[Kir24] J. C. Kirchhof:
From Design to Reality: An Overview of the MontiThings Ecosystem for Model-Driven IoT Applications.
In: Ernst Denert Award for Software Engineering 2022: Practice Meets Foundations,
E. Bodden, M. Felderer, W. Hasselbring, P. Herber, H. Koziolek, C. Lilienthal,
F. Matthes, L. Prechelt, B. Rumpe, I. Schaefer (Eds.),
pp. 45-71, Springer Cham, Mar. 2024.

From Design to Reality: An Overview of the MontiThings Ecosystem for Model-Driven IoT Applications

Jörg Christian Kirchhof

Abstract The Internet of Things (IoT) networks everyday objects that can perceive and influence their environment using sensors and actuators. Since IoT systems are inherently distributed systems, often built on fault-prone hardware and exposed to harsh environmental conditions such as vibration or humidity, developing such systems is challenging. In recent years, some DSLs for IoT system development have been introduced, yet they only slightly improve IoT system development. This chapter provides an overview of *MontiThings*, an ecosystem for model-driven development of IoT systems that covers the life cycle of IoT systems from design in the form of Component and connector (C&C) models, through (dynamic) deployment, to failure analysis. MontiThings is designed to handle different classes of errors and failures. By being able to make counter-suggestions to device owners, the requirement-based deployment algorithm enables device owners to customize their IoT systems to their needs. MontiThings also offers an app store concept to decouple hardware development from software development in order to prospectively reduce problems such as e-waste and security issues that result from too close a coupling. Overall, MontiThings demonstrates an end-to-end model-driven approach to IoT system development.

Note This chapter summarizes the thesis [16]. Thus, the content of this chapter is taken from [16]. In particular, all illustrations were taken from the dissertation and the respective papers the dissertation is based on.

J. C. Kirchhof (✉)
Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: kirchhof@se-rwth.de

E. Bodden et al. (eds.), *Ernst Denert Award for Software Engineering 2022*,
https://doi.org/10.1007/978-3-031-44412-8_3

1 Introduction

The Internet of Things (IoT) networks everyday objects. Sensors and actuators enable them to perceive and influence their environment. The data obtained from the sensors is often used to automate processes with the help of the actuators. For example, in a smart home, the heating can be switched off automatically as soon as the window is opened. Because IoT devices belong to real-world objects, IoT systems are inherently distributed systems. The programming languages with which such systems are mostly developed today are often the same General-purpose programming languages (GPLs) such as C++ or Python with which all other types of systems are developed, according to an analysis of GitHub projects [7] and developer surveys [10]. These GPLs were not designed with the (primary) goal of improving the development of IoT applications. Accordingly, these GPLs are not well suited to address the challenges of developing IoT systems [32]. According to [32], the differences to programming traditional applications like web applications include, among other things, multidevice programming, the always-on nature of the system, heterogeneity, and the need to write fault-tolerant software.

In contrast to GPLs, domain-specific (modelling) languages often focus on solving a specific problem. Such modelling languages raise the level of abstraction, allowing certain aspects of development to be solved systematically in a way that GPLs cannot, since they must provide a certain level of generality. In the last decade, quite a few modelling and programming languages have been published for the development of IoT applications, including ThingML [13, 24], Ericsson's Calvin [3, 28, 29], Eclipse Mita [11], CapeCode [4], FRASAD [26], and Node-RED [27]. However, these languages often offer only a low level of abstraction [9], ultimately leaving the complexity of challenges such as multidevice programming to developers or focus only on early development phases and mostly neglect deployment.

In this chapter, we present *MontiThings*, an ecosystem for the model-driven IoT application development that covers the life cycle from initial prototypes to deployment on IoT devices to analysis of deployed applications. MontiThings consists of several modelling languages that clearly separate the business logic from the technical aspects of development. In doing so, MontiThings supports developers through various mechanisms in the development of error-resilient applications. In the event that errors do occur, MontiThings offers various analysis procedures. Since different instances of an IoT system can differ greatly from each other, MontiThings offers device owners the possibility to influence the deployment. In its app store concept, MontiThings also decouples the software from the hardware development, thus perspectively avoiding e-waste and security problems caused by outdated software or required cloud services discontinued by the manufacturer.

