Data-Centric UML Profile based on the Model-Driven Architecture for the Internet of Things

Julian Eduardo Plazas Pemberthy

Data-Centric UML Profile based on the Model-Driven Architecture for the Internet of Things

Universidad del Cauca

Doctoral Thesis in Cotutelle

Julián Eduardo Plazas Pemberthy

Thesis Supervisors:
Dr Sandro Bimonte, INRAE Clermont-Ferrand
Dr Juan Carlos Corrales Muñoz, Universidad del Cauca

Thesis Adviser:
Dr Christophe de Vaulx, Université Clermont Auvegne

Universidad del Cauca	**Université Clermont Auvegne**
Doctorate Programme in Telematics Engineering	**Doctoral School of Science for Engineers**
Line of Research in eAgriculture	**Speciality in Computer Science**

Popayán - Colombia, October 13th 2022

Julián Eduardo Plazas Pemberthy

Data-Centric UML Profile based on the Model-Driven Architecture for the Internet of Things

Thesis submitted to the
Faculty of Electronics and Telecommunications Engineering
of the Universidad del Cauca, Colombia
and to the Doctoral School of Science for Engineers
of the Université Clermont Auvergne, France
for the acquisition of the academic degrees

Doctor en Ingeniería Telemática

Doctorat de l'Université, Spécialité Informatique

Jury Members:

Dr Sandro Bimonte	INRAE Clermont-Ferrand	Supervisor
Dr Juan Carlos Corrales Muñoz	Universidad del Cauca	Supervisor
Dr Christophe de Vaulx	Université Clermont Auvegne	Adviser
Dr Isabelle Wattiau	ESSEC Business School	Reviewer
Dr Karine Zeitouni	Université de Versailles	Reviewer
Dr Jerome Gensel	Université Grenoble Alpes	Examiner
Dr Alexandre Guitton	Université Clermont Auvegne	Examiner
Dr Giuseppe Polese	Università degli Studi di Salerno	Examiner
Dr Gustavo Ramirez	Universidad del Cauca	Examiner

Popayán (Cauca), Colombia
October 13th 2022

UNIVERSITÉ Clermont Auvergne

CLERMONT AUVERGNE INP — ÉCOLE DOCTORALE DES SCIENCES POUR L'INGÉNIEUR

Direction de la Recherche et des Etudes Doctorales
Services Etudes Doctorales et HDR
Campus Universitaire des Cézeaux
8 avenue Blaise Pascal
TSA 60026 - 63178 AUBIERE cedex
edspi.dred@uca.fr

ATTESTATION de DIPLOME

La Directrice de la Recherche et des Etudes Doctorales, soussignée, atteste que :

Monsieur PLAZAS PEMBERTHY Julian Eduardo
Né le 19 mai 1992 à Popayan (Colombie)

a soutenu une thèse le 13 octobre 2022 et a obtenu le grade de **Docteur d'Université en INFORMATIQUE**

Sujet :
Data-centric UML Profile based on the Model-Driven Architecture for the Internet of Things

Jury :
M. BIMONTE Sandro, Directeur de recherche, INRAE Clermont-Ferrand
M. CORRALES Juan, Professeur, Universidad Del Cauca, Colombie
M. GENSEL Jérôme, Professeur, Université de Grenoble
M. GUITTON Alexandre, Professeur, Université Clermont Auvergne
M. POLESE Giuseppe, Professeur, Université Salerno (Italie)
M. RAMIREZ Gustavo, Professeur, Universidad Del Cauca, Colombie
Mme WATTIAU Isabelle, Professeur, ESSEC Business School, Paris
Mme ZEITOUNI Karine, Professeur, Université de Versailles

Président de jury : M. GUITTON Alexandre

Attestation établie pour servir et valoir ce que de droit.

Fait à Aubière, le 24 octobre 2022

UNIVERSITÉ Clermont Auvergne

Pascale BOUVIER-MARION

Dedicado a mis padres, Adolfo y Luz Stella,
a mis hermanos, Santiago y Luz Adriana,
y a mi esposa, Carolina.
¡Gracias por tanto!
Los amo.

Acknowledgements

I want to thank God for giving me the life, strength and motivation to finalise my thesis. To my wife, Carolina, who always stood by my side and never stopped believing in me. I also thank my family for their constant support, trust and love, and my friends in Colombia and France for making this process more pleasant and always cheering me up.

I must also thank doctors Juan Carlos Corrales, Sandro Bimonte and Christophe de Vaulx, and professor Michel Schneider for their patience and efforts. Their guidance, tutoring and advice have made me a better person and researcher. I thank Universidad del Cauca, Université Clermont Auvergne and INRAE Clermont-Ferrand for supporting my research; the GIT and COPAIN groups and the LIMOS laboratory for their scientific and personal support; the French Embassy in Bogotá for my mobility scholarship; Minciencias Colombia for my PhD scholarship and thesis founding; and the Department of Telematics for completing my scholarship for the last year.

Abstract

Background: The Internet of Things (IoT) is a broad concept comprising large networks of smart objects connected to the Internet. These objects can sense, receive, transform and communicate data amongst themselves and external entities (systems or users). The data collection capacity of IoT makes it suitable for monitoring applications. These applications provide data about relevant objects and phenomena. Then, data-analysis systems can extract valuable insights from the IoT data. Decision-makers can thus leverage this information to take better courses of action.

Nevertheless, default IoT implementations do not always provide the necessary data for the best value extraction. Moreover, classical data analysis and storage systems do not support the direct integration of streaming IoT data.

In this context, conceptual data modelling is a powerful tool for the standard definition of the data needs of the users. Besides, it allows for a transparent data integration process. Moreover, model-driven frameworks leverage the representation of the system (IoT) to provide significant aid in the implementation process, even generating the complete application code.

Aims: Consequently, I propose to ease the definition and implementation of IoT data according to the end-users needs with a model-driven approach. In particular, the main objective of this thesis is to define a data-centric UML profile for IoT following the Model-Driven Architecture (MDA) guidelines. The UML profile should allow for a simple and readable representation of IoT data while enabling the (partial) generation of implementable code.

Methods: To achieve this aim, I divided this thesis into four phases. In the first phase, an analysis of different applications allowed identifying the main concepts for modelling sensor- and IoT-data applications. These concepts comprise the data-related components in these applications and their relationships and flows. Second, a conceptual modelling process based on MDA allowed defining a UML profile for sensor-data applications. This profile leveraged the identified concepts for sensor data. In the third phase, an extended UML profile for IoT-data applications derived from the sensor-data profile and the modelling concepts for IoT data. The sensor- and IoT-data profiles enable the generation of implementable code with model-to-code tools. Finally, the fourth phase validated the sensor- and IoT-data UML profiles and model-to-code tools through a theoretical quality assessment and a set of experiments.

Results: Through this process, this thesis provides four outcomes. First, it defines the prime data features that describe sensor- and IoT-data applications. While sensor-data applications are information systems based on a single device that provides data about a sensed entity, IoT-data applications are more complex, involving multiple devices with increased communication and computation capabilities. Second, it studies the literature about model-driven IoT using two complementary approaches: a scientometric analysis and a literature review. This study highlights the importance of data for IoT modelling and the low coverage it has in the literature. Third, it provides a simple UML profile and MDA approach for developing sensor-data applications. Fourth, it presents a UML profile and MDA approach for IoT-data applications. This approach supports multiple implementation options in different commercial and non-commercial platforms.

Conclusions: These outcomes have multiple contributions inside and outside this thesis. The description of the prime data features provided a conceptual framework to assess the related-works description of sensor and IoT data. Besides, this framework guided the conceptual-modelling process, allowing for complete yet simple representations of sensor- and IoT-data applications. In this way, end-users can provide standard models of their IoT data needs. Moreover, these models simplify the implementation and use of IoT data. I expect these concepts and constructs will help researchers in this area to build their IoT models with a clearer view of the data.

Keywords: Internet of Things (IoT); Conceptual Data Model, UML Profile, Model-Driven Architecture; Sensor Data; IoT Data.

Resumen

Antecedentes: El Internet de las cosas (IoT) es un concepto amplio que comprende grandes redes de objetos inteligentes conectados a Internet. Estos objetos pueden capturar, recibir, transformar y comunicar datos entre ellos y sistemas o usuarios externos. La capacidad de recopilación de datos de IoT lo hace adecuado para aplicaciones de monitoreo. Estas aplicaciones proporcionan datos sobre objetos y fenómenos relevantes. Luego, los sistemas de análisis de datos pueden extraer información valiosa de los datos de IoT. De esta manera, los tomadores de decisiones pueden aprovechar esta información para emprender mejores cursos de acción.

Sin embargo, las implementaciones de IoT predeterminadas no siempre proporcionan los datos necesarios para la mejor extracción de valor. Además, los sistemas clásicos de análisis y almacenamiento de datos no admiten la integración directa de los datos de IoT.

En este contexto, el modelado conceptual de datos es una herramienta poderosa para la definición estándar de las necesidades de datos de los usuarios finales que también permite un proceso de integración de datos transparente. Además, los marcos basados en modelos aprovechan la representación del sistema (IoT) para brindar una ayuda significativa en el proceso de implementación, incluso generando completamente el código de aplicación.

Objetivos: Por lo tanto, propongo facilitar la definición e implementación de datos de IoT de acuerdo con las necesidades de los usuarios finales con un enfoque basado en modelos. En particular, el objetivo principal de esta tesis es definir un perfil UML centrado en datos para IoT siguiendo las pautas de la arquitectura basada en modelos (MDA). El perfil UML debe permitir una representación simple y legible de los datos de IoT al tiempo que permite la generación (parcial) de código implementable.

Métodos: Para lograr este objetivo, he dividido esta tesis en cuatro fases. En la primera fase, un análisis de diferentes aplicaciones permitió identificar los conceptos principales para modelar aplicaciones de datos de sensores e IoT. Estos conceptos comprenden los componentes relacionados con los datos en estas aplicaciones y sus relaciones y flujos. En segundo lugar, un proceso de modelado conceptual basado en MDA permitió definir un perfil UML para aplicaciones de datos de sensores. Este perfil aprovechó los conceptos identificados para los datos del sensor. En la tercera fase, del perfil de datos de sensores y de los conceptos de modelado para datos de IoT derivó un perfil UML extendido para aplicaciones de datos de IoT. Los perfiles

de datos de sensores e IoT permiten la generación de código implementable con herramientas de modelo a código. Finalmente, la cuarta fase validó los perfiles UML de datos de sensores e IoT y las herramientas de modelo a código a través de una evaluación de calidad teórica y un conjunto de experimentos.

Resultados: A través de este proceso, esta tesis proporciona cuatro resultados. Primero, define las principales características de datos que describen las aplicaciones de datos de sensores e IoT. Mientras que las aplicaciones de datos de sensores son sistemas de información basados en un solo dispositivo que proporciona datos sobre una entidad monitorizada, las aplicaciones de datos de IoT son más complejas e involucran múltiples dispositivos con mayores capacidades de comunicación y computación. En segundo lugar, estudia la literatura sobre IoT basado en modelos utilizando dos enfoques complementarios: un análisis cienciométrico y una revisión de la literatura. Este estudio destaca la importancia de los datos para el modelado de IoT y la baja cobertura que tiene en la literatura. En tercer lugar, proporciona un perfil UML simple y un enfoque MDA para desarrollar aplicaciones de datos de sensores. Cuarto, presenta un perfil UML y un enfoque MDA para aplicaciones de datos IoT. Este enfoque soporta múltiples opciones de implementación en diferentes plataformas comerciales y no comerciales.

Conclusiones: Estos resultados tienen múltiples aportes dentro y fuera de esta tesis. La descripción de las principales características de los datos proporcionó un marco conceptual para evaluar la descripción de trabajos relacionados de los datos de sensores e IoT. Además, este marco guía el proceso de modelado conceptual, lo que permitió representaciones completas pero simples de aplicaciones de datos de sensores e IoT. De esta forma, los usuarios finales pueden proporcionar modelos estándar de sus necesidades de datos de IoT. Además, estos modelos simplifican la implementación y el uso de datos de IoT. Espero que estos conceptos y construcciones ayuden a los investigadores en esta área a construir sus modelos de IoT con una visión más clara de los datos.

Palabras Clave: Internet de las Cosas (IoT); Modelo Conceptual de Datos, Perfil UML, Arquitectura Basada en Modelos; Datos de sensor; Datos IoT.

Résumé

Contexte : L'Internet des objets (IoT) est un concept large comprenant de grands réseaux d'objets intelligents connectés à Internet. Ces objets peuvent détecter, recevoir, transformer et communiquer des données entre eux et avec des entités externes (systèmes ou utilisateurs). La capacité de collecte de données de l'IoT le rend adapté aux applications de surveillance. Ces applications fournissent des données sur les objets et les phénomènes pertinents. Ensuite, les systèmes d'analyse de données peuvent extraire des informations précieuses des données IoT. Les décideurs peuvent ainsi tirer parti de ces informations pour adopter de meilleures lignes de conduite.

Néanmoins, les implémentations IoT par défaut ne fournissent pas toujours les données nécessaires pour la meilleure extraction de valeur. De plus, les systèmes classiques d'analyse et de stockage de données ne prennent pas en charge l'intégration directe des données IoT en continu.

Dans ce contexte, la modélisation conceptuelle des données est un outil puissant pour la définition standard des besoins en données des utilisateurs. En outre, il permet un processus d'intégration de données transparent. De plus, les cadres pilotés par modèle exploitent la représentation du système (IoT) pour fournir une aide significative dans le processus de mise en œuvre, générant même le code d'application complet.

Objectifs : Par conséquent, je propose de faciliter la définition et la mise en œuvre des données IoT en fonction des besoins des utilisateurs finaux avec une approche basée sur les modèles. En particulier, l'objectif principal de cette thèse est de définir un profil UML centré sur les données pour l'IoT en suivant les directives de l'architecture pilotée par les modèles (MDA). Le profil UML doit permettre une représentation simple et lisible des données IoT tout en permettant la génération (partielle) de code implémentable.

Méthodes : Pour atteindre cet objectif, j'ai divisé cette thèse en quatre phases. Dans la première phase, une analyse des différentes applications a permis d'identifier les principaux concepts de modélisation des applications de données de capteurs et IoT. Ces concepts comprennent les composants liés aux données dans ces applications et leurs relations et flux. Deuxièmement, un processus de modélisation conceptuelle basé sur MDA a permis de définir un profil UML pour les applications de données de capteurs. Ce profil a tiré parti des concepts identifiés pour

les données des capteurs. Dans la troisième phase, un profil UML étendu pour les applications de données IoT dérivé du profil de données de capteur et des concepts de modélisation pour les données IoT. Les profils de données de capteur et IoT permettent la génération de code implémentable avec des outils de modèle à code. Enfin, la quatrième phase a validé les profils UML de données de capteurs et IoT et les outils de modélisation à code via une évaluation théorique de la qualité et un ensemble d'expériences.

Résultats : Grâce à ce processus, cette thèse fournit quatre résultats. Tout d'abord, il définit les principales caractéristiques de données qui décrivent les applications de données de capteurs et IoT. Alors que les applications de données de capteur sont des systèmes d'information basés sur un seul appareil qui fournit des données sur une entité détectée, les applications de données IoT sont plus complexes, impliquant plusieurs appareils avec des capacités de communication et de calcul accrues. Deuxièmement, il étudie la littérature sur l'IoT piloté par les modèles en utilisant deux approches complémentaires : une analyse scientométrique et une revue de la littérature. Cette étude met en évidence l'importance des données pour la modélisation de l'IoT et leur faible couverture dans la littérature. Troisièmement, il fournit un profil UML simple et une approche MDA pour développer des applications de données de capteurs. Quatrièmement, il présente un profil UML et une approche MDA pour les applications de données IoT. Cette approche prend en charge plusieurs options de mise en œuvre dans différentes plates-formes commerciales et non commerciales.

Conclusion : Ces résultats ont de multiples contributions à l'intérieur et à l'extérieur de cette thèse. La description des principales caractéristiques des données a fourni un cadre conceptuel pour évaluer la description des travaux connexes des données des capteurs et de l'IoT. En outre, ce cadre a guidé le processus de modélisation conceptuelle, permettant une représentation complète mais simple des applications de données de capteurs et IoT. De cette manière, les utilisateurs finaux peuvent fournir des modèles standard de leurs besoins en données IoT. De plus, ces modèles simplifient la mise en œuvre et l'utilisation des données IoT. Je m'attends à ce que ces concepts et constructions aident les chercheurs dans ce domaine à construire leurs modèles IoT avec une vision plus claire des données.

Mots-clés : Internet des Objets (IoT) ; Modèle Conceptuel de Données ; Profil UML ; Architecture Pilotée par les Modèles ; données de capteur ; Données IoT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Statement of the Problem

The continuous observation of different phenomena has allowed humanity to discover new knowledge and identify potentially beneficial or hazardous situations. Indeed, the Internet of Things (IoT) has (in part) gained relevance in several academic and industrial domains for its ability to automatically and continuously collect data from monitored objects and phenomena [6].

IoT has been defined as a set of physical devices connected to the Internet and capable of generating, computing, storing and sending data in real-time through different systems (*e.g.* ZigBee, Wi-Fi, LoRaWAN) [7].

This information allows decision-makers (*e.g.* researchers or managers) to take better courses of action and control the underlying processes to improve their outcomes or reduce damages and avoid problems [8, 9, 10, 11].

However, collecting, processing, storing, analysing and using IoT data can be challenging. The different sensors in these networks can generate different kinds of data with different configurations (heterogeneous data) while providing multidimensional and multiscale information about the sensed object or phenomenon [12, 13]. Besides, large IoT networks can be a continuous source of Big Data (BD) [6, 10, 14]. Moreover, data-analysis systems such as Business Intelligence (BI) must handle the IoT data to supply the users' requirements for decision support and control [9, 13, 15]. In this way, an IoT-based BI application becomes highly complex from the combination of different information systems (IoT as a source of data and BI for the analysis) and the heterogeneity, uncertainty and volume of IoT data.

Moreover, the implementation of business logic in IoT is usually a time-consuming and error-prone process that hinders IoT experts for responding to complex requirements that may vary constantly [16, 17, 18]. These issues in the data source also limit the capacity of BI experts for effectively providing the appropriate analysis. Therefore, IoT-based BI applications do not always behave as expected by decision-makers and domain experts (*i.e.* business users) despite the efforts of the experts [19, 20].

For these reasons, modern software engineering methodologies (such as Model-Driven) are well known and almost mandatory in the development of complex software and hardware systems (such as IoT-based BI applications) [21, 22, 23, 24]. Model-driven methodologies provide a software development framework focused on the definition of conceptual models that automatically transform into implementable code. Moreover, some approaches such as the Model-Driven Architecture (MDA) define a clear separation of views and abstraction levels. Designers can use the distinct levels to define different aspects of their systems, such as requirements, logic and implementation [23, 25].

The main goal of IoT-based BI applications is to acquire specific on-demand data to provide tailored analyses that derive into effective decision-making [9]. Therefore, an effective conceptual model for these applications must be data-centric [15].

The different levels of MDA can effectively support the design and development of IoT-based BI applications with highly abstract data models and system-specific implementation models. Indeed, multiple authors have provided model-driven approaches for BI [12, 26, 27, 28, 29] and IoT systems [17, 30, 31, 32, 33]. Nevertheless, proposals in the BI area often disregard the difficulties of integrating IoT data and do not focus on the code generation for this subsystem. Similarly, IoT works usually emphasise on the definition of the networks, providing no conceptual data models for data integration and use.

Consequently, the design and development of data-centric IoT applications and IoT-based BI applications do not rely upon model-driven methodologies. This situation hinders the systems' building and capacity to extract value and insights from IoT-generated data. Based on this scenario, I propose the following research question:

How to ease the exploitation of IoT-generated data?

1.2 Motivation

The agriculture and food-security (Agri-food) sector plays a prime role in the economy of several countries, not only for generating wealth and creating employment but also for the nutrition of the population in developed and developing countries. In Colombia, only the agricultural sector represents more than 10% of the National Domestic Product and the livelihood of almost 4 million people [34, 35, 36].

Different aspects like increasing the sector profitability, adapting to climate change, supplying the demands of emerging markets, or ensuring the products' quality, are currently challenging the Agri-food sector. Therefore, innovations like precision agriculture, smart farming, or product tracking are vital for overcoming these challenges [34, 35, 37].

Such innovations rely on the intensive monitoring of the products (*e.g.* crops) and their environments. Information analysis and forecasting systems allow for decision support and situation management and control. These complex systems usually are IoT-based BI applications. BI enables data analysis and supports decision-making, while IoT supplies valuable data from the products and their conditions. Moreover, model-driven approaches could enhance the design

and implementation of these systems in the Agri-food domain to increase their efficiency and effectiveness [23, 36, 38].

Therefore, an effective design of Agri-food information systems requires standardised, simple and readable representations of IoT data besides traditional BI data models and MDA support. UML profiles offer a widespread and standard way to represent these conceptual data models. Indeed, when used in an MDA framework, UML profiles seamlessly separate the system views into different abstraction levels for the multiple actors that define and build these applications.

This thesis focuses on defining an MDA-based UML profile for data-centric IoT applications since there are multiple conceptual models and model-driven approaches for BI systems [12, 13].

1.3 Objectives

1.3.1 General objective

The main objective of this thesis is to define a data-centric UML profile for IoT following the MDA guidelines.

1.3.2 Specific objectives

The specific objectives of this thesis are the:

1. identification of the main data-related components of a data-gathering IoT application.
2. determination of the data and control flows among the components of a data-gathering IoT application.
3. definition of a data-centric meta-model of IoT following the MDA guidelines.
4. validation of the defined meta-model in an agriculture-oriented case study.

1.4 Published Papers

This thesis produced three journal papers and two conference papers (including one international collaboration). The journal papers are:

- "**Data-Centric UML Profile for Wireless Sensors: Application to Smart Farming**", *International Journal of Agricultural and Environmental Information Systems* (IJAEIS) (ISSN 1947-3192), 2019, [36].
- "**A Conceptual Data Model and its Automatic Implementation for IoT-based Business Intelligence Applications**", *IEEE Internet of Things Journal* (ISSN 2327-4662), 2020, [39].

- "**Sense, Transform & Send for the Intenet of Things (STS4IoT): UML Profile for Data-Centric IoT Applications**", Data and Knowledge Engineering (ISSN 0169-023X), 2022, [40].

The conference papers are:

- "**Self-service Business Intelligence over On-Demand IoT Data: A New Design Methodology Based on Rapid Prototyping**", *24th European Conference on Advances in Databases and Information Systems* (ADBIS2020), Lyon, France, 2020, [41].
- "**On Modeling Data for IoT Agroecology Applications by means of a UML Profile**", *13th International Conference on Management of Digital EcoSystems* (MEDES'21), Hammamet, Tunisia, 2021, [42] (International collaboration).

1.5 Contents of the Thesis

This thesis is structured as follows:

Chapter 2. Main Concepts and Research Issues: Presents the most relevant concepts involving sensors, IoT and conceptual modelling. Introduces the case study that contextualises the thesis outcomes. And characterises sensor and IoT-data applications.

Some parts of this chapter are published in [36, 39, 40].

Chapter 3. Related Work: It Studies current literature in the area of model-driven IoT using a double approach: scientrometrics and literature review.

Some parts of this chapter are published in [39, 40].

Chapter 4. Design and Implementation of Sensor-Data Applications: Introduces the UML profile and MDA approach for the design and development of sensor-data applications. Besides, it defines a methodology to conceive and and build sensor- and IoT-based BI applications.

Several parts of this chapter are published in [39, 41].

Chapter 5. Design and Implementation of IoT-Data Applications: Extends the sensor-data MDA and UML profile to support the enhanced computation and communications capabilities of IoT.

Several parts of this chapter are published in [40].

Chapter 6. Theoretical and Practical Validation: Validates the theoretical and practical feasibility of the sensor and IoT-data UML profiles and MDA-based code generation in the context of the case study. Besides, it evaluates the capacity of the IoT-data profile to support the modelling of heterogeneous applications involving different devices.

Several parts of this chapter are published in [39, 40].

Chapter 7. Conclusions: Closes this thesis stating its main outcomes and contributions, exposing the lessons learnt, and proposing the future work.

Chapter 2

Main Concepts and Research Issues

This chapter introduces the most relevant theoretical and technological concepts for this thesis (Sec. 2.1). Besides, it presents a motivational case study on smart-farming that structures and illustrates the contributions of this thesis (Sec. 2.2). Finally, it states the new data-centric requirements for modelling sensor- and IoT-data applications (Sec. 2.3).

2.1 Background

This section provides a concise description of the main techniques, methods, and technologies for defining data-centric, model-driven IoT. Firstly, it clearly defines sensor, sensor data, and sensor-data application, especially considering the context of this thesis (2.1.1). Secondly, it describes IoT, extending the concepts of sensors and specifying how is it used in this thesis (2.1.2). Finally, it explains conceptual modelling, focusing on conceptual data models and their representation in UML (2.1.3).

2.1.1 Sensors

Existing literature provides multiple definitions of a sensor, from a particular material that changes its physical characteristics in the presence of certain stimuli to complex systems that leverage such features to provide detailed data about relevant phenomena, objects or events [43, 44, 45]. Even in a particular context such as agriculture, where this thesis validates its contributions, the sensor concept is ambiguous and loosely defined [46, 38].

Therefore, this thesis follows the definitions of [36, 38] to state that a *sensor device* is a programmable piece of hardware that has the necessary equipment to:

- Sample and digitalise data from a relevant phenomenon, object or event.
- Make computations on the digital data.
- And communicate the (raw or computed) data.

In particular, this thesis focuses on sensors that can at least store and aggregate the collected data and deliver them through a wired or wireless communication protocol. These features allow using sensors as modular, independent information systems that can also contribute as data sources in more complex structures.

Sensor data, *i.e.* digital samples collected, (possibly) computed, and delivered by a single sensor, can provide relevant information about an observed subject. Indeed, monitoring applications usually rely on sensors to acquire the necessary data. For example, [47] uses one sensor to monitor the environmental conditions in a silkworm incubator.

Besides, sensor data is even more advantageous beyond monitoring applications. *Sensor-based applications* leverage sensor data to extract better insights from the monitored subject and obtain benefits. Often, such benefits are automatic courses of action that directly affect the surveilled subject [48, 49]. Similarly, sensor-based applications can provide new knowledge bases or enable strategic planning for decision-making [50, 42].

In this sense, every sensor data is not valuable to extract relevant insights. The value of sensor data depends on multiple conditions that range from its positioning to the quality of the hardware and firmware components [19, 51]. The sensor configuration that allows it to provide the appropriate data is called a *sensor-data application*.

Proper processing can increase data value independently of the sensor's physical arrangement in a sensor-data application. Indeed, data aggregation is essential for any information system since it enables the discovery of relevant patterns, extraction of insights, and reduction of storage space and energy costs. Therefore, a sensor should at least be capable of aggregating its collected data [52].

The component that defines data processing, acquisition and delivery in sensor-data applications is the firmware. Several approaches exist for developing firmware. For example, assembly language [53], dedicated Integrated Development Environments [54], Embedded Operating Systems [55, 56], drag-and-drop designs [57], or model-driven development [33, 58]. Nevertheless, not all are widely used or convenient for current applications. This thesis focuses on Embedded Operating Systems (or Embedded OSes) since they usually provide standard programming for multiple kinds of devices [56].

Embedded Operating Systems

Sensors are embedded systems, *i.e.* computer-controlled machines. Therefore, the computer needs to handle the hardware components precisely to meet the application requirements. In this context, embedded OSes abstract the physical management of such features (*e.g.* registries, analogue to digital converter, memory allocation and reading time of a sensing probe) as accessible APIs that ease the configuration of the internal computer and thus the embedded system [56, 59].

This characteristic allows embedded OSes to provide a standard API for any hardware device they support. For example, Contiki-NG supports four different hardware architectures, while RIOT supports five and Zephyr six [56]. Hence, the same code should seamlessly run on multiple devices from various vendors if they support the same embedded OS and have the same facilities.

For instance, a sensor that only has temperature probes cannot provide rain data and vice-versa.

Nevertheless, the differences amongst embedded OSes go beyond the API. They also have particular design architectures, scheduling algorithms, programming models, and network support. Therefore, migrating from one embedded OS to another is a significant effort since it changes the programming paradigm [56, 59].

In this thesis's work we use two embedded OSes considering the accessibility to the supporting sensors and flexibility of the programming model:

- Simple Embedded OS (SEOS), developed by the LIMOS (UMR 6158 CNRS - Université Clermont Auvergne, France). From the devices that can run SEOS, we have used uSu EDU Boards (UEBs), also developed by LIMOS. These multi-purpose platforms include various useful probes, like air temperature and relative air humidity, and communication interfaces such as IEEE 802.15.4 by default. Besides, they can seamlessly add other specific-purpose modules (*e.g.* soil moisture probes). UEBs and SEOS are used for most examples in Chapters 4 and 5, and some experiments in Chapter 6.

- RIOT, conceived by an alliance between Freie Universität Berlin, INRIA France, and Hamburg University of Applied Sciences [55]. From the devices that can run RIOT, we have used three sensing platforms available in the FIT IoT-LAB testbed [3, 4]: IoT-LAB M3, Microchip SAMR21 Xplained Pro and Arduino Zero. These devices, can sense different variables (*e.g.* air temperature, light, atmospheric pressure) and support wireless (IEEE 802.15.4) and wired (UART) communications. These devices are used for some experiments in Chapter 6.

2.1.2 Internet of Things

When the evolution of embedded systems allowed controlling them and accessing their data through the Internet in real-time, the concept of the *Internet of Things* (IoT) was born. Nowadays, IoT usually refers to enormous networks of interconnected devices that share information, processing capabilities, and tasks execution around the globe, leveraging fast Internet connectivity with cutting-edge wireless communications technologies. However, simpler systems such as *Wireless Sensor Networks* (WSN) or even a single Internet-connected device are also considered IoT [39, 56, 59].

Thus, a sensor-data application is also part of the IoT. However, most *IoT-data applications* are more complex, involving multiple sensors with advanced processing and communications capabilities (*e.g.* gateways) [40, 59]. Indeed, the main difference between sensor data and *IoT data* is that the latter refers to data acquired, computed and provided by various devices, where any of the participating devices may process a part (or all) of the data (*i.e.* distributed processing) [60, 61, 8].

The increased complexity of IoT-data applications also increases the challenges and possibilities of *IoT-based applications* [10]. Even though the data-acquisition subsystem becomes a data source, the number of devices that participate, their disposition and roles (especially for the data acquisition and processing) affect the output data and may constrain its further analysis

[19, 13]. Besides, the data variety in IoT allows extracting more valuable insights and enables Big Data analytics and data science [10, 62, 63].

From the vast range of IoT-data applications, this thesis focuses on WSN since they are one of the most common implementations for pure data acquisition, especially in agriculture [64].

Wireless Sensor Networks

A sensor network consists of intercommunicated sensors (and possibly more complex devices like gateways) that share a common monitoring goal. In this way, sensors share their gathered data throughout the network, where each node (connected device) has a specific role and assigned tasks, *e.g.* sense, process, transmit or upload the data [36, 64].

When the inter-node data transmission relies on wireless communication protocols, it becomes a WSN. IoT-data applications implemented as WSNs can affordably cover larger areas with difficult access. Therefore, multiple application domains, from the military to agriculture, leverage WSNs to acquire their data [64, 65].

The definition of the roles and distribution of the nodes will depend on the application requirements. WSNs have multiple options to organise the nodes, known as network topologies: Bus, Tree, Star, Mesh, Ring, Circular or Grid [65].

From these variate options, this thesis focuses on the Star topology. Nodes in this topology have two roles: sensor or sink. The sensor node gathers, processes, and sends data to the sink. The sink receives, processes, joins and uploads the WSN data. Usually, this node can also sense data. The Star topology is relatively simple (considering the Circular and Grid topologies [65]); yet, it offers a compelling subject of study since it has at least two differentiated roles and a hierarchical structure with open possibilities.

Usually, the particular requirements of each role demand different devices to meet them. Therefore, sink nodes have often more robust hardware, with increased energy, processing, storage and communication capabilities than the sensor nodes. These WSNs are known as heterogeneous networks. Nevertheless, low-end (class 1 [56]) devices can meet the requirements for all the roles [55], arranging a homogeneous WSN. For the sake of simplicity, this thesis focuses on homogeneous WSNs.

2.1.3 Conceptual modelling

A *conceptual model* is an abstract, partial and simplified representation of a system under study that enables analysis and reasoning without the limitation of the implementation [23, 66]. Models allow sharing a collective vision amongst technical (*e.g.* IoT and data engineers) and non-technical parties (*e.g.* domain experts and decision-makers), facilitating the communication between them. Therefore, it provides effective and efficient design, development, and maintenance processes for complex systems; and unbiased and precise project control [67, 23].

Moreover, *meta-models* allow defining instance models representing particular implementations of a broader area. In this way, a meta-model can be considered a "model that defines the structure of a modelling language" [23].

Formal modelling languages, such as UML or ER, provide standard notations (*i.e.* a meta-model) for the definition of models. This representation allows for a broad understanding and usage of the models and provides guidelines for their proper construction. Indeed, in literature, models not adhered to formal modelling languages are often disregarded [23, 66].

Furthermore, models in software engineering can provide more value beyond system abstraction. Model-driven approaches leverage the formalisation and definition of conceptual models to ease the development process, usually generating a significant portion of the required code [66, 22].

This thesis proposes a conceptual data model for model-driven sensor- and IoT-data applications. A UML profile formalises the data representation, following the principles of MDA to provide a complete model-driven approach.

Conceptual data model

A conceptual data model is the meta-model representation of a particular kind of data (*e.g.* spatial, temporal, graph), where each element of the meta-model defines a respective graphical notation. Usually, conceptual data models come with CASE (Computer-Aided Software Engineering) systems that allow (amongst other features) for an aided graphical interface and automatic code generation. These models allow non-technical users to focus on the data and analysis requirements, abstracting the complexity of underlying technologies [39].

Conceptual models are widely used to represent transactional data stored in databases, by means of UML and ER. However, they are also used in more complex contexts.

For example, in the context of BI, several works propose the usage of ER and UML extensions to represent Data Warehouses since they convey the offered analysis capabilities through simple data representation [12]. Conceptual data models are also advantageous for IoT-based applications (*e.g.* Stream and Big Data systems), including the definition and integration of such data [68, 36, 41].

UML profile

A formal meta-model is also an extension of the structure and notations of its modelling language. In UML, these extensions can be heavyweight (when the semantics change by modifying the standard UML meta-model) or lightweight (when the semantics change but still follow the standard UML meta-model) [69]. Lightweight extensions (*i.e.* profiles) preserve the UML standard and its wide acceptability [22]. Besides, they are easier to learn and understand.

The purpose of UML profiles is to customise UML for particular domains or platforms by extending its meta-classes (*e.g.* Class, Property, Package) [70]. A profile is defined using three key concepts: stereotypes, tagged values and constraints. Stereotypes directly extend meta-classes and can be represented through the inscription «`stereotype-name`» or an icon. For example, a stereotype *«Species»* extends the UML meta-class "Class" to define the representation of species. At the model level, every class using this stereotype will denote a group of living organisms. Tagged values (or simply *Tags*) are meta-attributes, *i.e.* they are defined as

static properties of stereotypes. For example, the kingdom of any species is not supposed to change and can be considered a meta-attribute. Finally, a set of constraints should be attached to each stereotype, precisely defining its application semantics to avoid an arbitrary use by designers in models. Such rules can be defined in the Object Constraint Language (OCL), which the CASE tools can automatically apply to the instance models. For example, an OCL rule can guarantee that any *«Species»* class has a descriptive attribute called "*breed*", and the CASE tool will alert when this rule is broken [36].

Model-Driven Architecture

Model-driven approaches consider conceptual models as first-class citizens in software development beyond their still relevant role for documentation and users involvement. MDA outstands from these approaches for its high level of abstraction and specification of the modelling architecture. The goal of MDA is to allow for seamless and broad adoption of the models for applications. Therefore, its models are expressed in a well-defined notation, divided into concerns and hierarchical abstraction levels, and underpinned in formal meta-models. Indeed, UML profiling is almost the standard practice for MDA proposals [66, 22, 25, 23].

The separation of concerns indicates that each model should address a unique issue of the system under study. For example, the data, security and scheduling are all relevant for IoT systems, but a model that addresses every topic becomes too complex and difficult to understand. Besides, the hierarchical division of abstraction levels refers to the different levels of independence of the particular implementation. MDA proposes five abstraction levels [66, 25, 24]:

1. Computation-Independent Model (CIM): This level should only comprise users' requirements without considering technological limitations. This thesis does not address the CIM since it is constrained to sensor- and IoT-data applications and assumes that users can directly discuss data at the PIM level.

2. Platform-Independent Model (PIM): The high-abstraction level of the application models its prime logic and attributes regardless of the underpinning platform. In this thesis, the PIM represents sensor and IoT data and their gathering, transformation and delivery processes.

3. Platform-Specific Model (PSM): The low-abstraction level of the application models its implementation into a specific platform. In this thesis, the PSM links the data with the particular facilities, roles, tasks and disposition of the IoT devices.

4. Platform-Model (PM): Models the implementation platform independently of the application. This thesis calls the PM as Device Model (DM); it represents the IoT devices.

5. Implementation Code: Although the code is not a model for MDA, it implements the application logic and attributes into the platform. Thus, the code is the ultimate realisation of a model-driven process. This thesis enables the automatic generation of sensors' code from the DM (*i.e.* PM) and PSM models.

Note that the PIM is more related to the applications' Functional Requirements (FR), while the PSM must deal with more specific Non-Functional Requirements (NFR). FRs state the system functionalities that supply end-users' needs [71]. Thus, FRs guide the definition and validation of the system architecture and logic (*i.e.* the PIM). Besides, NFRs complete the FRs with the expected constraints and qualities of each functionality [71]. In this way, the technical implementation features of the system (*i.e.* the PSM) rely on the NFRs. In the context of sensor- and IoT-data applications, the FRs refer to the data, calculations and analysis (*e.g.* daily average temperature); and the NFRs to the associated reliability, topology and technology (*e.g.* take the samples from three points using UEB nodes in a star topology) [71, 72].

2.1.4 Business Intelligence Systems: Data Warehouse

Business Intelligence (BI) systems are first-class citizens of decision-support systems. They allow for exploration and analysis of data stored in transactional databases. BI systems include reporting, data mining, statistical and Data Warehouses-OLAP systems [73].

Data Warehouse (DW) and OLAP systems are main tools for Business Intelligence. They allow for the analysis of huge volumes of data stored according to the multidimensional model. Warehoused data are organised in facts (*i.e.* analysis subjects) and dimensions (*i.e.* analysis axes). Dimensions are composed of hierarchies that allow analysing data at different thematic, spatial and temporal granularities. Facts are described by numerical attributes called measures, which are aggregated along hierarchies with common SQL aggregation functions such as Min, Max or Sum. OLAP systems allow exploring and aggregating measures via OLAP operators. The most common OLAP operators are Roll-Up and Drill-Down, which allow climbing and aggregating measures over dimensions levels, and Slice and Dice, which allow selecting a subset of the warehoused data. Usually, DWs are implemented using classical Relational DBMS using the star-schema approach, which denormalises dimensional data to improve queries performance. When data is integrated in real time or coming from stream sources, DWs are named real-time and stream DW, respectively [73, 67].

Since BI systems are built on the top of data sources, their conceptual representation is additional to the one used to represent data sources [67, 74].

2.2 Motivation Case Study

Agriculture is a complex and unstable economic activity vital for humanity. Amongst the multiple IoT applications in precision farming, this thesis focuses on smart irrigation to contextualise and motivate its contributions. Irrigation activities are very dependent on crop variety and the environmental conditions of the particular field or plot. Thus, sensor- and IoT-based decision-support systems provide significant aid for farmers and agronomists [75, 38].

The IRRINOV© method states three environmental variables that allow estimating the water deficiency of a particular crop and thus indicate whether it needs irrigation. The variables of interest are [76]:

- Daily median **soil moisture** sampled once a day from at least three measuring probes for each depth (30 and 60 cm).
- Daily maximum and minimum of **air temperature** sampled every 20 minutes. These measures provide daily heat accumulation and allow calculating the Growth Degree Unit (GDU).
- And daily accumulated **rainfall**.

IoT devices can provide these variables to avoid problems, costs, and limitations of manual collection. Moreover, an IoT-data stream reporting system would allow farmers to visualise the irrigation needs by crop and plot in real-time. Indeed, a BI system can suggest some irrigation actions to the farmers. The decision rule is: "*When the soil moisture is superior to a 'lack of water in soil' threshold, they should start irrigation*". The water deficiency level depends on the growth of the crops (*i.e.* GDU). For example, in the case of corn, if the GDU is superior or equal to 570 degree-days, the threshold for "*lack of water in soil*" is 160 cbar. Thus, the BI application will suggest irrigating the specific corn plot if the aggregated soil moisture is 161 cbar. Although, if the rainfall quantity is superior to 10 mm, the irrigation is unnecessary [39, 76].

The BI application must check the above-described decision rule at different spatial, temporal and thematic levels. In particular, the application must provide continuous monitoring of the sensed climatic data along these three analysis axes (*i.e.* dimensions):

- **Spatial**: intra-plot (sensor-level), plot and farm to optimise and prioritise the usage of the irrigation equipment over the farm.
- **Thematic**: crop and type of plant to provide the appropriate amount of water.
- **Time**: day and week to keep track of the irrigation activities over time.

This kind of IoT-based BI system can be a real-time Data Warehouse (DW) that continuously analyses sensors or IoT data (*i.e.* the fact) according to these three dimensions, organised hierarchically [19, 41]. Consequently, at least three kinds of experts participate and cooperate in the design and implementation of the IoT-based BI application: the agronomists and farmers (**business users**), who know the agricultural context and can extract value from the analysis; the **BI experts** or data scientists, who develop and use the DW (*i.e.* data-analysis subsystem); and the **IoT experts**, who configure and implement the sensing infrastructure (*i.e.* data-supply subsystem).

Developing and implementing the IoT system is complex since it concerns data, sensors, and network modelling issues. Figure 2.1 shows a simple implementation option for such a sensing application. It uses only one powerful sensor to sample, process and provide all necessary data [49]. However, multiple implementation options exist. For instance, Figure 2.2 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

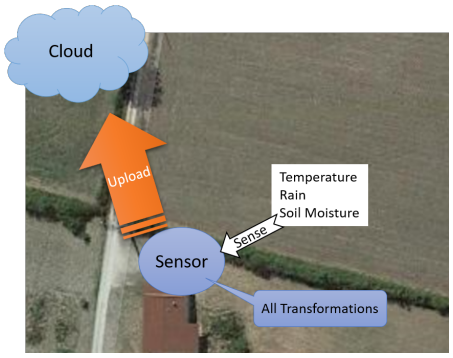

Figure 2.1: Sensor-data application example for IRRINOV© method.

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 [40].

These star topologies (*cf.* 2.1.2) use two kinds of sensor nodes:

- **Sink nodes** (SN) (such as a meteorological station and a laptop) [65], which have electrical power, the Internet connection, and significant computation and storage tools;
- **End nodes** (EN) (such as soil moisture sensors) [65], which have battery power, no Internet connection (but support some wireless protocols), and low computation and storage tools.

In all configuration options presented in Figures 2.1 and 2.2, the Internet access point is in the farmer's house, and thus one node (sensor or sink) is positioned there to upload data into the cloud. Also, temperature, rain, and soil moisture data are collected and uploaded to the cloud according to the [76] guide. Nevertheless, there are relevant differences between each option.

The example of Figure 2.1 uses only one sensor, eliminating the inter-node transmission costs and risks. However, it may have an increased cost for the physical installation of the soil-moisture probes into the plot.

The upper configurations in Figure 2.2 (Options A and 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-hours 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 inside the ENs [77].

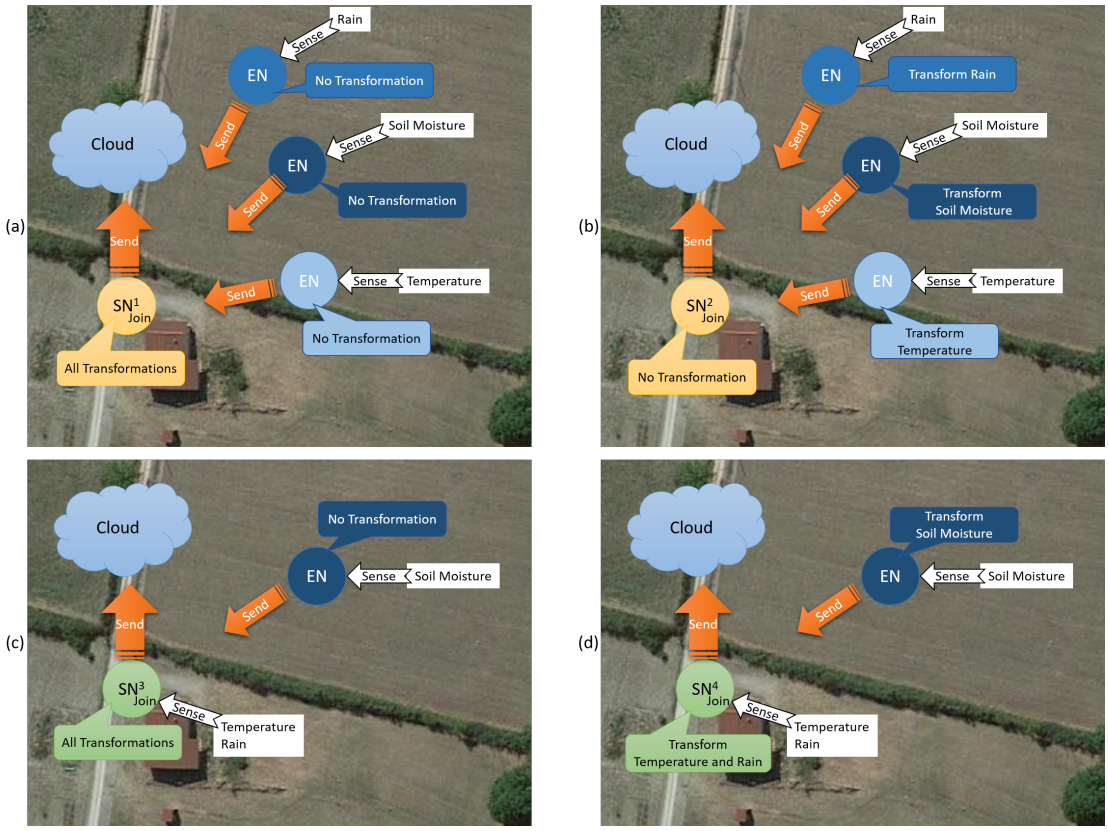

Figure 2.2: Some deployment options for the IoT-data application of the case-study.

The lower configurations (Figures 2.2-C and 2.2-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.

Figure 2.2-D represents the most efficient (multi-node) 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.

Figure 2.3 shows a simplified example of data and operations flows in the EN and SN of Figure 2.2-D. Both nodes collect their data in regular intervals denoted by T_x. The EN 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

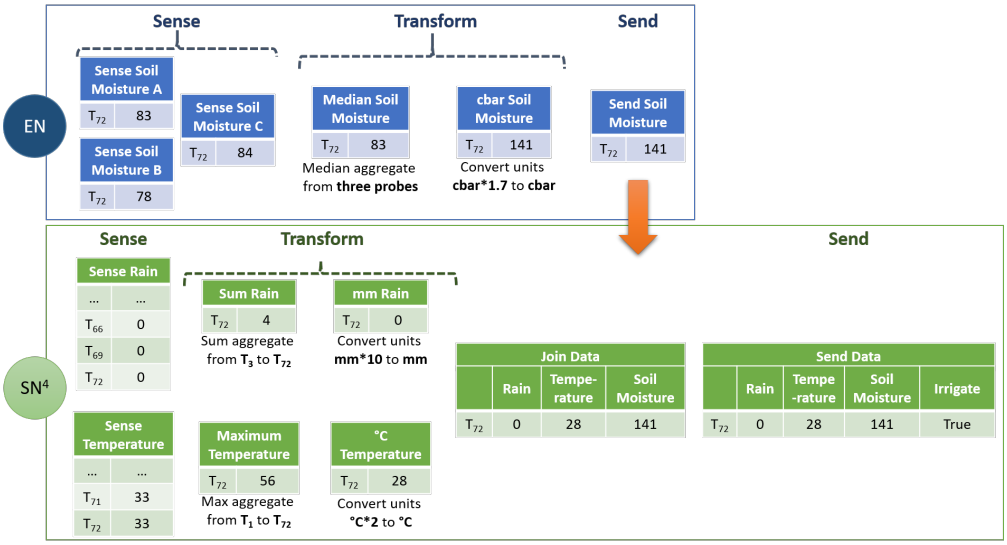

Figure 2.3: An example of simplified data and operations flows for the deployment in Figure 2.2-D.

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.

This case study shows the complexity and the diversity of the possible configurations of the same IoT data 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.

2.3 Requirements for Modelling Sensor-Data and IoT-Data Applications

Sensor- and IoT-based applications are usually data-centric applications. For example, in a BI system such as the one in the case study, OLTP (OnLine Transaction Processing) databases underpin the reporting and DWs support OLAP (OnLine Analytical Processing). Therefore, an effective conceptual data model representation is necessary for successful data-centric applications [73]. Indeed, traditional data models can be complex, involving several different kinds of data (spatial, temporal or thematic). IoT data increase the complexity of data models since they integrate continuous spatial-temporal data coming from sensors (*e.g.* sensed temperature)

to classical data and analysis.

Usually, the conceptual design of traditional data-centric applications involves business users and data scientists. The conceptual design of IoT-based applications also involves IoT experts, which distributes the responsibilities for the different parts of such applications. Business users interact with data scientists and IoT experts to define the data and analysis needs (*i.e.* functional requirements). Besides, data scientists and IoT experts focus on their respective (BI and IoT applications) implementations and NFR [6, 9, 78].

Even slight variations in the teams can generate compatibility, communication or performance issues in the system [17, 31]. In this context, conceptual data modelling is an effective and mandatory support to formalise the definition and development process, providing graphical representations that abstract the technical details of implementation and enable the involvement of all the roles. Conceptual models for IoT-based applications should be particular and systemic. Particular since each subsystem has specific issues that the respective expert must address. And systemic since both subsystems must cooperate to achieve a common goal [79].

Besides, an MDA-based approach defines a highly abstract data model to discuss amongst business users, data scientists and IoT experts; while also providing a specific implementation model and the (semi) automatic generation of implementable code. Nevertheless, there are multiple data-centric MDA approaches for BI and database systems in the literature [9, 73, 13].

Therefore, this section presents the (new) requirements for effective conceptual models of sensor- and IoT-data applications exemplified in the case study. Moreover, it explains the need for MDA support in these design approaches.

2.3.1 Main concepts to design sensor-data applications

Considering the characteristics of sensor data and sensor-data applications, a conceptual data model for this kind of system should focus on four data aspects [15, 20]:

- The *type of sensed variable* (*e.g.* temperature) since each variable has particular usage purposes, sources and computations.
- The *temporal validity* of the data since new values are always sensed and transmitted, replacing the old ones and thus determining a validity period. In this sense, the temporal validity depends on the *sensing* and *sending frequencies* [80].
- The *operations* that transform raw data (*e.g.* aggregation) since it constrains the further processing of sensor data. For instance, computing the mean from a set of averages is inappropriate. Note that a sensor-data application model should at least represent data aggregation (*cf.*2.1.1).
- The *temporal windows* on which the operations can compute a finite set of data [80].

Moreover, the data models must be readable for all the roles, including non-technical users [81]. Good readability increases the involvement of decision-makers since they can focus on

the data requirements and their further analysis [27]. It should thus disregard the complexity of technical features of implementation systems, *e.g.* the usage of a particular sensor interface and protocol or the data denormalisation used by DW [20]. In the same way, it should allow for transparent integration of sensor and classical data, *e.g.* associating the sensed temperature data to the corresponding plot and crop data for their analysis [39].

Consequently, the MDA approach should [19, 12, 27, 32]:

- Clearly *define the modelling concern*, *i.e.* IoT data in this case, since additional concepts, though relevant in the actual implementation, reduce readability and increase complexity.
- Separate the data model into *different abstraction levels*, allowing all the roles to converge their efforts for the FR definition while also enabling each expert to focus on their systems' implementation and NFR.
- Provide scope for classical and IoT data integration at the high-abstraction level. Indeed, considering that business users need to analyse the sensor data to extract value [9], a conceptual data model of sensor-data applications must also allow for an easy *integration with BI data models*.
- Define *model-to-model transformation* rules, which will further ease the development process.
- Similarly, define an *automatic and error-free code-generation* process from the models.

2.3.2 Main concepts to design IoT-data applications

Beyond the modelling concepts for sensor data, which are still relevant, IoT-data applications have a higher degree of complexity derived from the association of multiple features. First, IoT data evolve over continuous 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 [8]. Second, IoT devices can only compute data they handle (*i.e.* sense or receive). 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 [61]. And third, IoT also comprises advanced devices that can process data beyond aggregation functions.

Therefore, a conceptual data model for IoT-data applications has further requirements regarding data transformations, data communications, and MDA:

- *Data Transformations* inside the IoT must include at least three primary pre-processing transformations [82]:
  - *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 [52].

  - *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, the conceptual model should be *easily extendable* to include more transformations.

- A model of *Data Communications* between IoT objects must present three main features for a correct representation of data transmission and integration:
  - The explicit representation of the *Join* of data streams from external (and internal) sources.
  - The capacity to define *Composition Associations* of data from different sources.
  - And the abstract representation of the *IoT Network*.

- Finally, the *MDA* must allow designing *multiple implementation options* of the same data. Besides, it should support the *code generation* of each (correctly defined) implementation option.

Summary

The adoption of IoT (including sensors) has enabled the development of automatic monitoring applications, which acquire valuable data about a monitored subject and its environment. IoT-based applications analyse Sensor and IoT data to obtain insights that can support decision-making and thus benefit from the data. For example, the case study is an IoT-based application for irrigation in precision farming. A BI system receives and analyses IoT data from a plot in three dimensions to schedule irrigation activities, while the IoT-data application has multiple implementation options with different quality and economic issues.

This relatively constrained example shows that IoT-based applications are highly complex. It relies on multiple diverse systems that involve various roles in their definition, development and implementation. Therefore, a conceptual data model is necessary to abstract the complexity and differences of the underlying systems and provide a framework for collective discussion and design. Since multiple approaches exist for modelling BI systems, this thesis focuses on IoT- and sensor-data systems. The main concepts that a conceptual data model for IoT should comprise are the type of variable, its temporal validity, the operations (including at least aggregation, conversion and filter), and the temporal window for computing the data. Besides, the application representation should also define the network of sensors, the composition of data from various sources, and the Join of such data.

Moreover, the MDA principles have several advantages for conceptual data modelling. It allows defining a highly abstract data model and automatically implementing the system from it. This modelling guide includes the definition of concerns, separation in hierarchical abstraction

levels, model-to-model transformation, and model-to-code transformation for every implementation option.

Some of the outcomes of this chapter are published in [36, 39, 40]. In particular, [39, 40] relate the same case study, [36, 39] introduce the concepts to model sensor-data applications, and [40] presents the concepts to design IoT-data applications.

Chapter 3

Related Work

This chapter presents a thorough analysis of the closely-related research, providing valuable insights for the positioning and development of this thesis. It follows a combined approach based on scientometrics and literature review [1] to acquire and analyse the most relevant related works in model-driven IoT (Figure 3.1). In the first place, it presents the study methods (Sec. 3.1). Then, it presents the outcomes from the scientometric analysis (Sec. 3.2). Finally, it classifies and analyses the related work (Sec. 3.3).

3.1 Study Methods

Figure 3.1 depicts the main stages of our study, stating the number of yielded or analysed documents from each step. In the first step, we used the Scopus® and Web of Science® (WoS) databases to acquire all the research on model-driven IoT. In particular, the search string was `(model-driven OR (automatic AND "code generation")) AND (IoT OR "internet of things" OR sensor)`. This search returned a total of 1461 documents. In the second step, we limited our analysis to papers written in English after 2010, removing multiple conference reviews, editorial materials and errata besides those documents from previous years or in other languages. This filter reduced the document count to 1106. In the third step, the ScientoPy® tool removed the duplicates from the two databases, further reducing the number of papers to 848.

Then, we proceed with the scientometric analysis of such papers (step four in Figure 3.1), focusing on data-related topics. This analysis should identify the main topics and trends in the research area. Moreover, we emphasise on the study of the data-related topics to spot those papers working on data modelling. We used three tools [2] to analyse the research area: ScientoPy®, VosViewer® and natural language processing with the Natural Language Toolkit® (NLTK). ScientoPy® [83] shows the evolution of selected topics, especially from the index and author keywords. VosViewer® [84] defines word clusters and their relationships from the abstracts. And we used NLTK [85] to identify the associations of words (as bi-grams and tri-grams) and their evolution in titles and abstracts. This process allowed us to detect 105 interesting doc-

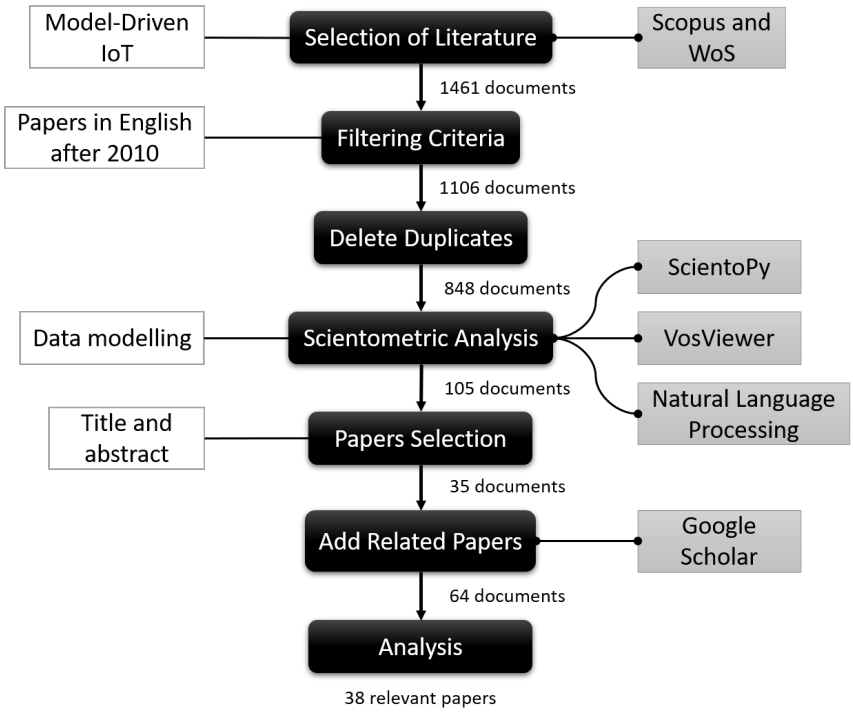

Figure 3.1: Scientometric analysis and literature review process based on [1, 2].

uments referencing data-related topics in their abstracts. Section 3.2 presents the results of the scientometric analysis.

