Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices

Arvid Butting
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Jörg Christian Kirchhof*
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Anno Kleiss
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Judith Michael
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Radoslav Orlov
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Germany
www.se-rwth.de

Abstract

Internet of Things (IoT) devices and the software they execute are often strongly coupled with vendors preinstalling their software at the factory. Future IoT applications are expected to be distributed via app stores. A strong coupling between hard- and software hinders the rise of such app stores. Existing model-driven approaches for developing IoT applications focus largely on the behavior specification and message exchange but generate code that targets a specific set of devices. By raising the level of abstraction, models can be utilized to decouple hard- and software and adapt to various infrastructures. We present a concept for a model-driven app store that decouples hardware and software development of product lines of IoT applications.

CCS Concepts: • **Computer systems organization** → *Embedded and cyber-physical systems*; • **Software and its engineering** → *Architecture description languages.*

Keywords: Internet of Things, Model-Driven Engineering, Low-Code, App Store, Architecture Description Language

ACM Reference Format:
Arvid Butting, Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Radoslav Orlov, and Bernhard Rumpe. 2022. Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564719.3568689

*Corresponding / main author

1 Introduction

Today, IoT devices and their software are often strongly intertwined: Manufacturers already install the software for IoT devices in the production process, tying customers to their proprietary solutions. This is a massive disadvantage for users, as they are limited to the hardware manufacturers' software and prevented from installing new software. Ultimately, it leads to IoT devices becoming e-waste [12] once the developers stop supporting their software, turn off necessary cloud services, or actively disable the devices. Up to now, products from major vendors still do not *talk* to each other, probably due to competitive relations [23]. Moreover, building IoT solutions is complex due to the highly fragmented market of IoT building blocks, such as IoT hardware and communication protocols [13, 35].

For these reasons, we need an IoT app store that provides a common ground for the variety of software solutions for IoT devices. Models are suitable for this purpose because they allow for abstraction [32] and representation of product lines in variability models [34]. If IoT device customers are to use such models, we can provide low-code editors that can handle the different technical backgrounds of the customers. Recent research about IoT app stores [5, 10, 29] does not consider model-driven approaches.

Thus, this paper makes the following contributions:

- A concept for an IoT app store for distributing product lines of IoT applications based on model-driven development
- A development methodology for decoupling device and software development that integrates handwritten hardware drivers with generated IoT applications

We apply model-driven techniques to the already existing idea of IoT app stores (*e.g.*, [10]) to address the specific challenges of the IoT domain. Our main concept is modeling IoT applications as product lines and configuring their instantiations via an app store just before the initial deployment of an instance of an application. Product lines of IoT applications need to manage a higher degree of heterogeneity than, for example, smartphones, because IoT devices can be very different from each other. In particular, the combination of devices used can (and likely does) vary from user to user. Through our combination of object constraint language (OCL) (for modeling the applications' components' requirements), object diagrams (for hardware specification) and class diagrams (for standardization), we enable hardware and software developers to be decoupled from each other via a common standard provided by the IoT app store. We outline how models (class diagrams) could be used as first-class citizens in a standardization of the currently fragmented IoT market. Using this model-driven approach, we open up the possibility for future work to leverage the wide range of research on Unified Modeling Language (UML)-based modeling also in the IoT domain.

The paper is structured as following: The next section presents preliminaries of the used model-driven technologies and Sec. 3 requirements for an IoT app store. Sec. 4 describes the IoT app store concept. In Sec. 7, we describe how the code generated from (architecture) models is linked to the mostly handwritten code of hardware drivers. Sec. 6 presents an automated process to check if hardware exists that matches the specifications made for the software using a Prolog generator. Sec. 5 shows how feature models and their tagging of software architectures can be used to define configurations in the IoT app store. In Sec. 8, we show the solution in a smart home application case study and discuss it in the following section. Sec. 10 shows related work and the last section concludes.

2 Preliminaries

This section introduces the MontiThings architecture and its accompanying ecosystem for deploying IoT applications. Moreover, we present information on tagging languages.

2.1 MontiThings

MontiThings [27] is a component and connector (C&C) architecture description language (ADL) for modelling IoT applications. MontiThings applications consist of components that can send and receive messages via ports. The ports of components can be connected to specify more complex behavior. If a component does not contain and connect subcomponents to define its behavior, it can define its behavior using a Java-like embedded language, statecharts, or handwritten C++ code. Fig. 1 gives an example of modelling a lawn watering application using MontiThings' graphical syntax. Note that this graphical syntax that we use throughout this paper is only a visualization of the textual syntax that MontiThings uses for specifying models.

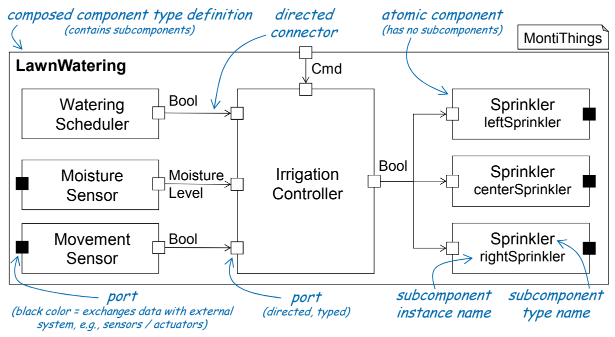


Figure 1. Exemplary lawn watering application modeled using MontiThings (adapted from [25]).

MontiThings models can be generated to C++ code. Additionally, MontiThings generates the necessary scripts to compile the generated code and package it as (Docker) container images. MontiThings, unlike many other IoT ADLs, also has a feature-rich ecosystem to deploy the generated applications to IoT devices [24]. In particular, it is possible for the user to specify rules that impose requirements on the deployment. These rules mainly relate to the locations where the IoT devices that are to run the software are placed. For example, it is possible to require that a certain software be executed in a certain room. MontiThings internally generates Prolog code from these rules, which is evaluated to determine which devices should run which software.

2.2 Tagging Languages

Tagging [20] is a mechanism for enriching a model with additional information. As opposed to annotations of a model, such as via stereotypes [22] in UML, the additional information is not contained in the model itself but in a separate artifact, the *tag model.* A tag model contains tags that refer to an element of the original model and add additional information to it. The advantage of this is that tagging does not "pollute" the original model with the additional information, which is beneficial if a model is used for different purposes or if different alternative tagging model exist for a given original model.

To restrict and guide possible tags, the tag model conforms to a *tag schema* that defines, which language elements of the language that the original model conforms to can be tagged and what they can be tagged with.

For example, to tag features in feature diagrams with class attributes of a class diagram language, the tag schema has to define, which language elements of the feature diagram language are allowed to be tagged. In this example, features are allowed but, *e.g.*, cross-tree constraints are not allowed to be tagged. Furthermore, the tag schema defines, what features can be tagged with. In this example, features should be tagged with class attributes and, hence, the tag schema language has to include syntax for defining the desired kinds of attributes.

To ensure consistency between the tag model, the tag schema, and the original model, the languages for these models have to be integrated. Through this, suitable checks can be implemented, *e.g.,* to assure that the tags of a tag model refer to existing elements of the tagged model only and that these are tagged with valid additional information as indicated in the tag schema.

3 Requirements & Assumptions

Our concept for an IoT app store is based on the following requirements:

(R1) The app store shall enable developers to distribute product lines of applications. The device infrastructures of different app store customers can differ greatly. On the one hand, customers can choose different devices on an open market, and on the other hand, the environments in which the devices are used can differ greatly. Consequently, IoT applications that are distributed via an app store and do not enforce a specific device infrastructure of a specific manufacturer, as is common today, must be designed as software product lines. An IoT app store must, therefore, offer applications a possibility to provide different features depending on the device infrastructure (or user preference) to an even greater extent than classic smartphone app stores.

