IoT data are out of the scope of the paper, these examples show that OCL could successfully express them in our UML profile.

Besides, particular data structures could offer an alternative for such quality rules. For example, our case study application could define two A_PIM_Source_Measure classes for each variable to reduce the amount of erroneous data. The first would define the sensing, and the second would define the transformation to select the most accurate value (*e.g.* median). The relation between the two classes should state the number of redundant nodes.

### *4.3. STS4IoT device model*

Before describing the PSM, we present the catalogue for the IoT software and hardware facilities (*i.e.* the DM).

The DM attempts to model the principal facilities of each implementation IoT platform. In this way, multiple DM instances can provide a repository of all the available IoT devices and their capabilities. This conceptual catalogue allows IoT experts to easily find and use the best-fitting device to meet the specific requirements and constraints of the application. It specifies the definition of PIM and PSM variables, PSM operations, and other implementation details in the PSM. In other terms, IoT experts should only ‘‘pick’’ from the DMs when defining PIM and PSM models to specify the available variables at conceptual and implementation levels, respectively (as described in Section 4.1).

Such a simplified abstract representation of IoT facilities eases the definition of the models while also enabling the automatic generation of implementable code. For the sake of simplicity and keeping a data-centric approach, the STS4IoT DM only considers three groups of facilities for a Device (Fig. 10):

- DeviceData represents the data variables that a node can handle. It has an abstract layer and an implementable layer. The first layer (D_PIM_Data) is for the PIM. It creates a map for the further implementation of each D_PIM_Variable with one or more D_Values. The implementable layer (D_PSM_Data) is for the PSM. It contains relevant information about how to sense those variables and their format. Each D_PSM_Variable must relate to a D_PIM_Variable through a D_VariableOfKind generalisation and define the Units. Besides, if the particular device can sense that variable, the variable should define a D_PSM_Sense Operation and belong to a D_PSM_Probe Package. A device can contain data it cannot sense in two cases: First, when it received it. Second, when an existing data operation (conversion) changes the platform data type or unit, and thus a new implementable variable should represent such change without changing the source sensing operation.

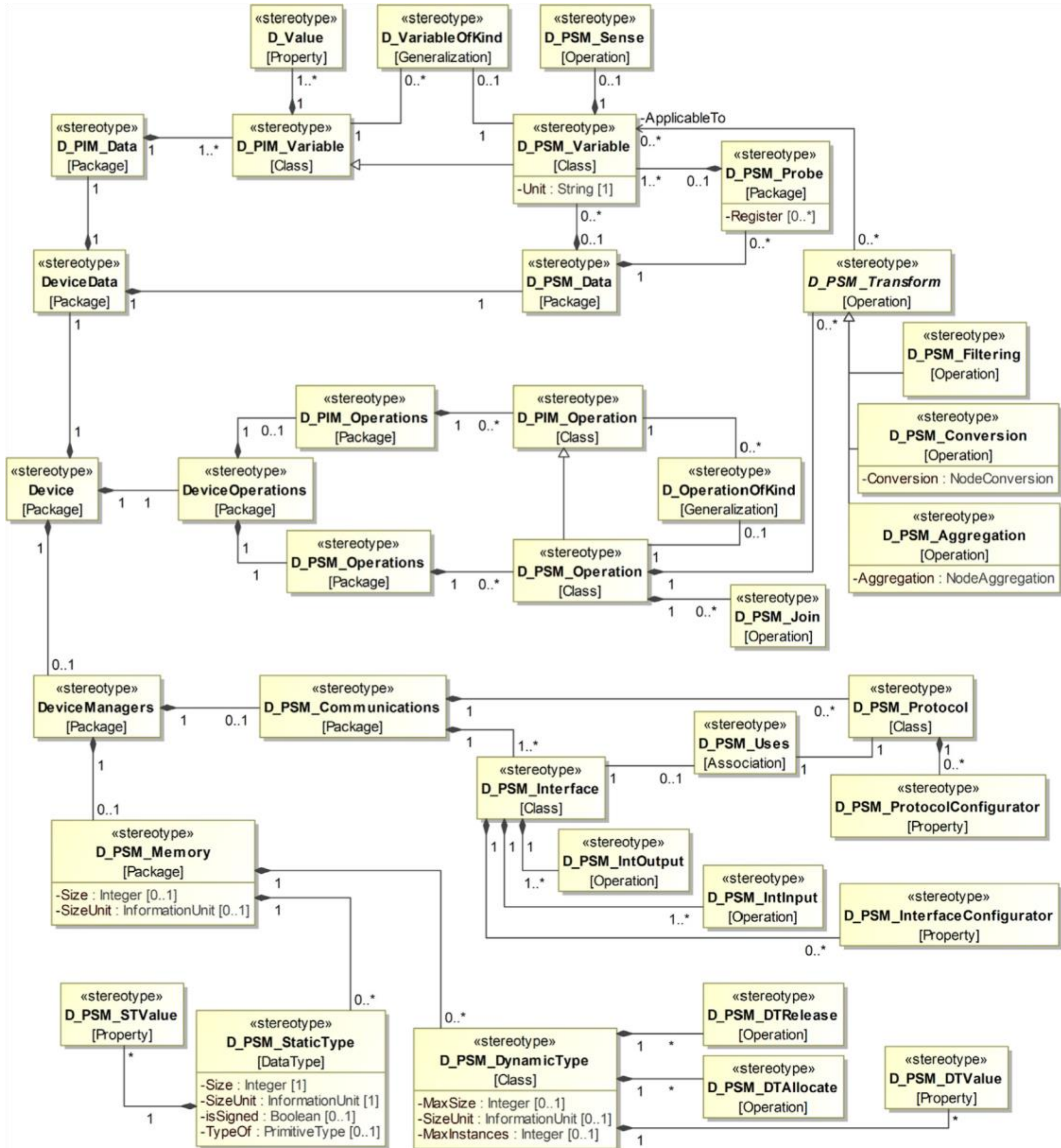


**Fig. 10.** STS4IoT DM Profile.

- DeviceOperations represents operations on data that a node can successfully execute. It has an abstract layer for the PIM

  (D_PIM_Operations) and an implementable layer for the PSM (D_PSM_Operations) as DeviceData. While the implementable layer is mandatory and used in the definition of the PSM, the abstract layer is optional and not used in the PIM. Indeed, every transformation (A_PSM_Transformation) or join (A_PSM_Join) in the PSM must relate to an existing operation in the corresponding DM using the DefinedAs tag to be implementable. Such operations are grouped in D_PSM_Operation classes according to their type: join operations (D_PSM_Join), filter transformations (D_PSM_Filtering), conversion transformations (D_PSM_Conversion) or aggregation transformations (D_PSM_Aggregation). The DM of a device can include these operations only if the node can

execute them. Therefore, these operations have a specific mapping in code that enables their automatic implementation. In this way, the profile force IoT experts to select appropriate devices for their PSM using the DM catalogues.

- DeviceManagers are other relevant hardware characteristics required for an appropriate code generation. To reduce the complexity of our profile, we have only included those facilities that are strictly necessary for generating the code: communications (D_PSM_Communications) and memory (D_PSM_Memory). The communications module contains the node

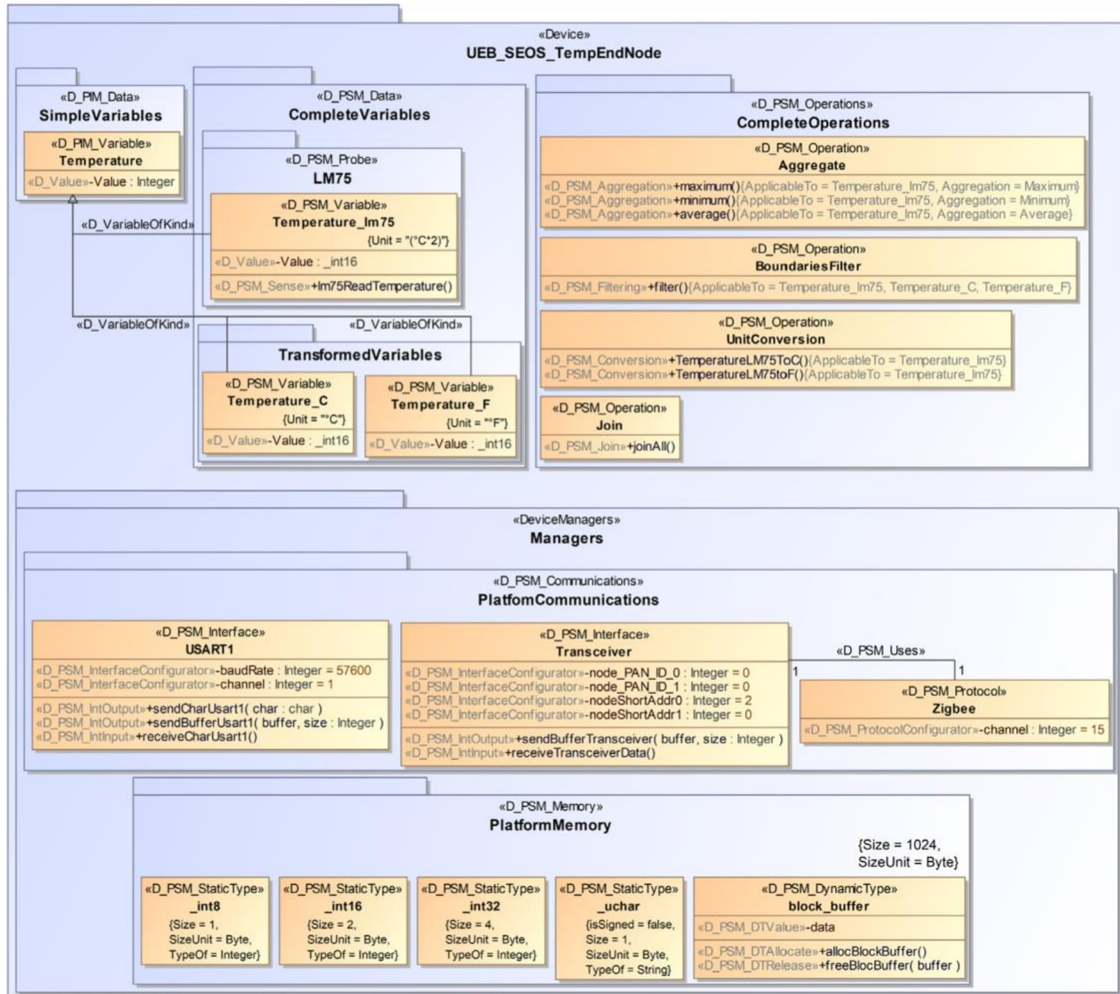


**Fig. 11.** DM model for the temperature-sensing end-node, used by the *Temperature_SensorNode* in Fig. 14.

interfaces (D_PSM_Interface). They allow sending and receiving data using D_PSM_IntOutput and D_PSM_IntInput operations, respectively. Besides, it can define multiple configuration options through D_PSM_InterfaceConfigurator attributes. Moreover, some interfaces should define the communications protocol (D_PSM_Protocol), configuring it with
D_PSM_ProtocolConfigurators.

