Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals

JÖRG CHRISTIAN KIRCHHOF, ANNO KLEISS, BERNHARD RUMPE, DAVID SCHMALZING, and PHILIPP SCHNEIDER, RWTH Aachen University, Germany
ANDREAS WORTMANN, University of Stuttgart, Germany

Today's **Internet of Things (IoT)** applications are mostly developed as a bundle of hardware and associated software. Future cross-manufacturer app stores for IoT applications will require that the strong coupling of hardware and software is loosened. In the resulting IoT applications, a quintessential challenge is the effective and efficient deployment of IoT software components across variable networks of heterogeneous devices. Current research focuses on computing whether deployment requirements fit the intended target devices instead of assisting users in successfully deploying IoT applications by suggesting deployment requirement relaxations or hardware alternatives. This can make successfully deploying large-scale IoT applications a costly trial-and-error endeavor. To mitigate this, we have devised a method for providing such deployment suggestions based on search and backtracking. This can make deploying IoT applications more effective and more efficient, which, ultimately, eases reducing the complexity of deploying the software surrounding us.

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

Additional Key Words and Phrases: Internet of Things, deployment, model-driven engineering, architecture description languages

ACM Reference format:
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 Trans. Internet Things* 3, 4, Article 30 (September 2022), 30 pages.
https://doi.org/10.1145/3549553

1 INTRODUCTION

The systems our society thrives upon are becoming increasingly interconnected. We are surrounded by the **Internet of Things (IoT)**: Our appliances, factories, power plants, medical devices, mobile healthcare applications, smart home security systems, vehicles, and more are connected to the Internet and continuously exchange data with other systems in the IoT.

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.
Authors' addresses: J. C. Kirchhof, A. Kleiss, B. Rumpe, D. Schmalzing, and P. Schneider, Software Engineering, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany; emails: kirchhof@se-rwth.de, anno.kleiss@rwth-aachen.de, rumpe@se-rwth.de, schmalzing@se-rwth.de, philipp.schneider4@rwth-aachen.de; A. Wortmann, Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Seidenstraße 36, 70174 Stuttgart, Germany; email: andreas.wortmann@isw.uni-stuttgart.de.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6207/2022/09-ART30 $15.00
https://doi.org/10.1145/3549553

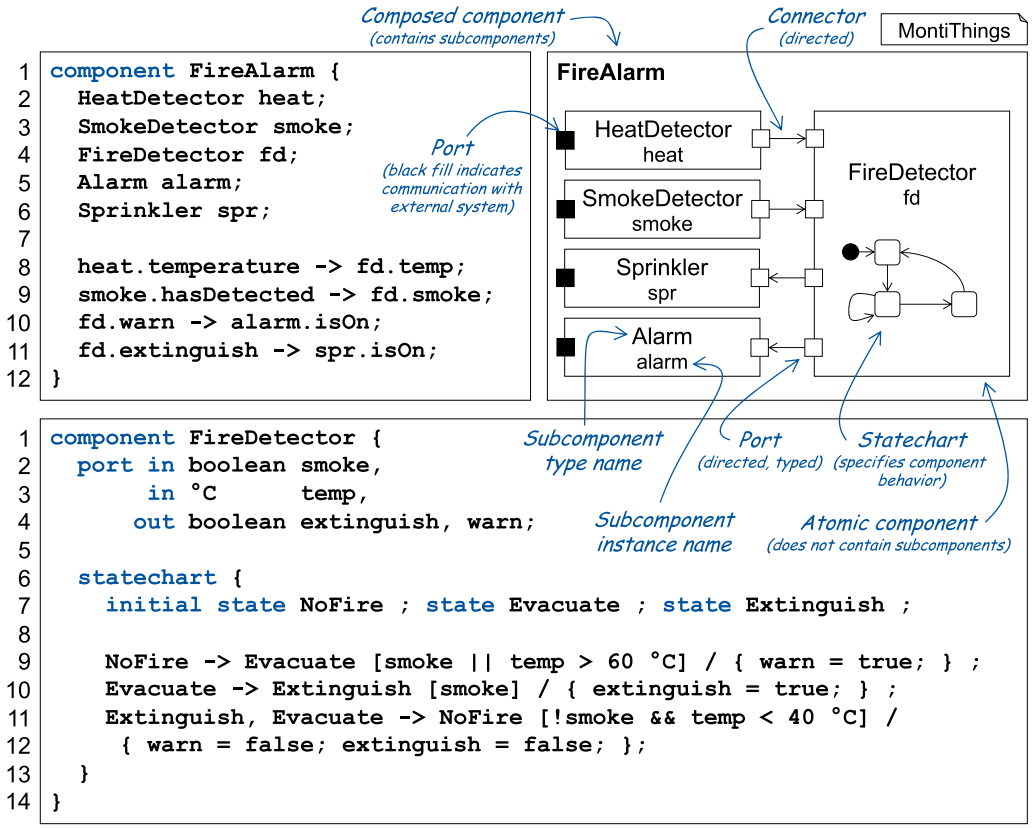
Today, IoT devices are tightly coupled with the applications they execute: Typically, hardware and software come from the same vendor, allowing the software to be pre-installed directly at the factory. If the manufacturer decides to stop supporting the software, then this can turn even technically flawless IoT devices into e-waste [24]. This is neither ecologically nor economically sustainable. For the future, IoT app stores [2, 10, 29], similar to smartphone app stores, are foreseen that decouple hardware and software from each other to a greater extent. For this vision to come true, robust deployment algorithms are needed to decide which devices should run which parts of an IoT application.

Alas, this decoupling also increases the complexity of IoT applications due to the increased variability of infrastructures to which an application might be deployed. Model-driven engineering [19] can reduce such complexity by offering different levels of abstraction [35] and has already been applied to many IoT systems successfully [4, 17, 22]. In particular, models can also support the deployment of IoT applications [4, 16]. However, existing approaches to computing deployments for IoT systems are either of limited expressiveness, solely focus on technical details, or only fit deployments on perfectly matching devices. If software cannot be deployed, then error messages are issued instead of actively suggesting improvements such as buying additional devices. This strongly restricts the usefulness of such approaches in the real-world deployment of IoT software via app stores.

We conceived the MontiThings [27] **component and connector (C&C) architecture description language (ADL)** infrastructure for modeling and deploying IoT applications across networks of heterogeneous hardware devices, i.e., devices with very diverse sets of available sensors and actuators. MontiThings features a novel method to assist in deploying IoT applications to variable networks of IoT devices: Instead of only computing whether the deployment requirements modeled with its components fit onto the target devices, it leverages search and backtracking in a novel approach to compute deployment alternatives as well as suggestions to improve the devices to fit the deployment requirements. In this article, we present this method. Since the hardware and software in industrial applications are usually planned by experts, our method here relates primarily to consumer applications, e.g., from the smart home domain. However, there is no conceptual limitation that would exclude the use of our method in an industrial context. Flexibly suggesting relaxations or alternative hardware eases finding useful deployments for large-scale, heterogeneous IoT applications and can make their deployment more effective and more efficient. Accordingly, we contribute the following:

- a self-adaptive deployment process for IoT app stores that involves device owners in addition to developers and enables the device owners to influence software deployment decisions,
- a deployment algorithm for C&C architectures based on generated Prolog code that can propose requirement and hardware modifications in case of unfulfillable requirements, and
- a concept for modeling dynamic reactions to changes in the deployment of C&C-based IoT applications.

Next, Section 2 introduces MontiThings preliminaries, and Section 3 motivates the need for powerful deployment by example. Based on this, Section 4 describes requirements for effective and efficient deployment of IoT applications. Afterward, Section 5 overviews the MontiThings deployment method before Section 6 details its mechanisms for integrating new IoT devices at runtime, and Section 7 explains the architecture of the MontiThings deployment infrastructure. Section 8 details the algorithm for calculating deployment modification proposals. Section 9 then details its application in a comprehensive case study. Finally, Section 10 details related work, Section 11 discusses observations, and Section 12 concludes.




```
1  component FireAlarm {
2    HeatDetector heat;
3    SmokeDetector smoke;
4    FireDetector fd;
5    Alarm alarm;
6    Sprinkler spr;
7
8    heat.temperature -> fd.temp;
9    smoke.hasDetected -> fd.smoke;
10   fd.warn -> alarm.isOn;
11   fd.extinguish -> spr.isOn;
12 }
```

```
1  component FireDetector {
2    port in boolean smoke,
3         in °C      temp,
4        out boolean extinguish, warn;
5
6    statechart {
7      initial state NoFire ; state Evacuate ; state Extinguish ;
8
9      NoFire -> Evacuate [smoke || temp > 60 °C] / { warn = true; } ;
10     Evacuate -> Extinguish [smoke] / { extinguish = true; } ;
11     Extinguish, Evacuate -> NoFire [!smoke && temp < 40 °C] /
12      { warn = false; extinguish = false; };
13   }
14 }
```

Fig. 1. Example of a fire extinguisher application in graphical syntax and textual syntax (adapted from Reference [26]).

2 PRELIMINARIES

We use MontiThings [27] for modeling IoT applications and generating code from models. MontiThings is a C&C ADL for IoT applications. As such, MontiThings describes the behavior of (distributed) IoT applications by means of components that exchange data with each other. More precisely, components exchange streams of messages between their interfaces, which consist of directed and typed ports. For this, two ports can be connected with a connector. The behavior with which MontiThings components react to incoming messages can be described using

(1) an IoT-focussed Java-like behavior language,
(2) embedded statechart behavior models,
(3) handwritten C++ code, integrated using the generation gap pattern [18], or
(4) subcomponents (i.e., other components) that are instantiated and connected to each other.

Components using subcomponents are called "composed components," and components without subcomponents are called "atomic components." Similar to **object-oriented programming (OOP)**, there is a distinction between component types (≈classes in OOP) and component instances (≈objects in OOP). Component types can be instantiated as subcomponents as often as needed and support configuration through instantiation parameters. The connection between ports of two components takes place at the instance level, i.e., the ports of two component instances can be connected to each other.

Ports can also describe the communication with external entities not generated by MontiThings, such as driver software of a sensor or actuator. To communicate with an external interface, developers must provide corresponding handwritten glue code that accepts and processes messages from the MontiThings architecture (in case of an outgoing port) or provides messages to the MontiThings architecture (in case of an incoming port). Out of the box MontiThings supports Python and C++ for this. MontiThings' type system for ports and variables provides primitive types similar to Java or C++, supports complex types in form of class diagrams, and supports using types from the **international system of units (SI)**.

MontiThings offers both a graphical and a textual syntax. The graphical syntax is only used for easier comprehension; only the textual models are used for code generation. Throughout this article, we use the graphical syntax for easier comprehensibility. Figure 1 shows a fire extinguisher application demonstrating this syntax. When smoke is detected or the temperature rises above 60°C, the `FireDetector` goes into its `Evacuate` state and sends a message on its `warn` port to turn on the `Alarm`. It will turn on the `Sprinkler` by sending a message on the `extinguish` port only if `smoke` is detected. The fire alarm is ended if no smoke is detected and the temperature falls below 40°C.

MontiThings takes models that can be accompanied by handwritten C++ code as input. After loading the models and creating their **abstract syntax tree (AST)**, MontiThings may apply various model-to-model transformations to add extra functionality, such as recording modules [25], to the models. Then, an Apache Freemarker-based[1] model-to-text generator creates C++ code from the resulting AST. This C++ code executes the behavior specified by the models and is intended to be executed by IoT devices. In this process, the generator also adds the necessary communication code that enables components to exchange messages even when being deployed to different devices, e.g., using **message queue telemetry transport (MQTT)**. Accordingly, the applications created from MontiThings models can be freely deployed to devices. Additionally, MontiThings creates numerous scripts that can be used for (cross-)compiling the code or packaging it as Docker images. Using the infrastructure presented in this article, these Docker images are deployed to IoT devices. The deployment method presented in this article is not bound to MontiThings but only requires these container images and (optionally) technical requirements, such as a specific sensor, for each container image.

