"This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884"

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

D3.4 Lowcomotive Integrations - Final Version

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Thomas Vyzikas

Work Package: WP3

Deliverable number: D3.4

Production date: October 31st 2022

Contractual date of delivery: September 30th 2022

Actual date of delivery: September 30th 2022

Dissemination level: Public

Lead beneficiary: British Telecom Plc.

Authors: Léa Brunschwig, Felicien Ihirwe, Panagiotis Kourouklidis, Joost Noppen

Contributors: The Lowcomote partners

Project Abstract

Low-code development platforms (LCPD) are software development platforms on the Cloud, provided through a Platform-as-a-Service model, which allow users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the need of non-programmers to develop personalised software, and focus on their domain expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCPDs to a new paradigm, Low-code Engineering Platforms (LCEPs). LCEPs will be open, allowing to integrate heterogeneous engineering tools, interoperable, allowing for cross-platform engineering, scalable, supporting very large engineering models and social networks of developers, smart, simplifying the development for citizen developers by machine learning and recommendation techniques. This will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate agility of our future professionals between the academic and industrial world.

Report Executive Summary

This document introduces an evolving design and implementation for a low-code development platform (LCDP) that can support low-code development activities in a range of application domains. The platform is intended to provide the base building blocks for creating specialised LCDPs in specific domains ranging from chatbots and data science to the Internet of Things. In this deliverable we report on the progress that has been made towards this goal and the future activities that are planned.

Contents

1 Introduction

The present document is a deliverable of the Lowcomote project (Grant Agreement n°813884), funded by the European Commission Research Executive Agency (REA), under the Innovative Training Networks Programme of the Marie Sklodowska Curie Actions (H2020-MSCA-ITN-2018). The purpose of this document is to provide an overview of the work towards app creation and integration as part of the Lowcomote research projects.

A core goal of the Lowcomote project is to explore the opportunities and potential of low-code development platforms (LCDP) in a variety of application domains. Among others the project explores domains ranging from chatbots and data science to the Internet of Things. Each of the areas naturally will cover domain specific concepts, workflows and challenges, but at the same time the Lowcomote project will explore common elements in these domains so that they can be supported by a uniform LCDP which in turn can be extended to support these specific domains. In this deliverable we report on the progress that has been made towards this goal and the future activities that are planned.

In this deliverable we will first cover the progress that has been made in defining approaches and driving towards base architectures for LCDPs within specific domains covered by the Lowcomote project and workpackages. This will cover progress in the domains of smart cities, data science and mobile applications. Second we will cover the overlap between these initial efforts and how this is shaping the future work that will lead to a uniform LDCP with integrations across various application domains.

2 Modelling on Mobile Devices

2.1 Introduction

Traditionally, modelling occurs either manually with a pen and paper or whiteboards, or directly on static environments such as desktop computers or laptops using modelling tools like the Eclipse Modeling Framework (EMF) [1]. However, the increasing number of mobile users along with the improvement of the capabilities of these devices, offer new perspectives for modelling. The main asset of such devices is the possibility to use them in mobility which permits modelling closer to the system under study. Moreover, the variety of embedded sensors and components (e.g. camera, GPS, gyroscope, ...) can enrich the specification of some domain requirements like geolocation data.

Some LCDPs propose to create mobile apps but they are commonly simple CRUD applications that interact with a backend server. The task 3.5 of work package 3 entitled "Low-code Development of Rich Collaborative Mobile Apps using Active DSLs" aims at enriching the supported apps with sophisticated features. These apps will have a graphical syntax where elements may be geolocated on maps or using augmented reality (AR). They will be able to interact with external services (e.g. weather, social networks, ...) or external devices (e.g. IoT, ...). They will support collaboration and enable hierarchies of users with permissions established according to their user roles. They might also adapt following contextual rules according to changing conditions like the time, the battery of the device or external events retrieved from APIs. Such rich collaborative apps will be deployed on mobile devices and traditional computers and could be used, for instance, in the domains of IoT, domotics, urban planning, tourism, and many more.

In this section, the following goals and corresponding contributions of task 3.5 will be presented:

- Definition of the state of the art of the use of modelling inside the mobile devices and the classification of these tools,
- Design and implementation of different DSLs to extend the notion of Active DSLs, including role-based access control, augmented reality and context,
- Implementation of app-based and web-based editors for the creation of Active DSLs,
- Integration of the solutions inside the common LCDP of the Lowcomote project.

The rest of this section is organized as follows. First, sub-section 2.2 introduces the notion of Active DSLs. Then, sub-section 2.3 will provide an overview of the results obtained in [2] regarding the state of the art of modelling on mobile devices. Next, the following sub-sections will deal with the main contributions which are related to the extension of the notion of Active DSLs with new DSLs. Sub-section 2.4 will sum up the results obtained in [3] with regard to the role-based access control DSL that was presented in deliverable 3.3. Sub-section 2.5 will present the work published in [4] related to augmented reality (AR) [5]. Sub-section 2.6 deals with the context DSL. Finally, sub-section 2.7 will wrap-up this section and discuss about the future work.

2.2 Background: Definition of the notion of Active DSLs

Generally, DSLs are deployed on desktop or web applications and do not offer fancy functionalities like an augmented reality concrete syntax or context adaptation. Moreover, only a few of thenm make use of collaboration and role-based access control. The novel notion of Active DSLs proposed by [6] can fill these gaps. Figure 1 represents the elements to describe and deploy Active DSLs, the elements in blue corresponds to the contributions detailed in the sub-sections 2.4, 2.5 and 2.6.

Active DSLs can be deployed on both desktop computers and mobile devices, however, the latter ones allow to use functionalities which are only relevant in mobility. An Active DSL is defined by a domain meta-model which is annotated by a set of "annotation meta-models" to enrich the domain meta-model with new sophisticated functionalities. The annotation models conform to the meta-models in the upper part of Figure 1, we will describe the three first ones in this section and the following ones in other sub-sections.

Active DSLs can have any kind of concrete syntax but we encourage the use of graphical syntax to embrace the idea of low-code development platforms and facilitate their use by end-users.

Active DSLs have the ability to interact with external systems. The external interaction meta-model offer the possibility to communicate with two kinds of external elements: external services (e.g. web services) and external devices (e.g. IoT devices).

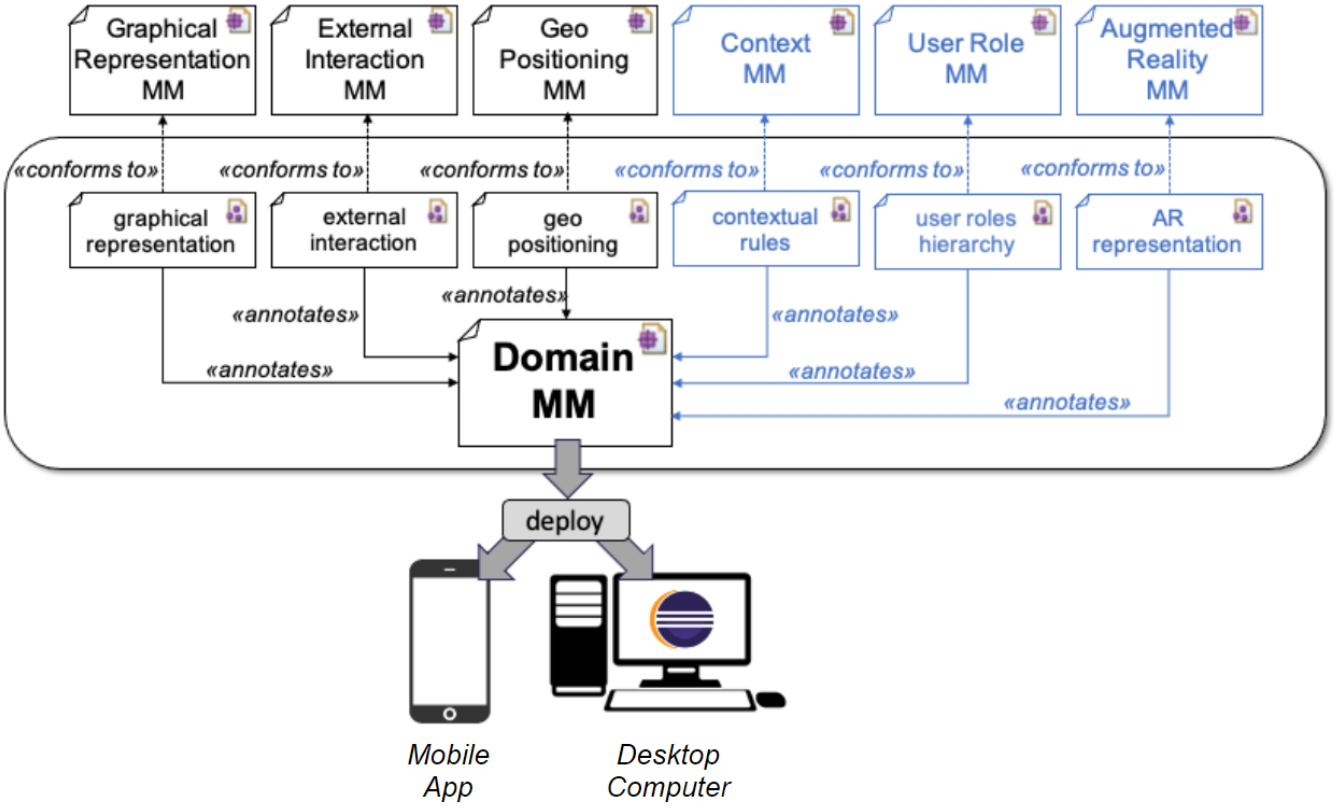


Figure 1: Elements to describe and deploy Active DSLs.

Finally, Active DSLs can profit from geolocation of some model elements in a map and can represent the DSL users in the model to identify their current position thanks to a geo-positioning meta-model.

In summary, Active DSLs are meant to be used on mobile devices to take advantage of their full potential. They are configured by a set of annotation models which enrich the domain meta-model of the user with new functionalities such as geolocation, context-awareness, interaction with external systems and many others.

2.3 State of the Art

To understand the current state of the art of mobile modelling tools, their features and the rationale of the approaches, we have conducted a systematic mapping study (SMS) [7] of the academic literature, as well as an analysis of the existing tools found in digital app distribution platforms (e.g. App Store for iOS and Google Play for Android). We retrieved a set of 888 papers and selected only 36 relevant papers published between 2005 and 2020, and we included 22 tools from the digital platforms out of 29 downloaded apps. The complete study has been reported in [2].

2.3.1 Research Method

According to the SMS method, we first identified the relevant keywords of our topic to compose the search strings. In our case, we built a search string using the lexical fields of model-driven engineering (MDE) and mobile development, for a total of 17 terms. We then applied these search strings to the following databases: Scopus, Web of Science, ACM Digital Library, IEEE Xplore and SpringerLink. We performed this query in November 2020, and we considered the results in English that included at least one term of each area in their title, abstract and keywords. Similarly, we repeated this process within distribution stores to obtain a panorama of the current practice in December 2020 on App Store and Google Play. The tools were selected when they were rated over 3 stars, downloaded at least 50.000 times, supported modelling and had a free version.

We classified the analysed tools for modelling on mobile devices according to three dimensions, developing the approach from [6] (see Figure 2):

- Modelling language,

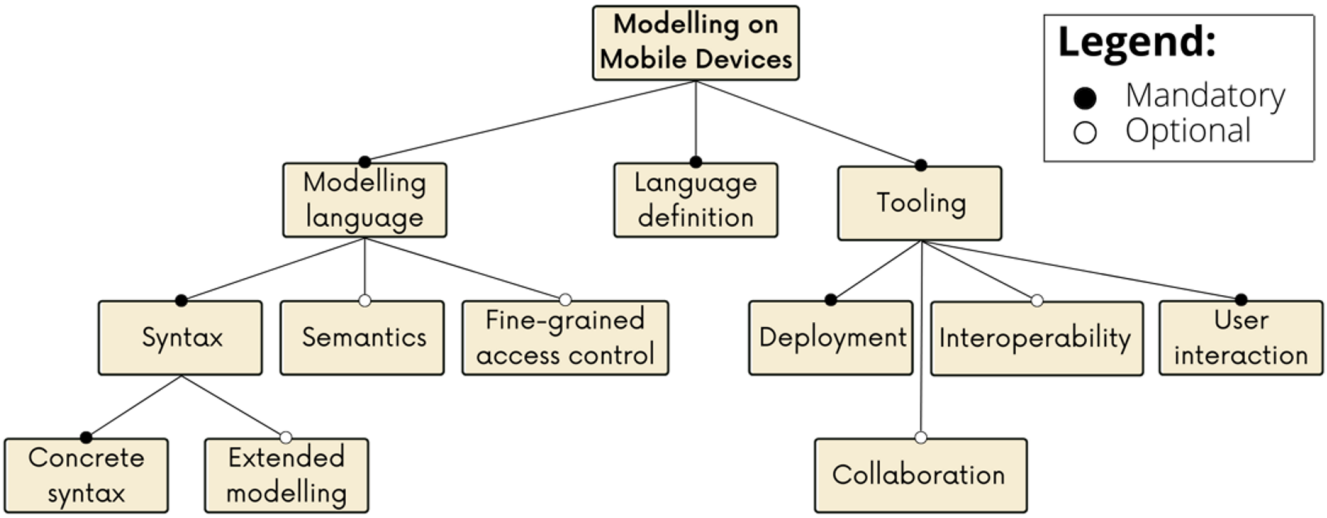


Figure 2: Dimensions of modelling tools for mobile devices.

- Language definition,
- Tooling.

2.3.2 Modelling Language

This dimension tackles the aspects related to the language(s) supported by the tool. This includes the style of its syntax and the extended modelling capabilities to translate informal model information along with its semantics or the ability to support fine-grained access control to language elements for different users.

2.3.2.1 Concrete syntax Languages have an abstract syntax which defines their primitives, properties, relationships and constraints regarding their specific use and they have a concrete syntax which describes how the models will be visualised. To be used on mobile devices, the concrete syntax has to be thought of according to the user interaction and experience using the particularities of the device. Figure 3 shows a classification of concrete syntaxes for mobile devices and screenshots of representative apps of some categories. A graphical syntax can be geolocated, the elements of the model are displayed on a map and the user can also be represented at its current position (user representation). The screenshot from DSL-comet [3] in Figure 3a illustrates it. Flexisketch [8] in Figure 3b uses sketching to mimic the pen-and-paper feel of traditional modelling using a touch screen. With augmented reality (AR), model elements can be superimposed on a view of the real world using computer-generated images as illustrated by HoloFlow [9] in Figure 3c. Tabular syntaxes display the model in a matrix or menus. Textual syntaxes are the most common syntaxes for traditional languages but their use is discouraged for mobile devices due to the small screens of most mobile devices. Natural language concrete syntaxes allow modelling via text or voice. In Socio [10], the user chats with a bot using written natural language (see Figure 3d).

According to the results illustrated by the diagrams in Figure 3, the great majority of the tools are using graphical concrete syntaxes and often sketching. However, geolocated syntaxes and augmented reality would be beneficial for on-site modelling. Syntaxes using natural languages could simplify the user interaction for modelling especially for non-modelling experts. A combination of different syntaxes could also cope with the idea of multi-experience development platforms (MXDPs) [11].

2.3.2.2 Extended modelling We call extended modelling the ability to include drawings or annotations on top of the model to enrich it with additional meaning (see Figure 4). Drawings designate the sketching, like circling or pointing an element by drawing an arrow, over a model. Annotations are pieces of information which can be attached to a model element in the form of symbols, images, or text, . . . According to our obtained results depicted in the graph from Figure 4, extended modelling features are poorly supported in mobile modelling apps.

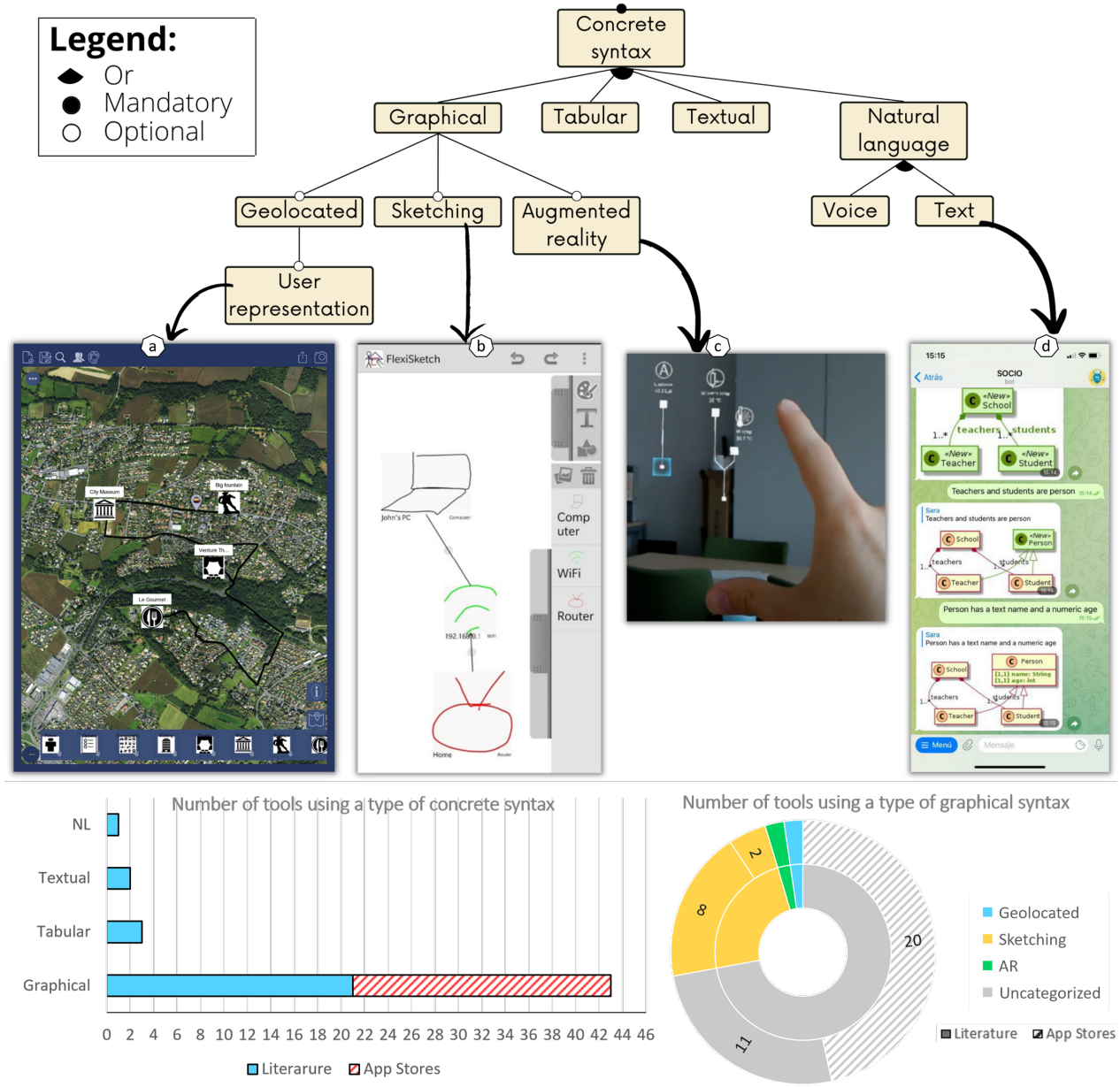


Figure 3: Feature model for concrete syntax of modelling languages on mobiles with screenshots of (a) DSL-Comet [3] (b) Flexisketch [8] (c) HoloFlow [9] (d) Socio [10] and diagrams of the results.

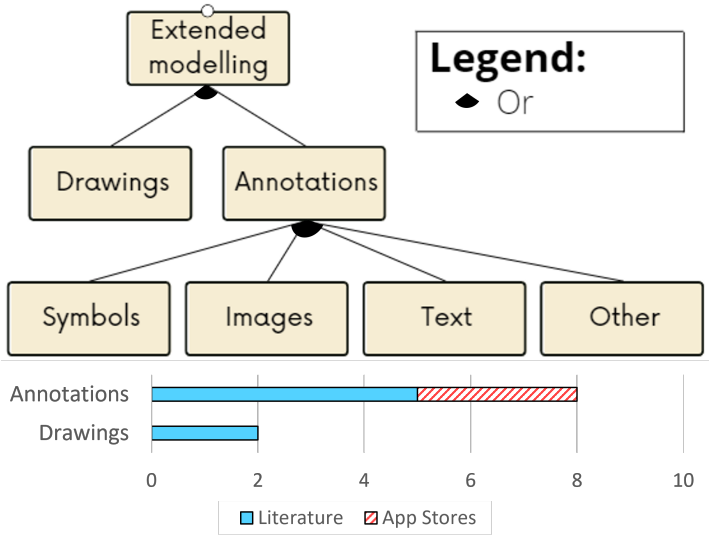


Figure 4: Feature model for extended modelling on mobiles and diagram of the results.

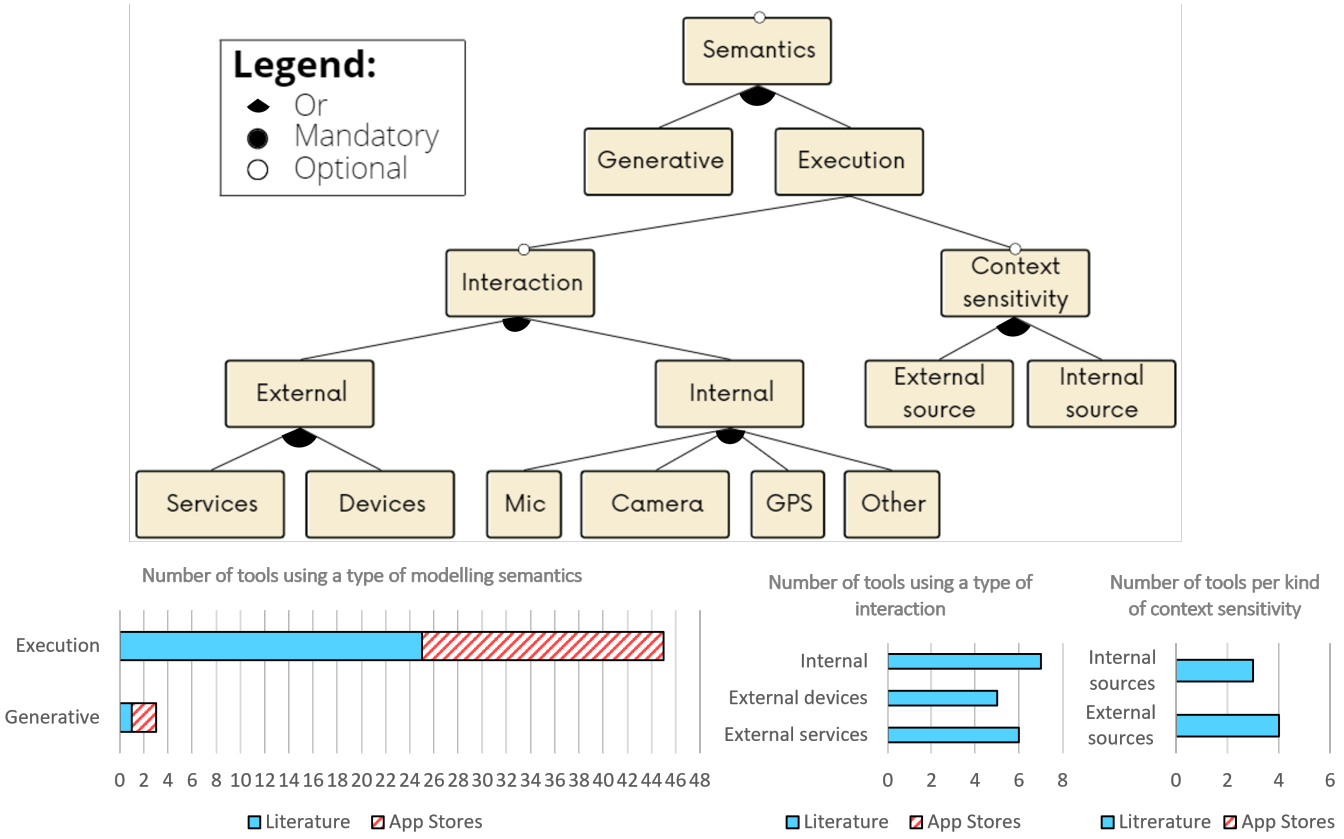


Figure 5: Feature model for semantics of modelling languages on mobiles and diagrams of the results.

2.3.2.3 Semantics Semantics conveys the meaning of a model. Figure 5 depicts the corresponding feature diagram along with the results obtained according to this feature. Semantics can either be generative, in this case, it is implicitly defined via code generation, or it can be based on execution via simulation or model interpretation. We identified two pertinent features of the model execution: interaction and context sensitivity. The first one is the ability to interact with external services, external devices, or internal features of the mobile device. The latter is the ability to adapt the model during execution according to an external or internal source of context. Even if it is an optional feature, Figure 5 shows that many tools propose semantics and the great majority via execution. However, only a few provide external interaction via web services or IoT devices and context adaptation is largely unexplored.

2.3.2.4 Fine-grained access control Finally, the modelling language dimension tackles mechanisms to provide fine-grained access control. A model may need to be used in collaboration with different stakeholders and access to some parts of the model should be controlled. A modelling language might need to define privileges for different categories of users. There is a gap for this feature and there is only the vision paper for the tool DSL-Comet [3] which presents some steps in this direction.

2.3.3 Language Definition

Previously, we have analysed the multiple aspects of the modelling languages. In this dimension, we study how these languages are defined.

Figure 6 shows a dichotomy between the tools that support fixed languages and those allowing for the creation of their modelling languages (meta-modelling support). In the latter case, the definition of the language can either be done internally, on the mobile device, or externally, likely in a web-based or desktop application. According to our study, tools from app stores are more likely to support mind maps and flowcharts, while approaches from the literature deal mostly with tools for the software design domain targeting BPMN or UML class diagrams. Better coverage of other diagrams is currently lacking. It could be improved by better metamodelling support but only 6 tools offer this possibility.

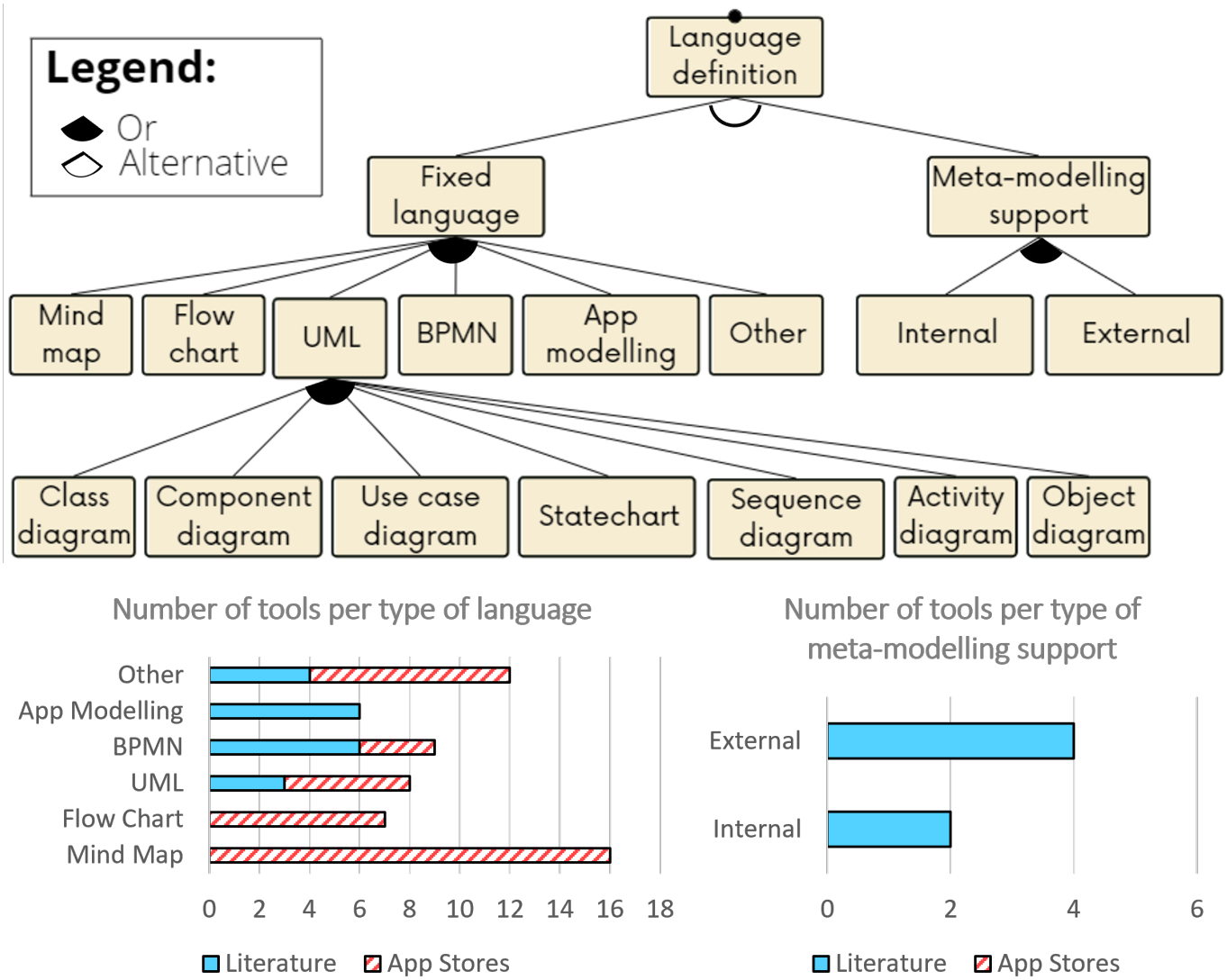


Figure 6: Feature model for language definition dimension and diagrams of the results.

2.3.4 Tooling

We have analysed the design of a modelling language of a mobile modelling solution, we will now focus on the modelling tool itself. We have identified several tool features illustrated in Figure 2 the deployment, the collaboration support, the interoperability mechanisms and the user interaction.