In the fifth step (Figure 3.1), we reviewed the title and abstract of the interesting documents and dropped those that seemed irrelevant and the publications from this thesis ([36, 39, 40]), further reducing them to 35 relevant works. Furthermore, in the sixth step, we searched these papers in Google Scholar® to identify possible updates and other relevant citing works, increasing the document count to 64.

For the last step (Analysis), we studied and classified each paper to identify their description of IoT data and their MDA compliance. Nevertheless, we discarded 26 documents for being same-study reports (using the most recent or complete in the classification) or not reporting at least one of the two relevant topics (IoT data modelling and model-driven approach). Section 3.3 presents the results of the literature review.

3.2 Scientometric Analysis

This section presents an overall analysis of the research around model-driven IoT. This scan provides valuable insights about the evolution of this research area, the current trends and mature topics. Besides, it allows identifying the relevant related work for this thesis.

The first step in the scientometric study is the generation of a word cloud from the index and author keywords as an overview of the research topics (Figure 3.2). This overview shows that *automatic programming* (including *code generation* and *automatic code generation*) and *software design* (including *software architecture*) are the most common topics after *IoT* and *model-driven approaches*. These keywords are promising since several model-driven proposals fail to generate implementable code [21]. Another relevant topic is *interoperability* (including *semantics* and *cloud computing*), which relates to the relevance of using IoT and IoT data. However, the word cloud (Figure 3.2) only shows two data-related topics (with a relatively low frequency): *big data* and *information systems*.

Consequently, we checked the use and trends of different model-driven approaches and modelling languages (Sec. 3.2.1). Moreover, we emphasised in the analysis of data-related topics to discover trends and relevant works regarding IoT data modelling for their further review (Sec. 3.2.2).

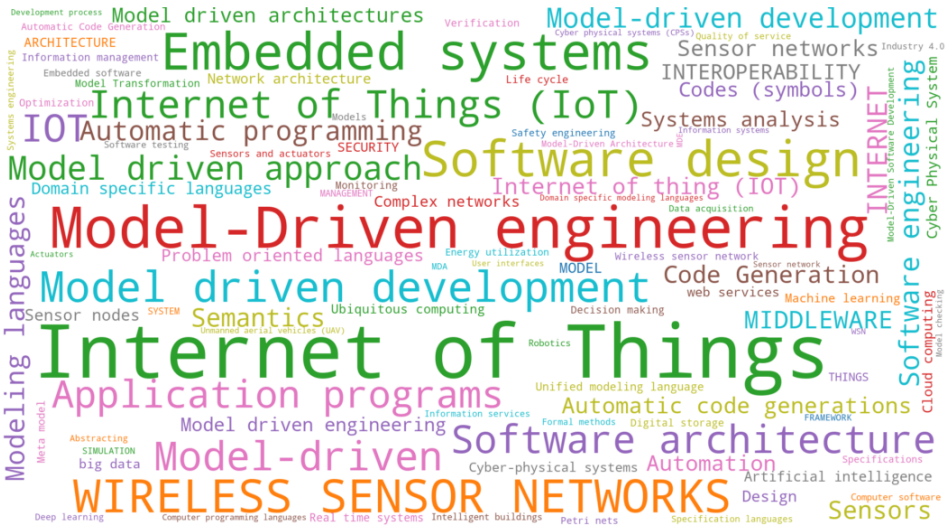

Figure 3.2: Overview of the research topics in model-driven IoT.

3.2.1 Model-driven approach study

We selected 11 relevant topics regarding different model-driven approaches and modelling languages for this study: *MDA* (Model-Driven Architecture), *MDD* (Model-Driven Development), *MDE* (Model-Driven Engineering), *Ontology*, *OWL* (Web Ontology Language), *UML* (Unified Modelling Language), *UML profile*, *SysML* (System Modelling Language), *DSL* (Domain-Specific Language), *EMF* (Eclipse Modeling Framework) and *MOF* (Meta-Object Facility). Then, we analysed the evolution of these topics in the authors and index keywords (Figure 3.3), and in the abstracts (Figure 3.4).

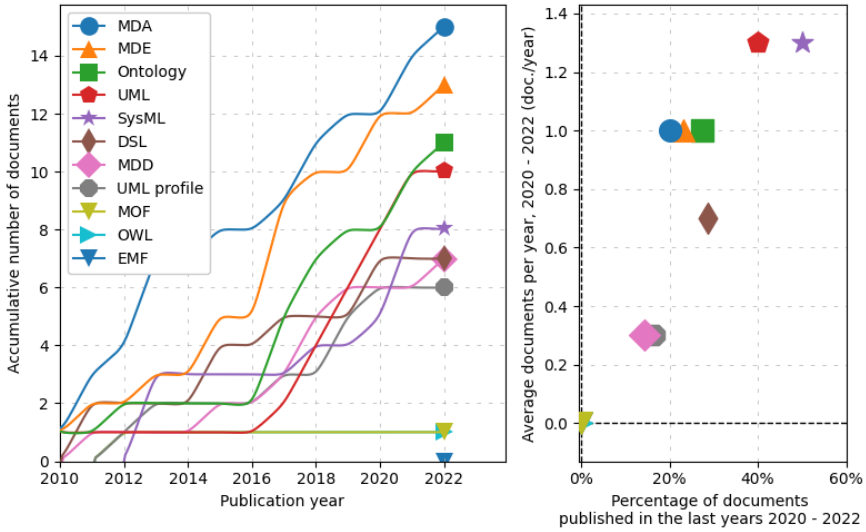

Figure 3.3: Evolution of the modelling topics in the authors and index keywords.

The evolution of the modelling topics in the keywords (Figure 3.3) shows that both *MDA* and *MDE* approaches have the highest relevance in the modelling-related keywords. Besides, both topics have maintained their relevance in the last three years. Regarding the modelling languages, *Ontology* and *UML* are the most common, followed by *SysML* and *DSL*. However, *UML* and *SysML* are the topics that had the highest relevance in the previous years.

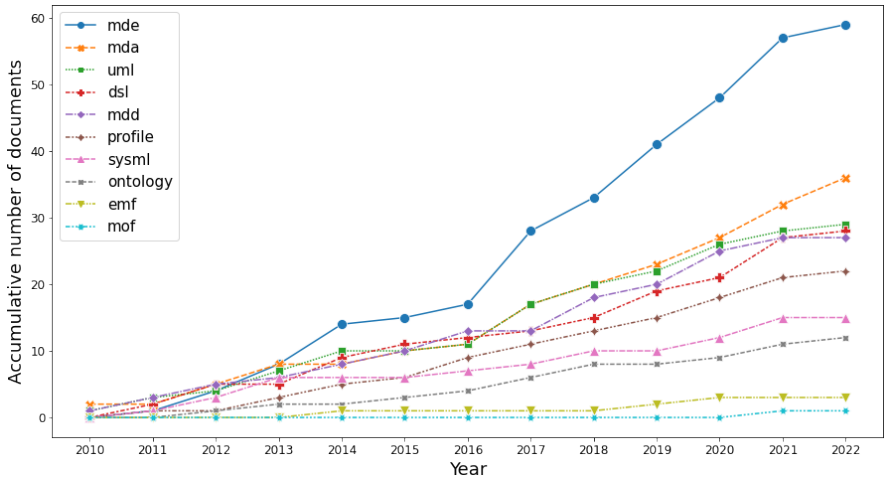

Figure 3.4: Evolution of the modelling topics in the abstracts.

Notwithstanding these trends, the abstracts show a broader view of the topics with more

samples (Figure 3.4). Regarding the model-driven approach, *MDE* and *MDA* are still the most common approaches, steadily increasing over time. However, *MDE* is significantly more used than *MDA*, while *MDA* has maintained its relevance better than *MDE* in the last three years. Besides, *MDD* is almost as common as *MDA*.

Moreover, the most common modelling languages in the abstracts are *UML* and *DSL*, followed closely by *profile*. However, these topics (and all the observed modelling languages) have reduced their relevance in the last year since there are less publications referring them. Yet, not all 2022 papers are already published and their numbers might increase.

These results show that even though the modelling topics have gained relevance in the research area, the presence of standard model-driven approaches and modelling languages is low. Consequently, more work is required in formal frameworks and meta-models that enable model-driven IoT.

Other studies have analysed the literature around model-driven IoT, obtaining similar results [21, 86, 87, 88, 89, 90]. In particular, [21, 87, 88, 89] stated UML and DSL as the most common (formal) languages for model-driven IoT. Nevertheless, [86, 89, 90] emphasise on the lack of formality to generate the IoT code.

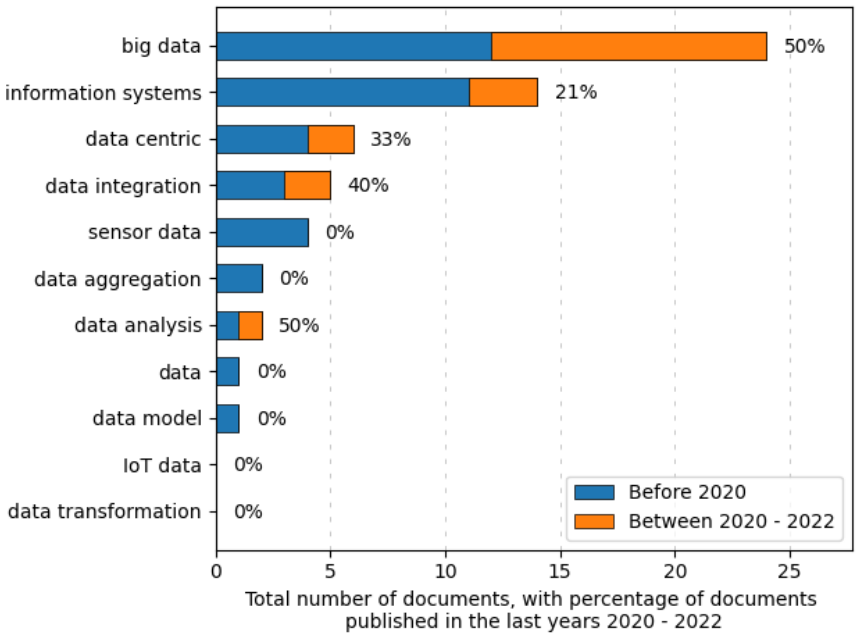

Figure 3.5: Appearance and recent relevance of the selected data-related topics.

3.2.2 Data modelling study

We selected 11 data-related topics to analyse their appearance and recent relevance in the papers' keywords (Figure 3.5), including those detected in Figure 3.2: *big data*, *information systems*, *data model*, *data integration*, *data-centric*, *IoT data*, *sensor data*, *data aggregation*, *data transformation*, *data analysis* and *data*.

Figure 3.5 shows that *big data* is the most relevant of the selected topics, although it only appears in 24 documents. This topic has recently increased its importance considering that 50% of the papers are from the last three years. Other topics that have increased their relevance lately are *data analysis*, *data integration* and *data-centric* with 50%, 40% and 33%, respectively. However, these topics appear in less than eight documents, *i.e.* 1% of the total documents.

Excepting *big data* and *information systems*, the data-related topics appear in very few documents. Besides, most of them were not relevant in the last three years. This outcome indicates that works on model-driven IoT are more focused on the system description, and the data modelling is an open field.

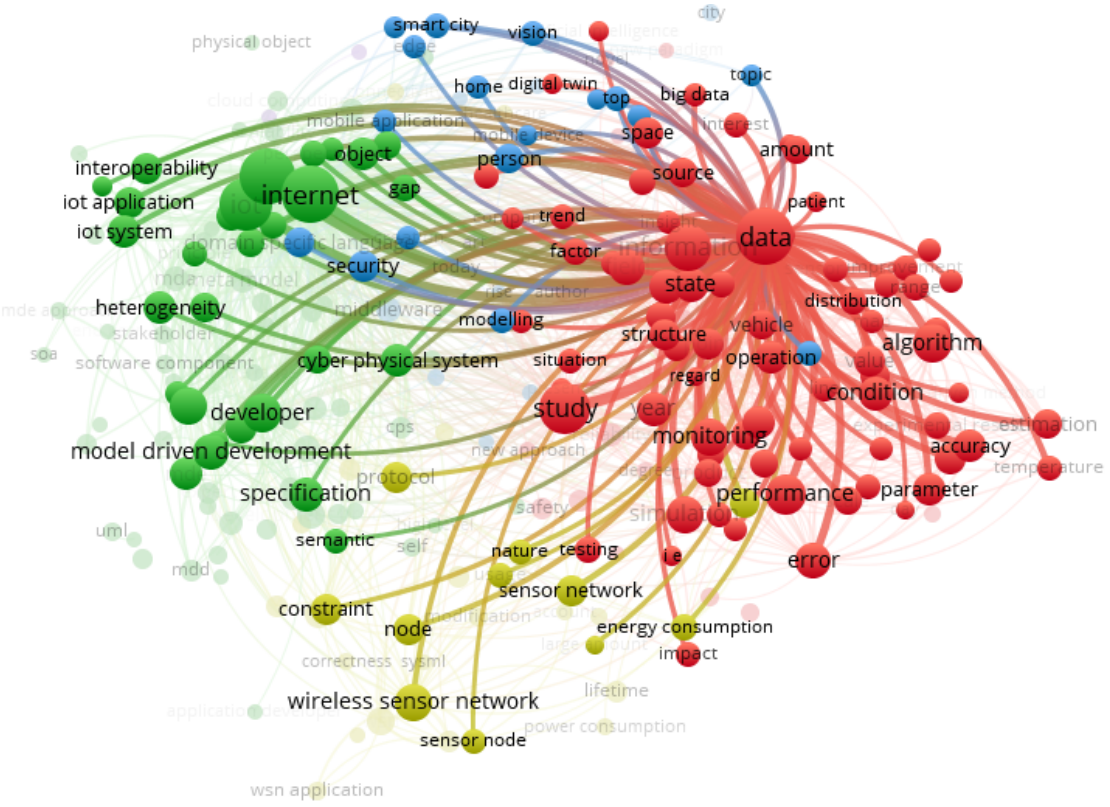

Figure 3.6: Data cluster map.

Notwithstanding the low relevance of data in the keywords, the abstracts' analysis offers

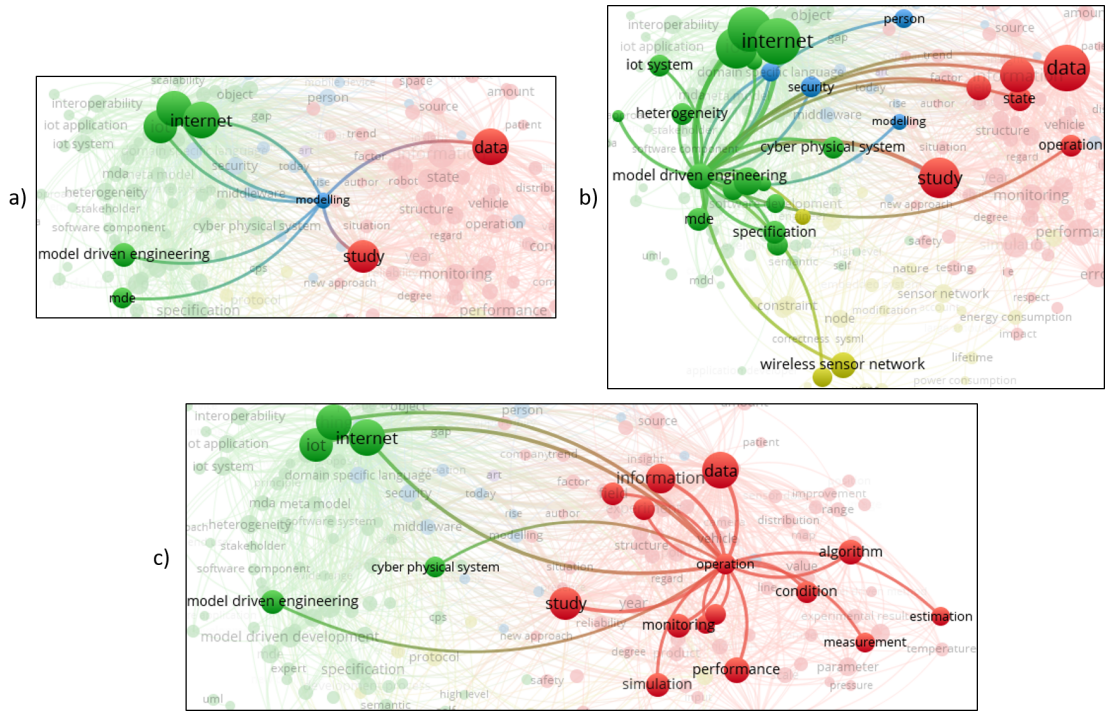

Figure 3.7: Cluster maps for modelling (a), model-driven engineering (b), and operation (c).

a different perspective. Figure 3.6 shows the cluster map for *data*. Indeed, data is a main cluster with several relationships, including *IoT*, *WSN*, *modelling* and *model-driven engineering*. Consequently, data is a transversal topic in several papers in the area. It has a high relevance but is not the main topic of research.

If *data* was a main research topic, it should also appear in the keywords (Figure 3.2) with a higher frequency. This difference between the abstract and the keywords may mean that authors generally refer to data in their abstracts because it is necessary but do not address data modelling. Figure 3.7-A evidences this situation. Modelling should be a main bridge between *data* and *MDE*. Instead, it is a small cluster with few relationships. Moreover, such a lesser cluster in a model-driven context raises doubts about whether the papers do report models in their MDE proposals.

Indeed, *MDE* connects with multiple words (Figure 3.7-B): *study*, *data*, *IoT*, *WSN*, *security*, *person*, *heterogeneity* and *operation*, amongst others. These associations are interesting since they show potential applications of MDE in IoT. Yet, only *IoT*, *data* and *study* connect with modelling (Figure 3.7-A). Consequently, there is a high possibility that most works leverage part of the MDE concepts to automatically generate solutions to different issues without providing a thorough modelling step.

For instance, data operations are one of the main concepts of sensor and IoT data (Sec. 2.3). Its representation should consider types (*e.g.* aggregation, conversion or fusion), the existence of

analysis windows, execution frequencies, or implementation algorithms. Even though the map has an operation cluster associated with data (Figure 3.7-C), its relationship with simulation, performance and estimation besides algorithm indicates that these papers are more focused on the implementation than the design.

Nevertheless, there are highly relevant related topics that the map alone cannot fully identify. Consequently, we used NLTK to discover the most common bi-grams and tri-grams (*i.e.* sets of two and three words) in the titles and abstracts of the papers. In this process, we unified the text format, removed special characters and dropped connectors, prepositions and conjunctions (*i.e.* stopwords) to reduce the noise and increase the consistency of the identified bi-grams and tri-grams.

Table 3.1 shows the title tri-grams and their frequency. The first and fourth tri-grams are the most relevant since those papers will likely provide IoT data models.

Table 3.1: Most common data-related tri-grams in the titles.

Tri-gram	**Papers**
model-driven data acquisition	3
data distribution service	2
data quality management	2
multi-sensor data fusion	2

Besides, Table 3.2 shows the abstract tri-grams and their frequency. The most promising tri-grams are *data acquisition systems*, *model-driven data acquisition*, *model-driven data-driven methods*, *complex data processing* and *model-driven data*. The other topics seem generic for most IoT applications and are not direct indicators of a data model. Nevertheless, we included all these papers for the review.

Table 3.2: Most common data-related tri-grams in the abstracts.

Tri-gram	**Papers**
large amount data	6
protocols data formats	4
data acquisition systems	4
model-driven data acquisition	4
model-driven data-driven methods	4
real sensor data	3
complex data processing	3
model driven data	3
remote sensing data	3
big data analytics	3

Regarding the bi-grams, the most relevant topics from the titles (Table 3.3) are *model-driven data*, *data fusion*, *data acquisition* and *data driven*. Compared to Table 3.1, Table 3.3 shows that

there are at least two topics for *model-driven data* besides *acquisition*, which will be relevant. There are also two additional papers about *data fusion* and one about *data acquisition*.

Table 3.3: Most common data-related bi-grams in the titles.

Bi-gram	Papers
big data	7
model-driven data	5
data fusion	4
data acquisition	4
data driven	4
data using	3
data based	3
traffic data	3

Furthermore, the data-related bi-grams from the abstracts were more common and numerous, allowing for trend analysis. Therefore, Figure 3.8 shows the evolution of the most common and relevant topics. *Sensor data* is the most common term, although it lost relevance in the previous years. Instead, *IoT data* has steadily gained relevancy since 2017, although it only accumulates ten papers. Similarly, *data analytics* has increased its importance but still seems to be a novel topic. One subject that had a medium frequency but has lately stalled is *data acquisition*. Indeed, even though it is closely related to *data collection*, *data collected*, and *data obtained*, these topics have lost relevance in the last year. Nevertheless, these four topics appear in 49 unique papers, making them the most common data-related subject.

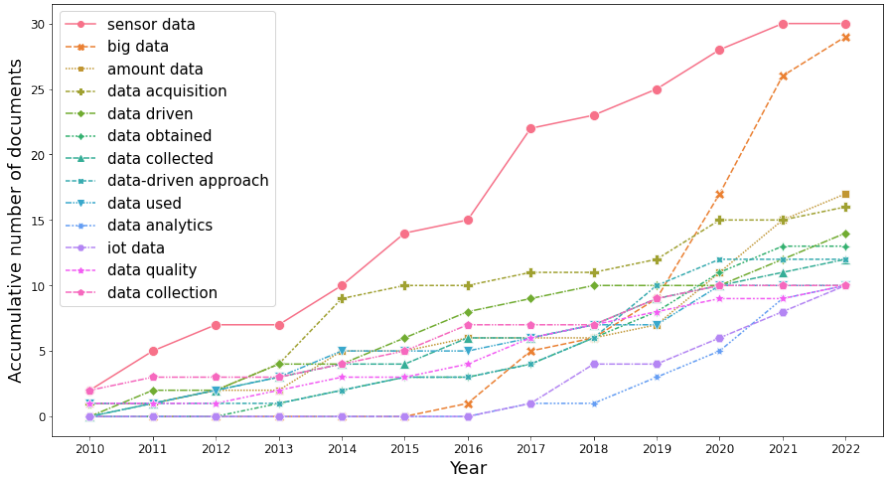

Figure 3.8: Evolution of the most common data-related bi-grams in the abstracts.

These results are consistent with previous literature reviews [21, 86, 87, 88, 89, 90]. These

works state that the most common modelled topics are service composition, application code and controllers, the network, and systems integration. However, only [89] reported IoT data amongst the (least frequent) modelling topics.

From the analysis of data-related topics in the abstracts, we identified 105 papers that could potentially report a model-driven approach for IoT focused on data. We focus on these papers for the literature review (Sec. 3.3).

3.3 Literature Review

This section presents the classification and analysis of the selected papers considering their data representation and MDA compliance regardless of their model-driven approach. This classification considers the main concepts to design sensor- and IoT-data applications and the MDA principles defined in Sec. 2.3.

To ease the display of the papers, we divide the classification into two parts. First, we check whether the related works define the modelling concepts for sensor-data applications (Sec. 3.3.1, Table 3.4). Second, considering that sensor-data concepts underpin the IoT-data features, we select the most relevant related works from Table 3.4 to verify their ability to model IoT-data applications (Sec. 3.3.2, Table 3.5).

3.3.1 Sensor data classification

Table 3.4 structures the classification of the related work using the following features:

- Existence of a **Conceptual Data Model** to represent the sensor data in different applications (Sec. 2.1.3).
- Presence of the **Sensor Data Features** required to design sensor data (Sec. 2.3.1): *Operations*, *Window*, *Sensing frequency*, *Sending frequency* and *Additional data features* (representation of other data characteristics).
- **Integration with BI Data Models** (Sec. 2.3.1).
- Presence of **Model-Driven Features**, *i.e.* characteristics that defines a complete MDA approach (Sec. 2.3.1): *Separation of implementation information* (*i.e.* different abstraction levels), *Separation of concerns*, *Model to model transformation* and *Code generation*.
- And **Scope**, *i.e.* the application domain.

Moreover, Table 3.4 divides the related work in two groups: Databases for Sensor, Real-time and Stream data; and IoT (including WSN and sensors).

Conceptual models for stream databases (including sensor and real-time data) extend classical data-centric conceptual models to integrate the features of sensors data as entities or classes in ER or UML formalism, respectively. Although they provide an easy integration of external

Table 3.4: Initial classification of the related work considering the sensor-data design.

Paper	Conceptual Data Model	Sensor Data Features					Integration with BI Data Models	Model-Driven Features				Scope
		Operations	Window	Sensing frequency	Sending frequency	Aditional data features		Separation of implementation information	Separation of concerns	Model to model transformation	Code generation	
Sensor, Real-time and Stream Databases												
[91]	Yes	No	No	Yes	No	Yes	Yes	No	No	No	No	Real-time databases
[92]	No	Yes	No	No	No	Yes	Partial	No	No	No	No	Data Cleaning
[93]	Yes	Yes	No	Partial	No	No	No	Yes	Yes	Partial	No	Stream data applications
[94]	No	Yes	No	No	No	Yes	Yes	No	No	No	No	IoT-data analysis
[95]	No	No	Yes	No	No	Yes	Yes	Yes	Yes	No	No	IoT-data analysis
[96]	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Real-time databases
[74]	No	No	No	No	Yes	Yes	Yes	No	No	No	No	Big Data systems
[97]	No	Yes	Yes	No	No	Yes	Yes	No	No	No	No	IoT-data analysis
[13]	Yes	Yes	Yes	No	No	No	Yes	Yes	No	No	No	Stream data BI Applications
[37]	No	Yes	No	No	No	Yes	Yes	Yes	Yes	No	No	IoT-data analysis
Internet of Things												
[98]	No	Yes	No	Partial	Partial	No	No	No	No	No	Partial	Monitoring applications
[99]	No	Yes	Partial	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Monitoring applications
[17]	No	No	No	No	No	No	No	No	Yes	No	Partial	IoT applications
[30]	No	Yes	No	Yes	No	Yes	No	No	Yes	Yes	Yes	IoT applications
[100]	No	No	No	No	No	No	No	Yes	Yes	No	Yes	Industrial IoT communications
[18]	No	No	No	Partial	Partial	Yes	No	No	Yes	No	Yes	WSN reconfiguration
[101]	No	Yes	Partial	Yes	No	Yes	No	No	Yes	No	No	Monitoring and action-taking in smart vehicles
[31]	No	No	No	No	No	No	No	No	Yes	No	Yes	Monitoring and action-taking applications
[102]	No	No	No	No	No	Yes	No	No	Yes	No	Partial	Systems interoperability
[103]	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	IoT-based BI applications
[104]	No	Yes	No	Yes	Yes	Yes	No	No	No	No	Yes	IoT applications
[105]	Partial	Yes	Partial	No	No	Yes	No	No	Yes	Yes	Yes	IoT applications
[58]	No	Yes	Partial	Yes	No	Yes	No	No	No	No	Yes	IDE for IoT applications
[106]	No	No	No	No	No	Yes	No	No	Yes	Yes	Yes	Model-driven methodology for IoT
[107]	No	No	No	Yes	Yes	No	Partial	No	No	No	Yes	IoT applications
[108]	No	No	No	No	No	No	No	Partial	No	Partial	Yes	Healthcare data collection
[109]	No	No	No	Partial	No	Yes	Partial	No	No	No	No	Structural Health Monitoring
[110]	No	Yes	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Service-oriented IoT
[111]	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Integration of IoT applications
[112]	No	Yes	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Integration of IoT applications
[60]	No	No	No	No	No	No	Yes	Yes	Yes	No	No	IoT as Microservices
[113]	No	Yes	No	No	No	No	Yes	Yes	Yes	No	Yes	Integration of IoT applications
[114]	No	No	No	No	No	No	Yes	Yes	No	Yes	Yes	Integration of IoT applications
[57]	No	No	No	Yes	No	Yes	No	No	No	No	Yes	IoT applications
[115]	No	No	No	Yes	Partial	No	Yes	No	Yes	No	Yes	IoT applications
[116]	Partial	Yes	No	No	No	Yes	Yes	No	No	No	Yes	IoT implementation and data analysis
[117]	No	Yes	No	No	No	Yes	Yes	No	No	No	Yes	IoT applications modelling
[33]	No	No	No	No	No	Yes	Partial	Yes	Yes	Yes	Yes	IoT applications

and BI data (since sensed data is represented in the same way as classical data), they lack in representing advanced sensors characteristics and do not provide any implementation in sensors OS. Indeed, most implementations are directed to the extension of classical DBMSs to receive temporal data.

On the other side, the main goal existing work in conceptual modelling for IoT is the design of all features concerning the physical implementation of sensors and networks. As expected from the scientometric analysis (Sec. 3.2), data plays a minor role in most of these works. They usually represent a flow of operations through the network nodes and even external systems. In this way, they provide a complete representations of IoT-based applications while disregarding the data use and analysis in BI systems.

In the following, we present the details of the fist group of works.

[91] presents a UML profile for real-time databases, where sensor data is represented as a UML class. They define particular stereotypes to define the timestamp and the validity duration of sensed data, and some types of sensing frequency.

[92] proposes a framework for cleaning and integrating IoT data into sensor databases. This framework relies on domain information from the IoT, the original data requirements, and the definition of patterns and alerts. Although it allows executing different operations on the IoT data for its preparation, the database receives the raw data and executes this process.

[13, 93] present a conceptual model for stream data based on UML and ER formalism, respectively. These models allow the conceptual representation of the transient aspect of stream data (*e.g.* operations on the data and windows). However, they are generic to stream data and do not fully represent the main characteristics of IoT data (*e.g.* frequency of acquisition/delivery, operation window). Moreover, they do not support any model-driven or automatic code generation approach for IoT.

[94] defines a method for integrating public sensor data available in diverse sources into a sensor database. Such a sensor database allows executing various operations and transformations on sensor data. However, it does not consider the concept of window for executing the operations since the data comes from static sources and not directly from the IoT.

[95] proposes a model-driven approach for integrating and analysing IoT data in databases. This approach maps the IoT data into stream databases and enables its analysis in temporal windows, although direct operations on data (*i.e.* to alter the sensed data) are not considered.

[96] presents a UML profile for the definition of design patterns for real-time databases. This approach allows modelling several characteristics of real-time data, including IoT data, and its integration into real-time analysis systems. However, it disregards the stream nature of IoT data, which is evidenced by the lack of an analysis window in the profile.

[74] presents a UML data model for the ingestion layer of big data systems, including sensors (IoT) as a data source. However, the data model is not complete, and includes several items corresponding to implementation details like memory size or hardware sensitivity. It does not also consider any separation of concerns or aid for developing the IoT.

[97] proposes the use of Complex Event Processing (CEP) engines for the analysis of IoT data in real time. This approach defines a simple mechanism for integrating IoT data and allows

transforming and analysing it in windows. However, it relies on existing IoT implementations.

[37] defines an architecture for integrating big data systems, IoT components, and knowledge-based systems for cutting-edge smart farms. In particular, it combines BI and data mining technologies to build comprehensive solutions underpinned on the sensing capabilities of IoT. Therefore, this architecture defines the operations and integration of IoT data. Yet, it does not provide mechanisms for the generation of the required IoT code.

As shown in Table 3.4, most works on IoT do not provide conceptual data models, but a generic representation of IoT data features. We discuss them in the following since they highlight the main modelling features for IoT data. Moreover, modern software engineering methods are mandatory for the design of Sensor- and IoT-based BI applications. Indeed, several works are focused on the system description to automate the implementation. Therefore, Table 3.4 also analyses this aspect despite the lack of conceptual data models.

[98] defines a UML-based development environment for WSN. In this environment, developers define the nodes and network behaviours in activity diagrams and rules, while nodes can directly run the models through an interpreter. However, the modelling approach does not allow for integration with BI data models, combines the data- and implementation-related details, and does not provide any separation of concerns.

[99] defines a Model-Driven Development (MDD) -based methodology for the design and implementation of WSN. A set of meta-models that describes the WSN from the level of nodes, groups and data-flow supports the methodology with a complete data representation. However, they do not consider the integration with BI data models and combine the implementation details with the data description.

[17] proposes a meta-model and model-driven methodology for the design and development of IoT applications. In this approach, they separate the concepts, components and roles related to the domain, deployment, functionality, platform, design and development. Moreover, they help the development and implementation process through semi-automatic code generation. However, it does not consider representing the IoT or BI data, providing only on the kind of sensed variable.

[30] defines a meta-model and rule-based programming language for IoT applications following an MDA approach at the level of IoT nodes. In this work, CIM and PIM are the rules, while PSM is the code for each specific platform. With this consideration, it provides both the model-to-model transformation and the automatic implementation (as code generation) in the same phase. This approach considers important data characteristics such as operation and sampling frequency, yet it fails on defining other relevant features like the window, which is necessary for operating stream data.

[100] provides a UML profile based on the OMA LWM2M (Lightweight machine-to-machine communication protocol) and standard Internet-Protocol Smart Objects (IPSO) for connecting and managing Industrial IoTs. The proposed profile constitutes an approach to automate the integration of mechatronic components in the IoT environment through the automatic generation of the LWM2M layer, leveraging IoT protocols in the development process of manufacturing systems. However, it does not consider IoT data or its operations in the profile.

[18] presents a UML meta-model for the design and implementation of reconfigurable WSN, including several features that enable automatic configuration in the nodes. Despite this meta-model represents various data aspects of IoT such as the sensing and sending frequencies (as timestamps), it does not consider the kind of sensed variable or any operations that modify the data. Besides, it does not consider the integration with data analysis systems like BI.

Besides its representation of IoT data, [18] considers some elements of a model-driven approach. Indeed, even though it does not consider different abstraction levels for the data and the implementation issues, it considers the separation of concerns between sensors and actuators and provides aid to the code generation process.

[101] provides a set of UML meta-models for the design of advanced driver assistance systems based on sensors and actuators. This work presents an almost complete representation of IoT data, including operations like data fusion and windows defined as a memory span. However, the integration of these models is not explicit, they do not help the implementation of the system, and the paper does not provide insights into the integration of IoT data into data analysis systems.

[31] defines a modelling and development framework for self-adaptative IoT based on MDE and MARTE. This MDE approach provides a separation of concerns in hardware and software modules, defining the multiple relationships amongst the multiple components that must cooperate in an IoT application. Besides, it also provides automatic implementation through code generation. However, it does not consider the particular issues and characteristics of IoT data, neither it defines any mechanism for integrating IoT data into BI systems.

[102] provides an MDE methodology and approach for interoperability between IoT- and cloud-based services, focusing on protocol and format compatibility. In their MDE approach, they separate the concerns of the application from the specification and define the roles of developers and testers. Moreover, this work achieves the integration of IoT data to the cloud, yet it focuses on the data format and communication protocols without representing IoT or BI data.

[103] defines a high-level framework for the definition of cloud-based mobile applications following the Model-View-Controller and MDA paradigms. Indeed, it provides a set of modelling patterns and general meta-model that allow defining the different components of such a complex system. However, it focuses on the integration of processes, and thus it does not describe the IoT or BI data.

[104] explains how ThingML, a model-driven approach for IoT applications, helps diversify the use of IoT in multiple domains. ThingML defines a platform-independent specification for the early development stages, describing components that do not depend directly on low-level system or hardware features. Besides, once developers choose a concrete IoT platform, the remaining components are implemented by leveraging the particular hardware features and available software libraries.

[105] presents a UML profile for Wireless Sensor and Actuator Networks (WSAN) with an MDA approach. The PIM profile defines a behavioural model to represent sensors' data and control flows, and a structural model to represent the network components and topology. Despite (partially) considering all the necessary data features, the PIM behavioural model is strongly

associated with the network topology. Besides, the PSM profiles are linked to the Embedded Operative System and are thereby difficult to use in different scenarios.

[58] presents a cloud-based IDE for developing IoT applications using a model-driven approach. Besides, it provides a mechanism for seamlessly deploying the applications in the FIT IoT-LAB [3] for experimentation. COMFIT (*i.e.* the IDE) allows representing some data features of IoT such as some operations and tuple-based windows. However, it does not provide an explicit mechanism for merging or composing data from different sources.

[106] proposes a methodology for the development of IoT applications following the Service-Oriented Architecture (SOA). The methodology uses a model-driven approach to build the applications, separating the business requirements, business logic and the actual solution. It also considers a brief representation of sensor data without transformations or frequencies.

[107] describes a model-driven process to implement and use IoT applications. The process starts from the network modelling, simulation, deployment and use. In this way, it describes both the data sensing and sending and the data loading into a database with different implementation options. Yet, it only allows for the uploading of raw data.

[108] defines a model-driven framework for developing mobile applications that collect healthcare data. The main approach for data collection is self-reporting in the mobile devices, although it could also integrate data from separate sensors. Nevertheless, it does not describe the sensor data or the integration mechanisms.

[109] studies different modelling frameworks for IoT systems and proposes a MOF-compliant meta-model for IoT, focusing on structural health monitoring and building information modelling. Although this approach clearly models the data, it only considers a partial representation of the sensing frequency and the kind of variable. Moreover, this approach does not consider any model-driven features like the separation of concerns.

[110] 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.

[111] 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.

[112] 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.

[60] 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.

[113] 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.

[114, 57] propose easy-to-use approaches for the simple definition and implementation of IoT. [114] defines AutoIoT, a user-oriented 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. [57] 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.

[115] 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.

[116] introduces a novel approach for optimising the energy consumption in IoT using Artificial Intelligence. It leverages historical IoT data to generate simple Machine Learning models that can directly run into the IoT objects. It automatically implements the models in the IoT devices and a cloud server. If the predicted value is close to the sensed value the devices avoid sending new data to reduce the energy consumption. Notwithstanding its relevance, this approach does not provide a representation of IoT data or its transformations.

[117] proposes SimulateIoT, a model-driven approach for simulating and implementing IoT applications. SimulateIoT allows for the definition of large IoTs integrated into BI and other data analysis systems. However, its description of IoT data is focused on the implementation data types (*e.g.* float or integer), disregarding the type of variable or the sensing and sending frequencies of IoT data.

[33] presents MontiThings, a model-driven approach for developing fault-tolerant IoT applications. It uses a component and connector architecture to build the applications, separating the business logic from the implementation details. It allows developers to implement complete applications that include complex error-handling mechanisms without digging into the details of such mechanisms. Nevertheless, it does not consider the execution of data operations inside the IoT objects.

From this study of the related works (Table 3.4), we have evidenced a lack of effective

conceptual data models for IoT-based BI applications. Firstly, *approaches from the database domain fail in providing all the required details of IoT data and helping the IoT implementation process*. Secondly, *approaches from the IoT domain fail to propose conceptual data models while also generating the applications' code*.

Consequently, we have selected the papers defining sensor data operations and code generation as the most relevant for the second classification process: [98, 99, 30, 104, 105, 58, 110, 112, 113, 116, 117].

3.3.2 IoT data classification

Table 3.5 classifies the most relevant related work considering the extended features of IoT data design:

- Modelling and implementation of **Data Transformations inside the IoT** (Sec. 2.3.2): *Aggregation*, *Conversion*, *Filtering*, *Additional Transformations* (*i.e.* other data operations explicitly included), and *Easily Extendable*.
- Modelling and implementation of **Data Communications between IoT objects** (Sec. 2.3.2): *Join*, *Composition Association of data*, and IoT *Network*.
- **Multiple implementation options** (Sec. 2.3.2) refers to the capacity of the model-driven approach to define and generate the code for multiple implementation options for the same data requirements (*i.e.* PIM or CIM).

Table 3.5: Further classification of the most relevant related work for the IoT-data design.

Paper	IoT Data Transformations					IoT Data Communications			Multiple Implementation Options
	Aggregation	Conversion	Filtering	Additional Transformations	Easy Extension	Join	Composition Association	Network	
[98]	Yes	No	Yes	None	No	No	No	Yes	No
[99]	Yes	No	No	None	No	Yes	No	Yes	No
[30]	Yes	No	Yes	None	No	No	No	Partial	No
[104]	No	No	No	None	Yes	No	No	Yes	Yes
[105]	Yes	No	No	None	Yes	No	No	Yes	Yes
[58]	Yes	No	No	None	No	No	No	Yes	Yes
[110]	No	Yes	No	Calculations	Yes	No	Yes	Yes	Yes
[112]	Yes	No	No	None	No	No	Yes	Yes	No
[113]	No	Yes	No	None	No	No	No	Yes	No
[116]	No	No	No	None	Yes	No	Yes	No	Yes
[117]	No	No	Yes	None	No	Yes	No	Yes	Yes

Table 3.5 shows that the most common data transformation in IoT design is Aggregation, which conforms with the importance of this operation [52]. Nevertheless, not all the authors included it into their models. The other selected transformations (Conversion and Filtering) appear in 18% and 27% of the papers, respectively. Besides, 27% of the papers represent more than one transformation: [98, 30] model Aggregation and Filtering, while [110] models Conversions and Calculations on data. Indeed, [110] is the only paper defining an additional transformation

besides the three basic ones extracted from [82]. Nevertheless, 36% of the papers provide mechanisms that ease the extension of their models for the definition of new transformations. From these papers, [104, 116] do not specify any data transformation; instead, they allow designers to define the necessary operations according to their needs.

Considering the IoT data communications features (Table 3.5), almost all the papers allow designers to represent a network. [30] only provides a partial representation of the network since it focuses on developing sensor applications. Designers can make two sensor nodes communicate and cooperate in this approach by designing their individual behaviours. Besides, [116] cannot represent a network since it focuses on the execution of machine-learning models at node level. Moreover, 27% of the papers allow creating composition associations for data from multiple sources. From these works, [110, 112] provide both a network representation and associate the data of different sensors. Besides, [116] do represent composition association of data (in the central databases and analysis systems) but not the network. Furthermore, only 18% of the works ([99, 117]) explicitly represent the joins. These works do not represent the composition of data. However, their network representation creates the implicit relationships amongst different sensors, which is declared with kinds of join operators.

Finally, even though all the most relevant related work can generate implementable code, 45% of the papers can only produce one implementation option from their abstract representation of the solutions (PIM).

Consequently, this scientometric and literature review study has consistently shown the relevance of data modelling and model-driven approaches for sensor- and IoT-based applications. However, we have not identified works presenting conceptual data models for IoT in a model-driven paradigm. Indeed, several authors propose interesting approaches that aid the implementation of IoT applications considering some data features. Yet, the data representation is not complete, and the models usually combine implementation information, which reduces the readability and understandability of the models.

The results of our study are consistent with those of previous literature reviews [21, 24]. Therefore, this thesis can effectively contribute to the research area by providing a data-centric approach to developing sensor- and IoT-data applications following the MDA principles.

Summary

The adoption of model-driven techniques for sensor and IoT applications has gained high relevance in the last decade. The most common approaches have been MDE and MDA. These approaches leverage the modelling capabilities of UML and SysML or the adaption capacity of DSLs and UML profiles for their definition.

Proposals in this area focus on building IoT applications from a conceptual level. The data generation capacity of IoT is a common topic in several works. Indeed, the word data is among the most common words in the abstracts besides information, internet, and IoT. However, our literature analysis reveals that authors have not entirely addressed the data modelling of IoT systems to ease their implementation using a model-driven approach.

Even though multiple works define some main concepts for sensor and IoT data to generate the applications' code, their data representation is not exhaustive. Indeed, only five papers model three or more sensor-data features and generate the required code. However, none of these papers provides a complete representation of IoT data.

Consequently, we found no works providing conceptual data models for IoT that generate the application code in a model-driven approach. This situation reduces the ability of business users and data experts to influence the design of IoT-data applications, reducing the perceived value of existing models in complex integrated applications.

Most of the finding of this chapter are published in [39, 40]. Particularly, [39] refers a part of the sensor data classification, while [40] refers the IoT data classification.

Chapter 4

Design and Implementation of Sensor-Data Applications

This chapter addresses the modelling of sensor-data applications based on one sensing device and data aggregation. For first, it describes the design and the implementation of sensor data, then it details how these sensor data models can be integrated into conceptual BI models.

For sensor data, this chapter provides an overview of the whole model-driven approach and a methodology that guides the definition of the models and their further implementation (Sec. 4.1). Then it presents a UML profile (Sec. 4.2) that formally abstracts the main requirements defined in Sec. 2.3.1. It follows the MDA guidelines to provide different abstraction levels and enable implementation. Moreover, Sec. 4.3 describes the implementation process from the low-abstraction level of applications to actual IoT hardware.

Then, for the association of sensor data and BI systems, Sec. 4.4 describes the integration of our UML sensor data profile in DW conceptual models.

4.1 Sensor Data Design Methodology

The proposed UML profile has two main parts: (i) **The Sensor Device Model Profile (SensorDeviceModel)**, and (ii) **the Sensor Data Application Model Profile (SensorDataModel)**; which are then declined according to MDA principles (*cf.* 2.1.3) in **Platform-Independent** (PIM) and **Platform-Specific** (PSM) models (Figure 4.1).

The *SensorDeviceModel* represents the implementation sensor (*e.g.* the UEB) by enabling the definition of the measurable data, the available communication interfaces, the time management, the code-specific memory variables, and the application execution. Besides, the *SensorDataModel* uses the SensorDeviceModel to create a conceptual representation of data used by the decision-making application.

Figure 4.1 presents an overview of the design process of sensor-based models using our UML profile. First, sensor experts help business users (decision-makers and domain experts) to represent their view of the needed data (data-centric functional requirements) with

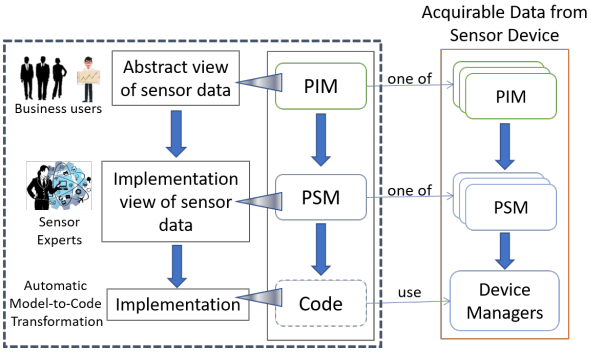

Figure 4.1: MDA Process for generating sensor-data applications leveraging our profile.

the SensorDataModel-PIM, using instances of the SensorDeviceModel-PIM. Then, the sensor experts map these conceptual data in the platform specification with the SensorDataModel-PSM using instances of the SensorDeviceModel-PSM, describing the implementation view of the needed data. Finally, the IoT code for the specific sensor platform is automatically generated by our tool.

4.2 UML Profile for Sensor Data

This section presents our MDA-based data-centric UML profile for sensor-data applications, including the meta-models and some examples of constraints in Object Constraint Language (OCL). We do not address the MDA CIM since we assume that our business users (*i.e.* BI experts and decision-makers) are able to directly define data-centric PIM models in UML.

4.2.1 Sensor-data application model

Application model PIM

This part of our profile allows for the high-level definition of the sensor-data applications (PIM). In the first place, the SensorDataModel-PIM profile (Figure 4.2) enables the decision makers and domain experts to define the data-related functional requirements of the sensors in a simple, standard and readable manner.

The SensorDataModel-PIM defines `A_PIM_Data` as the main Package, which contains the `A_PIM_Measure` Classes. These Classes describe only the data that is available for the user or other sensors. Thus, it only defines the sensed or aggregated variables and the delivery Operation. An `A_PIM_Measure` Class might have at least one or more variables, but it must have one delivery Operation.

For the delivery (a required Operation), there are two options:

1. `A_PIM_TupleDelivery`, which sends the data after a specific number of tuples or

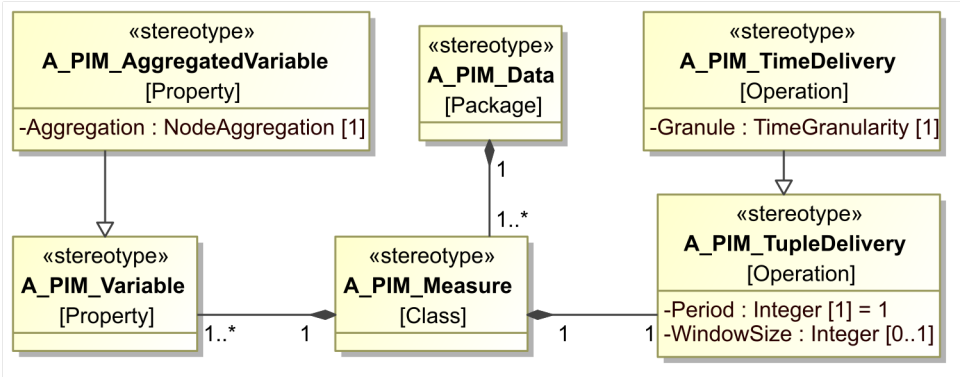

Figure 4.2: PIM profile for sensor-data applications.

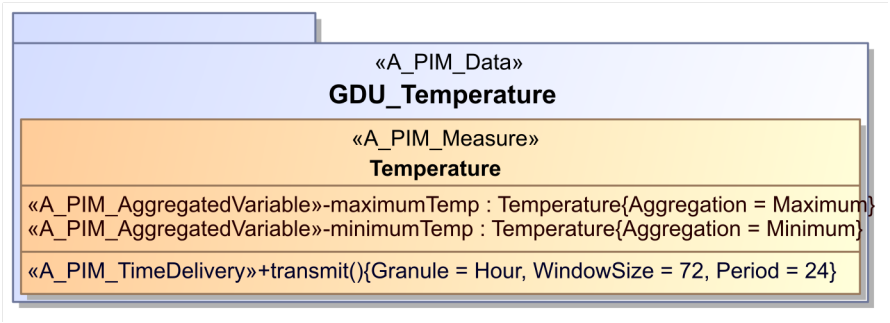

Figure 4.3: Use example of the sensor-data PIM profile for an application acquiring and aggregating temperature.

samples have been gathered (defined as `Period`) and defines the number of tuples that the node must keep as a `WindowSize`.

2. `A_PIM_TimeDelivery`, which sends the data periodically according to the `Period` and the time `Granule` of that `Period`. Both Operations define `WindowSize` in the same way.

For the raw variables, the `A_PIM_Measure` defines the `A_PIM_Variable` Property. Let us note that, the type of the `A_PIM_Variable` Property must be issued from the variables of the `D_PIM_Data` Package of the SensorDeviceModel detailed in Sec. 4.2.2 (as an example in Figure 4.9), forcing decision-makers to select data types from available devices (or, when required, the sensors experts to increase the supported variables). Moreover, the `A_PIM_Measure` also defines the `A_PIM_AggregatedVariable` Property for variables that are pre-processed inside the sensor with an aggregation operation.

Example 1 (SensorDataModel-PIM): Figure 4.3 presents a PIM for sensed (air) temperature values of our irrigation case study. The node is gathering some (72) temperature measurements

or samples, calculating the maximum and minimum temperature from these measurements, and delivering the daily maximum and minimum temperature values. Note that the type of sensed variable is chosen amongst the possible types (*i.e.* Temperature, Pressure, *etc.*) defined in the SensorDeviceModel-PIM of Figure 4.9. ❑

Furthermore, the SensorDataModel-PIM profile defines some rules in OCL to ensure that its instances (*i.e.* PIM models) are well-formed and semantically coherent.

Example 2 (OCL for SensorDataModel-PIM): Figure 4.4 shows one rule for the `Window-Size` Tag of `A_PIM_TupleDelivery`. It states that `WindowSize` cannot have zero or negative values since it represents a quantity of data. ❑

```
context A_PIM_TupleDelivery inv requiredWindowSize_PIM:
(owner.oclAsType(A_PIM_Measure).ownedAttribute
->exists(a|a.oclIsTypeOf(A_PIM_AggregatedVariable))
implies ((not self.WindowSize.oclIsUndefined()) and self.WindowSize>0))
```

Figure 4.4: Example OCL for the SensorDataModel-PIM profile concerning `WindowSize`.

Application model PSM

In the second place, the SensorDataModel-PSM (Figure 4.5) allows sensors experts to define the data-related implementation details of applications running on specific sensors, considering the functional requirements previously defined in the SensorDataModel-PIM. Since we have focused on the data, in the centre of the profile, we have defined the `A_PSM_Measure` abstract Class stereotype, which establishes the definition of all the data handled by the sensors. The `A_PSM_Measure` is composed of one or more `A_PSM_Variables` with a specific `Order`. Moreover, a `A_PSM_Variable` could have some pre-processing defining one `Aggregation` function (`A_PSM_AggregatedVariable`). Furthermore, the type of `A_PSM_Variable` Properties must be issued from the variables of the `D_PSM_Data` Package of the SensorDeviceModel-PSM in Sec. 4.2.2 (as an example in Figure 4.9). Nevertheless, the selected type must be `KindOf` the simple type established by the domain experts in the PIM. Indeed, as it is defined in Figure 4.9, this "`KindOf`" association grants that sensors experts are forced to use the type of data selected by decision-makers.

The main Package for describing the application in one single sensor is `A_PSM_Node`. The node must have an `ID` and an `Address` for communications. Additionally, the `A_PSM_Node` contains one or more `A_PSM_MeasurePacket` Packages, which group different interconnected `A_PSM_Measures` that help to accomplish one goal.

Furthermore, a sensor node should have two kinds of measures: the internal, private measures for *sensed variables* (`A_PSM_GatheredMeasure`); and the external, public measures for *delivered variables* (`A_PSM_DeliveredMeasure`). These Classes must be related by a `GatheredAs` Association.

In the left side of the SensorDataModel-PSM profile (Figure 4.5) we have the `A_PSM_GatheredMeasure` Class stereotype, a specification of `A_PSM_Measure`. This

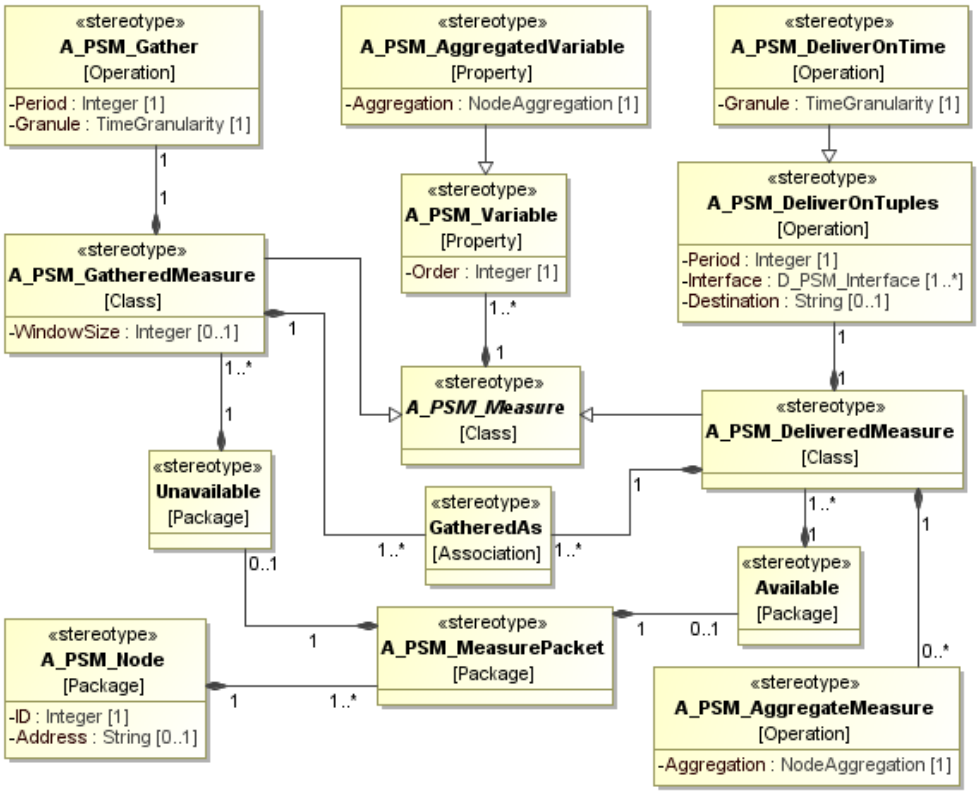

Figure 4.5: PSM profile for sensor-data applications.

Class enables the modelling of data being sensed directly from the probes, which exist only inside the sensor and are `Unavailable` for other nodes and the users. It must define one periodic Operation for sensing the data: `A_PSM_Gather`, along with the sensing `Period` and time `Granule` (*e.g.* second or millisecond). Also, if certain future operations (*e.g.* aggregation) require to keep more than one instance of gathered data, the `WindowSize` Tag must define the number of kept instances.

Furthermore, gathered data must be delivered in order to make it available for the users and other sensor nodes.

In the right side of the SensorDataModel-PSM profile (Figure 4.5) we have the `A_PSM_DeliveredMeasure` Class stereotype, a specification of `A_PSM_Measure`. This Class enables the modelling of data sent by the sensor to other nodes or the users, which is thus `Available`. It has to define one periodic Operation for sending the data with two options: `A_PSM_DeliverOnTuples`, if the `Period` is the number of gatherings before delivering; or `A_PSM_DeliverOnTime`, if the `Period` is an amount of time (also using the `Granule` Tag to define the time granularity). Nevertheless, both delivery options must define the output `Interface` and the `Destination` address.