MontiThings can be extended with various services such as recording and replaying executions of an application for error analysis [25], synthesized digital twins in enterprise information systems [26], or machine learning [5]. A more detailed description of MontiThings can be found in Reference [27].

3 MOTIVATING EXAMPLE

Currently, IoT applications are often sold as bundles consisting of a piece of hardware and preinstalled software by the same vendor. Future IoT applications will require very flexible deployment strategies. It is expected [10] that future IoT applications will be distributed in *app stores* similar to the app stores known from smartphones. This, naturally, means that the strong coupling between hardware and software will be loosened. These app stores will enable vendors to market software independent of the hardware. Accordingly, they need deployment algorithms capable of deciding which devices shall execute which software. This challenge is complicated by the fact that hardware owners often have their own ideas about which devices should (not) run which software. Moreover, the systems must be able to handle failing hardware. In addition, as customers often continue to expand their systems over time, IoT systems must be able to integrate new

[1] Apache Freemarker project website: https://freemarker.apache.org/.

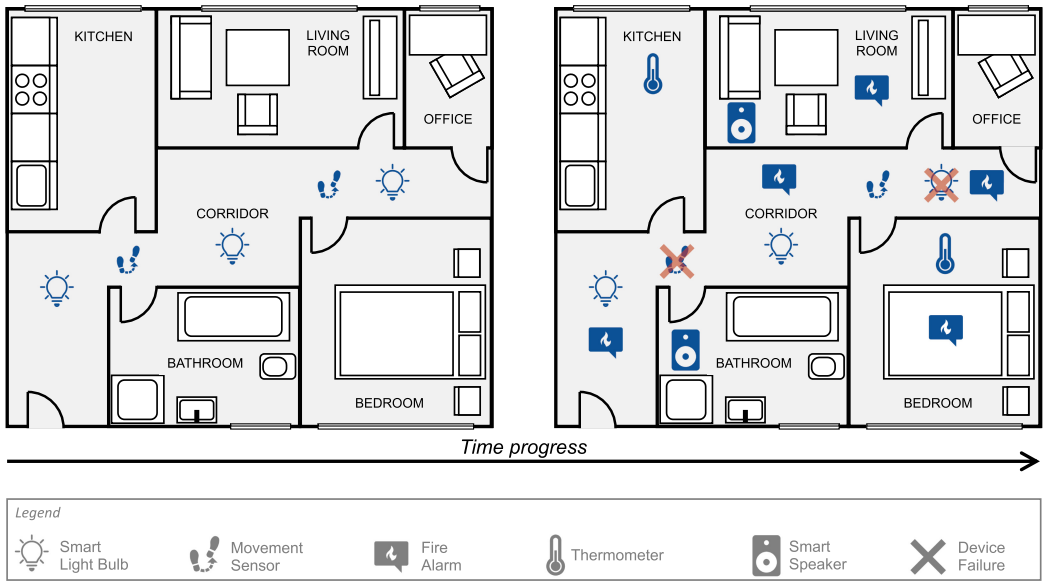


Fig. 2. A smart home application over its lifetime. Inhabitants buy new devices as they continue to build up their smart home. Other devices fail and do not get replaced immediately after failing.

hardware into the (software) system. Since the number of IoT devices in such a system can quickly grow large, IoT devices "must be managed en masse instead [of] receiving personal attention and care" [36].

Figure 2 demonstrates these problems using a smart home application. Initially, the smart home's residents only equipped their apartment with smart lighting control for the corridor. As soon as the motion sensors detect movement, the light may be switched on dimmed, depending on the time of day. After a certain period of time without any detected movement, the light is switched off again. Once the residents were satisfied with these devices, they kept adding to their smart home, such as a smart speaker (e.g., Amazon Echo, Apple Homepod) in the living room or smart fire alarms (e.g., Google Nest Protect). A smart deployment system needs to react to these changes in the available hardware by equipping the new devices with the appropriate software. Such hardware purchases are induced by the current systems' inability to solve some of the users' requirements. For example, the state of North Rhine-Westphalia, Germany, requires that a fire alarm must be installed in every room where people sleep, in corridors, and in children's rooms.[2] A smart deployment system shall support its users in finding deployments that fulfill their requirements.

Some of these devices come with many different hardware capabilities that could also be used to implement other applications. For example, a smart fire alarm, like the Google Nest Protect, might also include LEDs that can be used to (dimly) light a room. Alas, one of the movement sensors and one of the smart light bulbs failed over time. Since the fire alarm includes similar hardware

[2]§47 (3) Landesbauordnung 2018 – BauO NRW 2018. [German, Online]. Available: https://recht.nrw.de/lmi/owa/br_text_anzeigen?v_id=74820170630142752068.
Translation: "In apartments, bedrooms and children's rooms as well as corridors over which escape routes from common rooms lead must each have at least one smoke alarm. These must be installed or mounted and operated in such a way that smoke from a fire is detected and reported at an early stage. The person directly in possession of the smoke alarm must ensure that it is ready for operation, unless the owner assumes this obligation himself/herself."

to the smart light bulbs, a smart deployment system may choose to redeploy the software that was previously executed on the smart light bulbs to the fire alarms. This capability of adapting to changing conditions is referred to as *self-adaptation* [13]. It is desirable from both an economic as well as an ecological point of view that future systems provide this functionality to reduce the carbon emissions caused by unnecessarily replacing devices whose functionality could be provided by the devices already deployed.

While the capability of the devices to execute software from different vendors is crucial to providing an app store, some users may not be comfortable with certain software functionalities being executed on certain devices. For example, even though smart speakers may include a camera in some cases (e.g., Amazon Echo Look) users might not be comfortable giving all software vendors access to a camera in their bathroom. A smart deployment system must empower its users to clearly specify rules that define which kinds of software deployments are (not) allowed.

4 REQUIREMENTS AND ASSUMPTIONS

Motivated by the smart home application of the previous section, we will now formulate the requirements for a model-driven deployment system for IoT applications. Most importantly, the system needs to be able to automatically calculate deployments and adapt to changes in the infrastructure:

(R1) Self-adaptive Deployment. *The deployment shall automatically adapt to changes in the hardware the software is deployed to.* IoT systems are highly dynamic, i.e., new devices become available at runtime and existing devices are removed from the system [13]. The system must be able to detect such changes and decide which software components shall be added or removed to adapt to the changes in the available hardware.

(R2) Modeling Dynamics. *The modeling language shall enable modeling how to react to a changing set of software available at runtime.* The software of IoT systems and C&C languages often relies on the interaction of multiple components, instead of consisting of independent components. For C&C languages, this especially means that components instantiated or removed at runtime need to be integrated in the system, i.e., they must be enabled to start or stop exchanging messages with the other components of the system.

(R3) Requirement-based Deployment. *The deployment system shall make the decision on which device executes which software (semi-)automatically based on a set of requirements.* Due to the possibly large number of devices in IoT systems, they need to be managed "en masse" [36]. It is not desirable to manually specify for each device which software it is supposed to execute. Moreover, manually configuring each device would contradict a self-adaptation **(R1)** in case of hardware changes.

In some cases, human operators may specify requirements that cannot be fulfilled, e.g., because the system does not provide the necessary hardware to fulfill all requirements. Aborting simply with an error message can frustrate users and does not truly help them solve the underlying problem. A more user-friendly way of failing is to propose modifications. This is expressed by the following two requirements:

(R4) Propose Requirement Modifications. *If the requirements cannot be fulfilled, then the deployment system shall be able to propose modifications to the requirements.* One way of solving unfulfillable requirements is to adapt the requirements so they become fulfillable. This means that users have to deviate from their initial wishes. Thus, the system can propose changes and ask users if they accept them. However, the system may never silently apply such modifications. Otherwise, the user's requirements would be meaningless.

(R5) Propose Hardware Modifications. *If the requirements cannot be fulfilled, then the deployment system shall be able to propose modifications to the infrastructure it is supposed to deploy the software on.* Another way of solving unfulfillable requirements is to adapt the infrastructure so it fulfills the requirements. As available hardware can always be ignored without having to ask users to make changes, such adaptions can be reduced to proposing users to buy new hardware to extend their system. Compared to **(R4)** this enables the system to fulfill the users' original requirements but of course requires users to spend money on new hardware. Thus, some users may prefer not to buy new hardware.

By combining proposals to modify both requirements and hardware, users have more flexibility in deciding how much money they are willing to spend to fulfill their requirements.

To focus on the deployment process and algorithm in this article, rather than low-level details of copying software to IoT devices, we continue to make the following assumptions:

(A1) Linux and Container Engine. *The IoT devices execute Linux and a Docker-compatible container engine.* This assumption implies we focus on edge devices such as the Raspberry Pi instead of microcontrollers such as Arduino or ESP32. In particular, we assume hardware similar to those required by AWS Greengrass [3] and Azure IoT Edge [30]. Future IoT applications are expected to rely increasingly on container technologies [36]. This requirement allows us to abstract from the low-level technical challenges that arise, e.g., from the fact that it is difficult to remotely flash an Arduino with new software.

(A2) Internet access. *The IoT devices have a direct Internet connection.* With emerging mobile communication technologies such as LTE-M or 5G, more and more devices have Internet access. By making this assumption, we avoid the need to cache and forward software and commands via gateway devices.

5 DEPLOYMENT WORKFLOW OVERVIEW

As shown in the motivating example, the deployment of dynamic IoT applications is not only influenced by the applications' developers. For example, the inhabitants of a smart home might have additional requirements such as not deploying camera software in the bedroom. In current systems, they express these requirements by placing IoT devices according to their requirements. As IoT software becomes more independent of specific pieces of hardware in future IoT app stores, they will need to be capable of also expressing such requirements through software. Our deployment method relies on four roles that are involved at different times:

(1) The **global manager** (human expert; adapted from Reference [39]) is a member of the engineering team developing the architecture of the IoT applications and the IoT devices. From a software development point of view, global managers specify which technical requirements the components place on the devices that run their applications. For example, a global manager can specify that a component needs a certain (type of) sensor or actuator. From a hardware development point of view, global managers specify capabilities the IoT devices have;
(2) The **CI/CD pipeline** (software system), which takes care of building, testing, and packaging the software;
(3) The **local manager** (human expert; adapted from Reference [39]) owns and maintains the IoT devices. Local managers are, e.g., the facility manager of a smart factory or the inhabitants of a smart home. They do not necessarily have an engineering background. They decide which device to place where and which software to run on their infrastructure. They may have requirements about which devices are allowed to run which (parts of the) software such as not executing video recording applications in their bedroom; and

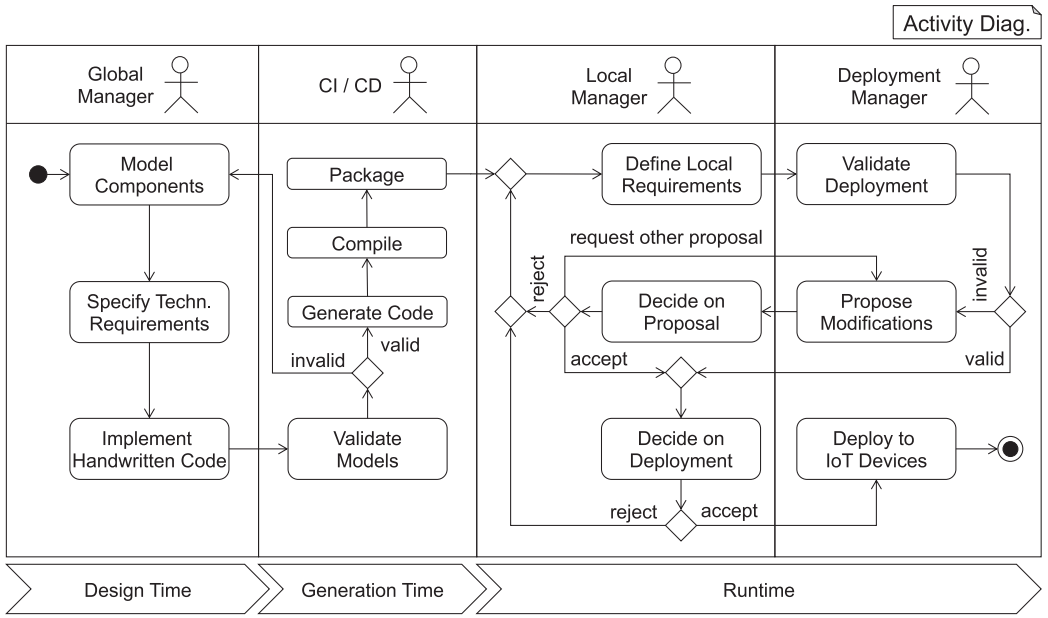


Fig. 3. Overall deployment process. Global managers develop the software and define their technical requirements. After building the software with a CI/CD pipeline, local managers may apply additional local requirements specifying which devices shall execute which software.

