Research Article

Modeling Internet of Things Software for Public Transportation

Sadik Arslan[1,2,*], Geylani Kardas[1]
[1] International Computer Institute, Ege University, Izmir, Türkiye
[2] Automotive Software Engineering, Vestel Elektronik A.Ş. Izmir, Türkiye
*Correspondence: sadik.arslan@vestel.com.tr
DOI: 10.51513/jitsa.1328020

Abstract: The Internet of Things (IoT) is a promising domain and one of the leading technologies used in public transportation in recent years. However, in addition to the heterogeneity and high complexity problems that are usually observed in the development of IoT systems, the specific needs of public transportation domain make the construction of such systems even harder for the public transportation. This paper proposes the use of a domain-specific modeling language (DSML), called DSML4PT, to facilitate the design and implementation of IoT-based public transportation systems. A metamodel is introduced that enables modeling IoT-based applications according to the different viewpoints and leads to the model-driven engineering of such applications for different IoT-based public transportation platforms. Furthermore, originated from this metamodel, design and implementation of the DSML4PT language with including its syntax and semantics definitions are all discussed in this paper. Use of this DSML supports both the design of the IoT-based public transportation software graphically and the automatic generation of the code required for the implementation. Based on the conducted case study, it has been observed that approximately 80% of a public transportation application can be generated only with using DSML4PT.

Keywords: Internet of Things, public transportation, smart cities, model-driven engineering, domain-specific modeling language

Toplu Taşıma İçin Nesnelerin İnterneti Yazılımlarının Modellenmesi

Özet: Nesnelerin İnterneti (IoT) pek çok alanda olduğu gibi son yıllarda toplu taşımada da sıklıkla kullanılan teknolojilerin başında gelmektedir. Ancak, IoT sistemlerinin geliştirilmesinde genellikle gözlemlenen heterojenlik ve yüksek karmaşıklık sorunlarına ek olarak, toplu taşıma alanının kendine özgü ihtiyaçları, toplu taşıma için IoT tabanlı sistemlerin inşasını daha da zorlaştırmaktadır. Bu makalede, IoT-tabanlı toplu taşıma sistemlerinin tasarımını ve uygulanmasını kolaylaştırmak için DSML4PT adlı alana özgü bir modelleme dilinin (DSML) kullanılması önerilmektedir. IoT-tabanlı uygulamaların farklı bakış açılarına göre modellenmesini sağlayan ve bu tür uygulamaların farklı IoT-tabanlı toplu taşıma platformları için model güdümlü mühendisliğine imkan veren bir üstmodel sunulmuştur. Ayrıca, bu üstmodele dayanan DSML4PT dilinin sözdizimi ve anlambilim tanımları da dahil olmak üzere tasarımı ve uygulanması bu makalede tartışılmaktadır. Bu DSML'in kullanımı, hem IoT tabanlı toplu taşıma yazılımının görsel olarak tasarlanmasını hem de uygulama için gerekli kodun otomatik olarak oluşturulmasını destekler. Yürütülen vaka çalışmasına dayanarak, bir toplu taşıma uygulamasının yaklaşık %80'inin yalnızca DSML4PT kullanılarak üretilebildiği gözlemlenmiştir.

Anahtar Kelimeler: Nesnelerin İnterneti, toplu taşıma, akıllı şehirler, model-güdümlü mühendislik, alana-özgü modelleme dili

ORCID: 0000-0003-4057-2030; 0000-0001-6975-305X (in hierarchical order)
Received 16 July 2023; accepted 6 October 2023
Peer review under responsibility of Bandirma Onyedi Eylul University.

1. Introduction

Today, systems such as fare collection, pollution measurement, passenger counting, vehicle system management, passenger information, advertisement management, vehicle tracking are used frequently in public transportation. These systems can work together or be standalone. Public transportation systems can be used in various environments such as buses, subways, trains and trams. In these environments, systems are exposed to harsh conditions due to high temperature differences, constant vibration, rain, sun and mobility (Evin et al., 2020).

As being one of the most promising technologies today, the Internet of Things (IoT) is used in many domains such as smart cities, public transportation, field monitoring, military, agriculture and healthcare. IoT is a structure where things are connected to each other and to the rest of the Internet. "Things" are any units that can connect to the Internet and exchange information. In IoT, objects are interconnected and can also exchange data (Rayes and Salam, 2019).

Heterogeneity and complexity of the hardware and software being used cause very important problems in the development of IoT-based public transportation systems. There is a wide variety of hardware and software in public transportation vehicles and in the cloud/server environment. Each unit in the system has its own development environment and settings. Especially the memory and processor power in the end devices are very limited (Aydin et al., 2019). Besides, hardware and software are highly interdependent. Although it is quite common to develop platform-dependent software in these systems, deployment and execution of the software directly on a different platform is challenging, even if the same programming language or software development tools are used. Moreover, the differences of various IoT systems, mobility issues, working in harsh physical conditions and the use of domain-specific "System-on-Chip" (SoC), hardware and their varying standards and protocols also make the implementation of IoT-based public transportation applications difficult (Arslan and Kardaş, 2021).

As successfully applied in many other domains (e.g. (Brambilla et al., 2017; Lelandais et al., 2019; Mohamed et al., 2021), model-driven engineering (MDE), mostly supported with the use of domain-specific languages (DSLs) (Kosar et al., 2019; Wasowski and Berger, 2023) and domain-specific modeling languages (DSMLs) (Kardas et al., 2023), may also help overcoming the abovementioned difficulties of IoT-based public transportation applications and facilitate the system development by leveraging the abstraction level during design and generating many components automatically for the complete implementation, i.e. software models of the public transformation systems can be automatically manipulated and transformed to the executable and deployable artifacts. Hence, in this paper, we investigate modeling IoT-based public transportation systems and introduce a DSML, called DSML4PT to support MDE of such system.

DSML4PT language's syntax is based on a metamodel enabling modeling IoT-based applications according to the different viewpoints and leads to the MDE of such applications for different IoT-based public transportation platforms. Furthermore, originated from this metamodel, design and implementation of the DSML4PT language, including its syntax and semantics definitions, are discussed in this paper. Use of this DSML supports both the design of the IoT-based public transportation software graphically and the automatic generation of the code required for the implementation. Evaluation of the usability of DSML4PT through a case study is also discussed in this paper.

The remainder of the paper is organized as follows: Section 2 gives the related work. Section 3 discusses a metamodel for modeling the public transportation software. Section 4 gives the syntax definition of DSML4PT based on the proposed metamodel. Transformational semantics of the language is given in Section 5. Section 6 discusses the use and evaluation of DSML4PT and Section 7 concludes the paper.

2. Related Work

In this section, firstly, studies proposing MDE for the development of IoT systems are reviewed. In the second part, MDE studies specific for IoT-based public transportation systems are discussed.

2.1. MDE Studies for IoT Applications

Many researchers find MDE promising and propose modeling approaches, languages and MDE tools for overcoming IoT development challenges (Arslan et al., 2023). For instance, Harrand et al. (2016)

focused on the fundamental IoT problems such as deployment and heterogeneity and they proposed the use of the "Internet of Things Modeling Language" (ThingML). ThingML supports visual modeling structures such as state and component diagrams, and modeling IoT applications covering different perspectives through a platform-dependent modeling language and from the architectural level to the behaviour level of individual devices. Platform dependent code generation such as Arduino, Raspberry Pi, Intel Edison and popular programming languages (C/C++, Java, Javascript) are supported in ThingML. Recent extensions of ThingML, like AIoTM (Hu et al., 2023), exists for the construction of the Artificial Intelligence-based IoT components across different modeling levels for the purposes of intelligent sensing and control. Similar to ThingML, the Vorto project (The Vorto project, 2018) led the creation of the Vorto DSL which is used to specify manufacturer-independent abstraction layers that describe the functions and features of tools at different levels of detail.

An MDE development framework, named UML4IoT was developed for industrial IoT production systems (Thramboulidis and Christoulakis 2016). Based on the well-known Unified Modeling Language (UML), IoT components in a production system were modeled in this study. Similarly, MDE applications for different IoT devices is being carried out in the industry. For example, a multi-view modeling approach was used in (Muthukumar et al., 2019) to perform design and verification of the Industrial Internet of Things (IIoT) enabled control in process industries. Use of the proposed MDE and IIoT architecture was exemplified for the quadruple tank process, a benchmark problem in control. Ahmed et al. (2019) introduced an interoperability architecture for data exchange in the smart gas distributed networks based on the MDE projection and transformation concepts. Metamodels and models for different abstraction layers of data exchange via smart hubs were described and their use in a smart gas distribution grid was discussed.

COMFIT environment (Faria et al., 2017) presented an application development paradigm based on the MDE infrastructure, where an application management and execution module use the cloud interface to develop automated IoT applications. The methodology in (Costa et al., 2020), which combined MDE and service-oriented architecture (SOA), provided different levels of abstraction in building software components of IoT systems. Thus, in IoT, network resources are dynamically allocated using application programming interfaces (APIs). An integrated MDE methodology to design and deploy a network of things was presented in (Berrouyne et al., 2022). The use of this methodology may provide the abstraction of the heterogeneous things and the control of these things as well as a mechanism to define constraints on the network. With this methodology, it is also possible to process the IoT models and generate the network artifacts by using a code generator, based on model transformation.

MDE has also been adopted for the Wireless Sensor Networks (WSN) especially considering the system implementation on the operating systems such as TinyOS and Contiki. Studies like (Marah et al., 2021; Vorapojpisut, 2018) introduced MDE techniques for graphical modeling of hardware modules, sensors, components and WSN configurations that also enables code generation for WSN programming languages such as nesC.