(R2) The app store shall enable users to deploy features of an IoT application without needing to understand the technical details of the architecture. Customers, both commercial and consumer, of IoT applications rarely buy a complete system at once. Instead, often only some of the existing "things" are initially replaced or expanded by IoT devices. Such partial provisioning may prevent an application from providing all features that could be provided if all available devices (types) were provisioned. If users should be able to deploy an IoT application regardless missing devices, they must be able to deploy a subset of features. Since users do not know the details of the software, this selection process needs to abstract from the development artifacts.

(R3) IoT applications distributed via the app store shall be deployable to different IoT devices as long as the devices provide the necessary hardware. An app store is supposed to offer its customers the possibility to install different applications. Since IoT infrastructures can be very heterogeneous [17], and different customers can be expected to own different devices, the app store must be able to deploy software to wide range of devices. In particular, this requirement also enables devices to be repurposed instead of becoming e-waste [12], for example if a software vendor discontinues software support. If vendors stop supporting their software without such a requirement, IoT devices today can become electronic waste in extreme cases. Therefore, we demand that the App Store makes the decision whether a software can be deployed (from a purely technical point of view) dependent on the hardware requirements of the app. Whether a user wants to deploy an IoT application to all devices that would technically support it is an independent question.

In addition to these requirements, we make the following assumptions:

(A1) We assume that IoT devices are capable of executing (Docker) containers. Already today, many IoT applications are built on container technologies. Examples include Microsoft Azure IoT Edge [2] and Balena [3]. The latter also offers an IoT-focused lightweight container engine called balenaEngine. According to the 2022 Eclipse IoT & Edge Developer Survey [14], containers are the most popular edge computing artifact (used by 49 % of respondents), with Docker being the most commonly used container orchestration technology (used by 43 % of respondents). Future IoT applications are expected to rely extensively on container technologies [33]. From a technology perspective, containers greatly simplify deployment because they offer a uniform method of downloading and executing software. In this paper we assume that the IoT devices support a (Docker-compatible) container engine to simplify prototype development. With the corresponding engineering effort, it would of course also be possible to implement support for devices without a container engine, deploying, *e.g.,* native binaries instead of containers.

(A2) We assume that the app store is aware of the types of hardware used by the IoT applications. In order to be able to decide whether an (part of an) IoT application is compatible with a specific IoT device, it must be possible to decide whether a hardware component (and its driver) fulfills the requirements of the application to be deployed. In order to create a common understanding between application developers and device developers, we assume that the app store is aware of the possible hardware components. This enables it to impose a common standard on developers on both sides to describe this hardware. In the same way, smartphone app stores use a common understanding of different components such as the camera [1].

With these requirements and assumptions in mind, we will now discuss our concept for an IoT app store.

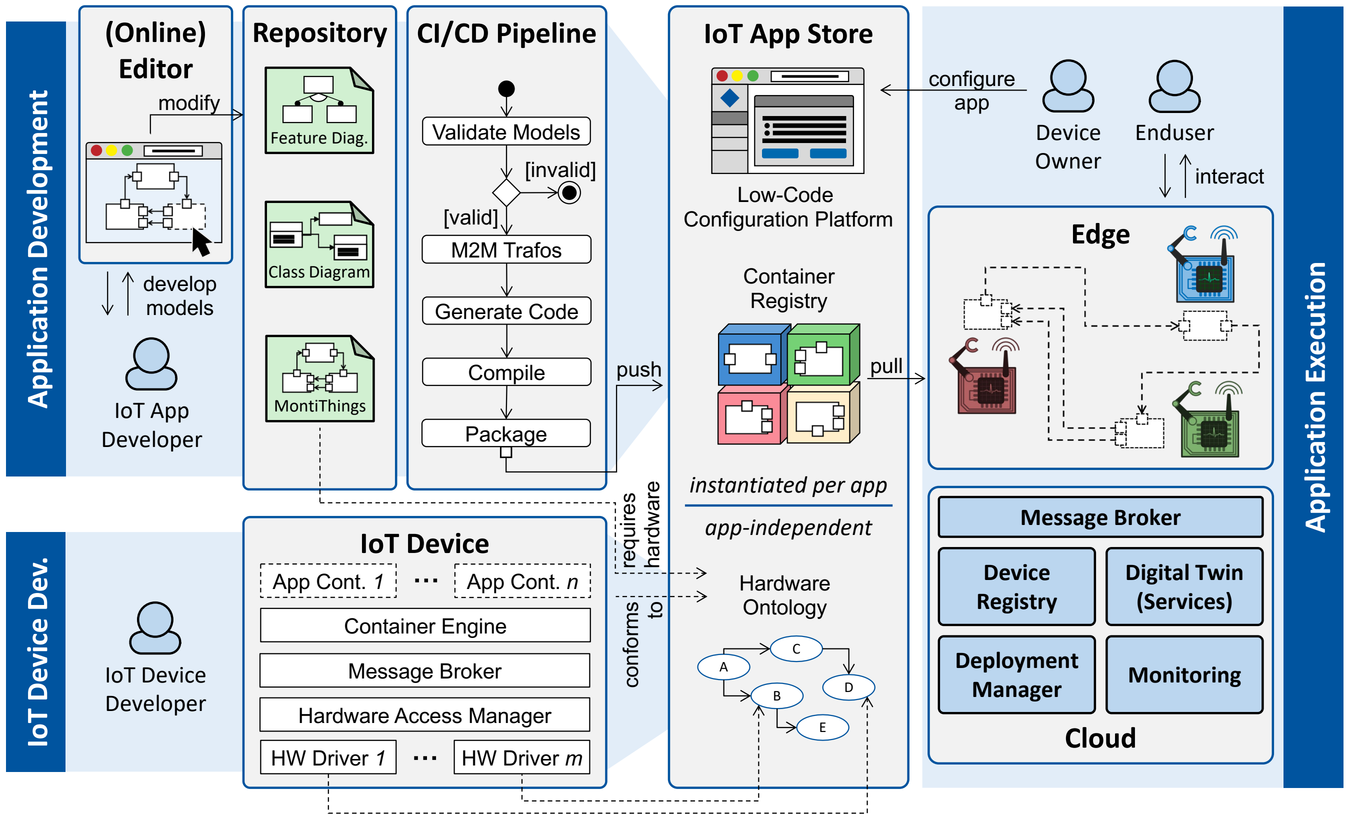


Figure 2. Concept for a model-driven app store. The app store allows device owners to configure app features modeled in a feature diagram using a webapp. The app store provides a common understanding of hardware in form of an ontology. This decouples hardware and software development.

4 IoT App Store Concept

App stores offer their users the possibility to equip their devices with various applications. The concept of app stores is already widely used today for smartphone and desktop applications. App stores must be able to handle a certain variability of the devices of the app store users, *e.g.*, different sensors (not) available in the smartphone or different processor architectures. Compared to the highly fragmented IoT market, however, the problem of variability is still comparatively manageable for smartphone and desktop applications. While smartphone applications can of course differ greatly from one another, smartphones built on common platforms, *i.e.*, Apple's iOS and Google's Android, even if they naturally differ in details such as the camera or display quality. In contrast, there are virtually no common standards for IoT applications. This is also due to the inherently higher variance of IoT use cases. While smartphone apps are essentially GUI applications with touch input, IoT applications lack this commonality and, instead, can range from lighting control in a smart home to intelligent plant irrigation in the agricultural sector to parking space monitoring. These drastically different use cases impose different requirements on available sensor and actuator hardware, power supply, network availability, resilience, and the like. Even type-similar hardware can differ greatly from each other: While a smartphone light may be slightly brighter or dimmer on one model than another, it remains a flashlight in principle. In IoT, a lamp can be a small LED or a construction spotlight.