(4) The **deployment manager** (software system) coordinates the deployment of software components to IoT devices. In doing so, it must take into account the requirements of the global and local managers as well as the available IoT devices. The result of its calculations is either a mapping that assigns software components to IoT devices or (if such a mapping cannot be found) modification proposals that tell the local manager how requirements that cannot be met could be fulfilled.

The *user* of the software is not involved in the deployment process. Users only interact with the system when it already has been deployed according to the local manager's requirements and technical requirements. Parts of the deployed software may, of course, contain software that is intended for interaction with the user. Thus, the deployment may influence how the user can interact with the system.

Regardless of the subsequent deployment, global managers are responsible for the development of the IoT devices. In doing so, they must specify what capabilities each device has. To do this, they can assign key-value pairs to each device that specify the hardware available on the device. We use the tagging functionalities of the deployment technologies for this purpose. For example, in the case of Kubernetes, nodes can be tagged with labels, while Microsoft Azure's IoT Hub Device Twin uses JSON to tag devices. In commercial implementations of IoT app stores, we expect the device manufacturers to set such capabilities of their devices already in the factory as part of the firmware. This would be in line with the process by which certifications such as *Works with Apple HomeKit* or *Azure Certified Device*[3] already indicate to customers today that IoT devices are compatible with certain ecosystems.

[3]Azure Certified Device program: https://www.microsoft.com/azure/partners/azure-certified-device.

The deployment process (Figure 3) starts with the development of the software. IoT applications are characterized by the fact that they connect everyday objects with each other and with the Internet. Therefore, an important part of the development is the interaction with sensors and actuators. Since not every device has every type of hardware, however, it follows that not every software component can be deployed on every device. Such technical requirements are specified by the global manager. As ontologies of the technical infrastructure are often project-specific, the term "technical requirement" is deliberately underspecified so it can be used as an extension point for models of such requirements, e.g., the feature models proposed in Reference [11]. For simplicity, we assume the technical requirements to be key-value pairs such as `sensor: ''DHT22''`. Based on the technical requirements, the global manager can also add handwritten code to the components. Such handwritten code can be used, e.g., to interact with the drivers of a specific sensor that was set as a technical requirement in the previous step. The global manager uploads the software to an online repository such as GitHub or GitLab, which triggers the CI/CD pipeline.

Initially, the CI/CD pipeline checks if the architecture models are indeed valid models of the ADL. If the models are invalid, then the pipeline fails with an error message and the global manager has to resolve all errors. If the models are valid, then the pipeline generates code from the models, (cross-)compiles the software, and packages it in images for different target devices. We generate a Docker image for each component. At generation time, it is not yet decided which device later shall execute which software. By generating a separate Docker image for each component, this process offers maximum flexibility in choosing which device shall execute which software. We minimize the overhead introduced by this approach by generating the Dockerfiles so MontiThings' dependencies are installed before the generated software is added to the image. Thereby, the layers of the image that install the dependencies can be cached by the IoT devices and shared across components. In some cases it is not desirable to deploy the subcomponents of a component independently. For example, a global manager may want all subcomponents of a `Refrigerator` component, e.g., the fridge light, to be deployed to the same device. If MontiThings was allowed to deploy the fridge light independently of the refrigerator, then it could also choose to deploy its software to the ceiling light instead. Deploying all components independently may also introduce considerable communication overhead, e.g., if the subcomponents only sum up values. In these cases, global managers may choose to prevent MontiThings from deploying subcomponents independently from the component that instantiates them to reduce communication. Note that code generation is optional. The only prerequisite for our deployment method is that container images are created as a result, which can be related to requirements. These container images are stored in a registry such as Docker Hub.

The application can then be instantiated by local managers who decide to deploy them on the IoT devices they manage. We do not assume that these local managers have the same technical background as the global managers involved in software development. Therefore, we deliver an automatically generated web application that serves as a frontend for the deployment manager for the local managers. Using this web application, the local managers can define *local requirements* about which software should be deployed to which devices **(R3)**. Each device also has a *location* that can be used by the requirements. In our prototypical implementation, this location consists of three strings containing a building name, floor name, and room name. Technically, however, the location can contain arbitrary location names. We choose this encoding to be able to provide a more visually appealing user interface in the web application. It is the responsibility of the global managers to divide up the software in such a way that the software components to be deployed are comprehensible to the local managers. Local requirements, unlike technical requirements, make no statement about the technical characteristics of IoT devices. Instead, local requirements describe how the software shall be distributed to the physical environment. For example, a local

requirement may prevent recording software from being executed in the users' bedroom for privacy reasons even if the bedroom contains the necessary hardware.

Moreover, unlike technical requirements, local requirements are negotiable, i.e., the deployment system may propose modifications to the requirements if they cannot be fulfilled. The local manager is, however, free to reject the proposed modification. While there is no technical limitation that keeps us from making technical requirements negotiable, we choose to make only local requirements negotiable, because the global manager is not involved in deploying software to the local infrastructure. We want only the person who has established a particular requirement to be able to relax that requirement, because we expect them to know best why this requirement exists. If technical requirements were negotiable, then the local manager would be allowed to relax requirements set by the global manager. Therefore, only local requirements are negotiable. Since the local managers are familiar with the environment they manage, they can specify considerably better requirements than a global manager could. For example, a local requirement could be to deploy a fire alarm to each room. Overall, our prototypical implementation supports the following four kinds of local requirements:

(1) A component may only/not be deployed at a specific location.
(2) A certain (minimum, maximum, or exact) number of instances of a software component at certain locations.
(3) A component instance requires at least a specified number of instances of a different component.
(4) Two software components may not be executed on the same device.

Properties of the location, i.e., the room, floor, and building, can also be set using quantors known from predicate logic. The quantors offered in the form of the *ANY* and *EVERY* keyword can express, e.g., that a component shall be deployed to *any room on the third floor*. To more easily apply requirements to all subcomponents of a component instance, a wildcard (*) can be used in the instance names. This wildcard matches arbitrary text.

Human errors and misunderstandings of the software, its deployment requirements, and available devices may produce configurations in which the IoT architecture is not deployable under the given constraints. Our algorithm recognizes this and can offer two types of proposed solutions in this situation. The first is to suggest adjustments to the local requirements themselves **(R4)**. The second is to suggest adjustments to the real-world infrastructure **(R5)**. If, for example, two fire detectors are required to be installed in the kitchen, even though only one fire detector exists there, then the solution could be either (1) to relax the requirement so only one fire detector is to be deployed or (2) to install another fire detector device in the kitchen. This is achieved by translating the requirements to according Prolog code. Prolog then calculates possible counterproposals via backtracking. This is described in detail in the following sections. If the local manager is not satisfied with the deployment manager's proposal, then they can request a different proposal or stop the deployment, go back, and adjust the local requirements manually.

There are also cases where human errors cannot be detected because all requirements entered by the local manager can be fulfilled. The most obvious mistake is entering wrong numbers in the requirements, e.g., requesting three fire alarms instead of four and thus deploying the software to fewer devices than intended. In this case, our system will try to fulfill the given requirements without questioning them. We try to mitigate this class of errors by offering quantifiers. This way, the local managers can express their intention directly, instead of having to remember the exact number of rooms. Otherwise, when remembering the number of rooms, the local manager could remember the number of rooms incorrectly, which could lead to an unintentional deployment. For

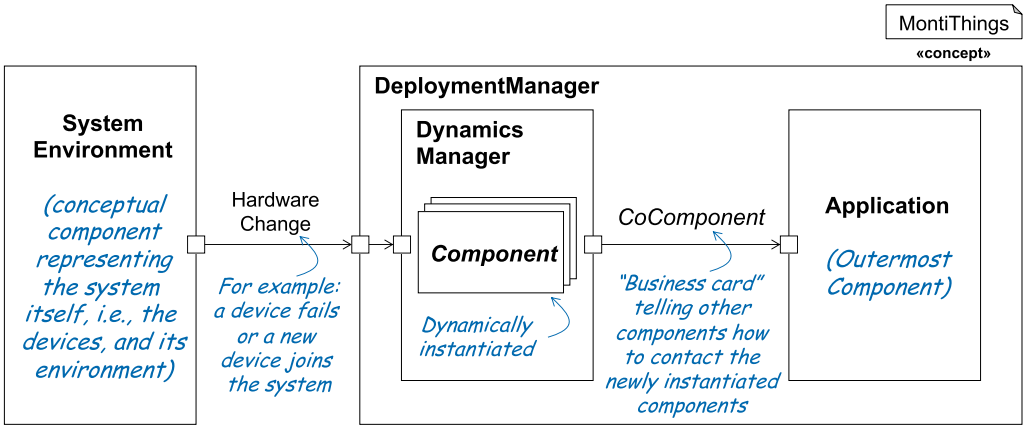


Fig. 4. Conceptual overview for adapting application-behavior to hardware changes in the environment at runtime.

example, the local manager can request a component to be deployed to *all* rooms on the third floor instead of *six* rooms.

Another class of human errors results from the combination of requirements. Even if each requirement makes sense on its own, the effects of their combination may not be intentional. For example, the local manager may require a heater to be deployed to each room, a temperature measurement to be deployed to each compatible device, and also that devices with temperature measurement not be run on the same device as the heater to prevent mutual interference. However, if each heating device also has a built-in temperature sensor, then the requirements can not be fulfilled together. These errors are generally difficult to detect for the user. Often they result in the system deploying considerably less software than intended.

Once all requirements are applicable, the local manager can decide to actually deploy the software to the IoT devices. The devices are then informed about which software they should execute. The devices then download this software and begin execution. The local manager of course has the ability to stop the execution of the software at any time. In our prototypical implementation, we assume that the IoT devices have a direct Internet connection. Conceptually, it is also envisionable that gateways forward the software to connected devices without direct Internet access. This technical limitation is on par with **Amazon web services (AWS)** Greengrass, which can also only push Lambda functions to Internet-connected devices running the AWS Greengrass Core.

6 DYNAMICALLY INTEGRATE COMPONENTS INSTANTIATED AT RUNTIME

Dynamic reconfiguration in MontiThings enables systems to react and adapt to changes in their context or hardware within a predefined framework. The language provides explicit mechanisms and constructs to adapt systems and to model possible configurations and reconfiguration triggers, which then define the framework in which the system can adapt to changes **(R1)**, **(R2)**.

A configuration here means the topology of subcomponents of a composed component. Using dynamic reconfiguration, this topology can be modified at runtime of the system, i.e., components and connectors can be added or removed. The instantiation or removal of components always depends on underlying changes in the available devices. Accordingly, components cannot be directly instantiated using the modeling language but are only instantiated by the deployment manager. Instead, MontiThings enables architects to model how the software reacts to a component being removed (e.g., because the underlying hardware fails) or added (e.g., because new hardware is integrated).