To cope with the challenges of IoT prototyping which are originated from cross-domain interoperability, reusability and customization cost and expertise requirement, Xiao et al. (2019) proposed a model-driven service composition architecture based on the finite-state machine descriptions. Model-based simulation of realistic smart home scenarios was discussed in (Kölsch, 2020) where IoT devices were used in the implementation of the related simulation model. The model was separated into three levels covering the simulation of core components of the house, events in the house and calculation of the power consumption of the house. A model-based design approach was applied in (Kotronis et al., 2018) for healthcare to create the structure and the interconnection of an IoT-based system-of-systems for the remote monitoring of elderly subjects. With MDE, it is also possible to formalize the mathematical relationships and validation expressions among the components and operational requirements of the system.

Although the above-mentioned studies provide various noteworthy approaches for modeling IoT systems, they do not address the problems of modeling IoT-based public transportation systems and only provide MDE methods and/or tools for the domains other than the public transportation.

2.2. MDE Studies for IoT-based Intelligent Transportation and Public Transportation

In the domain of intelligent transportation, which may have co-working spaces with public transportation, only a very limited number of studies exist for proposing modeling or discussing the case studies of applying MDE for IoT-based intelligent and or public transportation systems.

IADev framework (Rafique, 2020) enabled SOA-based IoT development according to model-driven development principles. The conceptual models of IADev lead the development process, i.e. modeling in IADev allows for IoT service orchestration by also providing transformation methods and automated platform-specific code development. The case study of this general-purpose framework was made over an intelligent transportation system. More emphasis was placed on traffic management and road safety. Similarly, a DSL, called SimulateIoT-Mobile, an extension of SimulateIoT that includes support for simulating IoT systems with mobile nodes was introduced in (Barriga et al., 2023). Although the main domain of the proposed DSL is mobile devices rather than the public transportation overall, an IoT simulation of personal mobility devices (PMD) based on public bicycles was also considered as one of the examples in this study.

Hause et al. (2018) stated that IoT is very important in public transportation and IoT system development can be facilitated with MDE approaches. Hence, they introduced an MDE approach for the development of IoT systems in smart cities. How to model both the whole system for components such as traffic lights, sensors, and the management of complexity and synchronization between the systems was discussed.

Iovino et al. (2019) provided a case study of an integrated card-based access control system authorizing people based on Near Field Communication (NFC) tags. An MDE approach was proposed that supports semi-automatic code generation in an IoT infrastructure. A solution was given for the pass authorization field with NFC. The NFC system used here is only considered as the system to be used for access at the doors. In fact, NFC systems can also be used for fare collection, which is one of the public transportation systems. However, that was not considered in (Iovino et al., 2019). With the use of NFC in public transportation, fare collection is provided together with special hardware structures, called SAM modules. Thus, the fare collection process becomes secure. In addition, NFC payment is only a small part of the entire fare collection system.

The ENACT DevOps (EDO) (ENACT Project, 2018) framework aimed to ensure the continuity of DevOps service quality and the best implementation of the development lifecycle of IoT systems, and hence it defined the necessary components in this direction. In addition, the difficulties experienced when using the ENACT tool in a testbed environment were discussed (ENACT Project, 2018). These difficulties mainly arose from the heterogeneity inside IoT platforms. Offering a DevOps approach, ENACT also examined similar IoT problems not only in terms of development but also in field operations. The test bed in this study can be used in intelligent transportation systems, especially for train control.

Vitruvius (Fernandez et al., 2014) is a platform where users without programming knowledge can design and quickly implement rich Web applications based on real-time data consumption from interconnected vehicles and sensors. A system that provides an MDE development environment for this platform, which can be installed on vehicles, was realized. The general aim is to design a system with MDE, which will collect and monitor sensor information such as CO2 amount, speed, location information, regardless of what type of vehicle (e.g., automobile, truck).

Mazzini et al. (2015) presented an MDE methodology to provide effective and efficient protection mechanisms that include real-time, non-functional features such as security and performance to be used in IoT system components. Although, it was mentioned that this new methodology can also be applied to the issues of verifying data distribution in Intelligent Transport Systems, there is no discussion of how the modeling capabilities exactly supported eliminating these issues and uncertainties.

Our work contributes to these efforts by introducing the first full-fledged modeling language for the MDE of IoT-based public transportation systems according to the different modeling viewpoints supporting high mobility, standards and protocols specific for public transportation, various SoC and hardware components being used as well as different underlying operating systems. Both all-embracing

metamodel of the entities of the IoT-based public transportation systems and automatic transformation of the model instances of this metamodel brought by the use of this new language lead to the design and implementation of wide range applications that will be deployed on different IoT-based public transportation platforms in contrast to the above mentioned studies which mostly consider specific case studies of public transportation, limit the implementation platforms and rarely support the automatic generation of the executables for the modeled systems.

3. A Metamodel for Public Transportation Software

The abstract syntax is used in DSMLs to describe concepts and their relationships in a domain. Additionally, abstract syntax is used to describe how the vocabulary of concepts provided by the language can be combined to create models or programs (Wasowski and Berger, 2023). From the MDE perspective, an abstract syntax is usually provided by a metamodel definition and metamodels define what models should look like (Kühne, 2022). Hence, the abstract syntax of our DSML (DSML4PT) for IoT-based public transportation applications is also built with a metamodel. This section introduces this metamodel.

The metamodel, which constitutes the abstract syntax of DSML4PT, was developed in this study by considering the connections and main elements of the public transportation systems including IoT technologies. Related metamodel entities and their relations are derived based on examining various public transportation specifications (such as (ITxPT, 2017; ITS Standardization, 2021; ISO 24014-1:2015, 2015)) and requirements of developing IoT software for public transportation systems discussed in (Arslan and Kardaş, 2021). The metamodel is divided into three different viewpoints. These are named IoT Core, Public Transport, and Service/Cloud. These viewpoints are based on the relationships of the key units in public transportation IoT systems. The metamodel is encoded with Eclipse Ecore (The Eclipse Foundation, 2015) and hence it is possible to integrate the metamodel with various MDEtools based on Eclipse Modeling Framework (EMF).

Grouping the units of the mass transit IoT domain according to different viewpoints in our metamodel does not only facilitate the development of the DSML4PT's syntax, but also enables this syntax to be more easily interpreted and used more effectively by public transportation IoT developers. Updating one of the related viewpoints is much easier than using a single viewpoint and working on a large supermodel. In any DSML4PT viewpoint, the complexity of removing and replacing elements or relations is reduced. However, with the multiple viewpoint structure, it is also possible for developers to extend the existing syntax in case of some special needs in the future.

The viewpoints developed for DSML4PT are discussed in the following subsections with their Ecore diagram representations. In each diagram of the viewpoints, metamodel entities (elements, units) are indicated by yellow rectangles. In addition, elements having the characters “<<” and “>>” prior to their names show relationships with other viewpoints. These elements are common elements between viewpoints, i.e. they exist separately in different viewpoints. For example, in Figure 1 where the IoT Core viewpoint is found, the Architecture entity, which is the common element of all viewpoints, is used with <<Public Transport Viewpoint>> and <<Service/Cloud Viewpoint>>. Elements and associations covered in each viewpoint are indicated in the text with italics.

3.1. IoT Core Viewpoint

The IoT Core viewpoint includes elements that define a generic IoT device, regardless of the characteristics of public transportation devices. These devices are generally IoT devices with microcontrollers such as STM32 and ArduinoUno that do not contain an operating system. They do not have units such as window systems or protocols that are specific to the operating system. When the studies in the literature are examined, there are examples of metamodels that focus on general use platforms such as Raspberry Pi and Arduino (Iovino et al., 2019; Alulema et al., 2017; Asici et al., 2019). However, platforms that are thought to be similar are actually can be quite different. Raspberry Pi devices are actually devices that contain an operating system. When it is possible to work in detail, it is seen that these development platforms actually have features such as window system, file and network manipulation, which include operating system features. Therefore, in the IoT Core viewpoint, there is no support for devices with microprocessors. In order to provide more comprehensive support, the support of microprocessor devices has been taken to the Public Transport viewpoint. There is no

example of this structure as developed in our study. Microprocessor and microcontroller structures are separated for the first time at the metamodel level.

There are also examples of IoT metamodels in the WSN domain (Marah et al., 2021; Asici et al., 2019). However, in these examples, studies can only be carried out on special hardware platforms where it is possible to use operating systems such as TinyOS or Contiki. The metamodels in these studies are different from the models that can be used in the studies in the general IoT domain, and their common usage is very difficult. In (Dautov and Song, 2019), the fleet management supermodel of the system consisting of IoT devices is presented. There is no compatibility with the target of our study, as the focus is not on how the IoT system works, but on how the fleet is managed. With the use of the metamodel found in (Hassine et al., 2017), the system is mostly created with elements such as the IoT device itself and the human user. However, with such an approach, the IoT system can be modeled as a general view, and human and device interaction can be shown. This approach is incomplete, and our metamodel focuses on the interactions of the working units of the system, where the software code can be generated, completely different from this structure.