While models are good at abstracting from technical details, this abstraction can become a problem when it inadvertently creates underspecification. When developers create an application, they often have a specific set of hardware in mind for which their application is intended. For example, someone developing a light-based alarm clock will probably think of it as a small lamp or perhaps ceiling lamp in a smart home. However, if the application is unexpectedly deployed on a car, this lamp could (at least in theory) also be a headlight that could create a safety hazard on the road if actuated in an uncontrolled manner. For this, the app store needs to create a common understanding of what properties hardware components have. In our concept, that is done by a hardware ontology that classifies hardware components (A2). In accordance with the hardware ontology, device developers specify their devices and IoT application developers specify

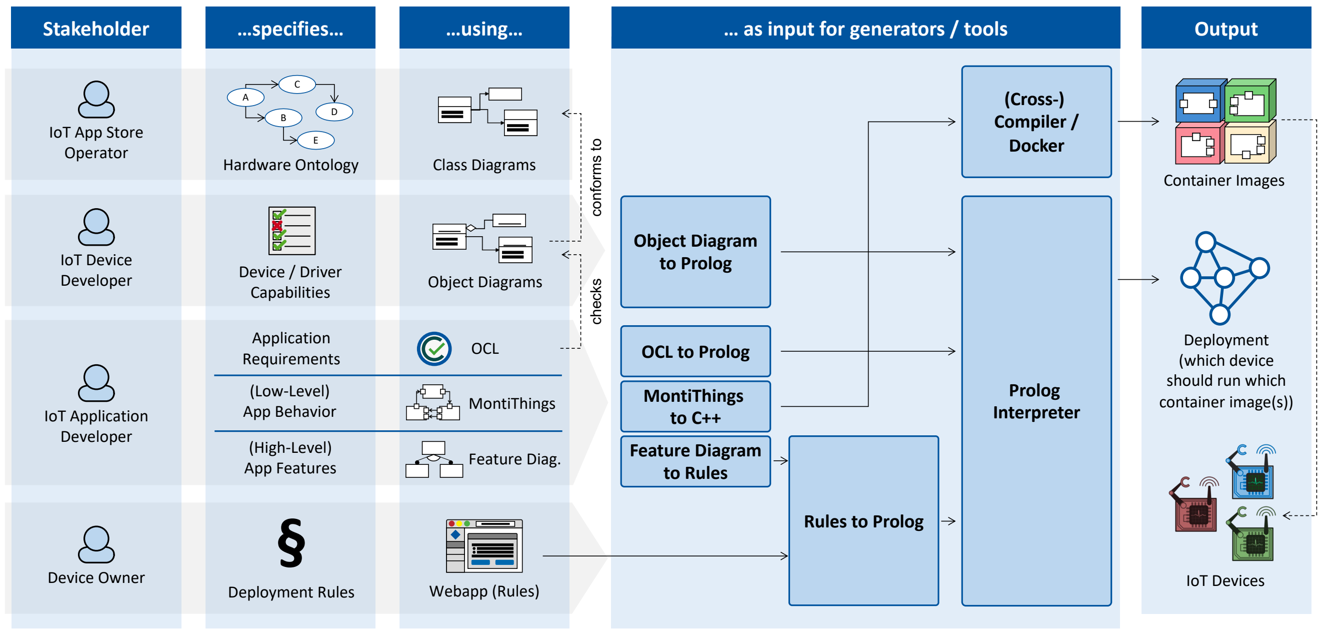


Figure 3. Relationship between stakeholders, their intents, the models used by the respective stakeholders, and the (code) generators and tools that process the models.

their requirements. Thereby, it is possible to check whether a particular application is compatible with a device that has certain components.

We envision that IoT devices would be certified in commercial implementations of such an app store in a similar way as smart home devices already prove their compatibility with different ecosystems (*e.g.,* Apple HomeKit or Amazon Alexa) today. For this purpose, it provides, in addition to an ontology of the hardware, the necessary application programming interfaces (APIs) with which an application can access a hardware component. In the context of a model-driven development process, such common APIs naturally also help to automate such hardware accesses through a code generator.

In addition to the differences between different IoT applications, different instances of the same IoT application can also be very different from each other. The reason for this is that the device infrastructures and the environments in which the devices are deployed can differ from case to case. In particular, it cannot be assumed that all customers own all devices that are offered and compatible with the application. In some cases, for example, to minimize their financial risk, customers may want to deliberately equip only part of their environment with IoT devices initially and expand the system later when the desired applications have proven themselves. As a result, IoT app stores should enable applications to be designed as product lines **(R1)**.

From these two aspects, the decoupling of hardware and software and the product line nature of the applications, we derive a model-driven concept for an IoT app store. Fig. 2 gives an overview of our app store concept and how it integrates with the different development processes and its stakeholders. Additionally, Fig. 3 shows which stakeholders use which models for which purpose and how these models are processed to deploy an IoT application. In the remainder of this section, we describe the individual elements of the concept. The following sections then detail how the models are processed.

Application Development. Application development is based on a model-driven approach. Here, IoT application developers interact with a (possibly low-code) editor to create and manipulate models. In our prototype, we used GitPod[1] for this purpose, as it already integrated with major repository platforms such as GitHub and GitLab. Essentially, we rely on three types of models here: Feature diagrams specify how an application is to be understood as a product line **(R1)**. Class diagrams define the data structures that the system uses. MontiThings architecture diagrams describe the business logic and data flows. As per MontiThings' normal deployment process [27], these models are versioned in an online repository, attached to a continuous integration (CI)/continuous deployment (CD) pipeline that is responsible for checking the models for correctness, transforming them if necessary, generating general-purpose language (GPL) code from the models, and compiling and packaging the generated code for deployment as container images **(A1)**. The container images are pushed to a container registry provided by the

[1]GitPod product website. [Online]. Available: https://gitpod.io/. Last accessed: 02.05.2022

app store. The model transformations can be used, for example, to automatically add additional functionalities such as monitoring [25] or digital twins [26]. A tagging language can be used to assign hardware requirements to MontiThings components during development **(R3)**. These hardware requirements are based on the ontology of the app store **(A2)**. Furthermore, feature diagrams can be used to group components of the architecture into high-level features that are understandable to endusers **(R2)**. The details of the feature diagram integration are described in Sec. 5.

IoT App Store. The IoT app store provides the hardware ontology in form of a class diagram **(A2)**. In this class diagram, the individual types of hardware components are represented by classes. Similar hardware can be grouped using inheritance, *e.g.,* "each distance sensor is a sensor" or "each VL53L0X Time-of-Flight sensor is a distance sensor". Attributes can be used to represent properties of the hardware, such as the fact that a distance sensor can have a minimum and maximum detectable distance.

In addition to hardware ontology and architecture for the IoT devices, the app store also offers services that are provided individually for each app. A container registry provides the IoT devices with the container images that contain the (parts of the) respective IoT application. As mentioned, IoT infrastructures can vary considerably from customer to customer. Thus, device owners must be involved in deciding which software runs on which device. In particular, this is not a decision that developers can make on their own, as personal requirements and wishes of the device owners can play a role here. For example, a user may wish not to deploy an audio recording application in the bathroom for privacy reasons, even if it would be technically possible with the existing equipment. To this end, the app store offers device owners a low-code configuration webapp where they can specify such requirements. These requirements are then translated into Prolog code and evaluated. The deployment process and infrastructure for enabling device owners to adjust their deployments using Prolog code is described in more detail in [24, 27].