2.3.4.1 Deployment The deployment of a modelling tool has a direct impact on its usability but also on the targeted audience. We tackle this feature by analysing the architecture and the deployment target as depicted in Figure 7. The architecture of a tool can either be one-tier, the tool runs stand-alone, or client-server, the tool is a client and communicates with a server. The deployment target can be a desktop computer, a mobile device and/or the web. Regarding mobile devices, we make the distinction between the target platforms (Android, iOS, Windows Phone) or whether they can run on different platforms (cross-device). Other refers to tools on head-mounted mobile devices or uncommon phone platforms. The tools which do not use external services or remote collaboration typically have a one-tier architecture while other tools use client-server architectures when they have a backend, a remote database or interactions with external services. Most of the tools target smartphones, and only two are specific to head-mounted devices.

2.3.4.2 Collaboration Modelling is often a collaborative activity which involves many stakeholders. As depicted in Figure 8, we are interested in the type of collaboration, whether it is taking place synchronously in the same location (serverless), remotely (server-based) or asynchronously where collaborators do not work at the same time. We also study the awareness in the collaboration where the different actors might have a different level of information about what each other is doing. According to the results depicted in Figure 8, about one-third of the mobile modelling tools are offering collaboration but we miss synchronous

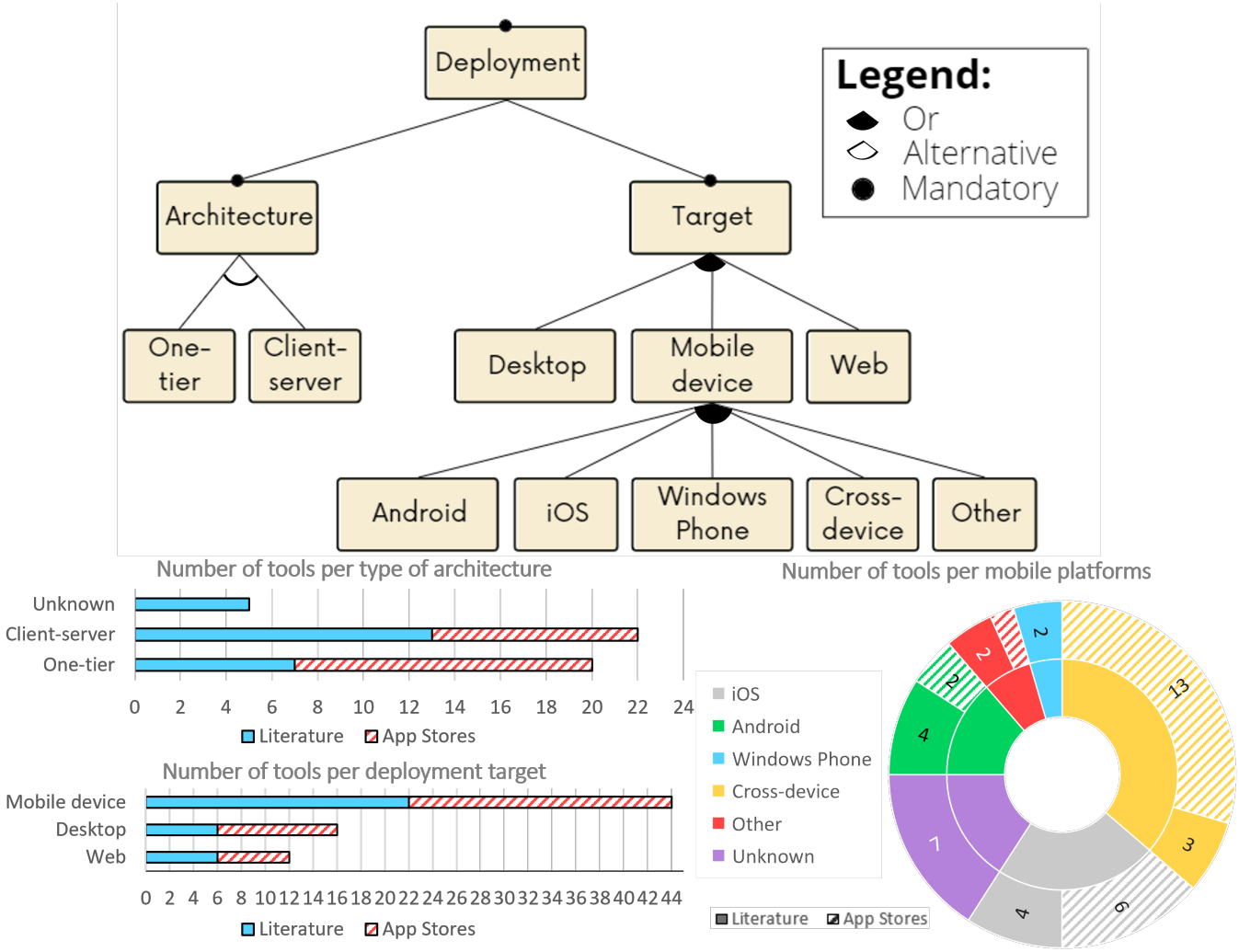


Figure 7: Feature model for tool deployment and diagrams of the results.

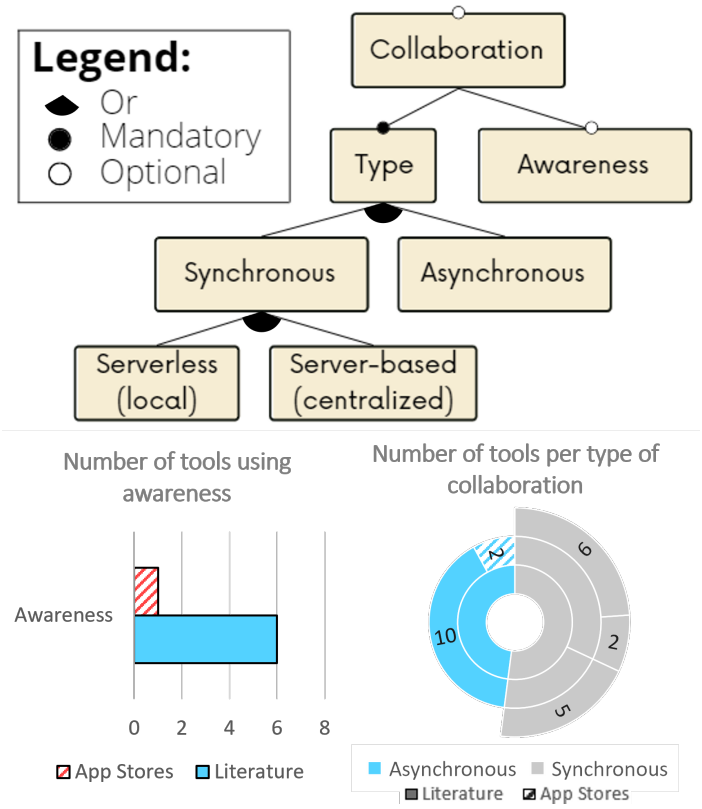


Figure 8: Feature model for collaboration on mobile tools and diagrams of the results.

collaboration approaches as well as remote collaboration.

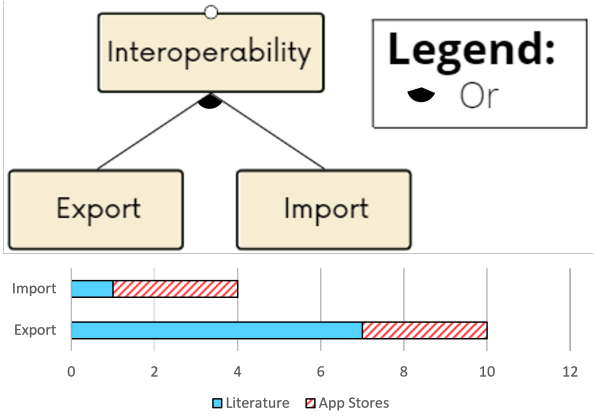


Figure 9: Feature model for interoperability of mobile modelling tools and diagrams of the results.

2.3.4.3 Interoperability The interoperability is the ability for a model built by a tool to be exported to other platforms and, reciprocally, the ability of the mobile modelling tool to import models built by other platforms (see Figure 9). This feature is determinant for the success and acceptance of a modelling tool however it is missing in most of the tools, especially for the import. Therefore, models implemented on other tools have to be redone on the mobile device and vice versa.

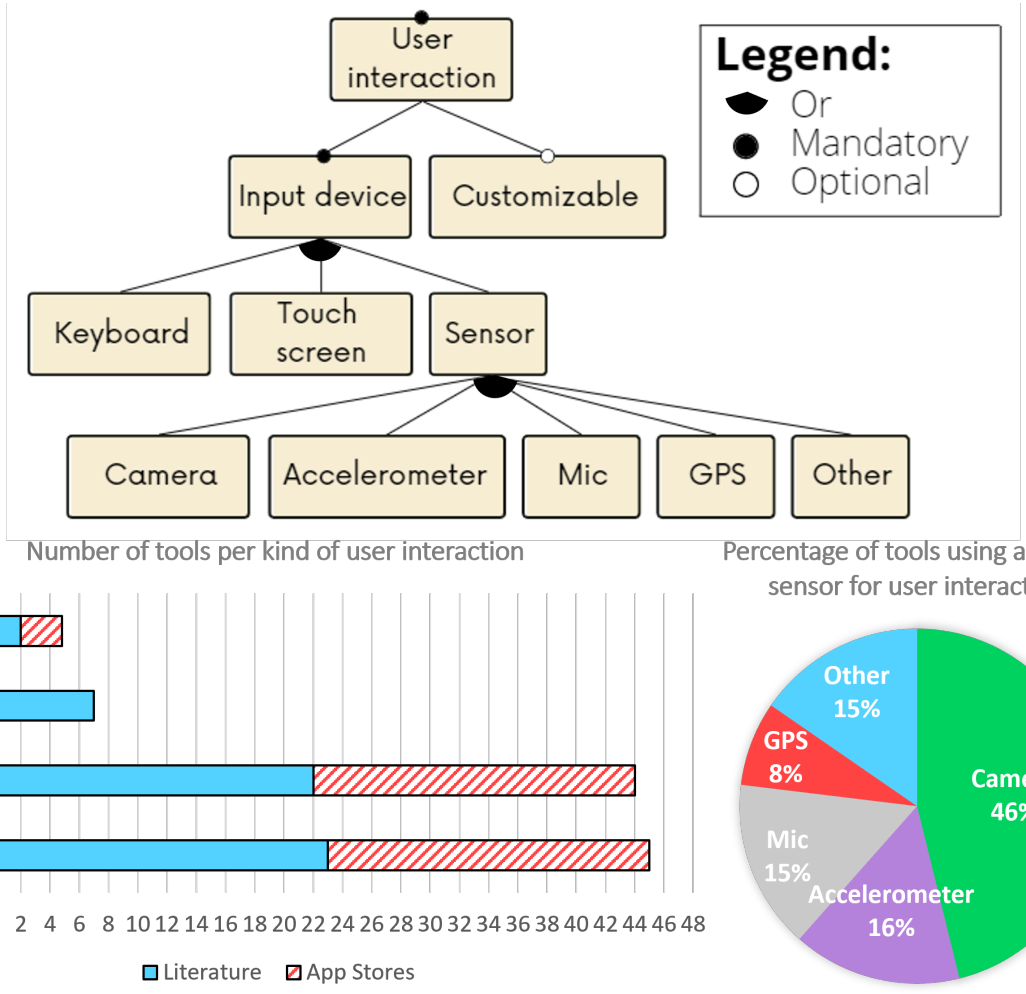


Figure 10: Feature model for user interaction in tooling and diagrams of the results.

2.3.4.4 User interaction Traditionally, to build models on desktop applications we use a keyboard and a mouse but on mobile devices, there is no mouse, instead, we use the touch screen. Moreover, modelling on mobile devices can bring many other possibilities for interacting as it is depicted in Figure 10, by using the

sensors of the device. In addition, we investigated if the interaction within the tools was customizable, for instance, the gestures needed to create/delete model elements. According to the results, the great majority of tools rely on the touch screen and the keyboard, but supporting several ways of interaction (e.g. voice, text, sketching) would support the idea behind MXDPs [11] and the use of sensors (e.g. GPS, accelerometer, ...) would be beneficial for on-site modelling. Mechanisms for customization are almost not present but they could improve the flexibility and efficiency of modelling on mobile devices.

2.3.5 Summary

We have presented a brief overview of our systematic mapping study on mobile devices. After analysing both approaches from academia and tools from app stores, we could detect the gaps and opportunities. Most approaches support graphical syntax, but advanced syntaxes, based on AR and natural languages, are barely used. Explicit model semantics is missing in most approaches despite the large number of possibilities brought by the rich set of components and sensors the mobile devices provide and can communicate with. Moreover, the large catalogue of sensors could lead to fancy user interaction but it is mostly limited to the keyboard and touch screen. Many approaches do not propose collaboration and fine-grained access control is nonexistent. Only a few tools offer language definition capabilities but the interoperability between the tools is mostly missing.

2.4 Role-based Access Control DSL

Some scenarios need to control the access to some part of the models or to secure their editing. For that purpose, the User Role meta-model of Figure 1 defines roles and permissions to add to Active DSls. This contribution has been described in [3] and was already introduced in the deliverable 3.3.

2.4.1 User Role DSL

The User Role MM permits to describe role hierarchies and permissions to secure the access and restrict the use of an Active DSL and its instances. Figure 11 shows the abstract syntax and an example of the User Role MM concrete syntax.

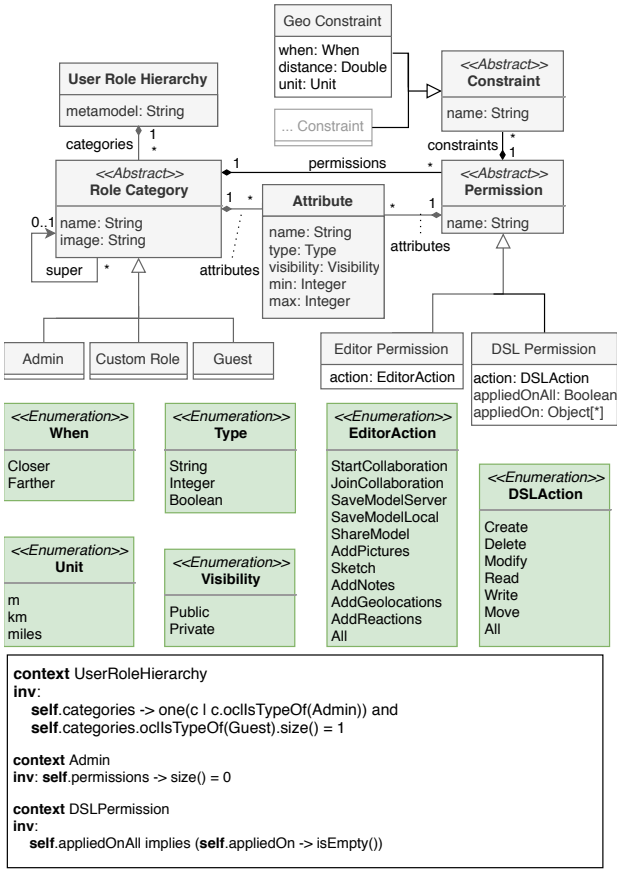
We distinguish the following three type of roles:

- *Admin* which refers to the author of the language and any granted users. Every Active DSL have to include exactly one role of this kind. It gives access to everyting with no restriction.
- *Guest* which refers to users who have not been assigned any other role. This is typically the role with the least permissions, and by default, it forbids access to the model. There is exactly one role of this type.
- *Custom Role* which refers to language-specific roles ("tourist" and "touristic guide" in the listing of Figure 11b).

In the same way, permissions are classified into two distinct groups which determine their action scope:

- *Editor permissions* are applied on functionalities of the editor e.g., allowing collaboration, model sharing, attaching pictures or reactions (visual alerts) to model elements, among others ("@DSL" in the listing of Figure 11b).
- *DSL permissions* concern the management (i.e., creation, deletion, modification, etc) of the elements of the domain meta-model. They can target classes as well as their attributes and references ("@Editor" in the listing of Figure 11b).

Permissions and role categories can have attributes that can be entered when the language is deployed. A permission could define the address or the name of an external service to restrict its exploitation by end-users. A role could define that tourists have a name and age, and touristic guides speak a number of languages. The value of this attribute should be informed for each particular user having the role. These attributes can either be public or private, which means a public attribute can be seen by all but a private one can only be seen by role higher in the hierarchy. The private attribute of an admin cannot be seen by anyone but the attributes of a guest can be seen by all.



(a) Abstract syntax of the User Role MM.

```
1  metamodel: "http://miso.es/dsls/tourismDSL.ecore"
2
3    Admin {
4      image: "http://www.miso.es/images/miso.png"
5    }
6
7    Guest{}
8
9    tourist {
10     // attributes
11     + string name
12     - integer age
13     // permissions
14     @DSL canRead (read) {
15       -> tourism.Cultural
16       -> tourism.Transportation
17       //...
18     }
19     @DSL giveOpinion (write) {
20       -> tourism.Opinion
21     }
22     @DSL changeOpinion (modify) {
23       -> tourism.Opinion.rate
24       -> tourism.Opinion.opinion
25     }
26     @Editor canAddPictures (AddPictures)
27             when closer 10 m
28   }
29
30   touristicguide extends tourist {
31     // attributes
32     + string[*] spokenLanguages
33     // permissions
34     @DSL touristicGuideCanDoAll(all) {
35       -> tourism.Museum
36       //...
37     }
38     //...
39   }
```

(b) Example of the User Role MM concrete syntax.

Figure 11: User role meta-model.

Finally, permissions can have contextual constraints for limiting them in certain situations. In Figure 11a, the class GeoConstraint is used in the listing of Figure 11b at line 27 and specifies that a tourist can add pictures only when they are within 10 metres of a location. The expansion of the role model with more contextual constraints will be the topic of future work with the use of the Context DSL of sub-section 2.6 for defining contextual rules.

2.4.2 Tool Support

The third goal of Task 3.5 concerns the implementation of an app-based and web-based editors for the creation of Active DSLs. We already have the former one which is called DSL-comet and we are working on the latter one.

2.4.2.1 DSL-Comet DSL-comet is a modelling editor aimed at supporting Active DSLs and it runs on iOS devices (iPhones and iPads). It can be installed from Apple's app store, its home page is https://diagrameditorserver.herokuapp.com and a demonstration video to illustrate some of its features is available at https://youtu.be/rzhl9yMFSxI. Figure 12 depicts the architecture of DSL-comet. Users can download the Active DSL definitions stored on a remote database, then start building models and store them either locally or in the database. DSL-comet provides an API broker service to manage interaction with external services and supports geo-services: models can be geo-positioned on the map and it can perform geolocation queries. DSL-comet website is used for managing the list of Active DSLs and their instances.

2.4.2.2 Web-based editor Figure 13 shows the architecture of our tool with the integration of the User Role DSL and some parts are still under development. On DSL-comet's app, users can consult their current role as well as the list of roles available for a specific meta-model however the behavior of roles and permissions needs to be implemented. On DSL-comet's website, the sign in and sign up functionalities have been added, the notion of user session, roles and user role models have been added to the database, and hierarchy of users and attribution of roles for a specific meta-model is possible. As a future work, we plan to finish the implementation of the User Role DSL in DSL-comet and to develop an Xtext editor in

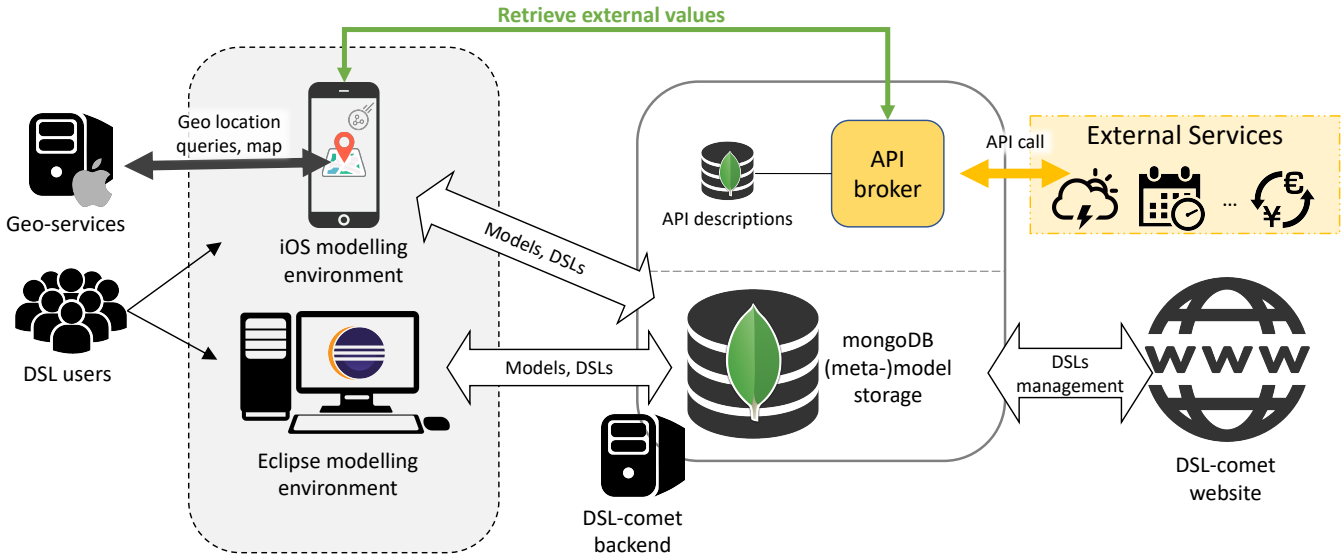


Figure 12: Architecture of DSL-comet.

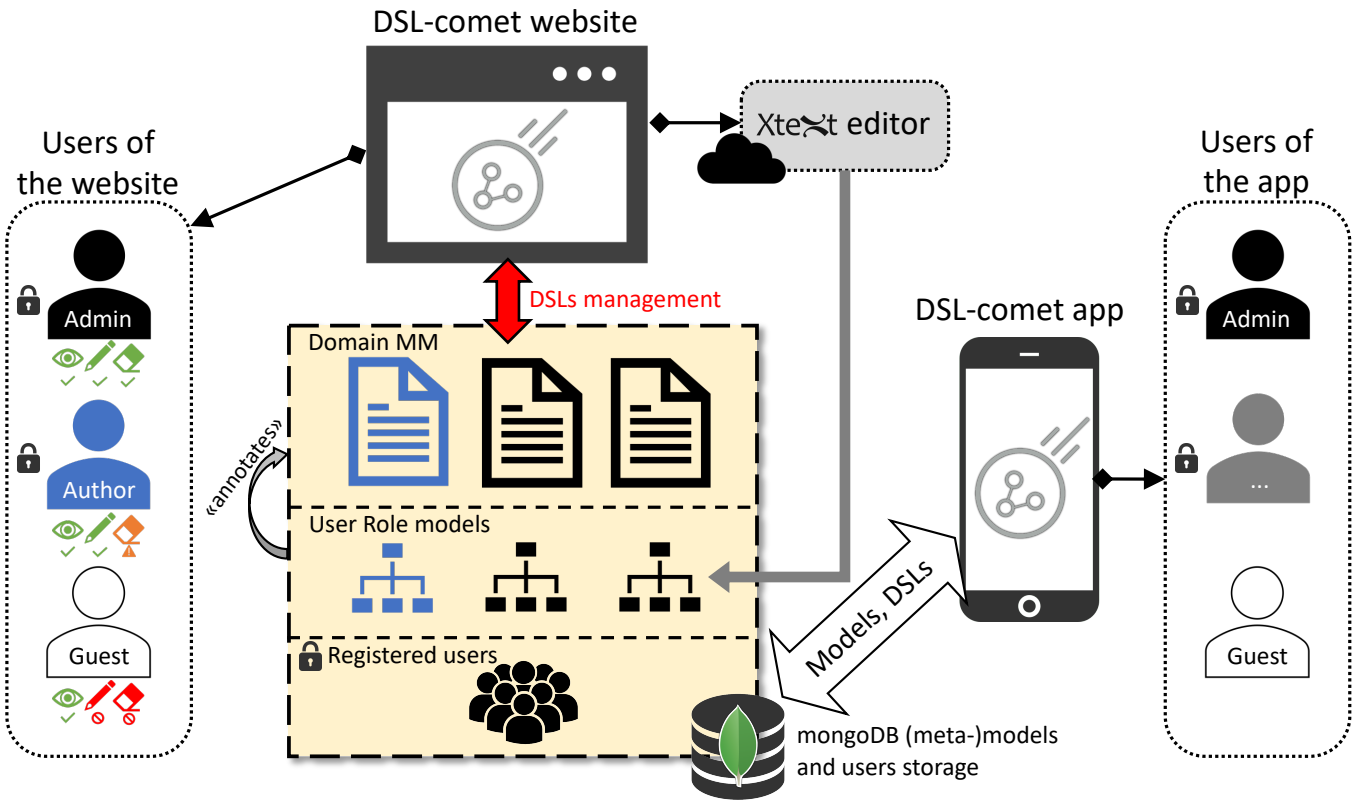


Figure 13: Management of roles within DSL-comet.

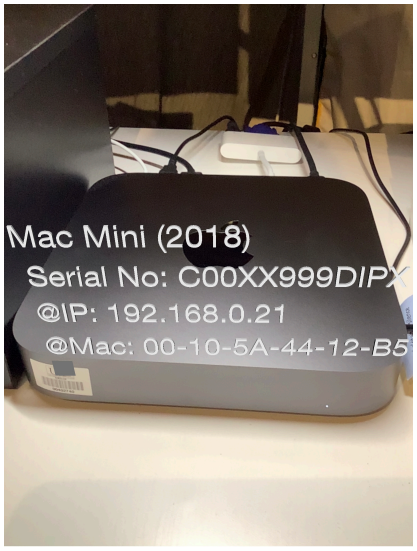
the cloud for defining the annotation model of the Active DSLs such as our User Role models, a possible alternative would be to use the common LCDP and integrate the user role model editor.

2.5 Augmented Reality DSL

According to our study [2] in sub-section 2.3, AR concrete syntaxes are barely used in the modelling commynity. Thus, we propose DSLs enhanced with augmented reality for their use within mobile devices.

In our vision, the user will be able to create domain-specific models using the mobile device, but instead of placing the objects on a blank canvas, the camera is used to place the objects (virtually) within the surroundings, using AR. This leads to a new range of modelling applications, like modelling environments for interior design, where furniture (elements of an AR-enabled DSL) can be placed and overlaid over the room the user is in (cf. Figure 14a); applications for computer inventory, where relevant computer information is displayed on top of the real computer (cf. Figure 14b); or leisure applications where tourists can display tips on points of interest by focusing the camera on them, and get directions towards nearby

(a) Interior design: Virtual table on a real room.



(b) Inventory: Displaying computer information.

(c) Leisure: Routes and tips about points of interest.

Figure 14: Examples of AR modelling applications.

touristic spots via arrows on the real-world (cf. Figure 14c).

2.5.1 Augmented Reality DSL

Habitually, the user interacts with a model of the system under study using a DSL and the studied system is disconnected from the model, in the sense that the model objects are not interwined with the real objects. Moreover, the physical location of the modeller is irrelevant. With our proposal, the interest in mobility takes on its full meaning since we take the model closer to the system under study by extending the DSL notation with AR. Figure 15 illustrates this idea. In this setting, the physical reality is augmented with model-based data, and the modeller location is used to display the (virtual) model objects positioned nearby.

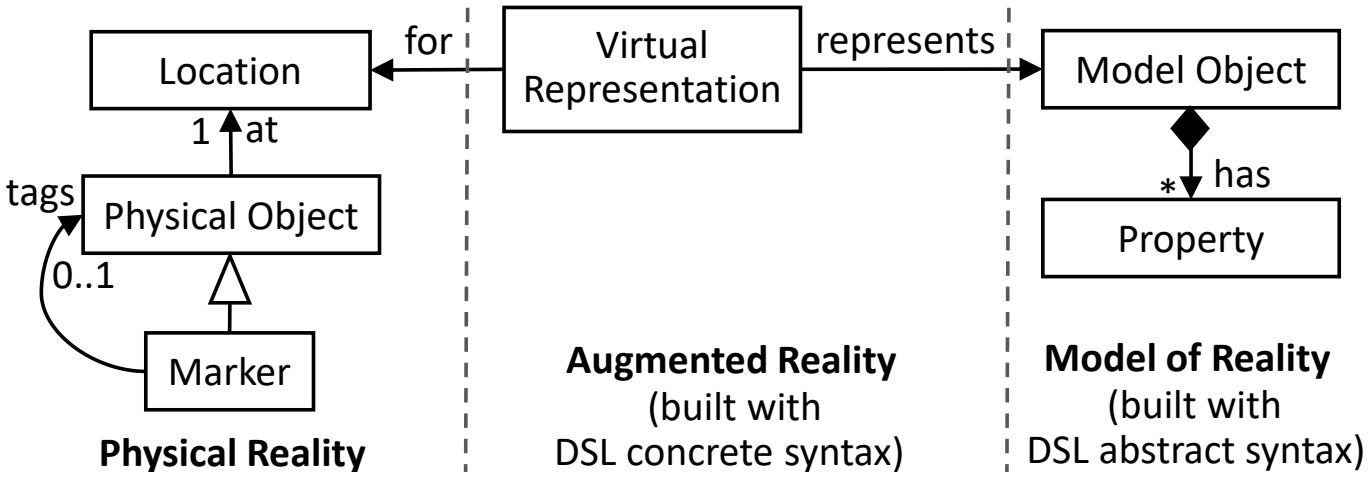


Figure 15: Augmenting the reality using an AR-based DSL.

The elaboration of an AR-based DSL involves the definition of an AR-based concrete syntax, hence, we have designed the meta-model of Figure 16. It permits defining AR-based syntaxes by annotating the domain meta-models that describe the abstract syntax.

Our AR-based syntax distinguishes *nodes* (class `ARNode` in Figure 16) and *connections* (class `ARConnection`). Class `ARNode` has references to the domain meta-model class it provides a representation for (`EClass`), and to the set of class properties to be displayed (reference `displays`). It may define several *visualization* versions (e.g., to display several types of chairs in an interior design application) using either `3DObjects` or `2DObjects`. Visualizations have a name and are referenced via a URL. They can be built with 3D authoring kits like Blender, or selected from libraries.