Figure 1 shows the IoT Core viewpoint of our metamodel. The *Architecture* element of the metamodel is the basic node from which the whole system is derived. From this node, other viewpoints of the system can also be generated. The *Architecture* element is in a has-a relationship with other elements derived from it. The *Controller* element determines the platform type of our microcontroller system. The type of microcontroller is selected with the *TypeController* variable. The communication interface type and parameters of the system are set with the *Communication* element derived from the Controller element. The *TypeCommunication* variable provides a selection of interfaces such as Wifi and Bluetooth. The programming language selection is provided by the *Language* item. The language in which the programming will be made is selected with the *TypeLanguage* variable. There are *Input* or *Output* elements derived from the *Port* abstract element to ensure the use of the sensors and actuators to be used. *Sensor* elements are derived from *Input* elements, and *Actuator* elements are derived from *Output* elements. Analog or digital type ports can be selected from the *TypePort* variable. Sensor types such as temperature and humidity are selected from the *TypeSensor* variable. *TypeActuator* variable includes actuators such as buzzers and speakers.

3.2. Public Transport Viewpoint

Public Transport viewpoint includes elements that describe the unique features of a public transportation system. In this viewpoint, the difference between public transportation devices from general IoT devices is defined at the metamodel level. It contains elements and relationships that enable the modeling of the domain. No specific metamodel for public transportation has been encountered in any source in the literature. Starting from the processor level, this metamodel has great differences such as supporting special hardware, standards and protocols created for general use metamodels. Microprocessor families working with the operating system are reflected in the metamodel in great detail from this point of view. Figure 2 shows the entities and their relationships in our metamodel's Public Transport viewpoint.

Architecture element common with other perspectives is the starting point of the Public Transport viewpoint. The *SoC* element provides the definition of the SoC platform to be used in the public transportation IoT system. Platforms such as iMX, Sitara, Raspberry Pi can be selected with the *TypeSoC* variable. With the *OperatingSystem* element derived from the *SoC* element, the operating system to be used in the public transportation IoT device is added. With the *TypeOperatingSystem* variable, operating systems such as Linux and derivative operating systems as well as Andorid Things can be selected. The window system to be used in IoT devices is made with the *TypeWindowSystem* variable in the *WindowSystem* element. In this variable, there are window systems such as X and Wayland. The language of the program to be used is set with the *Language* element derived from the *WindowSystem* element. The *TypeLanguage* variable enables the programming language to be selected among languages such as C, C++, Java, Python. *SpecialHardware* is the item that enables the support of public transportation specific hardware. With the *TypeSpecialHardware* variable, hardware such as SAM, SIM, which is thought to be used, is selected. Setting of standards and protocols specific to public transportation is done with *Standards* and *Protocols* items. With *TypeStandards* and *TypeProtocols* variables, it is possible to select public transportation specific standards such as ITxPT and protocols

such as CCTalk. With the *Communication* element derived from the *SoC* element, the communication interface type and parameters of the system are determined. *TypeCommunication* variable enables the use of interfaces such as Wifi and Bluetooth. Similar to the IoT Core viewpoint, *Input* or *Output* elements derived from the *Port* abstract element are used to enable the use of sensors and actuators. Sensor elements are derived from *Input* elements, and *Actuator* elements are derived from *Output* elements. Analog or digital type ports are selected from the *TypePort* variable.

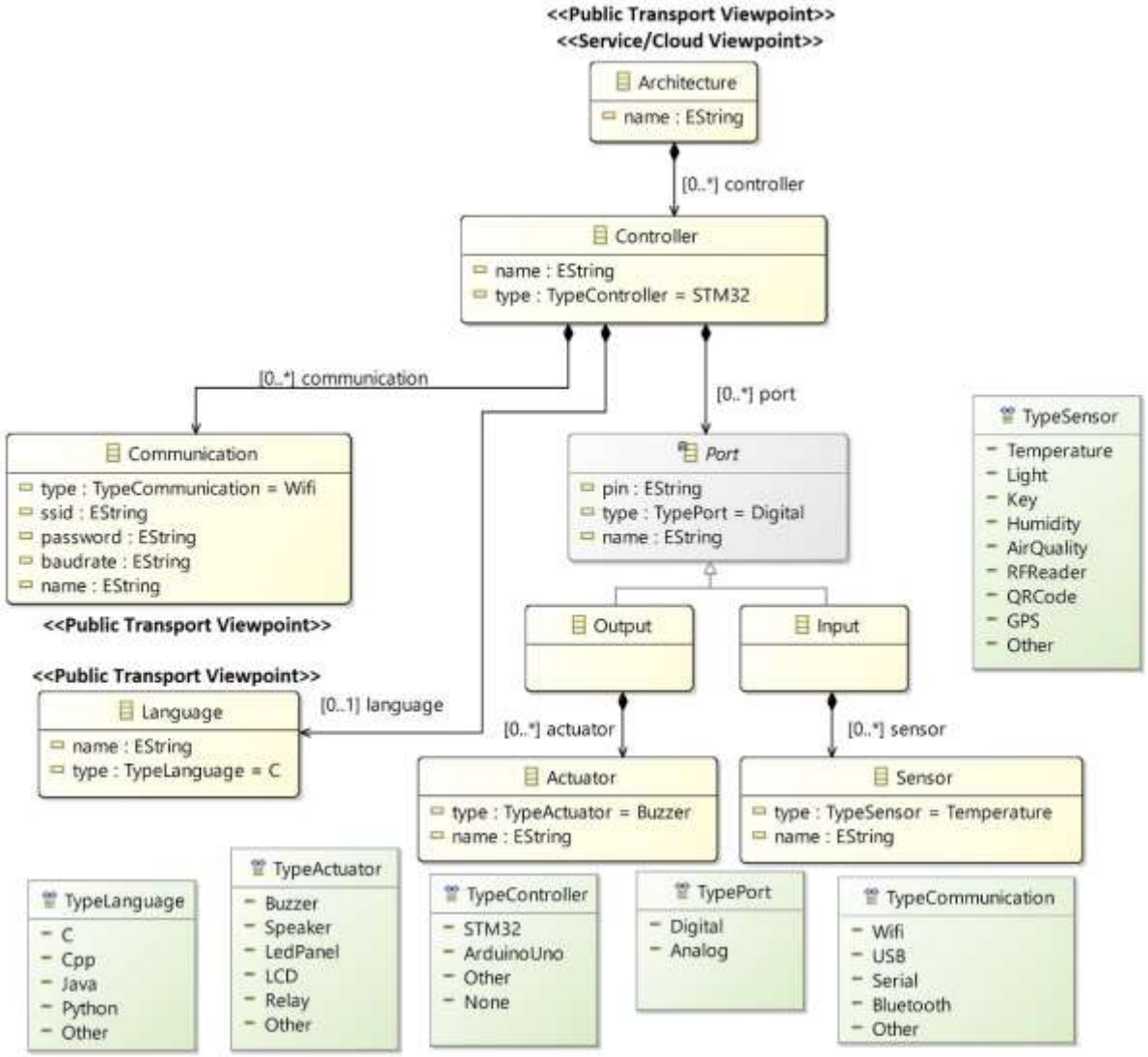

Figure 1. IoT Core viewpoint.

3.3. Service Cloud Viewpoint

In this viewpoint, elements and relationships are defined for service and cloud systems in public transportation IoT systems. The metamodels in general IoT systems do not contain structures for service and cloud mass transit. They are specialized for using the services of simple microcontrollers. In this perspective, there are smart phone and computer application elements that provide the monitoring of public transportation IoT devices. When the metamodels in the literature are examined, there are no studies that customize the service and cloud as in this study. In general, the service and cloud elements are included in the metamodels as a whole element (Sosa-Reyna et al., 2018; Alulema et al., 2017) or in SOA studies, the metamodels only represent service and cloud architectures (Costa et al., 2020; Cai et al., 2018; Betancourt et al., 2020). Hence, our metamodel contributes to these ongoing service and cloud metamodeling efforts by considering the service and cloud space from the perspectives of IoT systems separately.