IoT Device Development. In contrast to the development of IoT applications, the development of IoT devices is very close to hardware. The main task of IoT device developers is to provide access to the hardware in a form compatible with the app store. In terms of software, IoT devices consist of the following constituents:

- A container engine that executes the applications' containers **(A1)**.
- A set of application containers. These are created by the IoT application developer (via the CI pipeline) independently of the IoT devices. Thus, while they are present at runtime on the devices, the IoT device developer is unaware of which containers are executed by the device.
- A message broker that offers an implementation of a communication protocol between the application containers and the hardware drivers.
- A hardware access manager that orchestrates the communication between application containers and the hardware components they access. While this may seem unnecessary at first sight, because protocols like MQTT already offer a similar decoupling, it is necessary in some cases, *e.g.,* to prevent two applications from trying to control the same actuator.
- A set of hardware drivers. These hardware drivers are developed by the IoT device developer. They provide access to hardware component in a standardized form. To this end, every hardware driver also conforms to the hardware ontology provided by the app store **(R3)**.
- A specification of the device's hardware properties (*e.g.,* type, range, frequency, resolution, accuracy, ...) using an object diagram. The object diagram must conform to the class diagram provided by the IoT app store **(A2)**. If an object diagram defines objects that go beyond the class diagram, IoT applications need to be aware of this to make use of the information. Thus, it will likely be ignored by a large number of IoT applications.

This general structure, including the hardware access manager, is required by the app store. By providing standardized access to hardware components of IoT devices, the ontology and the Hardware Access Manager effectively act as a hardware abstraction layer for the IoT applications.

IoT Application Development. IoT application developers define the business logic of an application, *i.e.,* how the application is supposed to behave. They are responsible for developing the following artifacts:

- A specification of the application's requirements using OCL **(R3)**. The OCL expressions are based on the class diagram (*i.e.,* hardware ontology) provided by the app store **(A2)** and restrict which devices are capable of executing the individual components of their IoT application.
- A containerized IoT application **(A1)**. For this, MontiThings can be used to define IoT applications using C&C architectures. MontiThings then automatically generates C++ code from these models and builds (Docker) containers from them [27]. Note that while we recommend using MontiThings because it generates the necessary containers out of the box, using MontiThings is not a requirement for the IoT app store. As long as each container can be associated with its hardware requirements and product features,

other tools and languages can be used instead of MontiThings.
- High-level product features modeled using feature diagrams **(R2)**. The features of the feature diagram are tagged with the container images that implement each feature. Providing a feature diagram is not strictly necessary to deploy an application but gives devices owners a greater flexibility to only deploy parts of an application.

Application Execution. The application execution consists of both the edge and the cloud. The application containers are generally executed by IoT devices **(A1)**. Of course, they can also interact with cloud services where it is useful. If components do not have any requirements for specific hardware components, they could of course also be executed in the cloud. For its part, the cloud can offer services that are content-related to the application, such as image recognition, as well as general services such as a device registry for managing the available IoT devices. The deployment manager decides which device executes which parts of the application (*cf.* [24, 27]). Digital twin services and monitoring can help make the application more understandable. In the process, end users interact (consciously or unconsciously) with the IoT devices and thus influence the system. For example, they park their car in a certain parking space, thus unknowingly interacting with a parking space monitoring system.

5 Feature Diagramm Tagging

If an application is purchased from the IoT app store, it must first be decided which features are to be deployed. Feature models are a popular way to model product lines. Feature models specify the different features of a product and the dependencies between the features. By selecting a set of features, a *configuration* is formed that can represent, for example, the features implemented by a concrete product.

5.1 Architecture Feature Tagging

For our app store concept we use feature diagrams in combination with tagging. The IoT application developers model on one side a feature diagram that specifies all possible features of their product and on the other side use tagging to relate the features to the components of a C&C architecture. These models are made available to the app store, which displays them in web app, that enables endusers to select features and create a configuration in a graphical user interface (GUI) without working with text files **(R2)**. Of course, end users may want to select features that do not match the available hardware. Therefore, this process must be supported by appropriate analyses (see Sec. 5.2).

When users select a feature, it means that all components assigned to the feature via tagging must be deployed in the application. Ideally, application developers model the feature diagrams in such a way that no valid feature configuration can be selected, so that a component lacks other components it must communicate with in order to implement a feature. In complex architectures, however, it can be difficult to recognize how components relate to each other. To enable application developers to abstract from low-level details, we automatically assume tagging a component also includes all subcomponents a component might have. It is, thus, not possible to end up with *half-empty* components that cannot function normally because they miss some of their subcomponents.

Fig. 4 shows this concept using a fire extinguisher example. The `FireExtinguisher` root feature tags all components that need to be present regardless of the feature configuration. In this example, the `fex` component, that detects fires and takes appropriate action on its decision. The features below the root feature in the feature diagram mandate that every fire extinguisher needs some form of input, *i.e.,* a smoke or heat detector (or both). The two features for smoke and heat detection are tagged with the according `smoke` and `temp` components of the architecture. While every fire extinguisher needs an alarm system (which could also call the fire department), the sprinklers are optional. In case a configuration does not use the sprinkler, all messages the `fex` sends on its outgoing port will be discarded. In case a component that provides input to another component is not mandated, this is treated as if the sending component never sends a message.

We use the feature configuration to generate rules according to MontiThings' Prolog-based deployment algorithm [27]. For each component `X` that is tagged by a feature in the feature configuration, we create a rule that requires component `X` to be deployed at any location. If device owners require more complex rules for distributing their software across their devices, they can of course still use MontiThings' location-based deployment rules to, *e.g.,* require the fire extinguisher to be located in the bedroom.

5.2 Analyses

With feature models that represent product lines and feature configurations that specify individual products of the product line, a large number of analyses [8] can be formulated. The combination between feature diagrams and architectural model enables carrying out various further forms of analyses. This section presents some of these analyses that we realized. All presented analyses require that a product line is described by a given feature diagram, the software architecture is modeled as a given component & connector model and a tagging model (*cf.* Sec. 2) bridges the two models.

Deployable feature configuration. With a given feature configuration and a set of hardware components, an automated analysis can detect whether the feature configuration is realizable by deploying the software components that are tagged by features that are part of the feature configuration to the given hardware. The result of the analysis is either a

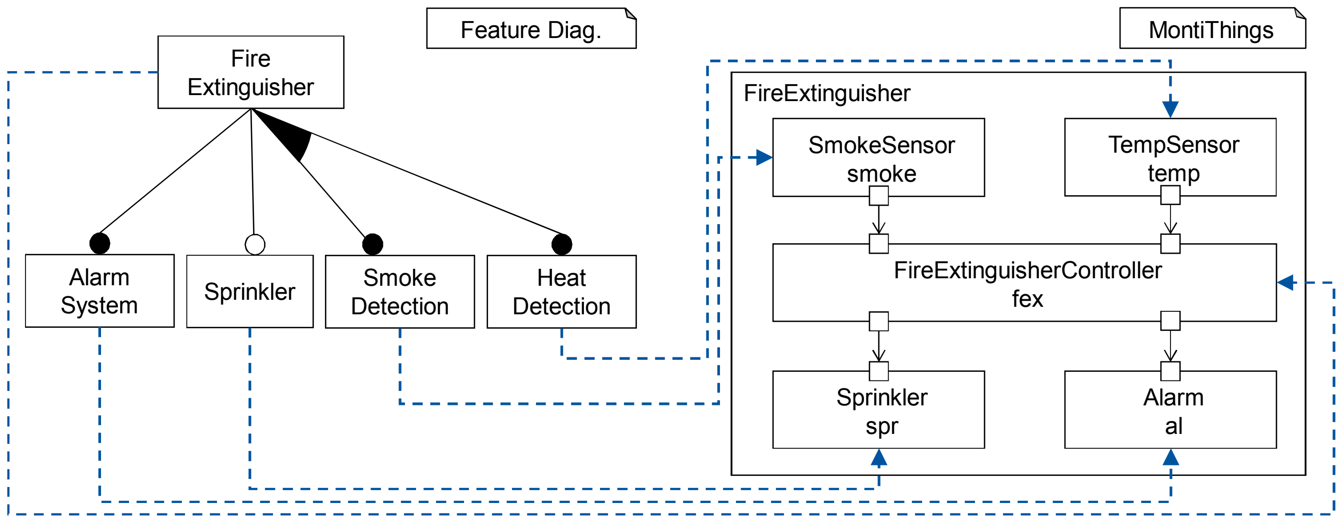


Figure 4. Example of tagging a feature diagram of a fire extinguisher application with components from an according architecture model. Each blue dotted arrow between the two models represents a tag. Architecture model taken from [26].

deployment configuration that deploys these components to the given hardware or empty, if no such deployment exists.