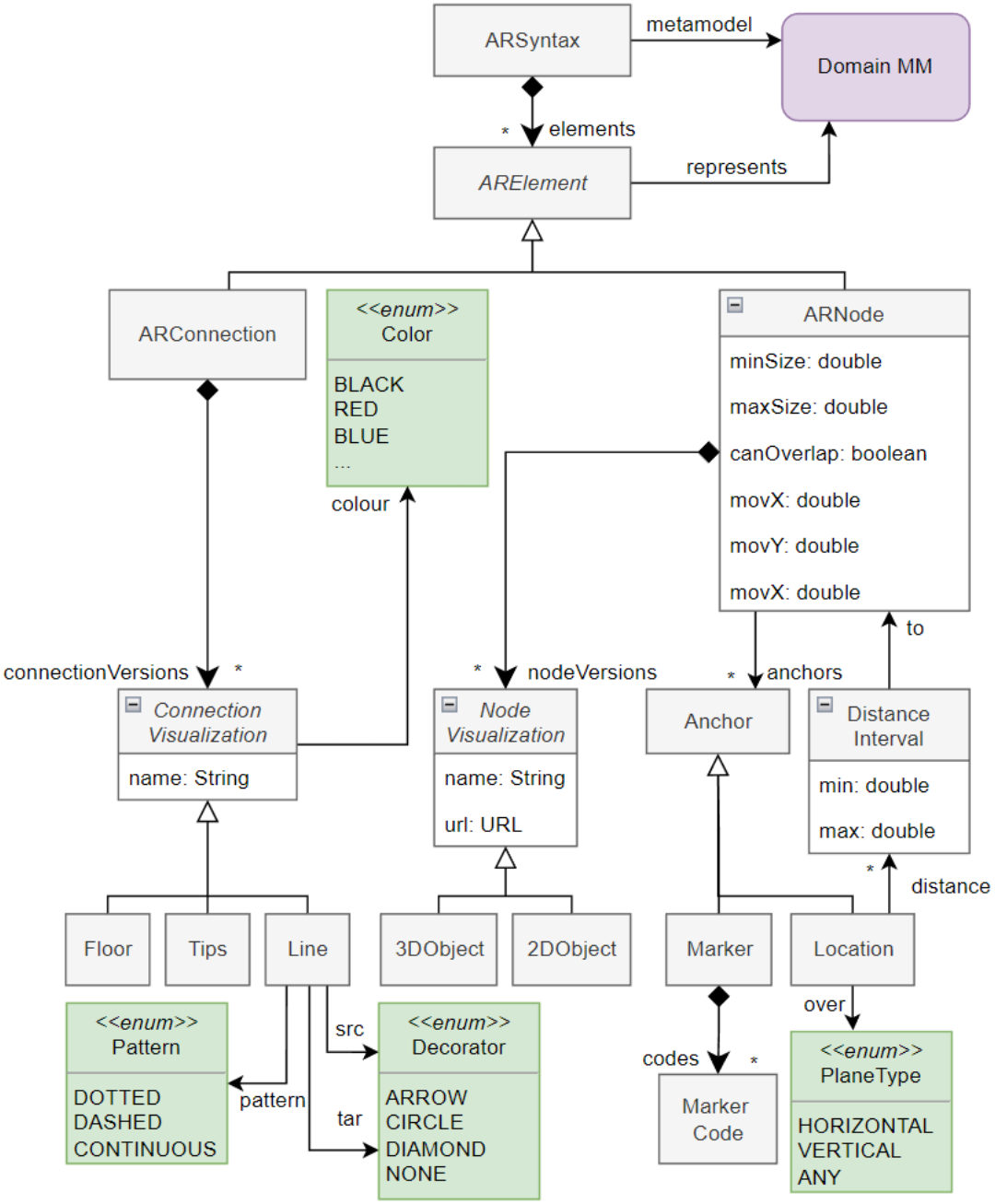


Figure 16: Meta-model for AR-based concrete syntaxes.

Nodes can be placed at suitable *anchors* (class `Anchor`). Anchors can be markers (e.g., QR codes) or constraints stating admissible locations, such as in a given plane (horizontal, vertical) or at a certain distance interval to another node. For example, a DSL for interior design may require that lamps are located on horizontal planes, within 1 meter to a socket.

Nodes can also define constraints on how users interact with them. Specifically, setting attribute `canOverlap` to false prevents that two objects are placed one over the other; attributes `minSize` and `maxSize` control the increase or decrease in the size of a placed node; and `moveX`, `moveY` and `moveZ` establish the perimeter where a virtual object can be moved once positioned in the world (if zero, the object cannot be moved).

Regarding AR connections, they are visual representations of references between instances of classes of the domain meta-model (class `EReference`). Similar to nodes, they can have different versions identified by a name and a colour. The meta-model supports three types of connections: `Line`, `Floor` and `Tip`. Lines are displayed as floating lines between two related objects, with decorations in the source and target ends. Floors display a reference as a path to follow in the floor from the source to the target object (cf. Figure 14c). Tips display arrows on the side of the camera steering the position of the target node, mimicking some first-person shooter games.

2.5.2 Tool Support

We propose the architecture in Figure 17 to define and use AR-based DSLs on mobile devices. It is supported by a prototype tool named AlteR (domAin modeLling using augmenTEd Reality). This is a native iOS app that uses ARKit [12] for creating and handling AR models. More details and videos are available at https://alter-ar.github.io.

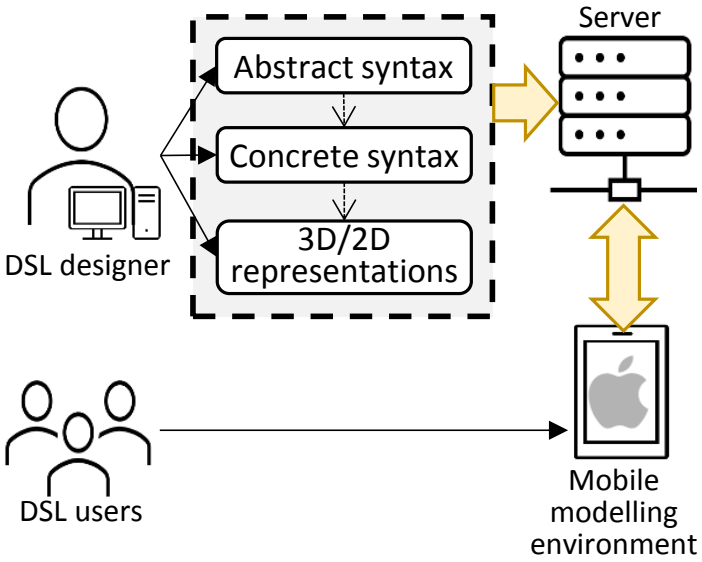


Figure 17: Architecture of AlteR.

First, to define the AR-based DSL, the DSL designer uses a desktop computer to specify the DSL abstract syntax (domain meta-model) and concrete syntax (an instance of the meta-model in Figure 16 and 3D/2D object representations), and upload them to a web server. We currently serialize the abstract and concrete syntax in JSON, and the 3D/2D graphics need to follow Apple's SceneKit Scene (SCN) format.

Then, the DSL users access the DSLs available in the server using AlteR on their iPhones/iPads. Upon selecting a DSL, users can start modelling by placing virtual objects (primitives of the DSL) in the real world. Every time a modelling session is started, AlteR/ARKit sets the world origin at the position of the camera of the device. When a virtual object is created, an anchor positioned relative to the established world origin is attached to the object. Models can be saved, which entails taking a snapshot and storing the world origin and the anchors of the model objects. When opening a previous model, the saved anchors are placed at their previous position as soon as the user points the camera to the same location of the snapshot.

Figure 18 shows a home networking AR-based DSL being used in AlteR. Label 1 is a palette with the items (virtual objects) that can be placed in the real world, named after the classes of the domain meta-model. Label 2 points to a 3D node and depicts the available interactions: *tap* places an item at the tapped location after selecting one from the palette with label 1; *swipe* changes between the visualization versions of a node; *pinch* resizes a node; *long press* a node displays the attributes of the virtual object for their editing, and allows deleting the node; and *pan* moves a node to another position in the world.

Label 3 shows the displayed attributes of a node as a key-value list. Label 4 points at a line connecting two nodes, which represents a reference between two model objects. Finally, there are two buttons at the top to save the model (label 5) and close the modelling session (label 6). Overall, this DSL enables users quick access to the whole network configuration, without checking in every device.

2.6 Context DSL

Context-awareness refers to systems that can both sense and react based on their environment. A model could adapt its elements according to the remaining battery of the device or the meteo at its current location. Thus, we need to mangage and collect the potential sources of context, define conditions that will trigger possible actions on target elements of the domain model. The association of all these elements resumes in contextual rules.

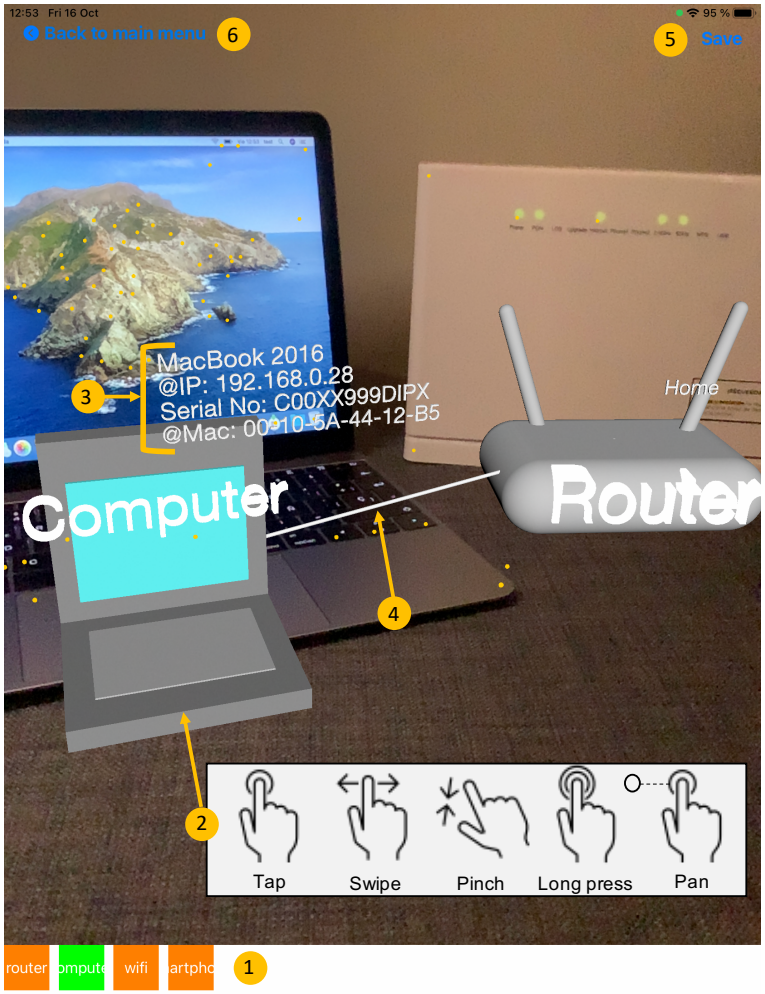


Figure 18: The main canvas of AlteR (running on an iPad).

2.6.1 Context DSL

The context meta-model permits defining contextual rules to render the domain meta-model context-aware. This permits an Active DSL to react to external events triggered by external interactions (like a web API and IoT devices). Similarly, they can react to the current state of built-in functionalities of the mobile device (such as sensors or internal clock). Figure 19 illustrates the context meta-model ang gives an example of its textual concrete syntax.

The *context description* is applied on a *device*. This latter one is the device on which the domain meta-model will be executed. The device has many *functionalities* which can either be **services** (e.g. **web** servies or *IoT* devices) or *features* of the device itlsef (in the case of a smartphone, it can be the clock, the gps, the battery, ...).

The *context description* describes a set of *contextual rules* defined with an *action* triggered when a *condition* is met. The former one can be actions targeting elements from the abstract syntax (e.g. create, delete), the concrete syntax (e.g. show, hide) or the modelling editor. The latter can use data from the device or the domain elements.

For analysis purposes, we propose to use the context DSL at runtime but also design time. Hence, *scenari* can be described in order to simulate some event to verify that the contextual rules are triggered accordingly. A scenario has a set of ordered *steps* which are composed of a set of *events*.

Finally, the *context description* tracks a set of *logs* in its *history*. This allows to analyse the trace and will eventually open the possibility to take in account the state induced by previous event into the condition of contextual rules.

2.6.2 Tool Support

The context DSL will be used in two situations. At design time, on static environment via an Eclipse plugin and at runtime, similarly to the Role DSL of sub-section 2.4. We are using Xtext for the textual syntax,

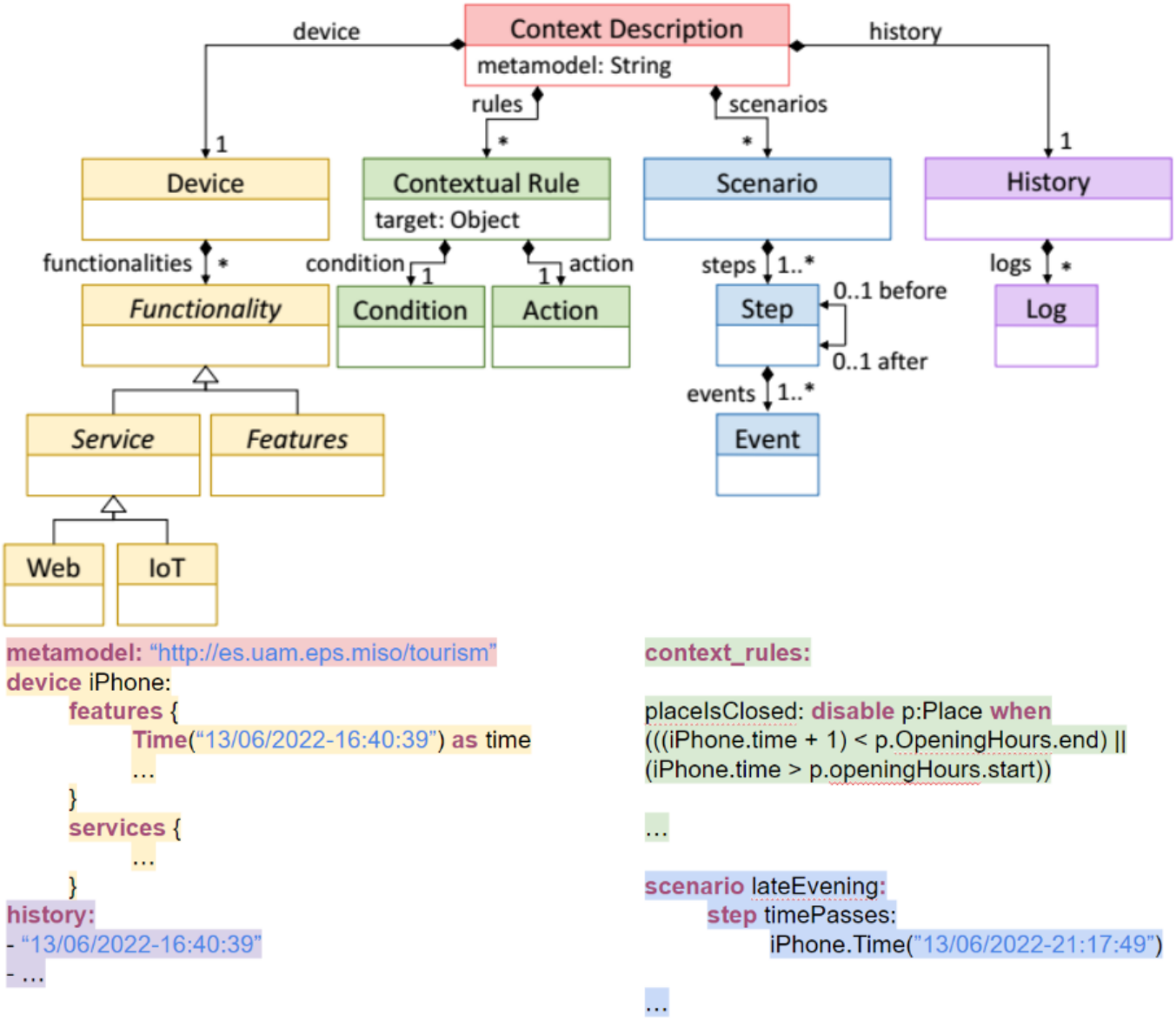


Figure 19: Meta-model for defining contextual rules and an example of its textual concrete syntax.

hence, it is possible to use the context MM directly on Eclipse but it is planned to offer a web version of the textual editor to cope with the idea of mobility.

2.6.2.1 Design phase At design time, the model created using the textual syntax will be given to an Eclipse plugin that will generate Henshin rules for analysis. Henshin provides many kind of analysis: conflicts, dependencies, simulation and state-space. These functionalies will highlight the weaknesses and inconsistencies of its contextual rules before pushing them on the final tools at runtime.

Currently, the plugin offer a wizard to provide templates of models according to the type of device it targets. The implementation of the transformation model to model (M2M) from the context model to the Henshin rules is currently in progress.

2.6.2.2 Runtime phase To accomplish the third goal of task 3.5, we envision to implement a web editor for the concrete syntax along with a backend server that manages the context description similarly to the role DSL. For the app-based editor, we plan to integrate the management of the contextual rules within DSL-Comet or AlteR.

2.7 Conclusion and Future Work

This section has summarized the progress of the research for task 3.5. The systematic mapping study conducted for the first goal of task 3.5 has confirmed the motivation argued for mobile modelling, the characteristic features of mobile modelling approaches and the research opportunities in mobile modelling.

For the following goals of task 3.5, we need to finish the implementation of the role DSL and context DSL, regarding the web editors but also their integration into our current app-based editor.

3 Monitoring the Performance of Machine Learning Models

3.1 Introduction

In recent years, a steadily increasing number of businesses have started utilizing machine learning (ML) techniques in hopes of gaining a competitive advantage in the market. However, as pointed out by researchers working within companies that were early adopters of ML techniques [13], the predictive model is a relatively small part of the overall system needed to effectively apply ML techniques in an operational setting.

As part of work package 3, a low-code solution has been developed that leverages model-driven engineering (MDE) principles [14] to facilitate performance monitoring of ML models. More specifically, the objective is to lower the technical barrier that data scientists face when developing systems for the detection and management of dataset shift. As it will be explained further in sub-section 3.2 with practical examples presented in sub-section 3.4, the term dataset shift encompasses a number of scenarios, where the data observed once an ML model has been deployed, follows a different statistical distribution from the data that was used for its training, with potentially negative consequences for its predictive accuracy. The proposed solution is comprised of a domain-specific language (DSL), which can be used by data scientists for producing high-level specifications of dataset shift management strategies, as well as a component that implements the desired runtime behaviour. This component makes minimal assumptions about the environment in which it is deployed and can interface with other, platform-specific components, developed by an organization's software engineers.

In summary, throughout this section, the following contributions are presented.

1. The Panoptes Domain Language (PDL) which can be used to specify dataset shift detection and management strategies in a technology and platform-agnostic manner.

2. A set of constraints based on the semantics of the modelled domain that aid data scientists in the production of error-free PDL models.

3. A runtime component that implements the behaviour specified in PDL models and which can be integrated into an ML platform.

4. An empirical study that supports the claim that the abstract syntax of the developed DSL is sufficiently expressive for the target domain.

The rest of this section is structured as follows: sub-section 3.2 provides the theoretical background of dataset shift and the motivation for following a model-based approach in operational settings. Sub-section 3.3 presents a high-level overview of the proposed solution. Sub-section 3.4 introduces the domain of ML model monitoring and presents the domain metamodel with the help of a running example. Sub-section 3.5 goes into the validation of PDL models based on the semantics of the domain. Sub-section 3.6 explains how PDL models are used to inform the runtime behaviour of the solution. Sub-section 3.7 presents a comparison of commercial products targeting the ML model performance monitoring domain. Finally, sub-section 3.8 presents the key takeaways of this work.

3.2 Background & Motivation

For the purposes of contextualizing the work presented, a brief overview of the statistical learning concepts related to the problem of dataset shift will be presented. The development of AI-enriched applications is also discussed, as a socio-technical process involving people of diverse skillets and how that affected some of the design decisions of the proposed solution.

Within the supervised learning subset of ML [15, 16], the objective is to extract a mapping from a set of labelled samples (known as the training set), that can be used to predict the value of a target variable Y, given the value of an observed variable X. The variables X and Y follow a joint probability distribution P(X, Y) which is unknown but usually assumed to remain unchanged between the time we extract the mapping and the time that we use it to predict Y [17].

When the assumption of a static joint probability distribution does not hold, the predictive accuracy of ML models can decline. Such scenarios have been studied for a number of years using numerous terms, including concept drift/shift [18, 19], covariate/sampling shift [20, 17, 19], prior probability shift [17, 19]

and others. In recent years, the more general term "dataset shift" has been introduced in [21] and further standardized in [22] and [23]. In this section we are going to adopt the term "dataset shift" and its subtypes as defined in [22].

While there is a plethora of proposed methods found in the literature for detecting dataset shift and counteracting the negative effects it can have on the performance of ML models [24, 19, 25, 17], researchers usually focus on the theoretical/algorithmic aspects of the problem. On the other hand, in practical settings, one might face additional challenges when seeking to apply dataset shift counteracting techniques for ML models deployed in production. In this scenario, they would not only need to apply the right algorithms for detecting and possibly counteracting dataset shift, but they would also need to architect a system that keeps track of the data that the various deployed ML models receive over time and apply the right techniques at the right time while also adhering to the various operational constraints of our production environment, such as minimizing the computational resources used or adhering to a maximum allowable latency for prediction responses.

The wide range of skills required to build operational ML systems means that they are usually built by multi-disciplinary teams comprised of members that specialize in different domains. Such teams can, for example, include data scientists that specialize in the statistical properties of the ML models that they develop and software engineers that specialize in the operational aspects of using the developed ML models to enable specific functionalities within a commercial product. This approach however, carries its own set of problems due to the overhead that occurs when people from different disciplines communicate.

Trying to address this problem, our proposed solution provides a DSL that can be used by data scientists to produce a high-level description of the desired behaviour of a system, combined with a mechanism for software engineers to expose the functionality of the underlying ML platform in a generic manner as opposed to possible ad-hoc solutions that might otherwise be adopted.

3.3 Solution Overview

In this sub-section, a high-level overview is presented, explaining how the behaviour specified using PDL is implemented at runtime in a manner compatible with potentially pre-existing software components.

The typical ML workflow is comprised of multiple stages. In [26], the authors list nine such stages which include model requirements, data collection, data cleaning, data labelling, feature engineering, model training, model evaluation, model deployment and model monitoring. As noted in relevant literature [27, 28, 29, 30] and confirmed by our own observations, data scientists use a combination of components to support these stages. These components can include, but are not limited to, notebook servers used to conduct exploratory data analysis and training of ML models, ML model registries used to store trained ML models alongside training metadata, data warehouses for the storage of an organization's data and ML model servers that respond to prediction requests. We use the term *ML platform* to refer to the set of components, such as the above, used to streamline the ML workflow.

The goal of Panoptes is not to provide additional ML platform components that directly support one or more stages of the ML workflow, but to provide a way for data scientists to leverage existing components for the detection and management of dataset shift. The novelty of the proposed solution is the decoupling of the specification of dataset shift related measures from any specific technology or platform. This is achieved by offering a DSL to data scientists, as described in sub-section 3.4, and also offering a mechanism for software engineers to expose the components that they develop in the context of an ML platform for data scientists to use.

This mechanism is enabled by the Panoptes orchestrator component. This component, as seen in Figure 20, is deployed alongside all the other components of an ML platform and it communicates with them. The orchestrator does not make any assumptions about the ML components it interfaces with except that they are exposed as web services that adhere to a pre-agreed API. As it will be explained in more detail in sub-section 3.6, the communication between the orchestrator and the various ML components allows it to keep track of the current state of the deployed ML models and to trigger certain processes when needed.

3.4 Domain analysis

This sub-section introduces a contrived example, in which dataset shift could potentially have a negative effect on the performance of a deployed ML model. Subsequently, an explanation will be presented of how PDL can be used by data scientists, to define strategies for detecting and counteracting dataset shift

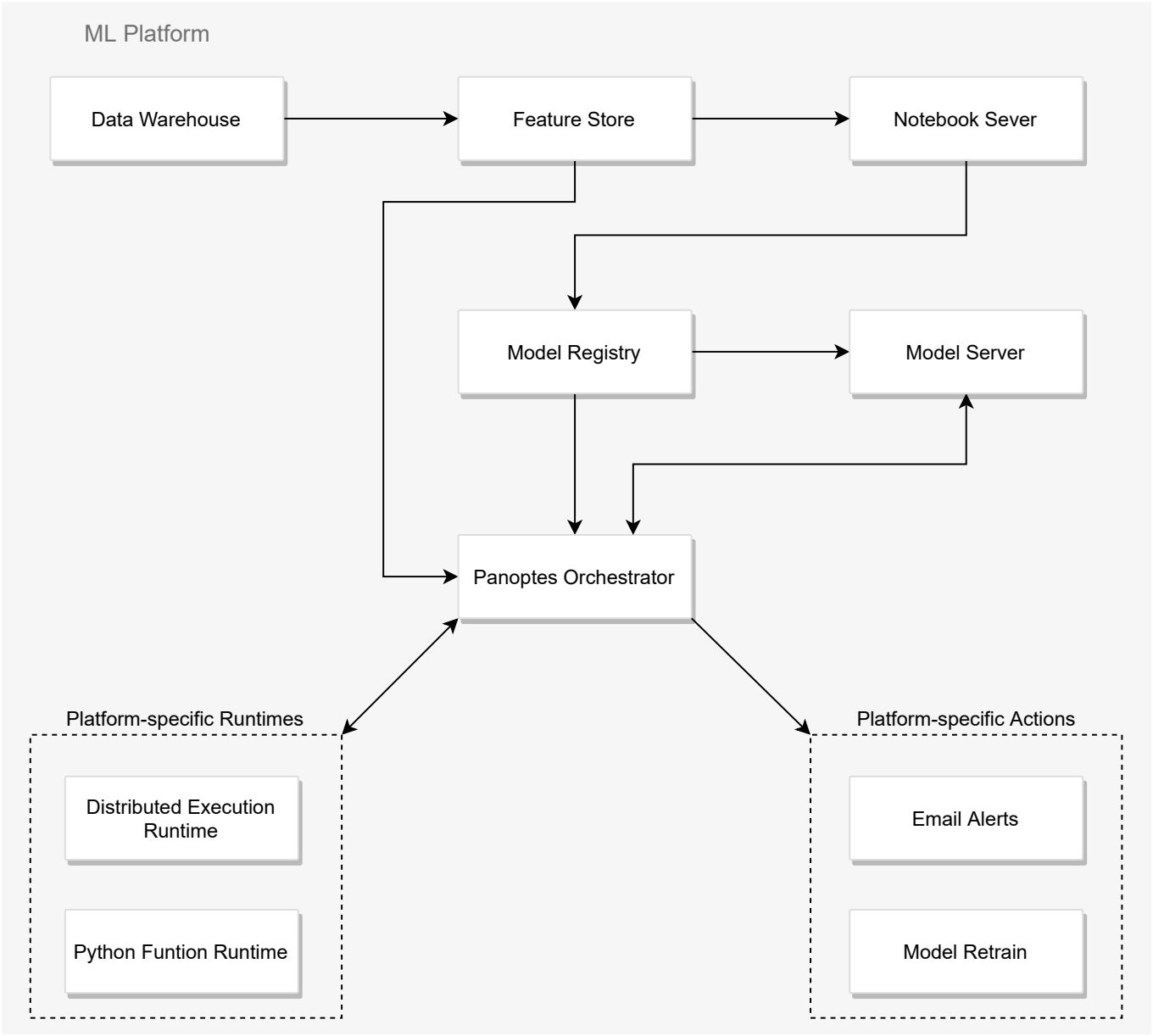


Figure 20: Example ML Platform

without having to take into consideration any technical details of the different technologies used in the ML platform.

In the example scenario, there is a data science team working for an internet service provider. The marketing department of the company has tasked the team with predicting which customers are likely to discontinue their subscription in the next 30 days (customer churn). This information could be used, for example, to offer promotional discounts to customers classified as churning in order to entice them to keep their subscriptions. To keep the example simple, it is assumed that the only parameter that influences customer churn is the quality of their connection. This is quantified as the number of faults that a connection has exhibited in the last 12 months. Further, it is assumed that despite the stochastic nature of the scenario due to differing customer sensitivities to technical faults, there is a causal relationship between the two variables, with fewer customers churning given a relatively low number of faults. After gathering the data and applying an appropriate ML algorithm, such as k-nearest neighbour [16], it is assumed that the data science team has produced an ML model that can classify customers as churning and non-churning, with high accuracy, based on the number of faults that their connections are exhibiting. The output of this ML model can subsequently be used to populate a dashboard with customer churn predictions, so that marketers can accomplish their goals.