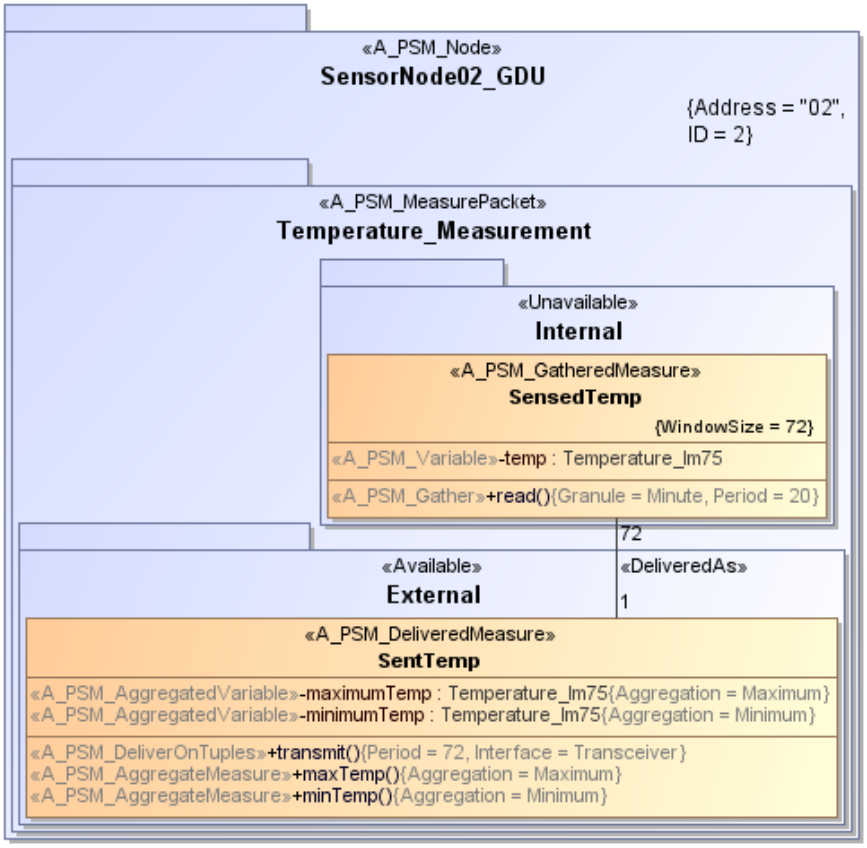

Figure 4.6: Use example of the SensorDataModel-PSM profile for a sensor-based application that needs aggregated temperature data for calculating the GDU.

Moreover, the sensor node could aggregate the data before delivering it by defining the `A_PSM_AggregatedVariable` with `Aggregation` and the corresponding `A_PSM_AggregateMeasure` Operation.

Example 3 (SensorDataModel-PSM): Figure 4.6 presents the PSM for an application running on the UEB device for the GDU defined in the case study. Every day, this example application gathers 72 temperature samples, finds the maximum and minimum values (aggregation), and sends only the two aggregates. In the top part of the model (Figure 4.6), *`SensedTemp`* is a `A_PSM_GatheredMeasure` with a `WindowSize` of 72 that senses one temperature variable from the LM75 probe every 20 minutes. *`SentTemp`* is a `A_PSM_DeliveredMeasure` in the bottom part of the PSM that aggregates the temperature values to calculate GDU (Figure 4.6). This Class sends the maximum and minimum values of 72 temperature samples (`WindowSize` in the associated `A_PSM_GatheredMeasure`) every 72 gatherings, *i.e.* every 24 hours (`Period` on the `A_PSM_DeliverOnTuples` Operation), as established for the GDU

in the case study and the SensorDataModel-PIM (Figure 4.2). Let us note that the *gathered* and *delivered* data types, as well as the delivery `Interface` are chosen by the sensors expert from the SensorDeviceModel-PSM example of Figure 4.9. ❑

Example 4 (OCL for SensorDataModel-PSM): The SensorDataModel-PSM profile also defines OCL rules to ensure that its instances (*i.e.* PSM models) are well-formed and semantically coherent. Figure 4.7 shows one rule for `A_PSM_GatheredMeasure`, which have to define the `WindowSize` Tag if there are any `A_PSM_AggregatedVariable` in at least one related `A_PSM_DeliveredMeasure`. ❑

```
context GatheredAs inv requiredWindowSize_PSM:
((self.relatedElement->exists(e|e.oclIsTypeOf(A_PSM_DeliveredMeasure)
and e.oclAsType(A_PSM_DeliveredMeasure).ownedAttribute
->exists(at|at.oclIsTypeOf(A_PSM_AggregatedVariable))))
implies (self.relatedElement->exists(e|e.oclIsTypeOf(A_PSM_GatheredMeasure)
and ((not e.oclAsType(A_PSM_GatheredMeasure).WindowSize.oclIsUndefined())
and e.oclAsType(A_PSM_GatheredMeasure).WindowSize>0))))
```

Figure 4.7: Example OCL for the SensorDataModel-PSM profile concerning `WindowSize`.

4.2.2 Sensor device model

The second part of our profile allows sensors experts to model the available resources (mainly data) that are relevant for developing applications in one specific sensor device, *i.e.* the implementation platform. In other terms, the SensorDeviceModel can be seen as a catalogue of sensors' software and hardware facilities (*e.g.* for the UEB with SEOS, Figure 4.9) that can provide some data.

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

Device model PIM

In the first place, the SensorDeviceModel-PIM (upper left corner in Figure 4.8) allows for a simple and abstract representation of the platform data, ignoring any implementation details and presenting a generic interface to access its sensed data. To describe these data, we defined a simple schema: a measurable variable (`D_PIM_Variable` Class) that has at least one value

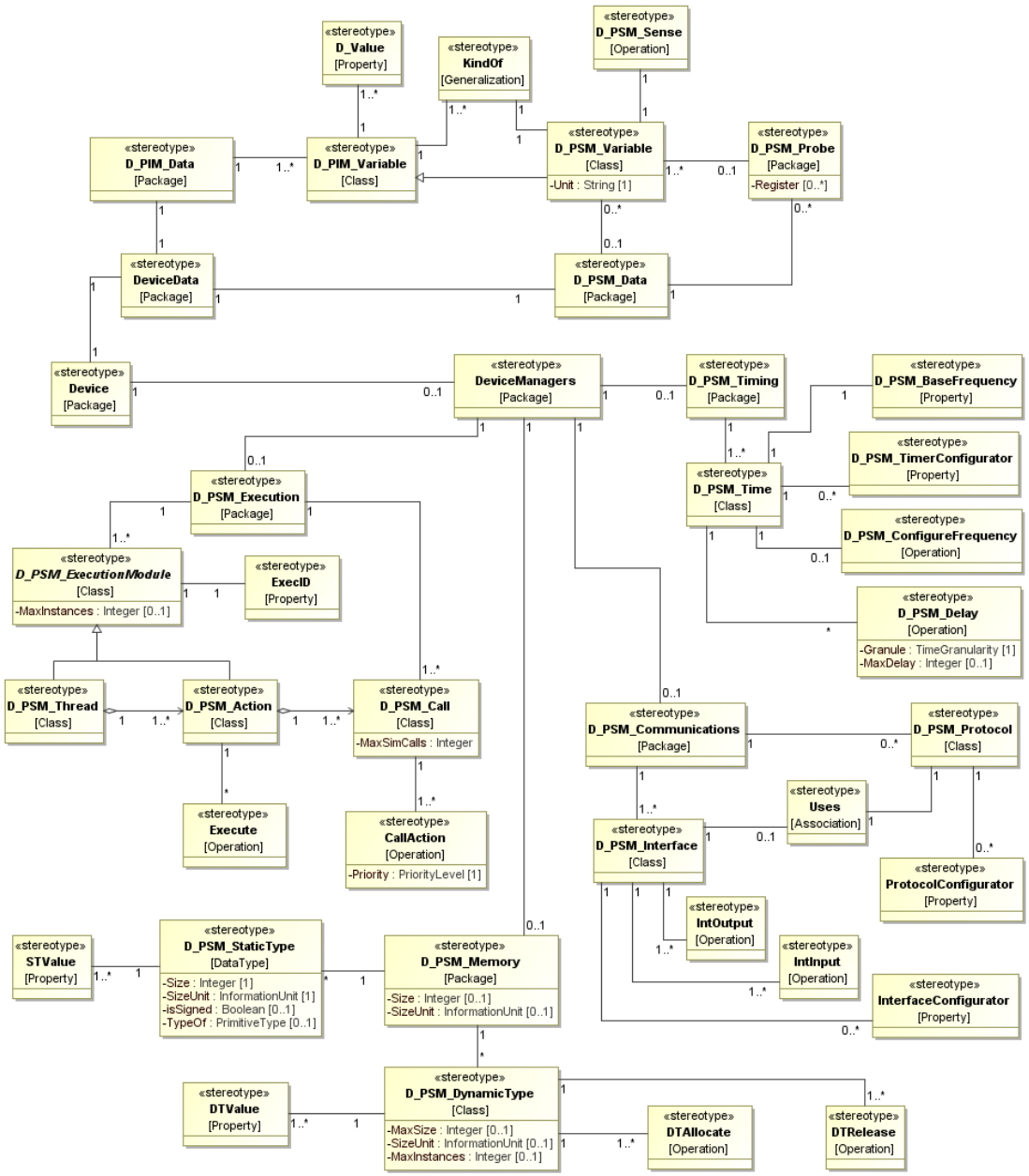

Figure 4.8: DM profile for sensor devices (*cf.* 2.1.1).

(D_Value Property) and belongs in a Package (D_PIM_Data) along with other simple variables. The application PIMs (based on the SensorDataModel-PIM profile) use these generic variables in their definition to ease the automatic transformation process.

Example 5 (SensorDeviceModel - D_PIM_Variable): The **dark-green** frame in Figure 4.9 shows the representation of a generic *Temperature* variable as a Class with the D_PIM_Variable stereotype, with one Property of type Integer using the D_Value stereotype. ❑

Device model PSM

In the second place, the SensorDeviceModel-PSM (upper right corner in Figure 4.8) provides a complete description of the device-provided data, specifying all the necessary information to describe the variables sensed by the platform: Firstly, the D_PSM_Probe Package stereotype describes the physical probes that can sense data in the platform; the (optional) Register Tag could indicate where to read the measurements from each probe. Secondly, the D_PSM_Variable Class stereotype will describe each variable sensed by the probe; the Unit Tag must indicate the measurement units. *It has to be generalised (KindOf) as a generic D_PIM_Variable*. Finally, the D_PSM_Sense Operation stereotype should define the method to gather the variable in the application code.

Example 6 (SensorDeviceModel - D_PSM_Variable): The **light-green** frame in Figure 4.9 shows the representation of the complete device data. Furthermore, the **dotted dark-green** frames show two different platform-specific temperature variables available in the UEB (*i.e.* the example sensor). One measured from an LM75 probe using the *lm75ReadTemperature* method, and another from a BMP180 probe using *bmp180ReadTemperature*. Both temperatures return one value of type *_int16* (a platform D_PSM_StaticType, **dotted light-green** frame in Figure 4.9) in a unit of $^{\circ}\mathrm{C} * 2$ and $^{\circ}\mathrm{C} * 10$, respectively. ❑

Device model managers

In the third place, the SensorDeviceModel-M (central and lower part of Figure 4.8) allows modelling the facilities and resources of the implementation device that are relevant for implementing data-centric applications, *i.e.* the communication interfaces, the timing configuration, the memory management, and the program execution management.

The D_PSM_Communications Package stereotype enables the representation of the platform communications capabilities. It can contain one or more D_PSM_Interface Classes, which represent the communications interfaces of the platform. They can use InterfaceConfigurator Properties to describe specific configurations, one or more IntInput Operations to state the methods for receiving data through the interface, and one or more IntOutput Operations to state the methods for delivering data through the interface. Moreover, a D_PSM_Interface can use one D_PSM_Protocol Class, which describes the protocol used in the interface to send and receive data using the ProtocolConfigurator Properties.

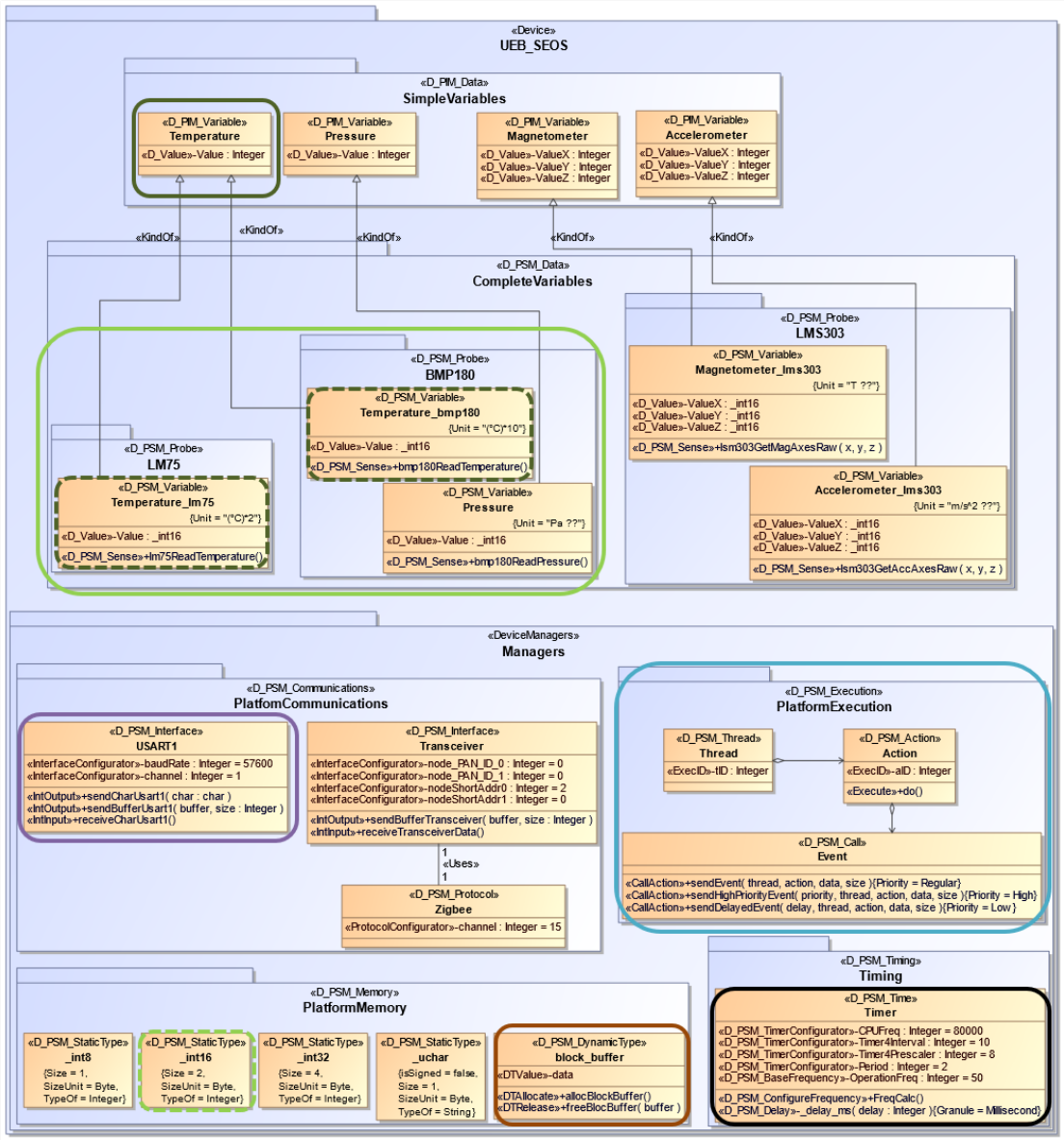

Figure 4.9: SensorDeviceModel model example for the UEB with SEOS

Example 7 (SensorDeviceModel - `D_PSM_Interface`*)*: The **purple** frame in Figure 4.9 shows the `USART1` serial communications interface of the UEB. This interface has only two configurations: `baudRate` of 57600, and `channel` 1. The interface also defines one method for receiving data and two methods for delivering data: `receiveCharUsart1` for the reception, and `sendCharUsart1` and `sendBufferUsart1` for the delivery. ❑

The D_PSM_Timing Package stereotype allows for configuring the base clock and frequency for the platform. Inside this Package, the D_PSM_Time Classes can use different D_PSM_TimerConfigurator and one D_PSM_BaseFrequency Properties with the D_PSM_ConfigureFrequency Operation to define the operation frequency of the platform. Besides, the D_PSM_Time class should also define D_PSM_Delay Operations, which allow stopping the platform from working during some time. These Operations should also define the time granularity (Granule Tag) and the maximum delay allowed (MaxDelay Tag).

Example 8 (SensorDeviceModel - D_PSM_Time): The **black** frame in Figure 4.9 shows an example of the clock configuration for the UEB. It has four D_PSM_TimerConfigurators: the original frequency of the CPU, *CPUFreq*; the interval divisor, *Timer4Interval*; the Prescaler divisor, *Time4Prescaler*; and the period divisor, *Period*. The D_PSM_ConfigureFrequency Operation, *FreqCalc* makes the following operation to calculate the base frequency: $OperationFreq = ((CPUFreq/Timer4Interval)/Time4Prescaler)/Period$. ❑

The D_PSM_Memory Package stereotype enables the definition of the code-specific variables to develop applications for the platform. This Package can also define the maximum memory space of the platform with two Tags: Size, for the available memory space; and SizeUnit, the unit of the memory space (*e.g.* bit or byte).

Inside this Package, we defined two possible kinds of data types. Firstly, the static data type, D_PSM_StaticType. It can have one or more values (STValue) and defines a static Size and a SizeUnit; it can also define if it is signed (isSigned) or its primitive type (TypeOf). Secondly, the dynamic data type, D_PSM_DynamicType. It can define a maximum storage space (MaxSize) with its related unit (SizeUnit) and the maximum number of instances (MaxInstances). Also, this type can have one or more values (DTValue) and must define at least one Operation to Allocate the required memory, and one Operation to Release the allocated memory.

Example 9 (SensorDeviceModel - D_PSM_Memory): The **dotted light-green** and **orange** frames in Figure 4.9 show examples of one D_PSM_StaticType and one D_PSM_DynamicType data types in the UEB with SEOS. The static type is a short integer of 2 bytes; this type is used in example 6 (light-green frame). The dynamic type will be used to generate the applications code, considering specific functional requirements. ❑

Finally, the D_PSM_Execution Package stereotype enables the description of the basic platform operation. It has D_PSM_Execution modules that can be D_PSM_Threads (macro-module) or D_PSM_Actions (micro-module). A group of connected D_PSM_Actions with one goal can compose a D_PSM_Thread. Moreover, the D_PSM_Actions should Execute some Operations, and they must Call other D_PSM_Actions.

Example 10 (SensorDeviceModel - D_PSM_Execution): The **aquamarine-blue** frame in Figure 4.9 shows the platform execution in the SEOS, on which each Thread and Action have an ExecID, and the Actions can Call other Actions with three priority levels: regular (*sendEvent*), high (*sendHighPriorityEvent*), and low (*sendDelayedEvent*).

❑

4.2.3 Integration of sensor device and application models

The two models (SensorDataModel and SensorDeviceModel) are correlated and dependent as evidenced in examples 1 and 3. Firstly, for the PIM models, `A_PIM_Variable` Properties must select a `D_PIM_Variable` Class as data Type. Secondly, for the PSM models, `A_PSM_Variable` Properties must select a `D_PSM_Variable` Class as data Type. Besides, the `A_PSM_DeliverOnTuples` Operations must define a `D_PSM_Interface` Class as value for the `Interface` Tag. The constraints that rule these relationships are stated in Figure 4.10.

context A_PIM_Variable **inv** varType_PIM: (self.type.oclIsTypeOf(D_PIM_Variable))
context A_PSM_Variable **inv** varType_PSM: (self.type.oclIsTypeOf(D_PSM_Variable))
context A_PSM_DeliverOnTuples **inv** interfaceType: (self.Interface.oclIsTypeOf(D_PSM_Interface))

Figure 4.10: OCL that rule the relationships between SensorDataModel and SensorDeviceModel.

4.3 Sensor Data Implementation

This section describes how to use our automatic-implementation tool with the sensor platform described in the case study (*i.e.* the UEB with SEOS). An application for SEOS must be written in C and is composed of two parts: the programming logic in an "*application.c*" file; and the constants and data types in an "*application.h*" file.

The top of Figure 4.11 illustrates the complete implementation process. Firstly, the sensor-data application is modelled using the CASE tool (MagicDraw®) and the SensorDataModel profiles. Secondly, we have developed a parser in Python that automatically generate the SEOS *application.c* and *application.h* files from the XMI document that represents the SensorDataModel-PSM in MagicDraw®. Finally, the SEOS tools compile the application files into hexadecimal code and load it to the UEB.

Besides, the bottom of Figure 4.11 shows fragments of both the *application.c* and *application.h* files generated by our Model-to-Code Tool for the Case Study (GDU) PSM presented in Figure 4.6. First, the fragment of *application.c* shows a modular operation (event) for the SEOS in which the device gathers and stores some temperature measurements, waiting 20 minutes between two consecutive gatherings. Second, the fragment of *application.h* shows that the number of gatherings (*i.e.* temperature samples) is 72 for a 24-hours period; it also presents the structures defined for the gathered and delivered data. Our tool takes less than five seconds to generate the application files in this example while running in a standard laptop.

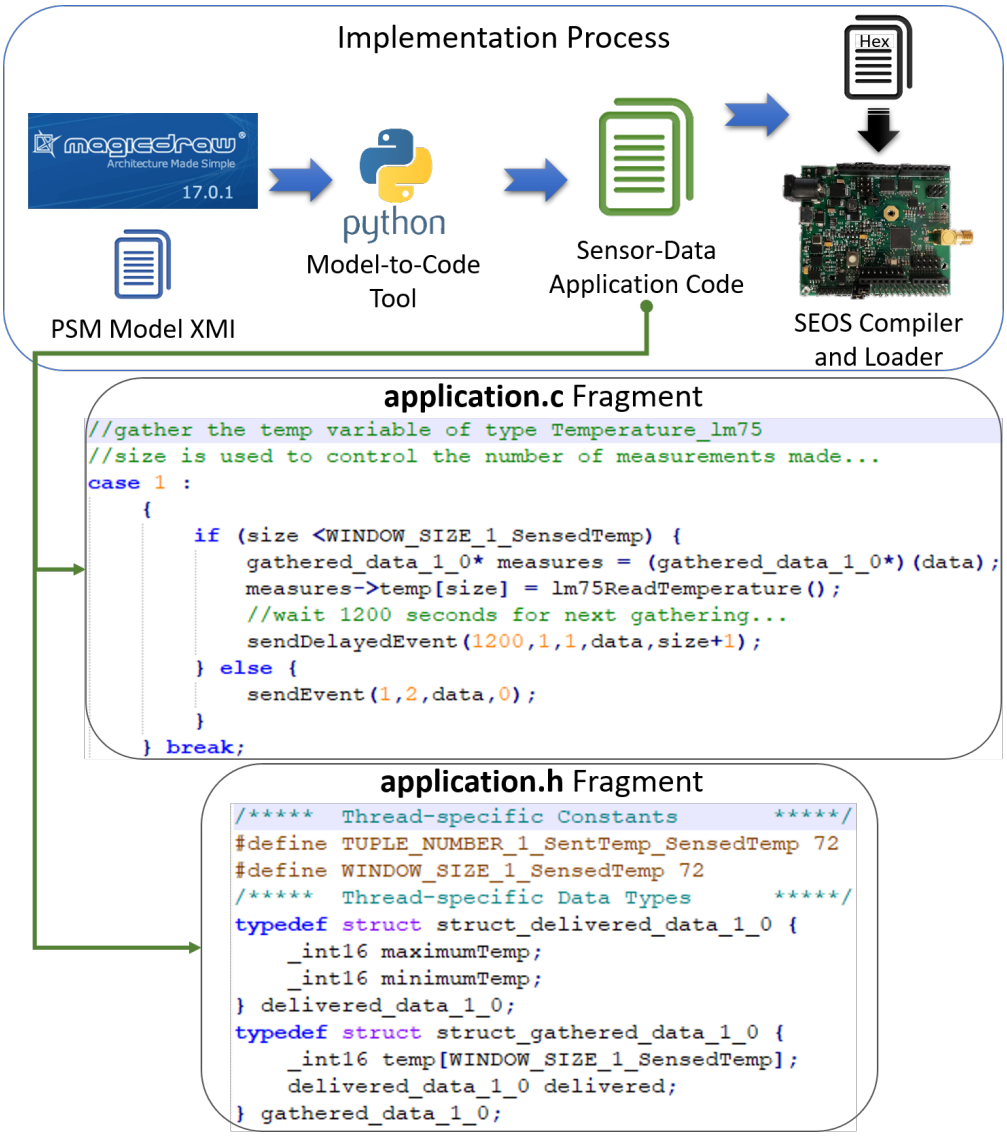

Figure 4.11: Implementation Process including fragments of the application.c and application.h files generated by our tool based on the Case Study (GDU) PSM.

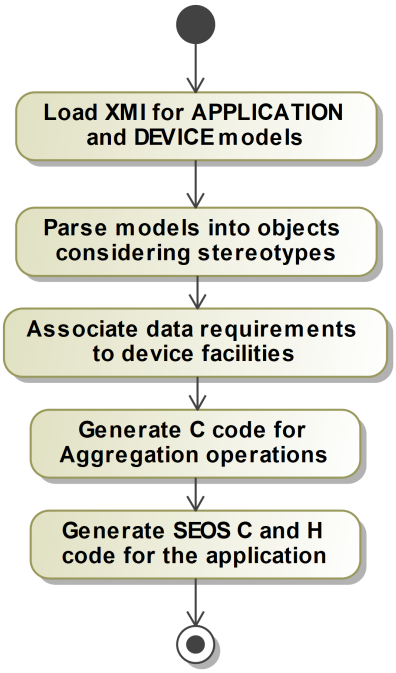

Figure 4.12: Activity diagram of the Sensor Model-to-Code tool.

Figure 4.12 represents the sensor-data model-to-code tool, implemented as a python script. The first step is to read the XMI files of the application's PSM and DM models. Second, the tool extracts the models' details leveraging the stereotypes and generates a set of python objects for the sensor-data application. Third, the parser prepares a standard sensor-application template for SEOS, fixing the details from the generated objects for the specific application. Then, it forges and includes the required aggregation operations into the precise code location. Finally, it outputs the generated files (C and H, *e.g.* Figure 4.11) that allow the sensor device to provide the required data.

4.4 Integrating Sensor Data into BI Systems

This section presents our methodology for Self-Service BI with On-Demand Data (SSBI-ODD), focusing on DW and IoT systems (Figure 4.13), which is an extension of the ProtOLAP methodology [118]. ProtOLAP is a methodology that allows for rapid prototyping of DW. It is mainly based on: (i) the automatic implementation of DW schema models from UML models; (ii) validation of decision-makers requirements via visualisation of simulated warehoused data by means of OLAP clients (right part of Figure 4.13). Full details of ProtOLAP are thoroughly explained in [118]. In particular, we have added a new set of steps in the methodology that

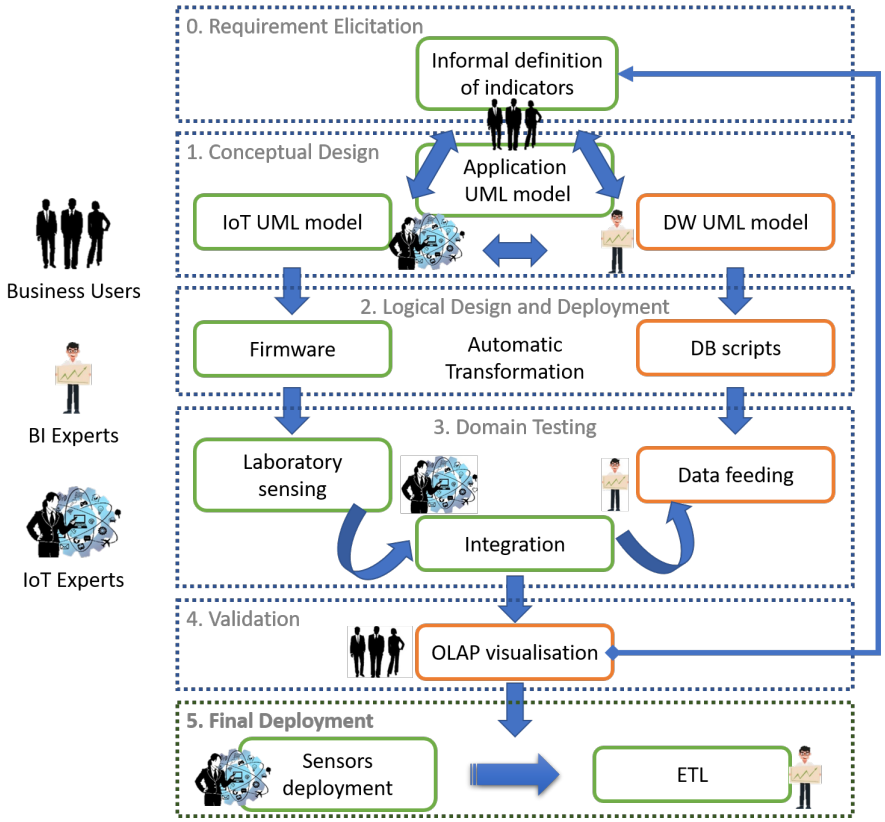

Figure 4.13: Methodology for self-service DW over on-demand IoT data.

allows for the definition of the IoT-data acquisition system (*i.e.* sensors in this case). In this way, it complements the goal of [118] with the direct integration of a different and complex data source. Figure 4.13 represents the steps from ProtOLAP that remain the same (for DW) with an orange frame and the new steps (for IoT) with a green frame.

4.4.1 Data integration methodology

Our methodology (Figure 4.13) is composed by six phases:

In the **Requirement Elicitation** step, the Business Users state their analysis needs in natural language (*i.e.* an informal definition). Throughout the discussion with the BI and sensors (IoT) experts, these analysis needs are defined in terms of *indicators* (*e.g.* measures and aggregations). This step is quite similar to one proposed in [118], but here the business users must also express their needs in terms of data collected by the IoT (*i.e.* what, when and where).

Then, in the **Conceptual Design** step, the experts in both IoT and BI areas define the best way to formalise the *indicators* as an *Application UML model*. The IoT and BI experts can then use this model to explain the technological solution to the business users, discuss with them,

and refine the application design. Once the *Application UML model* is complete and appropriate enough for the three roles (*i.e.* IoT and BI experts, and business users), the IoT (sensors) experts start working on the data-centric design of the IoT part (*IoT UML model*), while the BI experts work on the data-centric design of the BI part (*DW UML model*) as defined in [118]. Both models have to be compliant and integrated in only one common model.

In the **Logical Design and Deployment** step, the models for the IoT part (*IoT UML model*) and the BI part (*DW UML model*) are automatically transformed into *Firmware* and *Database (DB) scripts* respectively. The *Firmware* allows programming sensors or IoT devices, while the *DB scripts* define the relational schema and metadata of the DW as described in [118]. In this phase, automatic transformation from the conceptual models is a key feature that significantly reduce the implementation effort allowing for rapid prototyping. Therefore, we suggest the use of model-driven approaches such as [12], though different approaches might also be valid.

For the next step, **Domain Testing**, IoT and BI experts work together again. Firstly, the sensors (or IoT) experts prepare some sensors for gathering data in a test environment (*Laboratory sensing*). Secondly, the BI experts prepare an initial deployment of the DW capable of receiving data (*Data feeding*) as defined in [118]. Finally, they integrate the two experimental subsystems (*Integration*). This integration sub-step is done manually since it depends on the particular application needs as discussed in Sec. 4.4.2. In this phase, IoT and BI experts must verify the appropriate data collection, delivery, reception and analysis. This step produces an operative prototype of the sensor- or IoT-based BI system.

In the **Validation** step, business users check the prototype with a common *OLAP visualisation* tool. By exploring the data and analysis provided by such prototype, they can verify if the system (and thus the underlying conceptual model) correctly provides the defined *indicators*. If the prototype is not appropriate, the business users return to the discussion with the IoT and BI experts, further refining the *Application UML model*.

Otherwise, if the prototype is deemed appropriate, the process advances to the sixth and final phase: **Final deployment**. This phase consists in deploying the sensor or IoT devices, and finalising the ETL procedures to receive and load their data into the DW.

The following subsection provide deeper details and examples of the conceptual design of sensor-based BI applications, which enables SSBI-ODD. Furthermore, Sec. 4.3 provide insights in the automatic generation of *sensor Firmware*.

4.4.2 Conceptual integration of sensors data in BI

After both the sensor and DW are implemented in their first separated prototypes (Domain Testing phase), their integration should be as simple as defining a gateway that connects the output interface of the sensor and the input interface of the BI. The data acquisition (IoT) and reception (DW) processes must be compatible *from the design of the system*. Thereby, in this section, we describe three different approaches for providing a unified and complete data model of sensor-based BI applications. These approaches corresponds to different analysis needs expressed by business users and conceptual-integration levels.

The **Naive approach** considers that the sensor and the DW are represented in two different

data models. Thus, the DW `Fact` has equivalent attributes with the sensor `A_PIM_Measure` (*i.e.* there is no need for ETL or any transformation on the sensed data) and the two models can be related through a one-to-many association. *This approach allows OLAP analysis over IoT data that are permanently stored in the DW without requiring any transformation.*

The **ETL approach** is similar than the Naive one, it still considers that the IoT and the DW are represented in two different data models and the two models can be related through a one-to-many association. However, the attributes in the DW `Fact` are different than that of the IoT `A_PIM_Measure` and thus they require a transformation process (*i.e.* ETL) before being loaded into the DW. *This approach allows OLAP analysis over sensor data that are permanently stored in the DW with some previous transformations that cannot be achieved directly inside the sensor.* Indeed, IoT devices may have not enough memory, computational or battery resources to provide some transformations. Therefore, they must be executed outside of the IoT.

Example 11 (ETL integration): Using the ICSOLAP profile [12] to represent the DW part, Figure 4.14 shows the multidimensional representation of the irrigation data used in our case study for the GDU (white packages). The measure is indeed the GDU value needed to apply the irrigation formula described in Sec. 2.2. This value can then be aggregated in different ways. For example, using the maximum GDU per crop and plot to find which fields need more water, or the average per crop type and week to keep a record of the plants' growth [76]. The *Time* allows to analyse data from each *Day* or *Week*. The *Crops* can be analysed by each individual *Crop* or grouped by *CropType*. Besides, data can also be analysed at the level of *Sensor*s, *Plot*s, or for the whole *Farm*.

Therefore, the DW model of Figure 4.14 (white packages) allows answering to the queries defined for our case study, such as "*Which crop in which plot grew the most today?*", in different spatial, temporal and thematic granularities. However, the the DW model alone (*ClimaticConditions*) cannot show any details about the sensed data that is feeding the `Fact`. Thus, we can associate our SensorDataModel-PIM model of Figure 4.3 to the fact as shown at the bottom of Figure 4.14 (blue package) to provide a complete description of the sensor-based BI application. Moreover, the annotation of the fact contains the computation rule for the GDU. This annotation represents the ETL operation as defined by [119, 120] (although more comprehensive ETL models supporting MDA could also be used [28]). In this way, the annotation of the `Fact` and the association between the two profiles indicate that the sensed data is firstly transformed and then associated with one fact of the DW. This simple representation of an integrated IoT and BI system is possible since we have managed to keep a very simple representation of the sensor, focusing only on data. ❑

Furthermore, our `A_PIM_Measure` class could also be directly used as a `Fact` in the ICSOLAP profile [12] in a **Sensor Stream DW approach**. However, the DW model must support stream data facts before considering this kind of integration with sensor data.

Indeed, more recent works propose to provide OLAP queries directly over data streams (*e.g.* [13]). In this way, we can use our sensor profile to represent the stream data-source of a **Stream DW**, allowing for a continuous OLAP analysis over streamed IoT data without requiring

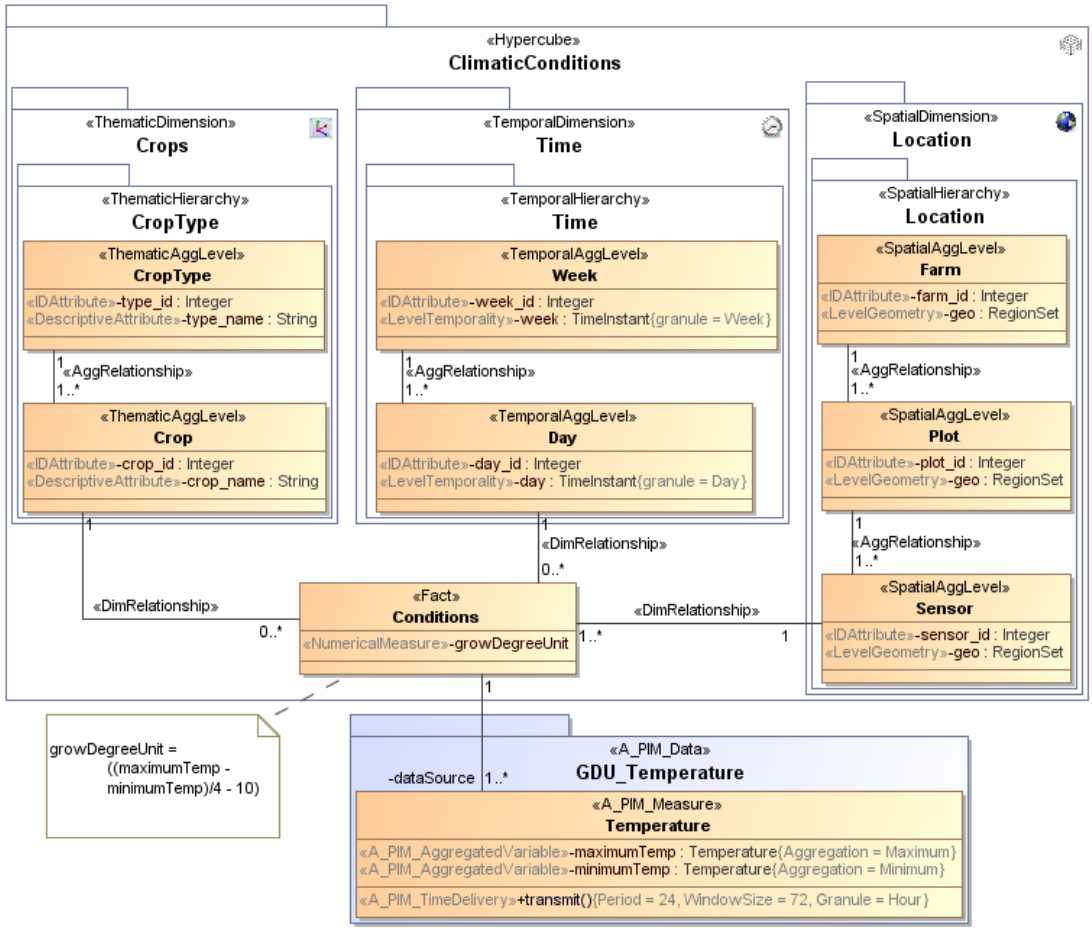

Figure 4.14: Example of *ETL* integration of sensor (blue) and DW (white) data models at PIM level.

additional transformations or permanent storage. This approach is different from existing ones since it models the whole sensor- and IoT-based BI application in a single model using a single profile.

More precisely, we integrate our SensorDataModel-PIM profile (Sec. 4.2.1) into the UML profile for Stream Data Warehouses (SDW) of [13]. This work extends the [12] profile for classical data with a `StreamFact` and a `WindowDimension`. The `StreamFact` allows representing temporally-constrained data (*i.e.* stream), while the `WindowDimension` defines for how long the SDW will keep the data to run the required analysis.

Figure 4.15 shows how we have further extended the [13] profile to use our sensors `A_PIM_Measure` as a `StreamFact` for SDW. Our extensions are highlighted with blue dotted frames.

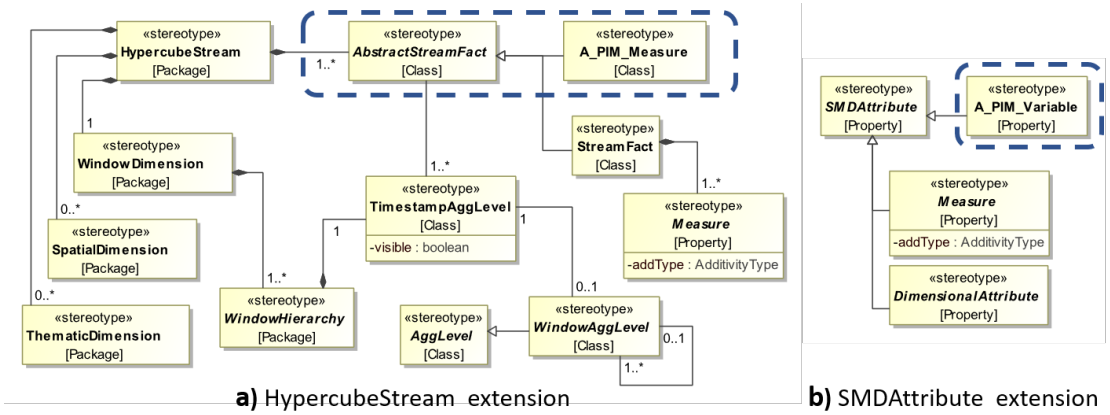

Figure 4.15: Extension of the Stream Data Warehouse profile.

Firstly, Figure 4.15-A shows the resumed SDW profile for the `HypercubeStream` Package, which replaced the classical `Hypercube` Package. To extend this part of the profile, we define the `AbstractStreamFact` Abstract Class Stereotype as a generalisation of `StreamFact`, associating it to the `HypercubeStream` and the `TimestampAggLevel`. Then, we add our `A_PIM_Measure` as one of the specifications of `AbstractStreamFact`, which allows us to analyse sensor data in the different dimensions.

Secondly, Figure 4.15-B shows the relationship between the `Measures` of the `StreamFact` (*e.g.* `NumericalMeasure`) and the `A_PIM_Variable` of the `A_PIM_Measure`. In this part of the profile, we add our `A_PIM_Variable` (and thus the `A_PIM_AggregatedVariable`) as one of the specifications of `SDMAttribute`, the generalisation of `Measure`. Note that these representations (Figure 4.15) are resumed, *i.e.* do not exhibit all the depth of the profiles. Therefore, their understanding must be complemented with the original SDW profile [13] and the sensor-data profile (Sec. 4.2).

Nevertheless, we provide an example in the agriculture domain using our extended profile for sensor-based BI applications (Figure 4.16).

Example 12 (Sensor Stream DW integration): Figure 4.16 provides a fully integrated model of a sensor-based BI application for the analysis of temperature (GDU in the case study). This model presents a `HypercubeStream` named *`FarmConditionsDataStream`* to analyse the temperature conditions on a farm over different *`Time Window`*s regarding the *`Crops`* and the *`Location`* (*i.e.* dimensions). The observation *`TimeWindow`*s can last one one day or one week; the data is then discarded or sent to a different (storage) system. The *`Crops`* and *`Location`* dimensions work in the same way than in Example 11 (Figure 4.14).

Furthermore, the *`TemperatureConditions`* are acquired directly by a sensor, which every 24 hours loads into the SDW the maximum and minimum air temperature values (*`maximumTemp`* and *`minimumTemp`*) from the last day. In this way, this model could answer queries such as: *"Which Plot presented the lowest minimum temperature today?"* Or "*List the*

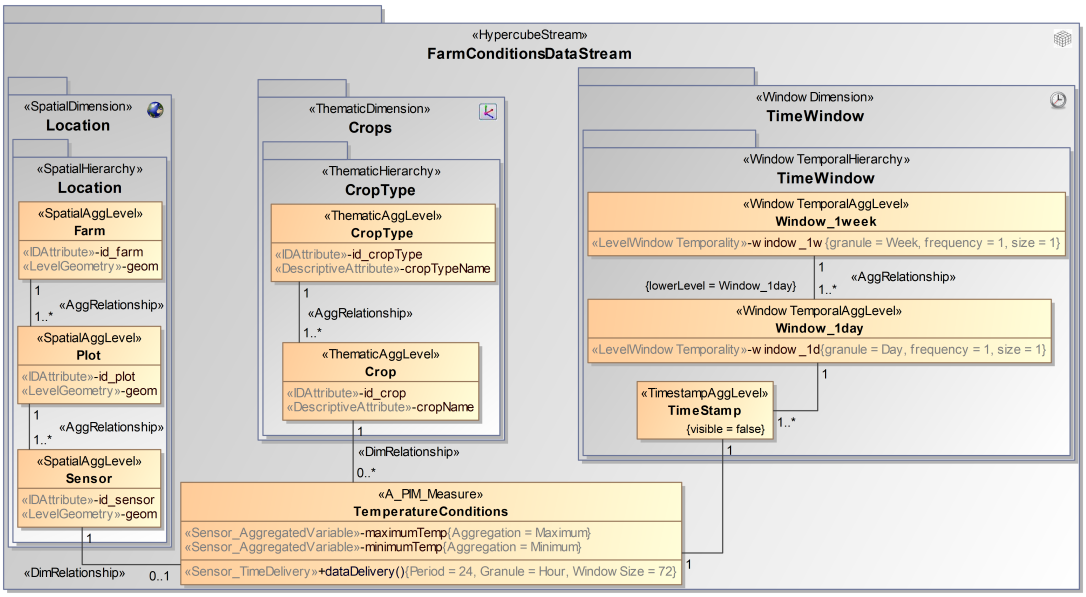

Figure 4.16: Example data model of a Stream Data Warehouse using Sensor data.

five avocado plots with the highest temperature difference this week". ❑

These examples of BI and sensor PIM models integration confirms how our data-centric representation of sensed data allows BI experts and decision-makers to focus only on data (*i.e.* what data represents) and analysis (*i.e.* how data is used) issues when designing a sensor- or IoT-based BI application. Indeed, no details about the technical features of the IoT nodes nor the DW are necessary for such integration.

Appendix A provides a different approach for the integration of sensor, IoT and classical data (amongst other data kinds). Particularly, it leverages the theoretical constructs of the SensorDataModel (Sections 2.3.1 and 4.2) to define an UML profile for polyglot-data applications [42].

Summary

The UML profile for sensor-data applications leverages the UML standard to abstract and represent the prime concepts for these applications: the type of sensed variables, their temporal validity, their computations (limited to aggregation) and the temporal windows. This abstract representation enables the definition of instance models for different applications that use the same design concepts.

Besides, it separates the design concerns and abstraction levels following the MDA guidelines. Therefore, it has a PIM with a highly abstract representation of the sensor data, a PSM

for implementing the sensor application, and a DM representing the sensor devices. Indeed, a model-to-code tool transforms the PSM models into implementable code for actual sensor hardware. This chapter only considers the code generation for UEB nodes running SEOS.

Similarly, the high abstraction of the PIM allowed seamless integration of the sensor-data profile with another data-centric profile for SDW. This integration, and the SSBI-ODD methodology, are the basis for the design and implementation of sensor-based BI applications.

The outcomes of this chapter are published in [39, 41]. [39] presents the MDA approach for sensor-data applications, including the design methodology, UML profile and implementation mechanism. [41] defines the integration methodology and conceptual data model for sensors and BI systems.

Chapter 5

Design and Implementation of IoT-Data Applications

This chapter addresses the modelling of IoT-data applications focusing on WSN. In the first place, it describes the design and implementation of IoT data. Secondly, it describes the integration of IoT data into BI systems following the methodology of Sec. 4.4.1.

For IoT data, this chapter defines a design and implementation methodology for IoT-data applications (Sec. 5.1). Then, it presents a UML profile that supports the design of IoT data (Sec. 5.2). Finally, it presents the implementation of IoT-data applications using the defined profile and UEB devices (Sec. 5.3).

In particular, the UML profile for IoT data extends the sensor-data profile (Sec. 4.2) with more complete data transformations and communications for designing networks of IoT devices. Following existing design methods for classical ETL systems, this profile structures the Sensing, Transformation and Sending of IoT data in multiple objects executing specific tasks for a common goal. Thus, we call the whole MDA approach (UML profile and code-generation tool) for IoT-data applications *STS4IoT* (Sense, Transform and Send for the Internet of Things).

For the IoT and BI data association, this chapter proposes a conceptual integration between the high-abstraction level of STS4IoT with the SDW profile used in Sec. 4.4 for the design of IoT-based BI applications (Sec. 5.4).

5.1 IoT Data Design Methodology

STS4IoT extends and updates the UML profile of Chapter 4 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.

Figure 5.1 shows how different actors can use the STS4IoT components to define IoT-data applications. In particular,

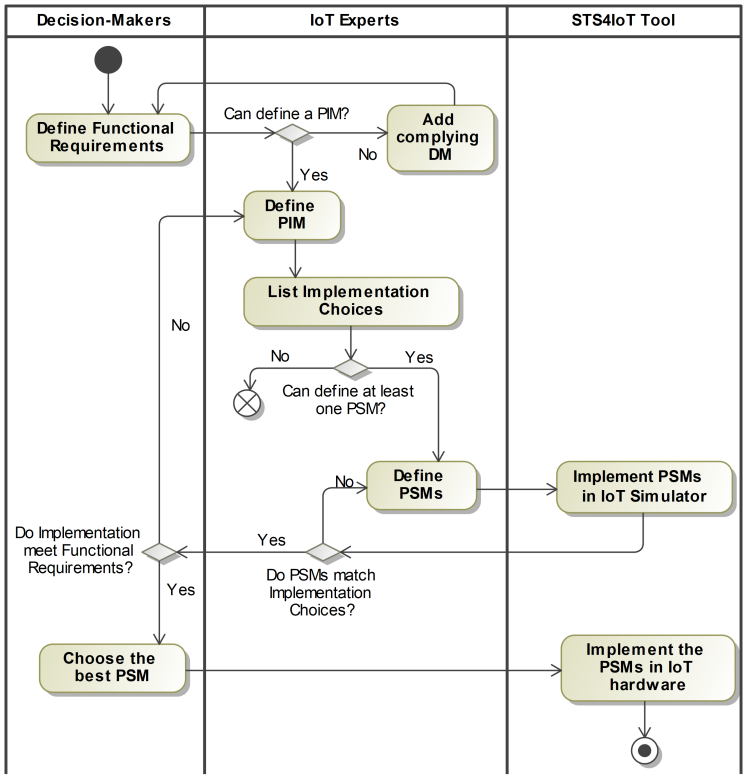

Figure 5.1: STS4IoT methodology

- 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, 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 Figures 2.2-C and 2.2-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.

5.2 UML Profile for IoT Data

This section presents the STS4IoT UML profile, dividing each application meta-model (PIM in 5.2.1 and PSM in 5.2.2) into the data structure and the STS operations. It also presents the meta-model for cataloguing IoT software and hardware facilities that acquire and process the required data (DM in 5.2.3).

5.2.1 STS4IoT PIM

The PIM allows formalising the business users' data needs in UML without providing any physical or implementation details. Figure 5.2 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 FR of IoT-data 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, Figure 5.3 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 case study (Sec. 2.2). To better understand the PIM, we describe it in two components: data structure (5.2.1) and STS operations (5.2.1).

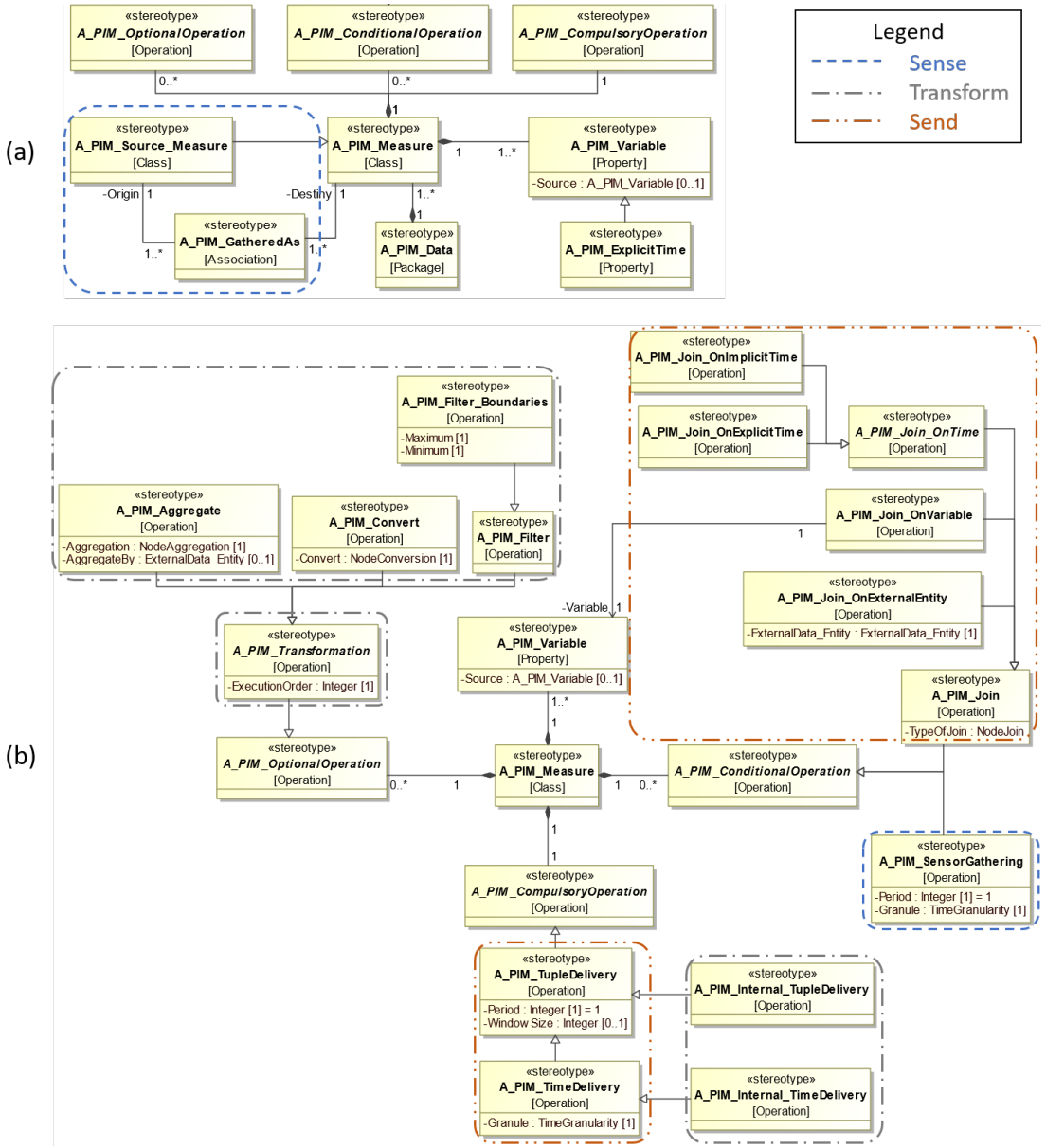

Figure 5.2: STS4IoT PIM profile data part (a) and STS part (b). Dashed blue lines are for the *Sense* stereotypes, dash-dotted grey lines for the *Transform* stereotypes, and dashed double-dotted orange lines for the *Send* stereotypes.

PIM data structure for STS

The PIM data structure (Figure 5.2-A) allows defining the variables required from the IoT-data 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 (Figure 5.3), 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 (Sec. 4.2.1), 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 Tags).

`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 SensorDataModel (Sec. 4.2.1), 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.

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 [80]. Therefore, we define a new operation stereotype that is `A_PIM_Join` (Figure 5.2-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_PIM_Join_OnExternal-Entity` 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 `Window-Size` 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

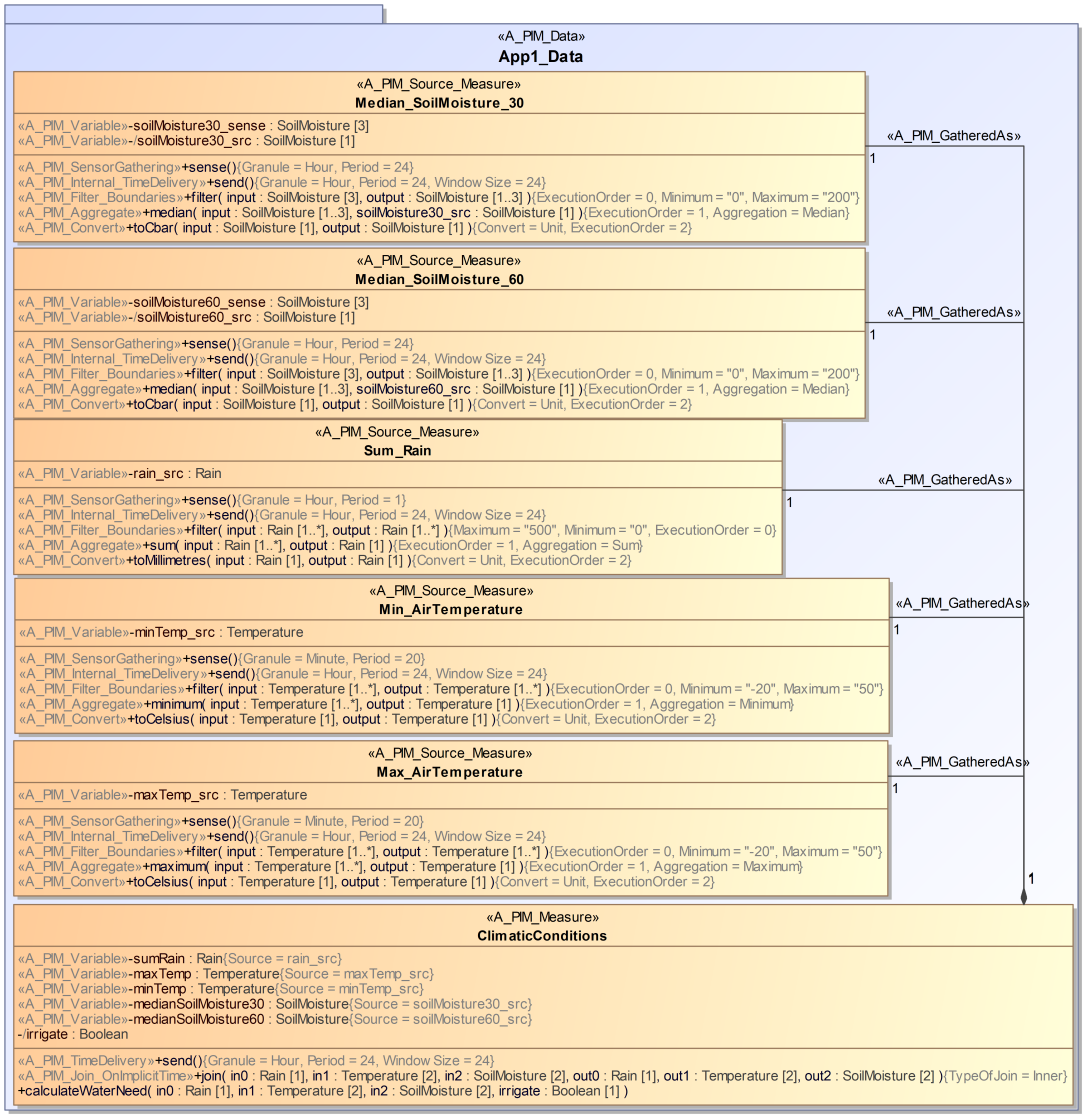

Figure 5.3: Possible PIM for the case study (Sec. 2.2) making all the transformations inside the IoT.

implicit time (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 1 (STS4IoT-PIM data structure): In the following, we present the PIM for the IoT data in our case study (Figure 5.3). Indeed, *ClimaticConditions* represent all data used by irrigation rules described in Sec. 2.2. 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 [76]. *ClimaticConditions* defines the *join* operation to join all the data received from the five different source measures during the last 24 hours (*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_PIM_GatheredAs association, and the *sumRain* property declares *rain_src* as its Source Tag. ❑

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 (Figure 5.2-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 (Figure 5.2-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 (Sec. 2.2), 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 (A_PIM_Convert).