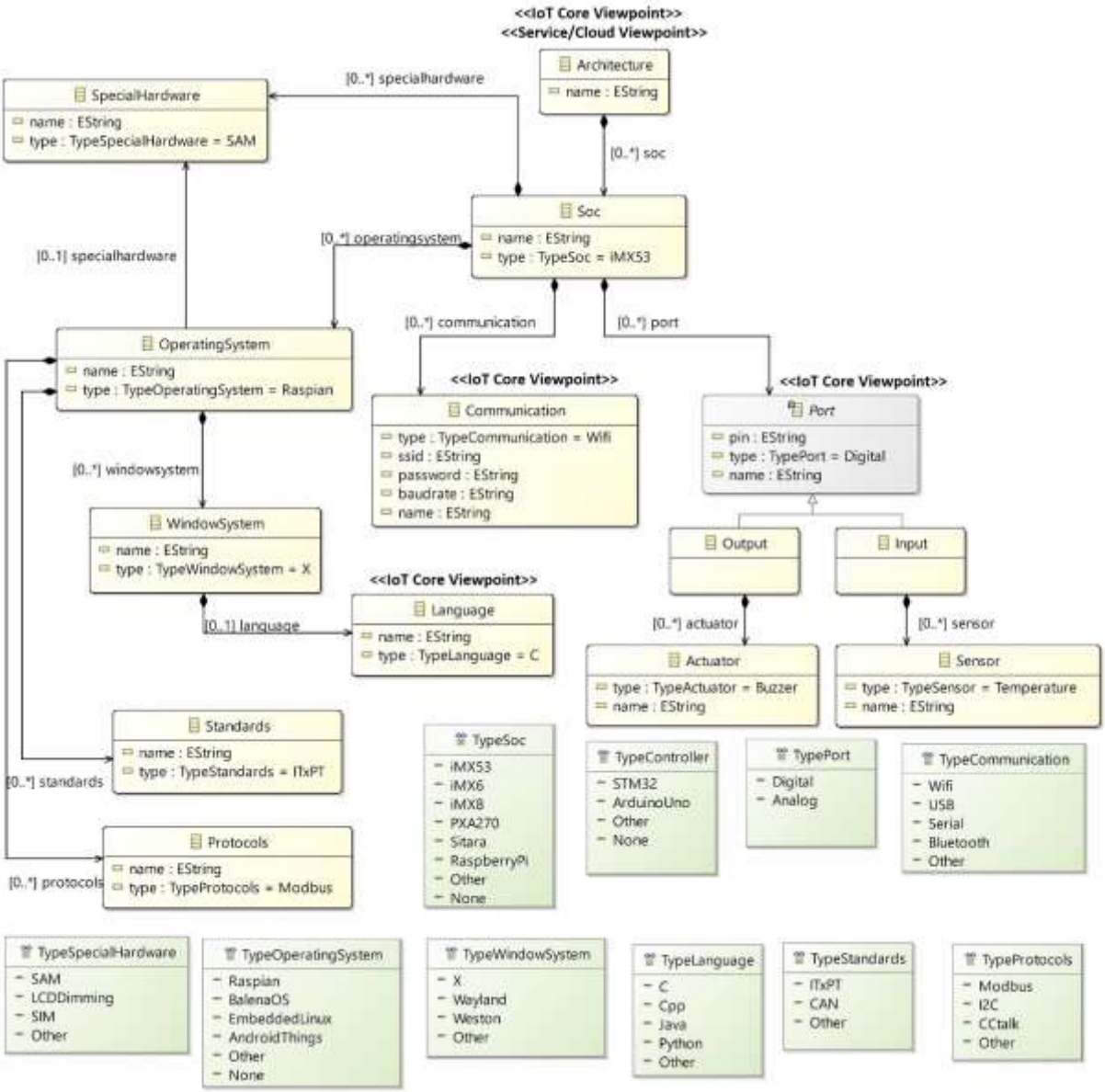

Figure 2. Public Transport viewpoint.

As with other viewpoints, the Service/Cloud view starts with the *Architecture* element (see Figure 3). Service and cloud configurations are made with the *ServiceApp* and *Cloud* elements derived from the *Service* abstract element. With the *TypeService* variable, the service type can be set in types such as REST and SOAP. With the *TypeCloud* variable, cloud type can be selected such as Azure or Firebase. The item for which the smartphone application is determined is the *SmartPhone* item. With the *TypeSmartPhone* variable, the type of the application is preferred as Android or iOS. The monitoring application in the computer program is set using the *MonitorApp* item. The language of the computer application is determined by the *TypeMonitorApp* variable.

4. Concrete Syntax for Public Transportation Modeling

Concrete syntax provides a mapping between concepts in abstract syntax and their representation on instance models. When the most basic function is considered, concrete syntax can be defined as a set of signs that facilitate the presentation and construction of the language. This section describes the graphical concrete syntax of DSML4PT that maps abstract syntax elements to their graphical representations.

In this study, the Sirius tool (The Sirius Project, 2023) based on Eclipse EMF and GMF technologies was preferred in the derivation of the concrete syntax. Sirius is an Eclipse project that allows to easily

create a graphical modeling environment using Eclipse modeling technologies. The Sirius tool was developed by Obeo and Thales to create a generic framework for MDE that can be easily adapted to specific needs. Complex architectures can be developed more easily in certain areas. A modeling environment built with Sirius consists of Eclipse editors that allow users to create, edit, and view EMF models. Editors are defined by a model that describes the entire structure of the modeling environment, its behavior, and all printing and navigation tools. This description of a Sirius modeling environment is dynamically interpreted within the Eclipse IDE. To support the specific requirement for customization, this tool can be extended in many ways.

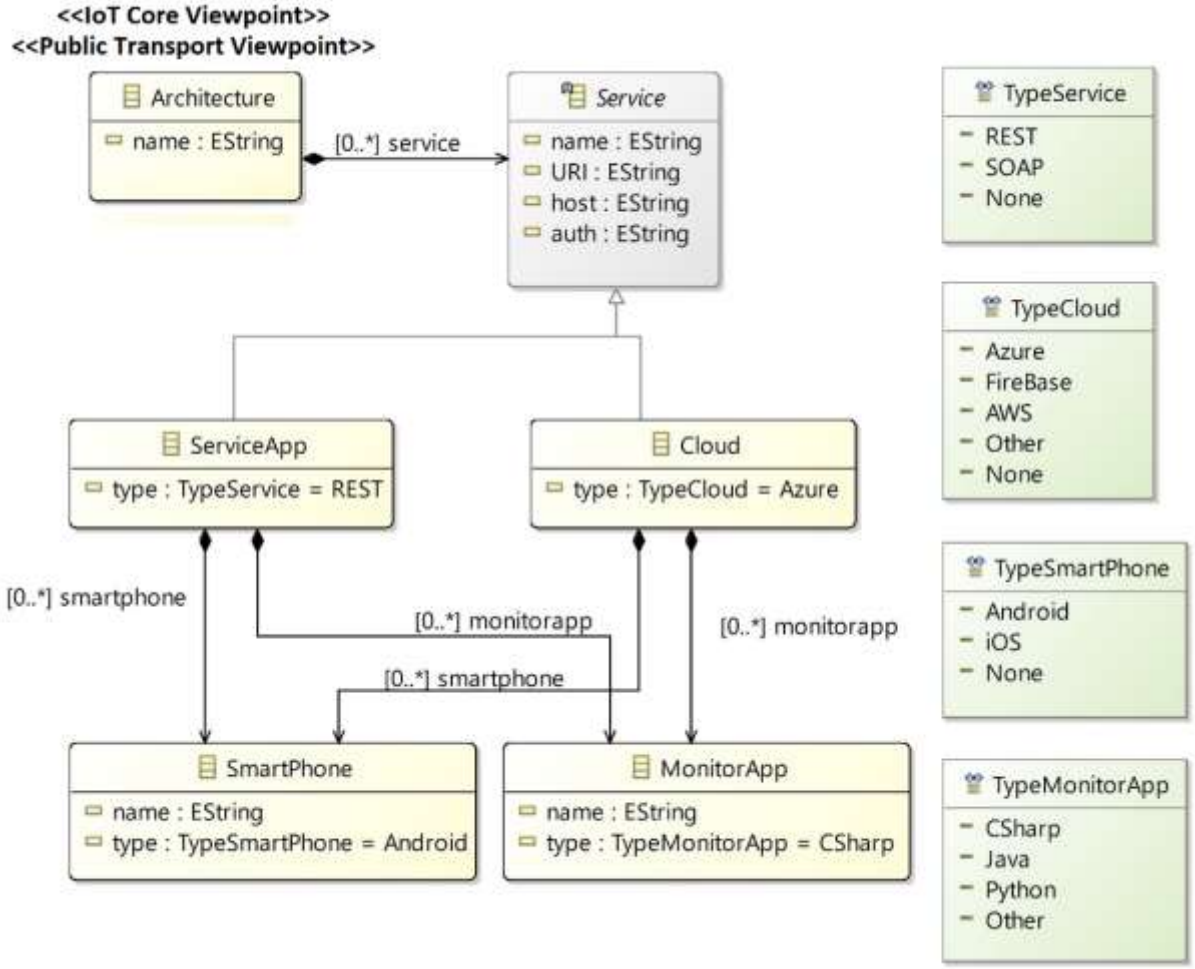

Figure 3. Service/Cloud viewpoint.

In order to develop concrete syntax, firstly, graphical representations for abstract syntax meta elements were prepared in this study. The Sirius tool is used to connect the nodes to the field concepts in the Ecore file. Tables 1, 2, and 3 provide graphical representations of all identified viewpoints of the DSML4PT. The left columns in the tables contain the names of the meta elements in the abstract syntax, while the right columns show the symbols in the DSML4DT syntax.

Ecore models are created in Sirius, so elements and their relationships can be easily seen in visual diagrams. Ecore models are used in the development process. During this process, symbols are set for both palettes and shapes. Icon geometry is explained and some constraint checks are considered. The designed system is a graphical editor where public transportation IoT developers can design models for each of the required viewpoints that match the DSML4PT concrete syntax.

DSML4PT language's syntax implements some restrictions/checks during the user modeling process. Some of these controls come from the metamodel and some from the UI tool. These controls are discussed in the following two subsections, "Model Constraints" and "Graphical Tools", respectively. Rules for validating the public transportation models are discussed in the third sub-section.

Table 1. Concrete syntax concepts and icons for the IoT Core viewpoint

Concept	Icon	Concept	Icon
Architecture		Output	
Controller		Input	
Communication		Language	
Port		Actuator	
Sensor			

Table 2. Concrete syntax concepts and icons for the Public Transport viewpoint

Concept	Icon	Concept	Icon
Architecture		Standards	
Soc		Protocols	
Communication		Port	
SpecialHardware		Sensor	
OperatingSystem		Output	
WindowSystem		Input	
Language		Actuator	

Table 3. Concrete syntax concepts and icons for the Service/Cloud viewpoint

Concept	Icon	Concept	Icon
Architecture		Cloud	
Service		SmartPhone	
ServiceApp		MonitorApp	

4.1. Model Constraints

Constraints for DSML4PT Ecore models are provided for instance models from all viewpoints. These restrictions can be classified as follows:

Splitting constraints: The compositional relationship between meta-elements in Ecore is a relationship from which another element is produced. For example, *Communication* and *OperatingSystem* elements can be generated from the SoC element in the IoT Core perspective. However, this does not happen for items where this relationship cannot be found.