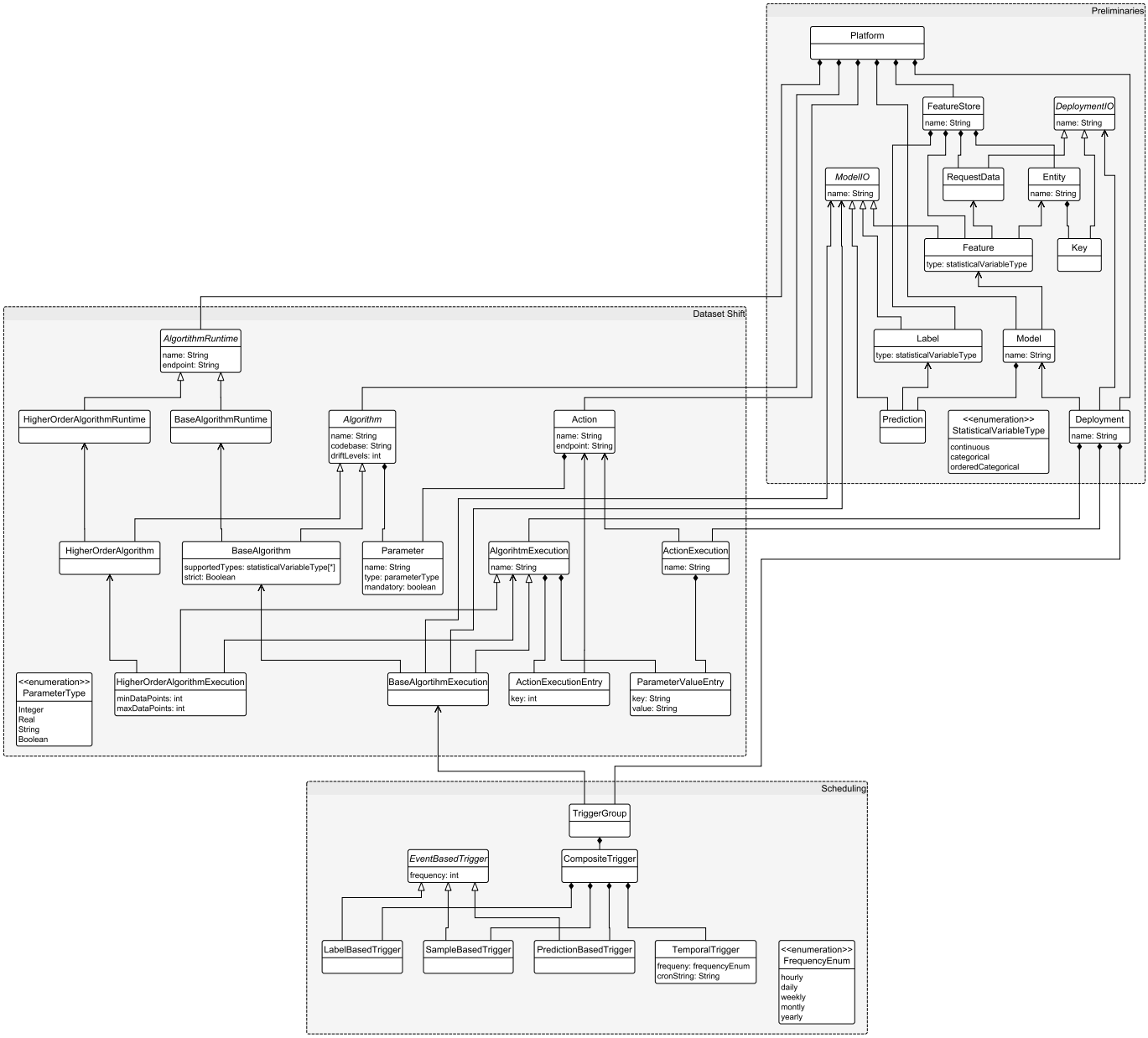


Figure 21: Domain Metamodel

3.4.1 Preliminary Concepts

Before discussing any concepts related to dataset shift, there are several preliminary concepts that need to be introduced first in the metamodel. These concepts form the necessary foundation for more advanced concepts related to dataset shift to be introduced. All classes discussed in this subsection can be seen in Figure 21, grouped under the label *Preliminaries*.

The first necessary concept is that of a *Platform*. A *Platform* instance is the top-level element in a PDL model. It represents the ML platform, as discussed in the previous sub-section, that data scientists can utilize throughout the various stages of their ML workflow, including the monitoring of deployed ML models. The components of the ML platform, developed by software engineers, that can be used in the context of dataset shift are represented in PDL models by *AlgorithmRuntime* and *Action* instances. It is envisioned that every time a software engineer adds a new dataset shift related component to an ML platform, they would also add the appropriate PDL model elements to the corresponding *Platform* instance. This way, there will eventually be a catalogue of model elements that data scientists can reference in their own PDL models allowing to utilize all of the capabilities of the modelled ML platform. The rest of the model elements directly contained in a *Platform* instance are managed by data scientists. These include model elements that are instances of the *Model*, *FeatureStore*, *Deployment* and *Algorithm* classes which are further explained below and in the following subsection.

A *Platform*, as described above, can be utilized to deploy one or more ML models in production. In the customer churn scenario for instance, the system responsible for the marketing dashboard has access to a

list of customers, identified by their customer IDs, and needs to obtain a churn prediction for each of them. It is up to the data scientists to explore the relevant data that the organization stores for each customer and train an accurate ML model. Over time, as more data becomes available regarding the company's customers and more advanced ML techniques are developed, the data scientists might chose to replace an older ML model with a newer one. This can be hidden from the marketing dashboard system as long as a customer churn prediction can still be provided for a given customer ID.

To accurately represent the above in our models, a distinction has been made between the concepts of *Deployment* and *Model*. *Model* instances represent ML models regardless of whether they are currently being used in production or not. An ML model receives one or more features as input and produces a prediction as output. This is reflected in the metamodel by the *Feature* and *Prediction* instances referenced by a *Model* instance. Additionally, a *Prediction*, references a *Label* that represents the ground truth values that the ML model is trying to predict. Since the *Feature*, *Prediction* and *Label* classes are all related to the input and output of ML models, they subclass the abstract class *ModelIO*. On the other hand, *Deployment* instances are used to represent a specific task that can leverage an ML model, such as predicting churn for a specific customer ID. Each *Deployment* references a *Model* instance and one or more *DeploymentIO* instances. The referenced *Model* instance represents the ML model that is currently used to accomplish the task. On the other hand, *DeploymentIO* instances represent the data that is presented to a *Deployment* as input.

Finally, to express the fact that each feature can be used by multiple ML models and those ML models might attempt to predict the values of the same variable, the concept of the feature store is introduced. Feature stores are represented in the metamodel by the *FeatureStore* class whose instances contain multiple *Feature* and *Label* instances so that they can be referenced by as many *Model* instances as needed.

With the metamodel classes introduced so far, the scenario at hand can be modelled using PDL. In Listing 1, an example model can be seen in a textual notation that was created using Xtext [31]. This is a preliminary textual notation used to illustrate the usage of the DSL. The development of additional, potentially more suitable concrete syntaxes remains as future work. The model elements include one *Model* named *churnKNN* which references one *Feature*. Additionally, there is a *Deployment* named *customerChurn* which references the *churnKNN Model* as well as a *DeploymentIO* instance named *customerID*. More precisely, *customerID* is an instance of class *Key* which subclasses *DeploymentIO* and all *Key* instances are contained by an *Entity* instance but this will be discussed in sub-section 3.5.

```
1 Model churnKNN{
2   uses faultNumber
3   outputs churnKNNPred predicts customerChurn}
4
5 FeatureStore{
6   features faultNumber
7   labels customerChurn}
8
9 Deployment customerChurnDeployment{
10  model churnKNN
11  inputs faultNumber}
```

Listing 1: PDL example 1

3.4.2 Dataset shift related concepts

Once the ML model is deployed and used in production, as described above, it is assumed that the data scientists responsible for this project would like to ensure that their ML model maintains an acceptable predictive performance over time. In order to do that, they would like to constantly monitor against dataset shift and apply corrective measures when needed.

3.4.2.1 Terminology As per the definitions found in [22], based on the causal model of the customer churn scenario (i.e a high number of faults causing customers to churn and not the other way around), the most common types of dataset shift that can be observed are **covariate shift** and **concept shift**.

For **covariate shift** to be observed, there would need to be a shift in the statistical distribution of connection faults for the company's customers without affecting the conditional probability of a customer churning given the number of faults of their connection. This could happen if, for example, due to lack of maintenance, customers' connection start exhibiting a higher number of faults on average but customers' attitudes towards connection faults hasn't changed. As expected, this would result in a higher number

of customers churning since more of them are having a bad user experience but that does not mean that the threshold for bad user experience, in terms of connection faults, has changed. From a data scientist's point of view, since the conditional probability is unchanged the ML model's performance has remained consistent with what was achieved during training. Despite that, they might still want to monitor for covariate shift because, for example, the shifted statistical distribution of the ML model's inputs might enable them to use a simpler ML algorithm, in terms of computational complexity [17], such as a linear classifier.

Alternatively, for **concept shift** to be observed, there would need to be a change in the conditional probability of a customer churning given the level of connection faults for their connection while the statistical distribution of connection faults for the company's customers as a whole has remained the same. This could happen if, for example, a large number of the company's customers start working from home and are thus more critical of connection faults. This would mean that despite customer's connection exhibiting the same number of faults, a higher percentage of customers are churning. From a data scientist's point of view, this is a difficult scenario to detect and counteract. Firstly, only observing recent feature values is not sufficient to detect concept shift. One would need a source of ground truth values/labels. Secondly, depending on the magnitude of concept shift, the samples in the training set might no longer be usable and thus a new round of data collection might be necessary.

3.4.2.2 Metamodel In order to model the above scenarios, new classes need to be introduced in our metamodel. These can be seen in Figure 21 grouped under the label *Dataset Shift*. The first relevant class is *Algorithm*. *Algorithm* instances represent algorithms that can detect dataset shift. There are two types of algorithms that can be represented in a PDL model by using a subclass of *Algorithm*. These subclasses are *BaseAlgorithm* and *HigherOrderAlgorithm*. Instances of *BaseAlgorithm* represent algorithms that can detect dataset shift from the *ModelIO* values (i.e features, predictions and labels) in the training and in-production datasets. On the other hand, *HigherOrderAlgorithm* instances represent algorithms that take as input the output of other algorithms over a period of time and attempt to detect gradual degradation of the performance of the monitored ML model in a more robust way compared to an algorithm that bases its decision on a fixed point in time. Algorithms are implemented by data scientists, using one of the technologies supported by a component of the ML platform, which in PDL are represented by *AlgorithmRuntime* instances. The technology-specific implementation artifacts for an algorithm are stored in a git repository whose URL is the *codebase* attribute value of the corresponding *Algorithm* instance. For instance, Listing 2 shows the implementation of the two-sample Kolmogorov-Smirnov statistical test [32] that will be used in the examples below, implemented as a Python function [33].

```
1 from scipy import stats
2
3 def ksTest(trainSet, liveSet, parameters):
4     pValue = stats.ks_2samp(trainSet, liveSet))[1]
5     if pValue < parameters['pValue']:
6         return 1
7     else:
8         return 0
```

Listing 2: Example algorithm in Python

While *Algorithm* instances represent general algorithms that can detect dataset shift regardless of the context in which they are used, *AlgorithExecution* instances represent the application of these algorithms in a specific scenario. For example, the two-sample Kolmogorov-Smirnov statistical test, shown above, can be used to determine whether two sets of samples are drawn from the same statistical distribution regardless of what the statistical variables in the samples represent (e.g weight, height, etc). This algorithm would be represented by an *Algorithm* instance in a PDL model. On the other hand, applying the above statistical test to determine whether or not the values of a particular feature (e.g height) in a training set follow the same statistical distribution as those observed in production would be represented by an *AlgorithmExecution* instance in a PDL model. *AlgorithmExecution* instances specify the data that should be passed as input to the algorithm for each execution and which actions should be executed depending on the result. *AlgorithmExecution* is subclassed by *BaseAlgorithmExecution* and *HigherOrderAlgortithmExecution* to represent the execution of *BaseAlgorithms* and *HigherOrderAlgorithms* respectively, since these two kinds of executions require different information to be specified, as it will be shown in the example below.

Furthermore, in order to minimize the amount of technology-specific functionality implemented by

data-scientists, the concept of algorithm runtimes is introduced, represented by *AlgorithmRuntime* instances. Algorithm runtimes, as already mentioned, are components of an ML platform that provide all of the functionality needed for retrieving the relevant datasets, executing the algorithm specified by an *AlgorithmExecution* and sending the result of the execution to the Panoptes Orchestrator component, which will be further discussed in sub-section 3.6. The one requirement data scientists need to satisfy, is the inclusion in every algorithm repository, of all the artifacts required by the algorithm runtime they have chosen for the execution of their algorithm. This is intentionally left up to algorithm runtime creators to document, outside the boundaries of a PDL model, to ensure that there are no restrictions on the kind of technologies algorithm runtimes can potentially support. Besides code, the required artifacts of an algorithm runtime could include, for example, pip requirement files [34] in case of a Python-based implementation or Maven POM files [35] in case of a Java-based implementation. In conclusion, for every algorithm, the chosen algorithm runtime component would be responsible for retrieving the relevant artifacts from the implementation repository, transform them into an executable form, depending on the technology used and follow all of the execution steps every time it receives a notification by the Panoptes Orchestrator to trigger an algorithm execution.

In [36], an example of such a mechanism is presented. The implemented software component receives as input a file containing a Python function, such as the one shown in Listing 2, along with a Python requirements file. It subsequently builds a container image containing the Python function in question, all of the Python packages listed as dependencies and a runtime that retrieves the relevant *ModelIO* datasets, pass it to the function and then finally, depending on the result, send out an email notification indicating dataset shift. With the introduction of the *AlgorithmRuntime* concept, the same approach can be extended to other technologies, as required.

Lastly, concepts related to the potential actions a data scientist would like to execute in the presence of dataset shift are introduced. An example of such an action could be the sending of an email notification to the data scientist that trained the affected ML model. The ability to execute actions is once again provided by specific software components included in the underlying ML platform. These software components are exposed to data scientist though the inclusion of an *Action* instance that represents them.

Based on the same principle that links *Algorithms* with *AlgorithmExecutions*, while *Action* instances represents a capability of the underlying platform, the usage of such a capability is represented by an *ActionExecution* instance that parametrizes the *Action* to fit in the context of a specific scenario.

3.4.2.3 Examples With the above classes of the PDL metamodel, covariate and concept shift scenarios can now be modelled in the running example of customer churn.

For covariate shift, the data scientists would like to compare the statistical distribution of the number of faults feature in the training set compared to the statistical distribution of the values recently observed in production. This is expressed in Listing 3 in lines 38-43 by the *BaseAlgorithmExecution* instance named *KSFaultNumber*. As shown in the Listing, *KSFaultNumber* references the *BaseAlgorithm* named *kolmogorovSmirnov*. The *severityLevels* attribute of *kolmogorovSmirnov* specifies that the set of possible algorithm execution results contains two values. This attribute can be set to any natural number greater or equal to two depending on how many discrete levels of dataset shift the creator of the algorithm would like to express. In this case, the *severityLevels* attribute is set to two as the implemented algorithm is a statistical test. The *kolmogorovSmirnov* model element also contains a *Parameter* instance denoting that it can be parametrized by providing a value for the *pValue* parameter. This means that a separate implementation of the algortithm is not needed for executing it with a different p-value. Lastly, the *kolmogorovSmirnov* element references the *pythonFunction BaseAlgorithmRuntime* to denote that the algorithm is implemented as a Python function, as seen in Listing 2.

The above PDL model elements define how covariate shift is to be detected, but don't specify any actions to be taken in case it is detected. This is defined by the *emailAction* and *emailMe* model elements. The *emailAction* element is an instance of *Action* and, as the name suggests, represents a notification in the form of an email that can be send out when dataset shift is detected. The *ActionExecution* instance *emailMe* is a specific execution of *emailAction* which specifies the value "user@example.com" for the *emailAddress* parameter of *emailAction*. The model specifies in line x that *emailMe* will be executed in case the algorithm execution defined by *KSFaultNumber* returns the value 1 as the result.

```
1 BaseAlgorithmRuntime PythonFunction{
2   endpoint  "http://platform.local/runtimes/pythonfunction"}
3
```

```
HigherOrderAlgorithmRuntime PythonFunctionHO{
  endpoint "http://platform.local/runtimes/pythonfunctionho"}

BaseAlgorithm kolmogorovSmirnov{
  codebase "http://repo.com/kolmogorovsmirnov"
  runtime PythonFunction
  severity levels 2
  parameters pValue}


BaseAlgorithm accuracyCheck{
  codebase "http://repo.com/accuracycheck"
  runtime PythonFunction
  severity levels 2}

HigherOrderAlgorihtm timeSeriesAnalysis{
  codebase "http://repo.com/timeseriesanalysis"
  runtime PythonFunctionHO
  severity levels 2}

Action emailSender{
  endpoint "platform.local/actions/emailsender"
  parameters email}

Deployment customerChurnDeployment{
  model churnKNN
  inputs customerID

  BaseAlgorithmExecution KSFaultNumber{
    algorithm kolmogorovSmirnov
    from live data use faultNumber
    from historic data use faultNumber
    actions 1->emailMe}

    BaseAlgorithmExecution accuracyCheckChurn{
    algorithm accuracyCheck
    from live data use churnKNNPred, customerChurn}

  HigherOrderAlgorithmExecution accuracyCheckChurnTS{
    algorithm timeSeriesAnalysis
    observed execution chiSquaredChurn
    actions 1->emailMe
    min observations 10
    max observations 100}

  ActionExecution emailMe{
    action emailSender
    parameter values email="user@example.com"}

  when
  (500 samples received
  and
  one day passed)
   or
  (250 labels received
  and
  one week passed)
  Execute KSFaultNumber

  when
  (500 labels
  and
  one day passed)
  or
  (250 labels
  and
  one week passed)
  Execute accuracyCheckChurn
```

```
72 }
```

Listing 3: PDL example 2

To model the concept shift scenario, a similar approach can be used. As we can see in Figure 3 the only notable difference compared to the covariate shift scenario is the choice of algorithm. As specified by the *accuracyCheckChurn* model element defined in lines 45-48, the data given to the algorithm is going to be the predictions made by the ML model along with their ground truth labels for recently observed data. Each execution of the *accuracyCheck* algorithm will determine the accuracy of the deployed ML model, usually defined as the sum of true positive and true negative predictions over the total number of predictions made.

Additionally, to make the detection more robust and avoid false alarms, it is assumed that the PDL user has chosen to conduct a time-series analysis on the results of *accuracyCheck*. As an example, in Listing 3, lines 50-56, this has been implemented as a *HigherOrderFunctionExecution* that executes the *timeSeriesAnalysis* algorithm with the outputs of the *accuracyCheck* algorithm execution used as inputs. The *accuracyCheckChurnTS HigherOrderAlgorithmExecution* also specifies, that it needs at least 10 datapoints to execute with a maximum of 100 datapoints. Lastly, similarly to *KSFaultNumber*, *accuracyCheckChurnTS* specifies that an email should be sent every time it returns 1 as a result. Alternatively, a time-series analysis could have also been implemented as a *BaseAlgorithm* that directly processes the recent predictions made by an ML model along with their ground-truth labels over a longer time period. The advantage of implementing it as a *HigherOrderAlgorithm* is that the outputs of *accuracyCheckChurn* can be reused for other *HigherOrderAlgorithmExecutions* or even to trigger *ActionExecutions* on their own.

3.4.3 Scheduling related concepts

An important aspect of the modelled domain is expressing the frequency of executing a dataset shift detection algorithm and the subsequent corrective actions if needed. For this purpose, classes that represent various kinds of triggers as well as classes that represent a combination of triggers are provided. More specifically, *TemporalTrigger*, *SampleBasedTrigger*, *PredictionBasedTrigger* and *LabelBasedTrigger* can express the frequency of one or more algorithm executions in terms of how long it has been since the latest execution, how many unlabelled samples have been received, how may predictions have been served and how may labels have been received for previously unlabelled samples respectively. For more complex scenarios, data scientists can create *CompositeTrigger* instances that contain up to one instance for each kind of individual trigger and can represent scenarios in which the requirements must be met for multiple individual triggers at the same time. Lastly, *CompositeTrigger* instances can themselves be grouped in a *TriggerGroup* instance. A *TriggerGroup* instance tracks which *BaseAlgorithmExecution* is to be executed when the requirements for at least one of the contained *TriggerGroups* are met.

An example of how the above classes can be used is shown in Listing 3 in lines 63-82. In the PDL textual notation used, the **and** keyword is used to group individual *Triggers* into *CompositeTriggers* and the **or** keyword is used to group *CompositeTriggers* into *TriggerGroups*. As can be seen, the PDL model of the example specifies that *KSFaultNumber* should be executed once a day, given 500 new samples, in this case *faultNumber* feature values, have been received since its last execution. Alternatively, if one week has passed with no execution, due to receiving less than 500 samples, the execution should be triggered even if as few as 250 new samples have been received since last execution. The scheduling of *accuracyCheckChurn* is very similar to that of *KSFaultNumber*, the only difference being that the triggering frequency is based on the number of ground-truth labels received since the last execution due to the algorithm relying on the availability of this data.

3.5 Validation

Although the metamodel of PDL already captures certain typing and cardinality constraints, there is still the possibility of creating a model that contains errors, according to the semantics of the domain. For this reason, a series of constraints based on the semantics of the domain are defined using EVL [37] and every PDL model is validated against these constraints. In this sub-section certain parts of the metamodel that haven't been presented so far will be explained. These are optional in their usage, in terms of describing dataset shift strategies, but further enable Panoptes to detect semantic errors by providing additional information about each specific scenario.

3.5.1 Feature Validation

The first check of semantic validity ensures that every deployment in a PDL model uses a suitable ML model. This is true when every feature that is used as input by the ML model is retrievable from the deployment's inputs. This can prove valuable when a deployment starts using a newer ML model that utilizes more and/or different features but at the same time one would like to ensure compatibility with the systems that consume the predictions served by the deployment.

To achieve the above, every feature specification in a PDL model should also specify how it can be retrieved. For instance, in the customer churn scenario, the number of faults a connection has exhibited is linked with the customer who is served by that connection. Furthermore, an individual customer can be uniquely identified by their customer ID. Thus, given a customer ID, one can retrieve the number of faults the customer's connection has exhibited and use it to make a customer churn prediction.

This is reflected in the PDL metamodel in the following way. Every *Feature* can reference zero or more *Entities*. An *Entity* represents a concept in the domain, within which a data scientist builds a predictive model, for example a customer in the customer churn scenario. *Entities* contain one or more *Keys*. For a PDL model to be valid, for each *Deployment*element, the *Keys* directly referenced as inputs must be a superset of all the indirectly referenced *Keys*. A *Deployment* can indirectly reference a *Key* through a chain of references starting with the referenced ML *Model* element which in turn references one or more *Features* and so on until zero or more *Keys* are referenced.

Alternatively, some features are calculated on-demand based on data that is only available at the time a prediction is requested. This, for example, could be a search query that a user submits to a search engine. This is represented in the PDL metamodel by the *RequestData* class. If a *Feature* references a *RequestData* instance, this has to also be referenced as input by any *Deployment* that uses an ML *Model* that uses said *Feature*.

Listing 4 shows a PDL model for the customer churn scenario where a new ML model, *churnKNNImproved*, has replaced the older one in the *customerChurn* deployment. This newer model utilizes the gender of the customer to produce more accurate churn predictions. According to the feature validation information added to the PDL model, this newer ML model is compatible with the *customerChurn Deployment* as the *gender* feature values required for inference can be retrieved for a given *customerID*.

```
1 Model churnKNNImproved{
2   uses faultNumber, gender
3   outputs churnKNNPred predicts customerChurn}
4
5 FeatureStore{
6   features faultNumber:ordered categorical{requires entities customer},
7   gender:categorical{requires entities customer}
8   entities customer{keys customerID}
9   labels customerChurn}
10
11 BaseAlgorithm kolmogorovSmirnov{
12   codebase "http://repo.com/kolmogorovsmirnov"
13   runtime PythonFunction
14   severity levels 2
15   accepts only continuous
16   parameters mandatory pValue:Real}
17
18 Deployment customerChurn{
19   model churnKNNImproved
20   inputs customerID
21
22   BaseAlgorithmExecution KSFaultNumber{
23     algorithm kolmogorovSmirnov
24     from live data use faultNumber
25     from historic data use faultNumber
26     actions 1->emailMe
27     parameter values pValue=0.05}
28 }
```

Listing 4: PDL with validation information

3.5.2 Type validation

An additional category of semantic validity checks relates to the classes in the metamodel that have some kind of type attribute. This includes the *Parameter* and *Feature* classes.

Instances of *Parameter* are contained by *Algorithm* and *Action* instances to denote that they are parametrizable at runtime. The model validation procedure here consists of checking that the *ParameterValueEntries* contained in *AlgorithmExecution* and *ActionExecution* instances contain valid parameter names as keys and valid values. For example, in Listing 4, the *kolmogorovSmirnov BaseAlgorithm* contains a *Parameter* named "pValue" of type "Real". Therefore it would be checked that every *AlgorithmExecution* that references this *BaseAlgorithm* contains a *ParameterValueEntry* with "pValue" as key and a value that is a valid real number. This is implemented as an EVL constraint that checks that the *value* attribute of each *ParameterValueEntry* can be parsed as an EOL [38] literal of the specified type, as the possible *ParameterType* values coincide with EOL's primitive types. Additionally, we check that every *Parameter* denoted as mandatory is given a value and no *ParameterValueEntries* are accidentally left with empty keys or values.

The *type* attribute of *Feature* and *Label* instances can optionally be given a value. This *type* attribute is enumerative and can take the values *continuous*, *categorical* and *orderedCategorical* that correspond to the different types of statistical variables found in the literature [15]. Additionally, *BaseAlgorithms* can optionally be given one or more values of the same enumerative type in their *supportedTypes* attribute. Based on this information, we can validate whether the *ModelIO* instances referenced by a *BaseAlgorithmExecution* are of suitable statistical type for the *BaseAlgorithm* used. Lastly, If the *strict* attribute of a *BaseAlgorithm* is set to false, only a warning will be generated in case of statistical variable type mismatch, instead of an error, which will not cause the rejection of the validated PDL model.

As an example, a subtle error was introduce in the PDL model of Listing 3, which can be detected with the added typing information of Listing 4. The error lies in the fact that while variants of the Kolmogorov-Smirnov statistical test can be used for statistical variables of ordered categorical type [39], the algorithm implementation shown in Listing 2 depends on the Kolmogorov-Smirnov implementation found in the SciPy Python package [40] which only supports continuous statistical variables. Therefore, the *KSFaultNumber BaseAlgorithmExecution* will not work as intended. This could be a difficult to detect error, especially considering the fact that the data scientist that implemented the algorithm could be a different person than the one using the algorithm to detect covariate shift in a specific context. To avoid this situation, the data scientist implementing the algorithm in question could have added validation information as shown in Listing 4.

3.6 Runtime Behaviour

In this sub-section the component of the Panoptes system responsible for converting PDL models into runtime behaviour is presented. The sub-section starts with a high-level discussion of the architecture used to detect and counteract dataset shift. It also provides a brief description of the technical implementation of the component.

3.6.1 Panoptes Orchestrator

As already touched upon in sub-section 3.3, the orchestrator component seen in Figure 20 is responsible for converting PDL models into runtime behaviour. While the PDL metamodel is designed so that data scientists can conveniently specify how deployed ML models ought to be monitored, it does not provide the orchestrator with a way to keep track of its internal state for the purposes of triggering algorithm and action executions. For this reason, after PDL models are received by the orchestrator, they are used to construct Finite State Machine (FSM) objects which directly map to the required runtime behaviour.

More specifically, for every *Deployment* contained in a processed PDL model, one or more FSMs, equal to the number of *TriggerGroups* that the *Deployment* contains, will be created. The FSMs are comprised of one state, called *STANDBY* and two kinds of transitions from *STANBY* back to itself that are triggered by events ingested by the FSM.

The first kind of transition executes a snippet of code that sends a request to the relevant *Algorithm-Runtime* web endpoint(s) with the information defined by the *AlgorithmExecution(s)* that is being triggered. This kind of transition is triggered once enough events related to the triggers contained in a *TriggerGroup* are ingested. These events, created by the relevant ML platform components and send to the orchestrator, inform the FSM that either: the ML model has served an inference request, a label has been received, or

sufficient time has passed since the previous execution. Each of these events, increments a relevant counter stored in the FSM object. If the latest event that is received fulfils the triggering conditions of the FSM's corresponding *TriggerGroup*, the FSM will reset the relevant counters and execute the transition.

The second kind of transition executes a snippet of code that will potentially send a request to a relevant *Action* endpoint, depending on whether the result of the *AlgorithmExecution* requires an *ActionExecution* to be triggered. This kind of transition is triggered by events that signify that an *AlgorithmExecution* has been completed with its result being embedded in the event message.

3.6.2 Panoptes Web Editor

In order to provide data scientists with a simple user experience for creating PDL models and sending them to the orchestrator, an Xtext-based web application was developed. This approach was chosen mainly to avoid cumbersome installation procedures. The user interface of this application consists of a text editor where users can create PDL models using the textual concrete syntax provided and a list of model elements already received by the orchestrator. When the user has finished creating or updating a PDL model, a request is send to the orchestrator for the new model to be processed in the way described in the previous subsection. If the submitted model contains an error, according to the validation checks described in subsection 3.5, the model will be rejected and a relevant message will be displayed to the user so they can correct the errors and resubmit.

3.7 Related Work

As ML models have started to play a significant role in the commercial success of an organization, there has been a recent increase in the number of commercial products focusing on monitoring the performance of ML models in production. This sub-section discusses how Panoptes is related to this class of products and highlights some of the gaps in their functionality that motivated the development of this solution.

Relevant solutions from the three largest cloud providers are Amazon's *Sagemaker*[41], Microsoft's *Azure ML*[42] and Google's *Vertex AI*[43]. These cloud providers are uniquely positioned to offer ML monitoring related products to a large customer base that already uses their infrastructure to store data and run ML workloads. Their products are therefore highly relevant when comparing practical applicability.

All three products offer a mechanism for covariate shift detection albeit with varying levels of customization support. *Vertex AI* exposes an API for creating/updating monitoring jobs that execute on a periodic basis. The algorithm used to detect covariate shift is fixed, namely *L-infinity distance* for categorical features and *Jensen-Shannon divergence* for continuous ones [44]. Every feature is considered in isolation so multivariate analysis is not possible. The user can set a threshold per feature for the output of the algorithm used, above which covariate shift will be detected. Upon detection the user can choose to receive an email notification or have a message passed to a message queue where other services can read from. This however would require additional software engineering effort as no integration is offered. *Vertex AI* itself offers no mechanism for merging ground truth labels with the values predicted by the deployed ML models. Therefore, concept shift detection is not supported.