The memory module (D_PSM_Memory) models the data types used by the embedded OS in the applications code. These types can be static (D_PSM_StaticType) or dynamic (D_PSM_DynamicType). Static types do not change their size in bytes even if their value changes, while dynamic types change their size according to the

memory requirements *e.g.* data buffers. The latter requires two operations, one to reserve the memory span (D_PSM_DTAllocate) and another to free it (D_PSM_DTRelease).

**Example.** Fig. 11 shows the DM for the temperature-sensing hardware platform (*UEB_SEOS_TempEndNode*) used in our running example. In *SimpleVariables*, *Temperature* is an abstract variable with little relation to the device. While in *CompleteVariables*, *Temperature_lm75* is an implementable variable that defines the sensing operation, the platform variable type, and the units. Besides, *Temperature_C* and *Temperature_F* represent the temperature in degrees Celsius and Fahrenheit for the *TemperatureLM75ToC* and *TemperatureLM75ToF* conversions, respectively. Moreover, the example node can communicate through *USART1* and *Transceiver* using *Zigbee*. Finally, every implementable variable such as *Temperature_lm75* uses one of the static data types of the OS (*_int16*). □

### *4.4. STS4IoT PSM*

This section addresses the low-abstraction level of STS4IoT (the PSM), which describes the essential implementation issues of the IoT application, providing an implementable yet simplified representation of the IoT application.

Specifically, the PSM leverages the DM to represent additional details to the PIM data design that enables the implementation. For example, hardware sensors, interfaces, and nodes communications and roles. The physical and implementation details of an IoT application are not trivial [48]. In particular, the definition of the node roles, tasks distribution, and deployment topology can vary significantly from one implementation to another.

For instance, the PIM of our running example might have different possible implementations depending on various conditions and constraints:

- *Deployment conditions*: a single node can monitor a small space (*e.g.* a room). Big open spaces will require several sensing nodes for effective monitoring (*e.g.* a farm). Besides, the nodes could directly upload the data to the cloud in places with internet access. Otherwise, they would require a sink node or a gateway.
- *Hardware constraints*: Specific hardware limitations (*e.g.* processing capabilities, energy supply, communication interfaces, memory) will affect the implementation design. For instance, if an end-node cannot compute the data, the next level (sink node or server) should perform the operations. Besides, even if the nodes are close to a suitable wireless Internet connection, they will not connect to the network if they do not have the required interface.
- *Application constraints*: The specific application could also impose some restrictions on the implementation. Moreover, some applications require high accuracy and failure control, while others can accept a lower precision. For example, critical realtime applications such as fire detection require low latency and failure control. Thus, multiple good-quality end-nodes should analyse the data in-situ to provide reliable alerts.
- *Processing conditions*: Some specific transformations (*e.g.* spatial aggregation) might require a particular topology and task distribution. For example, calculating the average temperature of a large field may require multiple temperature-sensing end-nodes sending raw data to one sink-node in a star topology.

Consequently, we have also defined a DM that allows identifying most of the hardware constraints of the available devices. The DM acts like an API for the PSM. It provides a catalogue of all the relevant IoT device facilities (*e.g.* available hardware sensors, memory and computation capabilities, and communications interfaces). Besides, this model aims to provide a map between the PSM and the actual code functionalities of the IoT devices.

When the IoT experts look into the DM catalogue of our running example, they notice there are seven kinds of devices available (*cf.* Fig. 1):

- A transformation Sink Node ($SN^1$) that does not sense any variable; but can receive, join and transform data from other nodes and send them to the cloud.
- A simple Sink Node ($SN^2$) that does not sense or transform any variable. It can only receive and join data from other nodes and send them to the cloud.
- A complete Sink Node ($SN^3$) that can sense and transform Temperature and Rain. Also, it receives, joins and computes data from other nodes; and sends these data to the internet.
- An advanced Sink Node ($SN^4$) that can sense and transform Temperature and Rain, receive and join data from other nodes (though not to transform them), and send them to the internet.
- Three End Nodes (EN), each of which can sense one of the variables (Temperature, Rain or Soil Moisture), transform and send the data to the Sink Node (but not to the internet).

Also, IoT experts know that the farm fields have internet access in one specific location only. With this information, they define four implementation options:

(1) Fig. 1-A Use the three EN to sense the required variables in the plot; and the $SN^1$ to receive, transform, join and send the data to the cloud. This option is better from the agronomic side since it gathers each variable from the most representative point of the plot. However, it is more expensive since it uses multiple devices and has high battery consumption. Indeed, sending data is the most consuming energy operation.

(2) Fig. 1-B Use the three EN to sense and transform the required variables in the plot; and the basic $SN^2$ to receive, join and send the data to the cloud. This configuration has the same agronomic advantages as option A. Besides, it lowers the battery consumption by transforming the data inside each EN and reducing the sending operations [23].

(3) Fig. 1-C Use only one EN to sense Soil Moisture in the plot; and the complete $SN^3$ to sample and transform Temperature and Rain, receive and transform the Soil Moisture, and join and send all data to the internet. This option is cheaper since it only uses two devices and reduces battery consumption. However, it is not ideal from the agronomic side since the internet-access point might not represent the whole plot.

(4) Fig. 1-D Use one EN to sense and transform Soil Moisture in the plot; and the advanced $SN^4$ to sample and compute Temperature and Rain, receive the transformed Soil Moisture, and join and send all data to the internet. This configuration further reduces the battery consumption in the EN since it must only deliver transformed data [23].

Business users and IoT Experts select option 4 (Fig. 1-D), considering that the SN point is enough to acquire the temperature and rainfall data of the whole plot. Therefore, IoT experts use the PSM part of our STS4IoT profile (Fig. 12) to define the selected implementation option in the UML formalism (Fig. 13). To better understand the PSM profile, we have decomposed it into two components: data structure 4.4.1 and STS operations 4.4.2.

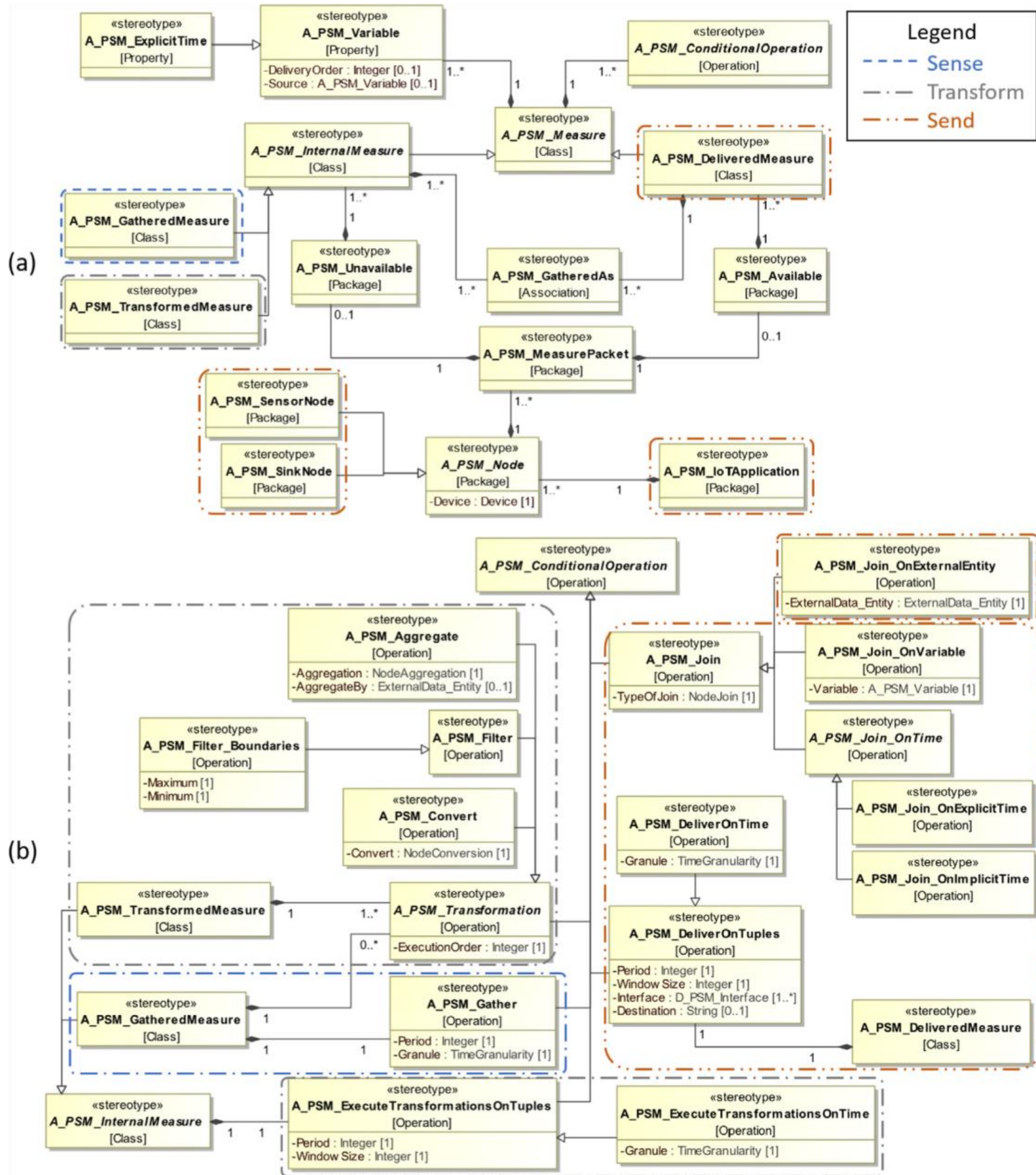


**Fig. 12.** STS4IoT PSM profile data part (a) and STS part (b). Dashed blue lines are for the *Sense* stereotypes, dash-dotted grey lines for the *Transform* stereotypes, and dashed double-dotted orange lines for the *Send* stereotypes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

### *4.4.1. PSM data structure*

The PSM data structure (Fig. 12-A) keeps almost all the original stereotypes from [16], since they provide a fair initial approach to STS for single nodes. The new stereotypes and tagged values improve that approach and model **data communications between IoT objects**, which allows defining multiple cooperative nodes with their respective roles to cope with the ***communication*** *capabilities* of STS.