Restrictions on the relationship count: The number of relationships between elements in the instance model is controlled based on the one-to-one, one-to-many, many-to-many relationships in Ecore. For example, only one *OperatingSystem* element can be derived from the *SoC* element. However, since the number of inputs and outputs can be higher, a large number of *Port* elements can be derived.

Relationship source and target constraint: The direction of the relationship defines the source and target of the relationship. This restriction is defined at the Ecore level. For example, if the relationship between *SoC* and *Communication* is reversed, this structure cannot be created in the model.

4.2. Graphical Tools

In addition to the metamodel constraints, DSML4PT includes some editorial constraints that assist the user in the design while creating the system models. These restrictions are listed below:

Switching between viewpoints: This restriction establishes the unity of the system by switching between editors of different viewpoints. This structure provides a step-by-step creation of the system. The tool is quite flexible when creating diagram files. In the case of a user request, it is possible to create each of the viewpoint diagrams separately. For example, the user can design a Public Transport diagram without designing the IoT Core diagram.

Combination: This feature provides system unification for all elements. Editor diagrams are based on the metamodel's viewpoints. On the other hand, the system model should be considered as a whole combining all instances of the viewpoints. Therefore, any identified element is saved in a list for unique use during the entire modeling process. This is achieved by having a tree structure showing all the elements that can be included in any point of the view diagram. An example is a *Communication* element created in the IoT Core viewpoint editor. Since this *Communication* element should be used in the other viewpoints, e.g. Public Transport viewpoint, it is automatically added into these viewpoint models too. Elements can be used in diagrams by dragging and dropping from palettes as needed.

Relationship-element integrity: According to this constraint, removing an element created in any sample model will remove all relationships from the model. This ensures that the integrity of the entire model is maintained and that the model is consistent after these changes. The main constraints are the number of relationships, the source and target relationships, and the integrity of the relationship-element constraints.

4.3. Validation Rules

In addition to the restrictions and features described above, it is also possible to define validation rules for DSML4PT language's semantics. For example, if some elements must be present in a DSML4PT instance model, validations can be added for them. Tables 4, 5 and 6 list a total of 61 validation rules defined for this purpose. In the tables, the Element title shows which element the rule is associated with. The Type header indicates an "Error" that must be fixed or a "Warning" that needs attention. Error messages are displayed in the integrated development environment of the DSML4PT language according to these rules when the "Validate Diagram" operation is performed on DSML4PT models prepared according to the viewpoints.

5. Model-to-Text Transformation for Code Generation

Model-to-Text (M2T) transformation rules were applied automatically onto the DSML4PT instance models to generate executable code for the public transportation software. These rules, in other words, allow the creation of the translational semantics of our language. To support the interpretation of DSML4PT models, M2T transformation rules were written by using (Acceleo, 2018) in this study. Acceleo is a pragmatic implementation of the OMG MOF M2T Language (MTL) standard. It also provides an Eclipse plugin where Acceleo transformations can be written, parsed, checked and executed directly inside the Eclipse environment. Some examples of Acceleo M2T transformation rules written for DSML4PT are discussed in Section 6.

It is worth indicating that the abovementioned M2T rules are applied at runtime on the public transportation system models conforming to the DSML4PT syntax and hence the code and/or files required for the system implementation are generated automatically. A user does not need to know the whole generation process including the details and the execution mechanism of the transformation rules.

The whole code generation process is abstract and there is no need to any human intervention. All code generation for languages such as C and Java that will be needed is in the Eclipse tool. The syntax rules of the batch overflow construct are also applied throughout the entire code generation phase.

Table 4. Validation rules for DSML4PT IoT Core viewpoint

Rule	Element	Type	Notification
IoTCore_1	Architecture	Error	Element name should not be empty!
IoTCore_2	Controller	Error	Element name should not be empty!
IoTCore_3	Controller	Error	Element type should be selected!
IoTCore_4	Controller	Error	At least one Communication element should be created!
IoTCore_5	Controller	Error	At least one Language element should be created!
IoTCore_6	Communication	Error	Element name should not be empty!
IoTCore_7	Communication	Error	Element type should be selected!
IoTCore_8	Communication	Warning	Ssid value should be entered.
IoTCore_9	Communication	Warning	Password should be entered.
IoTCore_10	Communication	Warning	Baudrate should be entered.
IoTCore_11	Port	Error	Element name should not be empty!
IoTCore_12	Port	Error	Element type should be selected!
IoTCore_13	Language	Error	Element name should not be empty!
IoTCore_14	Language	Error	Element type should be selected!
IoTCore_15	Actuator	Error	Element name should not be empty!
IoTCore_16	Actuator	Error	Element type should be selected!
IoTCore_17	Sensor	Error	Element name should not be empty!
IoTCore_18	Sensor	Error	Element type should be selected!

6. Case Study

In this section, we discuss the use of DSML4PT language and its IDE by taking into account the design and implementation of a fare collection system where validator devices are used in the public transportation vehicles. An iMX53 (NXP Semiconductor, 2017) series ARM core microprocessor is used in this validator device. It was a validator device with 1GB memory and 512MB hard disk space. There was a PN5180 (NXP Semiconductor, 2020) card reader circuit working with the Serial Peripheral Interface (SPI) in the device. This circuit was modeled as a sensor input and receives card reading information. In addition, there was a MAX9768 (Maxim Integrated, 2016) audio output circuit with Inter-Integrated Circuit (I2C) working as an actuator in the device and a 7” Liquid Crystal Display (LCD) output working with Low Voltage Differential Signal (LVDS) interface. The HE910 module was used as the GSM module.

6.1. System Modeling for the Device

Models for the validator device were created using the DSML4PT’s syntax for each modeling viewpoint. In DSML4PT’s IDE, any required modeling elements can be dragged-and-dropped from a palette including all visual notations of the entities for each DSML4PT viewpoint (see the right side of Figure 4). Hence any instance of modeling elements can be created and moved on the public transportation model currently designed by the user. For instance, the model instance created for the IoT Core view of the validator system is given in Figure 4. Embedded Linux Kernel V4 operating system was used. In addition, 2 port SAM hardware specific to public transportation, (ITxPT, 2017) standard and CCTalk protocol were used. The model created for the Public Transport viewpoint which includes all these structures specialized for the public transportation, can be seen in Figure 5. This validator system receives data via a REST service. The validator service coded with Java can communicate with Android and iOS smartphones. In addition, an administrator application coded in Java language provides system monitoring and management using the validator service. Hence, the Service/Cloud model according to these specifications was designed as can be seen in Figure 6.

Table 5. Validation rules for DSML4PT Public Transport viewpoint

Rule	Element	Type	Notification
PubTrans_1	Architecture	Error	Element name should not be empty!
PubTrans_2	Soc	Error	Element name should not be empty!
PubTrans_3	Soc	Error	Element type should be selected!
PubTrans_4	Soc	Error	At least one Language element should be created!
PubTrans_5	Soc	Error	An OperatingSystem element should be created!
PubTrans_6	Communication	Error	Element name should not be empty!
PubTrans_7	Communication	Error	Element type should be selected!
PubTrans_8	Communication	Warning	Ssid value should be entered.
PubTrans_9	Communication	Warning	Password should be entered.
PubTrans_10	Communication	Warning	Baudrate should be entered.
PubTrans_11	OperatingSystem	Error	Element name should not be empty!
PubTrans_12	OperatingSystem	Error	Element type should be selected!
PubTrans_13	OperatingSystem	Error	A WindowSystem element should be created!
PubTrans_14	SpecialHardware	Error	Element name should not be empty!
PubTrans_15	SpecialHardware	Error	Element type should be selected!
PubTrans_16	WindowSystem	Error	Element name should not be empty!
PubTrans_17	WindowSystem	Error	Element type should be selected!
PubTrans_18	WindowSystem	Error	A Language element should be created!
PubTrans_19	Language	Error	Element name should not be empty!
PubTrans_20	Language	Error	Element type should be selected!
PubTrans_21	Standards	Error	Element name should not be empty!
PubTrans_22	Standards	Error	Element type should be selected!
PubTrans_23	Protocols	Error	Element name should not be empty!
PubTrans_24	Protocols	Error	Element type should be selected!
PubTrans_25	Actuator	Error	Element name should not be empty!
PubTrans_26	Actuator	Error	Element type should be selected!
PubTrans_27	Sensor	Error	Element name should not be empty!
PubTrans_28	Sensor	Error	Element type should be selected!

Table 6. Validation rules for DSML4PT Service/Cloud viewpoint

Rule	Element	Type	Notification
Ser/Cloud_1	Architecture	Error	Element name should not be empty!
Ser/Cloud_2	ServiceApp	Error	Element name should not be empty!
Ser/Cloud_3	ServiceApp	Error	Element type should be selected!
Ser/Cloud_4	ServiceApp	Warning	URI should be entered.
Ser/Cloud_5	ServiceApp	Warning	Host should be entered.
Ser/Cloud_6	ServiceApp	Warning	Authentication should be entered.
Ser/Cloud_7	Cloud	Error	Element name should not be empty!
Ser/Cloud_8	Cloud	Error	Element type should be selected!
Ser/Cloud_9	Cloud	Warning	URI should be entered.
Ser/Cloud_10	Cloud	Warning	Host should be entered.
Ser/Cloud_11	Cloud	Warning	Authentication should be entered.
Ser/Cloud_12	SmartPhone	Error	Element name should not be empty!
Ser/Cloud_13	SmartPhone	Error	Element type should be selected!
Ser/Cloud_14	MonitorApp	Error	Element name should not be empty!
Ser/Cloud_15	MonitorApp	Error	Element type should be selected!