Azure ML offers very similar functionality to *Vertex AI*. In terms of covariate shift detection, there is a fixed number of metrics calculated per feature depending on its statistical type, such as *Wasserstein distance* for continuous features and *Euclidean distance* for categorical ones [45]. The main difference with *Vertex AI* is that covariate shift is not detected on a per-feature basis but holistically by combining every feature metric into one scalar value for which the user can set a detection threshold. However, the algorithm for calculating this scalar value is undocumented which may cause uncertainty when setting the detection threshold. Similarly to *Vertex AI*, Microsoft's product does not offer ground truth ingestion. Therefore, users interested in concept shift detection would need to build their own solutions to augment *Azure ML*.

Out of the three products examined, *Sagemaker* is the most feature-complete one. Similarly to the others, it offers functionality for detecting covariate shift using a pre-determined algorithm. Additionally, it supports the ingestion of ground truth labels that enable the detection of concept shift using a pre-determined algorithm. The main difference compared to the other services is that it allows users to change the detection algorithms used by providing their own container image. However, this extensibility mechanism does not facilitate an explicit separation between the tasks of data scientists and software engineers. The container image that must be provided, has to execute tasks largely unrelated to data science, such as reading and writing specific files based on environment variables. Of course, the required

container image can be collaboratively developed by data scientists and software engineers with the former providing the dataset shift detection algorithms and the latter providing everything else. In such a scenario, leveraging a mechanism that automates the integration of the algorithm with the rest of the container image required by *Sagemaker*, such as the one described in sub-section 3.4.2.2, could be beneficial in terms of lowering the technical barrier for data scientists as well as communication overhead.

The use of Panoptes is not mutually exclusive with the products discussed above. By leveraging the algorithm runtime extensibility mechanism described in sub-section 3.4, Panoptes can act as an interface between data scientists and the services offered by cloud providers. This could be achieved, for example, by implementing an algorithm runtime that upon receiving a message from the orchestrator to execute a dataset shift detection algorithm as described in sub-section 3.6, would trigger the relevant service through the web API provided by the cloud provider. Upon obtaining the result it would send a message back to the orchestrator. The main advantage of this approach is the avoidance of ”vendor lock-in” since the organization can choose to migrate to a different cloud provider while keeping the way data scientists interact with the platform constant. Additionally, the organization's ML platform can be incrementally enhanced with in-house components while once again keeping the interface for data scientists constant.

3.8 Conclusion

Throughout this section, a novel, low-code solution based on MDE principles has been presented that aids data scientists in the process of monitoring their deployed ML models against dataset shift. The key takeaways from the process of developing the proposed solution and collecting the feedback of data scientists through the presented empirical study and throughout the development process are the following:

- There is a plethora of tools addressing different stages of the ML workflow. The bigger challenge for data scientists stems from the complex interactions between tools and not from a lack of them.
- Different data scientists might have different technology preferences (e.g Python vs R). It is therefore beneficial for a model-based tool targeting the data science domain to support a variety of technologies.
- Model-based solutions have the potential to address challenges in the domain of ML model operationalization.
- Web-based tools are usually preferable within a corporate environment due to security policies that might restrict the installation of desktop applications.

4 Low-code Engineering for the Internet of Things

4.1 Introduction

With the increasing interest in Low-code/no-code development platforms, more and more engineering industries are looking in a way to take advantages on what such advanced technologies have to offer. Low-code development platforms aim at easing the development of fully functional applications by facilitating people with less or no experience in software engineering (eg, business managers) to develop business applications using simple graphical user interface [46]. These platforms drastically reduce the time it takes to build an application, reducing it all from months to days or even hours. Technically, most of such applications are developed through declarative and high level graphical abstractions languages and take advantage of cloud infrastructures, automatic code generation to develop entirely functioning applications [47].

However, such application often suffers from several issues as follows. On one hand, most of the LCDPs suffer from vendor lock-in problem in which the developed application only being able to be deployed on their own dedicated infrastructure [14]. In addition to that, their scalability in terms of how big and sophisticated such developed applications can be is still an questionable. On another hand, domains supported by such platforms is still limited too. For instance, so far LCDPs have been especially successful for the development of domain-specific applications in four market segments such as database applications, mobile applications, process applications, and request-handling applications [48].

Low-code Engineering Platforms (LCEPs) on the other hand aims at overcoming the current limitations related to scalability (i.e.,supporting the development of large-scale applications, and using artifacts coming from a large number of users), open (i.e., based on inter-operable and exchangeable programming models and standards), and heterogeneous (i.e., able to integrate with models coming from different engineering disciplines) [48]. These platforms aim to span more advanced and complex domains such as Internet of Things, industrial automation, data science, recommender systems and so on. Low-code Engineering (LCE) target to extend the development knowledge present LCDP by injecting in it with the theoretical and technical framework defined by recent research in Model Driven Engineering, Cloud Computing and Machine Learning techniques [48].

Model-driven engineering (MDE) has demonstrated a significant benefit in automating the software development process by promoting the use of models as first-class citizens in which sub-systems can be designed, developed, and analyzed independently before being integrated to form a fully functional system. In this chapter, we briefly present the latest developments in CHESSIoT, a model-driven engineering framework for engineering multi-layer IoT applications. CHESSIoT brings a unique possibility for the user to employ Low-code Engineering principles to design, develop, analyze, and deploy engineering IoT systems. Through CHESSIoT, a user can benefit from a multi-view development environment in which each of the supported views has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated.

The CHESSIoT environment is built on top of the CHESS toolchain [49] with the aim of providing a fully decoupled extension for supporting the development, analysis, and deployment of engineering IoT applications. The CHESSIoT system model is used to design the system-level physical architecture spanning all the layers present in the IoT ecosystem, namely Edge, Fog, and Cloud. CHESSIoT safety analysis extends the existing CHESS Failure Logic Analysis (FLA) [50] by offering a complete cross-domains but IoT-favourable safety analysis environment. This is achieved through the modeling of component failure behavior using error propagation and transformation rules. For instance, the new extension allows the specification of IoT-specific failure behavior, such as an internal failure of a component that does not have an input port. Furthermore, the new development allows the full generation of the system's Fault Trees (FTs) as well as performing qualitative and quantitative Fault Tree Analysis (FTA).

Following a component-based design methodology, CHESSIoT offers means for the user to compose all the software components of the system, their internal decomposition as well as their functional behaviors through the use of state machines. Later, a CHESSIoT2ThingML model to model transformation can be launched to generate a fully functional ThingML source model [51]. This model can later be used to generate platform-specific code ready to be deployed on hardware devices. The same model can still be used to perform early real-time schedulability analysis, relying on the existing infrastructure provided by the CHESS tool [52].

Furthermore, the given framework enables the modeling of an IoT deployment plan in which IoT nodes

at the edge, fog, and cloud may be decomposed into services and then generates deployment artifacts (a. yaml file) suited for execution on a docker server. The CHESSIoT employs deployment agents, which are annotated to the deployment nodes in the model, to provide run-time monitoring of the deployed services. A textual language for defining deployment rules is used to describe the agents' behavior, which are later transformed into shell scripts that can be invoked manually or automatically based on the status of the deployed configuration.

4.2 Software development approach

CHESSIoT's multi-view methodology enables users to create an end-to-end IoT system by providing a multi-staged design environment. CHESSIoT extends UML/SysML languages through mean of profiles, which include system profiles for modeling physical system architectures, software profiles for modeling functional architectures, and deployment profiles for cloud-based deployment and run-time service monitoring architectures. These defined languages are IoT-specific and capable of capturing all aspects presented in any typical IoT system across the Edge, Fog, and Cloud layers. Through the new introduced *"IoT sub-view"*, the user benefits from a dedicated IoT-specific modeling infrastructure consisting of specific diagrams and palettes which ensures a fully decoupled modeling environment and it can be applied in all design stages.

For instance, the Component view allows the user to decompose the system's key software components, sub-functions, and interrelationships. Each system main sub-function has its own state machine in which events, actions, and guards are associated with states and their transitions. Communication between components is only feasible through their in/out ports by means of the payload entities. When the model is complete, a CHESSIoT2ThingML model to model transformation can be used to generate a ThingML source model, that can then be used to generate platform-specific code.

4.2.1 ThingML framework

ThingML is amongst the most popular domain-specific model-driven engineering tools for the IoT domain. It comprises a custom textual modeling language, a supporting modeling tool, and advanced code generator capabilities. The ThingML language combines well-proven software modeling constructs aligned with UML (state-charts and components) and an imperative platform-independent action language to construct the intended IoT applications [46]. ThingML code generator targets many popular programming languages such as C/C++, Java, and Javascript, and about ten different target platforms (ranging from tiny 8bit microcontrollers to servers) and ten different communication protocols [51].

In ThingML, a Thing can be defined by properties, functions, messages, ports, and a set of state machines. Like in UML, *properties* are local variables to a Thing and can be accessible only to other local behavioural parts of a Thing, such as state machines. Functions are also functional behavioural of a Thing but can also be accessed outside the owner Thing. The interaction with a Thing is enabled through required or provided ports by exchanging messages. A message can have zero or multiple parameters to specify its format. ThingML introduced another kind of a Thing called *fragment* which contains declarations of different Thing's messages. ThingML has been used in many different industrial and commercial projects [51].

Several research approaches have been shown interest in applying ThingML as their modeling or code generation framework. To mention a few, in [53] ThingML has been used to generate code for CAPS, an architecture-driven modeling framework for the development of IoT Systems. In [54] ThingML has been used to specify the behaviour of distributed software components, and later it has been extended with mechanisms to monitor and debug the execution flow of a ThingML program. In [55], CyprIoT tool used and extended the ThingML modeling language to model the behaviour of IoT things and as a code generator for platform-specific code.

Although ThingML framework is very mature and looks very promising, it can not be one shoe fit for all aspects that involve IoT systems design and development. For instance, the ThingML framework does not provide the means to conduct any system-related analyses that are very important in the Industrial IoT domain. There is also a lack of system-level design and validation in ThingML. CHESSIoT envisions having a consolidated and fully automated environment where users can combine both ThingML and CHESS technologies for industrial IoT systems development.

4.2.2 The CHESSIoT2ThingML model transformation

Modeling software component in CHESSIoT goes hand in hand with defining its behaviours, resulting in platform-specific code. The CHESSIoT component's semantics differs from the ThingML's to some extent, and that is why the mapping of the elements is needed to solicit an efficient transformation. In the following, we discuss how the different CHESSIoT modeling constructs contribute to the generation of target ThingML elements. The CHESSIoT software metamode is presented in Figure 22.

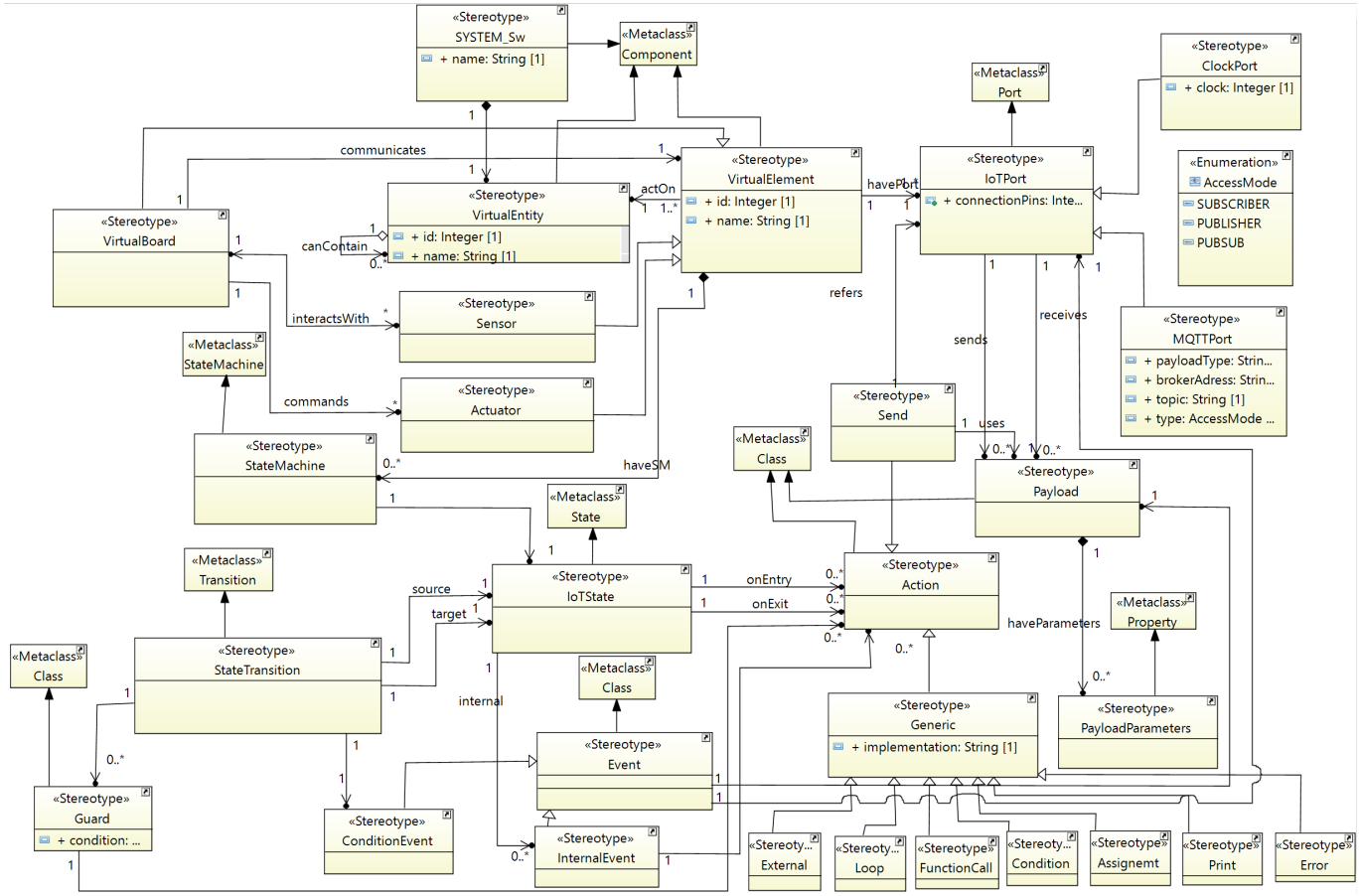


Figure 22: CHESSIoT Software meta-model

VirtualEntity: It represents the physical environment in which the devices are installed. For example, in a home automation system, virtual entities such as a room, house, car, and so on are considered. *VirtualEntity* acts as a passive entity that does not engage in the overall system functionality. This element is transformed to a ThingML project with the entire ThingML project structure.

VirtualElement: It represents any type of element that contributes to the overall functionality of the systems. In this case components such as *Sensor, Actuator, Virtual* are of its type. They are used to encapsulate the system's main part structure, operations, and behaviors. These components are mapped to ThingML's thing during the transformation.

IoTPort/MQTTPort: Ports are used to facilitate communication between two or more components that expose or require from other components' interfaces. In CHESSIoT, a port of the needed kind can be modeled to allow component messages be transmitted via port interface. MQTT ports, for example, are used simply to model a wireless MQTT port. The clock port is used to represent time-based functions like delays and periods. During the transformation process, the CHESSIoT component's ports are transformed to the **required/provided port** of a ThingML's thing, depending on the port type.

IoTState: This serves to keep the component state from its initial participation until its disposal in the system. *IoTState* extends actual UML states but in addition to that, *IoTState* carries information related to what events need to be taken care of at a certain point in time. For instance, *OnEntry* or *OnExit* events are triggered when entering or exiting a state. An IoT state can also trigger an internal event, which corresponds to an internal action to be taken. During the transformation, the IoTState will be mapped to the *ThingML state*, same goes to **State transition** that will also be mapped to their corresponding transition provided by ThingML.

StateMachine are used to hold the functional behavior of the component. During the transformation

process each component's *StateMachine* is transformed into a ThingML state charts.

Property: Properties represent variable attributes local to a component. The property can be primitive or be an instance of other components. Same as in ThingML, properties are used to retain the variable functional value of a Thing, in which during the transformation, the component's property will be mapped to thing's **property**.

Payload: This is a standalone and straightforward object to carry information to be passed between components. The payload will be mapped to a **Message** in the ThingML model. In CHESSIoT, the payload can have zero or many primitive or derived properties to be defined in a message. For instance, suppose a component message to be communicated among components contains a string value, an integer or even an instance of another payload. In this case, a payload will include three different attributes, which will be represented as message arguments in ThingML.

Action: Depending on the type of action to be performed, this can take several forms. The *Send* action, for example, is triggered to transmit the payload across a given port. When this is achieved, the message payload can be accessed on the other end via a generic action. A textitGeneric Action could be a loop type action, function call, condition, assignment, stdout print, error, or something else. Because this action type can be varied and depends on the desired outcome, it is possible to integrate platform-specific code that is fully injected in the ThingML model. *IoTAction* is mapped to ThingML actions during the transformation.

Event: Events in CHESSIoT are triggered in a different manner depending on the state of the component. An event can be of type *Conditional* or *Generic*. A conditional event is triggered during a state transition to check on the condition on the ports and execute an relevant action needed. In another hand, a *GenericEvent* is an event that gets triggered internally to the component to execute relevant internal state actions, for example, changing variable value. CHESSIoT events are mapped to corresponding ThingML *event(s)*.

State Transition: In CHESSIoT, state transitions enable transiting from the source state to a target state, abiding the trigger from the guard value. Guard expressions are boolean expressions defined based on state values. They serve to initiate a state transition by checking whether the *OnExit* event has been performed correctly. During the transformation, *State transition*s will be mapped to the corresponding transitions in ThingML.

Operation: In CHESSIoT, operations specify the functional behaviour of components. During the transformation, each component's operation is mapped to corresponding Thing's **function**.

Table 1: Proposed CHESSIoT2ThingML mapping

CHESSIoT element	ThingML element
`Component`	Thing
`Provided/required port`	Provided/required port
`Operation`	Function
`Property`	Property
`Payload`	Message
`IoTState/Transition`	State/Transition
`StateGuards`	Guards
`IoTEvent/Action`	Event/Action

Figure 23 displays a simple fan control system. This system is installed in a *Room* virtual entity. A *Button* element attached to the *Board* controls a *Fan* element as well as an *LED indicator*. The process ❶ from the figure represents an internal structure of a Room entity, whereas ❷ indicates an internal state machine of the board element. To model wirelessly communicated data, such as MQTT communication-based systems, a special port with MQTT stereotype is used which captures all MQTT-related information such as broker URL, client type and a topic. The output of CHESSIoT2ThingML transformation is a complete ThingML infrastructure which also includes including data types and the timer file required by ThingML (❸).

When the ThingML infrastructure is generated successfully, the model can now be run from a ThingML environment choosing a specific target platform and generate platform specific code as show in the process ❹. On the right side of the figure, the generated ThingML source code indicating the ports, payloads and state machine and how they map back to the CHESSIoT model is shown. Finally, the snippet ❺ shows the Proteus Simulation Software[1] in which the generated code was been simulated for deployment without issue.

[1]https://www.labcenter.com/

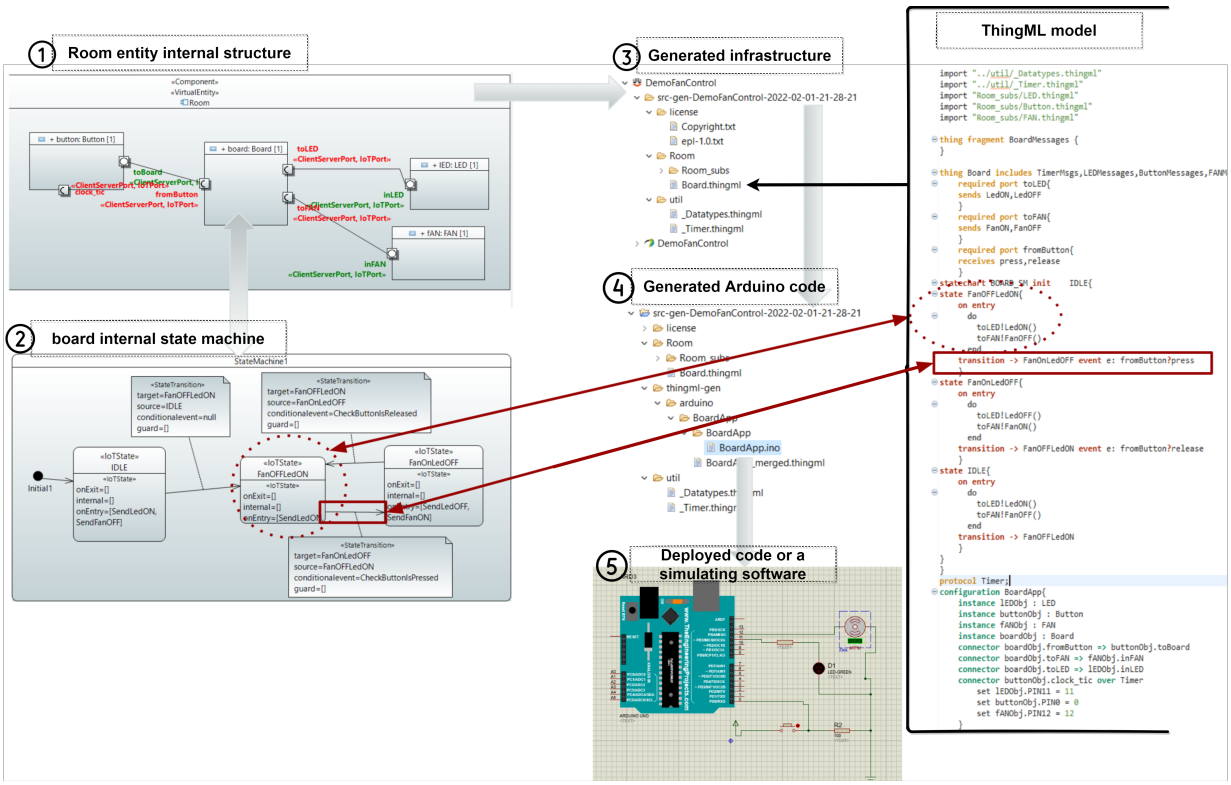


Figure 23: Fan Control example

4.3 System-level design and analysis

Due to the heterogeneity present in the IoT ecosystem, deciding the levels of abstraction can be tricky. Therefore, the CHESS tool provides a multi-view modeling environment by providing six main views: system-view, component-view, deployment-view, analysis-view, requirement-view, and PSM-view. Each supported view has its own underlined constraints that enforce its specific privileges on model entities and properties that can be manipulated. Furthermore, depending on the current stage of the design process, CHESS sub-views are adopted to enhance specific design properties or steps of the current process. More specifically, the system-view aims to provide a system-level modeling environment focusing on the physical aspects of the system to support early-stage analysis.

As the CHESS tool is a cross-domain modeling and development environment, depending on the domain, concepts can vary accordingly. To tackle that, it allows extending the current view with additional domain-specific sub-views, which can be activated based on the domain or the design development stage. This, in turn, draws specific constraints on specific model elements and palettes while offering a better user experience. To support IoT system-level modeling, CHESSIoT introduces *"IoTSub-view"*. Once applied at any design stage, the user will benefit from a dedicated IoT-specific modeling infrastructure consisting of specific diagrams and palettes.

In addition to that, we have defined a high-level UML/SysML profile extension to reflect the construct and semantics present in IoT system-level architectures. This was achieved by defining IoT-specific stereotypes and nomenclatures that better fits in IoT perspective. Figure 24 presents the abstract syntax of the CHESSIoT system-level profile. Such a level considers only the main physical architectures involved in achieving a full multi-level IoT system without covering any aspects related to system functional behavior. By referring to Figure 24, the *System* element defines a conceptual representation of an IoT system under development as a whole, and all the other elements are defined under it. Other sub-systems can also be contained, making it possible to design IoT system-of-systems architectures. To better support all the layers present in the IoT ecosystem, the CHESSIoT System profile spans all the layers, typically from the edge to the fog and the cloud layer.

At the edge layer, a *PhysicalElement* can be of any type, ranging from a tiny micro-controller to an element as big as a car, a plane, or a house. Any physical component that can play a part at the edge layer is represented as a physical element. The system can have one or more physical elements; each can have one or more communicating ports. The following four main element types extend the physical element. A *PhysicalEntity* on the other hand, can be almost any object or environment on which a running physical

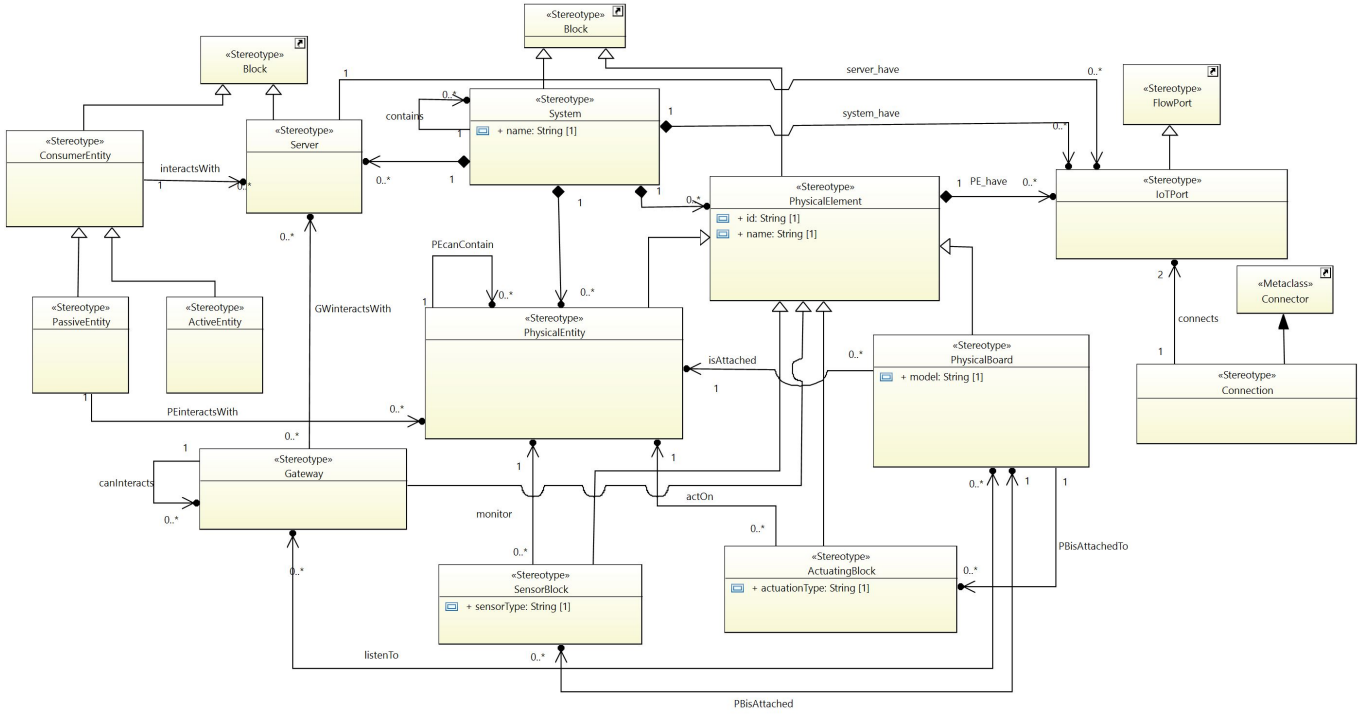


Figure 24: IoT System-level meta-model

element can act. A self-driving car software, for instance, runs on various boards attached to the car but not on the car itself. So a car is a physical entity in this case. Furthermore, a *PhysicalBoard*, as expected, represents a physical controller where the software runs. For example, a RaspberryPi board processing data collected by different sensors deployed in different building parts. Finally, the *SensorBlock* represents a sensor component at the thing layer that is in charge of collecting environmental data. In contrast, the *ActuatingBlock* in this case represents any actuating component.

At the fog layer, we only have the *Gateway*, which is just any kind of device that serves as a link between the physical world of things and the virtual world, in this case, a cloud infrastructure. This element communicates with both the remote server and the physical board. The CHESSIoT system-level model does not include any information related to any functional behavior of such a component but the main physical construct of the system. The cloud layer, on the other hand, is made up of a *Server* and a *ConsumerEntity*. A server is an element of the cloud infrastructure that stores data and cloud services. A consumer entity is any third-party element that can communicate with the server to consume its data. Depending on its role in the system, this component can be *active* or *passive*. A computer, for instance, that runs software to visualize and control remotely deployed sensors qualifies as an active consumer entity. In contrast, a traffic light actuator that receives commands from the server to function will be qualified as a passive consumer entity.

4.3.1 Safety analysis approach

Failures that could result in a risk to human life, injuries to the environment or properties are referred to as safety hazards. Indeed, IoT systems might fail because of a wide range of reasons: devices age, data sources issues, network problems, deployment environment, as well as external environment constraints, such as the human error. CHESSIoT safety analysis approach offers means to perform early safety analysis through the use of FTA.

Typically, safety analysis processes begin with the system engineer creating the model from the gathered system functional requirements. The system-level model includes all of the system's major functional components, parts, sub-components, and interconnections. The system components can be expressed as blocks in SysML Block Definition Diagrams (BDD), reflecting the abstract syntax meta-model depicted in Figure 24. Internal Block Diagrams (IBD) are used to depict their interdependences. The interactions between components aid in determining error propagation paths. Each part (block) can be assigned to a specific architectural subsystem or component at the most basic level. The physical architecture is expected to be extensible, allowing for adding new components/blocks as needed. Figure 25 depicts the entire safety analysis process.