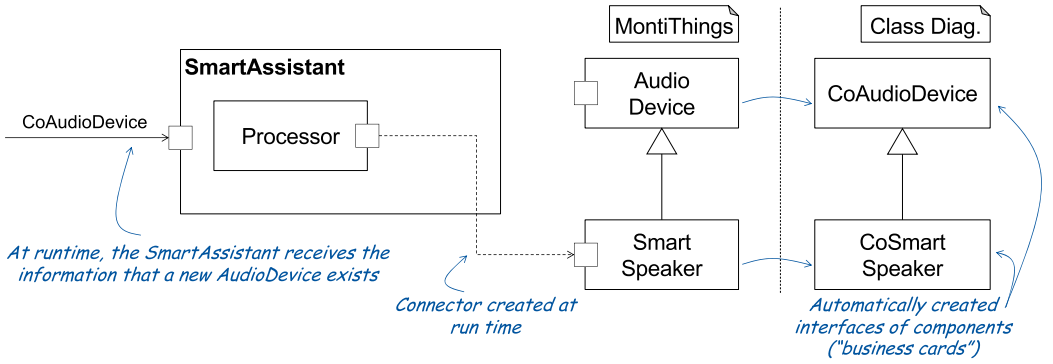


Fig. 5. At runtime, components may receive information about new components. This information contains (part of) the interface of the new component.

The extent to which a system can adapt to changes is thus determined by the degree to which the software can react to the addition or removal of components. How an externally added component is integrated into a topology of subcomponents is predetermined at design time. A component is informed about changes in its context via its interface consisting of ports. When a component is added to the system, it reports to the outermost component that represents the application. For this purpose, the added component sends a kind of business card that describes how other components can connect to it. Each business card describes the address of the component and its available ports. As such, a business card describes (part of) the interface of a component as information that can be sent over communication channels instead of the corresponding component.

For modeling MontiThings component models, a business card is basically an object that encapsulates the aforementioned information. The type of such an object functions as a template that describes the basic structure of some business cards. For each component model, MontiThings provides a type of business card that is specific to the component's interface. In addition, architects may define customized data types describing component interfaces via MontiThings' class diagrams. As business cards conform to data types of MontiThings' type system, components can send these business cards just like regular messages via ports of corresponding types. Consequently, these can also be used to define the type of ports and thus the required interface of dynamically and externally added components. For example, a camera with a built-in microphone could make the microphone available to external systems but not its light sensors.

The deployment manager is the only component in a system than can instantiate software components. That is, software components are instantiated via the deployment manager on the basis of the available hardware components and on the basis of the deployment requirements defined by the system architect. In addition, the deployment manager receives information from the environment about changes in the hardware. Given this information, the deployment manager can instantiate new software components at runtime. When a new component is instantiated or an existing component is removed, this is communicated to the outermost component of the architecture via a message on a port. The message contains the business card of the added or removed components, as well as this status information. The outermost component can then forward this information to its relevant subcomponent(s).

The concept of dynamic reconfiguration based on the deployment manager is shown in Figure 4. An IoT system consists of a deployment manager and an application. The former is generic for all IoT systems developed with MontiThings; the latter is modeled using MontiThings' architecture description language. The deployment manager receives information about hardware

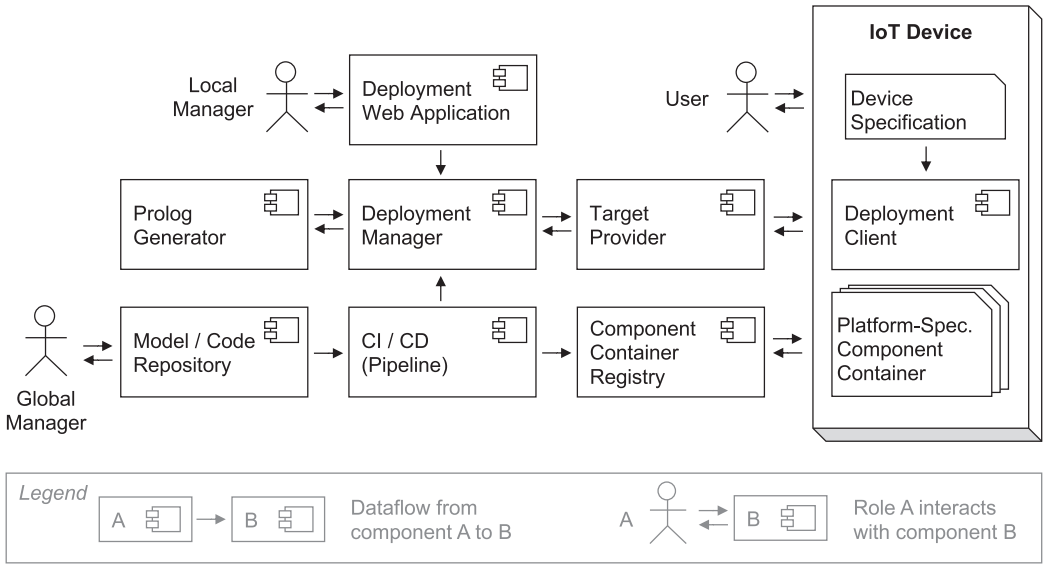


Fig. 6. Overview of the architecture of the interactions between the components of our deployment framework.

changes from the environment of the IoT system and can instantiate components based on this information. Information about these components is sent to the application via the deployment manager interface.

An example of this is depicted in Figure 5. The smart assistant may incorporate new audio devices at runtime. Information about such devices is provided by its environment via a dedicated port. This port is typed by a corresponding data class that describes the common interface of audio devices via which the smart assistant can connect. If a new device is available in the environment of the smart assistant, e.g., a smart speaker, then the smart assistant can connect its subcomponent with the provided interface of the audio device.

How a system reacts to the removal or addition of components is application-specific and is modeled by the behavior of the components. To this end, MontiThings offers a behavior description for composed components with which component developers can model how subcomponents delete existing connections or create new connections given a reconfiguration condition. In such a behavior description, composed components can read from ports to react to messages and reconfigure the system but explicitly cannot write on outgoing port to distinguish composed components from atomic components. The output behavior still results from the decomposition of the subcomponents.

7 SYSTEM OVERVIEW

As explained above, components are only instantiated as part of the deployment. This section outlines our deployment method. Figure 6 shows the basic architecture of our system based on the deployment process in Figure 3. The local manager interacts with the system via the frontend of a deployment web application. This web application is generated by MontiGem [1, 21], a tool for the model-driven development of web-based information systems. Using this web application, the local manager can manage devices, define local requirements, respond to proposals, and request to start or stop the deployment. The web application consists of a front- and a backend. The

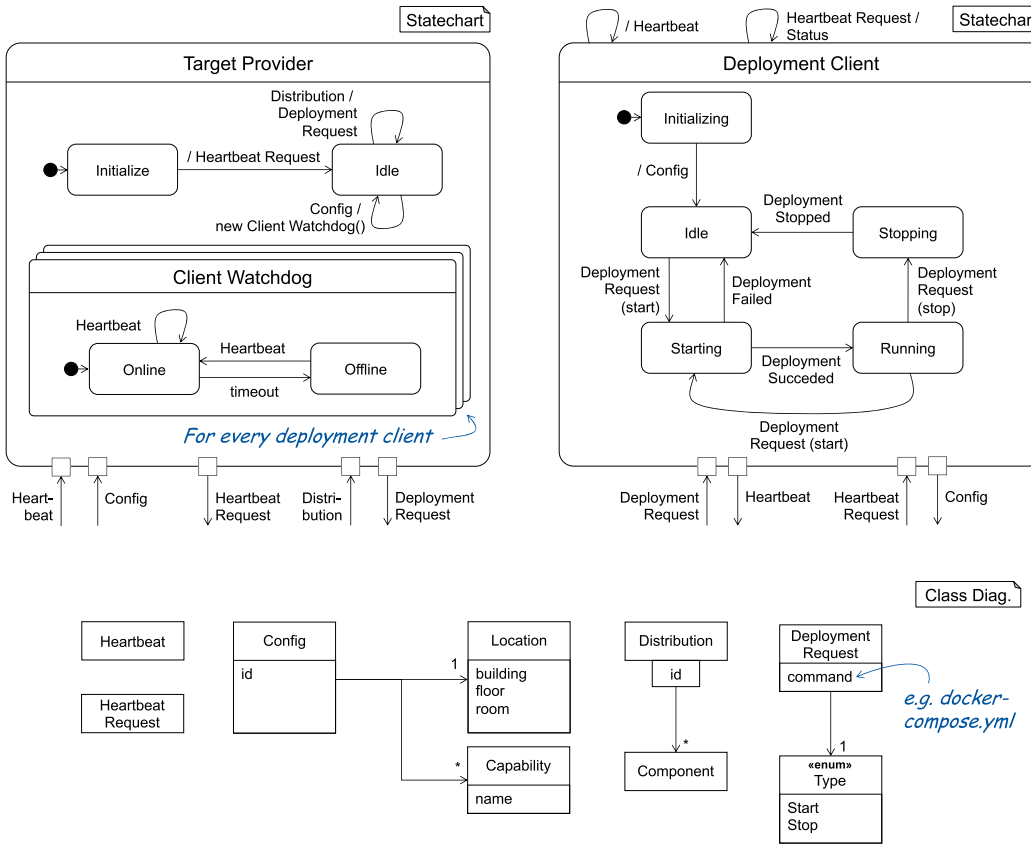


Fig. 7. Interaction between the deployment clients and target providers. Using heartbeat messages, target providers track for each client if the client is online or offline. Deployment clients provide their configuration to the target provider. Target providers may request the client to start or stop the execution of software (containers).

backend persists the local manager's requirements as well as the state of the system. Furthermore, the backend handles the communication with the deployment manager, i.e., forwards requests to calculate deployments, proposals in case of unfulfillable deployment requests, requests to execute the deployment when a valid deployment was found, and requests to stop the application.

The main task of the deployment manager is to coordinate communication between other system components. The deployment manager receives the local manager's requests and requirements from the backend of the web application and information about the IoT devices from the target providers. When the deployment manager receives a request or gets notified about a change in the infrastructure of IoT devices, it triggers a (re-)deployment. For this, the deployment manager uses a standalone Prolog generator that generates Prolog code from information about the system.

The Prolog generator is based on Apache Freemarker templates. These templates define the strategy used to make modification proposals, i.e., which alternative local requirements to try if a requirement cannot be fulfilled. For example, if a local requirement requests a certain number of components, then one strategy is to count that number down to zero until the requirement becomes satisfiable. The generated Prolog code calculates deployments and makes specific suggestions based on the strategy set by the templates. Accordingly, the Prolog generator offers two methods that both return a string containing Prolog code. The first translates the facts about the

available deployment clients into Prolog facts; the second generates a query calculating which deployment client shall execute which software components. This generation is described in the next section in more detail.

Target providers provide the deployment manager with interfaces to distribute software to the IoT devices via various technologies. To do so, target providers have a counterpart that is executed on the IoT devices: the deployment clients. Figure 7 shows the interaction between the target providers and deployment clients. After an initialization phase during which target providers set up necessary network connections, e.g., to an MQTT broker, and request a heartbeat from the clients, target providers become idle. In idle mode, target providers wait for messages from either the deployment manager or the deployment clients. The deployment manager may send target providers a new distribution. A distribution is a multimap that specifies which devices shall execute which software components. When receiving a distribution, the target provider is responsible for ensuring that all its managed deployment clients are updated to match the given distribution. To do so, it will send deployment requests to the individual deployment clients. These deployment requests contain the necessary commands to start the requested software components. For example, a deployment request can consist of a `docker-compose.yml` that tells the deployment client which images to pull from which container registry and with which parameters to execute them. It is also possible that the target provider specifically requests a deployment client to stop all of its containers. When a previously unseen client announces its presence by sending its configuration to the target provider, the target provider starts tracking its availability using a watchdog. This configuration contains the necessary information for calculating deployments, i.e., a unique ID of each device, its location, and the capabilities of the device (`Config` class in Figure 7). The capabilities in this configuration are the software and hardware capabilities that can be referenced by technical requirements of components. Target providers will forward this information to the deployment manager. Initially, each client will be considered online. It is considered online as long as it regularly sends heartbeat messages to the target provider. If the target provider does not receive a heartbeat message within its timeout period, then it will consider the client offline. Offline clients are ignored when calculating deployments. The client can become online again by continuing to send heartbeat messages. The deployment manager is subscribed to these status updates using the observer pattern [20]. Therefore, it can recalculate the deployment if the availability of clients changes or new clients join the system.