In particular, the PSM maps the PIM data over the different nodes used. Therefore, the PSM must represent how each node senses the data, transforms them, and sends data to other nodes. Indeed, as previously described, an end node could sense only single or multiple variables and send them to other nodes that transform them or merge them

with data from different nodes. These particular transformation operations and communications among the nodes are intentionally kept transparent at the PIM level since the end-user must not handle their implementation details, addressed by IoT experts only. Therefore, to address

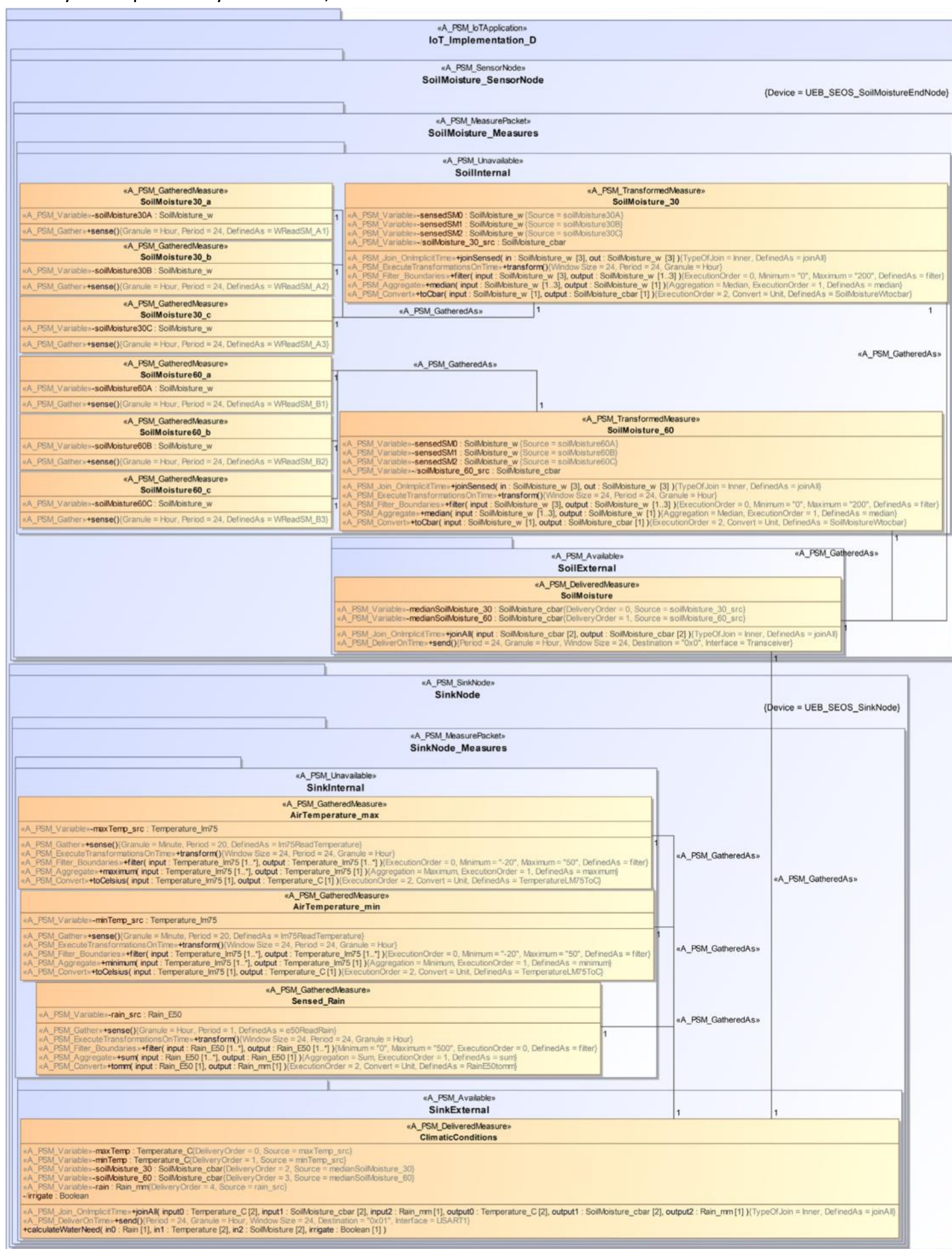


**Fig. 13.** Possible PSM for the PIM of the running example (Fig. 6) considering the configuration of Fig. 1-D.

computation and communication technical details underlying the PIM models, the PSM is structured in the following main classes:

A_PSM_GatheredMeasure, A_PSM_TransformedMeasure and A_PSM_DeliveredMeasure. They represent the sensed value of a node, the transformations applied by a node (mainly on received data), and the communication towards another node, respectively. For instance, in our case study (Fig. 13), the soil moisture values sensed at 30 cm by the EN are represented by three A_PSM_GatheredMeasure classes. One A_PSM_TransformedMeasure makes the calculus of the median value. And one A_PSM_DeliveredMeasure models the data sent to the sink node.

In the following, we provide the details of the PSM data structure meta-model.

The core stereotype of the PSM data structure is A_PSM_Measure, which will represent any set of variables sensed, transformed, or sent by one IoT node. An A_PSM_GatheredMeasure or an A_PSM_TransformedMeasure, should represent a node when it senses or transforms the data, respectively. Nevertheless, while A_PSM_GatheredMeasure can also represent transformations on its sensed data, A_PSM_TransformedMeasure should only represent the transformation of received or sensed data. Both stereotypes are generalised as A_PSM_InternalMeasures, which are available only inside the particular IoT node and thus belong to an A_PSM_Unavailable package. With this change, although a node can receive and transform data from other nodes, it cannot sense and transform its data.

Moreover, A_PSM_DeliveredMeasure represents the data sent by each IoT node. This data must be available outside the particular node and thus belong to an A_PSM_Available package. Since an IoT node can only send data already owned, being sensing, or received. Hence, it must be related to A_PSM_InternalMeasures in the same node or A_PSM_DeliveredMeasures of other nodes. Besides, the delivered variables must define the DeliveryOrder and Source tags, which respectively indicate the order to send the variables and the original related variable inside or outside the node. In this way, STS4IoT allows identifying the data (and operation) flows in the whole IoT application.

Indeed, as for PIM, all the join operations are also defined at the PSM level. In particular, a join operation merges both the sensed and received data inside the IoT node. It also allows sending a single data packet regardless of their origin or memory location. A_PSM_Join is the basic stereotype for this Operation; it provides the TypeOfJoin tag to define the use of an inner or full outer join. The join operation usually uses a common characteristic to merge the data. Thus, this stereotype has four implementable specifications for different attributes: A_PSM_Join_OnExternalEntity to combine the data linked to the same ExternalData_Entity. A_PSM_Join_OnVariable to join data that has the same value in a variable. A_PSM_Join_OnExplicitTime to merge data that has the same A_PSM_ExplicitTime value. And A_PSM_Join_OnImplicitTime to integrate all the data inside the window (defined in the sending or executing Operation).

Both the available and unavailable measures of a node also belong to an A_PSM_MeasurePacket package. It represents a single output stream of the node and should only contain one A_PSM_DeliveredMeasure class. This package helps organise the structure of complex nodes to ease the automatic code generation process (*c.f.* Section 5).

STS4IoT represents each IoT node as one A_PSM_Node package, using the Device tag to define the particular hardware platform for implementation (selected from the DM catalogue). The specification of A_PSM_Node declares the role of the node:

A_PSM_SensorNode represents end devices that sense, transform, and send data but cannot receive them. While A_PSM SinkNode defines the gateways that can sense or receive data from end nodes to compute and upload them to the internet. Finally, A_PSM_IoTApplication represents the IoT application that groups all the involved nodes, thus enabling the modelling of complex IoT data-sensing applications requiring multiple nodes with different roles.

For automatic implementation purposes, all involved nodes and variables must explicitly appear in the A_PSM_IoTApplication. Even though several nodes have the same behaviour and hardware platform, each node requires one package.

**Example.** Fig. 13 presents the PSM of our running IoT example considering the PIM and implementation option D (Figs. 1-D and 6). *IoT_Implementation_D* represents the whole IoT application. It contains both the EN (*SoilMoisture_SensorNode*) implemented in a *UEB_SEOS_SoilMoistureEndNode* device and the SN (*SinkNode*) implemented in a *UEB_SEOS_SinkNode* device.

In *SinkNode*, the *SinkInternal* package groups the data sensed or transformed by this node such as temperature, which is represented by *AirTemperature_max* and *AirTemperature_min*.

Besides, the *SinkExternal* package contains the data sent from the *SinkNode*, which is represented by *ClimaticConditions*. This class provides all the variables defined by IoT experts, sensed temperature and rain, and received soil moisture. *maxTemp* is the first delivered variable and comes from an internal measure, while *soilMoisture_30* is the third delivered variable and comes from the delivered measure of a different node. Indeed, *SinkNode* uses a A_PSM_Join_OnImplicitTime operation to join its sensed data with the data received from *SoilMoisture_SensorNode*. □

*4.4.2. PSM STS operations*

Furthermore, the PSM model must also define exactly *how* IoT nodes *sense, transform and send* data, relating to the specific DM to ease the automatic **implementation** of the model.

Already PIM STS operations define how the IoT objects acquire, compute and deliver their different variables. Nevertheless, multiple questions remain unsolved before addressing the implementation: ''*What probe is sensing data*?'', ''*Do the device support defined transformations*?'', and ''*What interface will send the data*?''. Therefore, PSM STS operations define different associations with the DM to explicitly answer these questions. For instance, the DefinedAs association of A_PSM_Gather and A_PSM_Transformation with D_PSM_Operation, and the Interface tag in A_PSM_DeliverOnTuples help answering these questions.

In our case study, LM75 probes senses temperatures, in the DM are defined their transformations, and through *USART1* are sent sink-node data.

We have defined 17 stereotypes to represent these operations on IoT data (Fig. 12-B). All operations in our profile have conditions that allow or restrict their use in certain cases, and thus they generalise into A_PSM_ConditionalOperation.

The operations already defined in [16] are **sense** (A_PSM_Gather), and **send** (A_PSM_DeliverOnTuples and A_PSM DeliverOnTime).

***Sense.*** A_PSM_Gather allows sampling data from probes with a time periodicity defined with the Period and Granule tags. This operation is mandatory and constrained to A_PSM_GatheredMeasure classes.