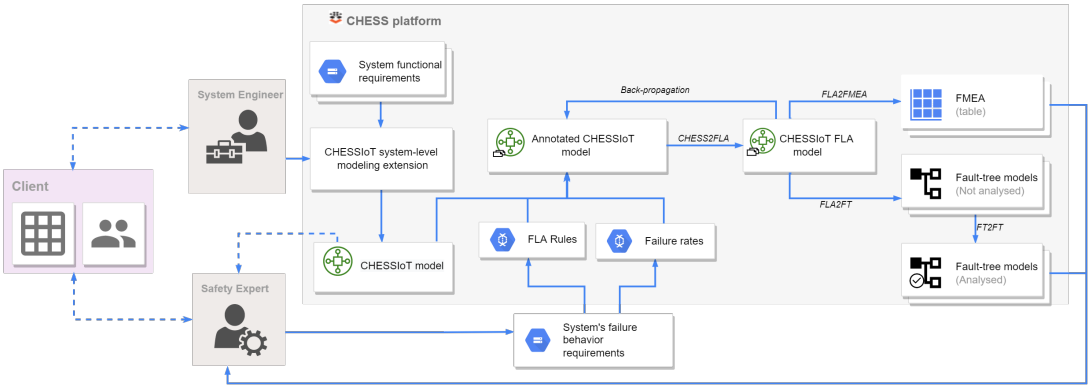


Figure 25: Safety analysis process

Once the system model is complete, it can be handed to the safety engineer for further safety analysis. Starting from identifying the typical system-level failures, the safety engineer derives the failure behavior for each component following the Failure Propagation Transformation Calculus (FPTC) notation. The CHESS-FLA analysis considers the component functional decomposition down to fundamental sub-functions. Then, each lower level fundamental sub-function is systematically evaluated to identify all its potential failure modes. The analyst must establish error propagation or transformation rules for each possible input failure or internal failure of the component. At this point, the CHESS-FLA transformation can be launched, generating the FLA model.

In practice, a component can act as a source of failure (for example, by causing a failure in output due to the activation of internal faults) or as a sink (a component is capable of avoiding failure propagation by detecting and correcting the failure in input). Furthermore, failures in a component can be propagated (i.e., a failure can be passed from input to output) or transformed (by changing the nature of the failure from one type to another from input to output) [50]. The model then undergoes CHESSFLA model to model transformation in which each lower level fundamental sub-function is systematically evaluated to identify all of its potential failure modes.

At each level of the analysis, the FLA results are back-propagated onto the original model to guarantee that each component's final state of failure is reflected in the model only after the study is done. The system FT and FMEA table can be automatically generated and analyzed before being sent back to the safety expert for consultation. If something is not right, changes can be made before the final inspection is completed. In the following sections, we go over each step of the supported analysis process in greater detail. Although the FMEA analysis is included in the CHESS safety analysis process, this chapter will only focuses primarily on the FT-based analysis approach.

4.3.2 Fault-Tree model

The FT generation process is performed following a FLA2FT model for the transformation of a CHESS-FLA model into a conforming FT model. Figure 26 shows an FT meta-model adopted from [56]. The FTModel element is the top element of the tree, and it is instantiated for each failure that propagates to the system output ports during the FLA analysis. Each FTModel element contains a logical network of events and gates that together form a FT. The entire FT generation process will be covered in the following sections.

4.3.3 Fault-tree events

In our proposed approach, each event can be graphically identified in the FT from its unique identifier (ID). An event ID is defined as a pair of "failure name" and "port name" in a given component. This ID never changes through the FT generation as well as in the FT analysis process. This can potentially help in the event tracking when comparing generated and analysed FTs. In addition to that, each event has its own name which by default combines the information regarding its corresponding failure and its effect in a given component. The effects can be of type *top failure* type at the system level, *local failure* caused by the system's intermediate failures across the tree, *injected failure* resulted from injected faults, and *internal failure* resulted from the component's internal faults. In the next listing, we will go over the various event types that we use to construct the FT and we will describe how events are generated throughout the transformation process.

- **Basic events:** A simple component may suffer different kinds of malfunctions, generating either one

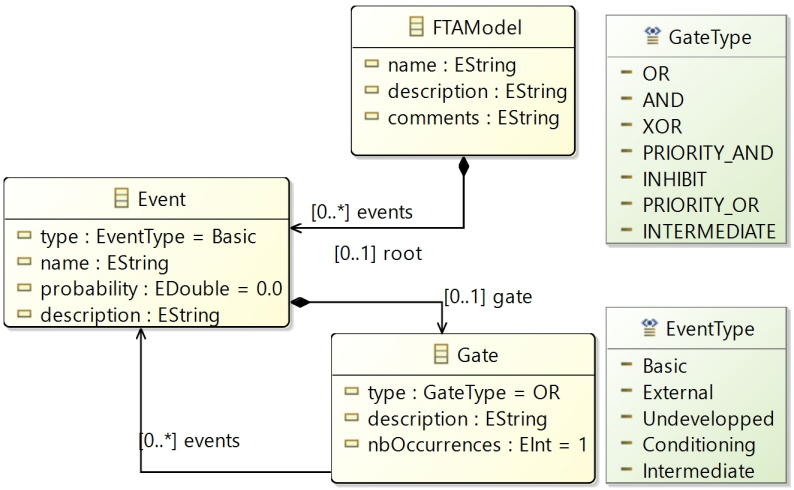


Figure 26: FT meta-model [56]

or more kind of internal failures. One or more notations may be required to define such events for a given component. A basic event is used to represent a failure that is initiated inside a simple component. This can be basically referred to a simple component acting as a source of failure. In this case, a failure condition is present on any of the output ports despite a no failure at its input port. In case a simple component does not posses any input port, the newly developed approach allow the definition of such condition following Expression *(1)*. On the other hand, in case a simple component possesses one or more input ports, its failure behaviour can be defined by explicitly initializing all the input failures of the component with a *"noFailure"* condition (Expression 2). Figure 27a shows the basic event representation in a FT resulted from an internal failure of a component.

$$(*) \rightarrow p.failure_{(out)}; \tag{1}$$

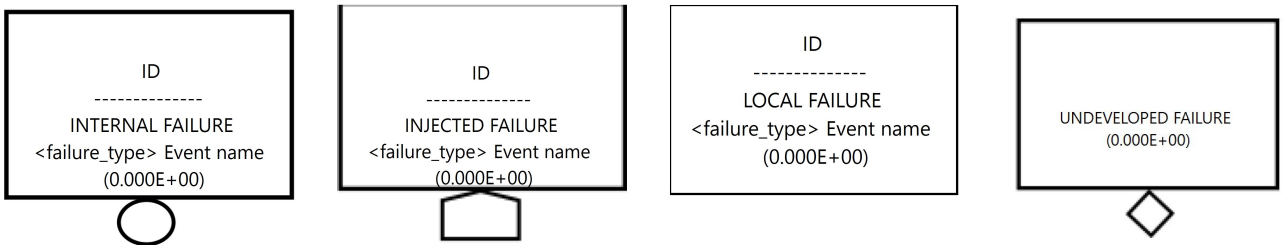
$$p_1.noFailure, ..., p_n.noFailure \rightarrow p.failure_{(out)}; \tag{2}$$

NOTE: *Considering p_1,p_2 to p_n to be the input ports of a simple component. If any of their corresponding failure condition is different to* ***"noFailure"****, then the above condition is not met, so, all* ***"noFailure"*** *conditions on other ports are ignored as they do not contribute toward to the logical failure behavior of the component.*

- **External events:** External events are used to represents failures that can be introduced from the environment outside the system boundaries. CHESS provides possibility to inject failures in the system through the system-level input ports. These faults are modeled with a comment annotation with *<<FPTCSpecification>>* stereotype attached to the relevant input port where the fault is being injected in SysML block diagrams. The injected fault comment specify the type of failure attribute being injected in the system. This fault injection can also be done on a composite sub-system under analysis. Figure 27b shows a graphical representation of an external event resulting from an injected fault in the system.

- **Intermediate events:** An intermediate event is used to describe the local failure effects resulting from a logical output of one or many events. In the presented approach, these events are generated to represent the failure condition at the input or output port(s) of the simple component resulting from an internal failure or other failure condition from the outside of that simple component. It is also used to represent the top event of a FT. Figure 27c is used to represent an intermediate event.

- **Underdeveloped events:** The underdeveloped event is used to specify an event resulting from a failure in which we do not have sufficient information about it. This can basically happen when a failure is introduced at the input port of a simple component without a preceding definition of how it was been propagated or injected at that input port. During the FT generation process the symbol in Figure 27d is used.



(a) Basic event (b) Injected event (c) Intermediate event (d) Undeveloped event

Figure 27: Event types

4.3.4 Failure propagation

A failure propagation occurs in a component when a single input port failure condition of a component is directly transferred to the output ports of the same component without changing its nature. This failure propagation can be modeled in CHESS-FLA using the notation in Expression 3. A propagation also occurs between two connected components, when a failure condition at the output port of the preceding component is transferred to the input port of the following component.

$$p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_1; \tag{3}$$

4.3.5 Failure transformation

A failure transformation occurs within a component when a failure condition present at the input port of a simple component is converted into another type before reaching the output port (Expression 4). A failure transformation can also occur when more than one failure expressions of any type the exception of a *"noFailure"* or *"wildcard"* at multiple input ports are transmitted to a single output port (Expression 5). Event if the failure has the same type, the fact that the component converts two failures at its input ports to a single failure at the output port is regarded as a failure transformation.

$$p_{(in)}.failure_1 \rightarrow p_{(out)}.failure_{(out)}; \tag{4}$$

$$p_{(in1)}.failure_1, ..., p_{(inN)}.failure_N \rightarrow p_{(out)}.failure_{(out)}; \tag{5}$$

4.3.6 Fault-tree generation process

The system FTs are generated through a series of model-to-model transformation mechanisms written using the Epsilon Transformation Language (ETL) [57]. The process starts by instantiating a number of FT objects equal to the number of failures that propagates to the output port(s) of the system. At this stage, each error that propagates to the output port(s) of the system is represented into its own FT. Note that when a *"noFailure"* condition is propagated to the output, it is ignored. This technically means that the system acts like a failure sink and it is able to mitigate its propagation to the output of the system, which is also true for all other sub-systems. To achieve that, the Algorithm 1 is followed.

Algorithm 1: Instantiate fault-tree

```
for p in allPorts do
    if p is output of the system then
        for f in failures assigned to p do
            if f is not "noFailure" then
                Create an FT relative to the failure f and p;
                Add FT to FTs sequence;
            end
        end
    end
end
```

In the next steps, each FT is built separately and recursively. The initial action involves the creation of a top event among all. A top event is generated as a result of the failure propagation to the system output port. In terms of logical gates used in the FT, only "AND" and "OR" gates are adopted. An **AND** gate is

used to indicate a failure transformation from an input to an output port of a component (see Section 4.3.5). An **OR** gate, on the other hand, is used to depict a failure propagation situation (see Section 4.3.4). The OR gate can also depict a scenario in which one or more failure outputs from distinct components are passed to the input of the following component. The whole FT generation population algorithm is described in Algorithm 2.

Algorithm 2: FLAComposite2FT rule algorithm

```
Data: FLA composite component (system-level)
Result: FT model
count=0;
for port in allPorts do
    if port is output of the system then
        for f in failures assigned to port do
            if f.name is correspond to an FT then
                Create an intermediate event ← TOP FAILURE;
                Assign an OR gate to it;
                Add it to its corresponding FT;
                for con_port in port.connectedPorts do
                    recurseFT(f,con_port,FT,topEvent);
                end
            end
        end
    end
end
```

When the top-event creation is done, the intermediate events are created and populated into the FT, based on the failure expressions and their components they are assigned to. The FT population involves a recursive transformation process in which from a component, we can have information on ports and from ports we can get to rules, rules to expressions and back to the components. So, at this stage the only crucial stopping case is when the transformation hits a condition matching an internal, underdeveloped or an injected failure. For instance, in Figure 28, a simple transformation example with indications showing a simple transformation mapping of the Expression 5 is shown. From the example, each of the output expression is mapped to an output event of a logical combination of the input expressions. Each input expression is mapped to an event and the type of such event is determined by the expression condition. In addition to that, the logical gate is defined based on the nature of the input expressions to satisfy the failure propagation and transformation concepts. A brief description of the followed algorithm is described in Algorithm 3.

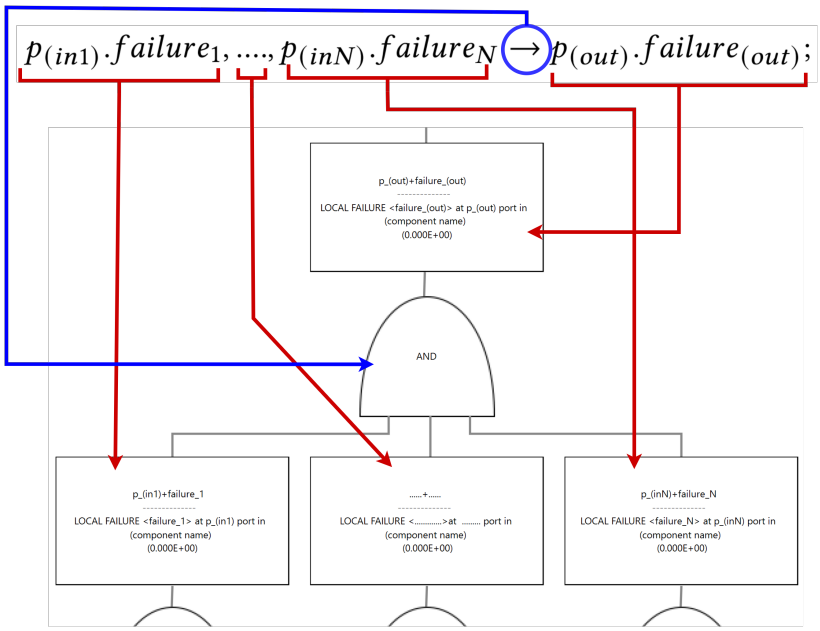


Figure 28: Expression 5 corresponding tree

In the newly developed environment, it is possible for a safety expert to assign failure rates of any of

the events leading to basic failure condition. These events includes the internal failure condition and the injected failures. In addition to the failure rate assignment, the user is also able to add a failure description in a textual format to reflect the proper cause of the failure. This is potentially important when, for instance, a simple component might have two different internal failure condition leading to two different outputs. For instance, an aging device can still work by providing wrong output leading to a *valueCoarse* or *valueSubtle* failure at the output port whereas a blown device fuse will automatically halt its functionality therefore leading to an "omission" failure at the output port. Assigning such information to the FT model will eventually improves its readability. This assignment is done way before the FLA2FT transformation process and when the transformation finishes, a probability file is generated separately from the model in an excel file format. This file is later loaded to the model to support the quantitative probabilistic analysis process.

4.3.7 Fault-Tree Qualitative analysis

The FT qualitative study is conducted using an FT2FT model-to-model transformation in which the FT meta-model in Figure 26 serves as both the source and the target meta-model. This effectively creates new FT representations in the workspace, permitting users to reuse both the generated and the analyzed FTs at the same time. The goal of this qualitative analysis approach is to provide a new representation of the existing FT that only includes the essential representations. Although the current qualitative analysis does not fully reflect the calculation of the minimal event sets needed for a system to fail (minimal cut-sets [58]), it does provide a much shorter and more readable FT that still reflects the goal for the analysis.

During the qualitative analysis process, the following actions are performed:

- ***1. Removal of internal component failure propagation:*** One of the goal of a FT is to help users to discover and trace down the source event of a system failure in a more easy and intuitive fashion. As described in Section 4.3.4, the internal component failure propagation occurs when a single input port failure condition of a component is directly transferred to the output ports of the same component without changing its nature. Although keeping such information in the FT is important, when the model becomes increasingly big, these information can be very exhausting to glance. Therefore, each path meeting such condition is omitted and removed from the FT. This process drastically reduces the vertical magnitude of a FT but it does not change its nature.

- ***2. Removal of external component-to-component failure propagation:*** This refers to a condition in which a component to component propagation is solicited from a single channel in the FT. For instance, if a single failure condition at the input port of a component is propagated from a single source, then this information is omitted in the analysed FT. Mainly on the basic events, events involved in a failure transformation and the top event are kept in the FT.

- ***3. Removal of basic events redundancy:*** A single failure can be initiated from a single source and passes through different propagation paths until it reaches the output port(s). If, in all the propagation paths, no transformation occurs, then, from the output failure conditions, only one path is considered and, from that path, all intermediate propagation representations are removed according to the two previous rules.

During the transformation process, only the paths that satisfy the internal or external failure transformation are kept in the tree. This is to help users to only care about the important information when tracing the origin of the failure. One special case of the propagation that is kept in the tree is when the FT that has to be analysed contains a single path in which a single basic event propagating up the tree all the way to the top event. Then, in such case, only the top event and the basic event are kept in the analysed FT. Finally, each of the omitted intermediate paths as well as each gate resulting from a simple component internal failure transformation is replaced by a feed-forward intermediate gate to enhance the FT readability.

For example, taking the generated FT branch shown in Figure 29, the event ⓿ is obtained from a logical "AND" output from 3 subsequent paths which makes this event a result of a component to component external transformation. The going doing to a single branch of our interest, from event ❷ with *"omission"* at the input port to event ❶ with *"commission"* at the output port indicates internal failure transformation. So in such case event ❶ and its following gate is kept permanently while event ❷ is kept temporally for future analysis. Going down to event ❸ with *"omission"* to ❷ with *"omission"* which is a component to component failure propagation, so event 3 will be permanently removed.

Algorithm 3: Recurse FT operation

```
Data: current_failure as f, current_port as p, current_FT as FT, event_to_construct as eG
Result: final populated FT
if p is of not system's port then
    if p is of a "Simple Component" then
        for outExp in p.owner.rules.outputExpressions do
            for f1 in outExp.failures do
                if f1==f then
                    Create an intermediate event e0 ← LOCAL FAILURE;
                    Assign gate to e0 based failures at the p port;
                    Add e0 to eG;
                    Add e0 to FT;
                    for inpExp in p.owner.rules.inputExpressions do
                        for f2 in inpExp.failures do
                            if f2 is not a "wildcard" then
                                if f2 is a "noFailure" then
                                    Create a basic event eT ← INTERNAL FAILURE;
                                    Add eT to eG;
                                    Add eT to FT;
                                else
                                    Create a intermediate event e1 ← LOCAL FAILURE;
                                    Assign an OR gate to e1;
                                    Add e1 to e0;
                                    Add e1 to FT;
                                    for p1 in inpExp.port.connectedPorts do
                                        recurseFT(f2,p1,FT,e1);
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    else
        for p1 in p.connectedPorts do
            recurseFT(f,p1,FT,ev);
        end
    end
else
    Create a external event eX ← INJECTED FAILURE;
    Add eX to eG;
    Add eX to FT;
end
```

Next we remain with event ❹ with *"omission"* to ❷ with *"omission"* which is a propagation as well, so because event ❹ is a basic event so event ❷ will be removed instead. Finally, the whole omitted part of the tree will be substituted by feed forward intermediate gate to enhance readability of the FT. The final version of the FT is provided in the right hand figure 29(b). The internal failure leading to an *"omission"* at the output port of a given component had transformed into an *"commission"* at some point in the system in which when combined with other two failure sources had cause a top failure event with a "omission" failure type.

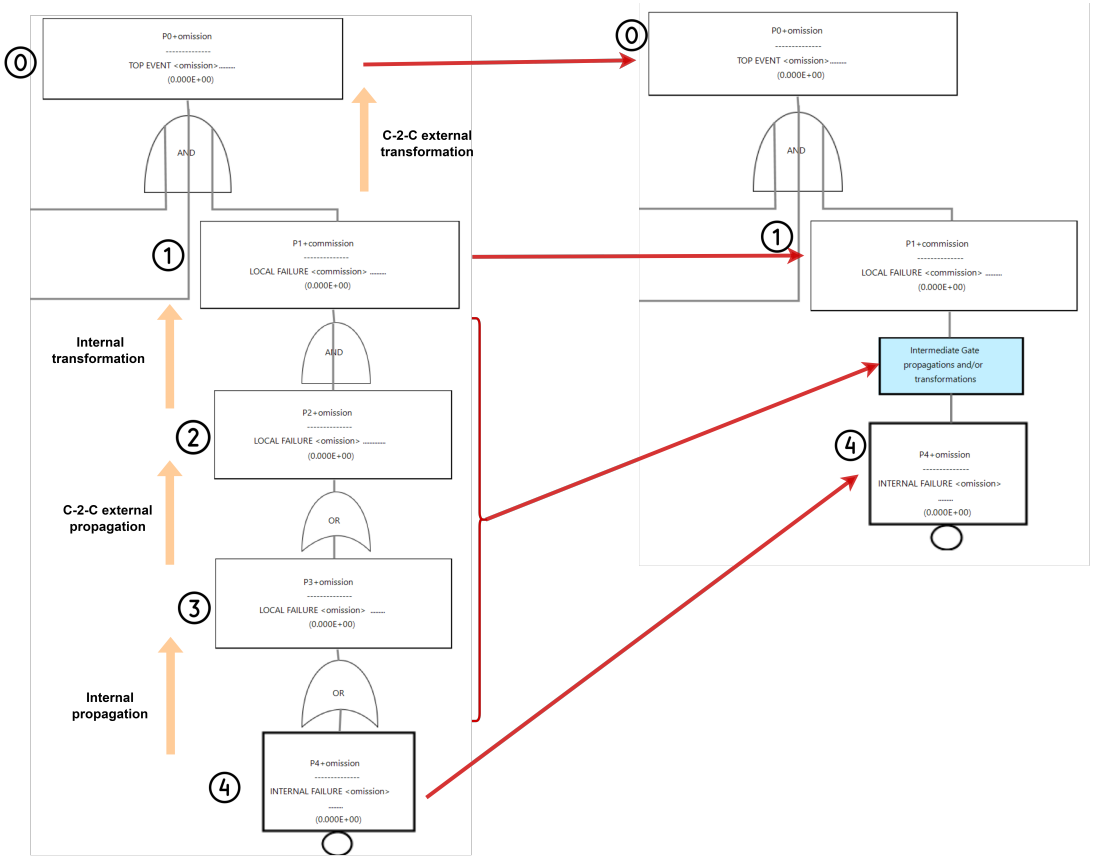


Figure 29: Qualitative transformation example (a) before, (b) after

4.3.8 FT Quantitative analysis

The quantitative probabilistic analysis is meant to automatically calculate the system-level (top event) failure rate. In the proposed approach, the user is able to assign the failure probability rates of the basic failure events such as internal failure and injected failure. This information can be supplied form the device manufacturer's data-sheet as well as the safety experts. The probability calculation follows a widely used formula for conducting a logical output of an "AND" or an "OR" gates in the FT [59, 60]. The output of an "AND" gate means that the output event will only happen when a combination of independent events occur at the same time. On the other hand, the output of an "OR" gate implies that the output event will occur if anyone of the input events occurs.

For each FT to be analysed, the system failure rate (the top event probability) is calculated following a recursive calculation of the intermediate probabilities to the intermediate events. Based on the probabilities of the basic events, the probability values of their parent event can be calculated from input event probabilities. The probability calculation follows the formula in Figure 30. Let N be the number of input events and P_{in} the probability of the input event, the output probability P_{out}, for both "AND" and "OR" gate types, is calculated as follows:

$$P_{out} = \begin{cases} \prod_{in=1}^{N} P_{in} & \text{for an } AND \text{ gate} \\ 1 - \prod_{in=1}^{N} (1 - P_{in}) & \text{for an } OR \text{ gate} \end{cases}$$

Figure 30: Probability calculation formula

During the probability calculation process, in case of an internal component failure transformation process, the failure probability at the input port is forwarded to the output port event. In addition to that, an probability of an event resulted from an undeveloped event being fed into an "OR" gate is first set to zero while in case of an "AND" gate is set to 1. This in fact does not affect the probability calculation process because the 0 and 1 numbers are neutral in the addition and multiplication process respectively.

4.4 Deployment design and automation approach

The CHESSIoT deployment modeling is performed under the *"Deployment view"* with the aim to support the decomposition of IoT system nodes namely Edge, Fog and Cloud layers. A node is considered as a composition of one or more machines in which each machine is internally composed by one or more services. As indicated from the meta-model shown in 31, a service can be of type broker, storage, data distribution, external application and so on. Each service have specific propoerties which are required to fulfill the runtime configuration of the corresponding docker container. For instance, a broker which is also a service captures its specific run-time properties such as type, anonymous access, persistence, username and password.

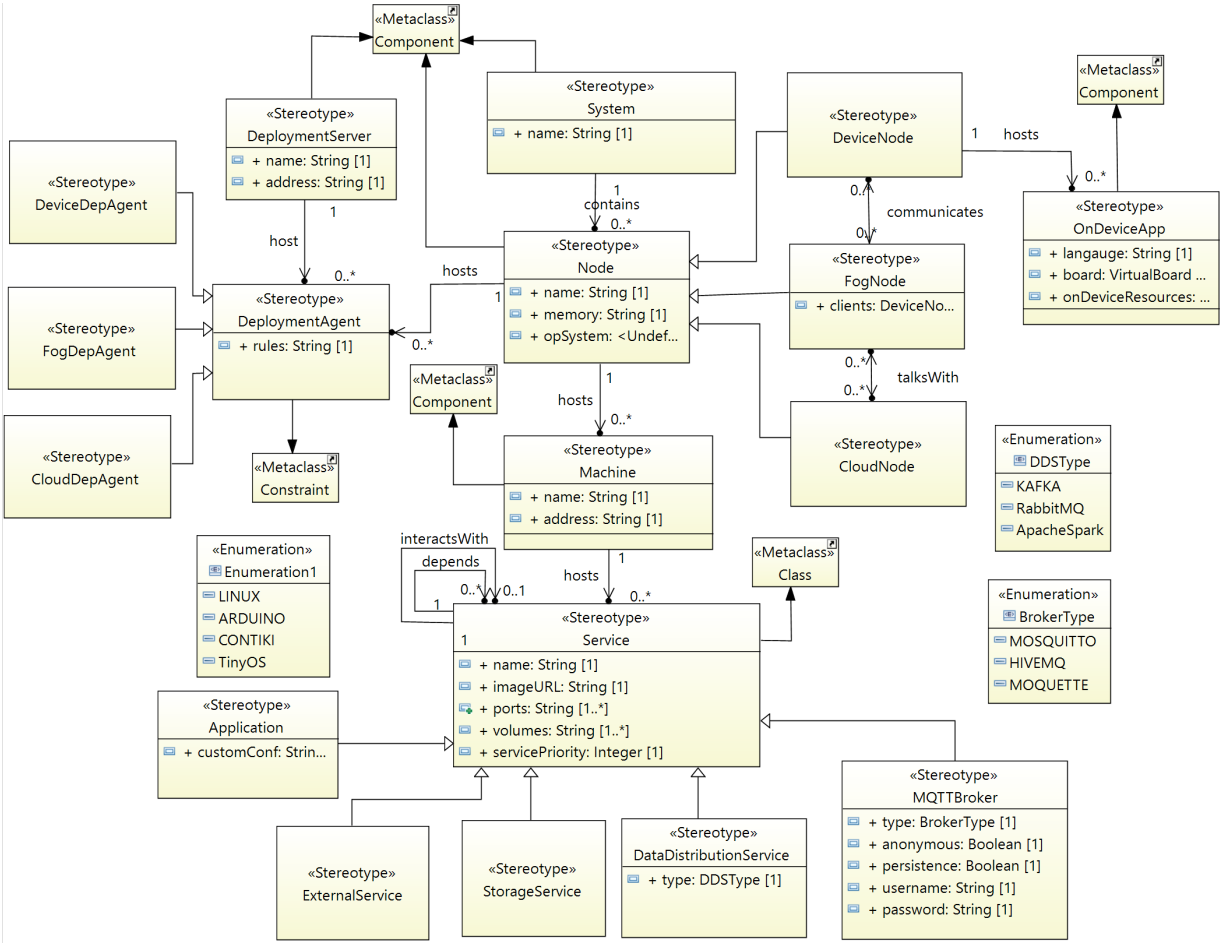


Figure 31: CHESSIoT deployment meta-model

At high-level, nodes can communicate between themselves through services that they hosts. For instance an MQTT broker hosted on a FogNode will have to communicate with an on-device applications. *"DeviceNode"* contains one or more on device application deployed on the devices. At each node, a number of *"DeploymentAgent"* can be attached to hold run-time deployment rules which are further used to generated Ansible scripts [61]. In the next Section 4.4.1, we will detail the runtime deployment automation modeling approach supported by CHESSIoT.

4.4.1 Run-time deployment automation modeling

To support the efficient deployment and monitoring automation of the deployed services at run-time, CHESSIoT provide means for defining the deployment rules through a textual grammar. This DSL support mainly the monitoring the life-cycle of deployed service containers through the use of so called *"agents"*. As shown in the DSL metamodel depicted in Figure 32, each node is annotated with run-time deployment rules which are referred to as *DeploymentAgents* depending on the envisioned runtime activities that are to be expected. The rules covers actions such as starting, stopping, restarting, redeploying, logging and monitoring.