Overall, the clients handle the communication with the target providers and execute the target providers' requests to start or stop the execution of certain software. We distinguish five different states:

(1) *Initializing*, for clients setting up their connection to their target provider and registering with the target provider using a unique ID before becoming idle. In particular, the clients also read a device specification in this phase. The device specification contains the config and the necessary information to communicate with the target provider, e.g., the IP address and port of an MQTT broker;
(2) *Idle*, for available clients not executing any software component;
(3) *Starting*, for clients that have received a request to execute a software component but have not yet finished initializing them;
(4) *Running*, for clients currently executing software components; and
(5) *Stopping*, for clients that have received a request to stop their components but have not yet executed that command.

Because the client's main task is to execute containers, we chose these states to be very similar to the states of Docker containers.

Overall, the deployment clients are responsible for ensuring that the platform-specific component containers requested by the deployment manager are executed on the device. The images in this registry are created as described in the previous section by the CI/CD pipeline using the input from the global manager. In our prototype implementation, we support four different target providers and clients:

(1) Docker Compose,[4]
(2) Kubernetes,[5]
(3) GeneSIS [15–17], and the
(4) Microsoft Azure IoT Hub.[6]

For Docker Compose, we implemented a small Python client that receives `docker-compose.yml` files from the deployment manager and executes them on the IoT devices. For Kubernetes, we assign *labels* to nodes (which represent IoT devices) to assign them properties such as a specific location or available hardware. For GeneSIS, we use their SSH-based deployment to send docker commands to the devices. We do not use their Docker-based deployment, because it requires a specific naming scheme our container images do not meet. For Azure, we use their IoT Edge[7] container solution for deploying Docker containers to IoT devices. To do so, the target provider generates deployment manifests[8] that specify which container images to deploy for each deployment client and forwards them to Azure using their REST API. In particular, the type of target provider also determines the registration process. If, as in the case of Microsoft Azure, its own device management is offered, then the target providers only serve to query its information. In this way, the scalability of automatic registration services, such as the Azure IoT Hub Device Provisioning Service,[9] can be leveraged. In the case of our self-implemented device management, e.g., our Docker Compose Python client, each device automatically registers with an automatically generated UUID. As IoT devices should be managed en masse [36], we prefer automatic over manual registration processes. If there are reasons to find individual devices in a commercial implementation of an IoT app store, then the hardware manufacturer could print the respective ID on the device, for example, or the web app could be given the option of letting a device play a sound to make itself known (like Apple's AirTags).

For creating the container images, we use MontiThings as the ADL for specifying our IoT applications. MontiThings can be used to automatically create Docker images from the individual modeled components. Using Docker Buildx,[10] we cross-compile these in the CI/CD pipeline for different target architectures. These multi-platform images are made available to the devices via a standard Docker container image registry.

8 PROLOG DEPLOYMENT ALGORITHM

Our Prolog generator receives information from the deployment manager describing facts and requirements. The facts are for each device the state of the device (online/offline), as tracked by the target providers, and their configuration, i.e., its capabilities (e.g., available hardware), its location,

[4]Docker Compose documentation: https://docs.docker.com/compose/.
[5]k3s project website: https://k3s.io/.
[6]Azure IoT Hub website: https://azure.microsoft.com/de-de/services/iot-hub.
[7]Azure IoT Edge website: https://azure.microsoft.com/de-de/services/iot-edge/.
[8]Azure Documentation: "Learn how to deploy modules and establish routes in IoT Edge": https://docs.microsoft.com/en-us/azure/iot-edge/module-composition?view=iotedge-2020-11.
[9]Azure IoT Hub Device Provisioning Service (DPS) Documentation: https://docs.microsoft.com/en-us/azure/iot-dps/ Last checked: 29.03.2022.
[10]Docker Buildx Documentation: https://docs.docker.com/buildx/working-with-buildx/.

and a unique identifier. The requirements are all technical requirements and local requirements defined by the global and local manager. To make the generation of the Prolog code easier, the deployment manager transforms all local requirements including a quantor using the following two transformations:

(1) The *any* keyword (∃ quantor) is replaced by an anonymous variable, allowing Prolog to match any term;
(2) The *every* keyword (∀ quantor) is replaced by individual requirements for each property that could match the requirement. For example, if a component shall be deployed to each floor of a three-story building, then this creates three individual requirements, one for each floor.

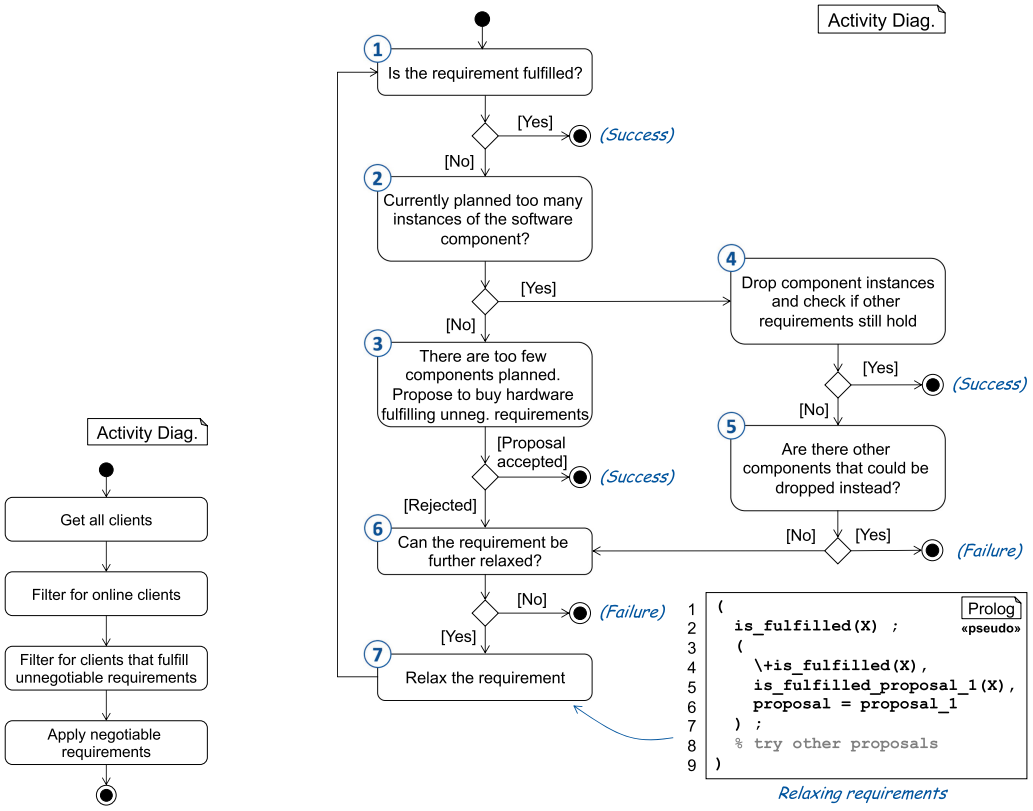
The generator's division into facts and requirements corresponds to the structure of Prolog programs, which also consist of facts, rules, and queries. From this input, the generator generates a program that can be used to evaluate which software artifact can be executed on which device. As Prolog's evaluation strategy is based on backtracking, Prolog is well suited for also computing counterproposals for unmet requirements.

The generated Prolog program essentially consists of two parts. First, we have a set of facts about the IoT devices. These facts always consist of a fact kind, a fact content, and a device identifier. Essentially, these are key-value tags for devices in the form `fact(kind, content, device).` The fact kind specifies what the fact is about, e.g., that a device is in a specific location. The fact content specifies the value of a fact, e.g., the name of the location of the device. While we do not restrict fact kinds to certain values, we use the following fact kinds:

(1) the device is online and available for deployments;
(2) the device has a specified capability, e.g., has a certain sensor;
(3) the device's location.

The second part of our program consists of the queries that calculate deployments. This program consists of a main query for finding the deployment and one side query per container image. The main query calls the queries for the individual artifacts and passes them free variables for the devices and for each of their requirements. Each side query is responsible for finding out to which devices a component shall be deployed. For this, the side queries combine all requirements and local requirements that were defined by the global and local managers for this component and evaluate them on the facts about the devices. There are two possible results to this calculation that are communicated by setting the free variables: If all requirements and local requirements can be fulfilled, then the side query returns a list of devices a component shall be deployed to. Otherwise, the side query produces counterproposals that would fulfill all requirements and local requirements. An important detail to note here is that the code generator decides which counterproposals could exist, but the Prolog interpreter's backtracking mechanism decides which counterproposals to use when evaluating the side queries. To avoid unnecessarily moving software from one device to the other, it is possible to request the Prolog algorithm to use a given deployment as a starting point. In this case, it will first check if it can fulfill all requirements with the given deployment. It will then only make changes if the given, i.e., current, deployment does not fulfill the requirements.

Figure 8 shows the workflow of the generated Prolog code for the side queries. Initially (Figure 8(a)), the Prolog code will start with a list of all clients and filter them for the online ones. From these, the Prolog code then filters the clients that fulfill all non-negotiable requirements, e.g., technical requirements. Using this list of candidate clients, Prolog will then try to find a set of clients that would fulfill all of the local manager's requirements or make proposals if the requirements cannot be fulfilled. Figure 8(b) shows the generated code's strategy for fulfilling a single



(a) Overall process to find the clients to execute a specific software component

(b) Strategy of trying to fulfill a requirement by adapting the deployment or making modification proposals to the local manager. Each "Failure" result causes Prolog to use backtracking to find a different solution. This is part of the *apply negotiable requirements* step of Fig. 8(a).

Fig. 8. Structure of the generated Prolog code for finding which deployment clients shall execute a given software component.

requirement. If there are multiple requirements, then they are in logical conjunction, i.e., all of them shall be fulfilled. For future variants of the system, it is also imaginable to enable users to specify more complex dependencies between requirements, e.g., ask the system to fulfill requirement A *or* requirement B.

Our strategy for fulfilling a requirement starts by first checking if the requirement can be fulfilled by the clients that are already selected for executing the software component (① in Figure 8(b)). If so, then no modifications are needed. If the requirement is not fulfilled, then the strategy differs between cases where there are too many software component instances to fulfill the requirement and cases where there are too few (②). If there are too few components, then the code will propose to buy new hardware that fulfills the non-negotiable requirements and place it at the location specified by the currently evaluated requirement (③). The local manager may choose to either buy the requested hardware, in which case the algorithm can continue later when the hardware was added, or decline buying new hardware. If the proposal to buy new hardware gets rejected by the local manager, then the system will try to make proposals for relaxing the requirements.