In detail, from the SensorDataModel profile (Sec. 4.2.1), 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 on IoT data to express tailored operations according to the business users' needs.

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 2 (STS4IoT-PIM STS operations): The PIM of our case study (Figure 5.3) presents various 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 minutes. These classes also define data *transformations*. For instance, *Median_SoilMoisture_30* defines two ones. Firstly, it aggregates the data with the *median* operation. Secondly, it converts the me-

dian 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 hours, operating the collected data of the last 24 hours. Similarly, *Max_AirTemperature* defines an internal *send* operation to transform the sampled data of the previous 24 hours every 24 hours. ❑

To finalise our PIM profile, we added a set of OCL rules 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, Figure 5.4 shows two rules that define which delivery Operation belongs in each class (main or source). Similarly, Figure 5.5 shows the OCL that restrict the sensing in A_PIM_Measure and control its use in A_PIM_Source_Measure.

```
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))
```

Figure 5.4: 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)
```

Figure 5.5: OCL rules for A_PIM_SensorGathering.

Finally, designers can also define OCL constraints over the instance models to set some data quality characteristics [51]. 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 Figure 5.6). Also, a second rule for consistency can state that the maximum air temperature must be above 0 degrees Celsius (second rule in Figure 5.6).

These examples show that OCL could successfully express data-quality rules in our STS4IoT UML profile. Nevertheless, we do not address the implementation of these rules in the model-to-code transformation. Besides, particular data structures could offer an alternative for such quality rules. For example, our case study application could define two

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

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

Figure 5.6: Example OCL rules for data quality.

`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. In the same way, `A_PIM_Filter` transformations could drop all misleading values (*e.g.* under-zero temperatures).

5.2.2 STS4IoT PSM

The low-abstraction level of STS4IoT (the PSM) describes the essential implementation issues of the actual application, providing an implementable yet simplified representation of the IoT-data application.

Specifically, the PSM leverages the DM catalogue 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 [16]. 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 (*cf.* 2.2):

- *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 real-time 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 (Sec. 5.2.3). 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.

For example, when IoT experts of our case study look into their DM catalogue, they may notice there are seven kinds of devices available (*cf.* Figure 2.2):

- 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 case-study farm fields have internet access in one specific location only. With this information, they could define four implementation options:

1. **Figure 2.2-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. **Figure 2.2-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 [77].
3. **Figure 2.2-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. **Figure 2.2-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 [77].

Business users and IoT Experts may select option 4 (Figure 2.2-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 (Figure 5.7) to define the selected implementation option in the UML formalism (Figure 5.8). To better understand the PSM profile, we have decomposed it into two components: data structure and STS operations.

PSM data structure

The PSM data structure (Figure 5.7-A) keeps almost all the original stereotypes from the SensorDataModel (Sec. 4.2.1), 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 IoT node senses, transforms and sends data to other nodes. Indeed, as previously described, an end node could only sense single or multiple variables and send them to other nodes that transform them or merge them with data from different nodes. These particular transformations and communications amongst the nodes are intentionally kept transparent at the PIM level since business users must not handle their implementation details, addressed by IoT experts only. Therefore, to address 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 the PSM for our case study (Figure 5.8), the soil moisture values sensed at 30cm 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 SN.

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

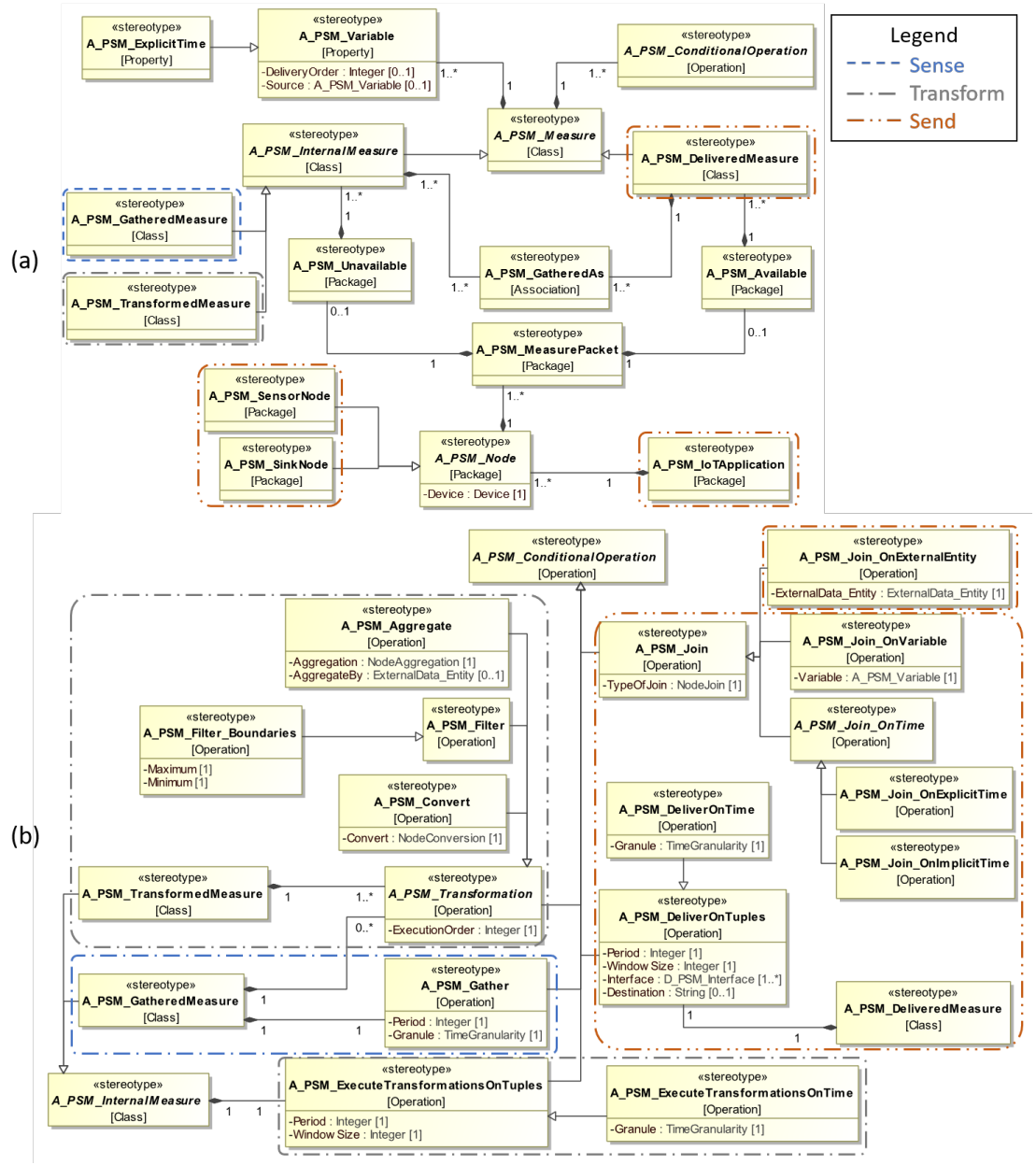

Figure 5.7: STS4IoT PSM profile data part (a) and STS part (b). Dashed blue lines are for the *Sense* stereotypes, dash-dotted grey lines for the *Transform* stereotypes, and dashed double-dotted orange lines for the *Send* stereotypes.

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. Therefore, a node can receive and transform data from other devices while also sensing and transforming 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. 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 (Sec. 5.3).

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 and each measure one Class.

Example 3 (STS4IoT-PSM data structure): Figure 5.8 presents the PSM of our running IoT example considering the PIM and implementation option D (Figures 5.3 and 2.2-D, respectively). *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*.❑

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.

PIM STS operations already 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*?", or "*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.

Example 4 (STS4IoT-PSM and STS4IoT-DM association): In our case study (Figure 5.8), LM75 probes sense temperature, the specific DM supports the transformations (as in Figure 5.11), and the SN sends the values through *USART1*. ❑

We have defined 17 stereotypes to represent these operations on IoT data (Figure 5.7-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 SensorDataModel (Sec. 4.2.1) 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.

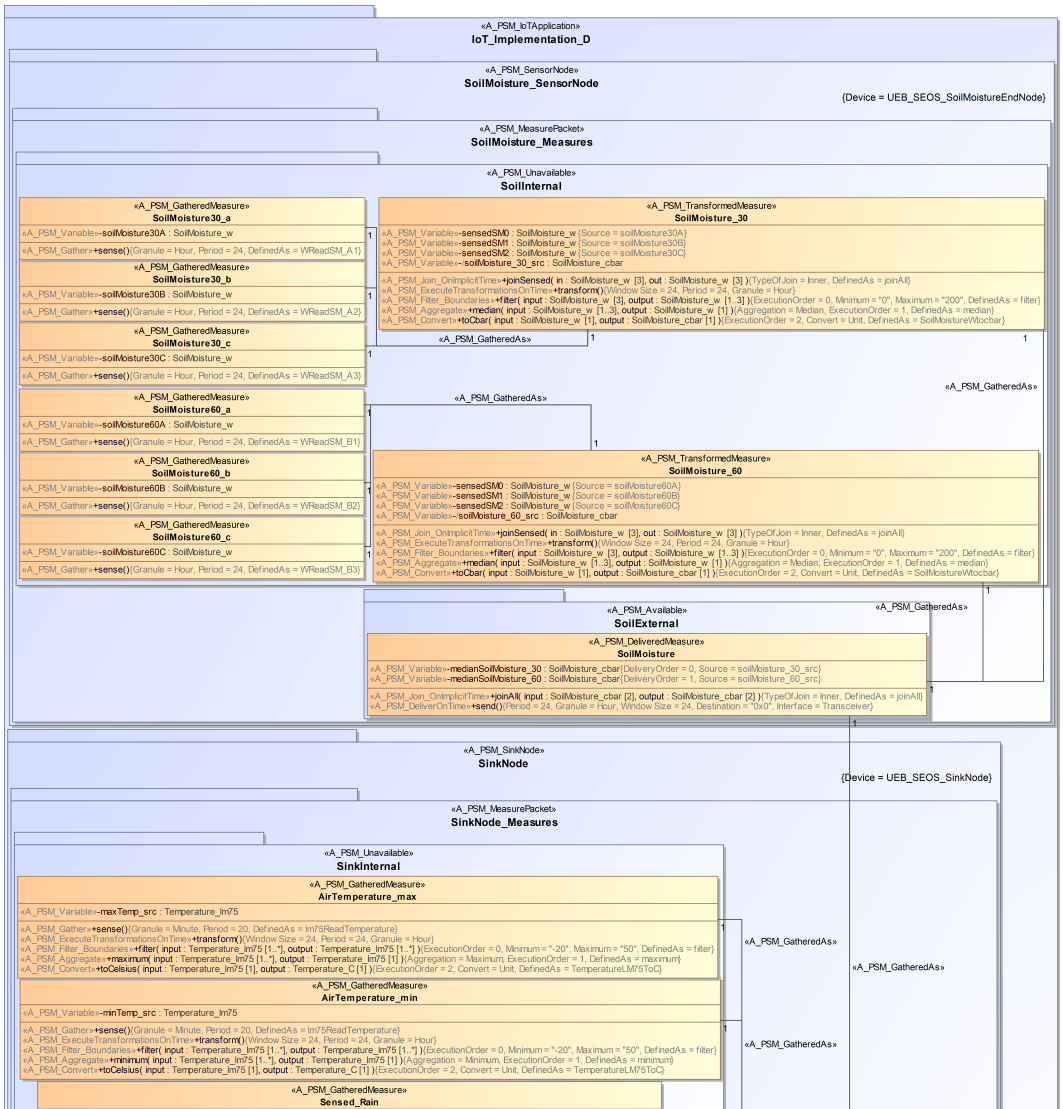

Figure 5.8: Possible PSM for the PIM of the running example (Figure 5.3) considering the configuration of Figure 2.2-D.

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.* Figure 5.3) 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 to calculate 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.

Furthermore, 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_PSM_Transformation in the same class.

Example 5 (STS4IoT-PSM STS operations): In the PSM of our running IoT example considering implementation option D (Figure 5.8), the A_PSM_GatheredMeasures of *Soil-Moisture_SensorNode* define *sense* operations to read the soil moisture every 24 hours.

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. 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 hours every 24 hours.

Finally, *ClimaticConditions* defines a *send* operation that sets a 24-hours window and a periodicity of 24 hours to execute a join of all the data in the IoT and deliver them through the *USART1* interface (*cf.* 5.2.3). ❑

The elements of the STS4IoT PSM profile allow modelling implementable IoT-data applications such as WSNs considering the STS characteristics. Furthermore, IoT experts can consider and use the PSM profile to design different implementation options for the same application (Sec. 2.2).

Example 6 (STS4IoT-PSM multiple implementations): Figure 5.9 displays the fragment of a PSM model to implement the deployment option in Figure 2.2-B, which also meets the application requirements defined in the PIM (Figure 5.3) but provides higher spatial reliability.

As shown in Figure 2.2-B, the PSM of Figure 5.9 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 hours; and *Temperature_SensorNode* (using the DM *UEB_SEOS_TempEndNode* of Figure 5.11), which provides the sink node with the maximum and minimum air temperatures in degrees Celsius.

The hidden devices (do not appear in Figure 5.9 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 millimetres to the sink node every 24 hours through the *Transceiver*. The sending period, granule, and interface are the same for the three ENs.❑

5.2.3 STS4IoT DM

The DM is a simplified abstract representation of IoT facilities that eases the definition of the models while also enabling the automatic generation of implementable code. Multiple DMs provide IoT experts with a conceptual catalogue to easily find and use the best-fitting devices for their IoT-data applications.

For the sake of simplicity and keeping a data-centric approach, the STS4IoT-DM (Figure 5.10) reduced the number of modelled facilities of IoT devices from the SensorDeviceModel

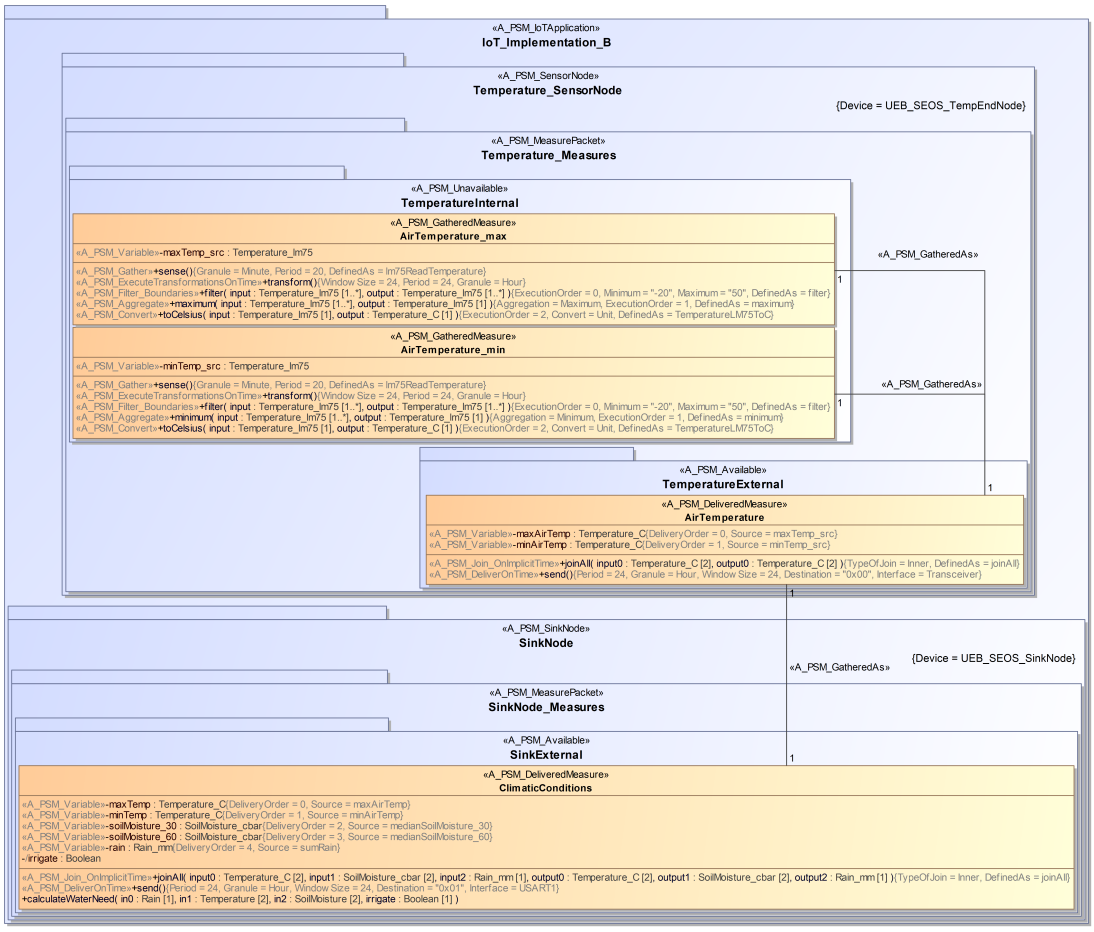

Figure 5.9: Fragment of another possible PSM for the running example considering the configuration of Figure 2.2-B (shows only the Temperature EN and SN[2]).

meta-model (Sec. 4.2.2). Yet, it considers operations as an essential facility for any `Device` besides its data and managers:

- `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_Variable-OfKind` 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

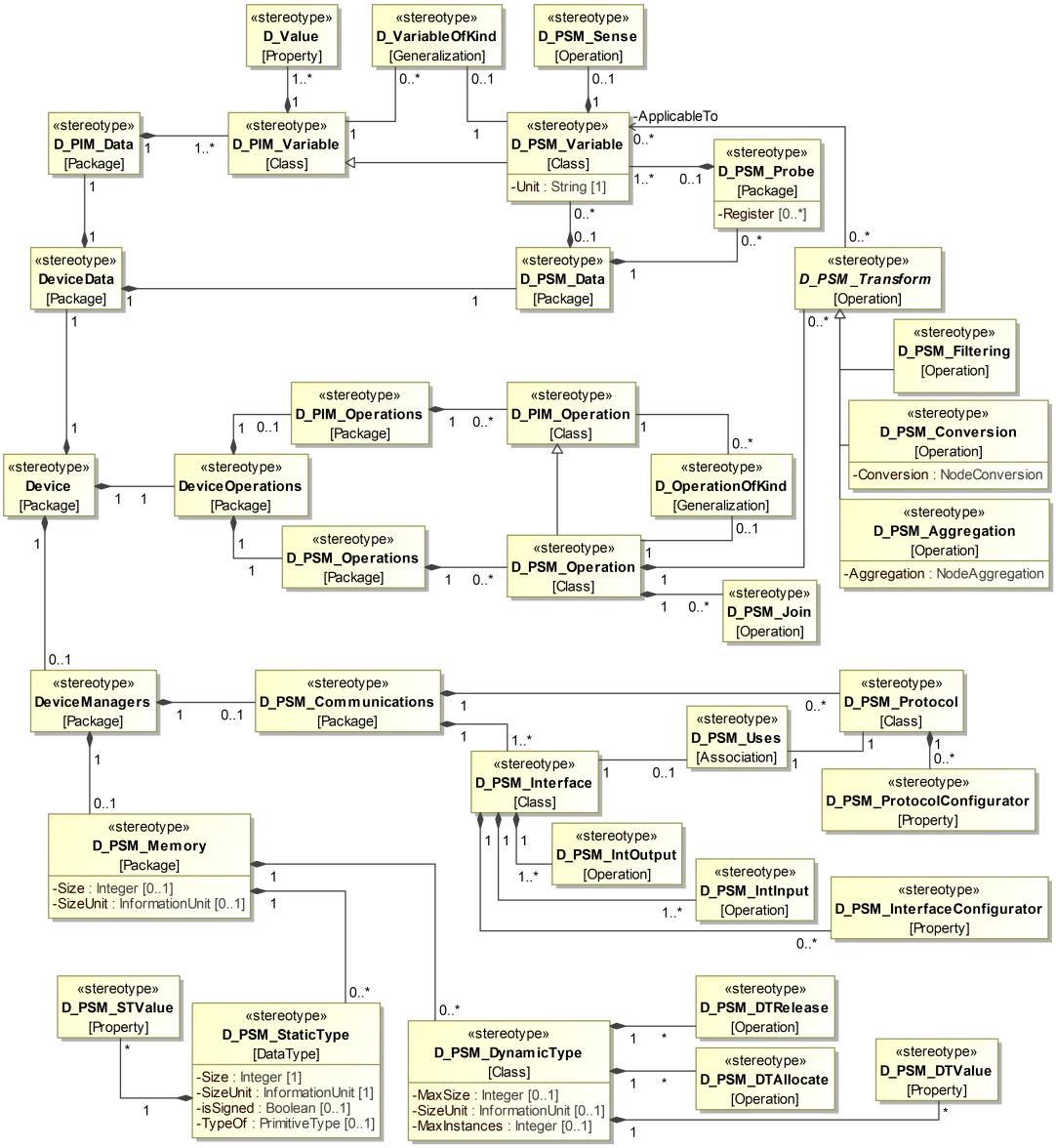

Figure 5.10: STS4IoT DM Profile.

D_PSM_Probe Package. A device can contain data it cannot sense in two cases: First, when it receives data from another device. Second, when an existing transformation (conversion) changes the platform data type or unit, and thus a new implementable variable should represent such change without changing the source sensing operation.

- 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. For the PIM, these associations are not necessary and may induce design errors.

  Implementable 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 modelling the PSM and generating the code: communications (D_PSM_Communications) and memory (D_PSM_Memory). The communications module contains the node 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 (*e.g.* SEOS for UEB) 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 7 (STS4IoT-DM): Figure 5.11 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

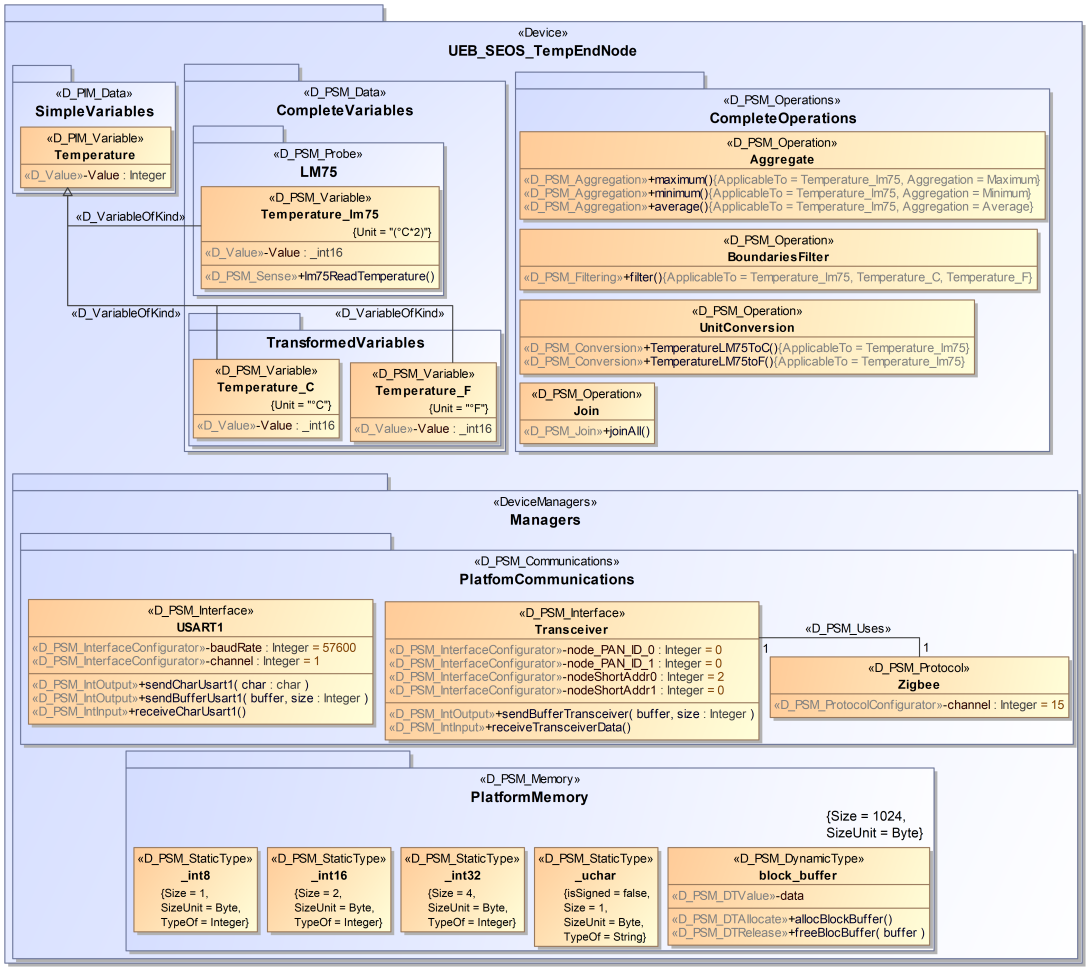

Figure 5.11: DM model for the temperature-sensing end-node, used by the *Temperature_SensorNode* in Figure 5.9.

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*). ❑

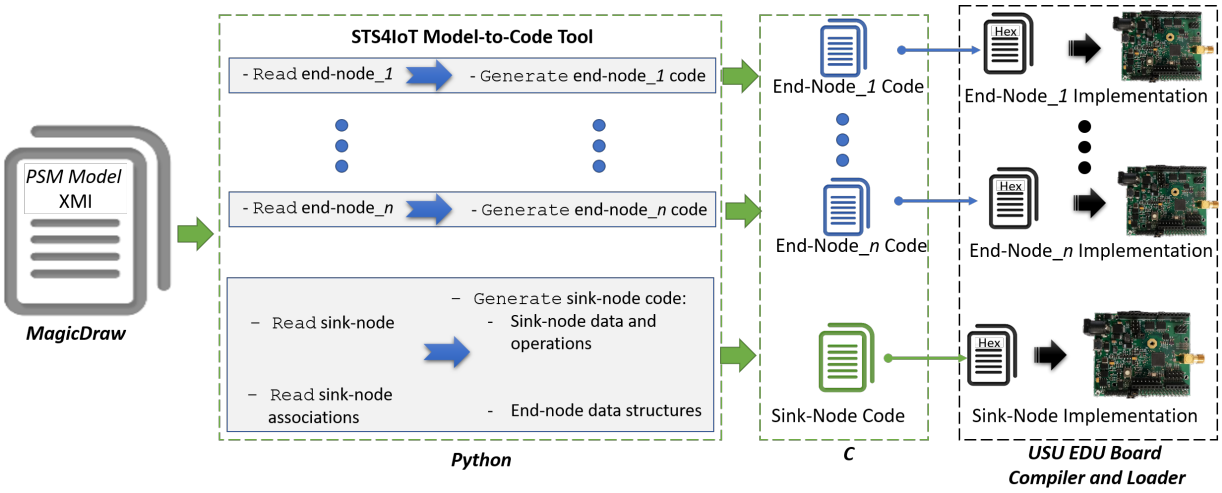

Figure 5.12: Implementation process in the UEB using the STS4IoT profile and model-to-code tool.

5.3 IoT Data Implementation

This section explains the use of *STS4IoT Model-to-Code Tool* to automatically implement STS4IoT instance models into UEB devices running SEOS. As stated in the previous chapters, applications for SEOS are written in C and consist of two files: programming logic (*application.c*), and constants, data types and functions declaration (*application.h*). Each UEB device in the IoT-data application requires its own set of code files.

Figure 5.12 illustrates the complete implementation process for an IoT-data application using the STS4IoT suite.

In the first place, business users and IoT experts define the IoT application PIM and PSM models using STS4IoT profile (Sec. 5.1). We use the CASE tool MagicDraw® for this step. MagicDraw® allows designers 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 [121] library to access the XMI data in a structured manner and the DateTime [122] library to generate code and output messages with timestamps. It follows the Object-Oriented paradigm to have independent modules for reading the models, organizing the information, and generating the IoT code. Thus, its extension to different IoT platforms is simpler.

In the first step (Figure 5.13), the *STS4IoT Model-to-Code Tool* reads the application's PSM and DM UML models in XMI. Then, it parses all the UML elements into Python objects to ease

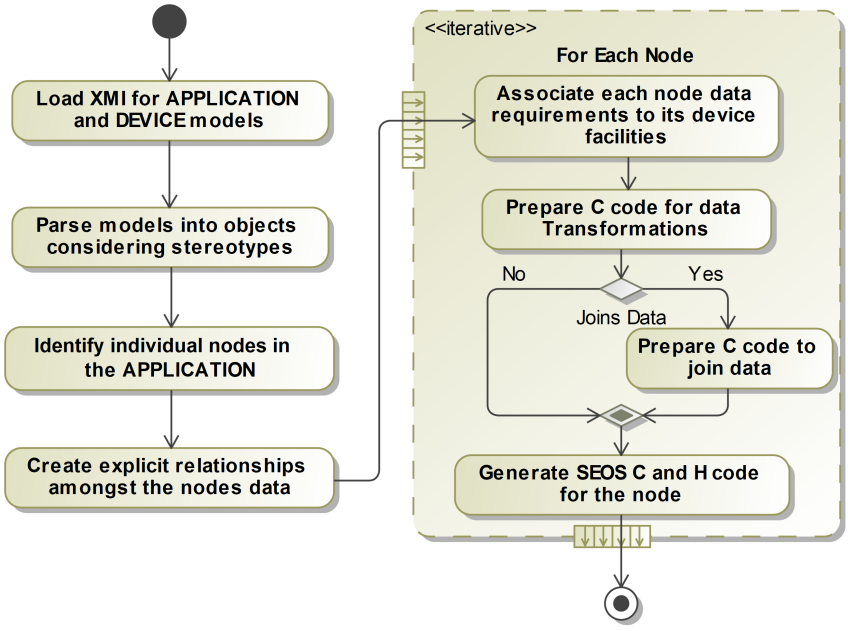

Figure 5.13: Activity diagram of the STS4IoT Model-to-Code Tool.

their analysis and use in the tool. From these objects, the script identifies each IoT (end or sink) node in the application and adds them their particular details:

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 [80]), with sending frequency and data destination.

With these details, the fourth step (Figure 5.13) is the explicit association of the node objects with their neighbours, *i.e.* the each EN with their SN, and the SN with its ENs. Then, the tool begins to generate the code for each IoT node, starting with the ENs and finalising with the SN.

This process (the iterative part in Figure 5.13) starts with the preparation of a code template for the particular implementation device that is filled with the application's details for the node. Then, the tool forges and includes the defined transformations in the appropriate locations according to the PSM. If the node must join data (*e.g.* the SN), the tool also adds the necessary code blocks. Finally, it generates the node's code files, *i.e. application.c* and *application.h*.

In this way, the *STS4IoT Model-to-Code Tool* can produce the code for each device in the IoT-data application.

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;
```

Figure 5.14: Code fragments of SN^2 for receiving transformed data.

Example 9 (SN code): Figure 5.14 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 (Figures 2.2-B and 5.9). At the top of Figure 5.14, 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 Figure 5.14, 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. 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. IoT experts have two options at this point: (i) They can use the UEB tools to compile the application code of each node into hexadecimal files and load them into the corresponding IoT hardware devices; (ii) Or they could also directly load the code into the UEB simulator to test their logic before the final implementation.

5.4 Integrating IoT Data into BI systems

The integration methodology of Sec. 4.4.1 (Figure 4.13) is valid for both sensor- and IoT-based BI applications. Consequently, the naive and ETL approaches for the conceptual design of the whole system are still valid. Nevertheless, the SensorDataModel profile limits the use of the Sensor Stream Data Warehouse approach for modelling IoT-data BI applications in a fully integrated model (Sec. 4.4.2).

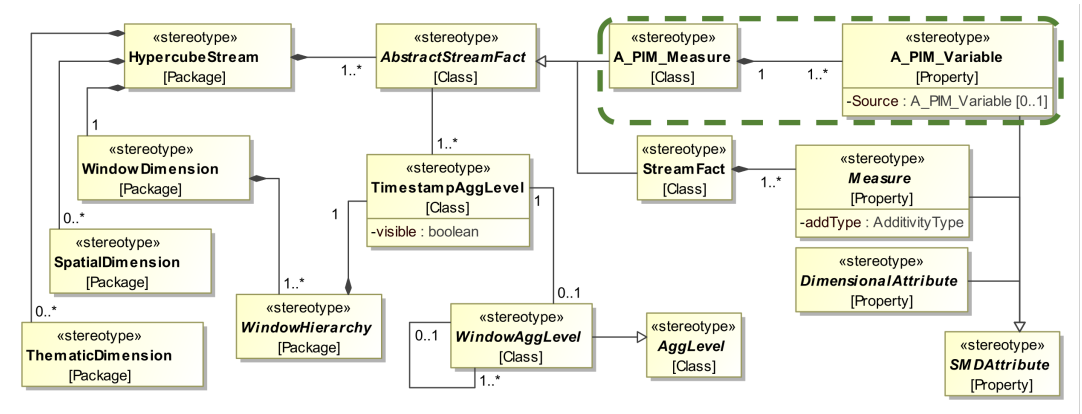

Figure 5.15: Simplified view of the IoT Stream Data Warehouse profile highlighting the updated stereotypes from STS4IoT.

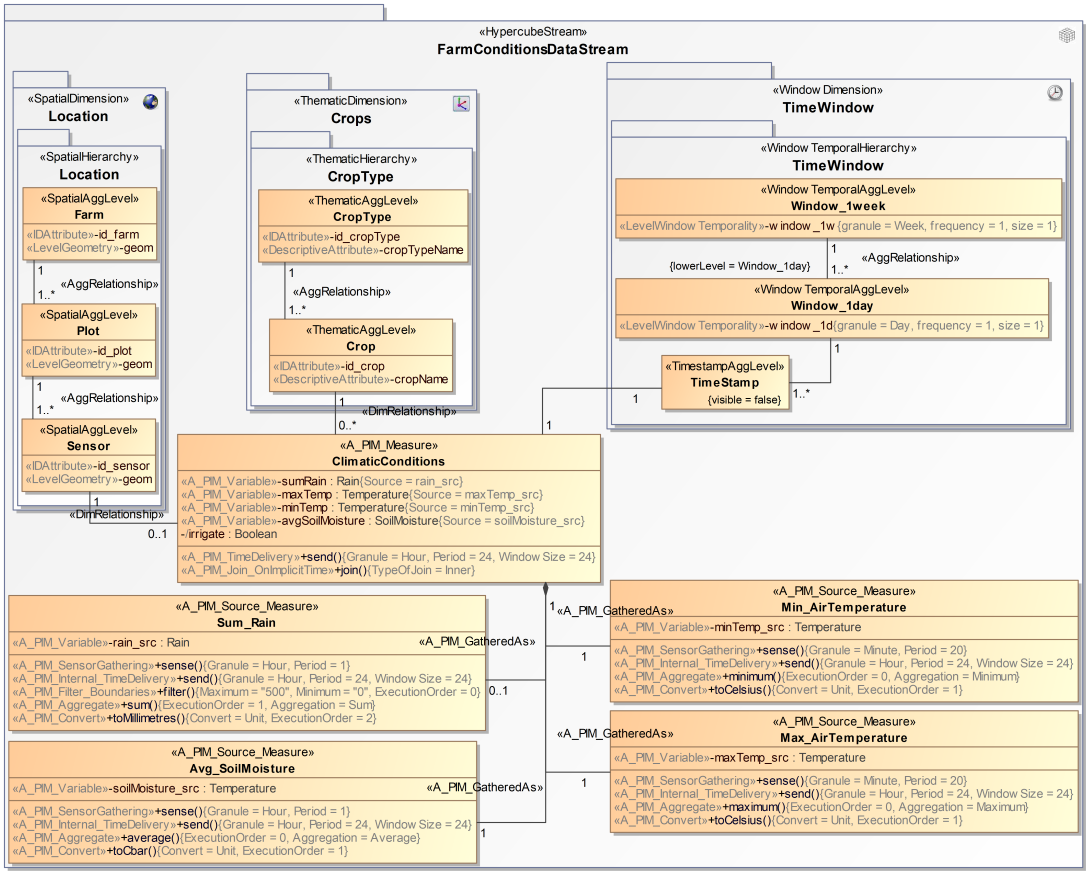

Figure 5.16: Example of an integrated model for the IoT-based BI application in our case study.

The STS4IoT meta-models are an extension of the SensorDataModel and SensorDeviceModel meta-models. Indeed, STS4IoT-PIM kept (although updated) the most relevant stereotypes for the integration: `A_PIM_Measure` and `A_PIM_Variable`. Therefore, a UML profile for IoT Stream Data Warehouses can rely on the same structure updating the relevant stereotypes from STS4IoT-PIM (Figure 5.15).

Figure 5.15 shows the updated profile for IoT-data BI applications, highlighting the novel stereotypes from STS4IoT. It looks very similar to the previous version (Figure 4.15). However, it is a simplified view of the whole profile. The updated stereotypes connect with the complete data and operations structure defined to support IoT data (Figure 5.2).

Example 8 (IoT Stream DW integration): Figure 5.16 defines a fully integrated model of the IoT-based BI application in our case study (Sec. 2.2). In particular, it integrates the PIM of the IoT-data application (Figure 5.3) into a SDW schema for analysing temporal streamed data according to the *`Location`* and *`CropType`*.

This model provides useful details about the sensed data that the previous version (Sec. 4.4.2) does not support. For instance, it defines individualised sensing and transformation rates and windows for every variable, besides all transformations that alter them. ❑

Summary

STS4IoT is the upgrade of the sensor-data MDA approach for supporting the design and implementation of IoT-data applications. Besides the basic concepts of sensors, the STS4IoT profile also models complex transformations of IoT data and the communications amongst devices, allowing for multiple implementation choices. Indeed, the STS4IoT model-to-code tool redefines the parsing process to generate the appropriate code for each IoT device.

Nonetheless, STS4IoT kept the high abstraction and separation of concerns from the sensor-data constructs. Indeed, its PIM seamlessly integrates into the sensor SDW profile to model and implement complete IoT-based BI applications following the SSBI-ODD methodology.

The STS4IoT MDA approach is published in [40], including the design methodology, UML profile and data-implementation process.

Chapter 6

Theoretical and Practical Validation

This chapter presents the theoretical assessments of the sensor-data and IoT-data profiles (Sec. 6.1), and completes them with a set of practical feasibility experiments of the MDA-based implementations (Sec. 6.2). Then, it demonstrates the capacity of the STS4IoT MDA approach to support multiple IoT devices for design and implementation (Sec. 6.3).

6.1 Theoretical Assessment

This section presents a set of theoretical assessments that validate the sensor-data (Sec. 6.1.1) and IoT-data (Sec. 6.1.2) profiles following different approaches in the literature [23, 123, 124, 5].

First, meta-models must comply with the specific structure and constraints of the modelling language [23]. In the same way, they must define and enforce rules to provide well-formed instance models [123]. Therefore, we verify the compliance of each profile with the UML standard and its capacity to define well-formed instance models with the CASE tool MagicDraw®.

Second, the structure and composition of meta-models are direct indicators of their quality, *i.e.* their expected ability to represent all systems in their domain. Hence, we assess the quality of the sensor-data and IoT-data profiles using the quality framework of [124, 5].

6.1.1 Sensor-Data profile validation

SensorDataModel and SensorDeviceModel CASE-tool validation

With the SensorDataModel and SensorDeviceModel profiles defined in the MagicDraw® CASE tool (Sec. 4.2), we checked their correctness and completeness in the UML standard [23, 123] using the default tests provided in the validation 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) [125].

The results of our profiles in all these tests were positive (Figure 6.1).

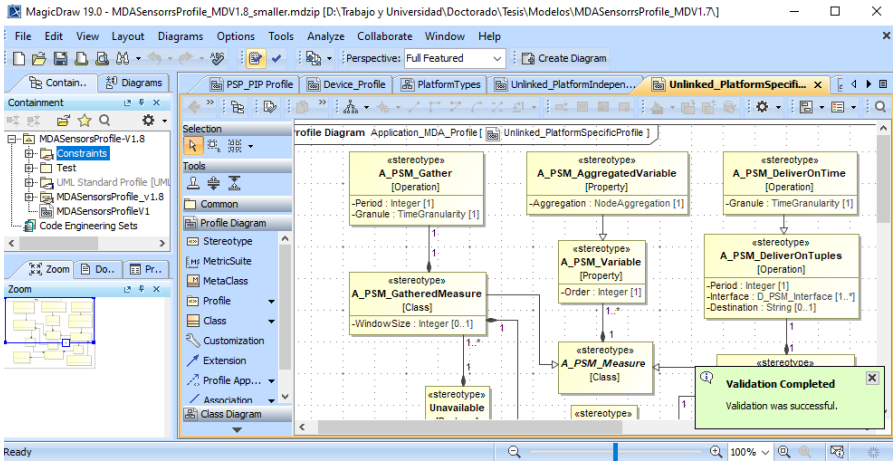

Figure 6.1: Successful validation of the SensorDataModel and SensorDeviceModel profiles in MagicDraw®.

Furthermore, we checked whether the SensorDataModel profiles could produce well-formed instance models [123]. Hence, we prepared a set of corrupted PIM and PSM models, and correct PIM and PSM models following all the OCL rules described in Figures 4.4 (`WindowSize` must be greater than one), 4.7 (`A_PSM_GatheredMeasure` must define the `WindowSize`) and 4.10 (SensorDataModel and SensorDeviceModel associations). MagicDraw® identified all the (intended) errors in the corrupted models (Figure 6.2), while the good models were considered appropriate.

SensorDataModel and SensorDeviceModel meta-models quality

Meeting the basic modelling language rules and providing well-formation constraints does not guarantee the usability of the SensorDataModel and SensorDeviceModel profiles. Indeed, meta-models can grow highly complex and become difficult to understand, maintain and use; or be so simple that they cannot represent most applications in their domain. Therefore, [124] defines a quality framework to assess the quality of a meta-model from its components and structure. This framework considers five quality properties of every meta-model:

- *Reusability* is the capacity of a meta-model to contribute with its constructs to the defi-

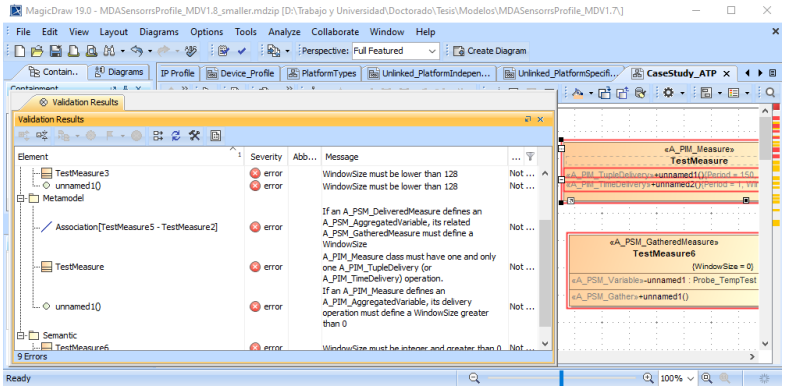

Figure 6.2: Failed validation in the corrupted SensorDataModel instance models.

nition of new meta-models. It is positively related to the number of meta-classes, rules, and meta-attributes; but negative related 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 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 decreases with the number of separate inheritance hierarchies and inter-associations amongst meta-classes.

Moreover, [5] implemented this quality framework [124] and evaluated more than 2500 meta-models from different domains available in the literature. Besides, Basciani *et al.* [5] published their raw results online, establishing a comparison framework for other meta-models such as our SensorDataModel and SensorDeviceModel profiles (Table 6.1).

Consequently, we compare the results of the sensor-data (PIM, PSM and DM) profiles with those from the literature using the percentile rank (Table 6.1). This ranking provides an estimation of the overall quality of our meta-models.

Table 6.1: Quality assessment and percentile rank of the sensor-data profiles considering the meta-models evaluated in [5].

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*
SensorDataModel-PIM	6.70	*76*	5.50	*99*	3.40	*53*	0.30	*11*	6.40	*99*
SensorDataModel-PSM	3.53	*59*	1.77	*99*	2.27	*29*	0.70	*22*	2.63	*99*
SensorDeviceModel (DM)	5.24	*70*	0.51	*97*	3.36	*52*	2.84	*56*	2.12	*99*

The percentile rank results (Table 6.1) evidence that both the *Understandability* and *Well-Structured* properties of our sensor-data meta-models are outstanding. Thus, designers can seamlessly instantiate them for their sensor applications. Besides, their *Reusability* is good (well above the median). Indeed, this allowed for a smooth integration of the PIM into another meta-model (Sec. 4.4.2).

These results are (partially) favoured by the simplicity of the models, especially the SensorDataModel-PIM. However, this simplicity also reduced the *Functionality* and *Extendibility*. Therefore, the range of supported applications (especially in the SensorDataModel-PSM) is narrow, yet, the supported devices (SensorDeviceModel) are around the median. Besides, the maintenance and upgrading of the profiles may require more effort, particularly for the PIM.

6.1.2 IoT-Data profile validation

STS4IoT CASE-tool validation

This test uses the validation suite of MagicDraw® to verify the STS4IoT meta-models (Sec. 5.2) correctness and completeness in the UML standard with the default tests provided in its suite [23, 123]. The STS4IoT profile resulted valid (Figure 6.3) in the nine default tests of the suite (*cf.* 6.1.1 and [125]): *Diagram Merge*, *Numbering Validation*, *Orphaned Proxies*, *Parameters Synchronisation*, *Relationship Ownership*, *Shape Ownership*, *Spelling*, *UML Completeness Constraints*, and *UML Correctness*.

This test also verifies whether the STS4IoT profile can produce well-formed instance models [123]. Hence, it validates the profile OCL rules (*e.g.* Figures 5.4 and 5.5) on sets of intentionally corrupted PIM and PSM models, and correct PIM and PSM models. For instance, these OCL check whether the sensing and delivery Operations are appropriate according to the Class Stereotype. MagicDraw® identified all the OCL violations of the corrupted models (Figure 6.4) and returned a successful validation for the correct ones.

STS4IoT meta-models quality

This section assesses the STS4IoT profile quality using the quality framework for meta-models proposed by [124] and implemented in [5].

In particular, we compare our proposal to other existing data-centric UML profiles and meta-models from multiple domains (not necessarily IoT) [5] using the percentile rank. Besides, we

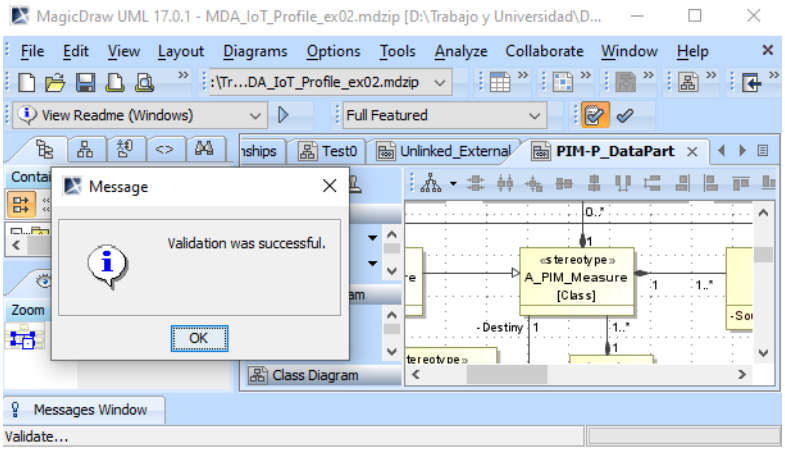

Figure 6.3: Successful validation of the STS4IoT profile.

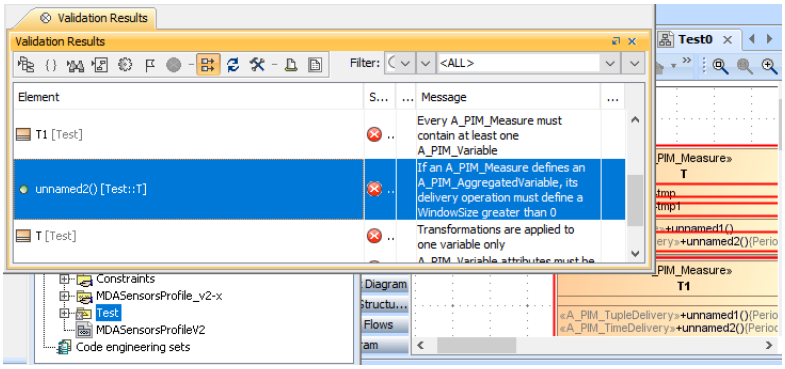

Figure 6.4: Failed validation in the corrupted models.

evaluate it against the SensorDataModel and SensorDeviceModel profiles (Sec. 6.1.1) to study the evolution of the profile.

The percentile rank comparison with [5] results (Table 6.2) allows noticing that the *Understandability* and *Well-structured* of STS4IoT PIM and DM profiles are very good. In the same way, even though these quality properties were lower in the PSM, it still ranks amongst the top 2% of meta-models in the literature. 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

Table 6.2: Quality assessment results and percentile rank considering the meta-models evaluated in [5].

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*

complete applications as in Sec. 5.4. Regarding the *Extensibility* of STS4IoT, it does not allow for easy upgrading since the three profiles are under the median, especially the PIM.

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 [5], 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 6.3 presents the quality assessment ([124]) of each meta-model in STS4IoT: PIM, PSM and DM; comparing them with their previous versions (Chapter 4).

Table 6.3: Upgrade in terms of quality: Comparison between STS4IoT and the sensor-data profile (Chapter 4).

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

For the PIM evolution (Table 6.3), 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 STS4IoT-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 6.3), 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.

6.2 Practical Feasibility: SEOS Experiments

The practical feasibility experiments leverage the context of the motivating case study (Sec. 2.2) to better evidence the impact of our MDA approaches for sensor- and IoT-data applications.

The case study proposes a smart-agriculture scenario requiring crop and environmental data for irrigation. The maximum and minimum temperature of each day allows for estimating the water necessities of a plant. The soil moisture indicates the capacity of the ground to provide the necessary water to the plants. The rainfall states whether irrigation is appropriate [76]. Moreover, the deployment of the sensor- or IoT-data applications depends on the specific constraints and characteristics of the monitored plot and the needs of business users (Figures 2.1 and 2.2).

The first experiment (Sec. 6.2.1) evaluates and compares the development process for a temperature-sensing application using the SensorDataModel and SensorDeviceModel profiles. The second experiment (Sec. 6.2.2) proves how the STS4IoT profile and model-to-code tool help the design and implementation process of the case-study IoT-data application. These experiments use the UEB platform running SEOS.

6.2.1 Sensor-Data MDA Feasibility Experiment

This experiment validates the feasibility and performance of our MDA approach for sensor-data applications in typical scenario based on the case study (Sec. 2.2). Since there are currently no directly comparable model-driven approaches in the literature (*cf.* Chapter 3), it provides a comparison with a manual approach using comparable criteria. Both approaches use UEB with SEOS as implementation platform.

Experimental setting

In particular, this experiment verifies the *Specification*, *Usage* and *Implementation* metrics of SensorDataModel instance models as defined by [126]:

- Specification is composed of *Legibility* and *Simplicity* (issued from the model [126], ranging from 0 (extremely low) to 1 (high)).
- Usage is composed of *Completeness* and *Understandability* (assessed by the business user as accepted or rejected according to their needs).
- Implementation is described by *Implementability* (the percentage of code automatically generated [17]), and *Maintainability* (the time until a new requirement is satisfied).

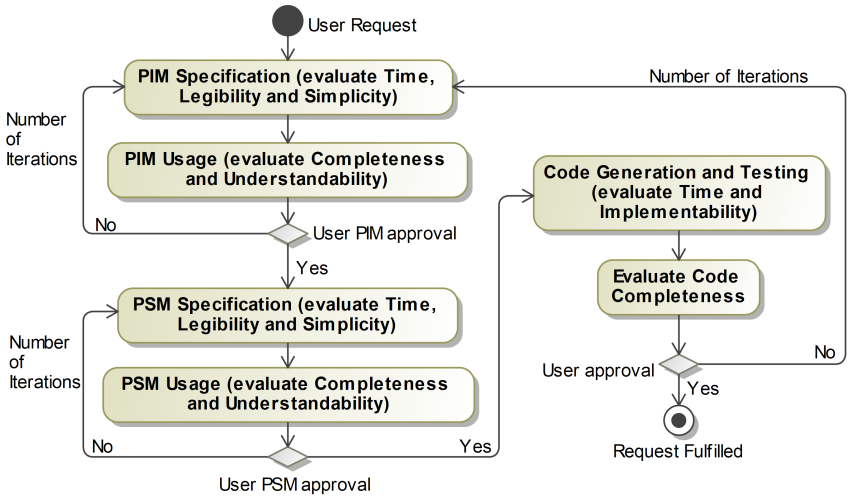

Figure 6.5: Flow of the sensor-data MDA feasibility experiment.

According to [126], **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 [124]. 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 meta-model [124]. 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 [124], and the capacity of the automatic implementation tool.

Moreover, this experiment considers Time as a metric for effort [17]. Indeed, it uses Time to compare the manual and model-driven processes since the previous quality metrics only apply to models [17].

The experiment scenario is the following one: A business user in the agricultural domain requests in natural language a sensors expert to collect specific data from a particular plot. To allow verifying *Maintainability*, the business user divides their requests in two: **i)** *Give me the average pressure, and maximum and minimum temperature from plot X every five minutes*. **ii)** *Now, add a light reading from the same point every five minutes*. The sensors expert uses our proposal to design and develop the sensor-data application, validating every step with the business user (Figure 6.5).

A PhD student working on irrigation systems played the role of business user, while a PhD student working on IoT systems played the role of sensors expert in this case. Both the business user and the sensors expert are skilled in UML. Besides, the sensors expert received preliminary training in our sensor-data MDA proposal. The previously described metrics are measured in the following way (Figure 6.5):

- We evaluate both the *Specification* and *Usage* on each validation round. If the user rejects

the *Completeness* or the *Understandability*, the expert must repeat the same step, creating a new iteration.

- We measure the *Implementability* after completing the modelling steps (PIM and PSM) when the expert uses our *Model-to-Code Tool* (*cf.* 4.3), and checks whether it could satisfy the user request. Besides, after receiving the working application, the user could accept it, or reject it and return to the first modelling step (Sec. 4.4). In particular, the implementation time in this experiment also considers the simulation time, which should take 11 minutes on average.

- Finally, to allow measuring the *Maintainability*, the user requests for a change into the accepted application and the expert follows the previous steps to provide it.

Furthermore, for comparing with a manual approach, the same business user made the same requests to a different sensors expert (a voluntary IoT engineer) previously trained in the use of the UEB platform (but not in our profile), who directly addressed the solution in code. For the non-automatic development we measured the number of iterations for each request, the time to deliver each iteration, the acceptance of the received application and the code reuse.

Experimental results

Tables 6.4 and 6.5 show the results for each iteration on each modelling and developing step for the first and second requests, respectively. Moreover, Figure 6.6 presents the final models for each request. For space reasons, we do not include the initial and intermediate models nor the code.

Table 6.4: Results for the first request

Step	Number of Iterations	Iteration	Time to deliver	Specification		Usage		Implementation
				Legibility	Simplicity	Completeness	Understandability	Implementability
PIM	2	1	0:22:43	1	1	Refused: Aggregate 20 samples.	Accepted	-
		2	0:00:45	1	1	Accepted	Accepted	-
PSM	2	1	0:17:59	1	0.67	Refused: Sample every 10 seconds.	Accepted: Improve operation names.	-
		2	0:03:08	1	0.67	Accepted	Accepted	-
Code	1	1	0:15:41	-	-	Accepted	-	99.57%
Total Time			**1:00:16**					

The results of this experiment (Tables 6.4 and 6.5) evidence that the models designed with our UML profile can be easily understood by business users with UML knowledge. Besides, they also exhibit that the high-abstraction level (PIM) is much simpler than the low-abstraction level (PSM), yet they are equally legible [126]. Moreover, the sensors expert had to modify less than 1% of the code for both requests, indicating a good Implementability. This metric was not perfect since there still some device-specific attributes that must be configured manually; yet, it

Table 6.5: Results for the second request, including Maintainability (Total Time).

Step	Number of Iterations	Iteration	Time to deliver	Specification		Usage		Implementation
				Legibility	Simplicity	Completeness	Understandability	Implementability
PIM	1	1	0:03:46	1	1	Accepted	Accepted	-
PSM	1	1	0:02:21	1	0.67	Accepted	Accepted	-
Code	1	1	0:14:18	-	-	Accepted	-	99.21%
Total Time			**0:20:25**					

is a significant aid for sensors experts. Finally, the Maintainability was good since the update process took about one third of the time than the original process with high acceptability from the user.