The current CHESSIoT deployment implementation enable a user-friendly environment and in case no data is provided for a given property, default values are used instead. For instance, the runtime deploy-

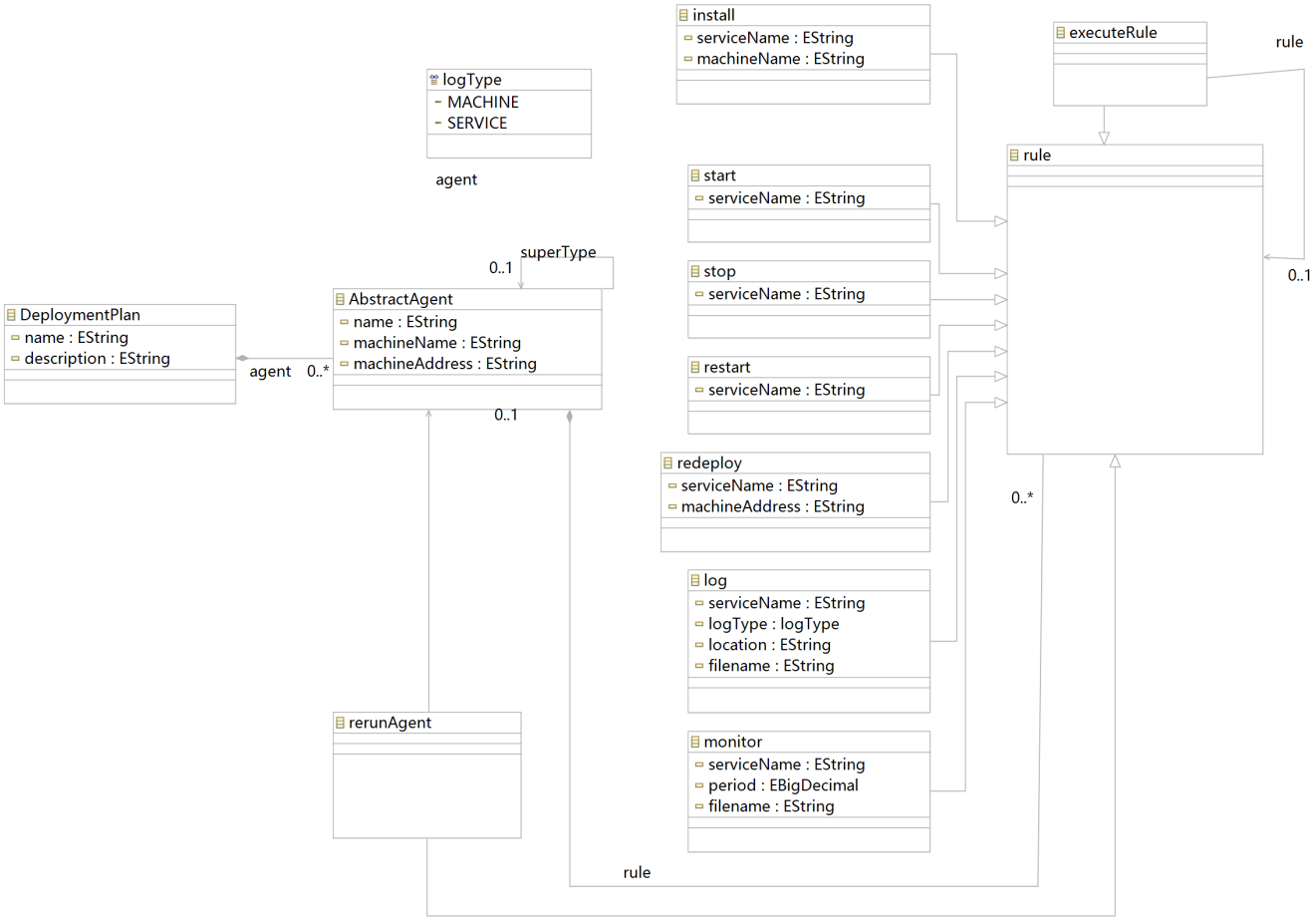


Figure 32: Runtime deployment automation DSL

ment language is injected the process the modeling procedure at runtime in which the autocompletion dependencies are suggested by referencing directly from the model data present in the model. In addition to that, each node can be annotated with one of more runtime deployment rules (agents) directly in the model. In the long run, the runtime deployment language will support the following main activities:

- Service life-cycle
- Service deployment
- Logging and monitoring
- Runtime self-adaptation (not yet implemented)

An example of the deployment rules is depicted in the listing 5. When the model is complete, these deployment rules are then transformed into a set of shell-scripts that are meant to be invoked either manually or automatically according the run-time status of the deployed configuration.

```
1   DeploymentPlan:Plan1 {
2     description:"install start and stop NodeRED on FOG node"
3     agent installandStartNodeRED{
4       machineName:"machine2"
5       address:{
6         ["123.123.123.123","124.345.245.123"]
7       }
8       RULE:install:"nodeRED"on"machine2"
9       RULE:start:"nodeRED"
10    }
11    agent stopAndRedeployMosquittoOnMachine3{
12      machineName:"machine3"
13      address:{
14        ["123.123.123.123"]
15      }
16      RULE:stop:"Mosquitto" on"machine3"
```

```
17        RULE:re-deploy:"Mosquitto"on"machine3"
18      }
19    }
```

Listing 5: Runtime deployment rules example

4.4.2 Deployment artifacts generation

When the model is complete, a model to text transformation can be launched which will generate a full yaml file containing the deployment configuration of the services at each machine ready to be executed on a docker server. Figure 33 depicts a fragment of CHESSIoT to yaml translation code written in the Acceleo M2T transformation language. As can be seen, each service goes through a separate transformation path before being added back to the parent configuration file. For example, if a service is of the type *"Broker"* and the anonymous access mode is set to false, a password file will be generated. This is an ongoing development, and we hope to support as many IoT-specific services as possible in the future. The full code and the instruction on how to use CHESSIoT extension can be found at https://github.com/fihirwe/CHESSIoT-features

```
[template public generateInfrastructure(name : String,node : Component, elt:Package ){
ExternalService : String = 'CHESSIoT::CHESSIoTDeployment::ExternalService';
DataDistributionService : String = 'CHESSIoT::CHESSIoTDeployment::DataDistributionService';
MQTTBroker : String = 'CHESSIoT::CHESSIoTDeployment::MQTTBroker';
ServiceST : String = 'CHESSIoT::CHESSIoTDeployment::Service';
StorageService : String = 'CHESSIoT::CHESSIoTDeployment::StorageService';
}]
[file ('/'+name+'/docker-compose.yaml', false, 'UTF-8')]
[generateCopyright()/]
[generateLicense()/]
version: "3.9"
services:
[for(c:Class | getSubClass(node,elt))]
[if(c.getAppliedStereotype(MQTTBroker)->notEmpty())]
[generateBroker(c,node,'/'+name+'/')/]
[elseif(c.getAppliedStereotype(DataDistributionService)->notEmpty())]
[generateStorageService(c,node)/]
[elseif(c.getAppliedStereotype(StorageService)->notEmpty())]
[generateDataDistributionService(c,node)/]
[elseif(c.getAppliedStereotype(ExternalService)->notEmpty())]
[generateExternalService(c,node)/]
[/if]
[/for]
networks:
[for(c:Class | getSubClass(node,elt))]
  [c.name/]_net:
    driver: bridge
[/for]
[/file]
[/template]
```

Figure 33: CHESSIoT to .yaml M2T transformation

4.5 Safety Analysis example: Patient monitoring system (PMS)

Due to the general rapid evolution of electronics and information technology, more powerful bedside patient monitors capable of complex bio-signal processing and interpretation are becoming available, and they are usually equipped with some highly specialised communication interfaces [62]. This goes hand in hand with the huge advances in IoT technologies allowing the integration on such devices of the capability to connect to the internet, which makes it possible to monitor the health state of multiple patients remotely. To support our evaluation process, we adopted an "Efficient Patient Monitoring for Multiple Patients Using WSN" case study [63]. The case study is an advanced system capable of reliably monitoring the multiple parameters of up to six hospitalized patients simultaneously in real-time.

The system investigates the potential of employing Wireless Sensor Networks (WSN) to reliably and wirelessly collect multiple parameters such as blood pressure, temperature, electrocardiography (ECG), electroencephalography (EEG), and pulse oximeter (SPO2). These parameters are collected through a set

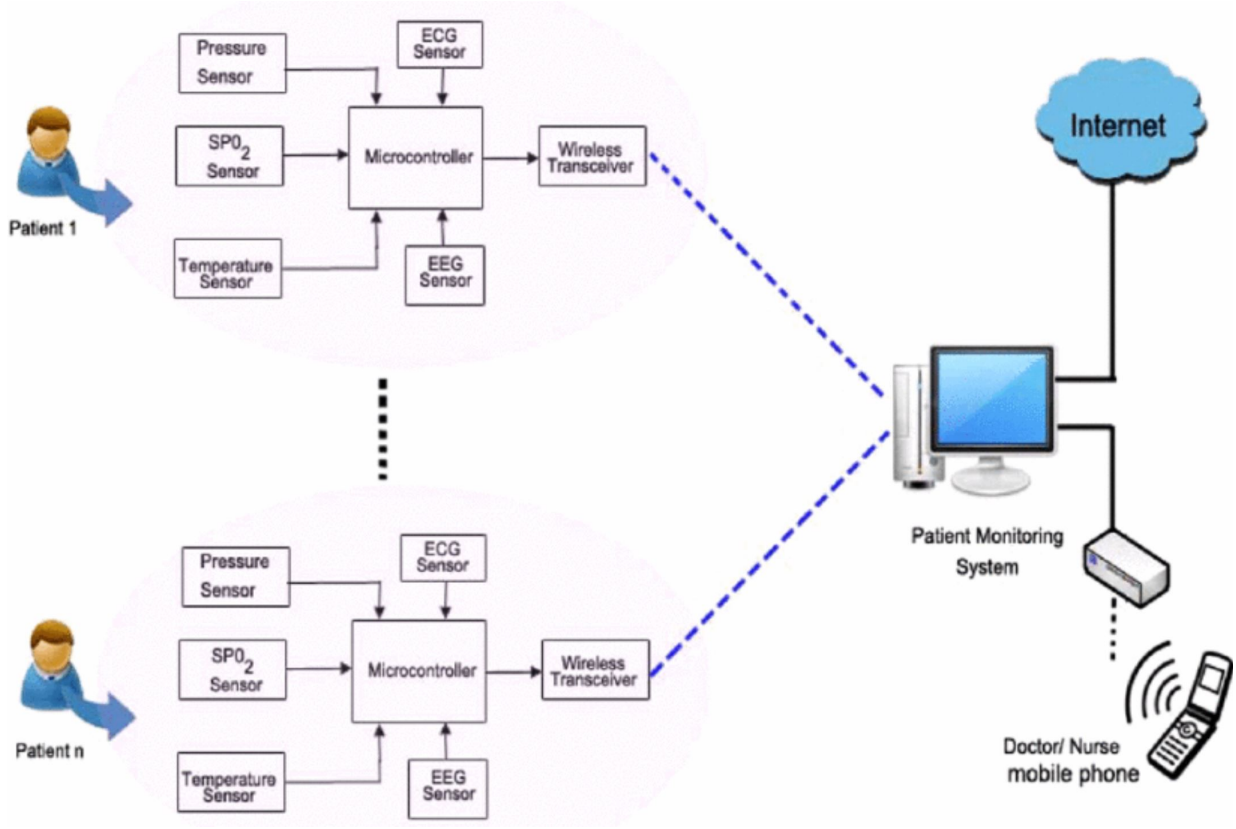


Figure 34: Basic architecture of Patient Monitoring System [63]

of sensors placed on different parts of the patient's body. For instance, the ECG Sensor is placed on the chest and on the limbs to extract patient's heart rate data, the EEG sensor is placed on the patient's head to read electrical activity generated by the brain. Furthermore, the Blood Pressure sensor is placed on the arm to detect the level of the pressure in the blood; while, the SPO2 sensor is placed on the patient's finger to measure the oxygen saturation of a patient's blood; and, finally, the Temperature sensor is placed to any part of the body to measure the temperature. Figure 34 describes the high level architecture of the system.

As a result, the recorded parameters are wirelessly transmitted to a computer running PMS software, which feeds them on the monitor screen in the doctor's office. The software can also wirelessly send alarming messages to the doctor's phone, if they are not present, to respond to the patient's requests. We will describe the architecture in great details in the next Section 4.5.1.

4.5.1 PMS system design

As it can be seen in Figure 34, the Patient Monitoring System (PMS), uses a set of sensors to collect sick patient data and send them to a remote server. The system can display the data on the monitor as well as sending alarming signal when something gets wrong. Figure 35 represents the internal physical architecture of the proposed system. For the sake of simplicity and to facilitate the analysis process able to produce presentable results in a paper, we have considered the following changes to the architecture presented in 34. Firstly, we designed a PMS that only monitors a single patient. Secondly, we introduced a remote server component that acts as a bridge by hosting the service that saves the received data and exposes them to other third parties services who might need them. Thirdly, we replaced the doctor's phone sub-system with an alarming system component that receives data from the PMS software on the monitor side. Finally, we added a "Human" component to reflect the role of a doctor in the overall system functionality.

As shown in Figure 35, a *"SensingUnit"* composite component consisting of five sensors namely ECG, EEG, SPO2, pressure, and temperature sensors. All the sensors are placed on a patient's body to collect patient's health parameters. They are directly send to the controller, which aggregates all of those information and send it to a gateway (in this case a transceiver). The gateway processes the data and forward them wirelessly to a remote server. The server hosts the software services that save the data as well as exposes them to other authenticated parties in need. On the other hand, the monitoring software deployed on the computer accesses such information and send them to a displaying screen. When something goes wrong, for instance, in terms of read sensor values which exceeds or below a certain threshold, the monitoring

software can decide to raise an alarm in order to alert the doctor about the unusual condition of the patient. In this case, a doctor checks on the displayed data and decides to act accordingly by either shutting off the alarm, changing configuration in the systems, or fixing some issues that might be related to the sensors.

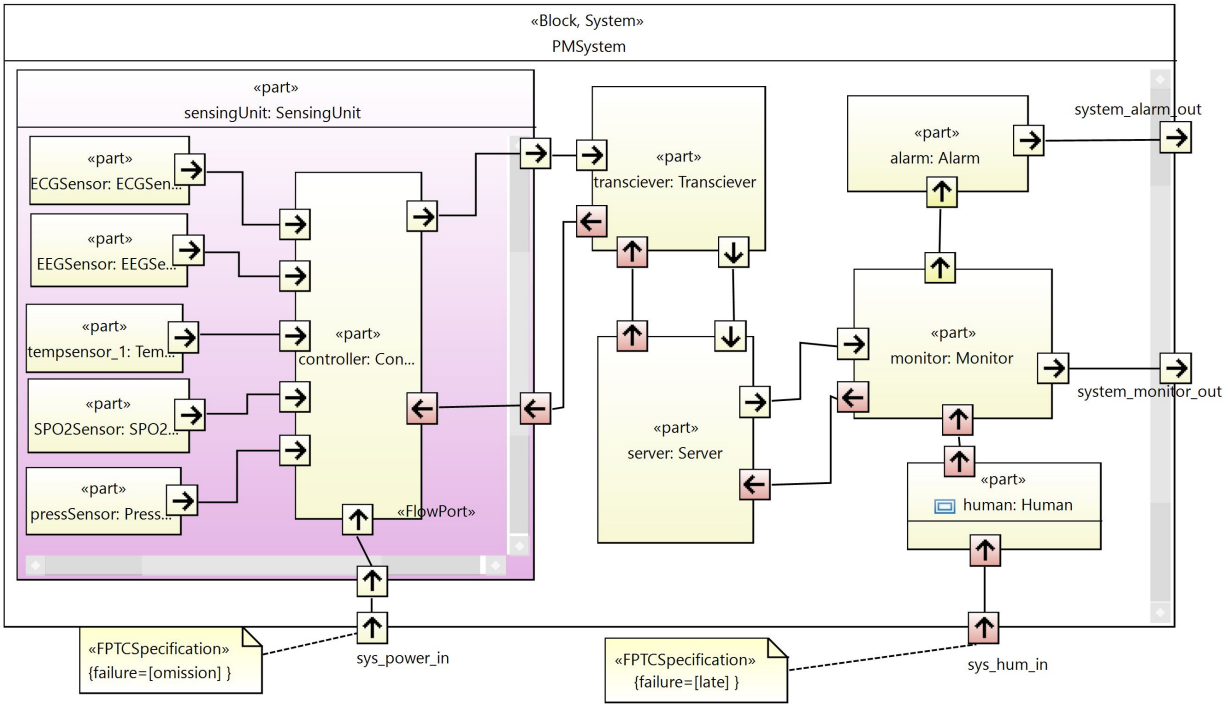


Figure 35: Patient monitoring system

To facilitate the modeling of the system failure behavior data needed for the safety analysis data, the infrastructure allows to annotate the failure behavior rules as well as the failure rates on each of the low-level simple components and these information are fully part of the model itself. As we all know, external influences can cause a system to fail. Through two system-level input ports, the presented architecture allows to simulate effects in which an external failure introduced in the system would affect the overall system functionality. For instance, the *sys_power_in* port, used to model power source outlet, was been injected with an "omission" failure which basically models the event in case there is a power outage. On the other hand, the *sys_hum_in* port is used for modeling the external influence of the doctor. In our case, a "late" failure was used to simulate an event in which a doctor reacted late due to some external factors. Finally, the system contains two output ports, namely system_monitor_out for modeling the output port from the monitor, and the system_alarm_out to model the output of the alarm system. According to the direction of the ports at the system level as well as the type of the failures that will be able to propagate to them, different FTs will be generated accordingly.

4.5.2 PMS System failure behavior

As it was previously anticipated, the above system is subjected to different kind of failures either being internally generated from the system or coming from the surrounding environment. As we described in the previous section, it is possible to model the individual components failure behavior that later get assessed in determining the failure behavior of a sub-system or an entire system. Note that we will not be focusing on software-level functional behavior but on physical failure behavior which can be even understood by non professional users. In order to understand the need for the conducted analysis, let's first discuss about different top failure scenarios that we have taken into account in defining individual components rules.

- **The alarm sub-system malfunctions by sending out a false signal**: In normal settings, this can occur when the alarm component receives a wrongly alarm notification. This is usually caused by the monitor software making a decision based on incorrect data from one of its input ports. The alarming system, on the other hand, can send false signals due to its internal failure for a variety of reasons such as poor internal configuration or simply aging.

- **The alarm subsystem has completely stopped working:** It is possible that the alarm system no longer works completely. This can be caused by several reasons; for example, alarm system being physically

disconnected, internal failure which causes a total black-out.

- **The monitor is displaying incorrect data:** As it is obvious, the main cause of this could be due to incorrect data being sent to the monitor. However, other factors, such as a faulty monitor, losing connection to the internet making it display last received data and so on. It should be noted that these are only generalized assumptions; the extensive individual study, as well as their corresponding failure rules, is shown in Table 2.

- **The monitor completely fails to display data on the screen:** This can occur due to internal and/or external monitor issues such as the monitor not being physically connected to the system power source, being unable to connect to the server, internal monitor malfunction due to aging, and so on. On the other hand, this could be caused by the monitor being properly connected but the server not receiving any data from the sensing unit.

Component	Rules	Description
ECG, EEG, Temp, SPO2 and Pressure sensor	❶ FLA:(*) → ecgsens_out.**omission**;	Sensor fails internally which make it unable to read and push any at output port
	❷ FLA:(*) → ecgsens_out.**valueCoarse**;	Sensors begin to fail as they age and provide incorrect data to the output; this can also be caused by sensor components that are not properly mounted to the patient body.
	Rules ❶ and ❷ apply to other sensors	Same as other sensors
Controller	❸ FLA:ecg_cont_in.**noFailure**,eeg_cont_in.**noFailure**→cont_trans_out.**omission**;	Controller fails completely omitting to send the data
	❹ FLA:mon_power_in.**omission**, trans_cont_in.**omission** → cont_trans_out.**omission**;	The controller fails to function due to a power outage at its input power port, no backup solution is available *(trans_cont_in port)*
	❺ FLA:ecg_cont_in.**omission**, eeg_cont_in.**omission**, press_cont_in.**omission**, spo_cont_in.**omission**, temp_cont_in.**omission** → cont_trans_out.**omission**;	All of the sensors simultaneously stop sending data, preventing the controller from sending any data to the server
	❻ FLA:ecg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	The controller receives inaccurate data from the ECG sensor and sends it to its output port
	❼ FLA:eeg_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the EEG sensor and sends it to its output port
	❽ FLA:press_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the blood pressure sensor and sends it to its output port
	❾ FLA:spo_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the SPO2 sensor and sends it to its output port
	❿ FLA:temp_cont_in.**valueCoarse** → cont_trans_out.**valueCoarse**;	controller receives inaccurate data from the temperature sensor and sends it to its output port
	⓫ FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**valueCoarse**;	The controller receives an undetected error at its from-system port, which impede sensor data transmission.
	⓬ FLA:trans_cont_in.**valueSubtle** → cont_trans_out.**omission**;	The controller receives an undetected error at its from-system port halting the data transmission process.
Transceiver	⓭ FLA:trans_in_fr_unit.**valueCoarse**→trans_out.**valueCoarse**;	The transceiver received wrong data and transmit to its output port
	⓮ FLA:trans_in_fr_unit.**noFailure**,trans_in_f_serv.**noFailure**→trans_out.**omission**;	The transceiver fails internally causing the halt of data transmission process
	⓯ FLA:trans_in_fr_unit.**omission** → trans_out.**omission**;	The transceiver receive no data and fails to transmit any data to its output port
	⓰ FLA:trans_in_f_serv.**valueSubtle** → trans_o_2_unit.**valueSubtle**;	The transceiver receive an undetected errors at its server port and forwards it back to the sensing unit
Server	⓱ FLA:server_in.**noFailure** → server_out.**omission**;	The server fails, bringing the transmission process to a halt
	⓲ FLA:server_in.**valueCoarse** → server_out.**valueCoarse**;	The server routes the incorrect data received at the input port to the output port
	⓳ FLA:server_in.**omission** → server_out.**omission**;	The server receives no data from its input port, and this error is forwarded to its output port
	⓴ FLA:serv_in_f_mon.**valueSubtle** → serv_2_trans_out.**valueSubtle**;	The server sends an undetected error from the monitor back to the transceiver's port
Monitor	㉑ FLA:monitor_in.**noFailure**→monitor_out.**omission**,mon_alarm_out.**omission**;	The monitor fails internally omitting to display the data on the screen as well as not communicating to the alarm component
	㉒ FLA:monitor_in.**omission** → monitor_out.**omission**, mon_alarm_out.**omission**;	The monitor receiving no data from the server omitting to display the data as well as not sending any communicating signal to the alarm component
	㉓ FLA:monitor_in.**valueCoarse** → monitor_out.**valueCoarse**, mon_alarm_out.**commission**;	The monitor receives inaccurate data and displays it on the screen, potentially sending a unexpected notification to the alarm component *(Commission)*
	㉔ FLA:hum_mon_in.**valueSubtle** → mon_2_serv_o.**valueSubtle**, mon_alarm_out.**commission**, monitor_out.**noFailure**;	The monitor receive an unpredicted error from the human nurse component, the failure propagates in the system in various ways with no direct effect on the data displayed on the screen before (Refer to Rules 30 and 31 for possible causes)
	㉕ FLA:hum_mon_in.**omission** → mon_2_serv_o.**valueSubtle**;	The monitor receives no engagement from the nurse intended to resolve the issue in the system, the cause of which we do not know. As a result, such a failure will go unnoticed by the system. (Refer to Rules 11 and 12 for possible effects)
Alarm	㉖ FLA:mon_alrm_in.**commission** → alarm_out.**commission**;	The alarm component received an inaccurate notification and immediately rings because it lacks any type of logical reasoning on the signal receiving other than ringing.
	㉗ FLA:mon_alrm_in.**noFailure** → alarm_out.**commission**;	The alarm starts failing due to internal failure which can make it malfunction by giving false alerts
	㉘ FLA:mon_alrm_in.**noFailure** → alarm_out.**omission**;	The alarm component fails completely which makes it unable to make any alert
	㉙ FLA:mon_alrm_in.**omission** → alarm_out.**noFailure**;	The alarm receive no data but that won't affect the internal functionality of the alarm component
Human nurse	㉚ FLA:human_in.**late** → human_out.**valueSubtle**;	The human nurse reacts very slowly in the event of a system failure, which may or may not affect the system in some way, which is why a "valueSubtle" is considered.
	㉛ FLA:human_in.**noFailure** → human_out.**omission**;	The absence of the doctor results in an omission at the output port

Table 2: PMS failure behavior table

The next step is to derive internal failure rules as well as the propagation rules to the basic components. For instance, for each sensor, two rules were defined to model two different scenarios in which a sensor can fail. A sensor can fail internally leading to a complete omission in providing the data to the output port, thus an "omission" failure will be propagated to the output port of the sensor. On another hand, a sensor can start to fail but not completely due to age. This may result in providing incorrect data to the output; this can also be caused by sensor components that are not properly placed the patient's body. In this case,

we consider that the value sent to the output port are of "valueCoarse" type. Hence the two different types of failure can be propagated to the same output port in different scenario, and they will be represented as indicated in Expression 6 and 7 respectively. We considered only the two failure conditions to apply for all of the sensors. As it can be seen from the Table 2, a detailed set of failure behavior rules and their description are represented.

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{omission} \tag{6}$$

$$FLA : (*) \rightarrow ecgsens\_out.\textbf{valueCoarse} \tag{7}$$

In CHESSIoT, to facilitate the quantitative analysis, the failure rates of the component internal failure events as well as the injected failures events have to be set separately. As we did not have the exact failure rates of the basic components, we considered the arbitrary failure rates of any component to be practically small in a range of 10^{-8} to the 10^{-7}. Figure 36 depicts the interfaces in which the internal failure and their description are set. Having the event description is practically good in order to facilitate the readability of the FT, but it is not mandatory to have for performing a qualitative analysis. When no data is provided to any of the rows, the default values are used. For instance, an unset basic event probability is assigned with a value of zero in the FT, while the unset basic event description will still follows the conventional naming of "<failure type> at <port name> in <component name>".

Probability Registration Form

Name	Probability	Failure description
ALARM: FLA:mon_alrm_in.noFailure->alarm_out.commission;	0.00000004	Alarm system start to fail increasingly due to age
ALARM: FLA:mon_alrm_in.noFailure->alarm_out.omission;	0.00000005	Alarm system fails completery or broken
HUMAN: FLA:human_in.noFailure->human_out.omission;	0.00000006	Human not present at all
PRESSSENSOR: FLA:(*)->psens_out.omission;	0.00000007	Pressure sensor fails completely or broken
PRESSSENSOR: FLA:(*)->psens_out.valueCoarse;	0.00000008	Pressure sensor starts to fails by age and provide wrong readings
SPO2SENSOR: FLA:(*)->spo2sens_out.omission;	0.00000009	SPO2Sensor fails completely or broken
SPO2SENSOR: FLA:(*)->spo2sens_out.valueCoarse;	0.000000011	SPO2Sensor starts failing due to age and provides worng readings
CONTROLLER: FLA:ecg_cont_in.noFailure,eeg_cont_in.noFailure->cont_trans_out.omission;	0.000000012	Controller fails completely internally due to unknown issues
TEMPSENSOR_1: FLA:(*)->tempsens_out.omission;	0.000000013	Temperature sensor fails completely or broken
TEMPSENSOR_1: FLA:(*)->tempsens_out.valueCoarse;	0.000000014	Temperature sensor starts failing due to age and provides worng readings
EEGSENSOR: FLA:(*)->eegsens_out.omission;	0.000000015	EEG sensor fails completely or broken
EEGSENSOR: FLA:(*)->eegsens_out.valueCoarse;	0.000000016	EEG sensor starts failing due to age and provides worng readings
ECGSENSOR: FLA:(*)->ecgsens_out.omission;	0.000000017	ECG sensor fails completely or broken
ECGSENSOR: FLA:(*)->ecgsens_out.valueCoarse;	0.000000018	ECG Sensor starts failing due to age and provides worng readings

SUBMIT CANCEL

Figure 36: PMS components failures rates set

As proved in the preceding discussion, our proposed approach is capable of satisfying all potential failure behaviors prescribed by the safety expert. As shown in Figure 35, our approach is capable of modeling the backward failure radiation pattern. For instance, a server failure will affect the monitor behavior, preventing data from being displayed on the screen. On the other hand, an erroneous command sent by the doctor's absence (for example, to fix an unmounted sensor) may eventually propagate back to the sensing unit, causing a wrong value error to be transmitted at the controller output port (valueCoarse failure) or possibly suspending the data transmission process (omission failure). Well, it is also worth noting that the ability to integrate all of their component failure rules as well as their failure rates in the same model has the potential to boost model consistency as well as the transparency in the modeling process.

4.5.3 PMS Fault tree analysis

The FT analysis begins after the CHESS-FLA transformation. The FT generation process is performed prior to running the FT analysis, in which each of the top events described in Section 4 results in its own FT. For instance, FT leading to a "omission" failure at the system monitor out port is generated to show the entire failure contribution leading to that top event. Other FT representing the remaining 3 top events are generated as well. At this stage, the generated FTs are very large as they contains every detail related to failure propagation and transformation from component to component, making it tricky to be read. Therefore, FT analysis can then be launched to automatically perform both qualitative and quantitative analysis on the model.

Figure 37 shows the analysed FT of the event "the monitor fails to display data completely on the screen". The presented FT showcases only the important events and logical gate combinations. It can be clearly anticipated that the analysed FT makes it easier to identify and trace any failure source events in its contribution to the top failure event. For instance, we can easily grasp that the monitor would display no data on the screen completely when any of the following events occur:

- Internal failure in the monitor (10^{-8} probability)
- Server is down ($2x10^{-8}$ probability)
- The transceiver (gateway) fails completely to transmit the data ($3x10^{-8}$ probability)
- A combination of events (low-left AND gate) in which there is a problem with the sensing unit power source and there is no person to fix that at the moment. ($5x10^{-15}$ probability)
- The controller of the sensing unit fails completely which halts the transmission process ($1.2x10^{-8}$ probability)
- An event in which all the the sensor does not send the data at all (This is more unlikely but possible, that's why we have a lower-middle and gate combination with $2.088x10^{-38}$ probability).
- An unknown human error occurred from the monitor side ($2x10^{-7}$ probability)

When the event of the monitor not displaying any data occurs, medical personnel can rely on such a narrow series of events to determine the cause of the event. Furthermore, medical personnel can use the probability associated with each basic event inn the list to quickly locate the source of failure, moving from the most probable basic event (highest probability) to the least probable event (lowest probability). The overall probability of this system-level failure event occurring is calculated to be 2.72x10−7, which is practically small, however this value is calculated automatically and is solely dependent on the arbitrary basic event failure rates as well as their logical combination analysis, as shown in the FT.

Other analysed FT on the event in which "monitor displaying incorrect data" and "PMS alarm sub-system alert false signal" is shown in figure 38 and 39 respectively. As it can be seen from the figure 38, the event in which the monitor will display incorrect data can be caused by any of the sensor (OR gate), as well as an unforeseen human error which transforms throughout the system and hinder the data that are being transmitted. The overall probability in which such event can occur is calculated to be about 3.39x10−7 which is higher that the event in which the monitor can stops working a all.

On the other hand, as shown in Figure 39, the same events that cause the monitor to display incorrect information can also cause the monitor to send a false signal via a failure transformation, resulting in a false alter event in the alarm system. It is also worth noting that an event like the alarm sub-system failing with a probability of 4x10−8 would also contribute to that cause. the overall probability of such event to occur is projected to be around 5.79x10−7 which is much higher that the previous two top events. Furthermore, while the "Human error" basic event appears twice in the tree, such failure passes through different channels and eventually transforms to other types throughout the system. This is practically important to understand which component of a system's error would change its nature, potentially causing a lot more damage than expected. Finally, It is worth noting that this FT does not include the top event in which the alarm sub-system to stop working completely. An FT reflecting such an event was generated and analyzed separately.