If the evaluation of already planned software components resulted in having too many components, then the algorithm will try to remove some of the component instances without violating

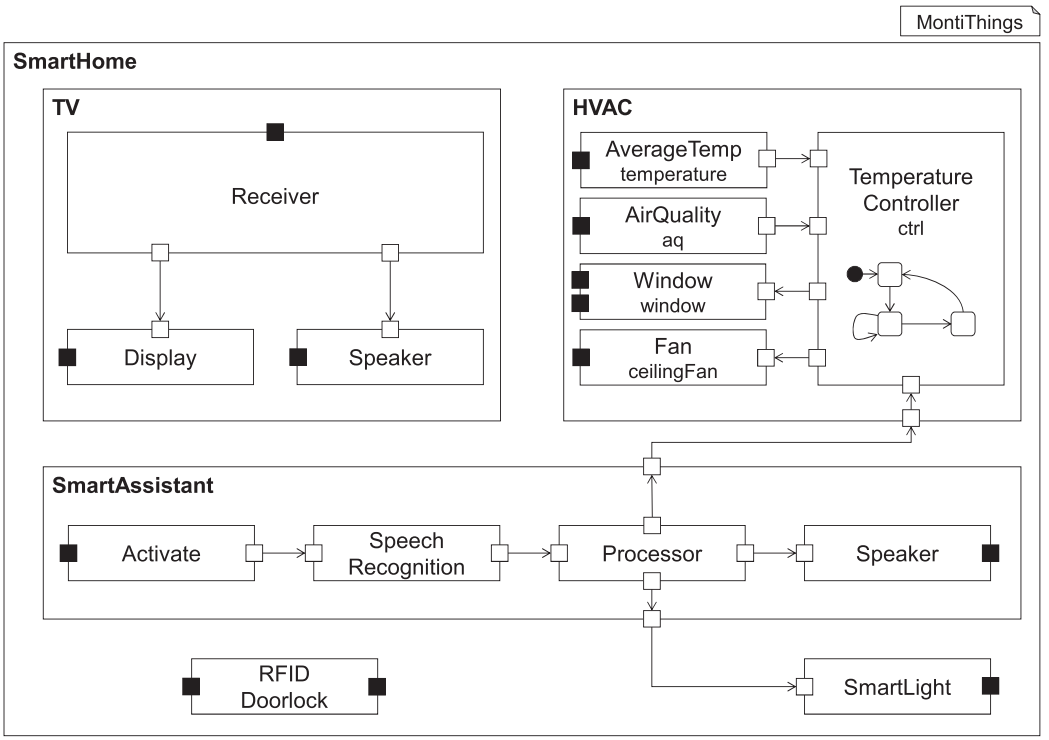


Fig. 9. Architecture diagram of the smart home.

the other, already fulfilled, requirements (④ and ⑤). If it is not possible to remove components, then the algorithm will try to relax the requirement (⑥ and ⑦). The Prolog pseudocode in Figure 8(b) shows how the algorithm handles this: If the query for fulfilling a requirement cannot be fulfilled, then the algorithm allows to *not* fulfill the query (line 4), try to fulfill a relaxed requirement (line 5), and set a free variable to the proposal (line 6) to keep track of which proposals were made during the evaluation of the (backtracking) algorithm.

The strategy may result in either a success or a failure. Ending with a success or failure means that the according Prolog predicate succeeds or fails accordingly. For every failure, Prolog will use its backtracking evaluation strategy and try alternative paths through the evaluation. In the activity diagram, this corresponds to going back through the path that led to the failure and trying different paths at the decision nodes. For example, assume Prolog reached a failure after activity ⑥, which it reached after removing some component instances in activity ④. Prolog could now choose to go back and try to remove different components. This backtracking extends beyond the evaluation of a single requirement, i.e., when a requirement cannot be fulfilled, Prolog will try to change the paths it used to fulfill the previous requirements.

To not overload the local manager with proposals that end up with a failure later in the process, Prolog will assume the local manager accepts all proposals and collect them using free variables while searching for a valid deployment. The proposals collected during this search will only be shown to the local manager if a valid deployment could be found. If the local manager then rejects the proposals, then the backtracking can be triggered to calculate other proposals. If there does not exist a possible path that leads to a valid deployment, then even when using modification proposals, the algorithm will fail. This is the case, e.g., when specifying contradictory requirements.

9 CASE STUDY

To demonstrate the feasibility of our deployment method, this section applies our deployment method to a smart home case study. We model a smart home application that is applied to a smart home and, following the running example from Reference [39], a smart hotel visited by the resident of the smart home. Figure 9 shows the MontiThings architecture created by the global managers to build their application. The application consists of a TV, a **heating, ventilation, and air conditioning (HVAC)** system, a smart assistant (e.g., Siri, Alexa), a digital door lock, and smart lighting. The black ports in the architecture denote that components access external hardware that is not part of the system. For example, the `Window` component controls a window and a fan, and the door lock reads in a **radio-frequency identification (RFID)** card and controls the door lock. Accordingly, the global manager specifies a technical requirement for each black port that requires a matching sensor or actuator. As the specific control of such hardware is not the focus of this article, we specified the components need to be executed on devices with according hardware capabilities but did not implement the drivers that actually access such hardware. Instead of connecting real hardware, the required sensors and actuators were mocked. As our deployment system does not analyze the contents of the deployed software beyond analyzing the models, the deployment system is not aware of the fact that the software does not actually access the hardware requested by the technical requirements. Information on how real sensors and actuators can also be connected with MontiThings can be found in Reference [27].

To deploy the software, the global manager uploads the models and code to a central repository, as described in Figure 6. In our deployment case study, we used GitLab for this purpose. As GitLab offers CI Pipelines and container registries in addition to Git repositories, GitLab can provide the bottom three components of Figure 6 without any modification. It would, of course, have been possible to replace GitLab with a similar service such as GitHub that offers similar services. Since Git, CI pipelines, and container registries are established industry standards and we do not require them to implement software specific to our deployment, the global manager can choose from a wide variety of vendors. The CI pipeline's main task is to build the container images that are later deployed to the IoT devices. To do so, the pipeline first generates C++ code from the MontiThings models using MontiThings' code generator. To cross-compile and package the applications in a container image, we utilized Docker Buildx.[11] We configured Docker Buildx to compile the generated software for the following platforms: linux/amd64, linux/arm64, linux/arm/v7, and linux/arm/v6. The pipeline pushes the resulting multi-arch container images to the container registry offered by GitLab as part of the repository that contains the code. It is of course also possible to push the images to a separate registry, e.g., provided by a cloud like Microsoft Azure or AWS. Overall, that led to a total of 17 multi-arch container images, i.e., one for every component type in Figure 9. Note, that the `TV` and the `SmartAssistant` component use two instances of the same `Speaker` component type.

After the software is available in a registry, it can be used by the IoT devices. We used 10 Raspberry Pi 4 Model B devices in our case study. Seven of them were deployed in Aachen, Germany, and represent our smart home. The other 3 are deployed in Stuttgart, Germany, and represent the smart hotel. Table 1 gives a detailed overview of the configuration of each device. To deploy the software, we first deployed the deployment web application, Prolog generator, and deployment manager on one of our servers using Docker compose. The web application was realized using MontiGem [1, 21], a tool for the model-driven development of enterprise information systems.

[11] Docker Buildx Documentation: https://docs.docker.com/buildx/working-with-buildx/.

Table 1. Overview of the Devices Used in the Case Study

Device	Main Purpose	Capabilities	Location	
			Room	Building
Pi1	TV	speaker, display, tvReceiver	Living Room	Smart Home, Aachen, Germany
Pi2	HVAC	sensorAirQuality, actuatorWindow, actuatorFan	Living Room	
Pi3	Smart Assistant	microphone, speaker	Bedroom	
Pi4	Bedroom Light	light	Bedroom	
Pi5	Bathroom Light	light	Bathroom	
Pi6	Livingroom Controller	sensorRFID, light, actuatorLock, sensorTemperature	Living Room	
Pi7	Soundbar	speaker	Living Room	
Pi8	Hotelroom Controller	light, sensorRFID, actuatorLock	Bedroom	Smart Hotel, Stuttgart, Germany
Pi9	TV	speaker, display, microphone, tvReceiver	Bedroom	
Pi10	HVAC	sensorTemperature, sensorAirQuality, actuatorWindow, actuatorFan	Bathroom	

All devices are Raspberry Pi 4 Model B.

After logging into the web application generated by MontiGem, the local managers can set their requirements. They chose the following set of requirements to deploy:

(1) at least one smart light to every building, every floor, and every room;
(2) exactly one HVAC (and all of its subcomponents) to any room and floor in every building;
(3) exactly one TV (and all of its subcomponents) to any room, floor, and building;
(4) exactly one smart assistant (and all of its subcomponents) in every building on any floor and in any room;
(5) exactly one `SmartHome` component per building.

Figure 10 shows how the local manager entered these requirements into the web system. The four tabs (excluding the *Control* tab), constitute the types of local requirements presented in Section 5. The web application also automatically creates requirements that all composed components require one instance of each of their subcomponents.

To make the local requirements unfulfillable, we did not connect device Pi6 to the system. As Pi6 is the only device in Aachen to offer RFID or a door lock, the `RFIDDoorlock` cannot be deployed without Pi6. After registering the smart home's Docker Compose target provider in the web application, all devices of the smart home but Pi6 (i.e., Pi1–Pi5, Pi7) show up in the **graphical user interface (GUI)**. The local manager now requests to deploy the software to the infrastructure. Internally, this causes the web application to request a deployment from the deployment manager. The deployment manager requests the Prolog generator to provide code according to the devices and requirements. After executing the code and parsing the results, the deployment manager sends an answer to the web application. As the local managers' requirements cannot be fulfilled, the web application informs the local managers that their request cannot be fulfilled but that it would be possible to show modification proposals. The modification proposal consists of not changing any requirements but deploying a new device to the living room (Figure 11). The local manager accepts the proposal. To fulfill it, we add Pi6 to the system and thus the application becomes deployable.

The Prolog algorithm decided to deploy all components without technical requirements to Pi6. The `HVAC`'s and `SmartAssistant`'s subcomponents are deployed to Pi2 and Pi3, respectively. The

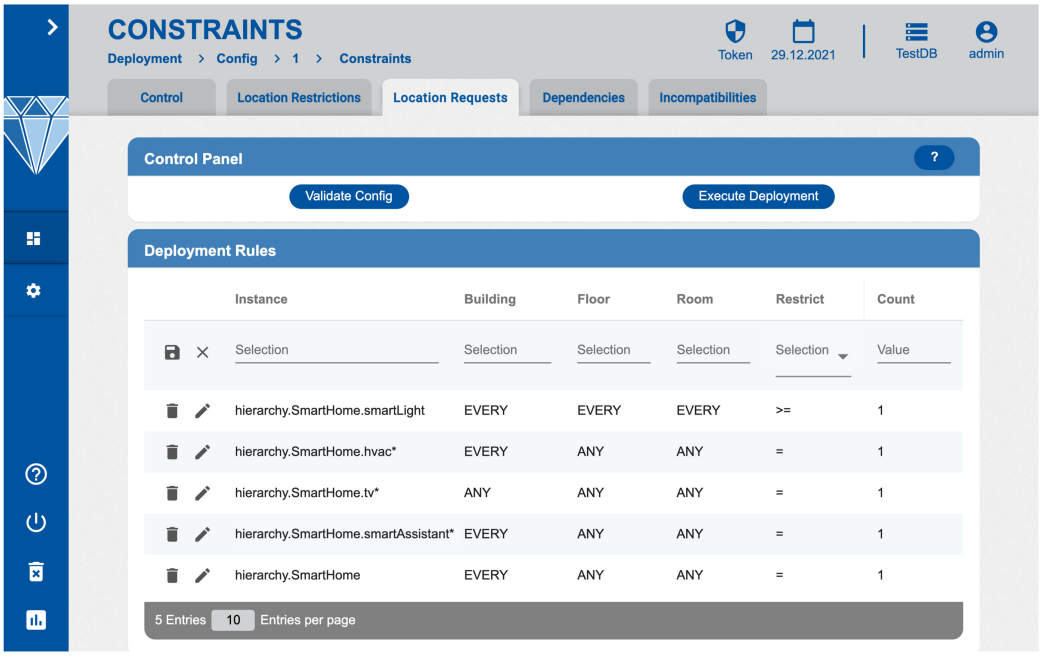


Fig. 10. Screenshot of the deployment web application for entering location requirements. Labels in the screenshot were translated from the originally German web application to English. Irrelevant website elements were removed to save space.

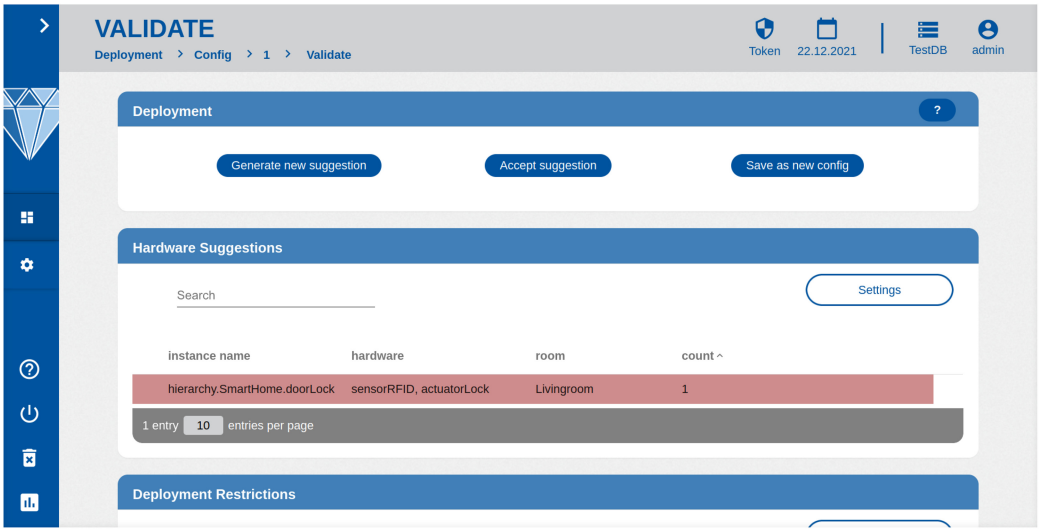


Fig. 11. Screenshot of the deployment web application suggesting to buy a new device with a `sensorRFID` and `actuatorLock` capability and placing it in the living room. Labels in the screenshot were translated from the originally German web application to English.