***Send.*** A_PSM_DeliverOnTuples and A_PSM_DeliverOnTime allow sending data to a destination (defined with the Destination tag) using particular communications interface (defined with the Interface tag) with certain periodicity. For A_PSM_DeliverOnTuples, the Period is defined as the number of tuples the node senses before sending. For A_PSM_DeliverOnTime, the Period is defined in terms of time with the additional Granule tag. Besides defining the data sending, these operations also define a WindowSize, which indicates (in terms of tuples or time, respectively) the operation windows for joining the data. These sending operations are mandatory and constrained to A_PSM_DeliveredMeasure classes.

***Transform.*** As described in the PIM section, the IoT application may require to make some transformations to the data to provide it as demanded by end-users (*e.g.* Fig. 6) and meet the STS requirements. A_PSM_Transformation represents all the possible transformations in our profile, which must define the ExecutionOrder tag to set the order on which the IoT node runs each transformation. Transformations are constrained to A_PSM_GatheredMeasures and A_PSM_TransformedMeasures, being mandatory in the second ones.

STS4IoT initially defines three basic data transformations:

- A_PSM_Aggregate calculates summary statistics of the sampled or received data, reducing the amount of data to transmit and increasing its value. These operations define two tags: Aggregation to define the operation (*e.g.* average, maximum), and the AggregateBy to group the data to operate when making spatial aggregation.
- A_PSM_Convert, which changes the format or presentation of the data, *e.g.* converting units.
- A_PSM_Filter, which attempts to drop potentially wrong or misleading samples. For instance, A_PSM_Filter_Boundaries drops samples that have values above the Maximum or below the Minimum permitted values.

Even though this is a basic yet powerful set of transformations, the UML profile is easy to upgrade with new transformation operations. In particular, the IoT expert should add a couple of new operation stereotypes specifying both A_PSM_Transformation in the PSM and A_PIM_Transformation in the PIM.

Nevertheless, IoT nodes handle data streams and thus need a window to run these transformations. Therefore, we provide the execution operations A_PSM_ExecuteTransformationsOnTuples and A_PSM_ExecuteTransformationsOnTime in STS4IoT. These operations define the windows for effectively transforming the stream data and the periodicity to execute such transformations. They are only permitted and mandatory when there is any A_PIM_Transformation in the same class.

**Example.** In the PSM of our running IoT example considering implementation option D (Fig. 13), the A_PSM_GatheredMeasures of *SoilMoisture_SensorNode* define *sense* operations to read the soil moisture every 24 h. Moreover, their DefinedAs tags state that each A_PSM_GatheredMeasure sense data from a different probe.

In the *SinkNode*, all A_PSM_GatheredMeasures transform their data and thus define execution operations. For instance, *AirTemperature_min* defines the *minimum* aggregation operation implemented as the DM operation *minimum* to calculate the lowest sensed value of temperature before converting its units. *AirTemperature_max* defines the conversion operation *toCelsius* implemented as *TemperatureLM75ToC* to transform the data from the units of the particular sensing probe (LM75) to degrees Celsius after aggregating the sensed values. Also, *Sensed_Rain* defines the *filter* operation implemented as *filter* to keep only samples between 0 and 500 before aggregating or converting the data. All these classes define a *transform* operation to transform the data from the last 24 h every 24 h.

Finally, *ClimaticConditions* defines a *send* operation that sets a 24-h window and a periodicity of 24 h to execute a join of all the data in the IoT and deliver them through the *USART1* interface (*cf.* Section 4.3). □

The elements of the STS4IoT PSM profile allow modelling implementable IoT data-gathering applications considering the STS characteristics. Furthermore, IoT experts can consider and use the PSM profile to design different implementation options for the same application (Section 2).

**Example.** Fig. 14 displays the fragment of a PSM model to implement the deployment option in Fig. 1-B, which also meets the application requirements defined in the PIM (Fig. 6) but provides higher spatial reliability.

As shown in Fig. 1-B, the PSM of Fig. 14 has four devices. The exposed devices are: *SinkNode* ($SN^2$), which receives the data from the other nodes, joins the variables and upload them through the *USART1* interface every 24 h. And *Temperature_SensorNode* (using the DM *UEB_SEOS_TempEndNode* of Fig. 11), which provides the sink node with the maximum and minimum air temperatures in degrees Celsius.

The hidden devices (do not appear in Fig. 14 but exist in the PSM) are *SoilMoisture_SensorNode* and *Rain_SensorNode*. The first provides the median soil moisture in centibars at 30 and 60 cm to the sink node, while the second sends the total (sum) rain in millilitres to the sink node every 24 h through the *Transceiver*. The sending period, granule, and interface are the same for the three end nodes. □

## 5. Implementation

This section explains the use of *STS4IoT Model-to-Code Tool* to automatically implement STS4IoT instance models in the IoT platform developed by LIMOS (UMR 6158 CNRS - Université Clermont Auvergne, France): uSu EDU Board (UEB) [16]. Applications

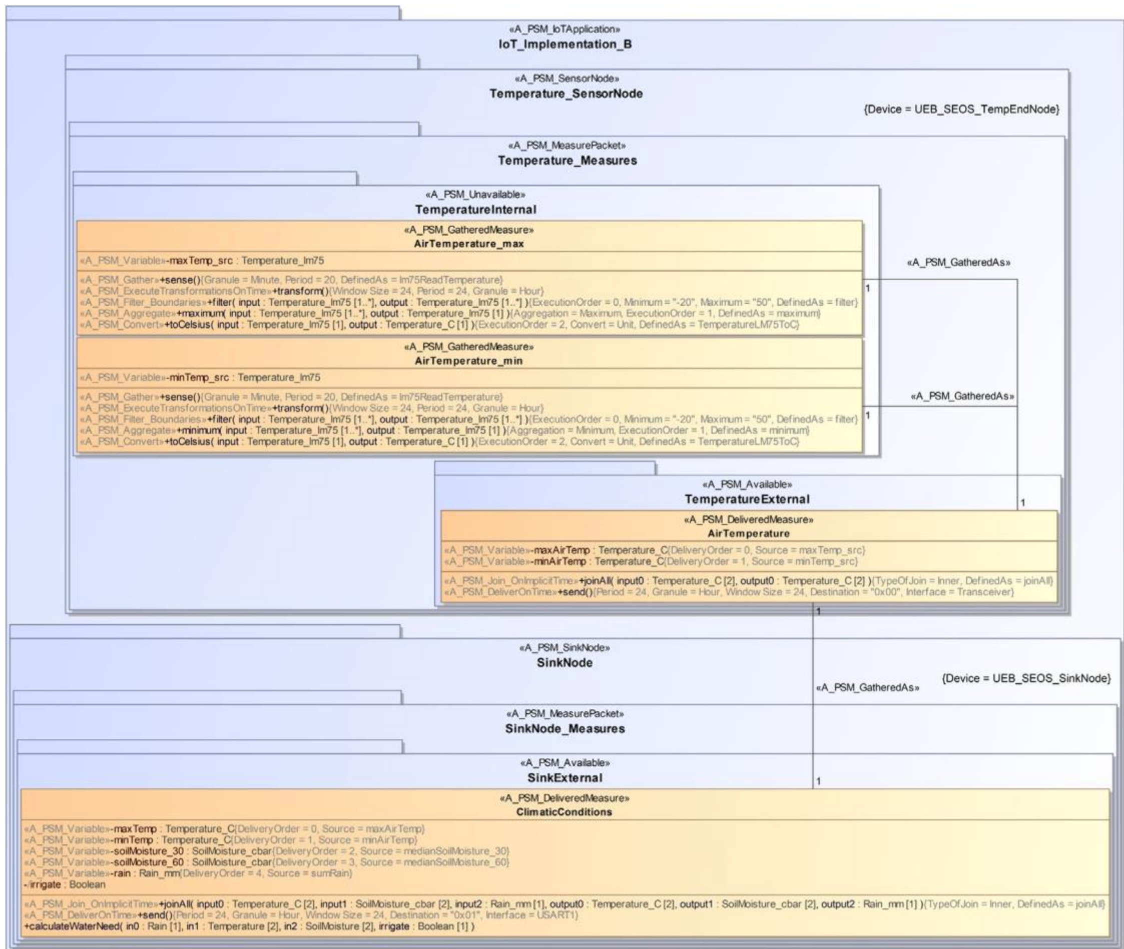


**Fig. 14.** Fragment of another possible PSM for the running example considering the configuration of Fig. 1-B (shows only the Temperature EN and SN[2]).

for UEB are written in C and consist of two parts: programming logic in a *application.c* file; and constants and data types in a *application.h* file. [16]. Each UEB device in the IoT application requires its own set of code files.

Fig. 15 illustrates the complete implementation process.

In the first place, business users and IoT experts define the IoT application PIM and PSM models using STS4IoT [21]. We use the CASE tool MagicDraw® for this step. The MagicDraw® allows you to graphically build models and save them in XML Metadata Interchange (XMI) format.

Second, we developed an XMI parser in Python 3.7 that reads all elements of the PSM model saved in the XMI format to identify the data, operations, and associations of the entire PSM model. The associations to the data stream from the sensing end-nodes to the loading sink-node.

This desktop script leverages the XML Element Tree [49] library to access the XMI data in a structured manner and the DateTime [50] library to generate code and output messages with timestamps. It follows the Object-Oriented paradigm to have independent modules for reading the models, organising the information, and generating the IoT code. Thus, its extension to different IoT platforms is simpler.

In the first step, the *STS4IoT Model-to-Code Tool* reads the UML model in XMI and parses all the packages for end-nodes before parsing the sink-nodes. It identifies for each (end and sink) node:

(1) The sensed data types and their sensing frequency, the transformations operations, and the transformation frequency;
(2) The received data types and their transformations inside the node;
(3) The data to send from the node, considering the temporal join operation (based on [46]), with sending frequency and data destination.

In the second step, the *STS4IoT Model-to-Code Tool* uses this information to define the required constants and data types for the data structure of each node. Then, it organises the operations (sensing, transforming, and sending) to generate modular threads for each item in the node logic.