Typically, safety engineers will collaborate with system engineers to keep the safety model up to date during the development process. Maintaining coherence between system architecture and safety model can be difficult as the model gets larger and more complicated. Having a framework that can integrate modeling and analysis processes from a single place would potentially improve consistency, transparency, and minimize analysis time. Overall, the proposed analysis approach is capable of achieving that by the means of automated qualitative and quantitative calculus.

4.6 Conclusion

In this chapter, we presented CHESSIoT, a full-fledged low-code engineering environment for designing, developing, analyzing, and deploying engineering IoT systems. We explained the CHESSIoT multi-view modeling approach for designing IoT software architecture as well as generating ThingML model used

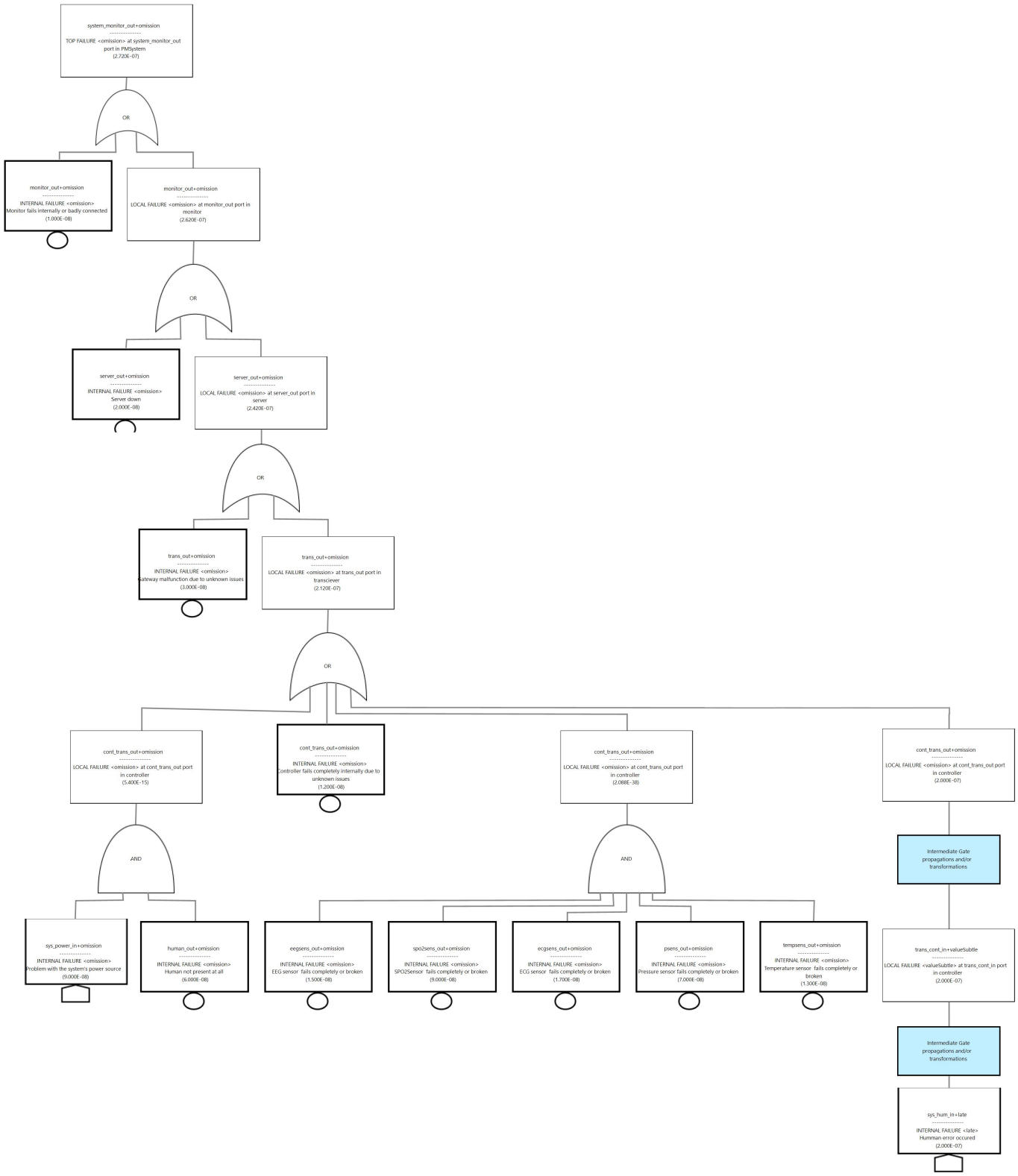


Figure 37: PMS monitor screen shown no data

for platform specif code generation purposes. In addition to that, this chapter presented CHESSIoT novel approach for system-level design and safety analysis technique for IoT systems. The proposed method combines rigorous automated analysis procedures with annotated failure behavior on components and associated failure rates to generate fault trees. The supporting tool can perform both qualitative and quantitative analysis on generated FTs.Through a simple example, we demonstrated how the proposed environment may be used to generate platform-specific code via the ThingML platform as well as its performing both qualitative and quantitative safety analysis. Finally, we briefly described how, using a series of model-to-text transformations, we effectively design and generate multiple deployment artifacts capable of autonomously deploy and monitor IoT services via agents deployed on machine nodes. Currently we are working to improve both code generation process to support more robust use-cases as well as

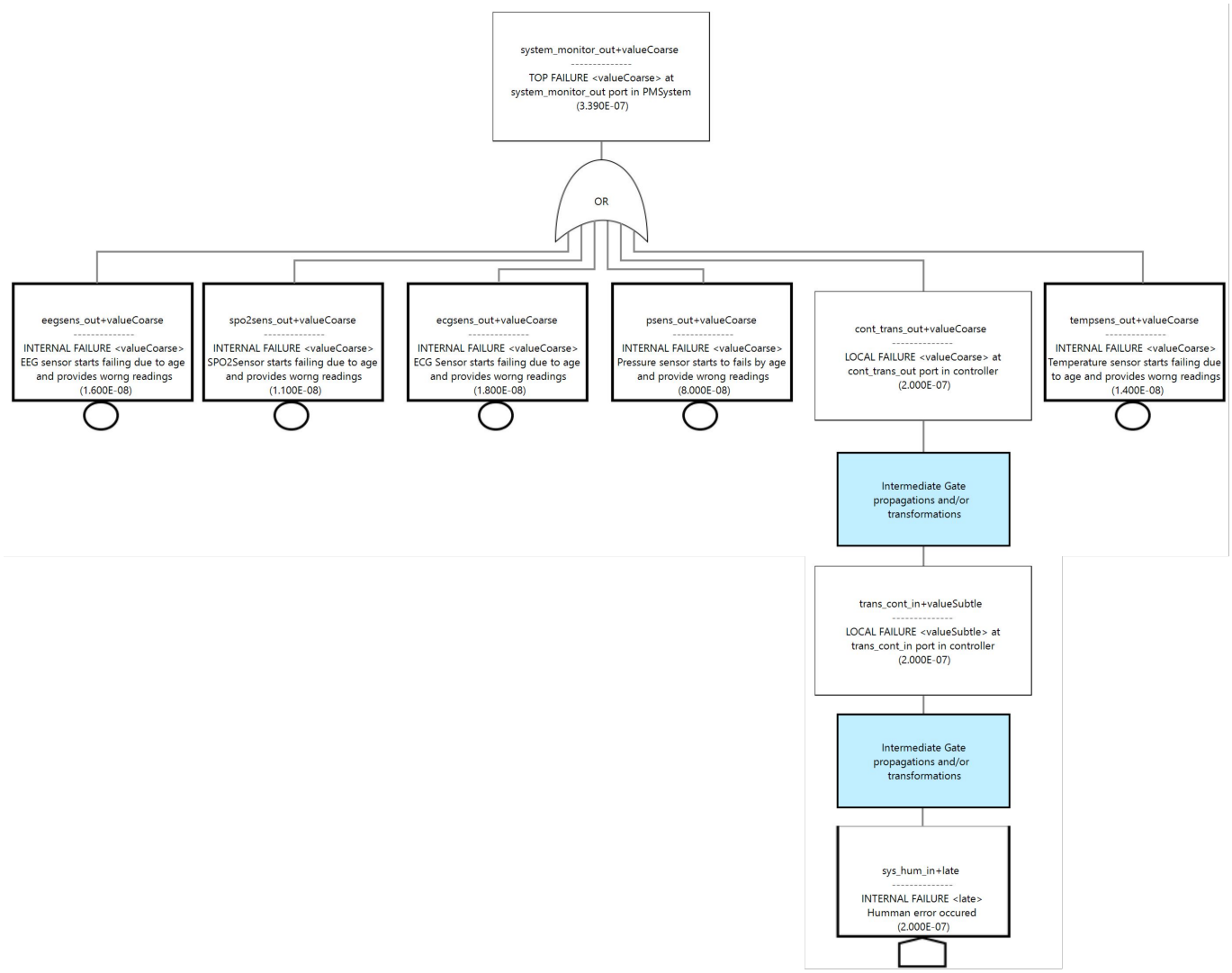


Figure 38: The monitor fails to display data completely on the screen

integrating time-based failure logic analysis into our safety analysis approach if a given failure would only affect the system for a given period of time.

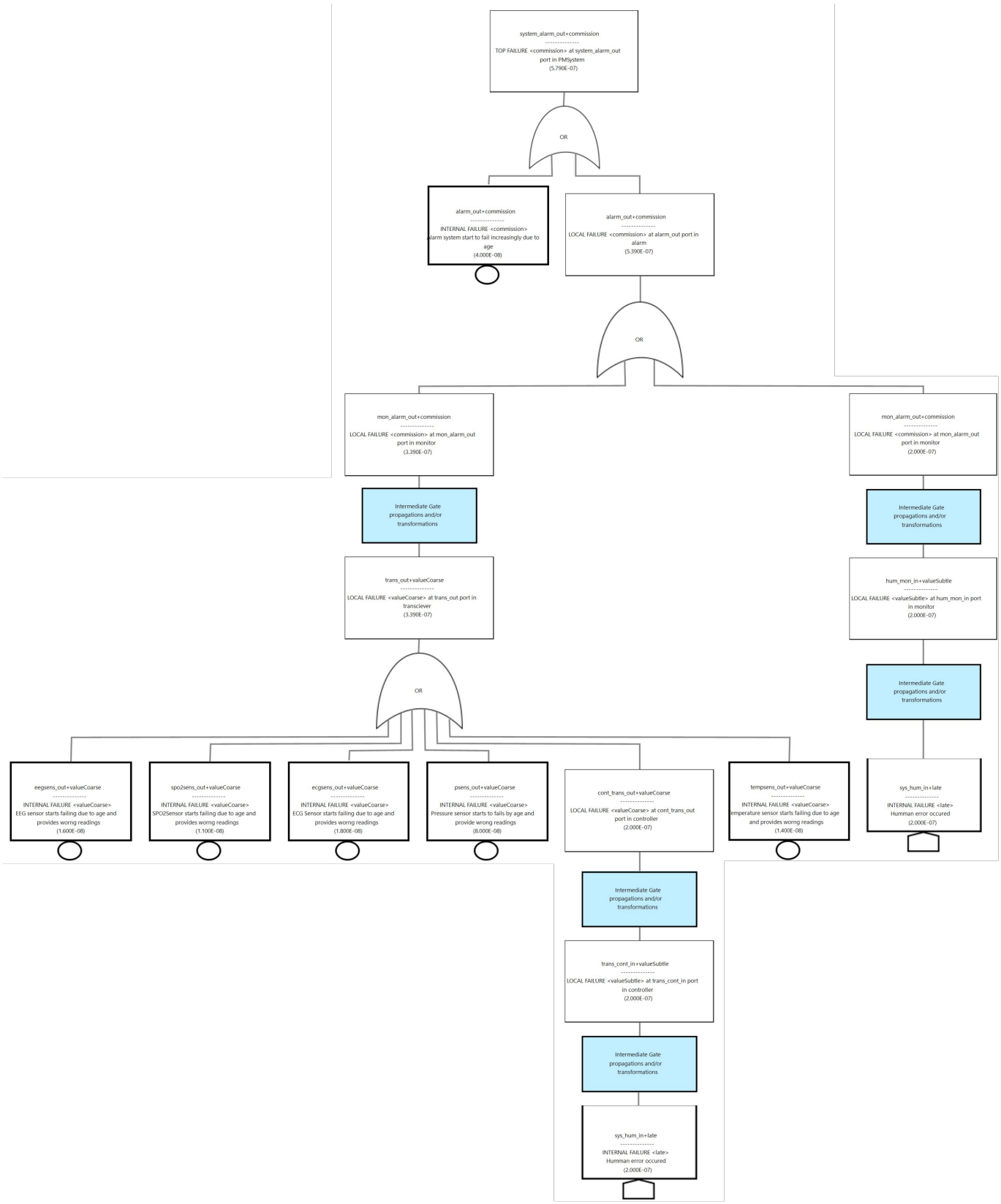


Figure 39: Alarm sub-system alert false signal

5 Integrations

This section presents the way that the domain-specific solutions presented in the previous sections, are exposed to the users of the Lowcomotive platform.

5.1 Active DSLs integration

Low-code development platforms offer interaction via dynamic graphical user interfaces, visual diagrams and declarative languages. We have described in section 2 that the different DSLs part of the Active DSLs answer this need in terms of user-friendly interactions by offering mobile applications with graphical user interfaces and visual diagrams for using these Active DSLs, and declarative languages for describing the different component of the Active DSLs.

The Active DSLs are based on EMF, using ecore language and persisted using the XMI format. Hence, they can be stored on the common repository of the Lowcomotive platform. In future work, the current DSL-comet server could be replaced by the Lowcomotive platform or communicate with the Lowcomotive storage instead of the current database.

Finally, to cope with the idea of mobility, we plan to integrate the edition of the models on the editor of the Lowcomotive platform which is based on EMF.cloud and by extension, is compatible with Xtext.

5.2 The Panoptes User Interface

To better enhance user-friendliness, Panoptes offers the web-based user interface shown in Figure 40. The user interface provides a text editor through which data scientists can fully specify the behaviour of the Panoptes system. The editor provides features to make its usage easier such as syntax and error highlighting. In case of erroneous input, the user it shown a message that helps them fix their mistake. Finally, after the user has finished editing their specification, they can save the changes and the system will automatically update its behaviour.

In terms of integration with the wider Lowcomotive platform, Panoptes is equipped as follows. Firstly, it uses standardized EMF technologies as its foundation. This allows the models developed by the users to be persisted in the repository developed by ESR 6 within the Lowcomote project. Additionally, the system's architecture is modular. This enables the substitution of certain components to enable even further integration with the rest of the Lowcomotive platform. The web editor, for example, could be replaced by an EMF.Cloud based editor that supports every DSL that has been developed as part of Lowcomote.

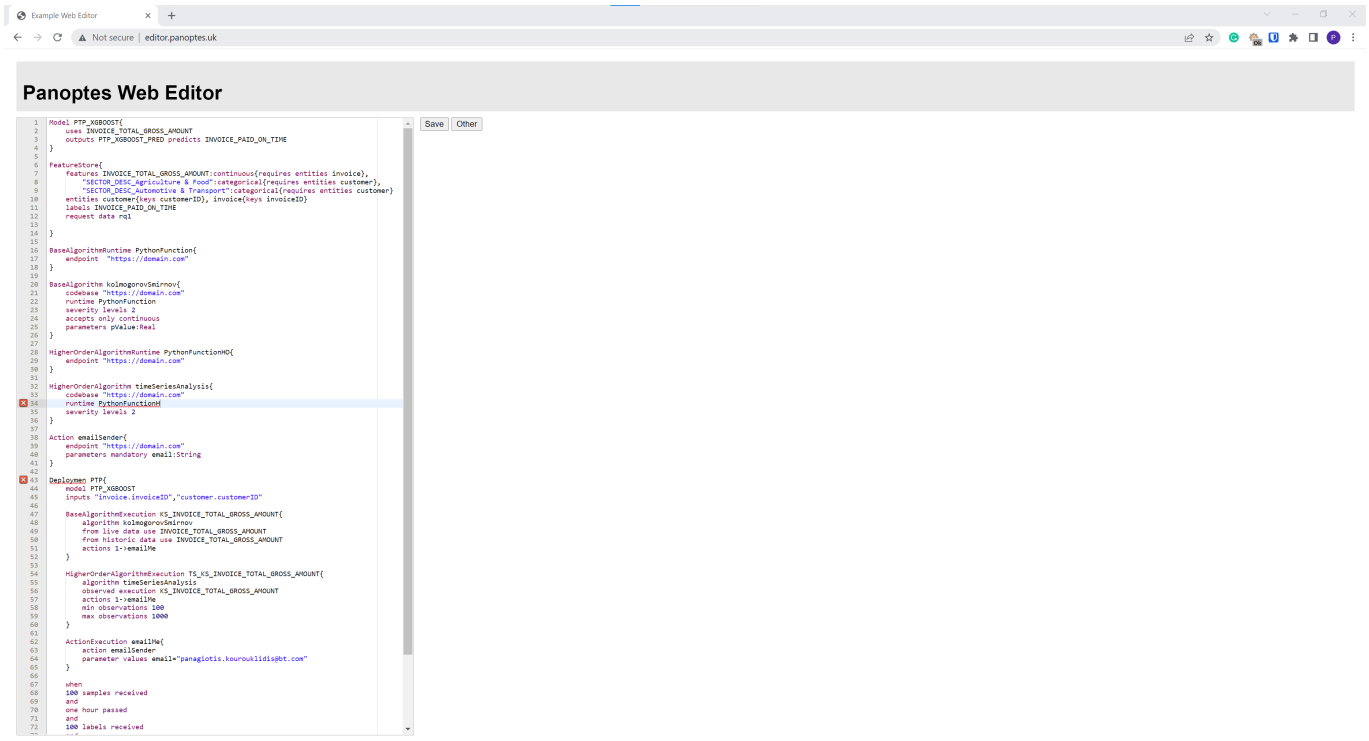


Figure 40: Panoptes User Interface

5.3 CHESSIoT supported integration

The CHESSIoT environment is built on top of the Eclipse Papyrus tool, an open source Model-Based Engineering environment for industrial application. This, in turn, allows for easy extensibility and customization as needed by the client. As illustrated in Figure 41, the user in the CHESSIoT environment can persist both the modeled and generated artifacts via a dedicated channel to facilitate direct connection with the Lowcomotive repository. Furthermore, the present CHESSIoT modeling infrastructure supports EMF models in .XMI and .Ecore formats, which are well supported and interpreted by the Lowcomotive repository, queries/transformations technologies developed by other partners in the Lowocmote project. Finally, CHESSIoT models can be well supported by EMF.cloud if it is chosen as a front end editing platform.

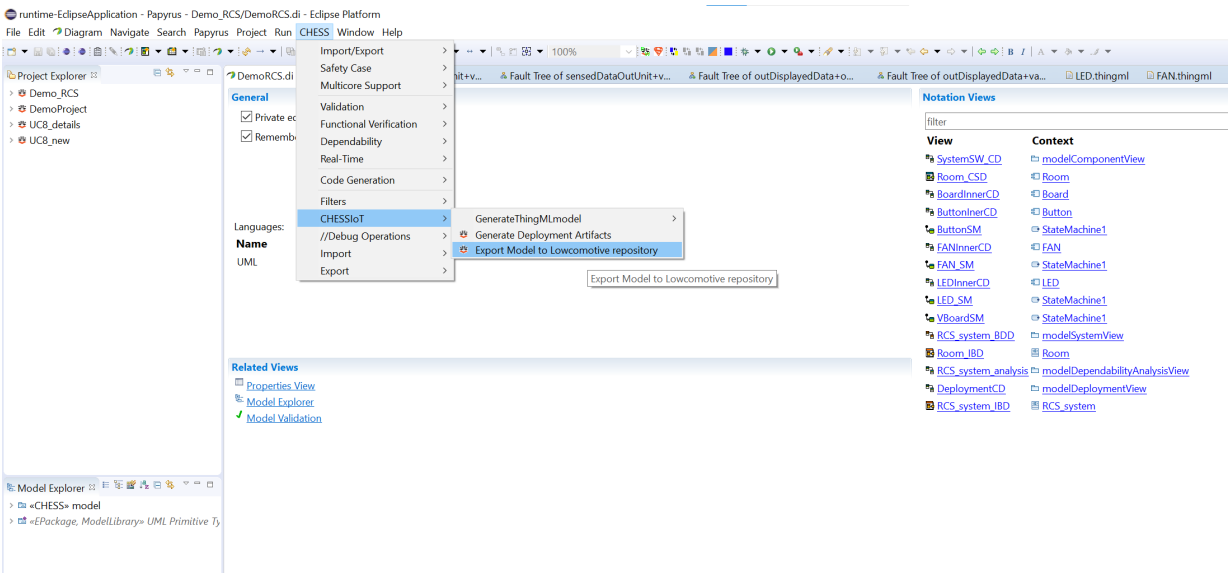


Figure 41: CHESSIoT export to Lowcomote repo

6 Conclusion

In this deliverable we have reported on the progress the Lowcomote project has made with respect to Low-Code Development Platforms (LCDP) in a variety of application domains. Within Lowcomote, a variety of domains is being studied within the context of LDCP, ranging from data science and mobile application to IoT and Smart Cities, and each of these domains offers unique challenges and development processes for its citizen developers. It is within the remit of this project to determine within this constellation of domains where commonalities can be found and how domain specific requirements and workflows can be supported by a uniform LDCP. The progress reported in this deliverable focusses around the areas of collaborative mobile apps, IoT and Smart Cities, and data science, and within these domains the first concepts and domain specific language elements have been identified and described based on in-depth analysis of previous work, case studies and prototype implementations. The next steps will consist of analysing the prototypes for commonalities and variabilities and defining the uniform architecture for the Lowcomote LDCP that makes these common elements available to serve as a foundation for specialised development platforms. This includes for example uniformised code generation capabilities and DSL support. The variability analysis of these prototypes will reveal where the base platform needs to support flexible integration and extensions that allow supporting domain specific actions, workflows and artefacts. The realisation of this will be covered in subsequent project deliverables.

References

[1] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[2] Léa Brunschwig, Esther Guerra, and Juan de Lara. Modelling on mobile devices. *Softw. Syst. Model.*, 21(1):179–205, 2022.

[3] Léa Brunschwig, Esther Guerra, and Juan de Lara. Towards access control for collaborative modelling apps. In Esther Guerra and Ludovico Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 67:1–67:10. ACM, 2020.

[4] Léa Brunschwig, Rubén Campos-López, Esther Guerra, and Juan de Lara. Towards domain-specific modelling environments based on augmented reality. In *43rd IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2021, Madrid, Spain, May 25-28, 2021*, pages 56–60. IEEE, 2021.

[5] Ronald T. Azuma. A survey of augmented reality. *Presence Teleoperators Virtual Environ.*, 6(4):355–385, 1997.

[6] Diego Vaquero-Melchor, Javier Palomares, Esther Guerra, and Juan de Lara. Active domain-specific languages: Making every mobile user a modeller. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 75–82. IEEE Comp. Soc., 2017.

[7] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 68–77. BCS Learning and Development Ltd., 2008.

[8] Dustin Wüest, Norbert Seyff, and Martin Glinz. Flexisketch: a lightweight sketching and metamodeling approach for end-users. *Software and Systems Modeling*, 18(2):1513–1541, 2019.

[9] Ronny Seiger, Maria Gohlke, and Uwe Aßmann. Augmented reality-based process modelling for the internet of things with holoflows. In *20th International Conference on Enterprise, Business-Process and Information Systems Modeling (BPMDS/EMMSAD@CAiSE)*, volume 352 of *LNBIP*, pages 115–129. Springer, 2019.

[10] S. Pérez-Soler, E. Guerra, and J. de Lara. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software*, 35(6):48–54, 2018.

[11] Mxdp. Accessed: 2020-11.

[12] ARKit. https://developer.apple.com/augmented-reality/, 2020.

[13] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2503–2511, 2015.

[14] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, pages 1–10, 2022.

[15] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[16] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[17] Amos Storkey. When training and test sets are different: characterizing learning transfer. *Dataset shift in machine learning*, 30:3–28, 2009.

[18] Jeffrey C. Schlimmer and Richard H. Granger. Incremental learning from noisy data. *Mach. Learn.*, 1(3):317–354, 1986.

[19] B Schölkopf, D Janzing, J Peters, E Sgouritsa, K Zhang, and J Mooij. On causal and anticausal learning. In *29th International Conference on Machine Learning (ICML 2012)*, pages 1255–1262. International Machine Learning Society, 2012.

[20] Marcos Salganicoff. Tolerating concept and sampling shift in lazy learning using prediction error context switching. *Artif. Intell. Rev.*, 11(1-5):133–155, 1997.

[21] Joaquin Quiñonero-Candela, Masashi Sugiyama, Neil D Lawrence, and Anton Schwaighofer. *Dataset shift in machine learning*. Mit Press, 2009.

[22] Jose G. Moreno-Torres, Troy Raeder, Rocío Alaíz-Rodríguez, Nitesh V. Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern Recognit.*, 45(1):521–530, 2012.

[23] Meelis Kull and Peter Flach. Patterns of dataset shift. In *First International Workshop on Learning over Multiple Contexts (LMCE) at ECML-PKDD*, 2014.

[24] Kyosuke Nishida and Koichiro Yamauchi. Detecting concept drift using statistical testing. In Vincent Corruble, Masayuki Takeda, and Einoshin Suzuki, editors, *Discovery Science, 10th International Conference, DS 2007, Sendai, Japan, October 1-4, 2007, Proceedings*, volume 4755 of *Lecture Notes in Computer Science*, pages 264–269. Springer, 2007.

[25] Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Trans. Neural Networks*, 22(10):1517–1531, 2011.

[26] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: a case study. In Helen Sharp and Mike Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 291–300. IEEE / ACM, 2019.

[27] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[28] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas. On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41(4):5–15, 2018.

[29] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 620–629. IEEE Computer Society, 2018.

[30] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data lifecycle challenges in production machine learning: A survey. *SIGMOD Rec.*, 47(2):17–28, 2018.

[31] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[32] John L Hodges. The significance probability of the smirnov two-sample test. *Arkiv för Matematik*, 3(5):469–486, 1958.

[33] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[34] Pip documentation. Accessed: 2022-08-05.

[35] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.

[36] Panagiotis Kourouklidis, Dimitris Kolovos, Joost Noppen, and Nicholas Matragkas. A model-driven engineering approach for monitoring machine learning models. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 160–164, 2021.

[37] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218. Springer, 2009.

[38] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[39] Taylor B Arnold and John W Emerson. Nonparametric goodness-of-fit tests for discrete null distributions. *R Journal*, 3(2), 2011.

[40] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[41] Sagemaker documentation. Accessed: 2022-08-05.
[42] Azure ml documentation. Accessed: 2022-08-05.
[43] Vertex ai documentation. Accessed: 2022-08-05.
[44] Vertex ai supported algorithms. Accessed: 2022-08-05.
[45] Azure ml supported algorithms. Accessed: 2022-08-05.
[46] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. Low-code engineering for internet of things: A state of research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, New York, NY, USA, 2020. Association for Computing Machinery.
[47] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178, 2020.
[48] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the next generation of experts in scalable low-code engineering platforms. *1st Junior Researcher Community Event*, page 67–76, 2019.
[49] Alberto Debiasi, Felicien Ihirwe, Pierluigi Pierini, Silvia Mazzini, and Stefano Tonetta. Model-based analysis support for dependable complex systems in chess. In *MODELSWARD,*, pages 262–269. INSTICC, SciTePress, 2021.
[50] Barbara Gallina and et al. A model-driven dependability analysis method for component-based architectures. In *38th Euro. Conf. SEAA*, pages 233–240, 2012.
[51] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. Thingml: A language and code generation framework for heterogeneous targets. MODELS '16, page 125–135, New York, NY, USA, 2016. Association for Computing Machinery.
[52] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. Towards a modeling and analysis environment for industrial iot systems. In *STAF Workshops*, pages 90–104, 2021.
[53] Mohammad Sharaf, Mai Abusair, Rami Eleiwi, Yara Shana'a, Ithar Saleh, and Henry Muccini. Modeling and code generation framework for iot. In *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0*, pages 99–115, 2019.
[54] Nicolas Ferry and et al. Development and operation of trustworthy smart iot systems: The enact framework. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 121–138, 2020.
[55] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. Cypriot: framework for modeling and controlling network-based iot applications. In *In 34th ACM/SIGAPP Symposium on Applied Computing*, pages 832–841, 2019.
[56] Bashar Alshboul, Dorina C. Petriu, Bashar Alshboul, and Dorina C. Petriu. Automatic derivation of fault tree models from sysml models for safety analysis. *Journal of Software Engineering and Applications*, 11:204–222, 5 2018.
[57] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
[58] He Ren, Xi Chen, and Yong Chen. Chapter 6 - fault tree analysis for composite structural damage. In He Ren, Xi Chen, and Yong Chen, editors, *Reliability Based Aircraft Maintenance Optimization and Applications*, Aerospace Engineering, pages 115–131. Academic Press, 2017.
[59] Stefan Markulik, Marek Šolc, Jozef Petrík, Michaela Balážiková, Peter Blaško, Juraj Kliment, and Martin Bezák. Application of fta analysis for calculation of the probability of the failure of the pressure leaching process. *Applied Sciences*, 11(15), 2021.
[60] A.S. Cheliyan and S.K. Bhattacharyya. Fuzzy fault tree analysis of oil and gas leakage in subsea production systems. *Journal of Ocean Engineering and Science*, 3(1):38–48, 2018.
[61] Lorin Hochstein and Rene Moser. *Ansible: Up and Running Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, Inc., 2nd edition, 2017.
[62] P. Varady, Z. Benyo, and B. Benyo. An open architecture patient monitoring system using standard technologies. *IEEE Transactions on Information Technology in Biomedicine*, 6(1):95–98, 2002.
[63] Rajesh Kannan Megalingam, Divya M. Kaimal, and Maneesha V. Ramesh. Efficient patient monitoring for multiple patients using wsn. In *2012 International Conference on Advances in Mobile Network, Communication and Its Applications*, pages 87–90, 2012.