`Speaker` of the TV is deployed to Pi7 (the soundbar), and the `Receiver` and `Display` are deployed to Pi1 (the TV). Furthermore, the `SmartLight` component is deployed to Pi4–Pi6.

As outlined in Section 3, unfortunately, not all smart home devices are built to last. To simulate this, we now stop the deployment client software on Pi7. After a couple of seconds, the target provider registers that Pi7 is no longer sending heartbeat messages. It informs the deployment manager about the change. The deployment manager regenerates the Prolog facts, to no longer consider Pi7 online. The application is still deployable, because Pi1 also offers the speaker hardware capability. The deployment manager automatically applies the change and the component is redeployed to Pi1.

Next, the resident of the smart home decides to visit a smart hotel. The hotel offers a set of IoT devices that can be configured by the hotel guests to their liking. As in the running example from Reference [39], the hotel guests in this case also act as the local managers. During the journey to the hotel, our local manager adds the target provider of the hotel room to the web application. To save energy while being away, the local manager also removes the target provider from the smart home. The smart hotel consists of three devices, as shown in Table 1. Pi8 acts as the door lock of the hotel room. As common in many hotels, it also controls the light of the hotel room so the lights are turned off automatically when the guest removes the room's key card from a slot in the hotel room. Pi9 is the hotel room's TV and Pi10 is an HVAC system in the hotel room's bathroom.

Unfortunately, when trying to deploy the system using all requirements from the smart home, not all requirements can be fulfilled by the hotel room, because the bathroom does not have the necessary hardware to deploy a smart light but one requirement requires the smart light to be deployed to every room. First, the system suggests to the local manager to buy a new smart light device and install it in the bathroom. As this is only a hotel room and the local manager is only a guest, buying new hardware for the hotel room does not seem appealing. The local manager rejects the suggestion and requests a different suggestion. Next, the system suggests relaxing the requirement so the bathroom may not have a smart light. The local manager accepts this suggestion and the system gets deployed. When they arrive at the hotel, local managers don't have to check in at the hotel, they can go directly to their room because the door lock is already running the software from their smart home and accepting the key card from their smart home.

Overall, the case study shows the feasibility of our approach. By providing modification proposals, our system can cope with the inherent dynamics of IoT systems and make suggestions that help users to adapt their IoT applications to their hardware. There are, however, still a couple of drawbacks to our method that became visible in the case study. The most important issue is that Prolog by design only searches for the first solution that fulfills all requirements, i.e., sets values to all free variables. This, however, means that the deployment may seem unintuitive if the requirements do not match exactly what the local manager was intending to deploy. For example, if a requirement requests *at least* three instances of a component, then Prolog will accept the first solution that deploys the software to *exactly* three devices even if more would be available. To mitigate this issue, we introduce the quantors (*every* and *any*; Section 8) that enable a more natural specification of requirements.

Moreover, Prologs evaluation strategy by design suggests a large number of very similar proposals. For example, if it suggests relaxing requirement A, then relaxing requirement A and requirement B will still be considered a different solution by Prolog. While this makes sense for a computer, this approach is very tedious for a human. Our web application thus automatically discards all proposals that are supersets of proposals that have already been rejected by the local manager.

10 RELATED WORK

MontiThings is a C&C architecture modeling infrastructure for IoT systems that is based on the MontiArc ADL. Aside from an ADL, it features several model transformations, code generators, and various tools, some of which have been reported on in the past [25–27]. In this article, we focus on its smart deployment assistance and, thus, discuss related architecture modeling infrastructures. Other aspects of MontiThings have been described in detail [27].

First, connecting to hardware devices through a variant of ports has already been proposed with **real-time object-oriented modeling (ROOM)** [34, 35]. In ROOM, service access points and service provision points act similar to ports in that they coordinate data exchange between components (called actors in ROOM). These service access and provision points are connected. Besides that, ports represent communication channels between components.

Second, there are modeling frameworks for distributed IoT systems, such as Calvin [4, 32], DIANE [37], Distributed Node-RED [7, 22], GeneSIS [15–17], or CapeCode [9]: Calvin, e.g., is an actor-based modeling framework for decentralized IoT systems that supports the automated deployment in which actors enact the roles of components. Actors have requirements and devices yield capabilities. By matching both, Calvin computes which devices can execute which actors. Additional deployment requirements may further restrict the deployment. With DIANE [37], technical units (components) can be deployed to deployment units (target platforms) automatically if the component fulfills the constraints of the target platform. For both, DIANE features textual modeling languages that describe language, assembly, and execution mechanisms for technical units and constraints, hardware, and deployment steps for deployment units. A deployment generator takes both kinds of units, computes a deployment, and then applies it while taking care of balancing the deployment across multiple servers if necessary. Similarly, Distributed Node-RED [7, 22] enables developers to model IoT architectures and also matches component requirements with device capabilities as well. GeneSIS is a model-based tool for IoT application deployment. It leverages a graph-based deployment model to structure how software artifacts are mapped to IoT devices. If GeneSIS detects a discrepancy between the current state of the system and the desired state, then it calculates the necessary changes to reach the desired state. Compared to MontiThings, GeneSIS is more focused on the technical aspects of deploying artifacts on the IoT devices (i.e., the physical transport of deployed software artifacts). It does not support reasoning about deployment. CapeCode extends Ptolomy [33] with actors using the accessor pattern. Essentially, accessors are actors that serve as proxies for software that is not modeled by components; potentially provided by a third party. These accessors also yield deployment requirements. Only if a platform meets all requirements of an accessor, it can execute the accessor. None of these support synthesizing deployment improvement suggestions.

Third, there are IoT modeling tools such as OpenTOSCA for IoT [14], which enables modeling MQTT-based IoT applications but focuses on the low-level specification of technical details of the IoT devices and does not support containerization of components or architectures. While this eases addressing more resource-constraint devices, it increases the accidental complexities [19] in deployment.

Further approaches, e.g., include leveraging feature models to describe possible deployments of unspecified, i.e., not based on a dedicated C&C ADL, components [11], which prevents leveraging the underlying architecture model and requires significant tailoring to a concrete application or does not support deployment at all [23, 31].

Table 2 summarizes the state of related approaches to IoT deployment along the following six questions:

(1) Modeling Language: Does the IoT tool support abstraction by modeling (parts of) the application?

Table 2. Overview of Related IoT Modeling and Deployment Approaches (● = Fulfilled, ◐ = Partly Fulfilled, ○ = Not Fulfilled)

IoT Tool	Modeling Language	Code Gen.	Automated Deployment	Dyn. Self-Adaptation	Modeling Dynamics	Req.-based Deployment	Modification Proposals	
							Req.s	Infrastructure
Arduino IoT Cloud	○	○	○	○	○	○	○	○
AWS Greengrass	○	○	●	●	○	●	○	○
Azure IoT	○	○	●	●	○	●	○	○
balenaCloud	○	○	●	●	○	○	○	○
CapeCode [9]	●	●	○ [1]	○	● [2]	○	○	○
Ericsson Calvin [4]	●	◐ [3]	●	●	● [4]	●	○	○
DIANE [37]	●	○	●	●	○	●	○	○
Distributed Node-RED [22]	●	●	○	○	○	○	○	○
Eclipse Mita	●	●	○	○	○	○	○	○
Foggy [38]	○	○	●	●	○	●	○	○
GeneSIS [17]	○ [5]	○ [5]	●	●	○	◐ [6]	○	○
MARTE + CONTREP [28]	●	●	○	○	○	○	○	○
MDE4IoT [13]	● [7]	●	●	●	○	◐ [8]	○	○
OpenTOSCA [8]	●	●	●	○	○	●	○	○
ThingML [23, 31]	●	●	○	○ [9]	● [9]	○	○	○
MontiThings (this article)	●	●	●	●	●	●	●	●

[1] "An accessor can be dynamically downloaded and instantiated" [9], but CapeCode does not include an automatic deployment algorithm.
[2] Because accessors can take accessors as an input and instantiate them.
[3] Calvin can generate JSON from CalvinScript. The actors are implemented using handwritten Python code.
[4] Using the Kappa extension [32].
[5] The GeneSIS modeling language defines deployment/orchestration. It does not define application logic.
[6] GeneSIS can ensure devices have certain capabilities (regarding security and hardware). It does, however, not have a reasoner for defining requirements similar to our *local requirements*.
[7] Does not use a custom IoT-focussed language but **Unified Modeling Language (UML)** profiles and **action language for foundational UML (ALF)** [12, 13].
[8] Uses MARTE to specify requirements of the devices. It does, however, not have the means for specifying requirements about the system as a whole like our *local requirements*.
[9] ThingML has "dynamic sessions" [31] for handling joining or leaving nodes. It does, however, not actively instantiate nodes to use available devices. ThingML can be used dynamically using GeneSIS: "ThingML code can be dynamically migrated from one device and platform" [15].

(2) Code Generator: Does the approach support translation of IoT architectures into executable code?
(3) Automated Deployment: Is there support for fully automated deployment?
(4) Dynamic Self-adaptation **(R1)**: Can the deployment of the IoT applications dynamically adapt to infrastructure changes?
(5) Modeling Dynamics **(R2)**: Does the IoT tool's modeling language provide elements to describe the dynamic reconfiguration of the IoT application at runtime?
(6) Requirement-based Deployment **(R3)**: Is it possible to describe the deployment requirements explicitly?
(7) Modification Proposals **(R4)**, **(R5)**: Can the deployment infrastructure suggest changes to (a) the deployment requirements and/or (b) to the infrastructure?

Deployment in MontiThings conceptually builds on the basis of ROOM. It differs from the modeling frameworks for distributed IoT systems by supporting more expressive deployment constraints

and by generating deployment suggestions on relaxing constraints as well as on providing hardware devices with requirement properties. It also differs from other IoT modeling solutions by focusing on the deployment of containers and abstracting from technical accidental complexities where possible.

11 DISCUSSION

This section discusses the deployment method and infrastructure of MontiThings IoT applications as presented in this article. A thorough discussion of the MontiThings ADL and its code generators is available as well [27].

MontiThings uses a centralized deployment manager in contrast to peer-to-peer-based approaches [4, 32]. We consider centralized approaches better suitable for IoT applications for three reasons:

(1) IoT devices are often battery-powered and, thus, enter sleep modes to reduce energy consumption. This means they may be unavailable for communication over extended periods of time. Also, communication causes high energy consumption and, hence, superfluous (peer-to-peer) communication should be avoided. By managing deployment-related activities using a central deployment manager, we can reduce communication between the IoT devices and, hence, can extend their lifetime.
(2) IoT target devices are often based on relatively cheap, low-performance hardware. This will become less important in the future, as we are observing increasing computational power of single-board computers (such as the Raspberry Pi variants) in the past. Hence, target devices will be able to allocate more resources to running IoT applications.
(3) In a true peer-to-peer approach, no single IoT device knows the complete topology of available devices. This restricts the expressiveness of the requirements used for deploying the components, as without knowledge of the whole system, the requirements can be based on properties of individual devices only, but cannot consider properties of the overall system. Since the deployment of MontiThings leverages a centralized approach, it can express and resolve constraints that require global knowledge about the system, e.g., that a certain number of devices should run a particular component.