A use case in the smart home example would be a user who owns a certain set of smart home devices and checks whether a desired set of smart home features is realizable with the given hardware constellation.

Largest configurations. Given a complete product line in terms of a feature diagram, a set of software components, and a set of hardware components, we can calculate different forms of *largest configurations*. In this, a largest configuration contains either the maximal number of hardware components or selects the maximal number of features in a feature diagram without violating its restrictions. A largest configuration is not necessarily unique, i.e., there may be different maximal configurations that have an equal number of features/hardware components. For example, if a mandatory feature of the feature diagram has two sub features A and B that mutually exclude each other, there are at least two maximal feature configurations where one contains A and the other one contains B.

A largest deployment configuration uses a given product line, a given feature configuration, and a given set of hardware components to calculate deployment configurations that maximize the number of components deployable to the given hardware. A use case in the smart home example would be a user who selects a set of desired smart home features and owns a set of smart home devices. The largest deployment configuration indicates, how the components can be deployed to ensure the greatest redundancy in critical software components.

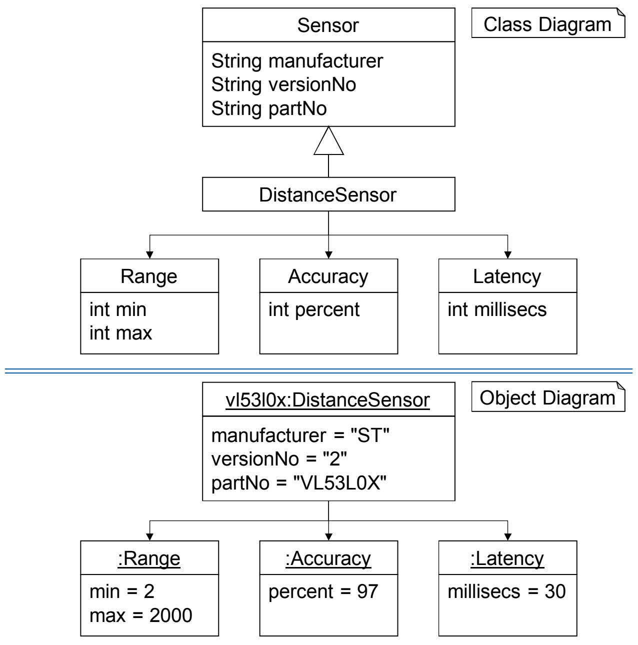
Similarly, we can calculate a largest feature configuration from a given product line and set of available hardware components. A largest feature configuration contains the maximal number of features that can be selected for a given set of hardware components. A use case in the smart home example would be a user who owns a set of smart home devices and wants to maximize the number of smart home features that can be realized with these devices.

Complete to a largest feature configuration. With partial feature configurations, users can predetermine that a certain set of features should be included or excluded in a (full) configuration while making no assumptions on other, "undecided" features. With such partial feature configurations, we can extend the analysis that calculates the largest feature configurations by calculating complete configurations from given partial configurations. In the smart home example, a use case would be a user who owns a set of smart home devices and has pre-selected certain smart home features she intends to have and others that she does not require. The analysis can complete the given partial selection to a complete selection of features that, again, is not necessarily unique.

6 Device Specification and Selection Using Generated Prolog Code

Both during the feature diagram analyses and the subsequent deployment, it is necessary to know which IoT devices can run which software. The automated process of determining whether hardware exists that matches the specifications made on the software side is carried out with the aid of a Prolog generator. The process consists of three steps:

1. object diagrams contributed by IoT device developers are translated into Prolog facts,
2. the OCL expression created by the IoT application developer is translated into a Prolog query, and



OCL

```
exists DistanceSensor sensor:
  let max = sensor.range.max;
      min = sensor.range.min
  in  max > min
      implies max - min > 1000 &&
              sensor.accuracy.percent > 90
```

Prolog
«gen»

```
1  % Facts (from object diagram)
2  deviceType(vL53L0X, "DistanceSensor").
3  manufacturer(vL53L0X, "ST").
4  versionNo(vL53L0X, "2").
5  partNo(vL53L0X, "VL53L0X").
6  latency__millisecs(vL53L0X, 30).
7  accuracy__percent(vL53L0X, 97).
8  range__min(vL53L0X, 2).
9  range__max(vL53L0X, 2000).
10
11 % Ensure __Sensor variable uses a single object
12 range__min(__Sensor, Sensor__range__min),
13 range__max(__Sensor, Sensor__range__max),
14 accuracy__percent(__Sensor, Sensor__accuracy__percent),
15 % Note: Latency is irrelevant for OCL expression
16 instanceOf(__Sensor, "DistanceSensor"),
17
18 % Evaluate properties of hardware attributes
19 (
20   Max is Sensor__range__max,
21   Min is Sensor__range__min,
22   (
23     \+ Max > Min ;
24     (
25       Max - Min > 1000,
26       Sensor__accuracy__percent > 90
27     )
28   )
29 ).
```

Figure 5. Example of an app store's hardware ontology (class diagram), the specification of the hardware of a device (object diagram) and the specification of the requirements of a component (OCL). Prolog code generated from the object diagram and OCL expression checks whether a device meets the requirements of the software.

3. the (generated) Prolog query is checked against the (generated) Prolog facts.

The object diagram and OCL expression are written in accordance to a class diagram that defines the hardware ontology but is not used for code generation. Fig. 5 gives an example of the artifacts used in this process. This approach enables hardware and software to remain decoupled until the deployment process. Thereby, both hardware and software can be developed independently of each other by different actors by only agreeing on a common ontology, *i.e.,* class diagram, provided by an IoT app store. The generated facts and queries are used by MontiThings' Prolog-based deployment algorithm to create a deployment [27].

Since Prolog is not an object-oriented programming language, generating facts from object diagrams requires transforming the object diagrams. Object diagrams can contain nested objects, *i.e.,* objects that are included in or referenced by other objects. We addressed this by generating a separate fact for each attribute that has a primitive data type. The name of the fact is generated so that it can be uniquely assigned to the original object again. For example, the fact that the VL53L0X in the object diagram has a reference to a latency object with the attribute `millisecs` is translated into a fact `latency__millisecs(vL53L0X, 30)`.

OCL expressions may evaluate to true or false. The expressions have to adhere to the class diagram shown at the top of Fig. 5. Since the goal is to check whether a device has the necessary hardware components required by the software, the outermost OCL expression of the requirements must always be an `exists`-expression. The resulting Prolog query is always divided into two parts: The first part asserts that the free variables referring to the object in the query need to be bound to the same variable. Thereby, we ensure that the required properties must be fulfilled by the same object, *i.e.,* piece of hardware, instead of allowing Prolog to mix the properties of different objects. An example for this can be seen in lines 11 to 16 in the generated Prolog code in Fig. 5. The second part of the Prolog code represents the OCL expression. We traverse the abstract syntax tree of the OCL expression using the *Visitor* pattern [18] and transform each part of the OCL expression in equivalent Prolog terms. The resulting code can be seen in lines 18 to 29 of the generated Prolog code in Fig. 5. If the hardware matches the software's requirements, the query is able to assign the free variables. For example, in Fig. 5, the generated Prolog code can assign the value `vL53L0X` to the variable `__Sensor`.