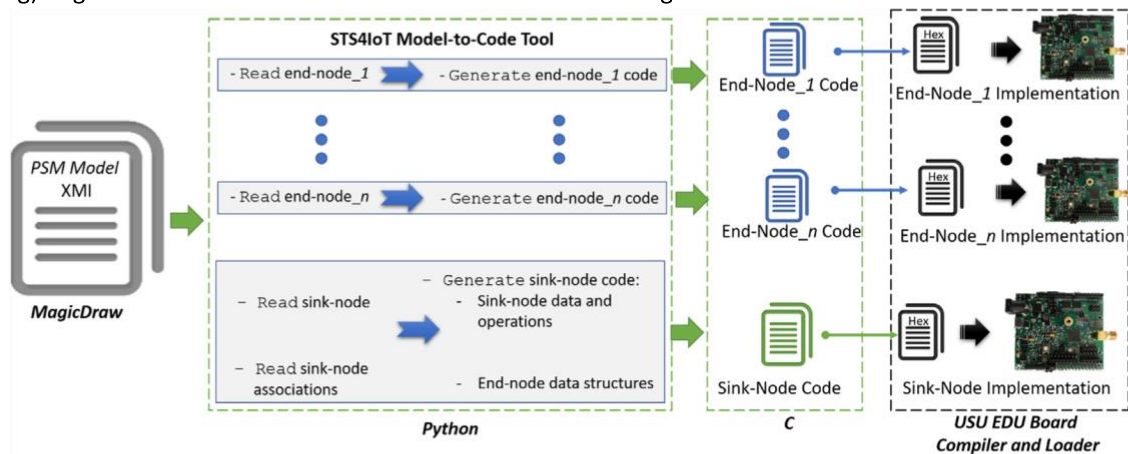


**Fig. 15.** Implementation process in the uSu EDU Board [16] using the STS4IoT profile and model-to-code tool.

Sink Node **application.c** Fragment

```
_uint8 receiverThread(_uint8 num_action, _uchar* data, _uint8 size)
{
    switch (num_action)
    {
        case 0 :
            {
                receivingBuffer = allocBlockBuffer();
            } break;
        case 1 :
            {
                received_data* measures = (received_data*)(receivingBuffer);
                if (data[0] == 1) {
                    received_data_1* received = (received_data_1*)(data);
                    measures->data_from_1 = *received;
                } else if (data[0] == 2) {
                    received_data_2* received = (received_data_2*)(data);
                    measures->data_from_2 = *received;
                } else if (data[0] == 3) {
                    received_data_3* received = (received_data_3*)(data);
                    measures->data_from_3 = *received;
                }
                freeBlocBuffer((block_buffer*)data);
            } break;
    }
    return 0;
}
```

Sink Node **application.h** Fragment

```
/**     Data received from node 2       **/
typedef struct struct_received_data_2 {
    /***    TODO: Check all the data types!     ***/
    _uint8 nodeID;
    _int16 avgSoilMoisture_30_2[1];
    _uint8 avgSoilMoisture_30_2_lastplace;
    _int16 avgSoilMoisture_60_2[1];
    _uint8 avgSoilMoisture_60_2_lastplace;
} received_data_2;

/**     Data received from node 3       **/
typedef struct struct_received_data_3 {
    /***    TODO: Check all the data types!     ***/
    _uint8 nodeID;
    _int16 sumRain_3[1];
    _uint8 sumRain_3_lastplace;
} received_data_3;

/**     All received data       **/
typedef struct struct_received_data {
    /***    TODO: Check all the data types on each struct!  ***/
    received_data_1 data_from_1;
    received_data_2 data_from_2;
    received_data_3 data_from_3;
} received_data;
```

**Fig. 16.** Code fragments of $SN^2$ for receiving transformed data.

In the third step, the *STS4IoT Model-to-Code Tool* generates the code for each node based on their specific data structure and operations logic. In the case of UEB, our tool generates one *application.c* and one *application.h* file for every node in the IoT application.

For example, Fig. 16 shows fragments of both the *application.c* and *application.h* files generated by *STS4IoT Model-to-Code Tool* for $SN^2$ in the running example (Figs. 1-B and 14). At the top of Fig. 16, the fragment of *application.c* shows the thread that incorporates the received data into the sink nodes internal variables. In ''case 1'', the sink-node reads the first position of the received data to find the source end-node, stores the data using the appropriate structure, and finally frees the memory space allocated for the incoming data. In the bottom of Fig. 16, the fragment of *application.h* displays some of the data structures that allow storing the received data. For instance, the first structure in the fragment represents the data received from the second end-node, which senses and transforms the Soil Moisture at 30 cm and 60 cm, finally sending the median at each depth. The second structure is for the rain end-node that sends the sum. And the last structure allows storing the data from the three end-nodes using their respective structures.

Using a standard laptop, *STS4IoT Model-to-Code Tool* takes five seconds or less to generate the application files in this example.

Finally, the generated code files are ready to be deployed in the sensors: (i) IoT experts can use the UEB tools to compile the application code into hexadecimal files and load them into the corresponding devices; (ii) Or they could also directly load the code into the UEB simulator to test their logic before the final implementation.

## 6. Theoretical validation

In this section, we present a set of tests to validate our STS4IoT profile following different approaches in the literature [10,51–53]. Firstly, we verify the compliance of STS4IoT with the UML standard and its capacity to define well-formed instance models with a CASE tool [51]. Secondly, we assess the quality of the STS4IoT profile using the quality model of [52].

### *6.1. CASE tool validation*

For this test, we used the validation suite of MagicDraw®. Once it is defined the STS4IoT meta-model in the CASE tool, we checked its correctness and completeness in the UML standard [10,51] with the default tests provided in its suite: *Diagram Merge* to check if diagrams and symbols are correctly merged. *Numbering Validation* to check if the element number is unique. *Orphaned Proxies* to identify referenced elements that have changed or no longer exist. *Parameters Synchronisation* to check whether the definition of the model arguments is synchronised with the corresponding modelling parameters. *Relationship Ownership* and *Shape Ownership* to detect misplaced symbols that lead to erroneous interpretations of the graphical models. *Spelling* to check the orthography of the elements' names. *UML Completeness Constraints* to check if a model is complete, has no gaps and has its essential information fields filled. And *UML Correctness* to check the most important rules of the UML meta-model (Ports compatibility, Pin types compatibility, Slot and Tags multiplicity correctness, amongst others) [54]. The results of our profile in all these tests were positive (Fig. 17).

Furthermore, we checked whether the STS4IoT profile could produce well-formed instance models [51]. Hence, we prepared a set of corrupted PIM and PSM models, and correct PIM and PSM models following all the OCL rules described in Figs. 6, 13 and 14. MagicDraw® identified all the (intended) errors in the corrupted models (Fig. 18), while the good models were considered appropriate.

### *6.2. Meta-model quality assessment*

Since the empirical evaluation of the quality of our proposal is a complex task due to the need of finding out several IoT experts with skills in UML modelling and decision-makers in the context of real-world case study, such as the one about irrigation presented in Section 2, in this work we provide a theoretical assessment of our UML profile.

Therefore, we based our assessment on the quality model for meta-models proposed by [52] and implemented in [53]. This model defines five quality properties that can be calculated from the meta-models and allow assessing their quality considering different aspects:

- *Reusability* is the capacity of a meta-model to contribute with its constructs to the definition of new meta-models. It is positively related to the number of meta-classes, rules, and meta-attributes; but negatively linked to the associations and inheritance between meta-classes.
- *Understandability* is the ability of a meta-model to be easily perceived and instantiated by its users. It is positively associated with the number of meta-attributes and rules; and negatively linked to the number of meta-classes, their associations and inheritance, and the number and depth of inheritance trees.
- *Functionality* is the overall capacity of a meta-model to model complete and diverse models. It is positively related to the number of associations and inheritance between meta-classes, rules, meta-attributes, and concrete meta-classes.

- *Extendibility* is the degree of ease in adding new modelling elements to the meta-model. More are the meta-classes in a meta-model, higher is its extension score. More are the associations and inherits amongst its meta-classes, more difficult is to extend.

- *Well-structured* determines the quality of the underpinning architecture and its composing meta-classes. Thus, this property increases with the number of well-defined meta-classes (with rules, attributes, and related meta-attributes). In the same way, it lowers with the number of separate inheritance hierarchies and inter-associations amongst meta-classes.

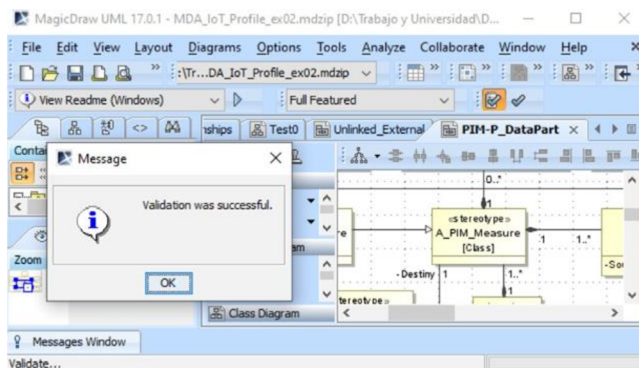


**Fig. 17.** Successful validation of the STS4IoT profile.

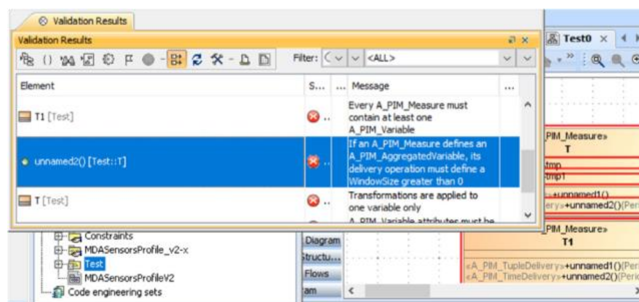


**Fig. 18.** Failed validation in the corrupted models.

In particular, we compare our proposal to other existing data-centric UML profiles and meta-models from multiple domains (not necessarily IoT) first. Then, we evaluate it against our previous work about IoT [16].

[53] used this quality model [52] to evaluate more than 2500 meta-models from the literature. Since Basciani et al. made their results available, we compared the results of STS4IoT with those from the literature using the percentile rank ( Table 2). This ranking allows us an estimation of the quality of our proposal.

Table 2 allows noticing that the *Understandability* and *Well-structured* of STS4IoT PIM and DM profiles are very good indeed. In the same way, even though these quality properties were lower in the PSM, it still ranks amongst the top 2%. These results further confirm the CASE-tool validation of its ability to define well-formed instance models. Therefore, their users can easily instantiate the STS4IoT profiles into proper application models, which allow for automatic implementation. In contrast, the *Functionality* is only above the median for the PIM, while the PSM is considerably low. Thus, STS4IoT can model a limited set of applications compared to other meta-models. This limitation is logical considering that we focus on IoT sensing applications only.

Besides, the *Reusability* of our PIM profile is good (amongst the top 25%). We could thus easily integrate it with other conceptual data models from different domains to design more complete applications (*e.g.* [21]). Regarding the *Extensibility* of STS4IoT, it does not allow for easy upgrading since the three profiles are under the median.

Nevertheless, we must note that the number of meta-classes highly increases *Extendibility* and *Functionality*. In this sense, our relatively small profiles cannot compete with the larger meta-models considered in [53], with an average of 28 meta-classes. On the other hand, this fact gives an advantage to the *Understandability* of our profiles.