The deployment itself currently relies on (Docker) containers, which are automatically generated for each component type instantiated by the architecture to be deployed and comprise the components' source code implementations. While it is possible to deploy the generated component implementations manually, MontiThings aims at large-scale IoT applications where manual deployment rarely is feasible. Yet, for operating with very computationally constrained target devices, containers might produce undesired computational overhead. Future extensions of MontiThings' code generator might address such devices. The use of containers naturally results in certain minimum technical requirements for the hardware of the IoT devices used **(A1)**. These hardware requirements are on par with AWS' [3] and Microsoft Azure's [30] edge computing services, which also rely on containers for IoT deployments. Future IoT applications are expected to be built on container technologies [36]. As hardware becomes more powerful, less expensive, and more power-efficient, we think this vision is realistic. Specialized IoT container engines like the balenaEngine[12] can reduce the footprint of the container engine compared to Docker and cope with IoT-specific problems such as intermittent connectivity. However, these hardware requirements still exclude low-power IoT devices such as most Arduinos or the ESP32, which do not run Linux or containers. MontiThings components can, of course, also be deployed (manually)

[12]balenaEngine website: https://www.balena.io/engine/.

on such less powerful devices that do not support containers. Solving the associated challenges of cross-compiling and installing the software and caching it if the device has no direct Internet connection **(A2)** is not the focus of this article. For deployment to devices that do not support containers, we refer interested readers to the GeneSIS research project [15–17], whose software we have also integrated into our tool as described above.

As IoT applications change their structure and available devices over time, the deployment of software needs to change as well. To ensure that software can be executed by more than just one type of device, it is important that the technical requirements refer only to the capabilities of the devices and not, for example, to their product name. In our prototypical implementation, the requirements and capabilities are expressed as simple key-value pairs, similar to the tags cloud providers use to enable developers to label resources. It is important to remember that the requirements and capabilities must be specified with sufficient precision to enable the associated software to be executed appropriately. Specifying that software wants to be able to control a light source, for example, leaves open whether that light source is a light bulb, a tiny LED, or a car headlight. Conversely, overly specific technical requirements may severely limit on which devices a component can be deployed. Finding a reasonable set of technical requirements that both allows for reasonably usable devices and prevents unwanted deployments is the responsibility of the global manager. To specify such specifications across manufacturers, ontologies could be utilized. Alas, a survey on IoT ontologies identified "that no existing ontology is comprehensive enough to document all the concepts required for semantically annotating an end-to-end IoT application as ontologies are often restricted to a certain domain" [6]. Developing such a standardized cross-domain and cross-vendor ontology for IoT app stores is, therefore, an open challenge.

We plan deployment using Prolog and backtracking, whereas translating the deployment challenge into a constraint satisfaction problem might lead to more efficient computation of deployments. However, without backtracking, generating alternative deployments through relaxation of requirements and adding compatible target devices is less efficient. With manually deploying taking significant time anyway, and our deployment planner computing deployments in milliseconds, we do not consider this a relevant issue. However, the time to calculate a new proposal can be in the range of several seconds. This is caused by the Deployment Manager automatically discarding many proposals that the local manager is likely to reject if it has already rejected lesser changes, as described in Section 9.

Our requirement types are mainly based on Calvin's capability- and location-focused constraints [4]. In our prototype implementation, the four types of local requirements mentioned in Section 5 are currently expressible. Since we are converting requirements to Prolog, and Prolog is based on predicate logic, there is no conceptual limitation that would prevent other types of requirements expressible in predicate logic. But while global managers (software developers) are likely capable of expressing deployment constraints using predicate logic, the local managers (device owners) might not be familiar with it. Hence, we opted for an accessible formulation of local deployment requirements using the four kinds of requirements specified in Section 5. This restriction enables us to provide a web application through which the local managers can specify requirements without being directly confronted with predicate logic. However, since our concept evaluates requirements on an event-driven basis, it is also not possible to specify requirements that would need to be evaluated continuously. In particular, it is not possible to define time requirements. For example, it is not possible to require that a software is not executed after 5 p.m.

Our approach aims to emancipate device owners to participate in the deployment of IoT software on their devices. As such, we aim for simple but expressive means to define local requirements (cf. Section 5) and a UI accessible to devices owners. Hence, evaluation of the usability of our

deployment process, its expressiveness for industrial users, and the degree to which its explanations are helpful to improve deployments are subject to future work.

12 CONCLUSION

We have presented a pervasive model-driven deployment method that supports global managers and local managers in specifying and resolving deployments of large-scale IoT applications. To this end, the code generator central to our method translates deployment requirements, information about the components to be deployed and about the target infrastructure into a Prolog program that can be solved for finding suitable deployments. Using backtracking, solving this program enables generating suggestions to (1) relax deployment requirements and (2) suggest required extensions of available target devices. Using a central deployment manager that informs the overall IoT application when devices on which parts of the application are deployed fail or new target devices become available, the application can be automatically adapted as required **(R1)**. How the application reacts to hardware changes in detail is subject to each specific application and, hence, expressed using MontiThings' language elements for describing (sub)component dynamics **(R2)**. Using a MontiThings IoT architecture, requirements for the deployment of its (sub)components can be specified flexibly using global and local deployment requirements **(R3)** that can be relaxed by the Prolog deployment program **(R4)**, which also suggests extensions to the target devices **(R5)**.

To support local managers in (re-)deploying their application instances, we have devised a set of deployment requirements that can capture common deployment challenges and generate a unique web application per modeled IoT application through which local managers can describe their deployment requirements.

Overall, the flexibility of local and global deployment requirements combined with backtracking-based modification proposals facilitates the specification of complex deployments required for deploying IoT applications to highly variable networks of IoT devices.

SOURCE CODE

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

REFERENCES

[1] 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.

[2] 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

[3] Amazon Web Services Documentation. [n.d.]. Setting up AWS IoT Greengrass Core Devices—Device Requirements. Retrieved from https://docs.aws.amazon.com/greengrass/v2/developerguide/setting-up.html#greengrass-v2-requirements.

[4] 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.

[5] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. 2021. Artifact and reference models for generative machine learning frameworks and build systems. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'21)*. ACM SIGPLAN, 55–68.

[6] Garvita Bajaj, Rachit Agarwal, Pushpendra Singh, Nikolaos Georgantas, and Valerie Issarny. 2017. A study of existing ontologies in the IoT-domain. SciTePress. DOI: https://doi.org/10.48550/ARXIV.1707.00112

[7] Michael Blackstock and Rodger Lea. 2014. Toward a distributed data flow platform for the web of things (distributed node-RED). In *Proceedings of the 5th International Workshop on Web of Things (WoT'14)*. Association for Computing Machinery, New York, NY, 34–39.

[8] Uwe Breitenbücher, Christian Endres, Kálmán Képes, Oliver Kopp, Frank Leymann, Sebastian Wagner, Johannes Wettinger, and Michael Zimmermann. 2016. The OpenTOSCA ecosystem—Concepts & tools. *European Space project on Smart Systems, Big Data, Future, Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome* (2016).

[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 (Sept. 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 Softw.* 34, 1 (2017), 54–61.

[11] Angel Cañete, Mercedes Amor, and Lidia Fuentes. 2021. Supporting IoT applications deployment on edge-based infrastructures using multi-layer feature models. *J. Syst. Softw.* (2021), 111086. DOI: https://doi.org/10.1016/j.jss.2021.111086

[12] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. 2015. On the generation of full-fledged code from UML profiles and ALF for complex systems. In *Proceedings of the 12th International Conference on Information Technology*. 81–88. DOI: https://doi.org/10.1109/ITNG.2015.19

[13] Federico Ciccozzi and Romina Spalazzese. 2016. MDE4IoT: Supporting the Internet of Things with model-driven engineering. In *Proceedings of the 10th International Symposium on Intelligent Distributed Computing*.

[14] Ana C. Franco da Silva, Uwe Breitenbücher, Kálmán Képes, Oliver Kopp, and Frank Leymann. 2016. OpenTOSCA for IoT: Automating the deployment of IoT applications based on the mosquitto message broker. In *Proceedings of the 6th International Conference on the Internet of Things (IoT'16)*. Association for Computing Machinery, New York, NY, 181–182.

[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 *Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. 870–875.

[16] Nicolas Ferry and Phu H. Nguyen. 2019. Towards model-based continuous deployment of secure IoT systems. In *Proceedings of the 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. *J. Obj. Technol.* 19 (2020), 16:1–23.

[18] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[19] Robert France and Bernhard Rumpe. 2007. Model-driven development of complex software: A research roadmap. *Proceedings of the Future of Software Engineering Conference (FOSE'07)*. 37–54.

[20] 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.

[21] 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*, Judith Michael and Dominik Bork (Eds.). CEUR Workshop Proceedings, 22–30.

[22] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. 2015. Developing IoT applications in the Fog: A distributed dataflow approach. In *Proceedings of the 5th International Conference on the Internet of Things*. 155–162.

[23] 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 International Conference on Model Driven Engineering Languages and Systems (MODELS'16)*. ACM, New York, NY, 125–135.

[24] Stacey Higginbotham and Mark Pesce. 2021. Internet of everything: Macro & micro. *IEEE Spect.* 58, 1 (2021), 18–19. DOI: https://doi.org/10.1109/MSPEC.2021.9311418

[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. *J. Syst. Softw.* 183 (Jan. 2022), 111087. DOI: https://doi.org/10.1016/j.jss.2021.111087

[28] Frédéric Mallet, Eugenio Villar, and Fernando Herrera. 2017. MARTE for CPS and CPSoS. *Cyber-Phys. Syst. Des. Archit. Anal. Viewp.* (2017), 81–108. DOI: https://doi.org/10.1007/978-981-10-4436-6_4

[29] M. Mazanec and Ondřej Macek. 2012. On general-purpose textual modeling languages. In *Proceedings of the Workshop on Databases, Texts, Specifications, and Objects*. CEUR-WS.org.
[30] Microsoft Azure Documentation. 2022. Azure IoT Edge Supported Systems. Retrieved from https://docs.microsoft.com/en-us/azure/iot-edge/support.
[31] Brice Morin, Nicolas Harrand, and Franck Fleurey. 2017. Model-based software engineering to tame the IoT jungle. *IEEE Softw.* 34, 1 (Jan. 2017), 30–36.
[32] Per Persson and Ola Angelsmark. 2017. Kappa: Serverless IoT deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing (WoSC'17)*. Association for Computing Machinery, New York, NY, 16–21.
[33] Claudius Ptolemaeus. 2014. *System Design, Modeling, and Simulation: Using Ptolemy II*. Vol. 1. Ptolemy. org, Berkeley.
[34] Bran Selic. 1996. Tutorial: Real-time object-oriented modeling (ROOM). In *Proceedings of the Conference on Real-Time Technology and Applications*. 214–217.
[35] Bran Selic, Garth Gullekson, Jim McGee, and Ian Engelberg. 1992. ROOM: An object-oriented methodology for developing real-time systems. In *Proceedings of the 5th International Workshop on Computer-aided Software Engineering*. 230–240.
[36] Antero Taivalsaari and Tommi Mikkonen. 2017. A roadmap to the programmable world: Software challenges in the IoT Era. *IEEE Softw.* 34, 1 (Jan. 2017), 72–80.
[37] Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. 2015. DIANE—Dynamic IoT application deployment. In *Proceedings of the IEEE International Conference on Mobile Services*. 298–305. DOI: https://doi.org/10.1109/MobServ.2015.49
[38] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. 2017. Foggy: A framework for continuous automated IoT application deployment in fog computing. In *Proceedings of the IEEE International Conference on AI Mobile Services (AIMS)*. 38–45. DOI: https://doi.org/10.1109/AIMS.2017.14
[39] Franco Zambonelli. 2017. Key abstractions for IoT-oriented software engineering. *IEEE Softw.* 34, 1 (2017), 38–45. DOI: https://doi.org/10.1109/MS.2017.3

Received January 2022; revised April 2022; accepted June 2022