The handling of these expressions is mostly a straight translation into similar Prolog expressions. Instantiations like `DistanceSensor sensor` in the query are translated into `instanceOf(__Sensor, "DistanceSensor")`. The `instanceOf` rule checks if the sensor has the given `deviceType` or a `deviceType` that inherits from it. The inheritance relationships are given

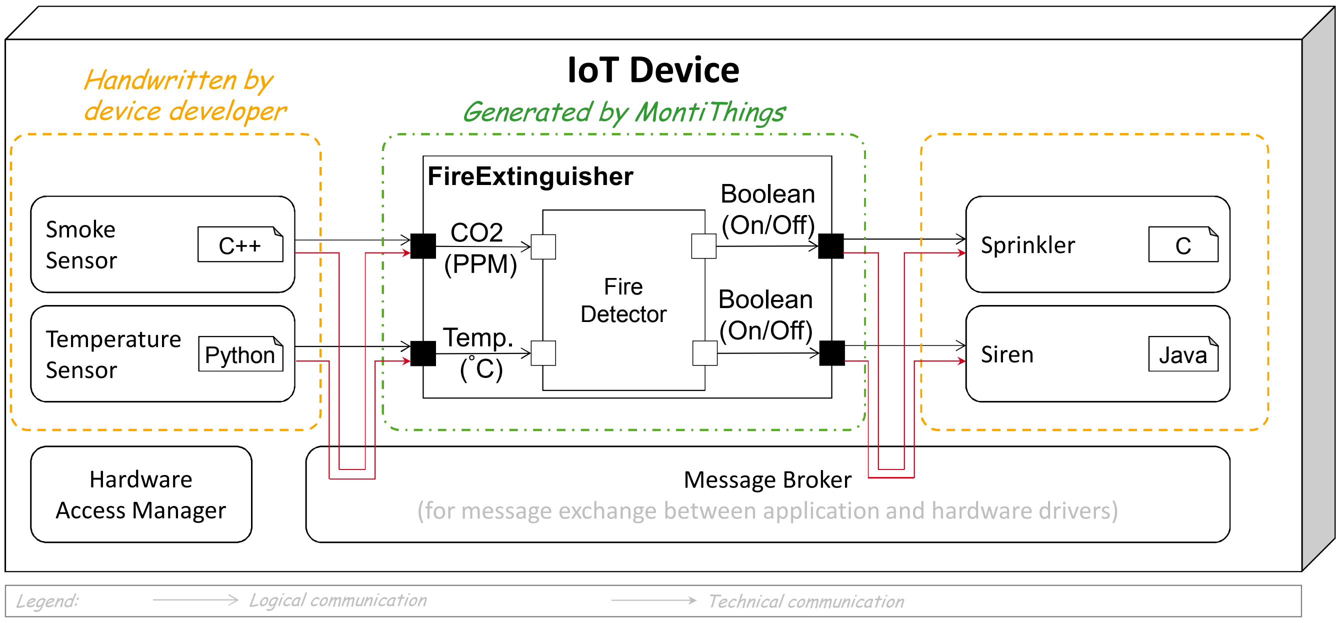


Figure 6. Hardware drivers and application components are decoupled via a message broker. The hardware manager assigns hardware drivers to the components' ports based on the ports' requirements and the app store's hardware ontology.

by the class diagram. Thereby, we support inheritance of different sensor types.

In our example, Prolog is able to find a valid assignments for all variables (especially `__Sensor = vL53L0X`). Accordingly, the device specified by the object diagram is able to execute the software whose requirements are given by the OCL expression.

7 Hardware Access

Once it has been decided which devices should run which software and the devices have downloaded the necessary containers, the high-level IoT application still needs to be connected to the hardware drivers. To decouple hardware access and application development, the two must not know each other at design time. However, shortly before the start of runtime, hardware and software must be connected. This raises the question of how the code generated from (architecture) models is linked to the mostly handwritten code of hardware drivers. The components of C&C architectures exchange data via ports. Accordingly, communication with external hardware in MontiThings also takes place via ports [27]. However, since the development of the IoT devices and the applications is decoupled in our app store concept, it is not intended here that the drivers for a hardware component are stored directly in a port. To break this link, the hardware drivers are deployed as standalone binaries on the IoT devices, which then exchange data via a message broker with the code generated from the architecture models (Fig. 6). One advantage of decoupling the hardware drivers from the code generated from the architecture is that the drivers do not necessarily have to be written in the same programming language as the generated code. All that is required is that they can exchange objects, *i.e.,* (de)serialize, via a common data format. In commercial implementations, libraries such as Google Protobuf[2] offer consistent (de)serialization across multiple GPLs.

There is a Hardware Access Manager on each IoT device that coordinates access to the hardware drivers. As soon as a hardware driver is started on an IoT device, it informs the Hardware Access Manager about its existence. In particular, it also informs the Hardware Access Manager of its specification according to the hardware ontology of the app store. The architecture behaves in a similar way and informs the Hardware Access Manager for which of its ports it requires hardware drivers with which specification. The Hardware Access Manager uses these requests from the components and offers from the hardware drivers to determine, based on the ontology, whether the offered hardware can fulfill the requests. The Hardware Access Manager tells the components which topics they can use to communicate with the hardware drivers assigned to them.

8 Smart Home Application Case Study

To validate our concept, we modeled a smart home application that could be deployed via an IoT app store. Technically, the case study was carried out using mainly Microsoft Azure.

[2]Google Protocol Buffers. [Online]. Available: https://developers.google.com/protocol-buffers. Last accessed: 02.05.2022

We used the an Azure IoT Hub for device management, a virtual machine for executing the configuration web app, three virtual machines representing the IoT devices using mocked sensors and actuators, and an Azure Container Registry for providing the applications' Docker images of to the IoT devices.

To develop the models, the application developer can use a web-based development environment. We have chosen GitPod for this purpose. In GitPod, developers get a VSCode-like development environment in their web browser, where we provide necessary tools for code generation or compilation. We chose a VSCode-like web interface to provide developers with an environment they are already familiar with from traditional application development.

At design time the IoT application developers create the following artifacts:

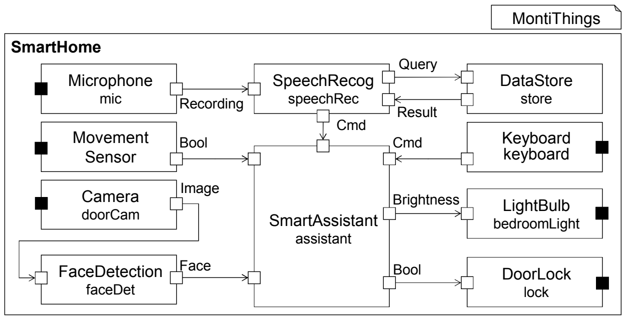
- the MontiThings models which describe the business logic,
- a feature digram which describes the high-level features from a user perspective,
- a tagging model which connects feature diagram and MontiThings components,
- OCL expressions which specify the MontiThings components' requirements.

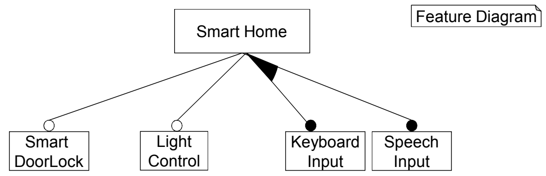
Fig. 7 shows an excerpt of the models used in the case study. Especially it shows the outermost MontiThings component, the feature diagram and tagging model to give a high-level overview of the application. Furthermore, the figure shows an OCL expression requiring a camera and an object diagram describing the Raspberry Pi Camera Module v2[3] that matches this requirement.

The IoT device developers create an object diagram for each IoT device that specifies the device's capabilities. The IoT application and IoT device developers are decoupled in creating these models. Their only connection is a class diagram defining the IoT app store's hardware ontology. This class diagram can be provided by the app store to unify the modeling of capabilities in the object diagrams, enabling OCL queries to be written against the class diagram without knowing the object diagrams. For example, the IoT application developer can request a camera with at least 4 megapixels (l. 2 of the OCL expression in Fig. 7) without knowing what specific camera the IoT device will be equipped with.