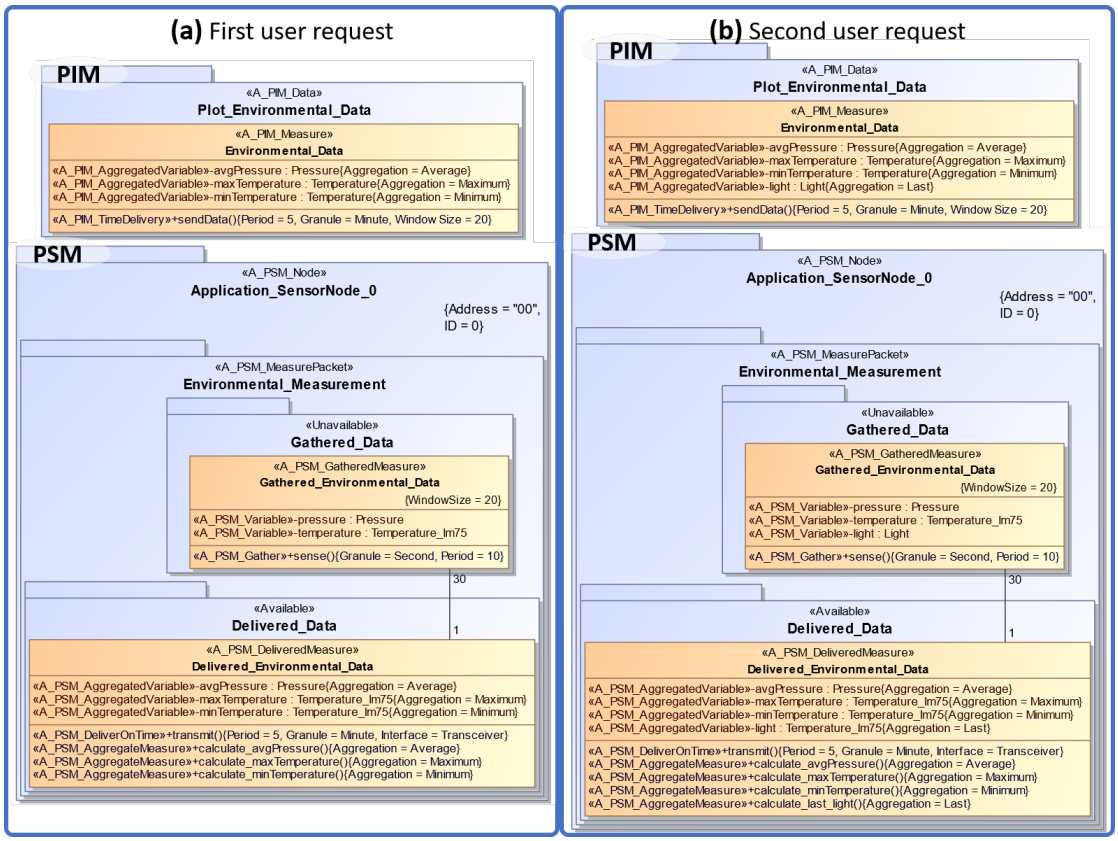

Figure 6.6: Resulting models for the first (a) and second (b) user requests.

Furthermore, Table 6.6 presents the results for the manual-development approach. It includes the resulting time for each iteration until achieving an appropriate delivery, including a brief summary of the rejection motives and the percentage of code reuse. The coding time also includes the simulations that the sensors expert had to make to validate the outcomes.

Comparing the results of our proposal (Tables 6.4 and 6.5) with those of the manual approach

Table 6.6: Results of the non-automatic development method.

Request	Number of Iterations	Iteration	Time to Deliver	Acceptance	Code Reuse
1	5	1	1:16:21	No: The sensor is sending 500 values per variable, only the aggregates are required.	0%
		2	0:28:38	No: Only the average Pressure, and the maximum and minimum Temperatures are required, other aggregations are not needed.	38%
		3	0:15:57	No: The sampling rate is 1.6Hz, reduce to 0.1Hz.	89%
		4	0:14:22	No: The aggregation window is 300 seconds, reduce to 200 seconds.	93%
		5	0:12:26	Yes	99%
	Request Time		2:27:44		
2	2	1	0:23:35	No: Only one value of Light is required, average is undesired.	80%
		2	0:16:55	Yes	88%
	Request Time		0:40:30		
Total Time			**3:08:14**		

(Table 6.6), the manual approach required less iterations, yet it took more than double the time. Besides, the sensors expert modified roughly 80% of their original code to get the acceptance of the user in the first request, and about 30% in the second one. Finally, we notice that all the rejection motives could have been discussed with our UML profile before writing the code, saving time and effort.

For this evaluation we considered a relatively simple case, yet our proposal was 50% faster. For more complex cases, particularly where the code volume is greater, the interest of an automatic approach would be much more apparent.

6.2.2 IoT-Data MDA Feasibility Experiment

This section reports the observations issued from the usage of STS4IoT MDA approach over the real case study described in Sec. 2.2. In particular, we put the design of PIM and PSM under test, analysing this process and the generated models (Figures 5.3, 5.8 and 5.9) following the evaluation framework of [126]. Also, we use some UEB devices running SEOS for the implementations of this experiment.

Experimental setting

In order to evaluate the instance of the proposed meta-model, we follow the quality assessment framework proposed by [126]. [126] propose a set of metrics to evaluate conceptual UMl models, grouped in three main classes: **Specification**,**Usage** and **Implementation** (measured as defined in Sec. 6.2.1). Besides, we also measure the time for each step as an estimation of the effort [17] as in the previous experiment (Sec. 6.2.1). However, the implementation time in this experiment does not include the simulation time since it can take at least two days, which

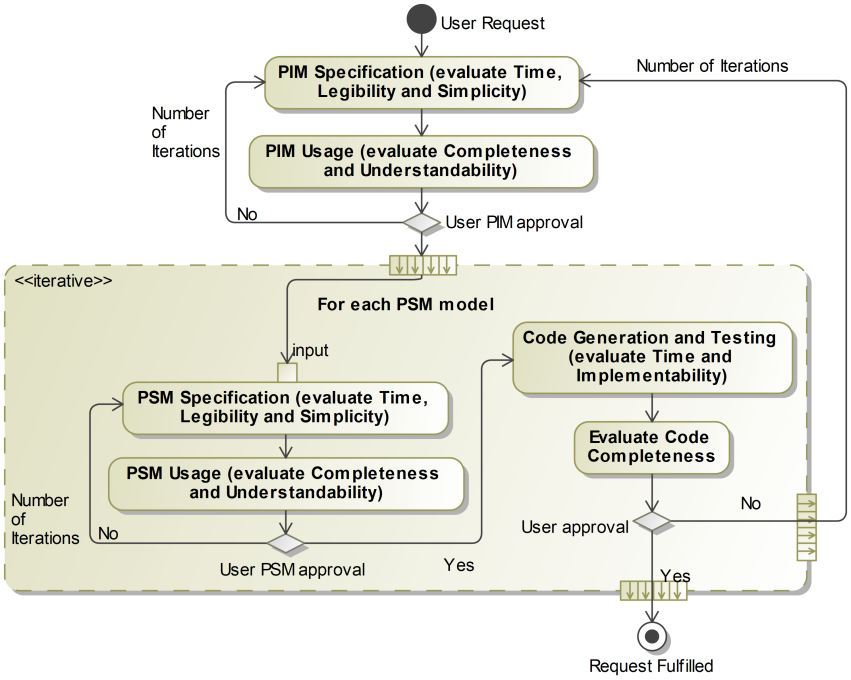

Figure 6.7: Design, implementation and evaluation flow of the IoT-data MDA feasibility experiment.

is longer than the rest of the process. Thus, we consider only the time to configure the STS4IoT model-to-code tool, generate and organise the code files, and successfully compile them.

For this experiment, a PhD expert in irrigation systems played the role of decision-maker and a PhD student working on IoT systems played the role of IoT expert. These users designed and implemented an IoT-data application following the IRRINOV method [76] described in Sec. 2.2.

The decision-maker and IoT expert followed the design methodology described in Sec. 5.1 to design and develop the experiment's application. Figure 6.7 provides closer details of the experiment flow, defining the measured metrics for each step [126]. 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. 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** (Figure 6.8).

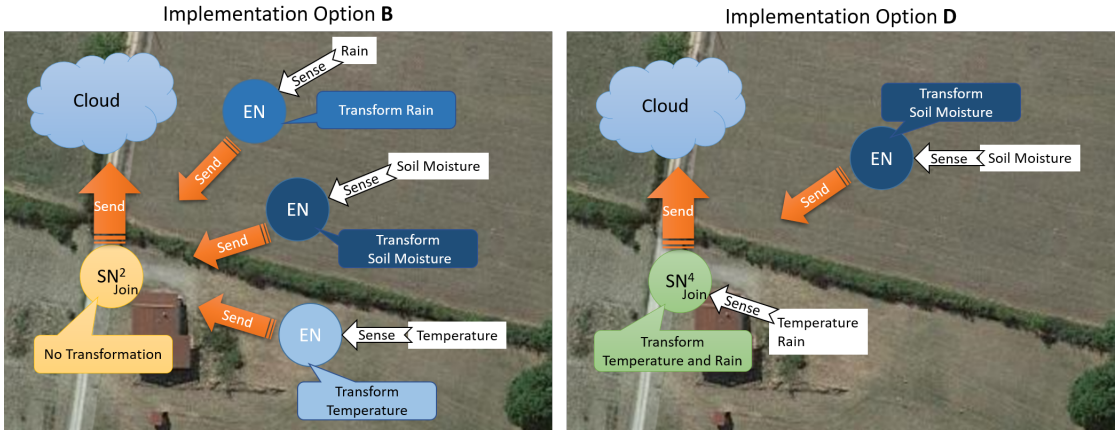

Figure 6.8: Recall of the case-study implementation options B and D, originally in Figure 2.2.

Experimental results

Table 6.7 shows the results for the PIM and PSM models of our case study. The models for the first implementation (Option **D**) are Figures 5.3 (PIM) and 5.8 (PSM). The PSM model for the second implementation (Option **B**) is in Figure 5.9. 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 [126]. 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 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

Table 6.7: Models analysis for implementation options **D** (first) and **B** (second). The second options does not require a PIM since it is the same model for both options.

Step	Number of Iterations	Iteration	Time to Deliver	Specification		Usage		Implementation	
				Legibility	Simplicity	Completeness	Understandability	Implement-ability	Maintain-ability
First Implementation Option (D)									
PIM	3	1	28:28	1	0.55	Reject: Filter measures; Calculate irrigation; Type of join.	Reject: Specify source measure.	-	**-**
		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%**

the rest of the code, and the manual modification is relatively short (around 30 minutes the first time for an application with two complex nodes).

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. For instance, the *Maintainability* of the PIM would be 100% considering it is the same model.

Consequently, the results in Table 6.7 are a good sign towards the confirmation of the theoretical assessment of our profile (Sec. 6.1.2). 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 and feasibility results would require further experimentation.

6.3 Supporting Multiple Devices: RIOT Example

The previous experiments (Sec. 6.2) evidence the feasibility of the Sensor-data and IoT-data profiles for real applications. These experiments concentrate on the UEB devices for modelling and implementation. Therefore, a question may arise concerning the generalisability of the whole MDA approach, *i.e.* the capacity to model and implement applications with different devices.

This section validates the support for various IoT devices in the IoT data profile to answer such a question. In particular, it tests whether the STS4IoT MDA approach can model (Sec. 6.3.1) and implement (Sec. 6.3.2) IoT-data applications using a different embedded OS: *RIOT* [55].

RIOT is an open-source OS widely used in industrial and academic applications. It supports several IoT devices of different ranges based on 8-bit, 16-bit and 32-bit microcontrollers. This OS allows for real-time and multi-threading processing, and supports multiple networking protocols including IPV6, Sigfox and LoRaWAN [55, 56].

6.3.1 Meta-model feasibility

The PIM, PSM and DM meta-models define the structures and notations for modelling IoT applications. Therefore, these meta-models must support the representation of any IoT implementation (and device) that meets the definitions in chapter 2. These definitions constrain IoT-data applications to homogeneous WSN and define IoT devices as advanced sensors that can receive, join and process data beyond aggregation.

According to the design methodology of Sec. 5.1, the DM is a necessary base for the PIM and specially the PSM. Therefore, this subsection presents the analysis of the STS4IoT meta-models and their instantiation for RIOT in that order.

In the first place, the DM meta-model (*cf.* 5.2.3) provides an abstract view of (some of) the facilities in an IoT device and its underlying embedded OS. It is thus a key meta-model for the support of various devices in the IoT-Data Profile. The main modelled concepts are supported variables, operations, communications interfaces and protocols, and static and dynamic data types. While data types are related to the embedded OS (which may have a broad range of devices), the other concepts are particular for each IoT device (although the OS determines their coding). For example, one RIOT device can sense light and use *Zigbee* for communications, while another kind of RIOT device can only sense temperature and use *BLE*. Therefore, each IoT device needs its own DM model, even if it uses the same embedded OS. Nevertheless, the meta-model should not require any particular adjustment to support such models. Indeed, the DM meta-model strictly focuses on the sensor and IoT device definitions in Section 2.1, only adding the particularities of data types, which are generic for embedded OS and do not inhibit the representation of the device.

Example 1 (STS4IoT-DM for a RIOT device): In the following, we present a DM for the IoT-LAB M3 node provided by the FIT IoT-LAB [3]. The M3 node is a low-end IoT device that supports RIOT amongst other embedded OSes. It can sense temperature, pressure and light from

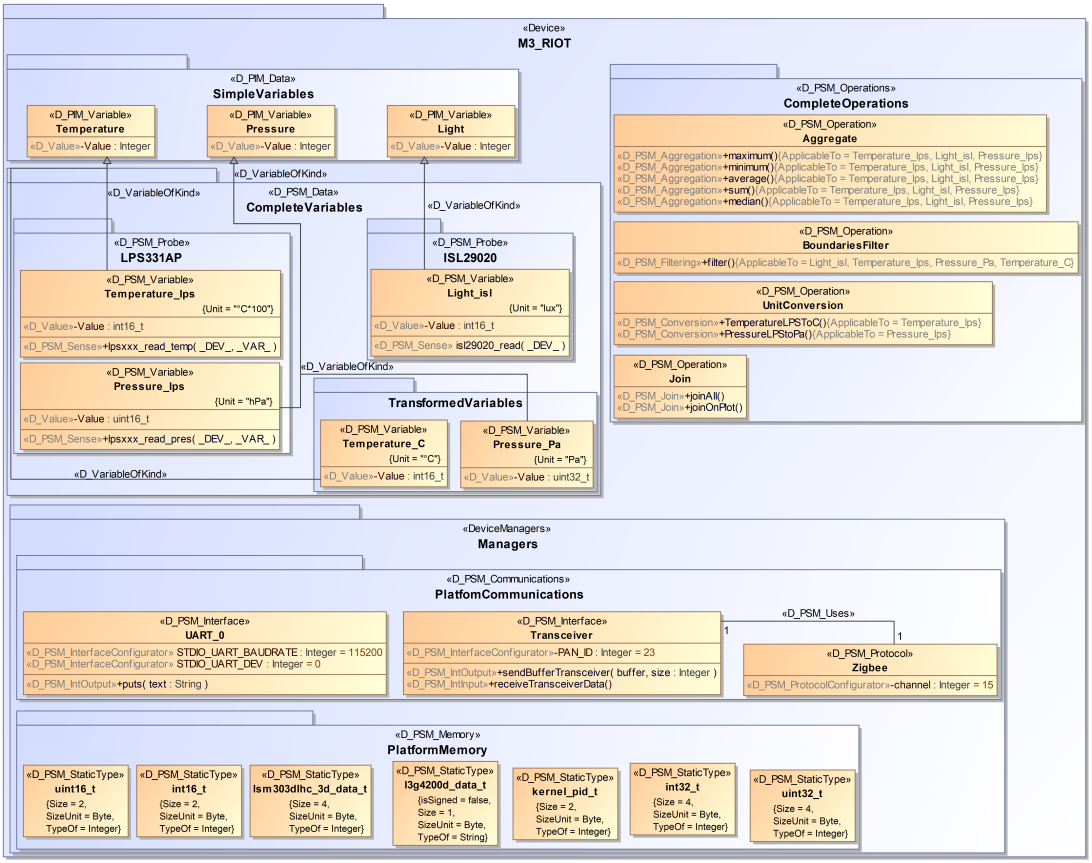

Figure 6.9: DM for IoT-LAB M3 node [3].

LPS331AP and ISL29020 proves; and also provides serial (UART) and wireless (IEEE 802.15.4 - Zigbee) communications facilities. Moreover, its core is an STM32F103RE microcontroller, which provides improved memory and processing capacity. Therefore, it is similar to the UEB in functionalities with different hardware and OS. Figure 6.9 shows the DM for this device. ❑

Secondly, the PIM meta-model (*cf.* 5.2.1) provides a simple view of IoT data without any particularity that may constrain it to a set of devices. Indeed, the data structure is independent of the network implementation and sensing capacities, and the operations are compulsory, conditional or optional according to the application requirements. Consequently, it should not have any problem modelling the data of any implementation that complies with the initial constraints (Sections 2.1 and 2.3).

Nevertheless, PIM instance models of the STS4IoT Profile (Sec. 5.2.1) are not free from IoT limitations. An expert review of the meta-model revealed that existing associations between a PIM and a DM restrict its capacity to represent user requirements [40]. A PIM cannot define a requested variable (*e.g.* light) if none of the available DMs senses it. Beyond altering traditional

MDA principles, this restriction reflects the reality of IoT since it cannot provide data if it does not have the hardware probe. Moreover, PIMs effectively represent users' data requirements independently of the implementation and devices if at least one device can supply the data.

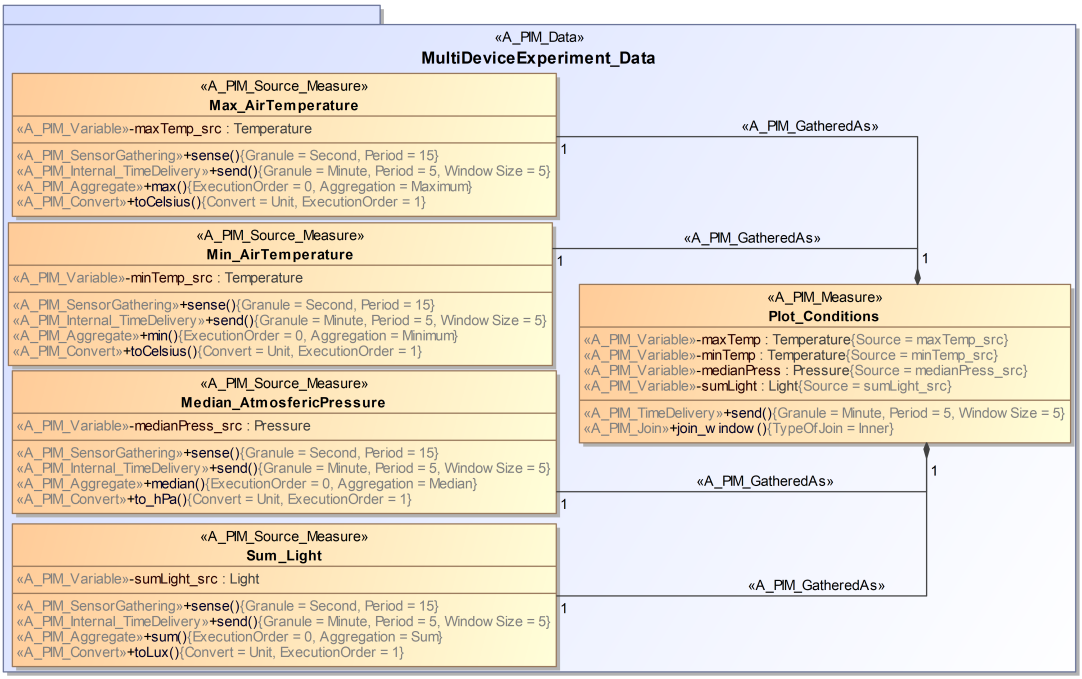

Figure 6.10: PIM for an IoT-data application based on the case-study that uses data from an M3 node.

Example 2 (STS4IoT-PIM using data from a RIOT device): Once the designers have the DMs that provide the data, they can proceed to model the PIM according to the users' requirements. This example bases its data requirements on the case study (Section 2.2). However, it changes soil moisture for atmospheric pressure and rainfall for light since the M3 nodes in the FIT IoT-LAB [3] cannot provide such data (Figure 6.9). It also simplifies the future testing by reducing the sensing and sending periods to 15 seconds and 5 minutes, respectively. Figure 6.10 shows the PIM for this IoT-data application using data provided by an M3 node. ❑

Finally, the PSM meta-model (*cf.* 5.2.2) provides the implementation view of IoT data, which may restrict its modelling scope to particular devices or implementations. The data structure is more oriented to design the network distribution of IoT data and its processing. Yet, it is not even limited to the star topology (*cf.* 2.3.2); it enables modelling single-device, tree and mesh implementations. Also, it does not force any particular data type; designers can specify this aspect in the model by linking the specific DM (Sec. 5.2). Similarly, all operations are conditional, *i.e.* they have mandatory or restricted usages. Besides, designers define their particular execution in the IoT device through links with the specific DM directly in the model (Sec.

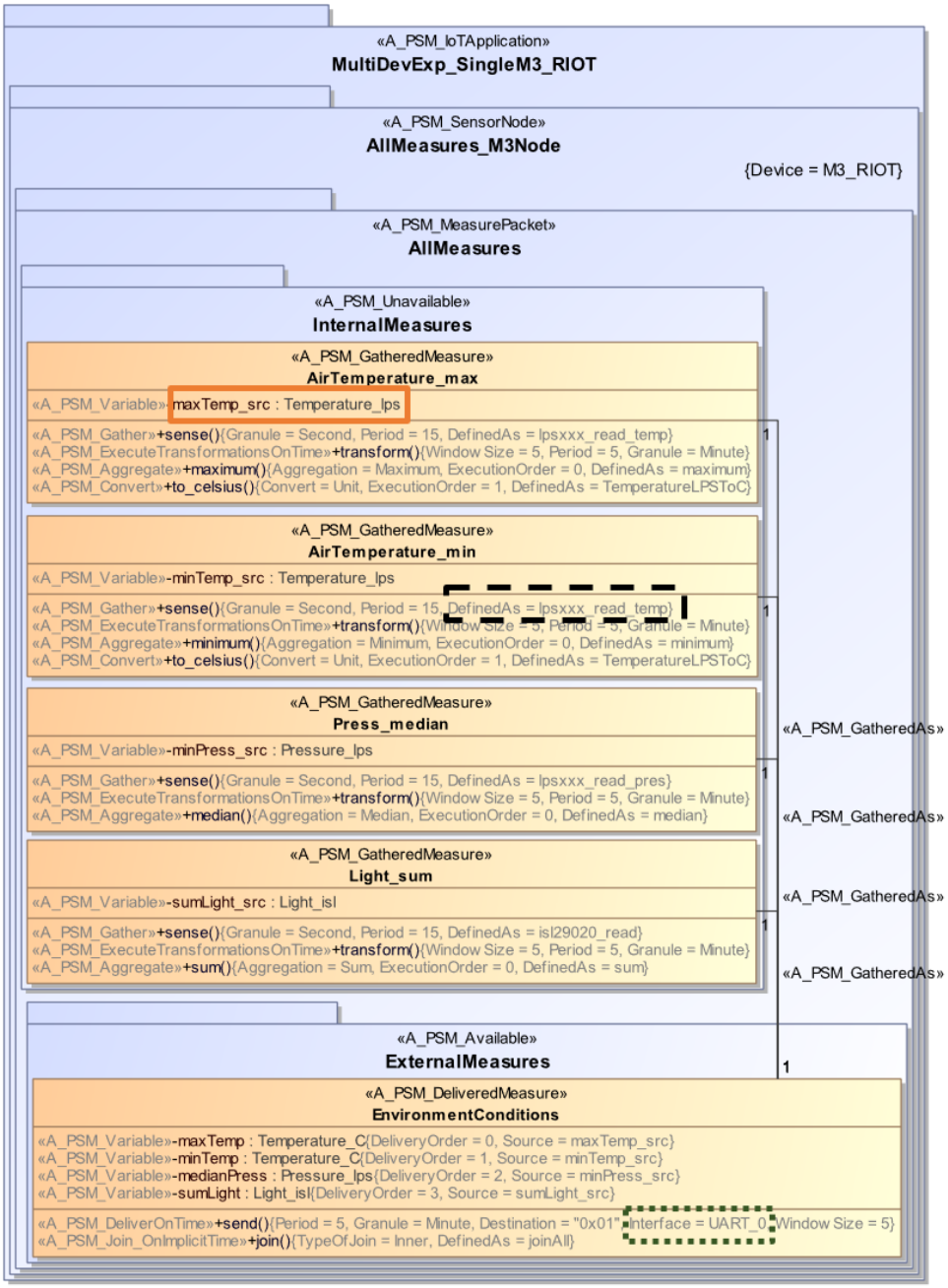

Figure 6.11: PSM for the IoT-data application in Figure 6.10 using a single M3 node.

5.2). In this way, designers have enough freedom to represent different implementations while are equally obliged to comply with some constraints. Even though these constraints may reduce the functionality of the meta-model (*cf.* 6.1.2), they secure that the model is implementable. Consequently, there are strong indications that the PSM meta-model does support any feasible IoT implementation considering the definitions in Sections 2.1 and 2.3.

Example 3 (STS4IoT-PSM using a RIOT device): After designing the PIM (Figure 6.10), IoT experts must define the PSM by considering the implementation option and device. For simplicity, this example uses a single IoT-LAB M3 node [3] running RIOT to provide the required data. Figure 6.11 provides the PSM for this implementation. Moreover, Figure 6.11 highlight three key parts that are bound to the M3 node DM (Figure 6.9): First, the simple orange line denotes the variable type for temperature (`Temperature_lps`), which links the LPS331AP probe. Similarly, the dashed black line encloses the `DefinedAs` Tag for sensing temperature (`lpsxxx_read_temp`) from the defined (LPS331AP) probe. Finally, the dotted green line emphasises the serial communication interface (`UART_0`) to send the sensed and transformed data. ❑

This analysis indicates that the IoT-data Profile is generic enough to support various implementations and devices, and it follows (most of) MDA guidelines. In the same way, instance models should keep the good *Specification* and *Usage* properties independently of the implementation platform [126].

6.3.2 Implementation feasibility

Furthermore, STS4IoT must also support the implementation in RIOT devices. Thus, this subsection presents the model-to-code tool update first. Then, it presents the RIOT implementation experiment.

This experiment aims to evidence that the updated model-to-code tool maintains the good *implementability* [126] showed in Sec. 6.2.2 independently of the implementation OS. Besides, it evaluates whether the automatically-generated RIOT code can provide the data defined in the models using the FIT IoT-LAB testbed [3].

Tool update

The MDA approach should also support the implementation of both the SEOS and RIOT PSMs in their respective hardware. Besides, the overall process should allow to generate the code for a new platform with minor modifications. Consequently, we have updated the STS4IoT model-to-code tool of Sec. 5.3 to support heterogeneous IoT-data applications involving multiple devices.

Figure 6.12 shows the activity diagram for the updated tool. The first five steps are the same (Figure 5.13) since they comprise the information extraction and initial parsing. Then, in the sixth step the tool must identify the target device of the particular node using the `Device` Tag. In the seventh step, the tool prepares the appropriate code template in the required language according to the OS specification. This template must consider the particular coding structure,

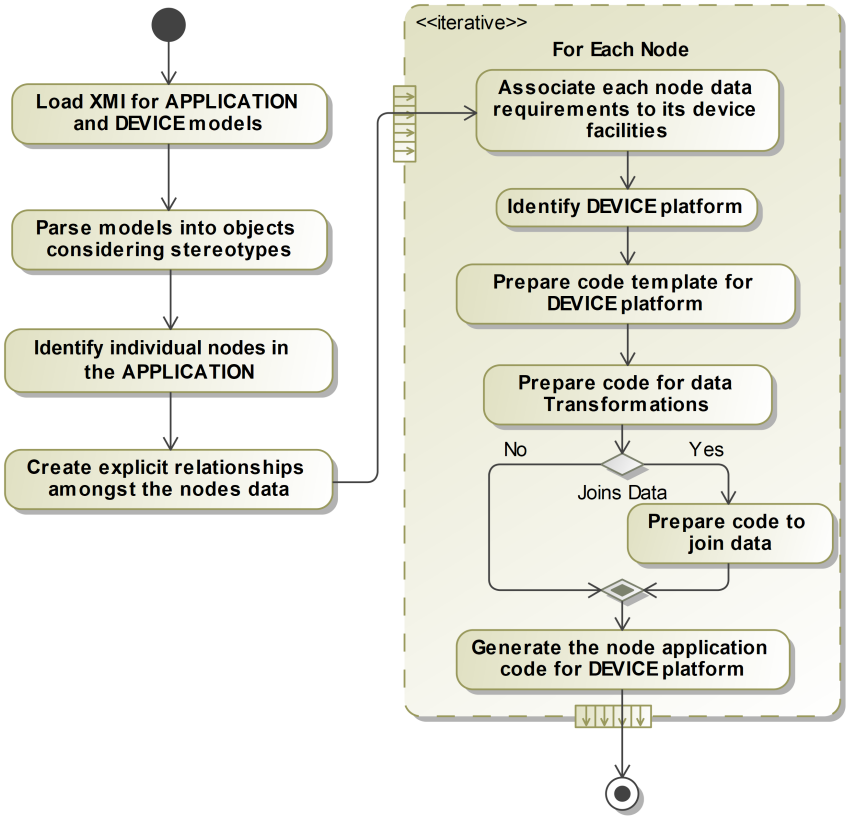

Figure 6.12: Activity diagram of the updated STS4IoT Model-to-Code Tool that supports multiple devices.

rules and APIs for each OS. In the eighth step, the tool prepares the necessary code to transform the data according to the PSM. In the ninth step, the tool prepares the code structures and logic to join internal and external data if the particular node requires it. Finally, in the 10th step, the tool generates the proper code for the particular node considering its implementation device.

Experimental setting

With the upgraded tool, designers can generate code for SEOS and RIOT devices from their respective PSMs. Therefore, this experiment evaluates the implementability of three heterogeneous RIOT PSMs, considering the percentage of code automatically generated by the tool and the coding time (*cf.* 6.2.2). Nevertheless, the implementability is deemed good if the objective IoT device can run the code [126, 17]. Consequently, this experiment also implements the IoT applications in the FIT IoT-LAB testbed [3] to evaluate whether the delivered data is properly sensed, transformed and sent.

The FIT IoT-LAB testbed is a European project to establish a suitable facility for experimenting with multiple heterogeneous IoT devices. IoT-LAB provides a web API to access, program, control and monitor more than 1000 physical devices of 26 types distributed amongst six sites in France. The testbed also supports multiple embedded OSes, although not all the devices support every OS. Currently, the most common OS in the testbed is RIOT [3, 4].

Figure 6.13: Physical IoT devices in the Saclay site of the FIT IoT-LAB. Source: [4].

Of the six IoT-LAB sites, Saclay provides the most heterogeneous display. In particular, room one contains six types of devices that support RIOT OS and IEEE 802.15.4 communications[1] (Figure 6.13). Nevertheless, only three types are selectable after an availability check:

- **IoT-LAB M3**[2]: Advanced sensor board designed for the testbed. It can provide temperature, light and atmospheric pressure data, amongst others. It uses a (comparatively) powerful Cortex M3 CPU, which allows it to run more complex programs.
- **Microchip SAMR21 Xplained Pro**[3]: Hardware platform to design and run IoT applications on a simple Cortex M0 CPU. The IoT-LAB versions include light and temperature sensors.
- **Arduino Zero**[4]: Development board using a Cortex M0 CPU for IoT applications. The IoT-LAB versions add an XBee communication module and an atmospheric pressure sensor, amongst other peripherals.

Figure 6.13 shows the actual IoT devices in the FIT IoT-LAB Saclay site.

This experiment uses the three nodes in a set of tests to verify the models' implementability in heterogeneous hardware and the compliance of the generated code with the models. It defines a star network based on the deployment option **A** of the Case Study (Section 2.2), with three simple end-nodes (ENs) that sense and send data and one complex sink node (SN) that joins and transforms each variable and joins and sends the variables (Figure 6.14).

[1] `https://www.iot-lab.info/docs/deployment/saclay/`
[2] `https://www.iot-lab.info/docs/boards/iot-lab-m3/`
[3] `https://www.iot-lab.info/docs/boards/microchip-samr21/`
[4] `https://www.iot-lab.info/docs/boards/arduino-zero/`

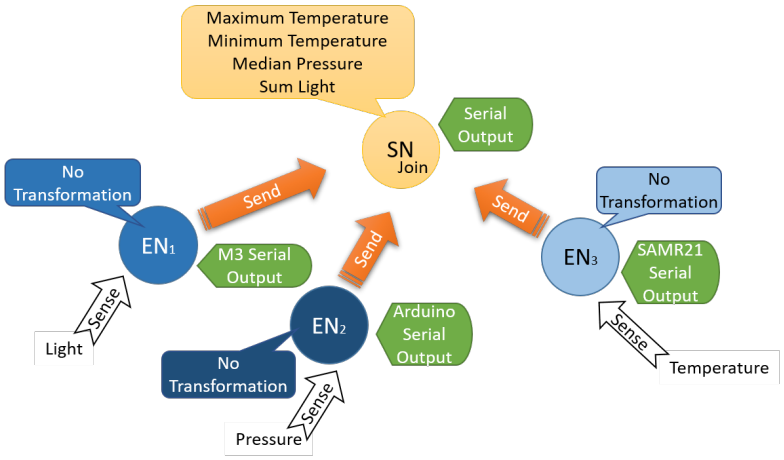

Figure 6.14: Experiment network following the deployment in Figure 2.2-A.

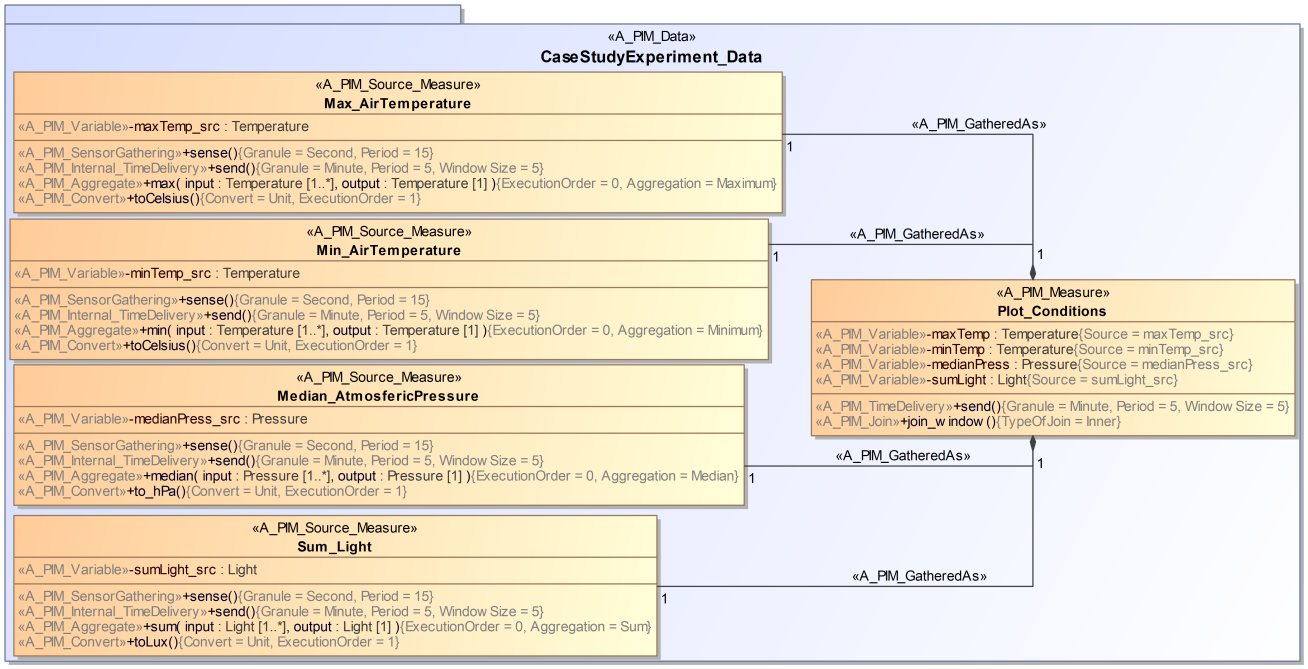

Figure 6.15: PIM of the experiment data.

In particular, the ENs must provide air Temperature, Light and atmospheric Pressure, while the SN must calculate *maximum* and *minimum* Temperature, *sum* Light, and *median* Pressure. Besides, all ENs send their data through both their wireless (IEEE 802.15.4) and wired (UART)

interfaces; the SN receives from its IEEE 802.15.4 interface and sends the transformed data through UART (Figure 6.14). The sensing and sending periods of ENs are 15 seconds, while the analysis and sending periods of the SN are 5 minutes. Figure 6.15 shows the PIM for such data.

In this way, the experiments mimic a real WSN scenario in a laboratory setting, where the serial aggregator[5] tool of the testbed collects the output data from all devices. Then, ENs serial outputs are processed using Pandas [127] to execute the same transformations and contrast them with the SN output.

Consequently, this experiment can evidence seven characteristics of the automatically-generated RIOT code:

1. All the selected devices can execute the code.
2. ENs sense their data according to the models.
3. All nodes can send their data as defined in the models.
4. The SN receives and incorporates data from the ENs.
5. The SN can join data from single and multiple sources.
6. The transformations are well-defined in the code following the models.
7. The application timing is consistent with the models.

Moreover, this experiment runs nine tests to increase its reliability, considering three different selections of devices and durations. In all the selections, one M3 node provides light, one SAMR21 node provides temperature, and one Arduino node provides pressure. The difference is in which node takes the role of SN (one selection for each type of node). The three durations are 30 minutes, one hour and 25 minutes (referenced simply as 1-hour), and one day.

Appendix B contains the complete PSMs for the three selections of devices: Figure B.1 for the experiments using **M3** as SN, Figure B.2 for the **SAMR21** SN, and Figure B.3 for the **Arduino Zero** SN.

Experimental results

Table 6.8 shows the implementability result for each selection (M3, SAMR21 or Arduino Zero as SN). The updated tool generated 100% of the RIOT code for each PSM in less than seven minutes. Indeed, only the first test (M3) was above five minutes since the second (SAMR21) and third (Arduino Zero) tests reused part of the initial code-generation and compilation process, saving time.

This result verifies that the IoT-Data MDA approach maintains the implementability of its models independently of the implementation platform. Furthermore, we need to evaluate the data sensing, transformation and sending.

[5] `https://www.iot-lab.info/docs/tools/serial-aggregator/`

Table 6.8: Implementability of an IoT-data application in RIOT.

RIOT IoT Device	Time	Implementability
M3	6:52	100.00%
SAMR21	4:25	100.00%
Arduino Zero	4:29	100.00%

Consequently, we implemented the generated IoT-data applications in the FIT IoT-LAB testbed with the different time configurations. Figure 6.16 shows the execution of two of the nine tests on the testbed. A maximum of two parallel tests were possible in the testbed due to a lack of available Arduino Zero devices.

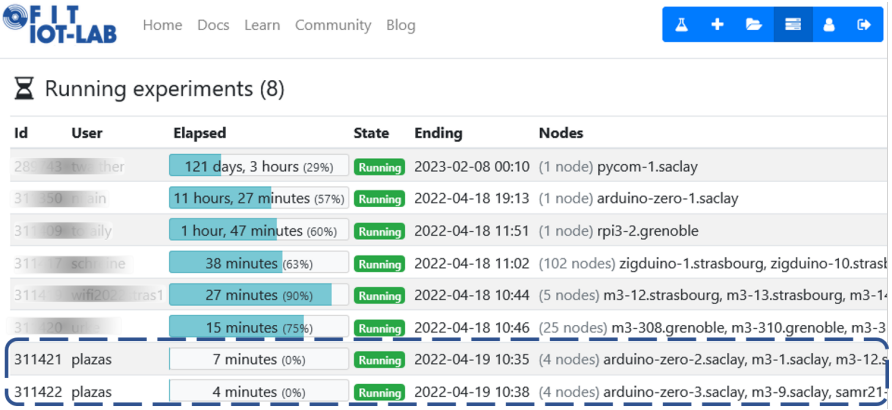

Figure 6.16: Tests execution in the FIT IoT-LAB testbed [3].

The generated RIOT code met almost all (**1**, **2**, **4**, **5** and **6**) defined characteristics in every test. Besides, data delivery (feature **3**) was appropriate in 78% of tests and timing (feature **7**) in 11%. Indeed, Table 6.9 presents the results of one error-free execution (30 minutes with SAMR21 as SN). The difference between sink-node data and externally-calculated data is 0.0 for all cases.

This result evidences that the implementation of automatically-generated code can provide data as requested in the PIM. Nevertheless, this was not always the case. For example, Table 6.10 shows the differences between sink-node data and externally-calculated data in the 1-hour test using an Arduino Zero board as SN. Although most data are consistent, problems with the wireless delivery process and time compliance caused the highlighted errors.

Firstly, the constant error in the Median Pressure (Table 6.10) could indicate that the EN is not sending data in the wireless interface, the SN is not receiving or joining data, or the transformation process in the SN is wrong. A problem with the transformation is highly improbable since the other executions (30-minutes and 1-day) of the same code did not present this problem in any instance. Problems receiving or joining the data are unlikely since the same device correctly receives and handles data from the other ENs in the same test. Moreover, an erroneous

Table 6.9: Comparison matrix for the 30-minutes test with SAMR21 as SN, which presented no errors.

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0

Table 6.10: Comparison matrix for the 1-hour test with Arduino Zero as SN. It highlights calculation errors in Light and Pressure caused by synchronisation and hardware problems.

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
0	0.0	0.0	0.0	*-756.0*
1	0.0	0.0	*60.0*	*-756.0*
2	0.0	0.0	0.0	*-756.0*
3	0.0	0.0	0.0	*-756.0*
4	0.0	0.0	0.0	*-756.0*
5	0.0	0.0	0.0	*-756.0*
6	0.0	0.0	0.0	*-756.0*
7	0.0	0.0	0.0	*-756.0*
8	0.0	0.0	0.0	*-756.0*
9	0.0	0.0	0.0	*-756.0*
10	0.0	0.0	*60.0*	*-756.0*
11	0.0	0.0	0.0	*-756.0*
12	0.0	0.0	0.0	*-756.0*
13	0.0	0.0	0.0	*-756.0*
14	0.0	0.0	0.0	*-756.0*

initialisation of the wireless module of the pressure EN would cause this effect on the data: it senses and sends pressure in the serial output but never transmits it to the SN. Indeed, this is an uncommon (but still relevant) issue with the Arduino-Zero nodes; the external XBee© module does not always start even though the code is correct [4]. This issue would also explain why 22% of the tests had the same problem (Appendix B). Consequently, there are hardware problems which may prevent the appropriate implementation of the models despite the correctness of the code.

Second, the two errors (13% of compared light values) may indicate timing issues. The small

```
1650133968.571483;m3-8;0, 60, 0
1650133969.598087;samr21-8;1, 26, 0
1650133971.040361;arduino-zero-3;3, 26, 0, 0, 0, 1200, 0, 0, 0
1650133974.515964;arduino-zero-2;2, 756, 0
1650133983.594497;m3-8;0, 60, 0
1650133984.666061;samr21-8;1, 25, 0
1650133989.853217;arduino-zero-2;2, 756, 0
1650133998.601205;m3-8;0, 60, 0
1650133999.734675;samr21-8;1, 25, 0
1650134005.190002;arduino-zero-2;2, 756, 0
1650134013.624307;m3-8;0, 60, 0
1650134014.802345;samr21-8;1, 26, 0
1650134020.528932;arduino-zero-2;2, 756, 0
1650134028.631078;m3-8;0, 60, 0
1650134029.869708;samr21-8;1, 26, 0
1650134035.866463;arduino-zero-2;2, 756, 0
1650134043.653745;m3-8;0, 60, 0
1650134044.936928;samr21-8;1, 26, 0
1650134051.204475;arduino-zero-2;2, 756, 0
1650134058.660768;m3-8;0, 60, 0
1650134060.005398;samr21-8;1, 26, 0
1650134066.542635;arduino-zero-2;2, 756, 0
1650134073.667650;m3-8;0, 60, 0
1650134075.072335;samr21-8;1, 26, 0
1650134081.879787;arduino-zero-2;2, 756, 0
1650134088.690707;m3-8;0, 60, 0
1650134090.139301;samr21-8;1, 26, 0
1650134097.217526;arduino-zero-2;2, 756, 0
1650134103.697509;m3-8;0, 60, 0
1650134105.206636;samr21-8;1, 26, 0
1650134112.556638;arduino-zero-2;2, 756, 0
1650134118.720568;m3-8;0, 60, 0
1650134120.272814;samr21-8;1, 26, 0
1650134127.895496;arduino-zero-2;2, 756, 0
1650134133.727180;m3-8;0, 60, 0
1650134135.341032;samr21-8;1, 26, 0
1650134143.233402;arduino-zero-2;2, 756, 0
1650134148.750132;m3-8;0, 60, 0
1650134150.407011;samr21-8;1, 26, 0
1650134158.572380;arduino-zero-2;2, 756, 0
1650134163.757082;m3-8;0, 60, 0
1650134165.473721;samr21-8;1, 26, 0
1650134173.910684;arduino-zero-2;2, 756, 0
1650134178.764040;m3-8;0, 60, 0
1650134180.541394;samr21-8;1, 26, 0
1650134189.249703;arduino-zero-2;2, 756, 0
1650134193.786778;m3-8;0, 60, 0
1650134195.610971;samr21-8;1, 26, 0
1650134204.588527;arduino-zero-2;2, 756, 0
1650134208.793798;m3-8;0, 60, 0
1650134210.676919;samr21-8;1, 26, 0
1650134219.926217;arduino-zero-2;2, 756, 0
1650134223.816638;m3-8;0, 60, 0
1650134225.743296;samr21-8;1, 26, 0
1650134235.264460;arduino-zero-2;2, 756, 0
1650134238.823566;m3-8;0, 60, 0
1650134240.811500;samr21-8;1, 26, 0
1650134250.602284;arduino-zero-2;2, 756, 0
1650134253.846360;m3-8;0, 60, 0
1650134255.878141;samr21-8;1, 26, 0
1650134265.940305;arduino-zero-2;2, 756, 0
1650134268.853226;m3-8;0, 60, 0
1650134269.736813;arduino-zero-3;3, 26, 0, 25, 0, 1200, 0, 0, 0
```

Figure 6.17: Fragment of the raw data output of the 1-hour test with Arduino Zero as SN highlighting one synchronisation error.

percentage and error value seems to be caused by only one omitted value. Therefore, errors in any other characteristic (*e.g.* join or transformation) are not probable.

Indeed, Figure 6.17 contains a fragment of the serial aggregator data from all ENs and the SN of this test, including the whole **Time Window 1**. It highlights in blue the SN output with the transformed data: Maximum Temperature of 26°C, Minimum Temperature of 25°C, and Sum Light of 1200 Lux; the previous paragraph discusses the erroneous Median Pressure (0 hPa). The SN produced this output at 18:37:49 (UTC). Thus, it should have considered data between 18:32:49 and 18:37:34 (*i.e.* 19 light values). The strict time consideration in Pandas made it clear that the light value in yellow arrived too soon and excluded it from the transformation. However, the EN puts the serial data shortly before sending it to the SN, and the time management in these devices is not perfect. Even though the total delay between two SN deliveries is 300 seconds, Figure 6.17 shows that the previous delivery was 298.7 seconds before. Hence, the SN had to consider the light value in yellow, but its data sending was one second too soon. Consequently, such tiny variations in timing caused multiple errors in 89% of tests, especially the longest ones. Appendix B contains the complete results from the nine tests.

These results and their analysis evidence that the automatically-generated RIOT code can produce IoT data as defined in the PIM. Nevertheless, actual applications could suffer from a lack of synchronisation and hardware-related problems which prevent delivering the appropriate data. Therefore, IoT implementations need to include better synchronisation and fault-tolerant strategies to provide the data as requested. Section 7.4 proposes possible future directions on this topic.

Summary

The theoretical qualities of the Sensor-data profile (Chapter 4) and the IoT-data profile (Chapter 5) are good. Indeed, the theoretical validation evidenced that the UML profile for sensor-data applications keeps the UML standard syntax and constraints, and instantiates well-formed application models. Besides, it has excelling quality parameters, although its maintainability and ability to represent several applications are below the average. Simmilarly, the theoretical validation of the STS4IoT profile evidenced that its overall quality is still outstanding in the most relevant aspects. Compared with the sensor-data profile quality, the PIM improved, the PSM slightly lowered, and the DM was about the same.

Moreover, the feasibility experiments have proved that MDA approaches using these profiles increase the business-users involvement, improve the communication process of IoT solutions, and significantly reduce the implementation time by generating most of the required code.

The Sensor-data profile produced the highest-quality instance models, obtaining a perfect Specification [126] for the PIMs and highly simple PSMs. Besides, the percentage of automatically-generated code was above 99%. Instance models of the IoT-data profile also had good quality (though not perfect) and generated 98% of the code. Nevertheless, this profile covers a broader and more complex range of applications that may have significantly reduced the overall quality. Yet, Simplicity [126] was the only metric that sensibly decreased.

Furthermore, experiments in this chapter also indicate that the IoT-data profile can (and do) support multiple devices and embedded OSes. In particular, they use the PIM, PSM and DM meta-models to represent applications and devices using RIOT OS in addition to the previous SEOS examples. Besides, they evaluate the automatically-generated code for RIOT in multiple tests on a public testbed [3]. Such tests reveal that the generated code was ready for compilation; all the selected devices executed the code; and the code allowed the devices to sense, receive, join and transform data. The experiment also evidenced that some boards had hardware-related failures sending their data in 22% of tests. Besides, slight variations in time (*e.g.* 1 second every 5 minutes) caused synchronisation anomalies in almost all tests. Future versions of the meta-models and code-generators should address these problems. Yet, implementations must also consider data quality and fault tolerance strategies.

Several parts of this chapter are published in [39, 40]. While [39] presents the CASE-tool and practical validations for the sensor-data MDA, [40] provides the theoretical and practical validations for STS4IoT.

Chapter 7

Conclusions

This thesis has presented an MDA-based approach for modelling and implementing sensor- and IoT-data applications. Two UML profiles model the prime concepts for these kinds of data. Each profile has a high-abstraction data view (PIM), an implementation view (PSM), and an abstract catalogue of available devices (DM). Besides, two python model-to-code tools define the required code to provide the modelled data with the selected devices.

To conclude this thesis, this chapter summarises its most relevant outcomes (Sec. 7.1), states the contributions (Sec. 7.2), presents the learnt lessons (Sec. 7.3) and proposes future work that could increase the possible impact of the MDA approach (Sec. 7.4).

7.1 Main Outcomes

This thesis produced four principal outcomes:

1. It defines the prime data features that describe sensor- and IoT-data applications (Sec. 2.3). Sensor-data applications are information systems based on a single device that provides data about a sensed entity (*e.g.* the environment). Therefore, a sensor-data model must define the type of sensed variable. However, these variables change in time and thus have a temporal validity. Besides, sensor devices can execute some operations on the sensed data, defining temporal windows to constrain the stream data.

   Moreover, IoT-data applications are a complex version of sensor-data applications involving multiple devices with increased communication and computation capabilities. Consequently, this kind of information system relies on the same principles. Besides, conceptual data models must also describe the advanced data transformations and associations to support the increased complexity of IoT. This thesis focuses on three basic operations: aggregation, conversion and filter. Similarly, it defines three communication aspects: data join, data association, and network representation.

2. It studies the literature about model-driven IoT using two complementary approaches: a scientometric analysis and a literature review (Chapter 3). The scientometric analysis dis-

plays the most commonly addressed topics involving data: big data and data collection. However, it highlights the low relevance of data modelling in the research area. The literature review corroborates these results. Even though some papers address some of the prime data features, their description is not exhaustive, and their main focus is implementation.

3. It provides a UML profile and MDA approach for developing sensor-data applications (Chapter 4). The UML profile effectively allows for the specification of instance models for particular applications. Besides, it has a high quality, especially its Understandability and Well-Structured properties. Moreover, its instance models are highly legible and clear, and the MDA can generate most of the implementable code in at least one sensor device.

4. It provides a UML profile and MDA approach for IoT-data applications (Chapter 5). It extends the sensor-data profile to support IoT's additional computation and communication capabilities. The UML profile maintains its high quality in the same properties, and its instance models are highly legible and clear. Moreover, this MDA can generate 100% of the implementable code in multiple particular or commercial IoT devices.

7.2 Main Contributions

The previously described outcomes have multiple (actual and expected) contributions inside and outside this thesis. The description of the prime data features provided us with a conceptual framework to assess how well the related works describe sensor and IoT data. In this way, we seamlessly verified the completeness of the IoT data models in the literature. Besides, this conceptual framework also guided most of our conceptual-modelling process and allowed us to define complete yet simple representations of sensor- and IoT-data applications. We expect these concepts to help researchers in this area to build their IoT models with a clearer view of the data.

The study of the related works provided a profound understanding of the research area, allowing us to position on a relatively unexplored subject of model-driven IoT. Besides, it helped us define theoretical and practical evaluation frameworks that validate the sensor- and IoT-data MDA approaches.

[39] and [40] contain parts of the literature review (and the prime data features). These results have already helped to conceive further research (*e.g.* [128]). We expect that researchers continue leveraging our findings to underpin and position their proposals.

The sensor-data MDA (published in [39]) provides a simple way to formalise and develop data acquisition applications based on a single device. Its high understandability and legibility allow business users to be involved in the definition of these information systems. Indeed, this MDA also defines a methodology and UML profile for sensor-based BI applications. This methodology (published in [41]) guides the interactions among three different actors that actively build such systems.

Similarly, the IoT-data MDA (published in [40]) defines a conceptual data model for complete data-gathering applications using the IoT. Besides, it introduces the concept of STS to conceive an IoT implementation as a complex data source for different systems that internally changes data. This approach combines highly abstract and readable models with a powerful code-generation tool that supports multiple platforms to ease the development and exploitation of IoT-data applications. Thus, it significantly reduces the effort to develop this kind of application while also providing high formality to the process.

Consequently, we expect researchers and engineers to use these MDAs (Sensor and IoT) in their projects. In this sense, we have already had an international collaboration to define a UML profile for polyglot-data systems using our sensor-data constructs as a base [42].

Finally, the publication of our main outcomes [39, 41, 40] has helped advance the data-related topics in model-driven IoT, positioning our institutions and countries among the top contributors to the subject. (Figure 7.1).

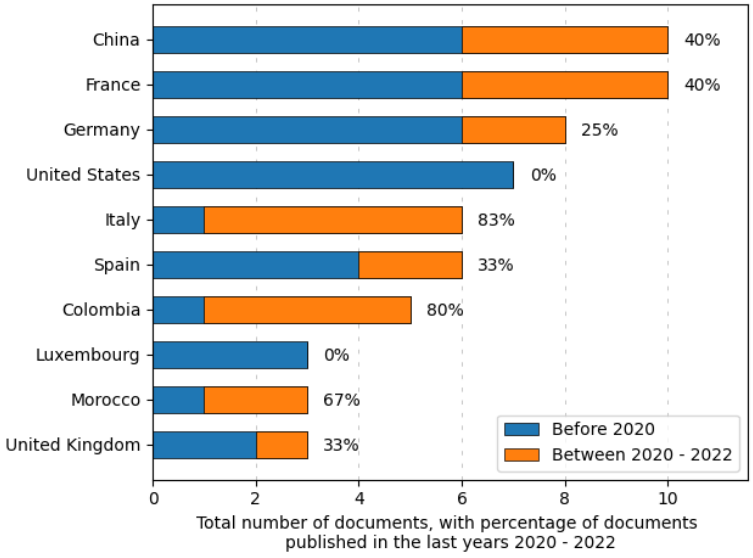

Figure 7.1: Contribution by country to the data-related topics in model-driven IoT. Colombia and France are in the top ten contributors with a significant increase since 2020.

7.3 Lessons Learnt

During the development of my thesis, I learned multiple lessons from the exercise of research and the particular investigation field. Some of the most relevant lessons are:

- Even though a good search query executed in various bibliographic databases will return most of the relevant related work, there is always a chance that some authors have used

different synonyms or databases, excluding their work from the results. To solve this issue, one could further expand the search in at least two ways: add the synonyms and databases or manually search other relevant works in Google Scholar. Both approaches have advantages and drawbacks; none of the two is perfect. In my experience, not all the bibliographic databases handle queries equally; their results often contain different information and fail to process multiple sets of logical operators. Consequently, I followed the second approach to find the most relevant related work (Chapter 3).

- In the literature review, I expected to find a pattern or standard in the representations of IoT systems. Instead, I discovered that these models vary significantly. The main application domain, the modelling focus and the modelling language notably influence the designs.

- In various papers, simple proofs of concept or theoretical assessments were enough to support authors' constructs in the conceptual models. However, the research area is becoming more strict with new proposals. Indeed, we had to provide theoretical and practical validations of our UML profiles and code-generation tools to evidence their feasibility and applicability before getting published.

- Nevertheless, the quality and performance of the automatically-generated code or the generation tools still have little relevancy. The studied papers rarely addressed this topic, and the reviewers of our publications did not pay significant attention to it. Yet, I think it will become more relevant in the future as part of more strict validation.

- STS4IoT makes it easy to change from one implementation device to another. However, enabling these changes between SEOS and RIOT was highly challenging for me. I had to learn multiple aspects of the two embedded OSes and become a middle-level user to abstract the applications' syntaxes and structures for the model-to-code tool. Even though I leveraged the same XMI parsing process, I had to adapt and add several parts of the SEOS-generating file to support the RIOT code generation.

- Even though the generated code is correct and the simulations run smoothly, the laboratory experiments in the FIT IoT-LAB showed that real applications have multiple implications that one cannot directly solve at design time. For example, various experiments in Sec. 6.3.2 suffered from synchronisation and hardware problems that are not related to the code quality but altered the delivered data.