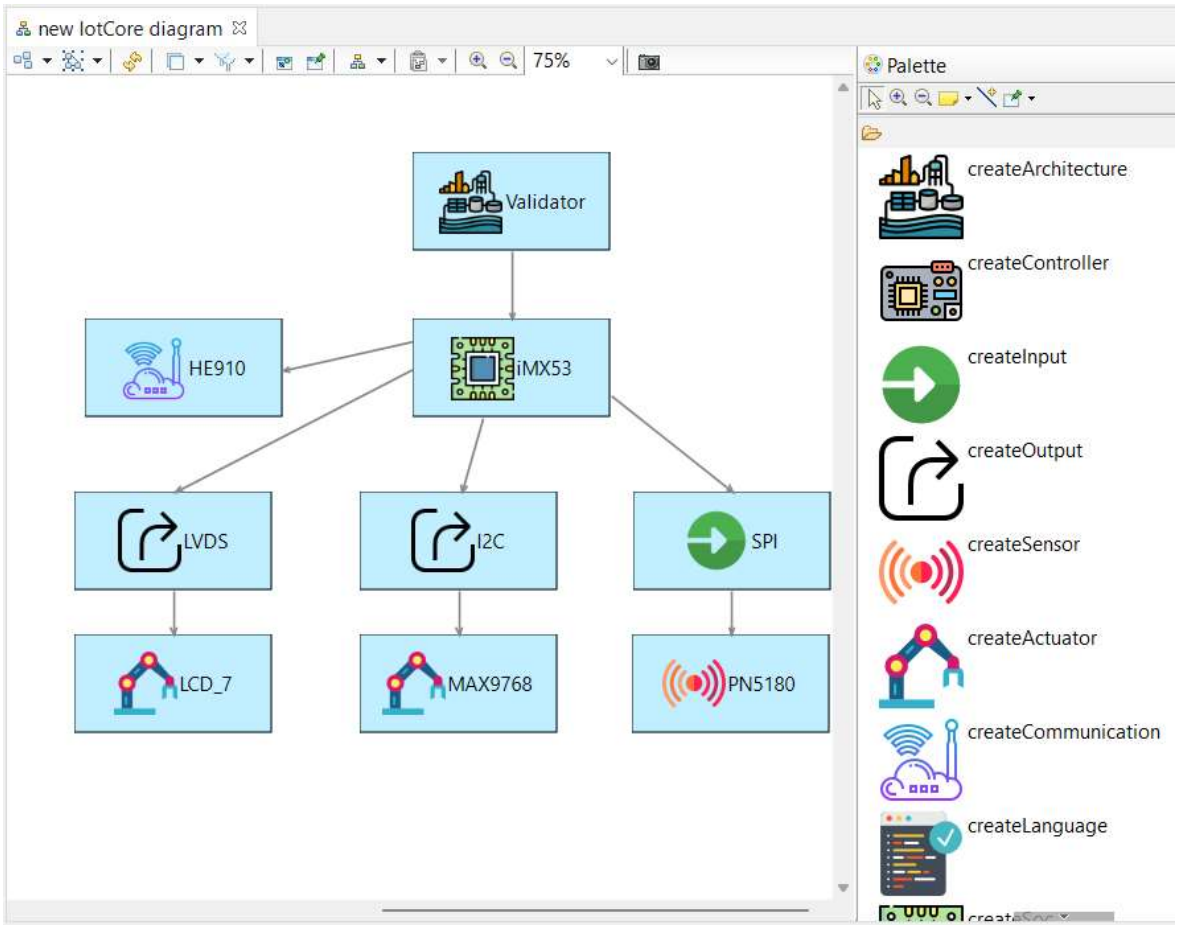

Figure 4. IoT Core viewpoint of the Validator IoT system.

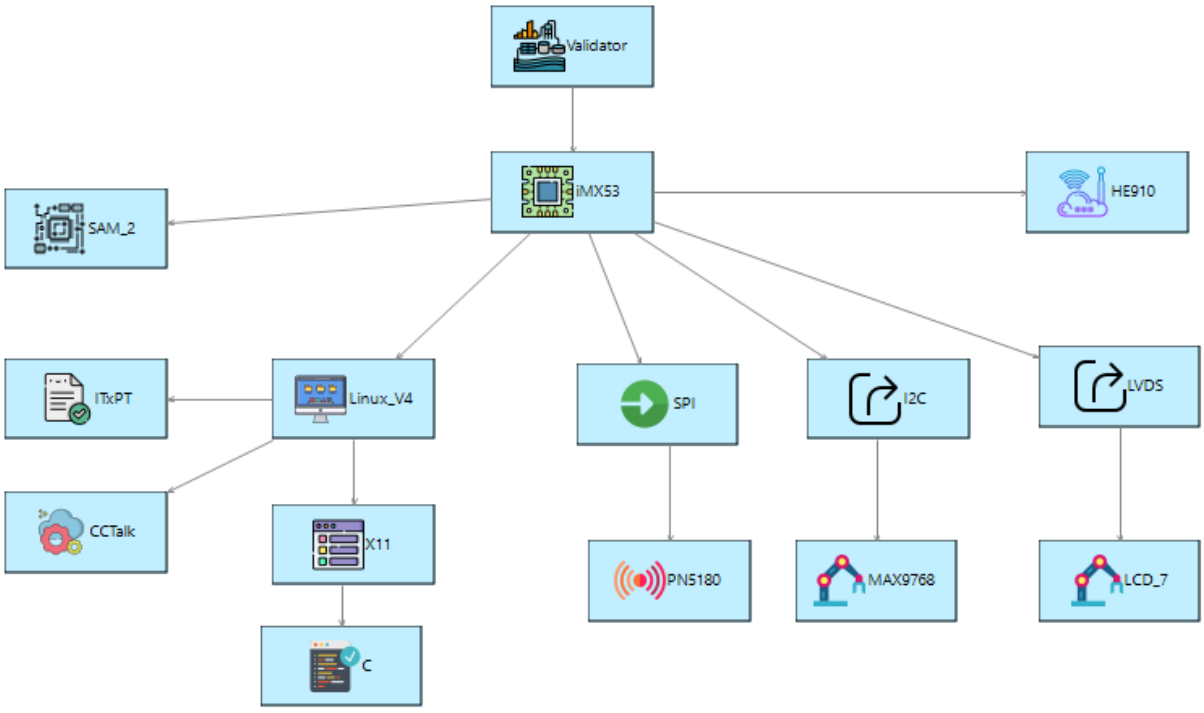

Figure 5. Public Transport viewpoint of the Validator IoT system.

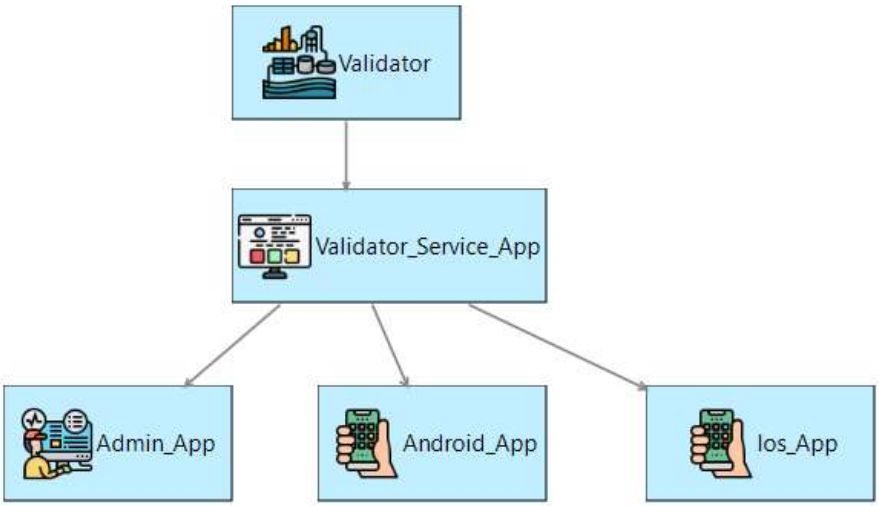

Figure 6. Service/Cloud viewpoint of the Validator IoT system.

6.2. Validation of the Modelled System

While creating public transportation models, the previously described constraints of DSML4PT's language are checked and the validation of the designed models is automatically performed. As mentioned before, there is a "Validate Diagram" selection inside the DSML4PT's IDE for this purpose. When it is selected, the software developer (user of the language) is notified with error messages, if there are situations contrary to the metamodel definitions and static semantic rules discussed in the previous section of this paper. For example, in Figure 7, an error can be seen during the modeling of the Validator device inside the Public Transport viewpoint. According to the PubTrans_1 rule of the public transportation metamodel, the Architecture node in the Public Transport diagram must have a name. The tool has identified a deficiency in this model element and this is reported to the software developer. Model design can only be completed after all these errors have been corrected.

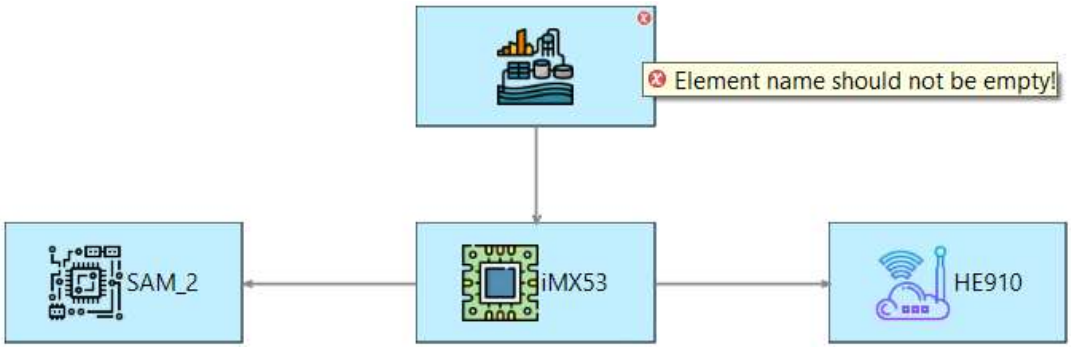

Figure 7. An example of a DT model validation.