Once the device owners, *i.e.*, the app store customers, have decided on an application, they can configure it via a configuration web app (which is also developed model-driven [4, 19]). This web app could be provided by the app store or be deployed on a computer of the device owner. In this web app they can see the features that have been developed in the feature diagram. The view of device owners is very different

[3]Raspberry Pi Documentation—Camera. [Online]. Available: https://www.raspberrypi.com/documentation/accessories/camera.html. Last accessed: 28.07.2022





Feature Tagging

```
tagging SmartHome {
  feature SmartHome     contains assistant, store;
  feature SmartDoorLock contains doorCam, faceDet, lock;
  feature LightControl  contains movementSensor, bedroomLight;
  feature KeyboardInput contains keyboard;
  feature SpeechInput   contains mic, speechRec;
}
```

OCL

```
1 exists Camera cam:
2         cam.resolution.megapix >= 4 &&
3         cam.fieldOfView.vertical isin { 60 .. 100 }
```

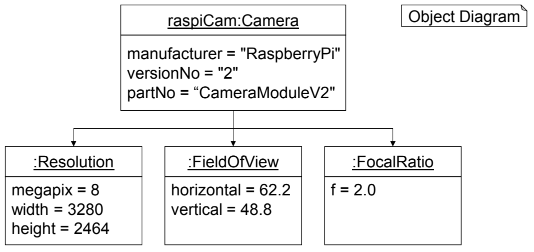


Figure 7. Excerpt of the models used in the case study. The smart home is based on the application in [25].

from the view for developers. Instead of an IDE, customers are presented with a web app that enables them to select features via checkboxes and buttons and perform analyses such as the largest possible configuration. The device owners proceeded as follows.

After asking the web app to validate its (still empty) feature configuration, the tool indicates that the configuration is currently invalid. The device owners now calculate the largest valid configuration and see that all features are selected here. Unfortunately, the configuration is not deployable like this, because they lack the keyboard hardware necessary to deploy the `keyboard` component. Since they are not willing to buy a keyboard, they next calculate the maximum deployable configuration with our existing hardware. As a result, all features except the `KeyboardInput` are selected. Since there is only one possible largest deployable configuration in this case, we decide to use this configuration.

In the background, this causes the web app to communicate with the IoT devices and request them to download the Docker images needed to run the application. The details of this process are described in [24]. Once a component gets started on an IoT device, it tries to connect to the sensors and actuators on the respective IoT devices. The hardware access manager preinstalled on the devices assigns corresponding instances of hardware drivers to each component or port that wants to communicate with hardware. Communication takes place via an message queue telemetry transport (MQTT) broker installed locally on the devices. This completes the deployment of the application and end users could now interact with the IoT devices.

9 Discussion

We are aware that the app store concept presented in this paper is focused on the deployment aspects of IoT app stores. Our prototype implementation is, thus, not exhaustive and neglects aspects like, *e.g.,* payment of the applications. We see our concept as a template for later commercial implementations to show how model-driven development can contribute to certain aspects of the development of such app stores. The market landscape of IoT hardware is very fragmented. Therefore, it is hardly possible to specify an all-encompassing ontology without neglecting certain hardware components. We expect that future IoT app stores will also have to impose certain restrictions on hardware manufacturers in this respect and will not be able to support every arbitrarily unusual hardware part type.

The definitions of the data types that can be exchanged with the hardware components also move in this area of tension. Eclipse UPM[4] is already trying to provide a library with which access to widely used sensors and actuators can be standardized. The problem here is that it is necessarily necessary to abstract from the specifics of a sensor. Some (more complex) IoT components offer fine-grained setting options in this respect, for example to balance the reading speed of an RFID reader against the accuracy of the reading. Often this is done by very close to the hardware, for example, by setting a certain value in a memory register. Since these setting options can also differ greatly between similar hardware, abstraction must necessarily neglect some functionalities here. We therefore decided not to abstract directly from hardware access in our app store concept, but to let device developers implement the hardware drivers with handwritten code and to subject only the values exchanged with the application to the app store's ontology.

Our use of (Docker) containers **(A1)** undoubtedly creates some overhead. The same applies to the message broker used locally on the IoT devices. We are of the opinion that this path is nevertheless promising for the future, since 1. the performance of IoT hardware continues to increase and 2. corresponding technologies are becoming more and more resource-efficient and IoT-friendly. For example, the balenaEngine[5], an Docker-compatible container engine focussed on IoT devices, has a considerably smaller memory footprint than Docker CE. In general, it is expected that future IoT applications will be based increasingly on container technologies [33]. Nevertheless, we want to emphasize that our assumption to use Docker containers was only made to keep the focus of this paper on the high-level concepts instead of discussing how to cross-compile and copy binaries to IoT devices.

Based on our use of Docker containers, it follows that strongly resource-constrained devices, *e.g.,* Arduino or ESP32, cannot be updated with new software by our prototype. To include such devices in the application, we envision an approach similar to the approach of Amazon web services (AWS) Greengrass: If the resource-constraint device can connect to a more powerful gateway device, it could act as a port of the gateway device by directly connecting to its Hardware Access Manager. Thus, the capabilities of the resource-constraint extend the capabilities of the gateway device. The code generated from MontiThings applications can of course also be deployed without (Docker) containers given that the target devices provide the necessary libraries. Overall, however, we consider this approach only as a transitional method and assume that IoT devices will rely much more on container technologies in the future. We refer readers interested in deploying software to resource-constraint devices to the GeneSIS project [15–17].

10 Related Work

Multiple authors have proposed IoT focussed app stores, *e.g.,* [5, 10, 29]. While these approaches share with us the general problem of decoupling hardware and software, we are the first to explore the potential of holistic model-driven development in this regard. Our use of feature diagrams also allows us to focus more clearly on product lines of IoT applications. Due to the great heterogeneity and variability of IoT infrastructures, we think considering IoT applications as product lines is an important but understudied aspect.

A theoretical approach to deploying IoT applications using feature diagrams has also been studied in [11]. Unlike our approach, it is focussed on developers optimizing an IoT application according to metrics (such as latency) for a specific infrastructure by modeling both software features and (hard- and software) infrastructure using feature diagrams. It does not take into account the app store use case of device owners (without a technical background) who want

[4]Eclipse UPM GitHub Projekt. [Online]. Available: https://github.com/eclipse/upm. Last accessed: 02.05.2022

[5]balenaEngine Product Website. [Online]. Available: https://www.balena.io/engine/. Last accessed: 02.05.2022

to deploy software without first modeling their device infrastructure and without involving developers. Furthermore, using OCL and object diagrams for specifying requirements and device capabilities, instead of only feature diagrams, our approach enables specifying more expressive requirements. Lastly, our approach goes beyond purely theoretical deployment calculations by also considering the decoupling of the software and device development process and providing the code generators to produce executable applications.

Academia and industry have proposed a large number of IoT domain-specific languages (DSLs) over the last decade. The of the two most prominent examples are ThingML [21, 28] and Calvin [6, 30, 31]. ThingML itself does not provide mechanisms for large-scale deployment of IoT software. The GeneSIS project [15–17] extends ThingML with deployment functionalities. However, GeneSIS focuses on the technical aspects of deployment. Aspects such as a reasoner, which decides which devices should run which software, are not considered. To connect different hardware components, ThingML often relies on developing different components for different platforms (*e.g.,* a component specifically for an Arduino device[6]). Unlike our approach, however, the hardware is already specified at design time. An app store with such a strong coupling would only be possible to a very limited extent because the app store can deliver corresponding components for all conceivable combinations of platforms and hardware. Ericsson's Calvin is also an IoT-focussed C&C ADL. Calvin comes out-of-the-box with a deployment mechanism. Components can serialize their state and send it on to neighboring devices to be deserialized and instantiated there. This creates a distributed network of IoT devices without a central instance. However, the lack of a central instance means that no IoT device has global knowledge about the network of IoT devices. Accordingly, it is not possible to define deployment rules that affect the network as a whole. For example, it is possible to require that software be deployed to *every device in the bathroom* because each device can decide for itself whether it is in the bathroom, but it is not possible to require that software be deployed to *3 devices in the bathroom.*

To connect legacy devices, Calvin uses a mechanism similar to CapeCode's [9] accessor pattern. A component is written as a wrapper around the service or hardware to be connected. In a sense, the hardware drivers in our concept serve a similar purpose. However, our concept differs in that 1. are based on the app store's ontology and thus decouple device and application, whereas accessors are just regular components of the architecture, 2. the hardware drivers are developed by the device developer, not by the application programmer, and 3. are also decoupled from the generated code on the IoT devices, *i.e.,* can be programmed in different programming languages.