7.4 Future Work

The current outcomes of this thesis provide a stable way to ease the development and exploitation of IoT-data applications. It also contributes to a novel research subject in model-driven IoT. However, there is always room for improvement in conceptual modelling. Besides, the theoretical and practical validations evidenced some constraints beyond the scope and resources of

this thesis. Therefore, we propose the following future work that could increase the impact and value of the proposed approach:

- A thorough revision of the two quality parameters below the median (Functionality and Extendibility) would provide valuable insights into how to increase the quality of the profiles without losing their focus. However, one must be careful not to make the metrics the goal at this point.
- Defining automatic model-to-model transformations would further increase the involvement of business users in the implementation process. Nevertheless, this process should allow for diversification in the definition of the implementation options. Otherwise, one data requirement will drive to only one implementation choice, which is far from ideal.
- Improving the architecture of the model-to-code tool with a clear distinction of the platform-specific code fragments could ease the addition of other embedded OSes. Moreover, it should also provide a modification protocol to identify the different code segments' purpose and their implementation in the known platforms.
- A thorough evaluation of the generated code performance could further validate our outcomes or help identify relevant drawbacks. This evaluation should guide a revision of the model-to-code tool to produce effective and efficient code.
- The hardware in IoT-data applications is usually exposed to harsh conditions and is thus prone to failure. Different strategies can mitigate the effects of such errors [129]. Certain levels of our MDA could support the definition of some of these fault-tolerance strategies. For example, the automatically-generated code could include time redundancy by default, or a boolean tagged value in the PSM could enable its generation. Also, IoT experts could define distributed recovery blocks or redundant nodes at the PSM level using some tags. Another alternative could be the generation of resilient code [33].
- Using a single node that synchronises the others in the IoT network or synchronised real-time clocks (RTCs) in each device could reduce the synchronisation problems. However, the PSM and the model-to-code tool would require several modifications to support these features.
- The inclusion of explicit data-quality concepts at the PSM level could highly increase the value of the MDA approach. Nevertheless, the automatic implementation of these concepts is highly complex.
- To test the sensor- and IoT-data MDA approaches with multiple users and practitioners to capture their perception of the overall usability of these approaches.

Bibliography

[1] R. Priyadarshi, S. Routroy, and G. K. Garg, "Postharvest supply chain losses: a state-of-the-art literature review and bibliometric analysis," *Journal of Advances in Management Research*, 2020.

[2] L. Santamaria-Granados, J. F. Mendoza-Moreno, and G. Ramírez-González, "Tourist recommender systems based on emotion recognition - A scientometric review," *Future Internet*, vol. 13, no. 1, p. 2, 2021.

[3] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noël, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT iot-lab: A large scale open experimental iot testbed," in *2nd IEEE World Forum on Internet of Things, WF-IoT 2015, Milan, Italy, December 14-16, 2015*, pp. 459–464, IEEE Computer Society, 2015.

[4] Future Internet Testing Facility, "FIT IoT-LAB," 2020.

[5] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "A tool-supported approach for assessing the quality of modeling artifacts," *Journal of Computer Languages*, vol. 51, pp. 173–192, 2019.

[6] E. Ahmed, I. Yaqoob, I. A. T. Hashem, I. Khan, A. I. A. Ahmed, M. Imran, and A. V. Vasilakos, "The role of big data analytics in internet of things," *Computer Networks*, vol. 129, pp. 459–471, 2017.

[7] L. Atzori, A. Iera, and G. Morabito, "Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm," *Ad Hoc Networks*, vol. 56, pp. 122 – 140, 2017.

[8] B. Omoniwa, R. Hussain, M. A. Javed, S. H. Bouk, and S. A. Malik, "Fog/edge computing-based iot (feciot): Architecture, applications, and research issues," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4118–4149, 2019.

[9] M. K. Saggi and S. Jain, "A survey towards an integration of big data analytics to big insights for value-creation," *Information Processing & Management*, vol. 54, no. 5, pp. 758–790, 2018.

[10] M. Bansal, I. Chana, and S. Clarke, "A survey on iot big data: Current status, 13 v's challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 6, pp. 131:1–131:59, 2021.

[11] C.-W. Tsai, C.-F. Lai, H.-C. Chao, and A. V. Vasilakos, "Big data analytics: a survey," *Journal of Big data*, vol. 2, no. 1, p. 21, 2015.

[12] K. Boulil, S. Bimonte, and F. Pinet, "Conceptual model for spatial data cubes: A UML profile and its automatic implementation," *Computer Standards & Interfaces*, vol. 38, pp. 113–132, 2015.

[13] S. Bimonte, O. Boussaid, M. Schneider, and F. Ruelle, "Design and implementation of active stream data warehouses," *IJDWM*, vol. 15, no. 2, pp. 1–21, 2019.

[14] J. Luo, L. Zhang, and 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)*, pp. 463–470, 2020.

[15] Y. Qin, Q. Z. Sheng, N. J. Falkner, S. Dustdar, H. Wang, and A. V. Vasilakos, "When things matter: A survey on data-centric internet of things," *Journal of Network and Computer Applications*, vol. 64, pp. 137–153, 2016.

[16] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, and R. Spalazzese, "Model-driven engineering for mission-critical iot systems," *IEEE software*, vol. 34, no. 1, pp. 46–53, 2017.

[17] P. Patel and D. Cassou, "Enabling high-level application development for the internet of things," *Journal of Systems and Software*, vol. 103, pp. 62–84, 2015.

[18] H. Grichi, O. Mosbahi, M. Khalgui, and Z. Li, "Rwin: New methodology for the development of reconfigurable wsn," *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 1, pp. 109–125, 2016.

[19] S. Dobson, M. Golfarelli, S. Graziani, and S. Rizzi, "A reference architecture and model for sensor data warehousing," *IEEE Sensors Journal*, vol. 18, no. 18, pp. 7659–7670, 2018.

[20] M. Amadeo, C. Campolo, J. Quevedo, D. Corujo, A. Molinaro, A. Iera, R. L. Aguiar, and A. V. Vasilakos, "Information-centric networking for the internet of things: challenges and opportunities," *IEEE Network*, vol. 30, no. 2, pp. 92–100, 2016.

[21] S. Teixeira, B. A. Agrizzi, J. G. P. Filho, S. Rossetto, and R. de Lima Baldam, "Modeling and automatic code generation for wireless sensor network applications using model-driven or business process approaches: A systematic mapping study," *J. Syst. Softw.*, vol. 132, pp. 50–71, 2017.

[22] G. Sebastián, J. A. Gallud, and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *J. Comput. Lang.*, vol. 56, p. 100935, 2020.

[23] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.

[24] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 5–13, 2020.

[25] Object Management Group (OMG), "Model Driven Architecture (MDA) - Guide revision 2.0," June 2014.

[26] J.-N. Mazón and J. Trujillo, "A hybrid model driven development framework for the multidimensional modeling of data warehouses!," *ACM SIGMOD Record*, vol. 38, no. 2, pp. 12–17, 2009.

[27] M. Golfarelli, S. Rizzi, and E. Turricchia, "Modern software engineering methodologies meet data warehouse design: 4wd," in *International Conference on Data Warehousing and Knowledge Discovery*, pp. 66–79, Springer, 2011.

[28] Z. El Akkaoui, E. Zimányi, J.-N. Mazón, and J. Trujillo, "A model-driven framework for etl process development," in *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*, pp. 45–52, 2011.

[29] A. Maté and J. Trujillo, "Tracing conceptual models' evolution in data warehouses by using the model driven architecture," *Computer Standards & Interfaces*, vol. 36, no. 5, pp. 831–843, 2014.

[30] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "Frasad: A framework for model-driven iot application development," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 387–392, IEEE, 2015.

[31] F. Ciccozzi and R. Spalazzese, "Mde4iot: supporting the internet of things with model-driven engineering," in *International Symposium on Intelligent and Distributed Computing*, pp. 67–76, Springer, 2016.

[32] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of uml models: a systematic review of research and practice," *Software & Systems Modeling*, vol. 18, no. 3, pp. 2313–2360, 2019.

[33] J. C. Kirchhof, B. Rumpe, D. Schmalzing, and A. Wortmann, "Montithings: Model-driven development and deployment of reliable iot applications," *Journal of Systems and Software*, vol. 183, p. 111087, 2022.

[34] R. J. Lehmann, R. Reiche, and G. Schiefer, "Future internet and the agri-food sector: State-of-the-art in literature and research," *Computers and Electronics in Agriculture*, vol. 89, pp. 158–174, 2012.

[35] J. Ramirez-Villegas, M. Salazar, A. Jarvis, and C. E. Navarro-Racines, "A way forward on adaptation to climate change in colombian agriculture: perspectives towards 2050," *Climatic change*, vol. 115, no. 3, pp. 611–628, 2012.

[36] J. E. Plazas, S. Bimonte, G. D. Sousa, and J. C. Corrales, "Data-centric UML profile for wireless sensors: Application to smart farming," *Int. J. Agric. Environ. Inf. Syst.*, vol. 10, no. 2, pp. 21–48, 2019.

[37] E. M. Ouafiq, R. Saadane, A. Chehri, and S. Jeon, "Ai-based modeling and data-driven evaluation for smart farming-oriented big data architecture using iot with energy harvesting capabilities," *Sustainable Energy Technologies and Assessments*, vol. 52, p. 102093, 2022.

[38] A. ur Rehman, A. Z. Abbasi, N. Islam, and Z. A. Shaikh, "A review of wireless sensors and networks' applications in agriculture," *Computer Standards & Interfaces*, vol. 36, no. 2, pp. 263–270, 2014.

[39] J. E. Plazas, S. Bimonte, C. de Vaulx, M. Schneider, Q. D. Nguyen, J. P. Chanet, H. Shi, K. M. Hou, and J. Carlos Corrales, "A conceptual data model and its automatic implementation for iot-based business intelligence applications," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10719–10732, 2020.

[40] J. E. Plazas, S. Bimonte, M. Schneider, C. de Vaulx, P. Battistoni, M. Sebillo, and J. C. Corrales, "Sense, transform & send for the internet of things (sts4iot): Uml profile for data-centric iot applications," *Data & Knowledge Engineering*, vol. 139, p. 101971, 2022.

[41] J. E. Plazas, S. Bimonte, M. Schneider, C. de Vaulx, and J. C. 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*, vol. 1259 of *Communications in Computer and Information Science*, pp. 84–93, Springer, 2020.

[42] A. Belhassena, S. Bimonte, P. Battistoni, C. Cariou, G. Chalhoub, J. C. Corrales, J. Laneurit, R. Moussa, J. E. Plazas, R. Wrembel, and M. Sebillo, "On modeling data for iot agroecology applications by means of a UML profile," in *MEDES '21: Proceedings of the 13th International Conference on Management of Digital EcoSystems, Virtual Event, Tunisia, November 1 - 3, 2021* (R. Chbeir, Y. Manolopoulos, L. Bellatreche, D. Benslimane, M. Ivanovic, and Z. Maamar, eds.), pp. 120–128, ACM, 2021.

[43] P. H. Picciani, F. M. Shimizu, Q. G. Olimpio, and R. C. Michel, "Sensing materials: Organic polymers," in *Reference Module in Biomedical Sciences*, Elsevier, 2021.

[44] T. Albrecht and J. B. Edel, "Introduction," in *Engineered Nanopores for Bioanalytical Applications* (J. B. Edel and T. Albrecht, eds.), Micro and Nano Technologies, pp. ix–xi, Oxford: William Andrew Publishing, 2013.

[45] K. Peters and D. Inaudi, "5 - fiber optic sensors for assessing and monitoring civil infrastructures," in *Sensor Technologies for Civil Infrastructures* (M. Wang, J. Lynch, and H. Sohn, eds.), vol. 55 of *Woodhead Publishing Series in Electronic and Optical Materials*, pp. 121–158, Woodhead Publishing, 2014.

[46] X. E. Pantazi, D. Moshou, and D. Bochtis, "Chapter 1 - sensors in agriculture," in *Intelligent Data Mining and Fusion Systems in Agriculture* (X. E. Pantazi, D. Moshou, and D. Bochtis, eds.), pp. 1–15, Academic Press, 2020.

[47] A. Duque-Torres, C. Rodriguez-Pabon, J. Ruiz-Rosero, G. Zambrano-Gonzalez, M. Almanza-Pinzon, O. M. Caicedo Rendon, and G. Ramirez-Gonzalez, "A new environmental monitoring system for silkworm incubators," *F1000Research*, vol. 7, p. 248, 2018.

[48] C. A. Gonzalez-Amarillo, J. C. Corrales-Muñoz, M. A. Mendoza-Moreno, A. M. G. Amarillo, A. F. Hussein, A. N., and G. Ramírez-González, "An iot-based traceability system for greenhouse seedling crops," *IEEE Access*, vol. 6, pp. 67528–67535, 2018.

[49] Q.-D. Nguyen, C. Roussey, M. Poveda-Villalón, C. de Vaulx, and J.-P. Chanet, "Development experience of a context-aware system for smart irrigation using caso and irrig ontologies," *Applied Sciences*, vol. 10, no. 5, 2020.

[50] S. Bimonte, E. Naoufal, and L. Gineste, "A system for the rapid design and implementation of personalized agricultural key performance indicators issued from sensor data," *Comput. Electron. Agric.*, vol. 130, pp. 1–12, 2016.

[51] R. Pérez-Castillo, A. G. Carretero, I. Caballero, M. Rodríguez, M. Piattini, A. Mate, S. Kim, and D. Lee, "DAQUA-MASS: an ISO 8000-61 based data quality management methodology for sensor data," *Sensors*, vol. 18, no. 9, p. 3105, 2018.

[52] S. Cai, B. Gallina, D. Nyström, and C. Seceleanu, "Data aggregation processes: a survey, a taxonomy, and design guidelines," *Computing*, vol. 101, no. 10, pp. 1397–1429, 2019.

[53] S. Davidson and B. D. Shriver, "An overview of firmware engineering," *Computer*, vol. 11, no. 5, pp. 21–33, 1978.

[54] Arduino, "Language Reference," 2022.

[55] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an open source operating system for low-end embedded devices in the iot," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4428–4440, 2018.

[56] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet Things J.*, vol. 6, no. 6, pp. 10375–10383, 2019.

[57] C. G. García, D. Meana-Llorián, V. García-Díaz, A. C. Jiménez, and J. P. Anzola, "Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering," *IEEE Access*, vol. 8, pp. 141872–141894, 2020.

[58] C. M. de Farias, I. C. Brito, L. Pirmez, F. C. Delicato, P. F. Pires, T. C. Rodrigues, I. L. dos Santos, L. F. Carmo, and T. Batista, "Comfit: A development environment for the internet of things," *Future Generation Computer Systems*, vol. 75, pp. 128 – 144, 2017.

[59] F. Javed, M. K. Afzal, M. Sharif, and B. Kim, "Internet of things (iot) operating systems support, networking technologies, applications, and challenges: A comparative review," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 3, pp. 2062–2100, 2018.

[60] B. Khalyly, A. Belangour, A. Erraissi, and M. Banane, "Devops and microservices based internet of things meta-model," *International Journal of Emerging Trends in Engineering Research*, vol. 8, no. 9, pp. 6254–6266, 2020.

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

[62] D. C. Corrales, A. Figueroa, A. Ledezma, and J. C. Corrales, "An empirical multi-classifier for coffee rust detection in colombian crops," in *Computational Science and Its Applications - ICCSA 2015 - 15th International Conference, Banff, AB, Canada, June 22-25, 2015, Proceedings, Part I* (O. Gervasi, B. Murgante, S. Misra, M. L. Gavrilova, A. M. A. C. Rocha, C. M. Torre, D. Taniar, and B. O. Apduhan, eds.), vol. 9155 of *Lecture Notes in Computer Science*, pp. 60–74, Springer, 2015.

[63] J. E. Plazas, J. S. Rojas, D. C. Corrales, and J. C. Corrales, "Validation of coffee rust warnings based on complex event processing," in *Computational Science and Its Applications - ICCSA 2016 - 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part IV* (O. Gervasi, B. Murgante, S. Misra, A. M. A. C. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. N. Stankova, and S. Wang, eds.), vol. 9789 of *Lecture Notes in Computer Science*, pp. 684–699, Springer, 2016.

[64] N. Wang, N. Zhang, and M. Wang, "Wireless sensors in agriculture and food industry—recent development and future perspective," *Computers and Electronics in Agriculture*, vol. 50, no. 1, pp. 1–14, 2006.

[65] D. Sharma, S. Verma, and K. Sharma, "Network topologies in wireless sensor networks: a review 1," *International Journal of Electronics & Communication Technology*, vol. 4, no. spl-3, pp. 93–97, 2013.

[66] A. W. Brown, "Model driven architecture: Principles and practice," *Softw. Syst. Model.*, vol. 3, no. 4, pp. 314–327, 2004.

[67] S. Bimonte, M. Schneider, and O. Boussaid, "Business intelligence indicators: Types, models and implementation," *Int. Journal of Data Warehousing and Mining (IJDWM)*, vol. 12, no. 4, pp. 75–98, 2016.

[68] V. C. Storey and I. Song, "Big data technologies and management: What conceptual modeling can do," *Data Knowl. Eng.*, vol. 108, pp. 50–67, 2017.

[69] S. Luján-Mora, J. Trujillo, and I. Song, "A UML profile for multidimensional modeling in data warehouses," *Data Knowl. Eng.*, vol. 59, no. 3, pp. 725–769, 2006.

[70] OMG, "About the Unified Modeling Language Specification Version 2.4," Mar. 2011.

[71] R. S. Wazlawick, "Chapter 3 - high-level requirements," in *Object-Oriented Analysis and Design for Information Systems* (R. S. Wazlawick, ed.), pp. 29–57, Boston: Morgan Kaufmann, 2014.

[72] S. Gupta, "Non-functional requirements elicitation for edge computing," *Internet of Things*, vol. 18, p. 100503, 2022.

[73] R. Kimball and M. Ross, *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[74] A. Erraissi and A. Belangour, "Data sources and ingestion big data layers: meta-modeling of key concepts and features," *International Journal of Engineering & Technology*, vol. 7, no. 4, pp. 3607–3612, 2018.

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

[76] Arvalis, Limagrain, and 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, May 2005.

[77] H.-Y. Zhou, D.-Y. Luo, Y. Gao, and D.-C. Zuo, "Modeling of node energy consumption for wireless sensor networks," *Wireless Sensor Network*, vol. 3, no. 1, p. 18, 2011.

[78] I. Jacobson, G. Booch, and J. E. Rumbaugh, *The unified software development process - the complete guide to the unified process from the original designers*. Addison-Wesley object technology series, Addison-Wesley, 1999.

[79] M. Ashouri, P. Davidsson, and R. 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*, pp. 155–162, IEEE, 2018.

[80] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo, "Join operations in temporal databases," *The VLDB Journal*, vol. 14, no. 1, pp. 2–29, 2005.

[81] M. Serrano, J. Trujillo, C. Calero, and M. Piattini, "Metrics for data warehouse conceptual models understandability," *Information and Software Technology*, vol. 49, no. 8, pp. 851–870, 2007.

[82] J. Trujillo and S. Luján-Mora, "A uml based approach for modeling etl processes in data warehouses," in *Conceptual Modeling - ER 2003* (I.-Y. Song, S. W. Liddle, T.-W. Ling, and P. Scheuermann, eds.), (Berlin, Heidelberg), pp. 307–320, Springer Berlin Heidelberg, 2003.

[83] J. Ruiz-Rosero, G. Ramírez-González, and J. Viveros-Delgado, "Software survey: Scientopy, a scientometric tool for topics trend analysis in scientific publications," *Scientometrics*, vol. 121, no. 2, pp. 1165–1188, 2019.

[84] A. Perianes-Rodríguez, L. Waltman, and N. J. van Eck, "Constructing bibliometric networks: A comparison between full and fractional counting," *J. Informetrics*, vol. 10, no. 4, pp. 1178–1195, 2016.

[85] NLTK Team, "NLTK :: Natural Language Toolkit v 3.7," Mar. 2022.

[86] S. Wolny, A. Mazak, and B. Wally, "An initial mapping study on mde4iot," in *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018* (R. Hebig and T. Berger, eds.), vol. 2245 of *CEUR Workshop Proceedings*, pp. 524–529, CEUR-WS.org, 2018.

[87] F. Essaadi, Y. B. Maissa, and M. Dahchour, "Mde-based languages for wireless sensor networks modeling: A systematic mapping study," in *Advances in Ubiquitous Networking 2 - Proceedings of the UNet'16, Casablanca, Morocco, May 30 - June 1, 2016* (R. E. Azouzi, D. S. Menasché, E. Sabir, F. D. Pellegrini, and M. Benjillali, eds.), vol. 397 of *Lecture Notes in Electrical Engineering*, pp. 331–346, 2016.

[88] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: an extended systematic mapping study," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 67–94, 2020.

[89] M. A. Mohamed, G. Kardas, and M. Challenger, "Model-driven engineering tools and languages for cyber-physical systems-a systematic literature review," *IEEE Access*, vol. 9, pp. 48605–48630, 2021.

[90] F. Ihirwe, D. D. Ruscio, S. Mazzini, P. Pierini, and A. Pierantonio, "Low-code engineering for internet of things: a state of research," in *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings* (E. Guerra and L. Iovino, eds.), pp. 74:1–74:8, ACM, 2020.

[91] N. Idoudi, C. Duvallet, B. Sadeg, R. Bouaziz, and F. Gargouri, "Structural model of real-time databases," in *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume ISAS-2, Barcelona, Spain, June 12-16, 2008* (J. Cordeiro and J. Filipe, eds.), pp. 319–324, 2008.

[92] N. Jiang and Z. Chen, "Model-driven data cleaning for signal processing system in sensor networks," in *2010 2nd International Conference on Signal Processing Systems*, vol. 1, pp. V1–237, IEEE, 2010.

[93] N. Lumineau, F. Laforest, Y. Gripay, and J. Petit, "Extending conceptual data model for dynamic environment," in *Conceptual Modeling - 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings* (P. Atzeni, D. W. Cheung, and S. Ram, eds.), vol. 7532 of *Lecture Notes in Computer Science*, pp. 242–251, Springer, 2012.

[94] D. Philipp, J. Stachowiak, F. Dürr, and K. Rothermel, "Model-driven public sensing in sparse networks," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services - 10th International Conference, MOBIQUITOUS 2013, Tokyo, Japan, December 2-4, 2013, Revised Selected Papers* (I. Stojmenovic, Z. Cheng, and S. Guo, eds.), vol. 131 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 17–29, Springer, 2013.

[95] A. Hussain and W. Wu, "Sustainable interoperability and data integration for the iot-based information systems," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Exeter, United Kingdom, June 21-23, 2017* (Y. Wu, G. Min, N. Georgalas, A. Al-Dubi, X. Jin, L. T. Yang, J. Ma, and P. Yang, eds.), pp. 824–829, IEEE Computer Society, 2017.

[96] H. Marouane, C. Duvallet, A. Makni, R. Bouaziz, and B. Sadeg, "An uml profile for representing real-time design patterns," *Journal of King Saud University-Computer and Information Sciences*, vol. 30, no. 4, pp. 478–497, 2018.

[97] R. Lamberti and L. Stojanovic, "Complex event processing as an approach for real-time analytics in industrial environments," in *17th IEEE International Conference on Industrial Informatics, INDIN 2019, Helsinki, Finland, July 22-25, 2019*, pp. 220–225, IEEE, 2019.

[98] G. Fuchs and R. German, "Uml2 activity diagram based programming of wireless sensor networks," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pp. 8–13, 2010.

[99] K. Tei, R. Shimizu, Y. Fukazawa, and S. Honiden, "Model-driven-development-based stepwise software development process for wireless sensor networks," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 4, pp. 675–687, 2014.

[100] K. Thramboulidis and F. Christoulakis, "Uml4iot - A uml-based approach to exploit iot in cyber-physical manufacturing systems," *Comput. Ind.*, vol. 82, pp. 259–272, 2016.

[101] H. Marouane, A. Makni, R. Bouaziz, C. Duvallet, and B. Sadeg, "Definition of design patterns for advanced driver assistance systems," in *Proceedings of the 10th Travelling Conference on Pattern Languages of Programs*, p. 3, ACM, 2016.

[102] P. Grace, B. Pickering, and M. Surridge, "Model-driven interoperability: engineering heterogeneous iot systems," *Annals of Telecommunications*, vol. 71, no. 3-4, pp. 141–150, 2016.

[103] H. Cai, Y. Gu, A. V. Vasilakos, B. Xu, and J. Zhou, "Model-driven development patterns for mobile services in cloud of things," *IEEE Transactions on Cloud Computing*, vol. 6, no. 3, pp. 771–784, 2016.

[104] B. Morin, N. Harrand, and F. Fleurey, "Model-based software engineering to tame the iot jungle," *IEEE Softw.*, vol. 34, no. 1, pp. 30–36, 2017.

[105] T. Rodrigues, F. C. Delicato, T. V. Batista, P. F. Pires, and L. Pirmez, "An approach based on the domain perspective to develop WSAN applications," *Software and Systems Modeling*, vol. 16, no. 4, pp. 949–977, 2017.

[106] C. M. Sosa-Reyna, E. Tello-Leal, and D. L. Alabazares, "Methodology for the model-driven development of service oriented iot applications," *J. Syst. Archit.*, vol. 90, pp. 15–22, 2018.

[107] A. Kifouche, R. Hamouche, R. Kocik, A. Rachedi, and G. Baudoin, "Model driven framework to enhance sensor network design cycle," *Trans. Emerg. Telecommun. Technol.*, vol. 30, no. 8, 2019.

[108] J. Schobel, T. Probst, M. Reichert, M. Schickler, and R. Pryss, "Enabling sophisticated lifecycle support for mobile healthcare data collection applications," *IEEE Access*, vol. 7, pp. 61204–61217, 2019.

[109] T. Fitz, M. Theiler, and K. Smarsly, "A metamodel for cyber-physical systems," *Advanced Engineering Informatics*, vol. 41, p. 100930, 2019.

[110] B. Costa, P. Pires, and F. Delicato, "Towards the adoption of omg standards in the development of soa-based iot systems," *Journal of Systems and Software*, vol. 169, 2020.

[111] D. Alulema, J. Criado, L. Iribarne, A. Fernández-García, and R. Ayala, "A model-driven engineering approach for the service integration of iot systems," *Cluster Computing*, vol. 23, no. 3, pp. 1937–1954, 2020.

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

[113] J. Novacek, A. Kuhlwein, S. Reiter, A. Viehl, O. Bringmann, and W. Rosenstiel, "Lemons: Leveraging model-based techniques to enable non-intrusive semantic enrichment in wireless sensor networks," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 561–568, 2020.

[114] T. Nepomuceno, T. Carneiro, P. Maia, M. Adnan, T. Nepomuceno, and A. Martin, "Autoiot: A framework based on user-driven mde for generating iot applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 719–728, 2020.

[115] J. Parri, F. Patara, S. Sampietro, and E. Vicario, "A framework for model-driven engineering of resilient software-controlled systems," *Computing*, vol. 103, no. 4, pp. 589–612, 2021.

[116] A. Moin, M. Challenger, A. Badii, and S. Günnemann, "A model-driven approach to machine learning and software modeling for the iot," *Softw. Syst. Model.*, vol. 21, no. 3, pp. 987–1014, 2022.

[117] J. A. Barriga, P. J. Clemente, J. Hernández, and M. Á. P. Toledano, "Simulateiot-fiware: Domain specific language to design, code generation and execute iot simulation environments on FIWARE," *IEEE Access*, vol. 10, pp. 7800–7822, 2022.

[118] S. Bimonte, É. Edoh-Alove, H. Nazih, M.-A. Kang, and S. Rizzi, "Protolap: rapid olap prototyping with on-demand data supply," in *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*, pp. 61–66, 2013.

[119] C. Ordonez and L. Bellatreche, "Enhancing er diagrams to view data transformations computed with queries.," in *DOLAP*, 2019.

[120] J. Trujillo and S. Luján-Mora, "A uml based approach for modeling etl processes in data warehouses," in *International Conference on Conceptual Modeling*, pp. 307–320, Springer, 2003.

[121] Fredrik Lundh, "The ElementTree XML API," Feb. 2017. original-date: 2017-02-10T19:23:51Z.

[122] "Basic date and time types," Feb. 2017. original-date: 2017-02-10T19:23:51Z.

[123] H. Marouane, C. Duvallet, A. Makni, R. Bouaziz, and B. Sadeg, "An uml profile for representing real-time design patterns," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, pp. 478–497, 2018.

[124] Z. Ma, X. He, and C. Liu, "Assessing the quality of metamodels," *Frontiers of Computer Science*, vol. 7, no. 4, pp. 558–570, 2013.

[125] No Magic, Inc., *MagicDraw User Manual*. No Magic, Incorporated, a Dassault Systèmes company, 19.0 ed., 2018.

[126] S. S.-S. Cherfi, J. Akoka, and I. Comyn-Wattiau, "Conceptual modeling quality-from eer to uml schemas evaluation," in *International Conference on Conceptual Modeling*, pp. 414–428, Springer, 2002.

[127] Pandas development team, "API reference — pandas 1.4.2 documentation," 2022.

[128] S. Bimonte, A. Belhassena, C. Cariou, J. Laneurit, R. Moussa, G. Chalhoub, R. Wrembel, G. Picard, L. Bellatreche, A. Journaux, T. Heirman, A. Hassan, S. Rizzi, and J. Georgé, "On designing and implementing agro-ecology iot applications: Issues from applied research projects," in *25th International Enterprise Distributed Object Computing Workshop, EDOC Workshop 2021, Gold Coast, Australia, October 25-29, 2021*, pp. 204–209, IEEE, 2021.

[129] M. T. Moghaddam and H. Muccini, "Fault-tolerant iot - A systematic mapping study," in *Software Engineering for Resilient Systems - 11th International Workshop, SERENE 2019, Naples, Italy, September 17, 2019, Proceedings* (R. Calinescu and F. D. Giandomenico, eds.), vol. 11732 of *Lecture Notes in Computer Science*, pp. 67–84, Springer, 2019.

[130] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *Conf. on Innovative Data Systems Research, CIDR*, www.cidrdb.org, 2021.

[131] C. Sørensen and D. Bochtis, "Conceptual model of fleet management in agriculture," *Biosystems Engineering*, vol. 105, no. 1, pp. 41 – 50, 2010.

[132] A. Souza, "Lambda architecture — how to build a big data pipeline," 2019. DZone.

Appendix A

UML Profile for the Design of Polyglot-Data Applications

Sensor and IoT data are not only persistent but also transient. These data arrive into a system in the form of streams and are processed in real-time by streaming analytics applications and Complex Event Processing (CEP) applications [6], to monitor and discover trends as well as to detect anomalies. Next, these data are typically stored in a repository - a *data warehouse*, *data lake*, or *lakehouse* [11, 130] for analysing them offline through OLAP applications. Indeed, IoT real-time data are often combined with offline data to provide more advanced analytics [6, 11]. We refer to data generated and handled in such a complex system as *polyglot data*.

Polyglot-data applications are characterised by a geographically distributed deployment of devices, and a network communication continuum over different layers (from the edge to the cloud) [8]. Therefore, Quality of Service (QoS) features play a significant role in polyglot data architectures, especially in the agricultural field of application, which is usually characterised by low quality communication networks. QoS can reflect some functional requirements, such as latency, which leads to a particular placement of data and computation over the different layers. For example, in the context of hard real-time applications, data and computation can be deployed at the edge level to improve performance.

Conceptual design of information systems has several advantages [68]. First, it allows to keep away implementation details and allows decision-makers and IS to exclusively focus application content and functionalities. Second, it provides a formal and non-ambiguous support used by decision-makers to validate their requirements. Third, it streamlines the implementation phase providing some technical guidelines (and sometimes also an automatic implementation). Although the conceptual design of data for polyglot-data applications is crucial for their successful implementations, this topic has not been intensively researched yet [39].

Motivated by this lack of comprehensive solutions and by the formal support for data conceptual models provided by UML profiles, we proposed a UML profile for the data-centric design of polyglot-data applications with international collaborators [42].

This appendix presents first the UML profile for polyglot-data applications (Sec. A.1). Then,

it presents an implementation example on smart agriculture systems as proof-of-concept (Sec. A.2). The contents of this appendix are published in [42].

A.1 UML profile for polyglot-data applications

In this section, we present our UML profile for data-centric agroecology IoT applications. The profile provides a graphical and formal notation for functional requirements (in terms of data). The profile provides a generic extension mechanism for customising UML models for particular domains and platforms. It is defined using stereotypes, Tag definitions, and constraints applied to specific model elements, like Classes, Attributes, or Operations. We opt for an extension of UML elements of class diagrams since they are the de-facto standard to represent data. Figure A.1 shows the meta-model of our UML profile. Further in this section, we first present a data model, then we explain how to combine the model elements into a unique coherent model. We have implemented our UML profile in the commercial CASE tool MagicDraw.

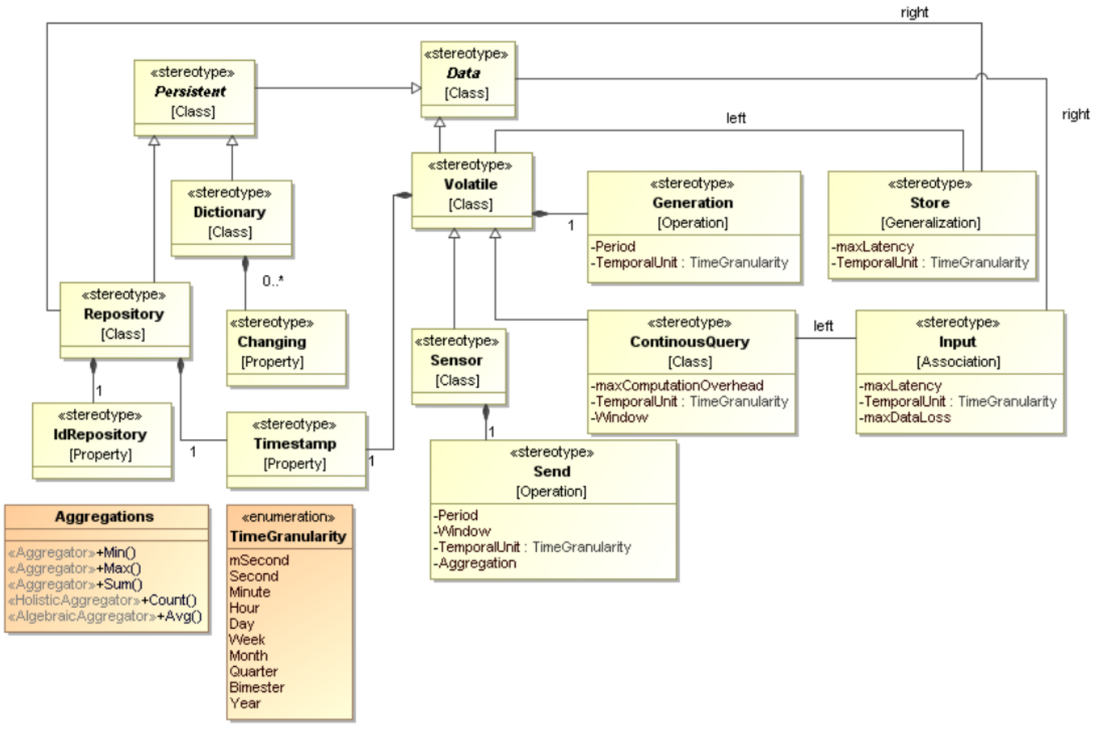

Figure A.1: The meta-model of our UML profile.

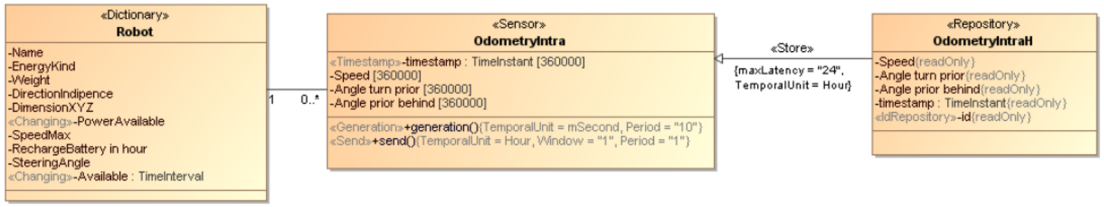

Figure A.2: Examples of *Dictionary*, *Sensor*, *Repository*, and *Store* association examples.

A.1.1 Data

Data are classified into two main groups: `Persistent` and `Volatile`, which are Class stereotypes. The `Persistent` stereotype is further specialised into `Dictionary` and `Repository`.

`Dictionary` represents transactional data (*i.e.* standard data) that can be deleted, updated, and inserted in an On-Line Transaction Processing (OLTP) system. `Dictionary` can include some attributes stereotyped as `Changing`. This stereotype means that attribute values can be updated, contrary to other attributes whose values do not change. The following associated Object Constraint Language (OCL) rule states that the attribute must not be changed (`isReadOnly=true`). Moreover, the `Dictionary` class must provide some attributes that uniquely identify its instances. This constraint is represented using the following OCL on such attributes: `isUnique=true`.

An example is shown in Figure A.2, where the `Dictionary` stereotype is applied to *Robot*. This class presents (1) some standard attributes (*e.g.*, *Name*, *SpeedMax*, *Weight*), and (2) some `Changing` attributes, like *Available*, which indicates when a robot is available for a particular task or is booked for another task within a given time slot.

`Repository` represents read-only historical data with the following characteristics:

- Attributes of the `Repository` class cannot be updated; only new values can be inserted. This constraint has been defined with OCL in the following way: `self.ownedAttribute->`
  `select(m|m.isReadOnly=false)->size()=0`.

- An instance of the `Repository` class cannot be deleted; it can only be inserted.

Moreover, `Repository` includes one attribute with stereotype `IdRepository` that uniquely identifies a datum in the collection of historical data (OCL: `ownedAttribute->select(m|m.oclIsTypeOf(IdRepository))->size()=1`). Finally, to model the temporality of the historical data represented by `Repository`, a `Timestamp` stereotype attribute is added, with an OCL constraint that forces it to have the `TimeInstant` type (OCL: `type.name='TimeInstant'`). Thus, `Repository` data represents historical data used

for analytical purposes, such as OLAP or Machine Learning applications. An example is shown in Figure A.2, where *OdometryIntraH* represents odometry historical data of robots.

`Volatile` represents data producers. These data are not permanently stored, and are characterised by a frequency generation represented by an operation with the `Generation` stereotype. `Generation` has two Tags:

- `Period` that represents a temporal generation frequency, *e.g.*, every second. In case of data generated on-demand, `Period` also accepts the *onDemand* value.
- `TemporalUnit` is the temporal granularity of `Period`. It takes values from enumeration `TimeGranularitiy`, *e.g.*, second, minute, hour. This enumeration can be easily extended with other temporal types.

Moreover, `Volatile` also has one `Timestamp` attribute representing the time of the data generation.

`Volatile` class is specialised into another class, called `Sensor`, which represents volatile data that are generated by physical sensors. `Sensor` extends `Volatile` with the `Send` operation, which represents the logic used for sending the data. It has the same Tags of `Generation` (`Period`, `TemporalUnit`), and the following additional ones:

- `Window` represents a temporal window used to collect and process data before being aggregated and sent.
- `Aggregation` represents the aggregation function used on the collected data in the window before being sent. It takes values from enumeration `Aggregations`, *e.g.*, Sum, Avg, Count (other aggregation functions can extend this enumeration).

It is crucial to specify these particular data sources at the design time since sensors must send data through a communication network, which can have substancial impacts on the system implementation.

An example of `Volatile` data is class *Demand* (Figure A.3-B). It represents the activity requests of working tasks performed by a farmer. The instances of this class are generated on-demand (Tag `Period=onDemand`.)

An example of sensor data is shown in Figure A.3-A. It represents meteo data acquired by a sensor. Data (*wind, rain, temperature*, etc.) are collected each minute (`Period=1` and `TemporalUnit=minute`). Then, averages in a moving 10-minutes window are calculated.

Commonly, a continuous query is executed over a data stream continuously. In our UML profile, a continuous query is represented by stereotype `ContinuousQuery`. It extends `Volatile` with:

- `ComputationOverhead` Tag, which represents the maximum time to compute the query,
- `Input` directed association, which represents the input data used for the query. `Input` has two Tags: `maxLatency` and `maxDataLoss`. `maxLa- tency` represents the

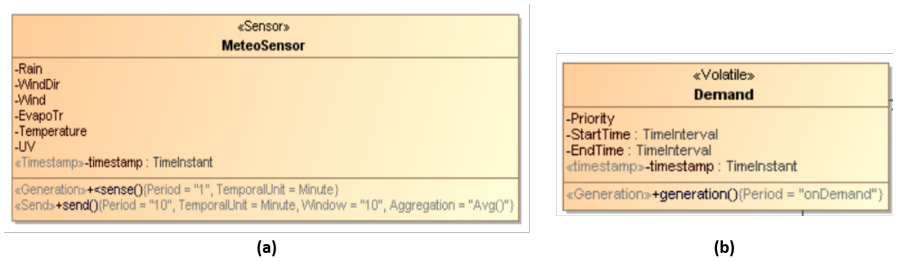

Figure A.3: Example instances of *Sensor* (**a**), and *Volatile* (**b**).

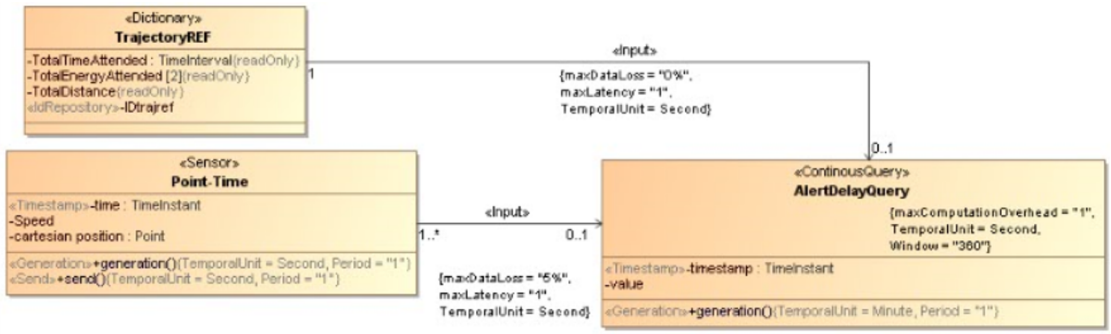

Figure A.4: An example of stereotype *ContinuousQuery*.

maximum tolerated time for input data to be transmitted to the system that implements `ContinuousQuery`. `maxDataLoss` represents the percentage of data that can be lost. These QoS constraints are issued from the application logic and come from the fact that data are generated in different points of a network, as described in the IoT architecture. Other network performance constraints exist; yet, they correspond to non-functional requirements (NFR), not the application logic. For instance, bandwidth is associated with a particular implementation of attribute data types (in terms of bytes used). Such NFR constraints should be represented at the Platform-Specific Model level following the Model-Driven Architecture, while our UML profile would correspond to the Platform Independent Model level.

Figure A.4 shows an example of `ContinuousQuery`. *AlertDelayQuery* computes in real-time the delay of a robot according to its predefined trajectory. It takes as inputs: *Point-Time*, which represents the real time position of the robot, and *TrajectoryRef*, which represents the planned trajectory. The Tag of the *Input* association states that these GPS data must be received in real-time for the alert delay computation. Moreover, *AlertDelayQuery* is computed each minute using the last 5 minutes of received data, and 5% of GPS data can be lost, contrary to *TrajectoryRef* that cannot be affected by data loss (*i.e.* all data of the trajectory of reference must be present). End-users define the configuration of *AlertDelayQuery* parameters.

Other kinds of queries can be implemented, *e.g.*, OLAP or OLTP. We refer the reader to the work of [67] for those approaches that are compatible with our proposal.

A.1.2 Associations

This section describes how `Volatile` and `Persistent` data can be transparently associated to define a single coherent model.

From Figure A.4, it clearly can be noticed that any data can be associated with `ContinousQuery` via `Input association`. Moreover, `Volatile`, `Sensors`, and `ContinousQuery` can be associated with `Repository` data via the `Store` association. This association means that initially volatile data are made persistent by using the `Repository` class. As for the `Input` association, it has a `maxLatency` Tag. This value represents a maximum time within which data must be stored in a repository. Since volatile data could be sent through a communication network they cannot be stored immediately, thus we use `maxLatency` value.

For example, Figure A.2 shows that data collected by *OdometryIntra* into the robots (odometry data collected 100 times per second, and send every hour) are stored into the `Repository` class *OdometryIntraH*. The `Store maxLatency` value is 24 hours since these data are stored in a data warehouse refreshed every 24h.

The association between `Store` and `Repository` is a generalisation, because `Repository` class must include in its structure all the attributes and associations of classes `Volatile`, `Sensor`, and `ContinuousQuery`. Moreover, `Repository` must not present methods of `Volalite` (OCL: `ownedOperation->size()=0`). Therefore, the `Store` association represents a total cloning operation of the `Volatile`, `Sensor`, and `ContinuousQuery` data in persistent storage.

Let us consider the example of Figure A.2 again. If a persistent storage stored only the values of the odometry attributes, such data would be incomplete. Note that *OdometryIntra* is associated with `Robot`. Without the associated robot that generated these data, it would not be possible to identify the robot that has generated such odometry data.

To conclude, this data-centric representation of all kinds of data and queries allow us to associate all these data among them without considering if the data is classical data, or sensor data or stream data or data resulting from computations.

A.2 Implementation Proof of Concept

This section introduces a case study based on the French ISITE CAP2025 *Superob* project (Sec. A.2.1). Then, it presents how each type of data of our agricultural case study is implemented (Sec. A.2.2), and discusses its corresponding implementation architecture (Sec. A.2.3).

A.2.1 Case Study

This case study is based on the French ISITE CAP2025 *Superob* project. The overall goal of the project is to develop and deploy an architecture for scheduling and monitoring field works of autonomous mobile robots used in agroecology practices. Autonomous agricultural robots represent an innovative solution for agroecology since they allow precise technical tasks and reduce environmental impacts. Indeed, Farms are more frequently equipped with physical sensors [75] to acquire meteorological data such as rain, temperature, or soil moisture from the fields. Furthermore, autonomous robots are applied to handle technical operations, such as plowing [131].

As a business-like example of the *Superob* project, let us consider a scenario where a farmer needs to supervise the activities of some robots in a field. To this end, real-time monitoring data are necessary. Therefore, different data must be handled in such kind of system. We distinguish three basic categories of data, namely: real-time, historical, and standard.

In this scenario, *Real-time streaming data* include, among others:

- *trajectories of robots* are necessary to: verify if a robot follows a scheduled trajectory, track the work in progress, and re-scheduling future tasks when necessary;
- *meteorological data* (e.g., rain and wind data) are necessary to check whether a given robot task can be done, e.g., some tasks such as spraying cannot be run when the wind is too strong;
- *odometry robot data* (i.e., mechanical robot data) are necessary to determine whether robots are experiencing any mechanical problem;
- *scheduling data* (i.e., demands from farmers) are necessary to define the organisation of robots' tasks.

Historical data are crucial for decision-making. Analytical queries analyse such data. For example, historical data corresponding to the same robot on the same field and its technical operations allow to compare the current work to the past ones, to decide if the robot has an abnormal behaviour.

Finally, *standard data* are needed to complement real-time and historical data. Examples of standard data include: lists of plots and fields (with their geometries and basic data), as well as robots characteristics. These data represent the contextual information associated with other data and play the role of dictionary data.

Decision-support applications usually provide *computations* over data to calculate new indicators. In our scenario, we compute continuous queries on real-time data to create meteorological, robots-fault, and delay alerts. These alerts must also be stored since, as described above, they are useful for the decision-making process.

The aforementioned scenario allows us to state that *agroecology IoT applications must cope with: (1) complex spatio-temporal data (such as robots trajectory), (2) stream data (such as meteo and robots, generated in real-time data), and (3) historical data*.

A.2.2 Data implementation

The implementation of our case study requires a complex digital eco-system. Despite a uniform representation of data at the conceptual level, the different kinds of data must be generated and handled by diverse subsystems. In particular, `Sensor` and `ContinousQuery` data require an implementation using programming languages and Data Stream Management System. `Persistent` data either in a classical storage systems (such as a relational database) or in a novel storage system (such as NoSQL systems when scalability is needed) can be deployed.

In our case study, the digital ecosystem is composed of:

- Meteorological data (`Sensor` *MeteoSensor* in Figure A.3-A) are implemented in an IoT device *uSu EDU Board*. Applications for this platform are written in C and must specify the behaviour of each involved device.
- Odometry data (*OdometryIntra* in Figure A.2) are implemented in Python in the *Fleet of Robots* using Robot Operating System (ROS). Besides, robots have tasks, trajectories, and timing constraints (*e.g.*, indicated speed).
- `Persistent` data (*e.g.*, *OdometryIntaH* in Figure A.2) are stored in the relational spatial DBMS PostGIS. These data are further loaded into a *Data Warehouse* implemented in Mondrian and JRubik to analyse them.
- The *AlertDelayQuery* continuous query (Figure A.4) is implemented in Scala (the Sedona framework, which is a spatial extension of Spark Streaming). This query joins GPS data coming from the robots (`Sensor` *Point-Time*) with data stored in PostGIS (`Dictionary` *TrajectoryREF*).

Moreover, associations between `Dictionary` and `Repository` can be simply implemented using foreign keys in a relational DBMS or integrity constraints in a NoSQL DBMS. However, such mechanisms do not exist for `Volatile` data (`Sensor` and `ContinousQuery`). They require an ad-hoc method: the data structure sent by sensors must include the identifiers of the associated `dictionary` data. This way, a software application receiving such sensed data can also identify and process the `dictionary` data.

In our case study, `sensor` data (*OdometryIntra* and *MeteoSensor*) are associated with `dictionary` data (*Robot* and *Plot*, respectively). Therefore, *OdometryIntra* sensors also send the name of the table storing robot data and its name (*i.e.* its identifier). At the same time, *MeteoSensor* sensors send also the plot name and its identifying code.

A.2.3 Multi-layer network architecture

Inspired by the Lambda reference architecture for IoT applications [132], we propose an architecture to host the technologies that handle the data required in our case study. It is composed of three main layers: *Field*, *Farm*, and *Cloud* (Figure A.5). The *Field* and *Farm* layers are implemented in each farm. In contrast, the *Cloud* layer is implemented only once for any number of farms.

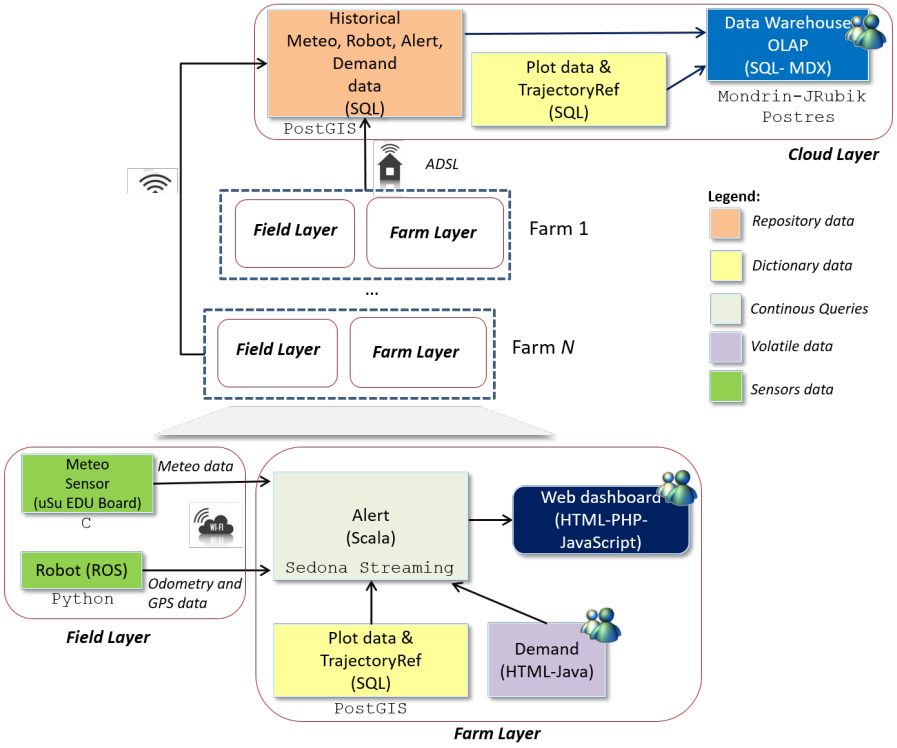

Figure A.5: Architecture implementation - data mapping.

The *Field* layer represents different data sources deployed in the field (*i.e.* `Sensor` data). In our case study, it is composed of *MeteoSensor*s and *Robot*s, which provide data and execute specific tasks. These IoT devices do not have direct Internet access on the fields (there is no cellular network coverage). Thus, we deploy a Wi-Fi network to collect these data at the *Farm* layer, which connects to the Internet through ADSL.

However, ADSL may not guarantee the QoS required by the `ContinuousQuery` of our case study. For instance, the `Input` association between *AlertDelayQuery* and *TrajectoryREF* (Figure A.4) requires a maximum latency of 1 second with no data lost. Therefore, we implement these queries in the *Farm* layer. Moreover, we host a DBMS for *TrajectoryREF* in the same layer (*Farm*) since this allows for a local connection to the DBMS with less latency and data loss issues. Finally, the *Farm* layer only sends processed data streams to the *Cloud* layer.

The *Cloud* layer implements the storage of `Repository` and `Dictionary` data coming from all the farms. This layer hosts a data warehouse that analyses the historical odometry data of the robots, providing an inter-farm analytical vision of all the available data.

To conclude, our case study evidences how the different kinds of data supported by our UML profile can be used to describe all the data used by this smart-agriculture polyglot-data application. Moreover, it shows how these data can be implemented using several languages, systems

and infrastructures that successfully communicate and cooperate to provide useful decision support. Finally, the latency and the data loss features can be used to provide some guidelines in the implementation of the IoT architecture and the distribution of the data along the different network layers.

Appendix B

Results from the Multiple-Devices Test Using RIOT

This annex presents the complete PSM models and results of the nine tests performed in the FIT IoT-LAB for the RIOT support experiment (Sec. 6.3). Sec. B.1 presents the PSM and the three tests using the IoT-LAB M3 node as Sink Node (SN). Sec. B.2 presents the PSM and the three tests using the Microchip SAMR21 Xplained Pro node as SN. Finally, Sec. B.3 presents the PSM and the three tests using the Arduino Zero node as SN.

B.1 Tests Results Using an IoT-LAB M3 as SN

Figure B.1 shows the PSM for these tests. Table B.1 shows the results for the 30-minutes test with this PSM. Table B.2 shows the results for the 1-hour test, and Table B.3 the 1-day test results.

Table B.1: Comparison matrix for the 30-minutes experiment using M3 as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	6.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0

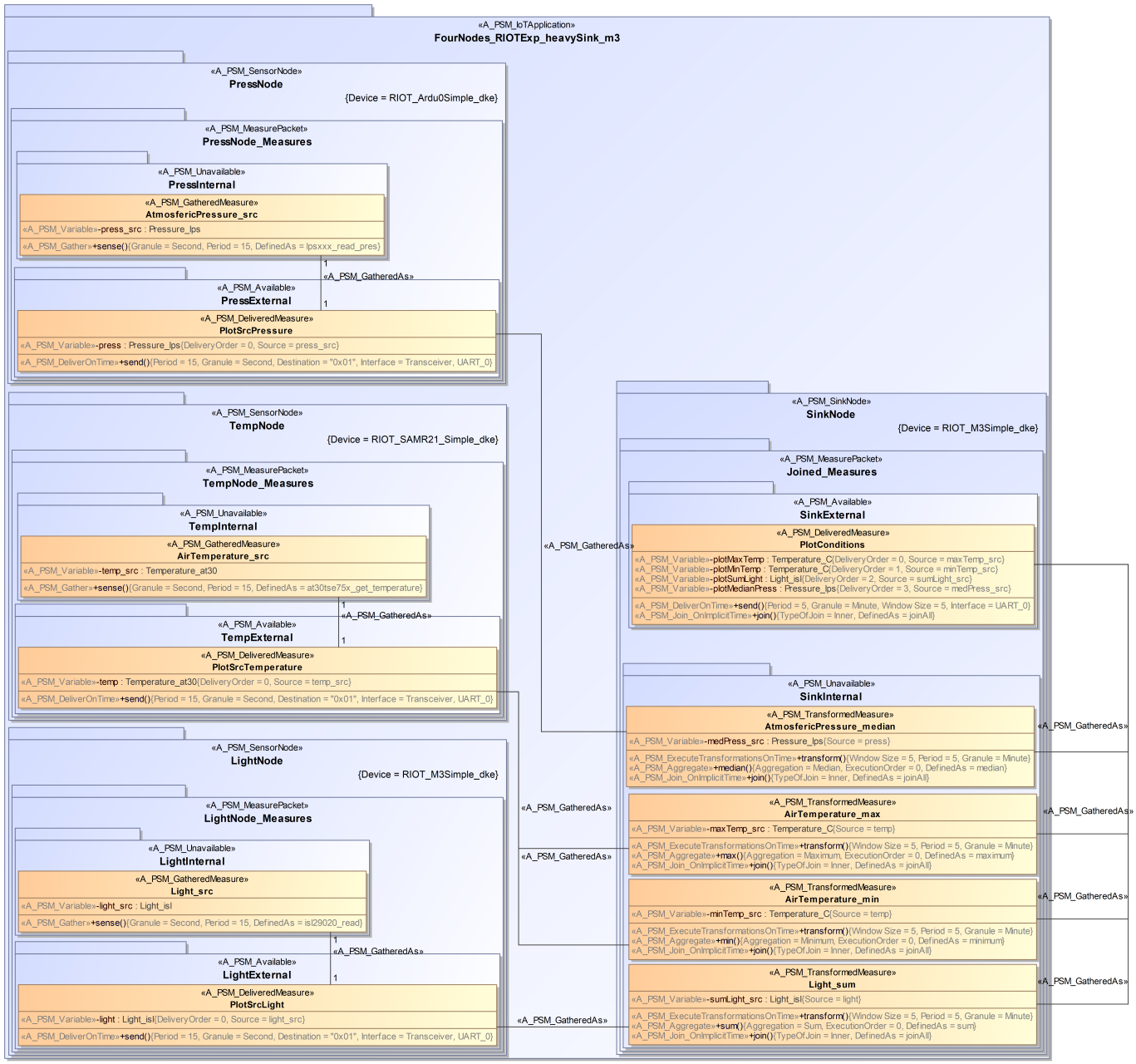

Figure B.1: PSM for the experiments implementing a **M3** board as SN.

Table B.2: Comparison matrix for the 1-hour experiment using M3 as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0
6	-1.0	0.0	0.0	0.0
7	1.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0
14	0.0	0.0	11.0	0.0

Table B.3: Comparison matrix for the 1-day experiment using M3 as SN.