6.3. Code Generation

A total of 4 code files were generated from the DSML4PT models designed for the entire Validator IoT system within the scope of our case study. These generated code files are: (1) GTK application suitable for Linux X11 window system written in C language to run on the Validator device; (2) REST service application written in Java language; (3) System administrator desktop application code written in Java language; (4) Client application code written in Java for Android smartphones. Figure 8 shows the names of the generated code files in the DSML4PT IDE's project view.

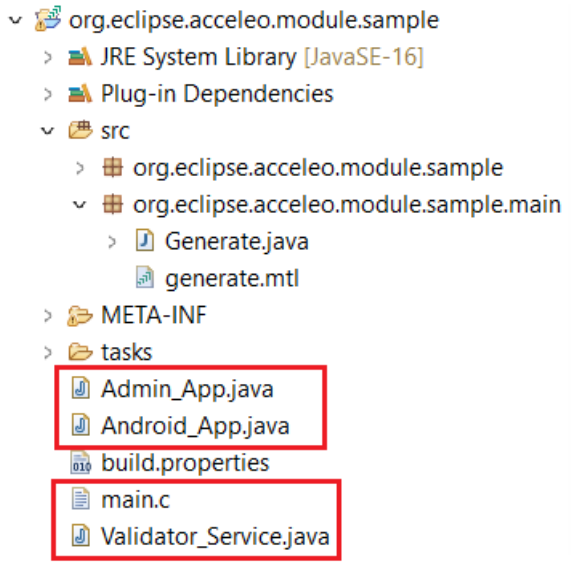

Figure 8. Generated code files shown in the tool project view.

As explained in the previous sections, the creation of the public transportation system starts with the *Architecture* element. This element should be present in the public transportation system and its properties should be specified. Hence, the generation of artifacts from the designed DSML4PT models starts from Architecture element instances. Listing 1 shows an excerpt from the Acceleo rules written for M2T transformations to generate code in our study. If the condition in Line 01 is met during parsing a DSML4PT model instance, the part between the [if] lines will be generated as the corresponding code. Listing 1 is the Acceleo code example to produce the Wireless LAN node discovery function. Listing shows that the code block inside the if structure will be generated in case there is a Wifi assignment. After the condition is met, the value of the ssid variable coming from the model is assigned in the code as in the 4th line. Listing 2 shows an excerpt from the auto-generated C code when the M2T rule given in Listing 1 was automatically applied to our Validator instance model.

```
01  [if (anArchitecture.soc.communication.type.Wifi->notEmpty())]
02  int wlan_detect() {
03          int i, ret;
04          char essid[20] = [anArchitecture.soc.communication.ssid/];
05          for(i = 0; i < 100; i++) {
06                  ret = WLGetESSID("wlan0", essid);
07                  if(ret == 0) {
08                          printf("Network found!\n");
09                  } else {
10                          printf("Network NOT found!\n");
11                  }
12                  printf("essid:%s\n", essid);
13                  sleep(1);
14          }
15          return 0;
16  }
17  [/if]
```

Listing 1: An excerpt from Acceleo code generating Wlan C code.

Listing 1 shows the Acceleo M2T example written for the automatic generation of Wireless LAN code corresponding to SoC and Communication elements in the Public Transport viewpoint of a DSML4PT instance. Similarly, code generation for all elements of the designed public transportation system model was successfully realized.

```
01    int wlan_detect() {
02        int i, ret;
03        char essid[20] = 654321;
04        for(i = 0; i < 100; i++) {
05            ret = WLGetESSID("wlan0", essid);
06            if(ret == 0) {
07                printf("Network found!\n");
08            } else {
09                printf("Network NOT found!\n");
10            }
11            printf("essid:%d\n", essid);
12            sleep(1);
13        }
14        return 0;
15    }
```

Listing 2: Excerpt from generated Wlan C code.

To provide the full implementation of the IoT Validator system, code generated by modeling should be completed by the users. Figure 9 lists the rate of manually added lines of code (LoC) in comparison with the auto-generated LoC for all viewpoints of the public transportation system. As can be seen from Figure 9, approximately 80% of the whole application was created by just modeling with DSML4PT. In other words, MDE with using DSML4PT enabled the auto-generation of a significant quantity of the software for the required system. We examined that the 20% of the application that could not be auto-generated from DSML4PT model instances consists of hardware specific configurations and a few IoT library definitions, which are too dependent onto the underlying hardware and naturally modeling such platforms-specific components could not be included with the platform-independent metamodel of DSML4PT to preserve the higher abstraction and extensive support for different execution platforms. A final note on the generation performance of DSML4PT can be added for the comparison of auto-generated LoC for each viewpoint. As can be seen again in Figure 9, the best code generation performance was obtained for the Public Transport viewpoint (85%). We believe that this slight increase in code generation encountered in this viewpoint is somehow related to DSML4PT's all-embracing model for the public transportation domain as well as the majority of the modeling public transportation model elements in the case study, i.e. the number of the IoT components, modules and their relations for the validator hardware is relatively more than the remaining IoT core and service elements in the system-to-be-implemented.

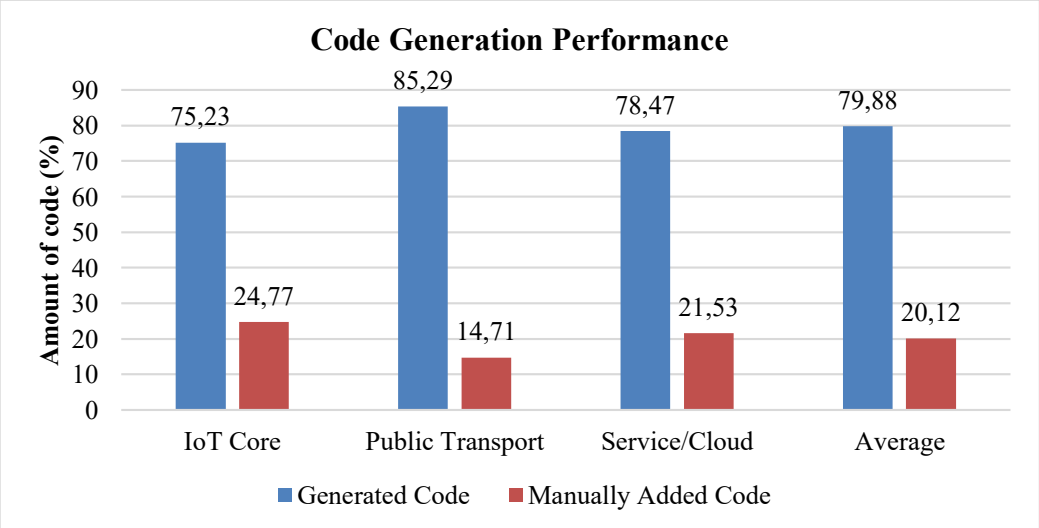

Figure 9. DSML4PT's code generation performance.

When the full implementation of the application was completed with manual code editions, it was tested and executed onto the validator hardware described at the beginning of this section for fare collections in public transportation. As intended, a charge was collected from the existing amount by reading the RF card on the device. A screenshot of this IoT application is given in Figure 10. As it can be seen from the screenshot, the fee of 10 Turkish Liras (TL) has been collected for the current metrobus trip from a passenger card having 120 TL and a balance of 110 TL has been obtained after the related transaction.

Figure 10. IoT validator application screen.

7. Conclusion and Future Work

Modeling IoT-based public transportation systems has been investigated and a DSML, called DSML4PT, has been introduced to support MDE of these systems. A metamodel including all entities and relations of IoT-based public transportation systems has been provided. Originating from this metamodel, we defined the syntax of DSML4PT language, so that instance models for the public transformation system software can be designed graphically from different system viewpoints. The semantics based on the model-to-text transformations led to the automatic generation of the code required for the implementation of the modeled systems. Based on the conducted study, we observed that almost 80% of an IoT-based public transportation application can be generated only with using DSML4PT.

In our future work, we aim at evaluating the use of DSML4PT during MDE of additional public transportation applications which will help the improvement and/or extension of the language features to leverage the support of IoT platforms, tools and hardware for the transportation systems. Another future work will be the construction of some sort of synchronization mechanisms between the DSML4PT models and the system implementations, i.e. any change made in the generated code can be automatically applied to the corresponding DSML4PT instance models and vice versa.

Researchers' Contribution Rate Statement

The authors' contribution rates in the study are equal.

Acknowledgment and/or disclaimers, if any

The study did not receive any support. There is no institution or person to thank.

Conflict of Interest Statement, if any

There is no conflict of interest with any institution or person within the scope of the study.

References

Acceleo (2018). Acceleo Model to Text Language. https://www.eclipse.org/acceleo/, (Access: 21 September 2023).

Ahmed, A., Kleiner, M. and Roucoules, L. (2019). Model-Based Interoperability IoT Hub for the Supervision of Smart Gas Distribution Networks. *IEEE Systems Journal*, 13 - 2.