Therefore, we can conclude that the quality of our STS4IoT profiles is appropriate in the aspects we considered as most relevant (*Well-structured*, *Understandability* and *Functionality*) considering its limitations. Besides, the PIM has a promising *Reusability* for its integration with other meta-models.

Furthermore, we also assess the upgrades in our profile in terms of quality. Table 3 presents the quality assessment [52] of each profile in STS4IoT: PIM, PSM and DM; comparing them with their previous versions of [16].

For the PIM evolution (Table 3), we evidence a stable *Understandability* and *Extendibility*; while *Functionality*, *Well-Structured* and *Reusability* increased. Thus, the PIM profile is slightly easier to use in STS4IoT due to better architecture. Also, it can represent a more variate range of applications.

In the PSM upgrade, *Extendibility* almost doubled, easing the addition of new transformation operations to our profile. However, the other attributes decreased due to the increased complexity.

Finally, the new DM is easier to understand and use with high *Understandability* and *Well-Structured* degree. Also, the range of possibilities in the design of devices is reduced, being positive for the automated generation of code.

With these results (Table 3), we evidence that the quality of the STS4IoT PIM profile has consistently improved, and the PSM and DM profiles met their goals at the expense of some of their quality attributes.

**Table 2**

Quality assessment results and percentile rank considering the meta-models evaluated in [53].

| Model | Reusability | | Understandability | | Functionality | | Extendibility | | Well-structured | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Profile Result | *Percentile Rank* | Profile Result | *Percentile Rank* | Profile Result | *Percentile Rank* | Profile Result | *Percentile Rank* | Profile Result | *Percentile Rank* |
| STS4IoT PIM | 7.35 | *78* | 5.50 | *99* | 4.60 | *69* | 0.30 | *11* | 6.80 | *99* |
| STS4IoT PSM | 3.18 | *56* | 0.62 | *98* | 1.88 | *19* | 1.34 | *37* | 1.66 | *98* |
| STS4IoT DM | 4.67 | *67* | 1.09 | *99* | 3.00 | *46* | 1.99 | *46* | 2.43 | *99* |

**Table 3**

Upgrade in terms of quality: Comparison between STS4IoT and [16].

| Model | | Reusability | Understandability | Functionality | Extendibility | Well-structured |
|---|---|---|---|---|---|---|
| PIM | STS4IoT | 7.35 | 5.50 | 4.60 | 0.30 0.30 | 6.80 |
| | [16] | 6.70 | 5.50 | 3.40 | | 6.40 |
| PSM | STS4IoT | 3.18 | 0.62 | 1.88 | 1.34 0.70 | 1.66 |
| | [16] | 3.53 | 1.77 | 2.27 | | 2.63 |
| DM | STS4IoT | 4.67 | 1.09 | 3.00 | 1.99 2.84 | 2.43 |
| | [16] | 5.24 | 0.51 | 3.36 | | 2.12 |

## 7. Observations from a real experience

To complete the theoretical quality assessment presented in Section 6.2, in this section, we report some observations issued from the usage of our UML profile over the real case study described in the previous sections. In particular, we put the design of PIM and PSM under test, analysing this process and the generated models (Figs. 6, 13 and 14) following the evaluation framework of [55].

In order to evaluate the instance of the proposed meta-model, we follow the quality assessment framework proposed by [55]. [55] propose a set of metrics to evaluate conceptual UMl models, grouped in three main classes: **Specification**,**Usage** and **Implementation**.

According to [55], **Specification** refers to a good structure and definition of the model. It is thus related to the *Understandability* and *Well-Structured* of the underlying meta-model. We measure it through *Legibility* and *Simplicity*. Besides, **Usage** refers to a good representation of the system under study from the user perspective. In this sense, it can be linked to the *Functionality* of the metamodel. It has two metrics: *Understandability* and

*Completeness*. Finally, **Implementation** is the amount of effort to implement and reuse the model. It has a slight association with the *Well-Structured* and *Functionality* properties of the meta-model, and the capacity of the automatic implementation tool. It is composed of *Implementability* and *Maintainability* [55].

*Legibility* (average of Clarity and Minimality) and *Simplicity* are issued from the models. These metrics range from 0 (extremely low) to 1 (perfect) [55]. *Completeness* and *Understandability* are assessed by the requesting user as accepted or rejected. *Implementability* is measured as the percentage of code automatically generated, and *Maintainability* as the relative time reduction in the modification of an existing version until a new requirement is satisfied [16,56]. We also count the number of iterations until the model is accepted. Finally, to measure the coding time, we consider only the time to configure the tool, generate and organise the code files, and compile it. We do not include the simulation time since it can take at least two days, which is longer than the rest of the process.

A Ph.D., expert in irrigation systems, played the role of decision-maker, and one of the authors played the role of IoT expert, conducing the experience we provide. These users designed and implemented a sensing application following the IRRINOV method [22] described in Section 2.

The decision-maker and IoT expert follow the methodology described in Section 4.1. During that process, we evaluate the metrics proposed in [55]. First, the decision-maker clearly explains the requirements of the IRRINOV method to the IoT expert as a request. Second, the IoT expert builds the PIM model for these requirements and presents it to the decision-maker for evaluation. In this step, the decision-maker evaluates the **Specification** and **Usage** of the model [55]. We also count the number of iterations until the model is accepted. Third, the IoT expert defines the PSM for one implementation option and presents it to the decision-maker, evaluating its **Specification** and **Usage** and counting the iterations until acceptance. Fourth, the IoT expert uses the STS4IoT model-to-code tool to generate the application code. In this step, the IoT expert assesses the code completeness and simulates it, presenting initial test results to the decision-maker to evaluate **Implementation**. If the decision-maker deems the results appropriate, we consider the application successfully generated. Finally, steps three and four are iterative for each implementation option. Notwithstanding, we only present the results for implementation options D and B (Fig. 1).

Table 4 shows the results for the PIM and PSM models of our case study. The models for the first implementation (Option **D**) are Figs. 6 (PIM) and 13 (PSM). The PSM model for the second implementation (Option **B**) is in Fig. 14. The PIM is the same for the two options, and thus it is not remodelled for the second one.

These results evidence that the *Legibility* of all models is perfect, and their Simplicity is good [55]. Even though *Simplicity* falls to 0.52 in the two PSM, values above 0.5 indicate a predominant presence of classes in the model, which reduces its complexity. Indeed, considering the high complexity of the IRRINOV method used for our case study, this result is very positive.

Moreover, the models received various rejects on their *Completeness* and *Understandability*, forcing three iterations in the PIM and two iterations in the PSM of option **D**. Yet, the mistakes of the designer generated them. Indeed, the decision-maker accepted the

**Table 4**

Models analysis for implementation options D (first) and B (second). The second options does not require a PIM since it is the same than for the first one.

| Step | Number of IterationsIteration | Time to deliver | Specification | | Usage | | Implementation | |
|---|---|---|---|---|---|---|---|---|
| | | | Legibility | Simplicity | Completeness | Understandability | Implementability | Maintainability |
| **First Implementation Option (D)** | | | | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| PIM | | 1 | 28:28 | 1 | 0.55 | Reject: Filter measures; Calculate irrigation; Type of join. | Reject: Specify source measure. | | 3–– |
| | | 2 | 21:22 | 1 | 0.55 | Reject: Multiplicity cannot be 0. | Accepted | | |
| | | 3 | 1:05 | 1 | 0.55 | Accepted | Accepted | | |
| PSM | 2 | 1 | 1:02:41 | 1 | 0.52 | Reject: Select interface. | Reject: Change transformed soil-moisture units. | – | – |
| | | 2 | 10:54 | 1 | 0.52 | Accepted | Accepted | | |
| **Modelling Time** | | | **2:04:30** | | | | | | – |
| Code | 1 | 1 | 32:12 | – | – | Accepted | – | 98% | – |
| **Time Until Implementation D** | | | **2:36:42** | | | | | | – |
| **Second Implementation Option (B)** | | | | | | | | | |
| PSM | 1 | 1 | 24:06 | 1 | 0.52 | Accepted | Accepted | – | **67%** |
| **Modelling Time** | | | **24:06** | | | | | | **81%** |
| Code | 1 | 1 | 7:49 | – | – | Accepted | – | 99% | **76%** |
| **Time Until Implementation B** | | | **31:55** | | | | | | **80%** |

PSM of the second implementation (option **B**) during the first iteration. Nevertheless, the decision-maker suggested the real-time analysis of rain data considering the possibility of automatic irrigation. After revising the IRRINOV method, the decision-maker decided to exclude the suggestion since it was outside the case-study scope.

Regarding **Implementability**, the percentage of code automatically generated is 98% for the first implementation (option **D**) and 99% for the second one (option **B**). The model-to-code tool does not parse the operation to calculate the water need since it is specific to the case study. Also, it does not parse the irrigation value since it is not a variable. However, it correctly generates the rest of the code, and the manual modification is relatively short (around 30 min the first time).

The *Maintainability* of the PSM (exclusive) is 67%, which is positive considering that it changes from two devices (option **D**) to four (option **B**). Besides, the time-to-code reduces by 76% since the IoT expert already knows how to modify it. In this way, the time to model and implement a different option is around 80%, which is advantageous to test multiple options before defining the best one.

Consequently, the results in Table 4 are a good sign towards the confirmation of the theoretical assessment of our profile. First, the high *Understandability* and *Well-Structured* of STS4IoT allowed for clear and coherent models that the decision-maker could understand and evaluate. Second, the *Functionality* was enough to model the sensing required in the case study but not the final data analysis. Yet, we could include it in the model (without stereotypes) thanks to the underpinning UML meta-model. Third, the *Well-Structured* and our automatic model-to-code tool allowed for an excellent **Implementation** of the models. However, the *Functionality* of STS4IoT slightly hindered the *Implementability* of the models since one function could not be stereotyped. It is worth noting that full confirmation of the theoretical results would require more experimental work.

## 8. Conclusion and future work

Conceptual modelling is paramount for successful projects, especially those involving multiple information systems such as IoTbased applications. However, existing model-driven approaches present several limitations to provide a complete description of the data computation and communication in the IoT. Therefore, in this paper, we proposed to move towards a comprehensive UML profile to design IoT-based applications, following the MDA guidelines.

In particular, our STS4IoT UML profile extends the data-centric model of [16], considering a vaster range of functional requirements and implementation options. Indeed, we studied the limitations of the previous model and focused on overcoming them without reducing the abstraction level.

The results of our theoretical validation show that the profiles in STS4IoT have a good architecture and are easy to understand. Also, they comply with the UML standards and allow building well-formed instance models. In this way, STS4IoT can model and generate implementable code for different IoT applications in various domains.