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
0	0.0	0.0	0.0	-756.0
1	0.0	0.0	0.0	-756.0
2	0.0	0.0	0.0	-756.0
3	0.0	0.0	0.0	-756.0
4	0.0	0.0	0.0	-756.0
5	0.0	0.0	0.0	-756.0
6	0.0	0.0	0.0	-756.0
7	0.0	0.0	0.0	-756.0
8	0.0	0.0	0.0	-756.0
9	0.0	0.0	0.0	-756.0
10	0.0	0.0	0.0	-756.0
11	0.0	0.0	0.0	-756.0
12	0.0	0.0	0.0	-756.0
13	0.0	0.0	0.0	-756.0
14	0.0	1.0	11.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
15	0.0	0.0	0.0	-756.0
16	0.0	0.0	0.0	-756.0
17	0.0	0.0	0.0	-756.0
18	0.0	0.0	0.0	-756.0
19	0.0	0.0	0.0	-756.0
20	0.0	0.0	0.0	-756.0
21	0.0	0.0	0.0	-756.0
22	1.0	0.0	0.0	-756.0
23	0.0	0.0	0.0	-756.0
24	0.0	0.0	0.0	-756.0
25	0.0	0.0	0.0	-756.0
26	0.0	0.0	0.0	-756.0
27	0.0	0.0	0.0	-756.0
28	0.0	0.0	0.0	-756.0
29	0.0	0.0	0.0	-756.0
30	0.0	0.0	0.0	-756.0
31	0.0	0.0	0.0	-756.0
32	0.0	0.0	0.0	-756.0
33	0.0	0.0	0.0	-756.0
34	0.0	0.0	0.0	-756.0
35	0.0	0.0	0.0	-756.0
36	0.0	0.0	0.0	-756.0
37	0.0	0.0	0.0	-756.0
38	0.0	0.0	0.0	-756.0
39	0.0	1.0	0.0	-756.0
40	0.0	0.0	0.0	-756.0
41	0.0	0.0	0.0	-756.0
42	0.0	0.0	0.0	-756.0
43	0.0	0.0	0.0	-756.0
44	1.0	0.0	0.0	-756.0
45	0.0	0.0	0.0	-756.0
46	0.0	0.0	0.0	-756.0
47	0.0	0.0	0.0	-756.0
48	0.0	0.0	0.0	-756.0
49	0.0	0.0	0.0	-756.0
50	0.0	0.0	0.0	-756.0
51	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
52	0.0	0.0	0.0	-756.0
53	0.0	0.0	0.0	-756.0
54	0.0	0.0	0.0	-756.0
55	0.0	0.0	0.0	-756.0
56	0.0	0.0	0.0	-756.0
57	0.0	0.0	0.0	-756.0
58	0.0	0.0	0.0	-756.0
59	0.0	0.0	0.0	-756.0
60	0.0	0.0	0.0	-756.0
61	0.0	0.0	0.0	-756.0
62	0.0	0.0	0.0	-756.0
63	0.0	0.0	0.0	-756.0
64	0.0	0.0	0.0	-756.0
65	0.0	0.0	0.0	-756.0
66	0.0	0.0	0.0	-756.0
67	0.0	0.0	0.0	-756.0
68	0.0	0.0	0.0	-756.0
69	0.0	0.0	11.0	-756.0
70	0.0	0.0	0.0	-756.0
71	0.0	0.0	0.0	-756.0
72	0.0	0.0	0.0	-756.0
73	0.0	0.0	0.0	-756.0
74	0.0	0.0	0.0	-756.0
75	0.0	0.0	0.0	-756.0
76	0.0	0.0	0.0	-756.0
77	0.0	0.0	0.0	-756.0
78	0.0	0.0	0.0	-756.0
79	0.0	0.0	0.0	-756.0
80	0.0	0.0	0.0	-756.0
81	0.0	1.0	0.0	-756.0
82	0.0	-1.0	0.0	-756.0
83	0.0	0.0	0.0	-756.0
84	0.0	0.0	0.0	-756.0
85	0.0	0.0	0.0	-756.0
86	0.0	1.0	0.0	-756.0
87	0.0	0.0	0.0	-756.0
88	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
89	0.0	0.0	0.0	-756.0
90	0.0	0.0	0.0	-756.0
91	1.0	0.0	0.0	-756.0
92	-1.0	0.0	0.0	-756.0
93	1.0	0.0	0.0	-756.0
94	0.0	0.0	0.0	-756.0
95	0.0	0.0	0.0	-756.0
96	0.0	0.0	0.0	-756.0
97	0.0	0.0	0.0	-756.0
98	0.0	0.0	0.0	-756.0
99	0.0	0.0	0.0	-756.0
100	0.0	0.0	0.0	-756.0
101	0.0	0.0	0.0	-756.0
102	0.0	0.0	0.0	-756.0
103	0.0	0.0	0.0	-756.0
104	0.0	0.0	0.0	-756.0
105	0.0	0.0	0.0	-756.0
106	0.0	0.0	0.0	-756.0
107	0.0	0.0	0.0	-756.0
108	0.0	0.0	0.0	-756.0
109	0.0	0.0	0.0	-756.0
110	0.0	0.0	0.0	-756.0
111	0.0	0.0	0.0	-756.0
112	0.0	0.0	0.0	-756.0
113	0.0	0.0	0.0	-756.0
114	0.0	0.0	0.0	-756.0
115	0.0	0.0	0.0	-756.0
116	0.0	0.0	0.0	-756.0
117	0.0	0.0	0.0	-756.0
118	0.0	0.0	0.0	-756.0
119	0.0	0.0	0.0	-756.0
120	0.0	0.0	0.0	-756.0
121	0.0	0.0	0.0	-756.0
122	0.0	0.0	0.0	-756.0
123	0.0	0.0	0.0	-756.0
124	0.0	0.0	0.0	-756.0
125	0.0	0.0	11.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
126	0.0	0.0	0.0	-756.0
127	0.0	0.0	0.0	-756.0
128	0.0	0.0	0.0	-756.0
129	0.0	0.0	0.0	-756.0
130	0.0	1.0	0.0	-756.0
131	0.0	0.0	0.0	-756.0
132	0.0	0.0	0.0	-756.0
133	0.0	0.0	0.0	-756.0
134	0.0	0.0	0.0	-756.0
135	0.0	0.0	0.0	-756.0
136	0.0	0.0	0.0	-756.0
137	0.0	0.0	0.0	-756.0
138	0.0	0.0	0.0	-756.0
139	0.0	0.0	0.0	-756.0
140	0.0	0.0	0.0	-756.0
141	1.0	0.0	0.0	-756.0
142	0.0	0.0	0.0	-756.0
143	-1.0	0.0	0.0	-756.0
144	1.0	0.0	0.0	-756.0
145	0.0	0.0	0.0	-756.0
146	0.0	0.0	0.0	-756.0
147	0.0	0.0	0.0	-756.0
148	0.0	0.0	0.0	-756.0
149	0.0	0.0	0.0	-756.0
150	0.0	0.0	0.0	-756.0
151	0.0	0.0	0.0	-756.0
152	0.0	0.0	0.0	-756.0
153	0.0	0.0	0.0	-756.0
154	0.0	0.0	0.0	-756.0
155	0.0	0.0	0.0	-756.0
156	0.0	0.0	0.0	-756.0
157	0.0	0.0	0.0	-756.0
158	0.0	0.0	0.0	-756.0
159	0.0	0.0	0.0	-756.0
160	0.0	0.0	0.0	-756.0
161	0.0	0.0	0.0	-756.0
162	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
163	0.0	0.0	0.0	-756.0
164	0.0	0.0	0.0	-756.0
165	0.0	0.0	0.0	-756.0
166	0.0	0.0	0.0	-756.0
167	0.0	0.0	0.0	-756.0
168	0.0	0.0	0.0	-756.0
169	0.0	0.0	0.0	-756.0
170	-1.0	0.0	0.0	-756.0
171	0.0	0.0	0.0	-756.0
172	0.0	0.0	0.0	-756.0
173	0.0	0.0	0.0	-756.0
174	0.0	-1.0	0.0	-756.0
175	0.0	0.0	0.0	-756.0
176	0.0	0.0	0.0	-756.0
177	0.0	0.0	0.0	-756.0
178	0.0	0.0	0.0	-756.0
179	0.0	0.0	11.0	-756.0
180	-1.0	0.0	0.0	-756.0
181	0.0	0.0	0.0	-756.0
182	0.0	0.0	0.0	-756.0
183	0.0	-1.0	0.0	-756.0
184	0.0	0.0	0.0	-756.0
185	0.0	0.0	0.0	-756.0
186	0.0	0.0	0.0	-756.0
187	-1.0	0.0	0.0	-756.0
188	0.0	-1.0	0.0	-756.0
189	0.0	0.0	0.0	-756.0
190	0.0	0.0	0.0	-756.0
191	0.0	0.0	0.0	-756.0
192	-1.0	0.0	0.0	-756.0
193	0.0	-1.0	0.0	-756.0
194	0.0	0.0	0.0	-756.0
195	0.0	1.0	0.0	-756.0
196	0.0	0.0	0.0	-756.0
197	0.0	-1.0	0.0	-756.0
198	0.0	0.0	0.0	-756.0
199	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
200	0.0	0.0	0.0	-756.0
201	0.0	0.0	0.0	-756.0
202	0.0	0.0	0.0	-756.0
203	0.0	0.0	0.0	-756.0
204	0.0	0.0	0.0	-756.0
205	0.0	0.0	0.0	-756.0
206	-1.0	0.0	0.0	-756.0
207	0.0	0.0	0.0	-756.0
208	0.0	-1.0	0.0	-756.0
209	0.0	0.0	0.0	-756.0
210	0.0	0.0	0.0	-756.0
211	0.0	0.0	0.0	-756.0
212	0.0	1.0	0.0	-756.0
213	0.0	0.0	0.0	-756.0
214	0.0	0.0	0.0	-756.0
215	0.0	0.0	0.0	-756.0
216	0.0	0.0	0.0	-756.0
217	0.0	0.0	0.0	-756.0
218	1.0	0.0	0.0	-756.0
219	0.0	0.0	0.0	-756.0
220	0.0	0.0	0.0	-756.0
221	0.0	0.0	0.0	-756.0
222	-1.0	0.0	0.0	-756.0
223	0.0	0.0	0.0	-756.0
224	0.0	-1.0	0.0	-756.0
225	0.0	0.0	0.0	-756.0
226	0.0	0.0	0.0	-756.0
227	0.0	0.0	0.0	-756.0
228	0.0	0.0	0.0	-756.0
229	0.0	0.0	0.0	-756.0
230	0.0	0.0	0.0	-756.0
231	0.0	0.0	0.0	-756.0
232	0.0	0.0	0.0	-756.0
233	0.0	0.0	0.0	-756.0
234	0.0	0.0	11.0	-756.0
235	0.0	0.0	0.0	-756.0
236	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
237	0.0	0.0	0.0	-756.0
238	0.0	0.0	0.0	-756.0
239	0.0	0.0	0.0	-756.0
240	0.0	0.0	0.0	-756.0
241	0.0	0.0	0.0	-756.0
242	0.0	1.0	0.0	-756.0
243	0.0	0.0	0.0	-756.0
244	0.0	0.0	0.0	-756.0
245	0.0	0.0	0.0	-756.0
246	0.0	0.0	0.0	-756.0
247	0.0	0.0	0.0	-756.0
248	0.0	0.0	0.0	-756.0
249	0.0	0.0	0.0	-756.0
250	0.0	0.0	0.0	-756.0
251	0.0	0.0	0.0	-756.0
252	0.0	0.0	0.0	-756.0
253	1.0	0.0	0.0	-756.0
254	0.0	0.0	0.0	-756.0
255	-1.0	0.0	0.0	-756.0
256	0.0	0.0	0.0	-756.0
257	1.0	0.0	0.0	-756.0
258	0.0	0.0	0.0	-756.0
259	0.0	0.0	0.0	-756.0
260	0.0	0.0	0.0	-756.0
261	0.0	0.0	0.0	-756.0
262	0.0	0.0	0.0	-756.0
263	0.0	0.0	0.0	-756.0
264	0.0	0.0	0.0	-756.0
265	0.0	0.0	0.0	-756.0
266	0.0	0.0	0.0	-756.0
267	0.0	0.0	0.0	-756.0
268	0.0	0.0	0.0	-756.0
269	0.0	0.0	0.0	-756.0
270	0.0	0.0	0.0	-756.0
271	0.0	0.0	0.0	-756.0
272	0.0	0.0	0.0	-756.0
273	0.0	0.0	0.0	-756.0

Table B.3 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
274	0.0	0.0	0.0	-756.0
275	0.0	1.0	0.0	-756.0
276	0.0	0.0	0.0	-756.0
277	0.0	0.0	0.0	-756.0
278	0.0	0.0	0.0	-756.0
279	0.0	0.0	0.0	-756.0
280	0.0	0.0	0.0	-756.0
281	0.0	0.0	0.0	-756.0
282	0.0	0.0	0.0	-756.0
283	0.0	0.0	0.0	-756.0
284	0.0	0.0	0.0	-756.0
285	0.0	0.0	0.0	-756.0
286	0.0	0.0	0.0	-756.0
287	0.0	0.0	0.0	-756.0

B.2 Tests Results Using an Microchip SAMR21 Xplained Pro as SN

Figure B.2 shows the PSM for these tests. Table B.4 shows the results for the 30-minutes test with this PSM. Table B.5 shows the results for the 1-hour test, and Table B.6 the 1-day test results.

Table B.4: Comparison matrix for the 30-minutes experiment using SAMR21 as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0

Table B.5: Comparison matrix for the 1-hour experiment using SAMR21 as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0
7	0.0	0.0	35.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0

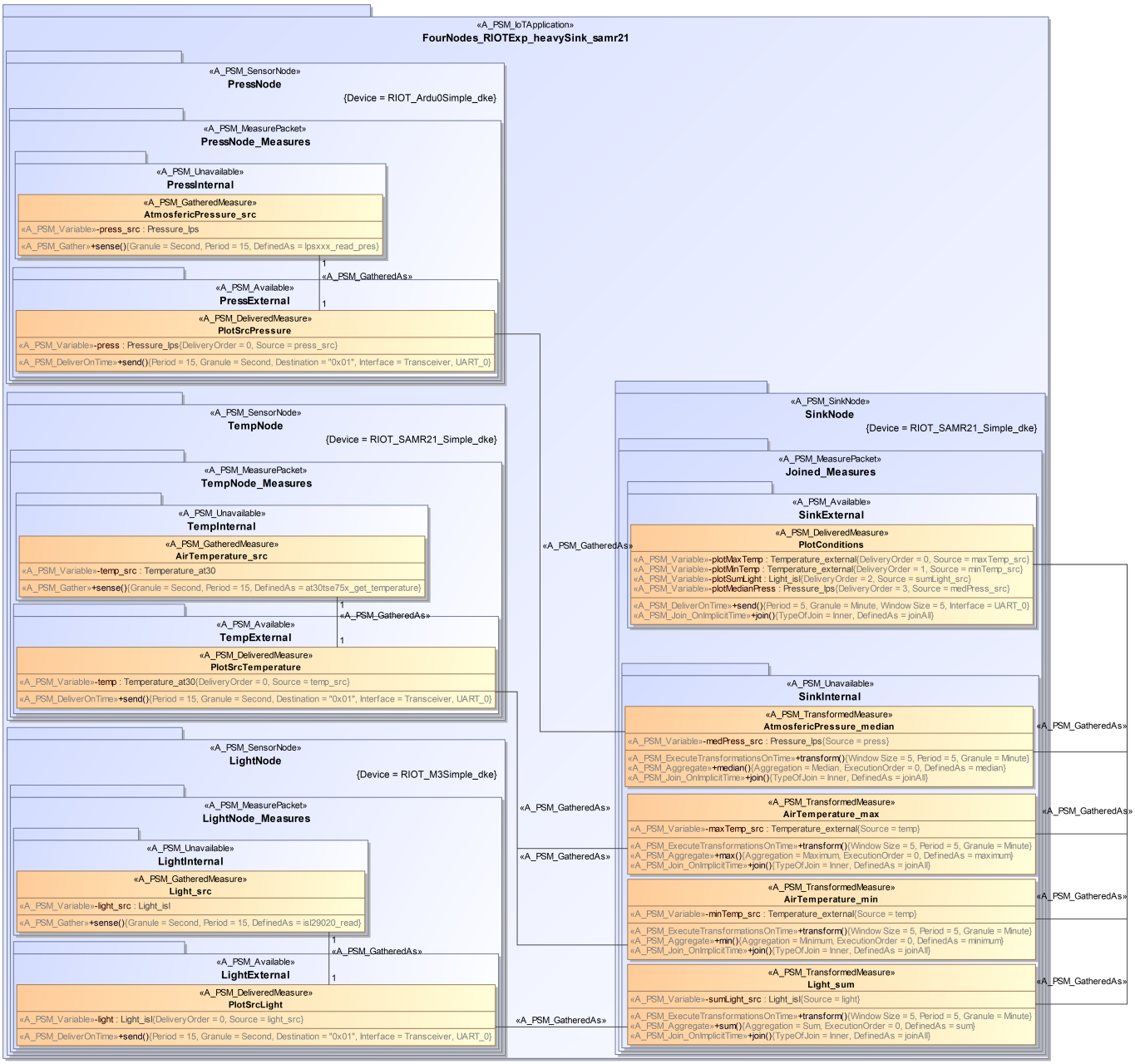

Figure B.2: PSM for the experiments implementing a **SAMR21** board as SN.

Table B.6: Comparison matrix for the 1-day experiment using SAMR21 as SN.

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0
6	0.0	0.0	35.0	0.0
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0
17	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0	0.0
19	0.0	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0
21	0.0	0.0	0.0	0.0
22	0.0	0.0	0.0	0.0
23	0.0	0.0	0.0	0.0
24	0.0	0.0	0.0	0.0
25	0.0	0.0	0.0	0.0
26	0.0	0.0	0.0	0.0
27	0.0	0.0	0.0	0.0
28	0.0	0.0	0.0	0.0
29	0.0	0.0	0.0	0.0
30	0.0	0.0	0.0	0.0
31	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0
33	0.0	0.0	0.0	0.0
34	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
35	0.0	0.0	0.0	0.0
36	0.0	0.0	35.0	0.0
37	0.0	0.0	0.0	0.0
38	0.0	0.0	0.0	0.0
39	0.0	0.0	0.0	0.0
40	0.0	0.0	0.0	0.0
41	0.0	0.0	0.0	0.0
42	0.0	0.0	0.0	0.0
43	0.0	0.0	0.0	0.0
44	0.0	0.0	0.0	0.0
45	0.0	0.0	0.0	0.0
46	0.0	0.0	0.0	0.0
47	0.0	0.0	0.0	0.0
48	0.0	0.0	0.0	0.0
49	0.0	0.0	0.0	0.0
50	0.0	0.0	0.0	0.0
51	0.0	0.0	0.0	0.0
52	0.0	0.0	0.0	0.0
53	0.0	0.0	0.0	0.0
54	0.0	0.0	0.0	0.0
55	0.0	0.0	0.0	0.0
56	0.0	0.0	0.0	0.0
57	0.0	0.0	0.0	0.0
58	0.0	0.0	0.0	0.0
59	0.0	0.0	0.0	0.0
60	0.0	0.0	0.0	0.0
61	0.0	0.0	0.0	0.0
62	0.0	0.0	0.0	0.0
63	0.0	0.0	0.0	0.0
64	0.0	0.0	0.0	0.0
65	0.0	0.0	0.0	0.0
66	0.0	0.0	0.0	0.0
67	0.0	0.0	0.0	0.0
68	0.0	0.0	0.0	0.0
69	0.0	0.0	0.0	0.0
70	0.0	0.0	0.0	0.0
71	0.0	0.0	35.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
72	0.0	0.0	0.0	0.0
73	0.0	0.0	0.0	0.0
74	0.0	0.0	0.0	0.0
75	0.0	0.0	0.0	0.0
76	0.0	0.0	0.0	0.0
77	0.0	0.0	0.0	0.0
78	0.0	0.0	0.0	0.0
79	0.0	0.0	0.0	0.0
80	0.0	0.0	0.0	0.0
81	0.0	0.0	0.0	0.0
82	0.0	0.0	0.0	0.0
83	0.0	0.0	0.0	0.0
84	0.0	0.0	0.0	0.0
85	0.0	0.0	0.0	0.0
86	0.0	0.0	0.0	0.0
87	0.0	0.0	0.0	0.0
88	0.0	0.0	0.0	0.0
89	0.0	0.0	0.0	0.0
90	0.0	0.0	0.0	0.0
91	0.0	0.0	0.0	0.0
92	0.0	0.0	0.0	0.0
93	0.0	0.0	0.0	0.0
94	0.0	0.0	0.0	0.0
95	0.0	0.0	0.0	0.0
96	0.0	0.0	0.0	0.0
97	0.0	0.0	0.0	0.0
98	0.0	0.0	0.0	0.0
99	0.0	0.0	0.0	0.0
100	0.0	0.0	0.0	0.0
101	0.0	0.0	0.0	0.0
102	0.0	0.0	0.0	0.0
103	0.0	0.0	0.0	0.0
104	0.0	0.0	0.0	0.0
105	0.0	0.0	0.0	0.0
106	0.0	0.0	0.0	0.0
107	0.0	0.0	0.0	0.0
108	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
109	0.0	0.0	0.0	0.0
110	0.0	0.0	0.0	0.0
111	0.0	0.0	0.0	0.0
112	0.0	0.0	35.0	0.0
113	0.0	0.0	0.0	0.0
114	0.0	0.0	0.0	0.0
115	0.0	0.0	0.0	0.0
116	0.0	0.0	0.0	0.0
117	0.0	0.0	0.0	0.0
118	0.0	0.0	0.0	0.0
119	0.0	0.0	0.0	0.0
120	0.0	0.0	0.0	0.0
121	0.0	0.0	0.0	0.0
122	0.0	0.0	0.0	0.0
123	0.0	0.0	0.0	0.0
124	0.0	0.0	0.0	0.0
125	0.0	0.0	0.0	0.0
126	0.0	0.0	0.0	0.0
127	0.0	0.0	0.0	0.0
128	0.0	0.0	0.0	0.0
129	0.0	0.0	0.0	0.0
130	0.0	0.0	0.0	0.0
131	0.0	-1.0	0.0	0.0
132	0.0	0.0	0.0	0.0
133	0.0	0.0	0.0	0.0
134	0.0	0.0	0.0	0.0
135	0.0	0.0	0.0	0.0
136	0.0	0.0	0.0	0.0
137	0.0	0.0	0.0	0.0
138	0.0	0.0	0.0	0.0
139	0.0	0.0	0.0	0.0
140	0.0	0.0	0.0	0.0
141	0.0	0.0	0.0	0.0
142	0.0	0.0	0.0	0.0
143	0.0	0.0	0.0	0.0
144	0.0	0.0	0.0	0.0
145	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
146	0.0	0.0	0.0	0.0
147	0.0	0.0	0.0	0.0
148	0.0	0.0	0.0	0.0
149	0.0	0.0	0.0	0.0
150	0.0	0.0	0.0	0.0
151	0.0	0.0	0.0	0.0
152	0.0	0.0	0.0	0.0
153	0.0	0.0	0.0	0.0
154	0.0	0.0	0.0	0.0
155	0.0	0.0	0.0	0.0
156	0.0	0.0	0.0	0.0
157	0.0	0.0	0.0	0.0
158	0.0	0.0	0.0	0.0
159	0.0	0.0	0.0	0.0
160	0.0	0.0	0.0	0.0
161	0.0	0.0	35.0	0.0
162	0.0	0.0	0.0	0.0
163	0.0	0.0	0.0	0.0
164	0.0	0.0	0.0	0.0
165	0.0	0.0	0.0	0.0
166	0.0	0.0	0.0	0.0
167	0.0	0.0	0.0	0.0
168	0.0	0.0	0.0	0.0
169	0.0	0.0	0.0	0.0
170	0.0	0.0	0.0	0.0
171	0.0	0.0	0.0	0.0
172	0.0	0.0	0.0	0.0
173	0.0	0.0	0.0	0.0
174	0.0	0.0	0.0	0.0
175	0.0	0.0	0.0	0.0
176	0.0	0.0	0.0	0.0
177	0.0	0.0	0.0	0.0
178	0.0	0.0	0.0	0.0
179	0.0	0.0	0.0	0.0
180	0.0	0.0	0.0	0.0
181	0.0	0.0	0.0	0.0
182	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
183	1.0	0.0	0.0	0.0
184	0.0	0.0	0.0	0.0
185	0.0	0.0	0.0	0.0
186	0.0	0.0	0.0	0.0
187	0.0	0.0	0.0	0.0
188	0.0	0.0	0.0	0.0
189	0.0	0.0	0.0	0.0
190	0.0	0.0	0.0	0.0
191	0.0	0.0	0.0	0.0
192	0.0	0.0	0.0	0.0
193	0.0	0.0	0.0	0.0
194	0.0	0.0	0.0	0.0
195	0.0	0.0	0.0	0.0
196	0.0	0.0	0.0	0.0
197	0.0	0.0	0.0	0.0
198	0.0	0.0	0.0	0.0
199	0.0	0.0	0.0	0.0
200	0.0	0.0	35.0	0.0
201	0.0	0.0	0.0	0.0
202	0.0	0.0	0.0	0.0
203	0.0	0.0	0.0	0.0
204	0.0	0.0	0.0	0.0
205	0.0	0.0	0.0	0.0
206	0.0	0.0	0.0	0.0
207	0.0	0.0	0.0	0.0
208	0.0	0.0	0.0	0.0
209	0.0	0.0	0.0	0.0
210	0.0	0.0	0.0	0.0
211	0.0	0.0	0.0	0.0
212	0.0	0.0	0.0	0.0
213	0.0	0.0	0.0	0.0
214	0.0	0.0	0.0	0.0
215	0.0	0.0	0.0	0.0
216	0.0	0.0	0.0	0.0
217	0.0	0.0	0.0	0.0
218	0.0	0.0	0.0	0.0
219	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
220	0.0	0.0	0.0	0.0
221	0.0	0.0	0.0	0.0
222	0.0	0.0	0.0	0.0
223	0.0	0.0	0.0	0.0
224	0.0	0.0	0.0	0.0
225	0.0	0.0	0.0	0.0
226	0.0	0.0	0.0	0.0
227	0.0	0.0	35.0	0.0
228	0.0	0.0	0.0	0.0
229	0.0	0.0	0.0	0.0
230	0.0	0.0	0.0	0.0
231	0.0	0.0	0.0	0.0
232	0.0	0.0	0.0	0.0
233	0.0	0.0	0.0	0.0
234	0.0	0.0	0.0	0.0
235	0.0	0.0	0.0	0.0
236	0.0	0.0	0.0	0.0
237	0.0	0.0	0.0	0.0
238	0.0	0.0	0.0	0.0
239	0.0	0.0	0.0	0.0
240	0.0	0.0	0.0	0.0
241	0.0	0.0	0.0	0.0
242	0.0	0.0	0.0	0.0
243	0.0	0.0	0.0	0.0
244	0.0	0.0	0.0	0.0
245	0.0	0.0	0.0	0.0
246	0.0	0.0	0.0	0.0
247	0.0	0.0	0.0	0.0
248	0.0	0.0	0.0	0.0
249	0.0	0.0	0.0	0.0
250	0.0	0.0	0.0	0.0
251	0.0	0.0	0.0	0.0
252	0.0	0.0	0.0	0.0
253	0.0	0.0	35.0	0.0
254	0.0	0.0	0.0	0.0
255	0.0	0.0	0.0	0.0
256	0.0	0.0	0.0	0.0

Table B.6 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
257	0.0	0.0	0.0	0.0
258	0.0	0.0	0.0	0.0
259	0.0	0.0	0.0	0.0
260	0.0	0.0	0.0	0.0
261	0.0	0.0	0.0	0.0
262	0.0	0.0	0.0	0.0
263	0.0	0.0	0.0	0.0
264	0.0	0.0	0.0	0.0
265	0.0	0.0	0.0	0.0
266	0.0	0.0	0.0	0.0
267	0.0	0.0	0.0	0.0
268	0.0	0.0	0.0	0.0
269	0.0	0.0	0.0	0.0
270	0.0	0.0	0.0	0.0
271	0.0	0.0	0.0	0.0
272	0.0	0.0	0.0	0.0
273	0.0	0.0	0.0	0.0
274	0.0	0.0	0.0	0.0
275	0.0	0.0	0.0	0.0
276	0.0	0.0	0.0	0.0
277	0.0	0.0	0.0	0.0
278	0.0	0.0	0.0	0.0
279	0.0	0.0	0.0	0.0
280	0.0	0.0	0.0	0.0
281	0.0	0.0	0.0	0.0
282	0.0	0.0	0.0	0.0
283	0.0	0.0	0.0	0.0
284	0.0	0.0	35.0	0.0
285	0.0	0.0	0.0	0.0
286	0.0	0.0	0.0	0.0
287	0.0	0.0	0.0	0.0

B.3 Tests Results Using an Arduino Zero as SN

Figure B.3 shows the PSM for these tests. Table B.7 shows the results for the 30-minutes test with this PSM. Table B.8 shows the results for the 1-hour test, and Table B.9 the 1-day test results.

Table B.7: Comparison matrix for the 30-minutes experiment using Arduino Zero as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	0.0	0.0	14.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0

Table B.8: Comparison matrix for the 1-hour experiment using Arduino Zero as SN.

Time Window	Max Temperature	Min Temperature	Sum Light	Median Pressure
0	0.0	0.0	0.0	-756.0
1	0.0	0.0	60.0	-756.0
2	0.0	0.0	0.0	-756.0
3	0.0	0.0	0.0	-756.0
4	0.0	0.0	0.0	-756.0
5	0.0	0.0	0.0	-756.0
6	0.0	0.0	0.0	-756.0
7	0.0	0.0	0.0	-756.0
8	0.0	0.0	0.0	-756.0
9	0.0	0.0	0.0	-756.0
10	0.0	0.0	60.0	-756.0
11	0.0	0.0	0.0	-756.0
12	0.0	0.0	0.0	-756.0
13	0.0	0.0	0.0	-756.0
14	0.0	0.0	0.0	-756.0
15	0.0	0.0	0.0	-756.0

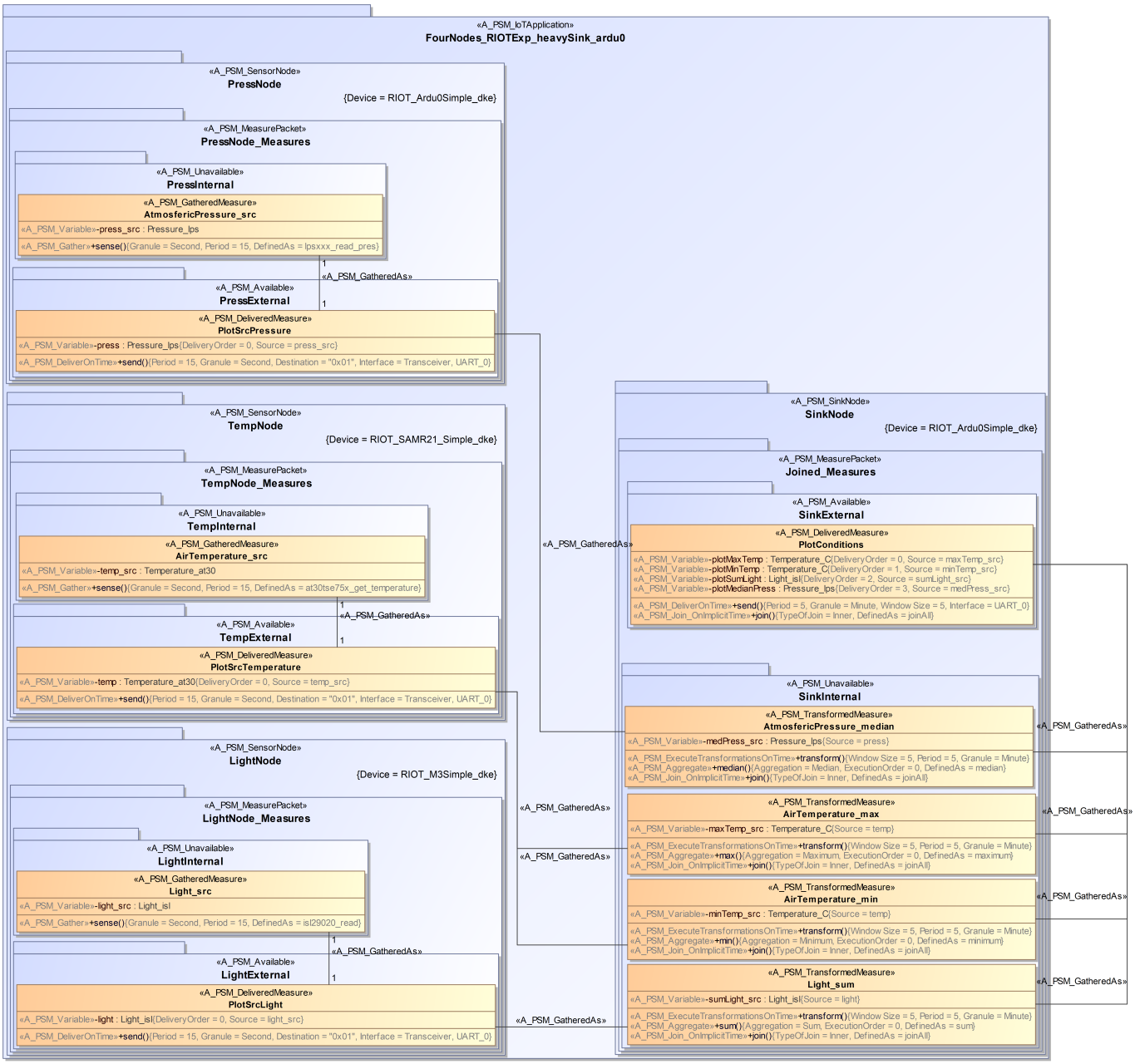

Figure B.3: PSM for the experiments implementing an **Arduino Zero** board as SN.

Table B.9: Comparison matrix for the 1-day experiment using Arduino Zero as SN.

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
0	0.0	24.0	0.0	0.0
1	0.0	0.0	60.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
10	0.0	0.0	60.0	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0
14	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0
17	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0	0.0
19	0.0	0.0	60.0	0.0
20	0.0	0.0	0.0	0.0
21	0.0	0.0	0.0	0.0
22	0.0	0.0	0.0	0.0
23	0.0	0.0	0.0	0.0
24	0.0	0.0	0.0	0.0
25	0.0	0.0	0.0	0.0
26	0.0	0.0	0.0	0.0
27	0.0	0.0	0.0	0.0
28	0.0	0.0	60.0	0.0
29	0.0	0.0	0.0	0.0
30	0.0	0.0	0.0	0.0
31	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0
33	0.0	0.0	0.0	0.0
34	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
35	0.0	0.0	0.0	0.0
36	0.0	0.0	0.0	0.0
37	0.0	0.0	60.0	0.0
38	0.0	0.0	0.0	0.0
39	0.0	0.0	0.0	0.0
40	0.0	0.0	0.0	0.0
41	0.0	0.0	0.0	0.0
42	0.0	0.0	0.0	0.0
43	0.0	0.0	0.0	0.0
44	0.0	0.0	0.0	0.0
45	0.0	0.0	0.0	0.0
46	0.0	0.0	60.0	0.0
47	0.0	0.0	0.0	0.0
48	0.0	0.0	0.0	0.0
49	0.0	0.0	0.0	0.0
50	0.0	0.0	0.0	0.0
51	0.0	0.0	0.0	0.0
52	0.0	0.0	0.0	0.0
53	0.0	0.0	0.0	0.0
54	0.0	0.0	0.0	0.0
55	0.0	0.0	60.0	0.0
56	0.0	0.0	0.0	0.0
57	0.0	0.0	0.0	0.0
58	0.0	0.0	0.0	0.0
59	0.0	0.0	0.0	0.0
60	0.0	0.0	0.0	0.0
61	0.0	0.0	0.0	0.0
62	0.0	0.0	0.0	0.0
63	0.0	0.0	0.0	0.0
64	0.0	0.0	0.0	0.0
65	0.0	0.0	60.0	0.0
66	0.0	0.0	0.0	0.0
67	0.0	0.0	0.0	0.0
68	0.0	0.0	0.0	0.0
69	0.0	0.0	0.0	0.0
70	0.0	0.0	0.0	0.0
71	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
72	0.0	0.0	0.0	0.0
73	0.0	0.0	0.0	0.0
74	0.0	0.0	60.0	0.0
75	0.0	0.0	0.0	0.0
76	0.0	0.0	0.0	0.0
77	0.0	0.0	0.0	0.0
78	0.0	0.0	0.0	0.0
79	0.0	0.0	0.0	0.0
80	0.0	0.0	0.0	0.0
81	0.0	0.0	0.0	0.0
82	0.0	0.0	0.0	0.0
83	0.0	0.0	60.0	0.0
84	0.0	0.0	0.0	0.0
85	0.0	0.0	0.0	0.0
86	0.0	0.0	0.0	0.0
87	0.0	0.0	0.0	0.0
88	0.0	0.0	0.0	0.0
89	0.0	0.0	0.0	0.0
90	0.0	0.0	0.0	0.0
91	0.0	0.0	0.0	0.0
92	0.0	0.0	0.0	0.0
93	0.0	0.0	60.0	0.0
94	0.0	0.0	0.0	0.0
95	0.0	0.0	0.0	0.0
96	0.0	0.0	0.0	0.0
97	0.0	0.0	0.0	0.0
98	0.0	0.0	0.0	0.0
99	0.0	0.0	0.0	0.0
100	0.0	0.0	0.0	0.0
101	0.0	0.0	0.0	0.0
102	0.0	0.0	60.0	0.0
103	0.0	0.0	0.0	0.0
104	0.0	0.0	0.0	0.0
105	0.0	0.0	0.0	0.0
106	0.0	0.0	0.0	0.0
107	0.0	0.0	0.0	0.0
108	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
109	0.0	0.0	0.0	0.0
110	0.0	0.0	0.0	0.0
111	0.0	0.0	0.0	0.0
112	0.0	0.0	60.0	0.0
113	0.0	0.0	0.0	0.0
114	0.0	0.0	0.0	0.0
115	0.0	0.0	0.0	0.0
116	0.0	0.0	0.0	0.0
117	0.0	0.0	0.0	0.0
118	0.0	0.0	0.0	0.0
119	0.0	0.0	0.0	0.0
120	0.0	0.0	60.0	0.0
121	0.0	0.0	0.0	0.0
122	0.0	0.0	0.0	0.0
123	0.0	0.0	0.0	0.0
124	0.0	0.0	0.0	0.0
125	0.0	0.0	0.0	0.0
126	0.0	0.0	0.0	0.0
127	0.0	0.0	0.0	0.0
128	0.0	0.0	60.0	0.0
129	0.0	0.0	0.0	0.0
130	0.0	0.0	0.0	0.0
131	0.0	0.0	0.0	0.0
132	0.0	0.0	0.0	0.0
133	0.0	0.0	0.0	0.0
134	1.0	0.0	0.0	0.0
135	0.0	0.0	0.0	0.0
136	0.0	0.0	60.0	0.0
137	0.0	0.0	0.0	0.0
138	0.0	0.0	0.0	0.0
139	0.0	0.0	0.0	0.0
140	0.0	0.0	0.0	0.0
141	0.0	0.0	0.0	0.0
142	0.0	0.0	0.0	0.0
143	0.0	0.0	60.0	0.0
144	0.0	0.0	0.0	0.0
145	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
146	0.0	0.0	0.0	0.0
147	0.0	0.0	0.0	0.0
148	0.0	0.0	0.0	0.0
149	0.0	0.0	0.0	0.0
150	0.0	0.0	0.0	0.0
151	0.0	0.0	60.0	0.0
152	0.0	0.0	0.0	0.0
153	0.0	0.0	0.0	0.0
154	0.0	0.0	0.0	0.0
155	0.0	0.0	0.0	0.0
156	0.0	0.0	0.0	0.0
157	0.0	0.0	0.0	0.0
158	0.0	0.0	60.0	0.0
159	0.0	0.0	0.0	0.0
160	0.0	0.0	0.0	0.0
161	0.0	0.0	0.0	0.0
162	0.0	0.0	0.0	0.0
163	0.0	0.0	0.0	0.0
164	0.0	0.0	0.0	0.0
165	0.0	0.0	60.0	0.0
166	0.0	0.0	0.0	0.0
167	0.0	0.0	0.0	0.0
168	0.0	0.0	0.0	0.0
169	0.0	0.0	0.0	0.0
170	0.0	0.0	0.0	0.0
171	0.0	0.0	0.0	0.0
172	0.0	0.0	0.0	0.0
173	0.0	0.0	60.0	0.0
174	0.0	0.0	0.0	0.0
175	0.0	0.0	0.0	0.0
176	0.0	0.0	0.0	0.0
177	0.0	0.0	0.0	0.0
178	0.0	0.0	0.0	0.0
179	0.0	0.0	0.0	0.0
180	0.0	0.0	0.0	0.0
181	0.0	0.0	60.0	0.0
182	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
183	0.0	0.0	0.0	0.0
184	0.0	0.0	0.0	0.0
185	0.0	0.0	0.0	0.0
186	0.0	0.0	0.0	0.0
187	0.0	0.0	0.0	0.0
188	0.0	0.0	60.0	0.0
189	0.0	0.0	0.0	0.0
190	0.0	0.0	0.0	0.0
191	0.0	0.0	0.0	0.0
192	0.0	0.0	0.0	0.0
193	0.0	0.0	0.0	0.0
194	0.0	0.0	0.0	0.0
195	0.0	0.0	0.0	0.0
196	0.0	0.0	60.0	0.0
197	0.0	0.0	0.0	0.0
198	0.0	0.0	0.0	0.0
199	0.0	0.0	0.0	0.0
200	0.0	0.0	0.0	0.0
201	0.0	0.0	0.0	0.0
202	0.0	0.0	0.0	0.0
203	0.0	0.0	60.0	0.0
204	0.0	0.0	0.0	0.0
205	0.0	0.0	0.0	0.0
206	0.0	0.0	0.0	0.0
207	0.0	0.0	0.0	0.0
208	1.0	0.0	0.0	0.0
209	-1.0	0.0	0.0	0.0
210	0.0	0.0	0.0	0.0
211	0.0	0.0	60.0	0.0
212	0.0	0.0	0.0	0.0
213	0.0	0.0	0.0	0.0
214	0.0	0.0	0.0	0.0
215	0.0	0.0	0.0	0.0
216	0.0	0.0	0.0	0.0
217	0.0	0.0	0.0	0.0
218	0.0	-1.0	0.0	0.0
219	0.0	1.0	60.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
220	0.0	0.0	0.0	0.0
221	0.0	0.0	0.0	0.0
222	0.0	0.0	0.0	0.0
223	0.0	0.0	0.0	0.0
224	0.0	0.0	0.0	0.0
225	0.0	0.0	0.0	0.0
226	0.0	0.0	0.0	0.0
227	0.0	0.0	0.0	0.0
228	0.0	0.0	60.0	0.0
229	0.0	0.0	0.0	0.0
230	0.0	0.0	0.0	0.0
231	0.0	0.0	0.0	0.0
232	0.0	0.0	0.0	0.0
233	0.0	0.0	0.0	0.0
234	0.0	0.0	0.0	0.0
235	0.0	0.0	0.0	0.0
236	0.0	0.0	60.0	0.0
237	0.0	0.0	0.0	0.0
238	0.0	0.0	0.0	0.0
239	0.0	0.0	0.0	0.0
240	0.0	0.0	0.0	0.0
241	0.0	0.0	0.0	0.0
242	0.0	0.0	0.0	0.0
243	0.0	0.0	0.0	0.0
244	0.0	0.0	60.0	0.0
245	0.0	0.0	0.0	0.0
246	0.0	0.0	0.0	0.0
247	0.0	0.0	0.0	0.0
248	0.0	0.0	0.0	0.0
249	0.0	0.0	0.0	0.0
250	0.0	0.0	0.0	0.0
251	0.0	0.0	0.0	0.0
252	0.0	0.0	60.0	0.0
253	0.0	0.0	0.0	0.0
254	0.0	0.0	0.0	0.0
255	0.0	0.0	0.0	0.0
256	0.0	0.0	0.0	0.0

Table B.9 continued from previous page

Time Window	Error in Max Temperature	Error in Min Temperature	Error in Sum Light	Error in Median Pressure
257	0.0	0.0	0.0	0.0
258	0.0	0.0	0.0	0.0
259	0.0	0.0	0.0	0.0
260	0.0	0.0	0.0	0.0
261	0.0	0.0	60.0	0.0
262	0.0	0.0	0.0	0.0
263	0.0	0.0	0.0	0.0
264	0.0	0.0	0.0	0.0
265	0.0	0.0	0.0	0.0
266	0.0	0.0	0.0	0.0
267	0.0	0.0	0.0	0.0
268	0.0	0.0	0.0	0.0
269	0.0	0.0	0.0	0.0
270	0.0	0.0	60.0	0.0
271	0.0	0.0	0.0	0.0
272	0.0	0.0	0.0	0.0
273	0.0	0.0	0.0	0.0
274	0.0	0.0	0.0	0.0
275	0.0	0.0	0.0	0.0
276	0.0	0.0	0.0	0.0
277	0.0	0.0	0.0	0.0
278	0.0	0.0	0.0	0.0
279	0.0	0.0	60.0	0.0
280	0.0	0.0	0.0	0.0
281	0.0	0.0	0.0	0.0
282	0.0	0.0	0.0	0.0
283	0.0	0.0	0.0	0.0
284	0.0	0.0	0.0	0.0
285	0.0	0.0	0.0	0.0
286	0.0	0.0	0.0	0.0
287	0.0	0.0	0.0	0.0

Appendix C

Résumé

Cette Annexe fournit un résumé détaillé de la base de la thèse et des résultats en langue française.

C.1 Introduction

L'observation continue de différents phénomènes a permis à l'humanité de découvrir de nouvelles connaissances et d'identifier des situations potentiellement bénéfiques ou dangereuses. En effet, l'Internet des objets (IoT) a (en partie) gagné en pertinence dans plusieurs domaines académiques et industriels pour sa capacité à collecter automatiquement et en continu des données à partir d'objets et de phénomènes surveillés.

L'IoT a été défini comme un ensemble d'appareils physiques connectés à Internet et capables de générer, de calculer, de stocker et d'envoyer des données en temps réel via différents systèmes (par exemple, ZigBee, Wi-Fi, LoRaWAN).

Ces informations permettent aux décideurs (par exemple, les chercheurs ou les gestionnaires) de prendre de meilleures mesures et de contrôler les processus sous-jacents pour améliorer leurs résultats ou réduire les dommages et éviter les problèmes.

Cependant, la collecte, le traitement, le stockage, l'analyse et l'utilisation des données IoT peuvent être difficiles. Les différents capteurs de ces réseaux peuvent générer différents types de données avec différentes configurations (données hétérogènes) tout en fournissant des informations multidimensionnelles et multi-échelles sur l'objet ou le phénomène détecté. En outre, les grands réseaux IoT peuvent être une source continue de Big Data (BD). De plus, les systèmes d'analyse de données tels que Business Intelligence (BI) doivent gérer les données IoT pour répondre aux besoins des utilisateurs en matière d'aide à la décision et de contrôle. De cette manière, une application BI basée sur l'IoT devient très complexe du fait de la combinaison de différents systèmes d'information (IoT comme source de données et BI pour l'analyse) et de l'hétérogénéité, de l'incertitude et du volume des données IoT.

De plus, la mise en œuvre de la logique métier dans l'IoT est généralement un processus long et sujet aux erreurs qui empêche les experts de l'IoT de répondre à des exigences complexes qui

peuvent varier constamment. Ces problèmes de source de données limitent également la capacité des experts BI à fournir efficacement l'analyse appropriée. Par conséquent, les applications BI basées sur l'IoT ne se comportent pas toujours comme prévu par les décideurs et les experts du domaine (c'est-à-dire les utilisateurs professionnels) malgré les efforts des experts.

Pour ces raisons, les méthodologies d'ingénierie logicielle modernes (telles que Model-Driven) sont bien connues et presque obligatoires dans le développement de systèmes logiciels et matériels complexes (tels que les applications BI basées sur l'IoT). Les méthodologies pilotées par les modèles fournissent un cadre de développement logiciel axé sur la définition de modèles conceptuels qui se transforment automatiquement en code implémentable. De plus, certaines approches telles que la Model-Driven Architecture (MDA) définissent une séparation claire des vues et des niveaux d'abstraction. Les concepteurs peuvent utiliser les niveaux distincts pour définir différents aspects de leurs systèmes, tels que les exigences, la logique et la mise en œuvre.

L'objectif principal des applications BI basées sur l'IoT est d'acquérir des données spécifiques à la demande pour fournir des analyses personnalisées qui se traduisent par une prise de décision efficace. Par conséquent, un modèle conceptuel efficace pour ces applications doit être centré sur les données.

Les différents niveaux de MDA peuvent prendre en charge efficacement la conception et le développement d'applications BI basées sur l'IoT avec des modèles de données hautement abstraits et des modèles de mise en œuvre spécifiques au système. En effet, plusieurs auteurs ont fourni des approches basées sur des modèles pour les systèmes BI et IoT. Néanmoins, les propositions dans le domaine de la BI ignorent souvent les difficultés d'intégration des données IoT et ne se concentrent pas sur la génération de code pour ce sous-système. De même, les travaux sur l'IoT mettent généralement l'accent sur la définition des réseaux, ne fournissant aucun modèle de données conceptuel pour l'intégration et l'utilisation des données.

Par conséquent, la conception et le développement d'applications IoT centrées sur les données et d'applications BI basées sur l'IoT ne reposent pas sur des méthodologies basées sur des modèles. Cette situation entrave la construction des systèmes et leur capacité à extraire de la valeur et des informations à partir des données générées par l'IoT.

Sur la base de ce scénario, cette thèse se concentre sur la résolution de la question de recherche suivante : **Comment faciliter l'exploitation des données générées par l'IoT ?**

C.2 Concepts Principaux

Cette section fournit une description concise des principales techniques, méthodes et technologies permettant de définir un IoT centré sur les données et piloté par un modèle. Premièrement, il définit clairement le capteur, les données de capteur et l'application des données de capteur, en particulier compte tenu du contexte de cette thèse. Dans un second temps, il décrit l'IoT, en étendant les concepts de capteurs et en précisant comment il est utilisé dans cette thèse. Enfin, il explique la modélisation conceptuelle, en se concentrant sur les modèles conceptuels de données et leur représentation en UML.

C.2.1 Capteurs

La littérature existante fournit plusieurs définitions d'un capteur, d'un matériau particulier qui modifie ses caractéristiques physiques en présence de certains stimuli à des systèmes complexes qui exploitent ces caractéristiques pour fournir des données détaillées sur les phénomènes, objets ou événements pertinents. Même dans un contexte particulier comme l'agriculture, où cette thèse valide ses apports, le concept de capteur est ambigu et vaguement défini.

Par conséquent, cette thèse suit les définitions de pour affirmer qu'un dispositif de capteur est un élément matériel programmable qui dispose de l'équipement nécessaire pour :

- Échantillonnez et numérisez les données d'un phénomène, d'un objet ou d'un événement pertinent.
- Faire des calculs sur les données numériques.
- Et communiquer les données (brutes ou calculées).

En particulier, cette thèse porte sur des capteurs qui peuvent au moins stocker et agréger les données collectées et les délivrer via un protocole de communication filaire ou sans fil. Ces fonctionnalités permettent d'utiliser des capteurs comme des systèmes d'information modulaires et indépendants qui peuvent également contribuer en tant que sources de données dans des structures plus complexes.

Les données des capteurs, c'est-à-dire les échantillons numériques collectés, (éventuellement) calculés et délivrés par un seul capteur, peuvent fournir des informations pertinentes sur un sujet observé. En effet, les applications de surveillance reposent généralement sur des capteurs pour acquérir les données nécessaires.

En outre, les données des capteurs sont encore plus avantageuses au-delà des applications de surveillance. Les applications basées sur des capteurs exploitent les données des capteurs pour extraire de meilleures informations du sujet surveillé et obtenir des avantages. Souvent, ces avantages sont des actions automatiques qui affectent directement le sujet surveillé. De même, les applications basées sur des capteurs peuvent fournir de nouvelles bases de connaissances ou permettre une planification stratégique pour la prise de décision.

En ce sens, toutes les données de capteur ne sont pas utiles pour extraire des informations pertinentes. La valeur des données du capteur dépend de plusieurs conditions allant de son positionnement à la qualité des composants matériels et micrologiciels. La configuration du capteur qui lui permet de fournir les données appropriées est appelée une application de données de capteur.

Un traitement approprié peut augmenter la valeur des données indépendamment de la disposition physique du capteur dans une application de données de capteur. En effet, l'agrégation de données est essentielle pour tout système d'information car elle permet la découverte de modèles pertinents, l'extraction d'informations et la réduction de l'espace de stockage et des coûts énergétiques. Par conséquent, un capteur doit au moins être capable d'agréger ses données collectées.

Le composant qui définit le traitement, l'acquisition et la livraison des données dans les applications de données de capteur est le firmware ou micrologiciel. Plusieurs approches existent pour développer un firmware. Par exemple, le langage d'assemblage, les environnements de développement intégrés dédiés, les systèmes d'exploitation embarqués, les conceptions par glisser-déposer ou le développement piloté par modèle. Néanmoins, tous ne sont pas largement utilisés ou ne conviennent pas aux applications courantes. Cette thèse se concentre sur les systèmes d'exploitation embarqués (ou systèmes d'exploitation embarqués) car ils fournissent généralement une programmation standard pour plusieurs types d'appareils.

Systèmes d'exploitation embarqués

Les capteurs sont des systèmes embarqués, c'est-à-dire des machines commandées par ordinateur. Par conséquent, l'ordinateur doit gérer les composants matériels avec précision pour répondre aux exigences de l'application. Dans ce contexte, les systèmes d'exploitation embarqués résument la gestion physique de ces fonctionnalités (par exemple, les registres, le convertisseur analogique-numérique, l'allocation de mémoire et le temps de lecture d'une sonde de détection) en tant qu'API accessibles qui facilitent la configuration de l'ordinateur interne et donc du système embarqué.

Cette caractéristique permet aux systèmes d'exploitation embarqués de fournir une API standard pour tout périphérique matériel qu'ils prennent en charge. Par exemple, Contiki-NG prend en charge quatre architectures matérielles différentes, tandis que RIOT en prend en charge cinq et Zephyr six. Par conséquent, le même code doit s'exécuter de manière transparente sur plusieurs appareils de différents fournisseurs s'ils prennent en charge le même système d'exploitation intégré et disposent des mêmes fonctionnalités. Par exemple, un capteur qui n'a que des sondes de température ne peut pas fournir de données de pluie et vice-versa.

Néanmoins, les différences entre les systèmes d'exploitation embarqués vont au-delà de l'API. Ils ont également des architectures de conception particulières, des algorithmes de planification, des modèles de programmation et un support réseau. Par conséquent, la migration d'un système d'exploitation embarqué vers un autre représente un effort important car cela change le paradigme de la programmation.

Dans les travaux de cette thèse, nous utilisons deux systèmes d'exploitation embarqués en tenant compte de l'accessibilité aux capteurs de support et de la flexibilité du modèle de programmation :

- *Simple Embedded OS (SEOS)*, développé par le LIMOS (UMR 6158 CNRS - Université Clermont Auvergne, France). Parmi les appareils pouvant exécuter SEOS, nous avons utilisé des cartes uSu EDU (UEB), également développées par LIMOS. Ces plates-formes polyvalentes comprennent diverses sondes utiles, comme la température de l'air et l'humidité relative de l'air, et des interfaces de communication telles que IEEE 802.15.4 par défaut. En outre, ils peuvent ajouter de manière transparente d'autres modules à usage spécifique (par exemple, des sondes d'humidité du sol). Les UEB et SEOS sont utilisés

pour la plupart des exemples dans les Chapitres 4 et 5, et certaines expériences dans le Chapitre 6.

- *RIOT*, conçu par une alliance entre la Freie Universität Berlin, l'INRIA France et l'Université des Sciences Appliquées de Hambourg. Parmi les appareils pouvant exécuter RIOT, nous avons utilisé trois plates-formes de détection disponibles dans le banc d'essai FIT IoT-LAB : IoT-LAB M3, Microchip SAMR21 Xplained Pro et Arduino Zero. Ces appareils peuvent détecter différentes variables (par exemple, la température de l'air, la lumière, la pression atmosphérique) et prendre en charge les communications sans fil (IEEE 802.15.4) et filaires (UART). Ces dispositifs sont utilisés pour certaines expériences du Chapitre 6.

C.2.2 Internet des Objets (IoT)

Lorsque l'évolution des systèmes embarqués a permis de les contrôler et d'accéder à leurs données via Internet en temps réel, le concept d'IoT est né. De nos jours, l'IoT fait généralement référence à d'énormes réseaux d'appareils interconnectés qui partagent des informations, des capacités de traitement et l'exécution de tâches dans le monde entier, tirant parti d'une connectivité Internet rapide avec des technologies de communication sans fil de pointe. Cependant, des systèmes plus simples tels que les réseaux de capteurs sans fil (WSN) ou même un seul appareil connecté à Internet sont également considérés comme IoT.

Ainsi, une application de données de capteur fait également partie de l'IoT. Cependant, la plupart des applications de données IoT sont plus complexes, impliquant plusieurs capteurs dotés de capacités de traitement et de communication avancées (par exemple, des passerelles). En effet, la principale différence entre les données de capteur et les données IoT est que ces dernières font référence aux données acquises, calculées et fournies par divers appareils, où l'un des appareils participants peut traiter une partie (ou la totalité) des données (c'est-à-dire un traitement distribué).

La complexité accrue des applications de données IoT augmente également les défis et les possibilités des applications basées sur l'IoT. Même si le sous-système d'acquisition de données devient une source de données, le nombre d'appareils qui participent, leur disposition et leurs rôles (en particulier pour l'acquisition et le traitement des données) affectent les données de sortie et peuvent limiter son analyse ultérieure. En outre, la variété des données dans l'IoT permet d'extraire des informations plus précieuses et permet l'analyse du Big Data et la science des données.

Parmi la vaste gamme d'applications de données IoT, cette thèse se concentre sur le WSN car il s'agit de l'une des implémentations les plus courantes pour l'acquisition de données pures, en particulier dans l'agriculture.

Réseaux de Capteurs Sans Fil (WSN)

Un réseau de capteurs se compose de capteurs inter communicants (et éventuellement de dispositifs plus complexes comme des passerelles) qui partagent un objectif de surveillance

commun. De cette manière, les capteurs partagent leurs données collectées sur tout le réseau, où chaque nœud (appareil connecté) joue un rôle spécifique et des tâches assignées, par exemple détecter, traiter, transmettre ou télécharger les données.