Alulema, D., Iribarne, L. and Criado, J. (2017). A DSL for the Development of Heterogeneous Applications. *5th International Conference on Future Internet of Things and Cloud Workshops*, Prague, Czech Republic, 21-23 August.

Arslan, S. and Kardaş, G. (2021). The Need for Model-driven Engineering in the Development of IoT Software for Public Transportation Systems. *15th Turkish National Software Engineering Symposium (UYMS)*, Izmir, Turkey, 17-19 November.

Arslan, S., Ozkaya, M. and Kardas, G. (2023). Modeling Languages for Internet of Things (IoT) Applications: A Comparative Analysis Study. *Mathematics*, 11, 1263.

Asici, T. Z., Karaduman, B., Eslampanah, R., Challenger, M., Denil, J. and Vangheluwe, H. (2019). Applying Model Driven Engineering Techniques to the Development of Contiki-Based IoT Systems. *IEEE/ACM 1st International Workshop on Software Engineering Research & Practices for the Internet of Things*, Montreal, QC, Canada, 27-27 May.

Aydin, M.B., Oz, C., Cetin Tulazoglu, D. and Kardas, G. (2019). Development of an ITxPT compliant information system for public transportation vehicles. *Journal of Intelligent Transportation Systems and Applications,* 2:2, 1-13.

Barriga, J.A., Clemente, P.J., Pérez-Toledano, M.A., Jurado-Málaga, E. and Hernández, J. (2023). Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile. *Pervasive and Mobile Computing*, 89, 101751.

Berrouyne, I., Adda, M., Mottu, J.M. and Tisi, M. (2022). A Model-Driven Methodology to Accelerate Software Engineering in the Internet of Things. *IEEE Internet Things Journal*, 9, 19757–19772.

Betancourt, V.P., Liu, B. and Becker, J. (2020). Model-based Development of a Dynamic Container-Based Edge Computing System. *IEEE International Symposium on Systems Engineering*, Vienna, Austria, 12 October - 12 November.

Brambilla, M., Cabot, J. and Wimmer, M. (2017). Model-Driven Software Engineering in Practice, 2nd ed.; Switzerland: Springer.

Cai, H., Gu, Y., Vasilakos, A.V., Xu, B. and Zhou, J. (2018). Model-Driven Development Patterns for Mobile Services in Cloud of Things. *IEEE Transactions on Cloud Computing*, 6:3, 771-784.

Costa, B., Pires, P.F. and Delicato, F.C. (2020). Towards the adoption of OMG standards in the development of SOA-based IoT systems. *The Journal of Systems & Software*, 169.

Dautov, R. and Song, H. (2019). Towards IoT Diversity via Automated Fleet Management. *4th International Workshop on Model-Driven Engineering for the Internet-of-Things*, Munich, Germany, 15-17 September.

Evin, E., Aydin, M.B. and Kardas, G. (2020). Design and implementation of a CANBus-based eco-driving system for public transport bus services. *IEEE Access 2020*, 8:1, 8114-8128.

ENACT Project, (2018). https://www.enact-project.eu/ (Access: 21 September 2023).

Farias, C.M., Brito, I.C., Pirmez, L., Delicato, F.C., Pires, P.F., Rodrigues, T.C., Santos, I.L., Carmo, L.F.R.C. and Batista, T. (2017). COMFIT: A development environment for the Internet of Things. *Future Generation Computer Systems*, 75, 128–144.

Fernandez, G.C., Espada, J.P., García-Díaz, V., García, C.G. and Fernandez, N.G. (2014). Vitruvius An expert system for vehicle sensor tracking and managing. *Journal of Network and Computer Applications*, 42, 178-188.

Harrand, N., Fleurey, F., Morin, B. and Husa, K.E. (2016). Thingml: a language and code generation framework for heterogeneous targets. *The ACM/IEEE 19th International Conference on Model Driven Engineering, Languages and Systems*, Saint-malo, France, 2-7 October.

Hassine, T.B., Khayati O. and Ghezala, H.B. (2017). An IoT domain meta-model and an approach to software development of IoT solutions. *International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*, Gafsa, Tunisia, 20-22 October.

Hause, M., Hummell, J. and Grelier, F. (2018). MBSE Driven IoT for Smarter Cities. *13th Annual Conference on System of Systems Eng.*, Paris, France, 19-22 June.

Hu, M., Cao, E., Huang, H., Zhang, M., Chen, X. and Chen, M. (2023). AIoTML: A Unified Modeling Language for AIoT-Based Cyber-Physical Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, DOI: 10.1109/TCAD.2023.3264786.

Iovino, L., Sanctis, M.D. and Rossi, M.T. (2019). Automated Code Generation for NFC-based Access Control. *4th International Workshop on Model-Driven Engineering for the Internet-of-Things*, Munich, Germany, 15-17 September.

ISO 24014-1:2015 (2015). Public transport - Interoperable fare management system - Part 1: Architecture. https://www.iso.org/standard/61545.html (Access: 21 September 2023).

ITS Standardization (2021). Public transport Standards. https://www.itsstandards.eu/25-2/wp-2/ (Access: 21 September 2023).

ITxPT (2017). ITXPT Standards. Information Technology for Public Transport Organization. https://itxpt.org/ (Access: 21 September 2023).

Kardas, G., Ciccozzi, F. and Iovino, L. (2023). Introduction to the special issue on methods, tools and languages for model-driven engineering and low-code development. *Journal of Computer Languages*, 74, 1–2.

Kosar, T., Bohra, S. and Mernik, M. (2019). Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:1, 77–91.

Kotronis, C., Nikolaidou, M., Dimitrakopoulos, G., Anagnostopoulos, D., Amira, A. and Bensaali, F. (2018). A Model-based Approach for Managing Criticality Requirements in e-Health IoT Systems. *13th Annual Conference on System of Systems Engineering*, Paris, France, 19-22 June.

Kölsch, J., Post, S., Zivkovic, C., Ratzke, A. and Grimm, C. (2020). Model-based development of smart home scenarios for IoT simulation. *8th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems*, Sydney, NSW, Australia, 21 April.

Kühne, T. (2022). Multi-dimensional multi-level modeling. *Software and Systems Modeling,* 21, 543-559.

Lelandais, B., Oudot, M.P. and Combemale, B. (2019). Applying model-driven engineering to high-performance computing: Experience report, lessons learned, and remaining challenges. *Journal of Computer Languages,* 55:1, 100919.

Marah, H. M., Kardas, G. and Challenger, M. (2021). Model-driven round-trip engineering for TinyOS-based WSN applications. *Journal of Computer Languages*, 65.

Maxim Integrated (2016). MAX9768 10W Mono Class D Speaker Amplifier with Volume Control. https://www.maximintegrated.com/en/products/analog/audio/MAX9768.html (Access: 21 September 2023).

Mazzini, S., Favaro, J. and Baracchi, L. (2015). A Model-Based Approach Across the IoT Lifecycle for Scalable and Distributed Smart Applications. *IEEE 18th International Conference on Intelligent Transportation Systems*, Gran Canaria, Spain, 15-18 September.

Mohamed, M.A., Kardas, G. and Challenger, M. (2021). Model-driven engineering tools and languages for cyber-physical systems - A systematic literature review. *IEEE Access*, 9:1, 48605-48630.

Muthukumar N., Srinivasan, S., Ramkumar, K., Pal, D., Vain, J. and Ramaswamy, S. (2019). A model-based approach for design and verification of Industrial Internet of Things. *Future Generation Computer Systems*, 95, 354–363.

NXP Semiconductor (2017). i.MX 6Dual/6Quad Applications Processor Reference Manual, IMX6DQRM Rev. 4, 09/2017.

NXP Semiconductor (2020). PN5180 Full NFC Forum-Compliant Frontend IC. https://www.nxp.com/products/rfid-nfc/nfc-hf/nfc-readers/full-nfc-forum-compliant-frontend-ic:PN5180 (Access: 21 September 2023).

Rafique, W., Zhao, X., Yu, S., Yaqoob, I., Imran, M. and Dou, W. (2020). An Application Development Framework for Internet-of-Things Service Orchestration. *IEEE Internet of Things Journal*, 7:5, 4543-4556.

Rayes, A. and Salam, S. (2019). Internet of Things From Hype to Reality: The Road to Digitization. Cham, Switzerland: Springer.

Sosa-Reyna, C. M., Tello-Leal, E. and Lara-Alabazares, D. (2018). Methodology for the model-driven development of service oriented IoT applications. *Journal of Systems Architecture*, 90, 15-22.

The Eclipse Foundation (2015). Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmp/ (Access: 21 September 2023).

The Sirius Project (2023). The Eclipse Sirius Modelling Project. http://www.eclipse.org/sirius/ (Access: 21 September 2023).

The Vorto Project, (2018). https://www.eclipse.org/vorto/ (Access: 21 September 2023).

Thramboulidis, K. and Christoulakis, F. (2016). UML4IoT: A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry*, 82, 259–272.

Vorapojpisut, S. (2018). Model-based Design of IoT/WSN Nodes: Device Driver Implementation. *International Conference on Embedded Systems and Intelligent Technology & International Conference on Information and Communication Technology for Embedded Systems*, Khon Kaen, Thailand, 07-09 May.

Wasowski, A. and Berger, T. (2023). Domain-Specific Languages: Effective Modeling, Automation, and Reuse. 1st ed.; Switzerland: Springer.

Xiao, R., Wu, Z., and Wang, D. (2019). A Finite-State-Machine model driven service composition architecture for internet of things rapid prototyping. *Future Generation Computer Systems*, 99, 473–488.