Indeed, the observations from a real experience are promising. The STS4IoT PIM profile allows modelling the FR of complex applications involving multiple heterogeneous IoT devices that run different operations without losing the focus on data or the strict separation of concerns. Besides, the STS4IoT PSM profile allows designing numerous implementation options for the same data, which are implemented seamlessly with the STS4IoT model-to-code tool.

Additionally, considering the high reusability of the STS4IoT PIM profile, we could easily integrate it into different (cloud-based) information systems, such as stream data warehouses [21].

Regarding the successive steps, we are currently working on increasing the usability of our model-to-code tool with a user-centred visual interface. Such an interface will further ease the implementation process for both business users and IoT experts. Besides, we are also working on extending the range of supported hardware devices to leverage the advantages of STS4IoT in different domains. In this sense, we propose the extension of the DM profile with some generic elements common to various IoT platforms and devices.

Finally, as future works, we propose to (i) include data-quality concepts in our profiles, *e.g.* redundancy and clusters. (ii) To extend other data-centric profiles for different information systems (*e.g.* [57]) with our profile as a data source (such as stream data or images). These extensions should allow modelling more complex (and real) applications that support the conceptual integration and implementation of the sink nodes with the cloud. And (iii) evaluate our proposal with different experiments to understand its actual design capacities and limitations.

Furthermore, we consider that the definition of a Domain Specific Language fully integrated into a CASE tool could allow for more complex data transformations and operations. Indeed, it could recognise and implement user-defined OCL rules used to express quality features on IoT data (*e.g.* criteria proposed by [47]). Or it could include data-relevant non-functional requirements as defined in [58].

### CRediT authorship contribution statement

**Julian Eduardo Plazas:** Conceptualization, Methodology, Investigation, Formal analysis, Software, Writing – original draft. **Sandro Bimonte:** Supervision, Conceptualization, Validation, Visualization, Writing – review & editing. **Michel Schneider:** Validation, Writing – review & editing. **Christophe de Vaulx:** Resources, Visualization, Writing – review & editing. **Pietro Battistoni:** Visualization, Software. **Monica Sebillo:** Visualization, Software. **Juan Carlos Corrales:** Conceptualization, Visualization, Formal analysis, Writing – review & editing.

### Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

### Acknowledgements

This work was partially supported by the PhD project ''*Data-Centric UML profile based on the Model-Driven Architecture for the Internet of Things*'' of Universidad del Cauca (ID 5639) in Colombia. We also thank MINCIENCIAS Colombia for the PhD scholarship granted to Julián Eduardo Plazas.

### References

[1] Bahar Farahani, Farshad Firouzi, Victor Chang, Mustafa Badaroglu, Nicholas Constant, Kunal Mankodiya, Towards fog-driven IoT eHealth: Promises and challenges of IoT in medicine and healthcare, Future Gener. Comput. Syst. 78 (2018) 659–676, http://dx.doi.org/10.1016/j.future.2017.04.036.

[2] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, E.M. Aggoune, Internet-of-things (IoT)-based smart agriculture: Toward making the fields talk, IEEE Access 7 (2019) 129551–129583.

[3] Luigi Atzori, Antonio Iera, Giacomo Morabito, Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm, Ad Hoc Netw. 56 (2017) 122–140, http://dx.doi.org/10.1016/j.adhoc.2016.12.004.

[4] Yongrui Qin, Quan Z. Sheng, Nickolas J.G. Falkner, Schahram Dustdar, Hua Wang, Athanasios V. Vasilakos, When things matter: A survey on data-centric internet of things, J. Netw. Comput. Appl. 64 (2016) 137–153, http://dx.doi.org/10.1016/j.jnca.2015.12.016.

[5] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, Alfonso Pierantonio, Grand challenges in model-driven engineering: an analysis of the state of the research, Softw. Syst. Model. 19 (1) (2020) 5–13.

[6] Chaowei Yang, Keith Clarke, Shashi Shekhar, C. Vincent Tao, Big spatiotemporal data analytics: a research and innovation frontier, Int. J. Geogr. Inf. Sci. 34 (6) (2020) 1075–1088, http://dx.doi.org/10.1080/13658816.2019.1698743.

[7] B. Omoniwa, R. Hussain, M.A. Javed, S.H. Bouk, S.A. Malik, Fog/Edge computing-based IoT (fecIoT): Architecture, applications, and research issues, IEEE Int. Things J. 6 (3) (2019) 4118–4149.

[8] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, C. Lin, Edge of things: The big picture on the integration of edge, IoT and the cloud in a distributed computing environment, IEEE Access 6 (2018) 1706–1717.

[9] Stewart Robinson, Gilbert Arbez, Louis G. Birta, Andreas Tolk, Gerd Wagner, Conceptual modeling: definition, purpose and benefits, in: Proce. Winter Simulation Conference, Huntington Beach, CA, USA, 2015, pp. 2812–2826.

[10] Alberto Rodrigues da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Comput. Lang. Syst. Struct. 43 (2015) 139–155, http://dx.doi.org/10.1016/j.cl.2015.06.001.

[11] Gabriel Sebastián, Jose A. Gallud, Ricardo Tesoriero, Code generation using model driven architecture: A systematic mapping study, J. Comput. Lang. 56 (2020) 100935.

[12] Tina Samizadeh Nikoui, Amir Masoud Rahmani, Ali Balador, Hamid Haj Seyyed Javadi, Internet of things architecture challenges: A systematic review, Int. J. Commun. Syst. 34 (4) (2021) e4678.

[13] Matteo Golfarelli, Dario Maio, Stefano Rizzi, The dimensional fact model: A conceptual model for data warehouses, Int. J. Cooperative Inf. Syst. 7 (2–3) (1998) 215–247, http://dx.doi.org/10.1142/S0218843098000118.

[14] Kleanthis Thramboulidis, Foivos Christoulakis, UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems, Comput. Ind. 82 (2016) 259–272, http://dx.doi.org/10.1016/j.compind.2016.05.010, URL http://www.sciencedirect.com/science/article/pii/S016636151630094X.

[15] Claudio M. de Farias, Italo C. Brito, Luci Pirmez, Flávia C. Delicato, Paulo F. Pires, Taniro C. Rodrigues, Igor L. dos Santos, Luiz F.R.C. Carmo, Thais Batista, COMFIT: A development environment for the internet of things, Future Gener. Comput. Syst. 75 (2017) 128–144, http://dx.doi.org/10.1016/j. future.2016.06.031.

[16] J.E. Plazas, S. Bimonte, C. de Vaulx, M. Schneider, Q.-D. Nguyen, J.-P. Chanet, H. Shi, K.M. Hou, J. Carlos Corrales, A conceptual data model and its automatic implementation for IoT-based business intelligence applications, IEEE Internet Of Things Journal 7 (10) (2020) 10719–10732, http://dx.doi.org/10.1109/JIOT.2020.3016608.

[17] William Yeoh, Andy Koronios, Critical success factors for business intelligence systems, J. Comput. Inf. Syst. 50 (3) (2010) 23–32, http://dx.doi.org/10. 1080/08874417.2010.11645404.

[18] Ejaz Ahmed, Ibrar Yaqoob, Ibrahim Abaker Targio Hashem, Imran Khan, Abdelmuttlib Ibrahim Abdalla Ahmed, Muhammad Imran, Athanasios V. Vasilakos, The role of big data analytics in internet of things, Comput. Netw. 129 (2017) 459–471, http://dx.doi.org/10.1016/j.comnet.2017.06.013, Special Issue on 5G Wireless Networks for IoT and Body Sensors.

[19] Claudia M. Sosa-Reyna, Edgar Tello-Leal, David Lara-Alabazares, Methodology for the model-driven development of service oriented IoT applications, J. Syst. Archit. 90 (2018) 15–22, http://dx.doi.org/10.1016/j.sysarc.2018.08.008.

[20] Ralph Kimball, Margy Ross, The Data Warehouse Toolkit: the Complete Guide to Dimensional Modeling, second ed., Wiley, 2002, URL https://www. worldcat.org/oclc/49284159.

[21] Julián Eduardo Plazas, Sandro Bimonte, Michel Schneider, Christophe de Vaulx, Juan Carlos Corrales, Self-service business intelligence over on-demand IoT data: A new design methodology based on rapid prototyping, in: European Conference on Advances in Databases and Information Systems, in: Communications in Computer and Information Science, vol. 1259, Springer, 2020, pp. 84–93, http://dx.doi.org/10.1007/978-3-030-54623-6_8, URL DOI: 10.1007/978-3-030-54623-6_8.

[22] Limagrain Arvalis, Chambre d'Agriculture Puy-de dÔme, Guide de l'utilisateur, Carnet de terrain: Pilotez l'irrigation avec la méthode IRRINOV, Technical report 05G04, Arvalis, Puy-de-Dôme, Limagne & Val d'Allier, France, 2005, p. 40p.

[23] Hai-Ying Zhou, Dan-Yan Luo, Yan Gao, De-Cheng Zuo, Modeling of node energy consumption for wireless sensor networks, Wirel. Sensor Netw. 3 (1) (2011) 18.

[24] Nenad Petrovic, Milorad Tosic, MADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing, Simul. Model. Practice Theory 101 (2020) 102033.

[25] J. Luo, L. Zhang, X. Li, A model-driven parallel processing system for IoT data based on user-defined functions, in: 2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics, ICCCBDA 2020, 2020, pp. 463–470, http://dx.doi.org/10.1109/ICCCBDA49378.2020.9095646.

[26] P. Cedillo, W. Valdez, P. Cárdenas-Delgado, D. Prado-Cabrera, A data as a service metamodel for managing information of healthcare and internet of things applications, Commun. Comput. Inf. Sci. 1307 (2020) 272–286, http://dx.doi.org/10.1007/978-3-030-62833-8_21.

[27] A.F. Subahi, K.E. Bouazza, An intelligent IoT-based system design for controlling and monitoring greenhouse temperature, IEEE Access 8 (2020) 125488–125500, http://dx.doi.org/10.1109/ACCESS.2020.3007955.

[28] Judith Awiti, Alejandro A. Vaisman, Esteban Zimányi, Design and implementation of ETL processes using BPMN and relational algebra, Data Knowl. Eng. 129 (2020) 101837, http://dx.doi.org/10.1016/j.datak.2020.101837.

[29] J. Parri, F. Patara, S. Sampietro, E. Vicario, A framework for model-driven engineering of resilient software-controlled systems, Computing 103 (4) (2021) 589–612, http://dx.doi.org/10.1007/s00607-020-00841-6.