Lorsque la transmission de données inter-nœuds repose sur des protocoles de communication sans fil, elle devient un WSN. Les applications de données IoT mises en œuvre en tant que WSN peuvent couvrir de manière abordable des zones plus vastes avec un accès difficile. Par conséquent, plusieurs domaines d'application, de l'armée à l'agriculture, exploitent les WSN pour acquérir leurs données.

La définition des rôles et la répartition des nœuds dépendront des exigences de l'application. Les WSN ont plusieurs options pour organiser les nœuds, appelées topologies de réseau : *Bus*, *Tree*, *Star*, *Mesh*, *Ring*, *Circular* ou *Grid*.

A partir de ces options variables, cette thèse s'intéresse à la topologie en *Star*. Les nœuds de cette topologie ont deux rôles : capteur ou récepteur. Le nœud de capteur collecte, traite et envoie des données au récepteur. Le récepteur reçoit, traite, joint et télécharge les données WSN. Habituellement, ce nœud peut également détecter des données. La topologie en *Star* est relativement simple (compte tenu des topologies *Circular* et en *Grid*) ; pourtant, il offre un sujet d'étude convaincant puisqu'il a au moins deux rôles différenciés et une structure hiérarchique avec des possibilités ouvertes.

Habituellement, les exigences particulières de chaque rôle exigent différents appareils pour y répondre. Par conséquent, les nœuds récepteurs ont souvent un hardware plus robuste, avec des capacités d'énergie, de traitement, de stockage et de communication accrues que les nœuds capteurs. Ces WSN sont appelés réseaux hétérogènes. Néanmoins, les appareils bas de gamme peuvent répondre aux exigences de tous les rôles, en organisant un WSN homogène. Par souci de simplicité, cette thèse se concentre sur les WSN homogènes.

C.2.3 Modélisation conceptuelle

Un modèle conceptuel est une représentation abstraite, partielle et simplifiée d'un système à l'étude qui permet l'analyse et le raisonnement sans la limitation de la mise en œuvre. Les modèles permettent de partager une vision collective entre les parties techniques (par exemple, les ingénieurs IoT et de données) et non techniques (par exemple, les experts du domaine et les décideurs), facilitant la communication entre eux. Par conséquent, il fournit des processus de conception, de développement et de maintenance efficaces et efficients pour les systèmes complexes ; et un contrôle de projet impartial et précis.

De plus, les méta-modèles permettent de définir des modèles d'instance représentant des implémentations particulières d'un domaine plus large. Ainsi, un méta-modèle peut être considéré comme un "modèle qui définit la structure d'un langage de modélisation".

Les langages de modélisation formels, tels que UML ou ER, fournissent des notations standard (c'est-à-dire un méta-modèle) pour la définition des modèles. Cette représentation permet une large compréhension et utilisation des modèles et fournit des lignes directrices pour leur construction appropriée. En effet, dans la littérature, les modèles qui n'adhèrent pas aux langages de modélisation formels sont souvent ignorés.

De plus, les modèles en génie logiciel peuvent apporter plus de valeur au-delà de l'abstraction du système. Les approches pilotées par les modèles tirent parti de la formalisation et de la définition des modèles conceptuels pour faciliter le processus de développement, générant généralement une partie importante du code requis.

Cette thèse propose un modèle de données conceptuel pour les applications de données de capteurs et IoT pilotées par modèle. Un profil UML formalise la représentation des données, suivant les principes de MDA pour fournir une approche complète basée sur un modèle.

Modèle de données conceptuel

Un modèle de données conceptuel est la représentation méta-modèle d'un type particulier de données (par exemple spatiale, temporelle, graphique), où chaque élément du méta-modèle définit une notation graphique respective. Habituellement, les modèles de données conceptuels sont livrés avec des systèmes CASE (Computer-Aided Software Engineering) qui permettent (entre autres fonctionnalités) une interface graphique assistée et une génération de code automatique. Ces modèles permettent aux utilisateurs non techniques de se concentrer sur les exigences en matière de données et d'analyse, en faisant abstraction de la complexité des technologies sous-jacentes.

Les modèles conceptuels sont largement utilisés pour représenter les données transactionnelles stockées dans les bases de données, au moyen d'UML et d'ER. Cependant, ils sont également utilisés dans des contextes plus complexes.

Par exemple, dans le contexte de la BI, plusieurs travaux proposent l'utilisation d'extensions ER et UML pour représenter les entrepôts de données car ils véhiculent les capacités d'analyse offertes par une simple représentation des données. Les modèles de données conceptuels sont également avantageux pour les applications basées sur l'IoT (par exemple, les systèmes Stream et Big Data), y compris la définition et l'intégration de ces données.

Profil UML

Un méta-modèle formel est aussi une extension de la structure et des notations de son langage de modélisation. En UML, ces extensions peuvent être lourdes (lorsque la sémantique change en modifiant le méta-modèle UML standard) ou légères (lorsque la sémantique change mais suit toujours le méta-modèle UML standard). Les extensions légères (c'est-à-dire les profils) préservent la norme UML et sa large acceptabilité. De plus, ils sont plus faciles à apprendre et à comprendre.

Le but des profils UML est de personnaliser UML pour des domaines ou des plates-formes particuliers en étendant ses méta-classes (par exemple, Classe, Propriété, Package). Un profil est défini à l'aide de trois concepts clés : les stéréotypes, les valeurs étiquetées et les contraintes. Les stéréotypes étendent directement les méta-classes et peuvent être représentés par l'inscription « nom-stéréotype » ou une icône. Par exemple, un stéréotype « Espèce » étend la méta-classe UML « Class » pour définir la représentation des espèces. Au niveau du modèle, chaque classe utilisant ce stéréotype désignera un groupe d'organismes vivants. Les valeurs étiquetées (ou

simplement les balises) sont des méta-attributs, c'est-à-dire qu'elles sont définies comme des propriétés statiques de stéréotypes. Par exemple, le règne d'une espèce n'est pas censé changer et peut être considéré comme un méta-attribut. Enfin, un ensemble de contraintes doit être attaché à chaque stéréotype, définissant précisément sa sémantique d'application pour éviter une utilisation arbitraire par les concepteurs dans les modèles. De telles règles peuvent être définies dans le langage OCL (Object Constraint Language), que les outils CASE peuvent automatiquement appliquer aux modèles d'instance. Par exemple, une règle OCL peut garantir que toute classe « Espèce » a un attribut descriptif appelé "race", et l'outil CASE alertera lorsque cette règle est enfreinte.

Architecture pilotée par les modèles (MDA)

Les approches pilotées par les modèles considèrent les modèles conceptuels comme des citoyens de premier ordre dans le développement de logiciels au-delà de leur rôle toujours pertinent pour la documentation et l'implication des utilisateurs. MDA se distingue de ces approches par son haut niveau d'abstraction et de spécification de l'architecture de modélisation. L'objectif de MDA est de permettre une adoption transparente et large des modèles pour les applications. Par conséquent, ses modèles sont exprimés dans une notation bien définie, divisée en préoccupations et niveaux d'abstraction hiérarchiques, et étayée par des méta-modèles formels. En effet, le profilage UML est presque la pratique standard pour les propositions MDA.

La séparation des préoccupations indique que chaque modèle doit répondre à un problème unique du système à l'étude. Par exemple, les données, la sécurité et la planification sont toutes pertinentes pour les systèmes IoT, mais un modèle qui aborde tous les sujets devient trop complexe et difficile à comprendre. En outre, la division hiérarchique des niveaux d'abstraction fait référence aux différents niveaux d'indépendance de l'implémentation particulière. MDA propose cinq niveaux d'abstraction :

1. Modèle indépendant du calcul (CIM) : ce niveau ne doit comprendre que les besoins des utilisateurs sans tenir compte des limitations technologiques. Cette thèse n'aborde pas le CIM car il est limité aux applications de données de capteurs et IoT et suppose que les utilisateurs peuvent discuter directement des données au niveau du PIM.

2. Modèle indépendant de la plate-forme (PIM) : le niveau d'abstraction élevé de l'application modélise sa logique principale et ses attributs, quelle que soit la plate-forme sous-jacente. Dans cette thèse, le PIM représente les données des capteurs et de l'IoT et leurs processus de collecte, de transformation et de livraison.

3. Modèle spécifique à la plate-forme (PSM) : le niveau d'abstraction faible de l'application modélise son implémentation dans une plate-forme spécifique. Dans cette thèse, le PSM relie les données aux installations, rôles, tâches et dispositions particuliers des appareils IoT.

4. Modèle de plate-forme (PM) : modélise la plate-forme de mise en œuvre indépendamment de l'application. Cette thèse appelle le PM comme Device Model (DM) ; il représente les appareils IoT.

5. Code d'implémentation : bien que le code ne soit pas un modèle pour MDA, il implémente la logique et les attributs de l'application dans la plate-forme. Ainsi, le code est la réalisation ultime d'un processus piloté par un modèle. Cette thèse permet la génération automatique de code de capteurs à partir des modèles DM (ou PM) et PSM.

Notez que le PIM est plus lié aux exigences fonctionnelles (FR) des applications, tandis que le PSM doit traiter des exigences non fonctionnelles (NFR) plus spécifiques. Les FR énoncent les fonctionnalités du système qui répondent aux besoins des utilisateurs finaux. Ainsi, les FR guident la définition et la validation de l'architecture et de la logique du système (c'est-à-dire le PIM). Par ailleurs, les NFR complètent les FR avec les contraintes et qualités attendues de chaque fonctionnalité. De cette manière, les caractéristiques techniques de mise en œuvre du système (c'est-à-dire le PSM) reposent sur les NFR. Dans le contexte des applications de données de capteurs et IoT, les FR font référence aux données, aux calculs et à l'analyse (par exemple, la température moyenne quotidienne) ; et les NFR à la fiabilité, la topologie et la technologie associées (par exemple, prélever les échantillons de trois points en utilisant des nœuds UEB dans une topologie en *Star*).

C.2.4 Systèmes d'intelligence d'affaires (BI) : entrepôt de données

Les systèmes de BI sont des citoyens de premier ordre des systèmes d'aide à la décision. Ils permettent l'exploration et l'analyse de données stockées dans des bases de données transactionnelles. Les systèmes de BI comprennent les systèmes de rapport, d'exploration de données, statistiques et entrepôt de données-OLAP.

Les systèmes d'entrepôt de données (DW) et OLAP sont les principaux outils de BI. Ils permettent l'analyse d'énormes volumes de données stockées selon le modèle multidimensionnel. Les données stockées sont organisées en faits (sujets d'analyse) et en dimensions (axes d'analyse). Les dimensions sont composées de hiérarchies qui permettent d'analyser les données à différentes granularités thématiques, spatiales et temporelles. Les faits sont décrits par des attributs numériques appelés mesures, qui sont agrégés selon des hiérarchies avec des fonctions d'agrégation SQL courantes telles que Min, Max ou Somme. Les systèmes OLAP permettent d'explorer et d'agréger des mesures via des opérateurs OLAP. Les opérateurs OLAP les plus courants sont *Roll-Up* et *Drill-Down*, qui permettent d'escalader et d'agréger des mesures sur des niveaux de dimension, et *Slice* and *Dice*, qui permettent de sélectionner un sous-ensemble des données stockées. Habituellement, les DW sont implémentés à l'aide d'un SGBD relationnel classique utilisant l'approche du schéma en étoile, qui dénormalise les données dimensionnelles pour améliorer les performances des requêtes. Lorsque les données sont intégrées en temps réel ou proviennent de sources de flux, les DW sont nommés respectivement temps réel et flux DW.

Étant donné que les systèmes de BI sont construits au-dessus des sources de données, leur représentation conceptuelle s'ajoute à celle utilisée pour représenter les sources de données.

C.3 Travaux Connexes

L'adoption de techniques basées sur des modèles pour les applications de capteurs et d'IoT a gagné en pertinence au cours de la dernière décennie. Les approches les plus courantes ont été MDE et MDA. Ces approches tirent parti des capacités de modélisation d'UML et de SysML ou de la capacité d'adaptation des DSL et des profils UML pour leur définition.

Les propositions dans ce domaine se concentrent sur la création d'applications IoT à partir d'un niveau conceptuel. La capacité de génération de données de l'IoT est un sujet commun à plusieurs travaux. En effet, le mot "*data*" est parmi les mots les plus courants dans les résumés en plus de l'information, de l'internet et de l'IoT. Cependant, notre analyse de la littérature révèle que les auteurs n'ont pas entièrement abordé la modélisation des données des systèmes IoT pour faciliter leur mise en œuvre en utilisant une approche basée sur les modèles.

Même si plusieurs travaux définissent certains concepts principaux pour les données de capteurs et IoT pour générer le code des applications, leur représentation des données n'est pas exhaustive. En effet, seuls cinq articles modélisent trois caractéristiques de données de capteur ou plus et génèrent le code requis. Cependant, aucun de ces articles ne fournit une représentation complète des données IoT.

Par conséquent, nous n'avons trouvé aucun travail fournissant des modèles de données conceptuels pour l'IoT qui génèrent le code d'application dans une approche basée sur un modèle. Cette situation réduit la capacité des utilisateurs professionnels et des experts en données à influencer la conception des applications de données IoT, réduisant ainsi la valeur perçue des modèles existants dans les applications intégrées complexes.

C.4 Problèmes de Modélisation

Les applications basées sur les capteurs et l'IoT sont généralement des applications centrées sur les données. Par exemple, dans un système de BI tel que celui de l'étude de cas, les bases de données OLTP (OnLine Transaction Processing) sous-tendent le rapport et les DW prennent en charge OLAP (OnLine Analytical Processing). Par conséquent, une représentation efficace du modèle conceptuel de données est nécessaire pour les applications centrées sur les données. En effet, les modèles de données traditionnels peuvent être complexes, impliquant plusieurs types de données (spatiales, temporelles ou thématiques). Les données IoT augmentent la complexité des modèles de données car elles intègrent des données spatio-temporelles continues provenant de capteurs (par exemple, la température détectée) aux données et analyses classiques.

Habituellement, la conception conceptuelle des applications traditionnelles centrées sur les données implique des utilisateurs professionnels et des scientifiques des données. La conception conceptuelle des applications basées sur l'IoT implique également des experts de l'IoT, qui répartissent les responsabilités pour les différentes parties de ces applications. Les utilisateurs métier interagissent avec des scientifiques des données et des experts IoT pour définir les besoins en données et en analyse (c'est-à-dire les exigences fonctionnelles). En outre, les scientifiques des données et les experts IoT se concentrent sur leurs implémentations respectives (applications

BI et IoT) et sur le NFR.

Même de légères variations dans les équipes peuvent générer des problèmes de compatibilité, de communication ou de performances dans le système. Dans ce contexte, la modélisation conceptuelle des données est un support efficace et obligatoire pour formaliser le processus de définition et de développement, en fournissant des représentations graphiques qui résument les détails techniques de mise en œuvre et permettent l'implication de tous les rôles. Les modèles conceptuels pour les applications basées sur l'IoT doivent être particuliers et systémiques. Particulier puisque chaque sous-système a des problèmes spécifiques que l'expert respectif doit résoudre. Et systémique puisque les deux sous-systèmes doivent coopérer pour atteindre un objectif commun.

En outre, une approche basée sur MDA définit un modèle de données hautement abstrait à discuter entre les utilisateurs professionnels, les scientifiques des données et les experts de l'IoT ; tout en fournissant un modèle d'implémentation spécifique et la génération (semi) automatique de code implémentable. Néanmoins, il existe plusieurs approches MDA centrées sur les données pour les systèmes de BI et de bases de données dans la littérature.

Par conséquent, cette section présente les (nouvelles) exigences pour des modèles conceptuels efficaces d'applications de données de capteurs et d'IoT illustrées dans l'étude de cas. De plus, cela explique le besoin de support MDA dans ces approches de conception.

C.4.1 Principaux concepts pour concevoir des applications de données de capteurs

Compte tenu des caractéristiques des données de capteur et des applications de données de capteur, un modèle de données conceptuel pour ce type de système devrait se concentrer sur quatre aspects des données :

- Le type de variable détectée (par exemple, la température) puisque chaque variable a des objectifs d'utilisation, des sources et des calculs particuliers.
- La validité temporelle des données puisque de nouvelles valeurs sont toujours captées et transmises, remplaçant les anciennes et déterminant ainsi une période de validité. En ce sens, la validité temporelle dépend des fréquences de détection et d'émission.
- Les opérations qui transforment les données brutes (par exemple, l'agrégation) car elles limitent le traitement ultérieur des données de capteur. Par exemple, calculer la moyenne à partir d'un ensemble de moyennes est inapproprié. Notez qu'un modèle d'application de données de capteur doit au moins représenter l'agrégation de données.
- Les fenêtres temporelles sur lesquelles les opérations peuvent calculer un ensemble fini de données.

De plus, les modèles de données doivent être lisibles pour tous les rôles, y compris les utilisateurs non techniques. Une bonne lisibilité augmente l'implication des décideurs puisqu'ils

peuvent se concentrer sur les besoins en données et leur analyse plus approfondie. Il doit donc faire abstraction de la complexité des caractéristiques techniques des systèmes de mise en œuvre ; par exemple, l'utilisation d'une interface de capteur et d'un protocole particulier ou la dénormalisation des données utilisées par DW. De la même manière, il devrait permettre une intégration transparente des données des capteurs et des données classiques ; par exemple, associer les données de température détectées aux données de parcelle et de culture correspondantes pour leur analyse.

Par conséquent, l'approche MDA devrait :

- Définissez clairement le problème de modélisation, les données IoT dans ce cas, car des concepts supplémentaires, bien que pertinents dans la mise en œuvre réelle, réduisent la lisibilité et augmentent la complexité.
- Séparez le modèle de données en différents niveaux d'abstraction, permettant à tous les rôles de faire converger leurs efforts pour la définition du FR tout en permettant à chaque expert de se concentrer sur la mise en œuvre de ses systèmes et le NFR.
- Fournit des possibilités d'intégration de données classiques et IoT au niveau d'abstraction élevée. En effet, étant donné que les utilisateurs métier doivent analyser les données des capteurs pour en extraire de la valeur, un modèle de données conceptuel des applications de données de capteur doit également permettre une intégration facile avec les modèles de données BI.
- Définissez des règles de transformation de modèle à modèle, ce qui facilitera davantage le processus de développement.
- De même, définissez un processus de génération de code automatique et sans erreur à partir des modèles.

C.4.2 Principaux concepts pour concevoir des applications de données IoT

Au-delà des concepts de modélisation des données de capteurs, qui sont toujours pertinents, les applications de données IoT ont un degré de complexité plus élevé dérivé de l'association de plusieurs fonctionnalités. Premièrement, les données IoT évoluent sur des architectures de réseau continu où les objets échangent des données dans une organisation physique progressive impliquant différents types d'appareils qui joignent, transforment et renvoient des données sur Internet. Deuxièmement, les appareils IoT ne peuvent calculer que les données qu'ils gèrent (c'est-à-dire qu'ils détectent ou reçoivent). Cependant, le calcul centralisé des données est généralement coûteux et ne tient pas compte des capacités des appareils individuels. L'IoT doit ensuite coordonner les opérations de données à travers les différentes couches de l'architecture du réseau pour obtenir un calcul global cohérent. Et troisièmement, l'IoT comprend également des appareils avancés qui peuvent traiter des données au-delà des fonctions d'agrégation.

Par conséquent, un modèle de données conceptuel pour les applications de données IoT a des exigences supplémentaires concernant les transformations de données, les communications de données et le MDA :

- Les transformations de données à l'intérieur de l'IoT doivent inclure au moins trois transformations de prétraitement principales :
  - L'agrégation, qui permet de synthétiser un ensemble de données pour en augmenter la valeur ; il s'agit d'une transformation cruciale dans les systèmes IoT qui nécessite la création de fenêtres temporelles.
  - La conversion permet de changer les types et le format des données (par exemple, différentes unités) sans altérer la signification intrinsèque des données.
  - Et le filtrage, qui supprime les mesures indésirables ou potentiellement erronées.
  - De plus, le modèle conceptuel devrait être facilement extensible pour inclure plus de transformations.
- Un modèle de communication de données entre objets IoT doit présenter trois caractéristiques principales pour une représentation correcte de la transmission et de l'intégration des données :
  - La représentation explicite de la jointure des flux de données provenant de sources externes (et internes).
  - La capacité de définir des associations de composition de données provenant de différentes sources.
  - Et la représentation abstraite du réseau IoT.
- Enfin, le MDA doit permettre de concevoir plusieurs options de mise en œuvre des mêmes données. En outre, il doit prendre en charge la génération de code de chaque option d'implémentation (correctement définie).

C.5 Étude de Cas

L'agriculture est une activité économique complexe et instable, vitale pour l'humanité. Parmi les multiples applications de l'IoT en agriculture de précision, cette thèse s'intéresse à l'irrigation intelligente pour contextualiser et motiver ses apports. Les activités d'irrigation dépendent fortement de la variété des cultures et des conditions environnementales du champ ou de la parcelle en particulier. Ainsi, les systèmes d'aide à la décision basés sur les capteurs et l'IoT apportent une aide significative aux agriculteurs et aux agronomes.

La méthode IRRINOV© énonce trois variables environnementales qui permettent d'estimer le manque d'eau d'une culture particulière et ainsi indiquer si elle a besoin d'être irriguée. Les variables d'intérêt sont :

- Humidité médiane quotidienne du sol échantillonné une fois par jour à partir d'au moins trois sondes de mesure pour chaque profondeur (30 et 60 cm).
- Maximum et minimum quotidiens de la température de l'air échantillonnée toutes les 20 minutes. Ces mesures fournissent une accumulation de chaleur quotidienne et permettent de calculer l'unité de degré de croissance (GDU).
- Et les précipitations accumulées quotidiennement.

Les appareils IoT peuvent fournir ces variables pour éviter les problèmes, les coûts et les limites de la collecte manuelle. De plus, un système de rapport de flux de données IoT permettrait aux agriculteurs de visualiser les besoins d'irrigation par culture et parcelle en temps réel. En effet, un système BI peut proposer certaines actions d'irrigation aux agriculteurs. La règle de décision est : "Lorsque l'humidité du sol est supérieure à un seuil de 'manque d'eau dans le sol', ils doivent commencer l'irrigation". Le niveau de carence en eau dépend de la croissance des cultures (c'est-à-dire GDU). Par exemple, dans le cas du maïs, si le GDU est supérieur ou égal à 570 degrés-jours, le seuil de "manque d'eau dans le sol" est de 160 cbar. Ainsi, l'application BI proposera d'irriguer la parcelle de maïs spécifique si l'humidité du sol agrégée est de 161 cbar. Cependant, si la quantité de pluie est supérieure à 10 mm, l'irrigation est inutile.

L'application BI doit vérifier la règle de décision décrite ci-dessus à différents niveaux spatiaux, temporels et thématiques. En particulier, l'application doit fournir un suivi continu des données climatiques captées selon ces trois axes d'analyse (c'est-à-dire les dimensions) :

- Spatial : intra-parcelle (au niveau du capteur), parcelle et ferme pour optimiser et prioriser l'utilisation de l'équipement d'irrigation sur la ferme.
- Thématique : culture et type de plante pour fournir la quantité d'eau appropriée.
- Temps : jour et semaine pour suivre les activités d'irrigation au fil du temps.

Ce type de système BI basé sur l'IoT peut être un entrepôt de données (DW) en temps réel qui analyse en continu des capteurs ou des données IoT (c'est-à-dire le fait) selon ces trois dimensions, organisées de manière hiérarchique. Par conséquent, au moins trois types d'experts participent et coopèrent à la conception et à la mise en œuvre de l'application BI basée sur l'IoT : les agronomes et les agriculteurs (utilisateurs professionnels), qui connaissent le contexte agricole et peuvent extraire de la valeur de l'analyse ; les experts en BI ou scientifiques des données, qui développent et utilisent le DW (c'est-à-dire le sous-système d'analyse de données) ; et les experts IoT, qui configurent et mettent en œuvre l'infrastructure de détection (c'est-à-dire le sous-système d'approvisionnement en données).

Le développement et la mise en œuvre du système IoT est complexe car il concerne des problèmes de données, de capteurs et de modélisation de réseau. Par exemple, la figure 2.2 affiche quatre options de mise en œuvre pour cette application IoT en tenant compte de certaines contraintes matérielles et contextuelles, des données collectées et des opérations sur les données. Ces configurations sont assez courantes dans le contexte agricole, où des capteurs sont déployés

dans les champs pour communiquer leurs données via une application décisionnelle déployée sur le cloud. De plus, comme la plupart des capteurs ont rarement une connexion Internet, ils doivent être connectés au cloud via une autre couche de communication et de calcul déployée au niveau de la ferme, selon l'architecture bien connue edge-fog-cloud adoptée pour l'IoT.

Ces topologies en *Star* utilisent deux types de nœuds capteurs :

- Les nœuds récepteurs (SN) (tels qu'une station météorologique et un ordinateur portable), qui disposent de l'alimentation électrique, de la connexion Internet et d'importants outils de calcul et de stockage ;

- Nœuds terminaux ou capteur (EN) (tels que les capteurs d'humidité du sol), qui sont alimentés par batterie, sans connexion Internet (mais prenant en charge certains protocoles sans fil) et avec de faibles outils de calcul et de stockage.

Dans toutes les options de configuration, le point d'accès Internet se trouve dans la maison de l'agriculteur, et donc un nœud (capteur ou récepteur) y est positionné pour télécharger les données dans le cloud. De plus, les données de température, de pluie et d'humidité du sol sont collectées et téléchargées dans le cloud conformément au guide IRRINOV. Néanmoins, il existe des différences pertinentes entre chaque option.

Cette étude de cas montre la complexité et la diversité des configurations possibles d'une même application de données IoT, ce qui impacte directement les données et les coûts de ces applications.

Par conséquent, une conception très abstraite est cruciale pour que les agriculteurs et les experts en agronomie se concentrent exclusivement sur les exigences fonctionnelles des applications (c'est-à-dire leurs besoins en données et en analyse). En outre, la modélisation de niveau inférieur et la transformation du modèle en code sont essentielles pour que les experts de l'IoT définissent les problèmes techniques les plus pertinents (par exemple, le rôle et l'affectation de chaque nœud) et parviennent de manière transparente à une implémentation réussie.

C.6 Conception et Mise en œuvre d'Applications de Données de Capteurs

Cette section résume la modélisation des applications de données de capteurs basées sur un dispositif de détection et l'agrégation de données. Pour commencer, il décrit la conception et la mise en œuvre des données de capteurs, puis il détaille comment ces modèles de données de capteurs peuvent être intégrés dans des modèles conceptuels de BI.

Le profil UML proposé comporte deux parties principales : (i) le profil de modèle de dispositif de capteur (SensorDeviceModel) et (ii) le profil de modèle d'application de données de capteur (SensorDataModel) ; qui sont ensuite déclinées selon les principes MDA en modèles Platform-Independent (PIM) et Platform-Specific (PSM).

Le SensorDeviceModel représente le capteur d'implémentation (par exemple l'UEB) en permettant la définition des données mesurables, les interfaces de communication disponibles, la gestion du temps, les variables de mémoire spécifiques au code et l'exécution de l'application.

En outre, le SensorDataModel utilise le SensorDeviceModel pour créer une représentation conceptuelle des données utilisées par l'application décisionnelle.

Le processus de conception de modèles basés sur des capteurs utilisant notre profil UML commence par les utilisateurs professionnels (décideurs et experts du domaine) qui représentent leur vision des données nécessaires (exigences fonctionnelles centrées sur les données) avec le SensorDataModel-PIM, en utilisant des instances du SensorDeviceModel -PIM. Ensuite, les experts en capteurs cartographient ces données conceptuelles dans la spécification de la plateforme avec le SensorDataModel-PSM en utilisant des instances du SensorDeviceModel-PSM, décrivant la vue de mise en œuvre des données nécessaires. Enfin, le code IoT de la plateforme de capteurs spécifique est généré automatiquement par notre outil.

C.6.1 Modèle d'application de données de capteur

Cette sous-section présente les trois niveaux d'abstraction du modèle de données de capteur : PIM, PSM et DM.

SensorDataModel-PIM

En premier lieu, le profil SensorDataModel-PIM permet aux décideurs et aux experts du domaine de définir les exigences fonctionnelles liées aux données des capteurs de manière simple, standard et lisible.

Le SensorDataModel-PIM définit <A-PIM-Data> comme package principal, qui contient les classes <A-PIM-Measure>. Ces classes décrivent uniquement les données disponibles pour l'utilisateur ou d'autres capteurs. Ainsi, il ne définit que les variables détectées ou agrégées et l'opération de livraison. Une classe <A-PIM-Measure> peut avoir au moins une ou plusieurs variables, mais elle doit avoir une opération de livraison.

Pour la livraison (opération obligatoire), deux options s'offrent :

1. <A-PIM-TupleDelivery>, qui envoie les données après qu'un nombre spécifique de tuples ou d'échantillons ont été collectés (défini comme <Period>) et définit le nombre de tuples que le nœud doit conserver comme <WindowSize>.

2. <A-PIM-TimeDelivery>, qui envoie les données périodiquement selon la <Period> et le temps <Granule> de cette <Period>. Les deux opérations définissent <WindowSize> de la même manière.

Pour les variables brutes, la <A-PIM-Measure> définit la propriété <A-PIM-Variable>. Notons que le type de la Propriété <A-PIM-Variable> doit être issu des variables du Package <D-PIM-Data> du SensorDeviceModel, obligeant les décideurs à sélectionner les types de données parmi les dispositifs disponibles (ou, le cas échéant, les experts capteurs pour augmenter les

variables prises en charge). De plus, <A-PIM-Measure> définit également la propriété <A-PIM-AggregatedVariable> pour les variables qui sont prétraitées à l'intérieur du capteur avec une opération d'agrégation.

De plus, le profil SensorDataModel-PIM définit certaines règles dans OCL pour garantir que ses instances (c'est-à-dire les modèles PIM) sont bien formées et sémantiquement cohérentes.

SensorDataModel-PSM

En second lieu, le SensorDataModel-PSM permet aux experts en capteurs de définir les détails d'implémentation liés aux données des applications exécutées sur des capteurs spécifiques, en tenant compte des exigences fonctionnelles précédemment définies dans le SensorDataModel-PIM. Puisque nous nous sommes concentrés sur les données, au centre du profil, nous avons défini le stéréotype de classe abstraite <A-PSM-Measure>, qui établit la définition de toutes les données manipulées par les capteurs. La <A-PSM-Measure> est composée d'une ou plusieurs <A-PSM-Variables> avec un <Order> spécifique. De plus, une <A-PSM-Variable> pourrait avoir un pré-traitement définissant une fonction <Aggregation> (<A-PSM-AggregatedVariable>). De plus, le type de Propriétés <A-PSM-Variable> doit être issu des variables du Package <D-PSM-Data> du SensorDeviceModel-PSM. Néanmoins, le type sélectionné doit être <KindOf> le type simple établi par les experts du domaine dans le PIM. En effet, telle qu'elle est définie, cette association <KindOf> admet que les experts en capteurs sont contraints d'utiliser le type de données sélectionné par les décideurs.

Le progiciel principal pour décrire l'application dans un seul capteur est <A-PSM-Node>. Le nœud doit avoir un <ID> et une <Address> pour les communications. De plus, le <A-PSM-Node> contient un ou plusieurs packages <A-PSM-MeasurePacket>, qui regroupent différents <A-PSM-Measures> interconnectés qui aident à atteindre un objectif.

De plus, un nœud de capteur devrait avoir deux types de mesures : les mesures internes privées pour les variables détectées (<A-PSM-GatheredMeasure>) ; et les mesures publiques externes pour les variables livrées (<A-PSM-DeliveredMeasure>). Ces classes doivent être liées par une association <GatheredAs>.

Dans la partie gauche du profil SensorDataModel-PSM, nous avons le stéréotype de classe <A-PSM-GatheredMeasure>, une spécification de <A-PSM-Measure>. Cette classe permet la modélisation des données détectées directement à partir des sondes, qui n'existent qu'à l'intérieur du capteur et sont <Unavailable> pour les autres nœuds et les utilisateurs. Il doit définir une opération périodique pour détecter les données : <A-PSM-Gather>, ainsi que la détection <Period> et le temps <Granule> (par exemple, seconde ou milliseconde). De plus, si certaines opérations futures (par exemple l'agrégation) nécessitent de conserver plus d'une instance des données collectées, la balise <WindowSize> doit définir le nombre d'instances conservées.

De plus, les données collectées doivent être livrées afin de les rendre disponibles pour les utilisateurs et les autres nœuds capteurs.

Dans la partie droite du profil SensorDataModel-PSM, nous avons le stéréotype de classe <A-PSM-DeliveredMeasure>, une spécification de <A-PSM-Measure>. Cette classe permet de modéliser les données envoyées par le capteur aux autres nœuds ou aux utilisateurs, qui est donc

<Available>. Il doit définir une Opération périodique d'envoi des données avec deux options : <A-PSM-DeliverOnTuples>, si la <Period> est le nombre de collectes avant livraison ; ou <A-PSM-DeliverOnTime>, si la <Period> est une durée (en utilisant également la balise <Granule> pour définir la granularité temporelle). Néanmoins, les deux options de livraison doivent définir la sortie <Interface> et l'adresse <Destination>.

De plus, le nœud capteur pourrait agréger les données avant de les fournir en définissant <A-PSM-AggregatedVariable> avec <Aggregation> et l'opération <A-PSM-AggregateMeasure> correspondante.

SensorDataModel-DM

En troisième lieu, le SensorDataModel-DM permet aux experts en capteurs de modéliser les ressources disponibles (principalement des données) pertinentes pour le développement d'applications dans un dispositif de capteur spécifique, à savoir la plate-forme de mise en œuvre. En d'autres termes, le SensorDeviceModel peut être considéré comme un catalogue d'installations logicielles et matérielles de capteurs (par exemple pour l'UEB avec SEOS) qui peut fournir certaines données.

Le SensorDeviceModel tente de modéliser les principales fonctionnalités de chaque plate-forme de mise en œuvre. De cette manière, plusieurs instances de SensorDeviceModel peuvent fournir un référentiel de tous les dispositifs de capteur disponibles et de leurs capacités. Ce catalogue conceptuel permet aux experts en capteurs de trouver et d'utiliser facilement l'appareil le mieux adapté pour répondre aux exigences et contraintes spécifiques de l'application de données de capteur. Il spécifie la définition des variables PIM et PSM, les opérations PSM et d'autres détails de mise en œuvre dans le PSM. En d'autres termes, les experts en capteurs ne doivent "choisir" que parmi les SensorDeviceModels lors de la définition des modèles PIM et PSM pour spécifier les variables disponibles aux niveaux conceptuel et de mise en œuvre, respectivement.

C.7 Conception et Mise en œuvre d'Applications de Données IoT

Cette section résume la modélisation des applications de données IoT axées sur WSN. Tout d'abord, il décrit une méthodologie de conception et de mise en œuvre pour les applications de données IoT. Ensuite, il détaille le profil UML qui prend en charge la conception des données IoT.

En particulier, le profil UML pour les données IoT étend le profil de données de capteur avec des transformations de données et des communications plus complètes pour la conception de réseaux d'appareils IoT. Suivant les méthodes de conception existantes pour les systèmes ETL classiques, ce profil structure la détection, la transformation et l'envoi de données IoT dans plusieurs objets exécutant des tâches spécifiques pour un objectif commun. Ainsi, nous appelons l'ensemble de l'approche MDA (profil UML et outil de génération de code) pour les applications de données IoT STS4IoT (*Sense, Transform and Send for the Internet of Things*).

C.7.1 Méthodologie de conception de données IoT

STS4IoT suit les directives MDA, avec trois niveaux d'abstraction : PIM pour la conception des données, PSM pour la conception de l'implémentation et DM pour les informations de la plate-forme. En outre, il fournit également un système d'implémentation automatique : l'outil de modélisation à code STS4IoT.

Différents acteurs peuvent utiliser les composants STS4IoT pour définir des applications de données IoT.

- Tout d'abord, les utilisateurs métier (c'est-à-dire les décideurs et les experts du domaine) définissent leurs exigences fonctionnelles pour l'application en langage naturel : quelles données détecter, comment les acquérir, comment les transformer, et quelles données envoyer et comment les envoyer.

- Ensuite, les experts IoT traduisent ces exigences en un modèle PIM. Si les modèles DM disponibles n'offrent pas les mesures requises, ils doivent ajouter un nouveau modèle DM pour un appareil qui peut fournir les données nécessaires. Cette étape est un processus itératif qui exige une communication constante avec les utilisateurs métier.

- Une fois que le PIM respecte les exigences fonctionnelles, les experts IoT peuvent résoudre les problèmes de mise en œuvre. Ils répertorient les fonctionnalités techniques que l'implémentation doit prendre en charge, telles que les périphériques et les capteurs, la topologie et les communications. En fonction de ces problèmes techniques, les experts IoT définissent certains (un ou plusieurs) modèles PSM qui cartographient le PIM dans des implémentations particulières, en respectant à la fois les exigences fonctionnelles et les problèmes techniques. Cette étape est également un processus itératif.

- L'outil de modélisation à code STS4IoT implémente automatiquement ces modèles PSM. Le code est utilisé dans un simulateur qui simule la détection de données à des fins de test. Les experts IoT doivent redéfinir les PSM défaillants chaque fois que les simulations de test ne correspondent pas aux choix d'implémentation. Enfin, les utilisateurs professionnels reçoivent les systèmes résultants et testent les données.

- Il y a toujours un écart entre la définition conceptuelle et la mise en œuvre. Cet écart n'est pas facile à combler, même après de bons processus de communication. Par conséquent, les utilisateurs métier doivent évaluer les résultats reçus par rapport à leurs exigences initiales. Si la mise en œuvre ne correspond pas aux besoins des utilisateurs métier, une nouvelle phase de conception de la conception PIM est requise et le processus redémarre.

- Enfin, les utilisateurs professionnels sélectionnent la mise en œuvre simulée qui répond le mieux à leurs besoins. Ensuite, les experts IoT utilisent le PSM et le code correspondants pour la configuration et le déploiement du hardware IoT.

C.7.2 Profil UML pour les données IoT - PIM

Le PIM permet de formaliser les besoins en données des utilisateurs métier en UML sans fournir de détails physiques ou d'implémentation. Le niveau d'abstraction élevé de STS4IoT (le PIM) décrit les exigences fonctionnelles des applications de données IoT. Plus précisément, le PIM définit les variables à échantillonner, le taux d'échantillonnage, les transformations requises, les fenêtres de transformation (temporelles), le taux de transmission et les données transmises. Ce niveau vise à fournir une représentation très lisible de l'application IoT pour les utilisateurs professionnels.

La structure de données PIM (Figure 5.2-A) permet de définir les variables requises à partir de l'application de données IoT. De cette manière, il constitue un modèle de données conceptuel pour les données IoT.

Cette représentation des données repose sur deux classes principales : <A-PIM-Source-Measure> et <A-PIM-Measure>. Ils représentent respectivement des données captées simples et composées. <A-PIM Source-Measure> contient des variables détectées et transformées simples. Les <A-PIM-Measure> présentent un opérateur de jointure qui résume la logique derrière la composition de ces variables simples. Dans notre étude de cas, les <A-PIM-Source-Measure> individuelles représentent chaque type de données détectées (par exemple, la pluie, la température et l'humidité du sol), tandis qu'une seule <A-PIM-Measure> représente la composition de toutes ces données détectées.

Néanmoins, la structure de données IoT définit uniquement les variables que l'application IoT doit fournir, et non comment elle doit les fournir. Par conséquent, les opérations PIM STS (Figure 5.2-B) permettent de représenter les transformations de données à l'intérieur de l'IoT.

Les applications IoT peuvent généralement définir trois grandes classes d'opérations : détection, transformation et envoi. Les opérations de transformation divergent en trois classifications principales : agrégation, filtrage et conversion. Les opérations de détection et d'envoi décrivent comment les objets IoT collectent et fournissent des données, respectivement. Ces opérations peuvent s'appuyer sur des politiques temporelles et numériques pour définir la fréquence et les fenêtres. Notre profil UML représente ces méthodes comme des opérations des classes précédemment décrites. En particulier, <A-PIM-Aggregate>, <A-PIM-Convert> et <A-PIM-Filter> sont utilisés pour la transformation ; <A-PIM-TupleDelivery> et <A-PIM TimeDelivery> pour l'envoi ; et <A-PIM-SensorGathering> pour le sens.

Dans notre étude de cas, les variables ne sont pas seulement détectées et envoyées. Ils nécessitent des transformations spécifiques telles que l'agrégation médiane (<A-PIM-Aggregate>) ou un changement d'unités (<A-PIM-Convert>).

C.7.3 Profil UML pour les données IoT - PSM

Le niveau d'abstraction faible de STS4IoT (le PSM) décrit les problèmes de mise en œuvre essentiels de l'application réelle, fournissant une représentation implémentable mais simplifiée de l'application de données IoT.

Plus précisément, le PSM exploite le catalogue DM pour représenter des détails supplémen-

taires à la conception de données PIM qui permet la mise en œuvre. Par exemple, les capteurs matériels, les interfaces et les communications et rôles des nœuds. Les détails physiques et de mise en œuvre d'une application IoT ne sont pas triviaux. En particulier, la définition des rôles des nœuds, la répartition des tâches et la topologie de déploiement peuvent varier considérablement d'une implémentation à l'autre.

Par exemple, le PIM de notre exemple en cours d'exécution peut avoir différentes implémentations possibles en fonction de diverses conditions et contraintes :

- Conditions de déploiement : un seul nœud peut surveiller un petit espace (par exemple, une pièce). Les grands espaces ouverts nécessiteront plusieurs nœuds de détection pour une surveillance efficace (par exemple, une ferme). En outre, les nœuds pourraient télécharger directement les données sur le cloud dans des endroits disposant d'un accès Internet. Sinon, ils auraient besoin d'un nœud récepteur ou d'une passerelle.

- Contraintes hardware : des limitations hardware spécifiques (par exemple, les capacités de traitement, l'alimentation en énergie, les interfaces de communication, la mémoire) affecteront la conception de l'implémentation. Par exemple, si un nœud final ne peut pas calculer les données, le niveau suivant (nœud récepteur ou serveur) doit effectuer les opérations. De plus, même si les nœuds sont proches d'une connexion Internet sans fil appropriée, ils ne se connecteront pas au réseau s'ils ne disposent pas de l'interface requise.

- Contraintes d'application : l'application spécifique peut également imposer certaines restrictions à la mise en œuvre. De plus, certaines applications nécessitent une grande précision et un contrôle des pannes, tandis que d'autres peuvent accepter une précision moindre. Par exemple, les applications critiques en temps réel telles que la détection d'incendie nécessitent une faible latence et un contrôle des pannes. Ainsi, plusieurs nœuds terminaux de bonne qualité doivent analyser les données in situ pour fournir des alertes fiables.

- Conditions de traitement : certaines transformations spécifiques (par exemple, l'agrégation spatiale) peuvent nécessiter une topologie et une répartition des tâches particulières. Par exemple, le calcul de la température moyenne d'un grand champ peut nécessiter plusieurs nœuds d'extrémité de détection de température envoyant des données brutes à un nœud récepteur dans une topologie en *Star*.

La structure de données PSM (Figure 5.7-A) conserve presque tous les stéréotypes originaux du SensorDataModel, car ils fournissent une approche initiale équitable de STS pour les nœuds uniques. Les nouveaux stéréotypes et valeurs étiquetées améliorent cette approche et modélisent les communications de données entre les objets IoT, ce qui permet de définir plusieurs nœuds coopératifs avec leurs rôles respectifs pour faire face aux capacités de communication de STS.

En particulier, le PSM cartographie les données PIM sur les différents nœuds utilisés. Par conséquent, le PSM doit représenter la manière dont chaque nœud IoT détecte, transforme et envoie des données à d'autres nœuds. En effet, comme décrit précédemment, un nœud final

ne peut détecter qu'une ou plusieurs variables et les envoyer à d'autres nœuds qui les transforment ou les fusionnent avec des données provenant de différents nœuds. Ces transformations et communications particulières entre les nœuds sont intentionnellement maintenues transparentes au niveau PIM, car les utilisateurs métier ne doivent pas gérer les détails de leur implémentation, traités uniquement par des experts IoT. Par conséquent, pour traiter les détails techniques de calcul et de communication sous-jacents aux modèles PIM, le PSM est structuré dans les classes principales suivantes : <A-PSM-GatheredMeasure>, <A-PSM-TransformedMeasure> et <A-PSM-DeliveredMeasure>.

Ils représentent respectivement la valeur détectée d'un nœud, les transformations appliquées par un nœud (principalement sur les données reçues) et la communication vers un autre nœud. Par exemple, dans le PSM de notre étude de cas, les valeurs d'humidité du sol détectées à 30 cm par l'EN sont représentées par trois classes <A-PSM-GatheredMeasure>. Un <A-PSM-TransformedMeasure> fait le calcul de la valeur médiane ; et un <A-PSM-DeliveredMeasure> modélise les données envoyées au SN.

En outre, le modèle PSM doit également définir exactement comment les nœuds IoT détectent, transforment et envoient des données, relatives au DM spécifique pour faciliter la mise en œuvre automatique du modèle.

Les opérations PIM STS définissent déjà comment les objets IoT acquièrent, calculent et délivrent leurs différentes variables. Néanmoins, plusieurs questions restent en suspens avant d'aborder l'implémentation : "Quelle sonde détecte les données ?", "L'appareil prend-il en charge les transformations définies ?", ou "Quelle interface enverra les données ?". Par conséquent, les opérations PSM STS (Figure 5.7-B) définissent différentes associations avec le DM pour répondre explicitement à ces questions. Par exemple, l'association <DefinedAs> de <A-PSM-Gather> et <A-PSM-Transformation> avec <D-PSM-Operation>, et la balise <Interface> dans <A-PSM-DeliverOnTuples> aident à répondre à ces des questions.

Les éléments du profil STS4IoT PSM permettent de modéliser des applications de données IoT implémentables telles que les WSN en tenant compte des caractéristiques STS. De plus, les experts IoT peuvent considérer et utiliser le profil PSM pour concevoir différentes options de mise en œuvre pour la même application.

C.7.4 Profil UML pour les données IoT - DM

Le DM est une représentation abstraite simplifiée des installations IoT qui facilite la définition des modèles tout en permettant la génération automatique de code implémentable. Plusieurs DM fournissent aux experts IoT un catalogue conceptuel pour trouver et utiliser facilement les appareils les mieux adaptés à leurs applications de données IoT.

Un DM peut représenter les variables de données qu'un nœud peut gérer en tant que <DeviceData>, les opérations sur les données qu'un nœud peut exécuter avec succès en tant que <DeviceOperations> et d'autres caractéristiques hardware pertinentes requises pour une génération de code appropriée en tant que <DeviceManagers>.

C.8 Validation des Profils

Nous avons validé nos profils de données de capteurs et de données IoT sous quatre aspects : la conformité aux normes UML via un test de l'outil CASE, la qualité des méta-modèles via une comparaison avec plusieurs méta-modèles de la littérature, la qualité et l'implémentation des modèles d'instance via l'implémentation et évaluation, et la généralisabilité des méta-modèles grâce à l'expérimentation avec différents hardware IoT. Le Chapitre 6 présente les résultats complets pour toutes ces validations.

C.8.1 Conformité UML

Cette validation utilise la suite de validation de MagicDraw® pour vérifier l'exactitude et l'exhaustivité des méta-modèles de données de capteurs et IoT dans la norme UML avec les tests par défaut fournis dans sa suite. Les deux profils étaient valides dans les neuf tests par défaut de la suite : *Diagram Merge, Numbering Validation, Orphaned Proxies, Parameters Synchronisation, Relationship Ownership, Shape Ownership, Spelling, UML Completeness Constraints, and UML Correctness*.

Cette validation vérifie également si les deux profils peuvent produire des modèles d'instance bien formés. Par conséquent, il valide les règles OCL des profils sur des ensembles de modèles PIM et PSM intentionnellement corrompus, et des modèles PIM et PSM appropriés. Par exemple, ces OCL vérifient si les opérations de détection et de livraison sont appropriées selon les stéréotypes de classe. MagicDraw® a identifié toutes les violations OCL des modèles corrompus et a renvoyé une validation réussie pour les modèles corrects.

C.8.2 Qualité du méta-modèle

Le fait de respecter les règles de base du langage de modélisation et de fournir des contraintes de formation ne garantit pas l'utilisabilité de nos profils UML. En effet, les méta-modèles peuvent devenir extrêmement complexes et devenir difficiles à comprendre, à maintenir et à utiliser ; ou être si simples qu'ils ne peuvent pas représenter la plupart des applications dans leur domaine. Par conséquent, nous avons utilisé un cadre de qualité pour évaluer la qualité d'un méta-modèle à partir de ses composants et de sa structure. Ce cadre considère cinq propriétés de qualité de chaque méta-modèle :

- La réutilisabilité : est la capacité d'un méta-modèle à contribuer avec ses construits à la définition de nouveaux méta-modèles. Il est positivement lié au nombre de méta-classes, de règles et de méta-attributs ; mais négatif lié aux associations et à l'héritage entre méta-classes.
- Compréhensibilité : c'est la capacité d'un méta-modèle à être facilement perçu et instancié par ses utilisateurs. Il est positivement associé au nombre de méta-attributs et de règles ; et négativement au nombre de méta-classes, leurs associations et leur héritage, et le nombre et la profondeur des arbres d'héritage.

- Fonctionnalité : c'est la capacité globale d'un méta-modèle à modéliser des modèles complets et diversifiés. Il est positivement lié au nombre d'associations et d'héritage entre les méta-classes, les règles, les méta-attributs et les méta-classes concrètes.

- Extensibilité : c'est le degré de facilité à ajouter de nouveaux éléments de modélisation au méta-modèle. Plus il y a de méta-classes dans un méta-modèle, plus son score d'extension est élevé. Plus il y a d'associations et d'héritages parmi ses méta-classes, plus il est difficile d'étendre.

- Bien structuré : détermine la qualité de l'architecture sous-jacente et des méta-classes qui la composent. Ainsi, cette propriété augmente avec le nombre de méta-classes bien définies (avec des règles, des attributs et des méta-attributs associés). De même, il décroît avec le nombre de hiérarchies d'héritage séparées et d'inter-associations entre méta-classes.

De plus, ce cadre a été utilisé pour évaluer plus de 2500 méta-modèles de plusieurs domaines. Ainsi, nous avons également comparé nos profils de données de capteurs et de données IoT à ces évaluations en utilisant le rang centile pour estimer leur qualité attendue dans ces propriétés.

Le profil de données de capteur a obtenu des résultats exceptionnels en intelligibilité et en bonne structuration ; la réutilisabilité était également bien au-dessus de la médiane. Cependant, la fonctionnalité et l'extensibilité étaient faibles. Ainsi, la maintenance et la mise à jour des profils peuvent nécessiter plus d'efforts, en particulier pour le PIM.

Le profil de données IoT a amélioré presque toutes les propriétés du PIM et du DM, ce qui signifie qu'il est plus facile à utiliser, à comprendre et à entretenir. Néanmoins, le PSM a eu une moins bonne performance. Pourtant, sa compréhensibilité et sa bien structuré figuraient parmi les 3% meilleurs méta-modèles de la littérature.

C.8.3 Qualité des modèles d'instance

Cette validation teste la faisabilité et les performances de nos approches MDA pour les applications de données de capteurs et de données IoT dans des scénarios typiques basés sur l'étude de cas.

En particulier, ces expériences vérifient les métriques de spécification, d'utilisation et de mise en œuvre des modèles d'instance. La spécification fait référence à une bonne structure et définition du modèle. Elle est donc liée à la Compréhensibilité et au Bien Structuré du méta-modèle sous-jacent. En outre, l'utilisation fait référence à une bonne représentation du système étudié du point de vue de l'utilisateur. En ce sens, il peut être lié à la Fonctionnalité du méta-modèle. Enfin, la mise en œuvre est la quantité d'efforts pour mettre en œuvre et réutiliser le modèle. Il a une légère association avec les propriétés Bien Structuré et Fonctionnalité du méta-modèle, et la capacité de l'outil d'implémentation automatique.

Expérience de données de capteur

Le scénario de l'expérimentation est le suivant : Un utilisateur métier du domaine agricole demande en langage naturel à un expert capteurs de collecter des données spécifiques sur une parcelle particulière. Pour permettre de vérifier la maintenabilité, l'utilisateur métier divise ses requêtes en deux : i) "Donnez-moi la pression moyenne, et la température maximale et minimale de la parcelle X toutes les cinq minutes". ii) "Maintenant, ajoutez une lecture légère à partir du même point toutes les cinq minutes". L'expert en capteurs utilise notre proposition pour concevoir et développer l'application de données de capteur, en validant chaque étape avec l'utilisateur professionnel.

De plus, pour comparer avec une approche manuelle, le même utilisateur métier a fait les mêmes requêtes à un autre expert capteurs préalablement formé à l'utilisation de la plateforme UEB (mais pas dans notre profil), qui a adressé directement la solution en code. Pour le développement non automatique, nous avons mesuré le nombre d'itérations pour chaque requête, le temps de livraison de chaque itération, l'acceptation de l'application reçue et la réutilisation du code.

Les résultats de cette expérience prouvent que les modèles conçus avec notre profil UML peuvent être facilement compris par les utilisateurs métier ayant des connaissances UML. En outre, ils montrent également que le niveau d'abstraction élevé (PIM) est plus simple que le niveau d'abstraction faible (PSM), mais ils sont tout aussi lisibles. De plus, l'expert en capteurs a dû modifier moins de 1% du code pour les deux requêtes, indiquant une bonne mise en œuvre. Enfin, la maintenabilité était bonne puisque le processus de mise à jour prenait environ un tiers du temps que le processus d'origine avec une grande acceptabilité de la part de l'utilisateur.

Expérience de données IoT

Cette section rapporte les observations issues de l'utilisation de l'approche STS4IoT MDA sur l'étude de cas réel décrite précédemment. Pour cette expérimentation, les utilisateurs sont un décideur expert en systèmes d'irrigation et un expert IoT. Ces utilisateurs ont conçu et mis en œuvre une application de données IoT selon la méthode IRRINOV.

Dans un premier temps, le décideur explique clairement les exigences de la méthode IRRINOV à l'expert IoT sous forme de demande. Deuxièmement, l'expert IoT construit le modèle PIM pour ces exigences et le présente au décideur pour évaluation. Troisièmement, l'expert IoT définit le PSM pour une option de mise en œuvre et le présente au décideur, qui l'évalue et compte les itérations jusqu'à l'acceptation. Quatrièmement, l'expert IoT utilise l'outil de modélisation à code STS4IoT pour générer le code d'application. Dans cette étape, l'expert IoT évalue l'exhaustivité du code et le simule, en présentant les résultats des tests initiaux au décideur. Si le décideur juge les résultats appropriés, nous considérons que l'application a été générée avec succès.

Les résultats de cette expérience prouvent que la Lisibilité de tous les modèles est parfaite, et leur Simplicité est bonne. Même si la simplicité tombe à 0,52 dans les deux PSM, des valeurs supérieures à 0,5 indiquent une présence prédominante de classes dans le modèle, ce qui réduit

sa complexité. En effet, compte tenu de la grande complexité de la méthode IRRINOV utilisée pour notre étude de cas, ce résultat est très positif.

C.8.4 Généralisabilité des méta-modèles

Les expériences précédentes démontrent la faisabilité des profils de données de capteur et IoT pour des applications réelles. Ces expériences se concentrent sur les dispositifs UEB pour la modélisation et la mise en œuvre. Par conséquent, une question peut se poser concernant la généralisabilité de l'ensemble de l'approche MDA, c'est-à-dire la capacité à modéliser et à mettre en œuvre des applications avec différents appareils.

Cette section valide la prise en charge de divers appareils IoT dans le profil de données IoT pour répondre à une telle question. En particulier, il teste si l'approche STS4IoT MDA peut modéliser et implémenter des applications de données IoT à l'aide d'un système d'exploitation embarqué différent : RIOT.

RIOT est un système d'exploitation open source largement utilisé dans les applications industrielles et académiques. Il prend en charge plusieurs appareils IoT de différentes gammes basés sur des microcontrôleurs 8 bits, 16 bits et 32 bits. Ce système d'exploitation permet un traitement en temps réel et multithread, et prend en charge plusieurs protocoles de réseau, notamment IPV6, Sigfox et LoRaWAN.

Les méta-modèles PIM, PSM et DM définissent les structures et les notations pour la modélisation des applications IoT. Par conséquent, ces méta-modèles doivent prendre en charge la représentation de toute implémentation (et appareil) IoT. Ces profils suivent strictement les définitions présentées au Chapitre 2. Ces définitions contraignent les applications de données IoT à un WSN homogène et définissent les appareils IoT comme des capteurs avancés qui peuvent recevoir, joindre et traiter des données au-delà de l'agrégation. Par conséquent, le profil de données IoT est suffisamment générique pour prendre en charge diverses implémentations et appareils, et il suit les directives MDA. De la même manière, les modèles d'instance doivent conserver les bonnes propriétés de spécification et d'utilisation indépendamment de la plate-forme d'implémentation.

De plus, STS4IoT doit également prendre en charge la mise en œuvre dans les appareils RIOT. Par conséquent, nous avons défini une expérience utilisant trois types de nœuds RIOT dans un ensemble de tests pour vérifier la mise en œuvre des modèles dans du hardware hétérogène et la conformité du code généré avec les modèles. Notre expérience définit un réseau en *Star* avec trois nœuds terminaux simples (EN) qui détectent et envoient des données et un nœud récepteur complexe (SN) qui rejoint et transforme chaque variable et rejoint et envoie les variables.

Les résultats de ce test vérifient que l'approche IoT-Data MDA maintient la mise en œuvre de ses modèles indépendamment de la plate-forme d'implémentation. Ils montrent également que la mise en œuvre d'un code généré automatiquement peut fournir les données demandées dans le PIM. Néanmoins, cela n'a pas toujours été le cas ; divers tests ont présenté des différences entre les données des nœuds puits et les données calculées en externe. Bien que la plupart des données soient cohérentes, des problèmes liés au processus de livraison sans fil et au respect des délais

ont causé les erreurs. Par conséquent, les implémentations IoT doivent inclure une meilleure synchronisation et des stratégies de tolérance aux pannes pour fournir les données demandées.