A large number of IoT ontologies already exist in the literature [7]. Many of these build on the W3 Semantic Sensor Network Ontology. Unfortunately, there is still no sign of a common standard emerging from the various ontologies. We have, therefore, avoided proposing a concrete ontology for IoT app stores to avoid giving the impression of defining *yet another ontology*. Instead, we recommend future IoT app store operators to base their ontologies on the existing solutions. By defining ontologies using class diagrams, we enable future them to use existing solutions but also adapt them to their needs if necessary.

11 Conclusion

Within this paper, we have introduced a model-driven approach for creating IoT app stores and have shown the application of the approach for a smart home case study. Using web applications, such IoT app stores enable device owners to configure features of the IoT software. An underlying ontology of hardware components ensures that the specification of devices by device developers and the specification of requirements by IoT application developers fit together.

The presented approach decouples hardware and software facilitating the creation of IoT app stores and providing IoT device owners with more freedom in their software choices.

Source Code

MontiThings is available on GitHub:
https://github.com/MontiCore/montithings

Acknowledgements

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy—EXC 2023 Internet of Production—390621612. Website: https://www.iop.rwth-aachen.de

References

[1] [n. d.]. App Manifest Overview—Android Developer Documentation. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. Last checked 29. April 2022.

[2] [n. d.]. Azure IoT Edge Product Website. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/. Last checked 29. April 2022.

[3] [n. d.]. Balena Website. [Online]. Available: https://www.balena.io/. Last checked 29. April 2022.

[4] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19) (LNI, Vol. P-304)*. Gesellschaft für Informatik e.V., 59–66.

[5] Shabir Ahmad, Faisal Mehmood, Asif Mehmood, and DoHyeun Kim. 2019. Design and Implementation of Decoupled IoT Application Store: A Novel Prototype for Virtual Objects Sharing and Discovery. *Electronics* 8, 3 (2019). https://doi.org/10.3390/electronics8030285

[6] Code example from ThingML's GitHub repository. [Online]. Available: https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/sos2/hardware/arduino/arduino.thingml. Last accessed: 02.05.2022

[6] Ola Angelsmark and Per Persson. 2017. Requirement-Based Deployment of Applications in Calvin. In *Interoperability and Open-Source Solutions for the Internet of Things*, Ivana Podnar Žarko, Arne Broering, Sergios Soursos, and Martin Serrano (Eds.). Springer International Publishing, Cham, 72–87.

[7] Garvita Bajaj, Rachit Agarwal, Pushpendra Singh, Nikolaos Georgantas, and Valérie Issarny. 2017. A study of existing Ontologies in the IoT-domain. *CoRR* abs/1707.00112 (2017). arXiv:1707.00112 http://arxiv.org/abs/1707.00112

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[9] Christopher Brooks, Chadlia Jerad, Hokeun Kim, Edward A. Lee, Marten Lohstroh, Victor Nouvelletz, Beth Osyk, and Matt Weber. 2018. A Component Architecture for the Internet of Things. *Proc. IEEE* 106, 9 (September 2018), 1527–1542.

[10] Arne Bröring, Stefan Schmid, Corina-Kim Schindhelm, Abdelmajid Khelil, Sebastian Käbisch, Denis Kramer, Danh Le Phuoc, Jelena Mitic, Darko Anicic, and Ernest Teniente. 2017. Enabling IoT Ecosystems through Platform Interoperability. *IEEE Software* 34, 1 (2017), 54–61.

[11] Angel Cañete, Mercedes Amor, and Lidia Fuentes. 2022. Supporting IoT applications deployment on edge-based infrastructures using multi-layer feature models. *Journal of Systems and Software* 183 (2022), 111086. https://doi.org/10.1016/j.jss.2021.111086

[12] Rustem Dautov, Hui Song, and Nicolas Ferry. 2021. Towards a Sustainable IoT with Last-Mile Software Deployment. In *IEEE Symposium on Computers and Communications (ISCC)*. 1–6. https://doi.org/10.1109/ISCC53001.2021.9631250

[13] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2022. Designing and constructing internet-of-Things systems: An overview of the ecosystem. *Internet of Things* 19 (2022), 100529. https://doi.org/10.1016/j.iot.2022.100529

[14] Eclipse Foundation. 2022. IoT & Edge Developer Survey Report. [Online]. Available: https://outreach.eclipse.foundation/iot-edge-developer-survey-2022. Last accessed: 16.10.2022.

[15] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. 2019. GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems. In *IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 870–875.

[16] Nicolas Ferry and Phu H. Nguyen. 2019. Towards Model-Based Continuous Deployment of Secure IoT Systems. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 613–618.

[17] Nicolas Ferry, Phu H. Nguyen, Hui Song, Erkuden Rios, Eider Iturbe, Satur Martinez, and Angel Rego. 2020. Continuous Deployment of Trustworthy Smart IoT Systems. *Journal of Object Technology* 19 (2020), 16:1–23.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers* (Vienna). CEUR Workshop Proceedings, 22–30.

[20] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. 2015. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE, 34–43.

[21] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proceedings of the ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '16)*. ACM, 125–135.

[22] B. Henderson-Sellers and C. Gonzalez-Perez. 2006. Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0. In *Model Driven Engineering Languages and Systems*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–26.

[23] Mengda Jia, Ali Komeily, Yueren Wang, and Ravi S. Srinivasan. 2019. Adopting Internet of Things for the development of smart buildings: A review of enabling technologies and applications. *Automation in Construction* 101 (2019), 111–126. https://doi.org/10.1016/j.autcon.2019.01.023

[24] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. 2022. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things* 3, 4 (November 2022).

[25] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. 2021. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21)*. ACM SIGPLAN, 197–209.

[26] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2020. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 90–101.

[27] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. 2022. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software* 183 (January 2022), 111087.

[28] Brice Morin, Nicolas Harrand, and Franck Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (January 2017), 30–36.

[29] Dejan Munjin and Jean-Henry Morin. 2012. Toward Internet of Things Application Markets. In *IEEE International Conference on Green Computing and Communications*. 156–162. https://doi.org/10.1109/GreenCom.2012.33

[30] Per Persson and Ola Angelsmark. 2015. Calvin – Merging Cloud and IoT. *Procedia Computer Science* 52 (2015), 210 – 217. 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT).

[31] Per Persson and Ola Angelsmark. 2017. Kappa: Serverless IoT Deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing* (Las Vegas, Nevada) *(WoSC '17)*. ACM, 16–21.

[32] T. Stahl, M. Voelter, and K. Czarnecki. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[33] Antero Taivalsaari and Tommi Mikkonen. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 34, 1 (Jan 2017), 72–80.

[34] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (jun 2014), 45 pages. https://doi.org/10.1145/2580950

[35] Thumeera R. Wanasinghe, Raymond G. Gosine, Lesley Anne James, George K. I. Mann, Oscar de Silva, and Peter J. Warrian. 2020. The Internet of Things in the Oil and Gas Industry: A Systematic Review. *IEEE Internet of Things Journal* 7, 9 (2020), 8654–8673.

Received 2022-08-12; accepted 2022-10-10