[30] B. Costa, P.F. Pires, F.C. Delicato, Towards the adoption of OMG standards in the development of SOA-based IoT systems, J. Syst. Softw. 169 (2020) http://dx.doi.org/10.1016/j.jss.2020.110720.

[31] D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala, A model-driven engineering approach for the service integration of IoT systems, Cluster Comput. 23 (3) (2020) 1937–1954, http://dx.doi.org/10.1007/s10586-020-03150-x.

[32] J.C. Kirchhof, J. Michael, B. Rumpe, S. Varga, A. Wortmann, Model-driven digital twin construction: Synthesizing the integration of cyber-physical systems with their information systems, in: Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2020, 2020, pp. 90–101, http://dx.doi.org/10.1145/3365438.3410941.

[33] B.E. Khalyly, A. Belangour, A. Erraissi, M. Banane, Devops and microservices based internet of things meta-model, Int. J. Emerg. Trends Eng. Res. 8 (9) (2020) 6254–6266, http://dx.doi.org/10.30534/ijeter/2020/217892020.

[34] J. Novacek, A. Kuhlwein, S. Reiter, A. Viehl, O. Bringmann, W. Rosenstiel, Lemons: Leveraging model-based techniques to enable non-intrusive semantic enrichment in wireless sensor networks, in: Proceedings - 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, 2020, pp. 561–568, http://dx.doi.org/10.1109/SEAA51224.2020.00092.

[35] T. Nepomuceno, T. Carneiro, P.H. Maia, M. Adnan, T. Nepomuceno, A. Martin, AutoIoT: A framework based on user-driven MDE for generating IoT applications, in: Proceedings of the ACM Symposium on Applied Computing, 2020, pp. 719–728, http://dx.doi.org/10.1145/3341105.3373873.

[36] Cristian González García, Daniel Meana-Llorián, Vicente García-Díaz, Andrés Camilo Jiménez, John Petearson Anzola, Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering, IEEE Access 8 (2020) 141872–141894, http://dx.doi.org/10.1109/ACCESS.2020.3012503.

[37] Juan Trujillo, Sergio Luján-Mora, A UML based approach for modeling ETL processes in data warehouses, in: Il-Yeol Song, Stephen W. Liddle, Tok-Wang Ling, Peter Scheuermann (Eds.), Conceptual Modeling - ER 2003, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 307–320.

[38] Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu, Data aggregation processes: a survey, a taxonomy, and design guidelines, Computing 101
(10) (2019) 1397–1429.

[39] Alberto Abelló, José Samos, Fèlix Saltor, YAM2: a multidimensional conceptual model extending UML, Inf. Syst. 31 (6) (2006) 541–567.

[40] Jose-Norberto Mazón, Juan Trujillo, A hybrid model driven development framework for the multidimensional modeling of data warehouses!, ACM SIGMOD Rec. 38 (2) (2009) 12–17.

[41] Kamal Boulil, Sandro Bimonte, François Pinet, Conceptual model for spatial data cubes: A UML profile and its automatic implementation, Comput. Stand. Interfaces 38 (2015) 113–132, http://dx.doi.org/10.1016/j.csi.2014.06.004.

[42] Zineb El Akkaoui, Esteban Zimányi, Jose-Norberto Mazón, Juan Trujillo, A model-driven framework for ETL process development, in: Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP, 2011, pp. 45–52.

[43] Alejandro Maté, Juan Trujillo, A trace metamodel proposal based on the model driven architecture framework for the traceability of user requirements in data warehouses, Inf. Syst. 37 (8) (2012) 753–766.

[44] Alejandro Maté, Juan Trujillo, Tracing conceptual models' evolution in data warehouses by using the model driven architecture, Comput. Stand. Interfaces 36 (5) (2014) 831–843.

[45] Majid Ashouri, Paul Davidsson, Romina Spalazzese, Cloud, edge, or both? Towards decision support for designing IoT applications, in: 2018 Fifth International Conference on Internet of Things: Systems, Management and Security, IEEE, 2018, pp. 155–162.

[46] Dengfeng Gao, Christian S Jensen, Richard T Snodgrass, Michael D Soo, Join operations in temporal databases, The VLDB J. 14 (1) (2005) 2–29.

[47] Ricardo Pérez-Castillo, Ana G. Carretero, Ismael Caballero, Moisés Rodríguez, Mario Piattini, Alejandro Mate, Sunho Kim, Dongwoo Lee, DAQUA-MASS:
an ISO 8000-61 based data quality management methodology for sensor data, Sensors 18 (9) (2018) 3105, http://dx.doi.org/10.3390/s18093105, URL DOI: 10.3390/s18093105.

[48] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, Romina Spalazzese, Model-driven engineering for mission-critical iot systems, IEEE Softw. 34 (1) (2017) 46–53.

[49] Fredrik Lundh, The ElementTree XML API, 2017, Python Software Foundation, URL https://github.com/python/cpython/blob/041bfafeb00b0e5e565986b01c00bf9cddce3b4c/Lib/xml/etree/ElementTree.py, original-date: 2017-02-10T19:23:51Z.

[50] Basic date and time types, 2017, Python Software FoundationURL https://github.com/python/cpython/blob/041bfafeb00b0e5e565986b01c00bf9cddce3b4c/ Lib/datetime.py, original-date: 2017-02-10T19:23:51Z.

[51] Hela Marouane, Claude Duvallet, Achraf Makni, Rafik Bouaziz, Bruno Sadeg, An UML profile for representing real-time design patterns, J. King Saud Univ. - Comput. Inf. Sci. 30 (4) (2018) 478–497, http://dx.doi.org/10.1016/j.jksuci.2017.06.005.

[52] Zhiyi Ma, Xiao He, Chao Liu, Assessing the quality of metamodels, Front. Comput. Sci. 7 (4) (2013) 558–570.

[53] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio, A tool-supported approach for assessing the quality of modeling artifacts, J. Comput. Lang. 51 (2019) 173–192.

[54] Inc. No Magic, MagicDraw User Manual, 19.0 ed., 2018, URL https://docs.nomagic.com/display/MD190/User+Guide.

[55] Samira Si-Said Cherfi, Jacky Akoka, Isabelle Comyn-Wattiau, Conceptual modeling quality-from EER to UML schemas evaluation, in: International Conference on Conceptual Modeling, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 414–428, http://dx.doi.org/10.1007/3-540-45816-6_38.

[56] Pankesh Patel, Damien Cassou, Enabling high-level application development for the Internet of Things, J. Syst. Softw. 103 (2015) 62–84, http://dx.doi.org/10.1016/j.jss.2015.01.027.

[57] Sandro Bimonte, Omar Boussaid, Michel Schneider, Fabien Ruelle, Design and implementation of active stream data warehouses, Int. J. Data Warehous. Min. (IJDWM) 15 (2) (2019) 1–21.

[58] Amina Belhassena, Sandro Bimonte, Pietro Battistoni, Christophe Cariou, Gerard Chalhoub, Juan Carlos Corrales, Jean Laneurit, Rim Moussa, Julian Eduardo Plazas, Robert Wrembel, Monica Sebillo, On modeling data for IoT agroecology applications by means of a uml profile, in: Proceedings of the 13th International Conference on Management of Digital Ecosystems, in: MEDES, vol. 21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 120–128, http://dx.doi.org/10.1145/3444757.3485109.

**Julian Eduardo Plazas** is a Ph.D. student in Telematics Engineering and Computer Science in Universidad del Cauca (Colombia) and Université Clermont Auvergne (France). He has worked in the definition of rural Early Warning Systems through Converged Services and Complex Event Processing for agricultural environments, in the implementation of Machine-Learning Classifiers for agricultural Decision Support Systems, and in Conceptual Modelling for agricultural Wireless Sensor Networks. His current research topics include the Internet of Things, Business Intelligence, Data Analytics and Conceptual Modelling for Intelligent Agriculture Systems.

**Sandro Bimonte** is a researcher at INRAE, and more exactly he is at TSCF. He received his Ph.D. from INSA-Lyon, France (2004–2007). From 2007–2008, he carried out researches at IMAG, France. He is an editorial board member of International Journal of Decision Support System Technology, and International Journal of Data Mining, Modeling and Management and a member of the Commission on GeoVisualisation of the International Cartographic Association. His research activities concern spatial data warehouses and spatial OLAP, visual languages, geographic information systems, spatiotemporal databases and geovisualisation.

**Dr. Michel Schneider** is emeritus Professor of Databases and Information Systems at the University of Clermont-Ferrand and has over 35 years of experience as lecturer and researcher in the information technology field. He has contributed to many different research topics including : decision support systems, data warehouses, mediation systems and semantic models for databases. He continues to participate in several editorial boards covering these topics.

**Christophe de Vaulx** was born in France in 1976. He received the Ph.D. degree in computer science from University Blaise Pascal, France in 2003.

From 2004 to 2005, he was postdoctoral researcher at Laboratory of Computing, Modelling and Optimization of the Systems (LIMOS) a Mixed Unit of Research (UMR 6158) in computing (University Blaise Pascal). Since 2005 he has been associate professor at Polytech Clermont and LIMOS (University Blaise Pascal from 2005 to 2016, University Clermont Auvergne since 2017). He is the author of more 40 articles and 2 inventions. His research interests include Cyber–physical systems, Internet Of Things, Wireless Sensors Networks, Embedded Systems and Wireless Communication Protocols.

**Pietro Battistoni** is actually a Ph.D. student in Computer Science at University of Salerno (Italy). With a Master's Degree in Computer Science, he owned a microenterprise specialised in hardware and software development for industrial automation for 30 years, before starting his Ph.D. project. His research interests include spatial databases, geographic information systems, Human–GIS interaction, Indoor locating systems, IoT and distributed architectures. He is currently involved in several national research projects and has collaborations with national and international institutions.

**Monica Sebillo** is an Associate Professor in Information Processing Systems at the Department of Computer Science, University of Salerno (Italy). She received a Ph.D. degree in Applied Math and Computer Science. Monica is an ACM senior member and the president of AMFM GIS Italia, member of EUROGI. Her research interests include spatial databases and geographic information systems, Human–GIS interaction and territorial intelligence services. As GIS Laboratory Director, Monica is involved in international and national research projects and has several collaborations with national and international institutions. Monica is the author of more than 100 publications in international journals and conference proceedings.

**Juan Carlos Corrales** received the Dipl-Ing and master's degrees in telematics engineering from the University of Cauca, Colombia, in 1999 and 2004 respectively, and the Ph.D. degree in sciences, speciality computer science, from the University of Versailles Saint-Quentin-en-Yve-lines, France, in 2008. At present, he is a full professor and leads the Telematics Engineering Group (GIT) at the University of Cauca. His research interests focus on machine learning and data analytics.