Figure 1 provides a brief overview of the MontiThings ecosystem. At design time, the IoT developers are developing various artifacts. Through MontiThings C&C architectures, the business logic of the application can be defined. Data structures are specified using class diagrams. If the MontiThings C&C language

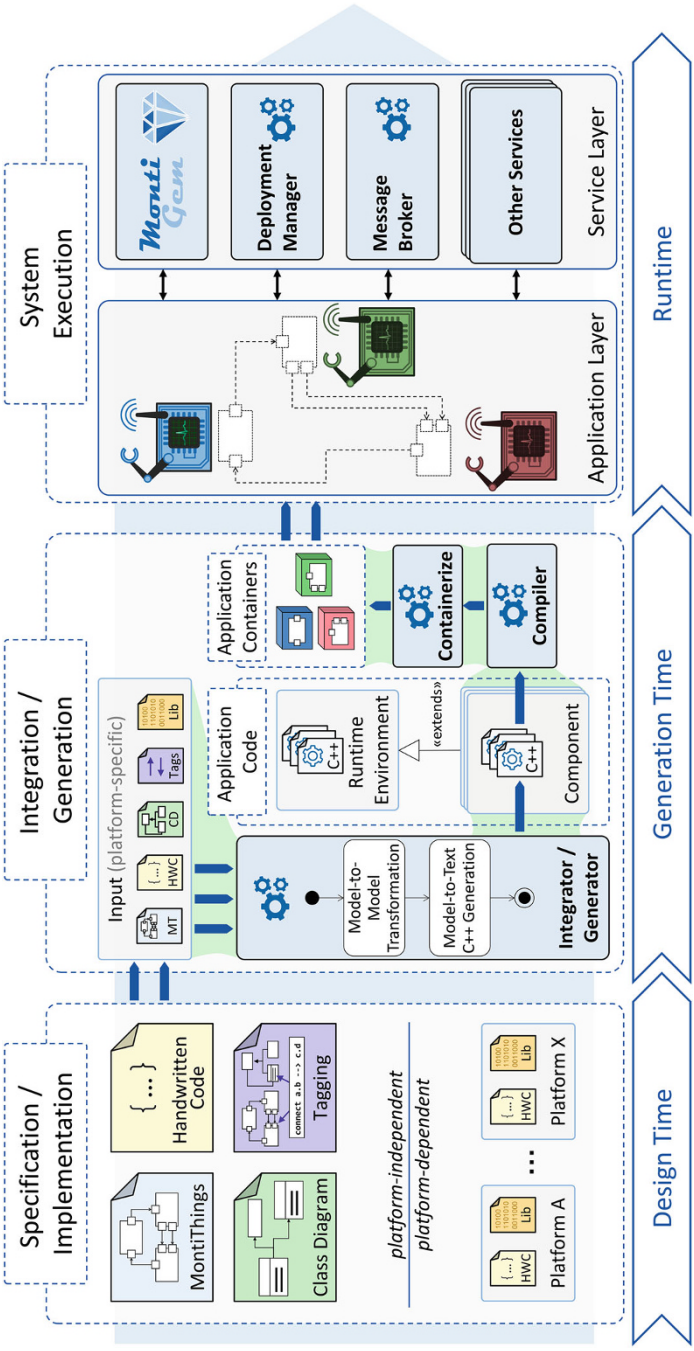

Fig. 1 Overview of the MontiThings ecosystem for model-driven IoT development. Models and code are used to generate C++ code, which is containerized and then downloaded by IoT devices. Multiple services interact with the devices to offer, e.g., communication and digital twins. Figure taken from [22]

is unsuitable to express a certain behavior, handwritten code in a GPL can be used as a supplement. Tagging languages can be used to define additional functionalities such as digital twins. In addition to these platform-independent artifacts, platform-specific artifacts, e.g., certain libraries for controlling a sensor, can also be used. All these artifacts are uploaded as input to an online repository (e.g., GitLab). There, a Continuous integration (CI) pipeline checks the artifacts, performs model-to-model transformations if necessary (e.g., to add components for digital twins), and then generates C++ code from the models. The generated code is linked against an RTE that provides common functionality such as communication between components. The generated code is then containerized and offered via a registry. From there, the IoT devices download the container images relevant to them. The Deployment Manager decides which images are relevant in each case. The Deployment Manager is one of several additional services that are operated at runtime alongside the actual application. These additional services enable communication between the components, provide digital twins, or offer analysis services, for example.

The rest of this chapter presents some parts of MontiThings in more detail: Sec. 2 first introduces the MontiThings language family. Sec. 3 then explains MontiThings' deployment algorithm. After that, Sec. 4 shows how tagging can be used to add digital twins to the application. Sec. 5 introduces MontiThings' app store concept. Sec. 6 shows different methods for error handling and analysis. Sec. 7 concludes. The MontiThings ecosystem can only be briefly described in this chapter. Please find additional information on the respective papers [6, 17–20, 20, 22] and dissertation [16].

2 The MontiThings Language Family

The core of MontiThings is a C&C language. This language is used to describe the business logic of IoT applications. For this purpose, IoT developers specify components that exchange data with other components via typed and directed ports. Instances of the components are connected to each other via connectors.

Figure 2 shows an example of such an application. The example shows a section of a fire alarm system. MontiThings uses both a textual and a graphical syntax. However, only the textual models are actually processed. The graphical models exist only for better understanding. Thus, the top two models are therefore two different representations of the same `FireAlarm` component.

The behavior of a component can be defined in four different ways:

1. By instantiating subcomponents and connecting them to the ports of the component instantiating them,
2. through a Java-like behavior language,
3. using statecharts,
4. using handwritten GPL code (e.g., in C++ or Python).

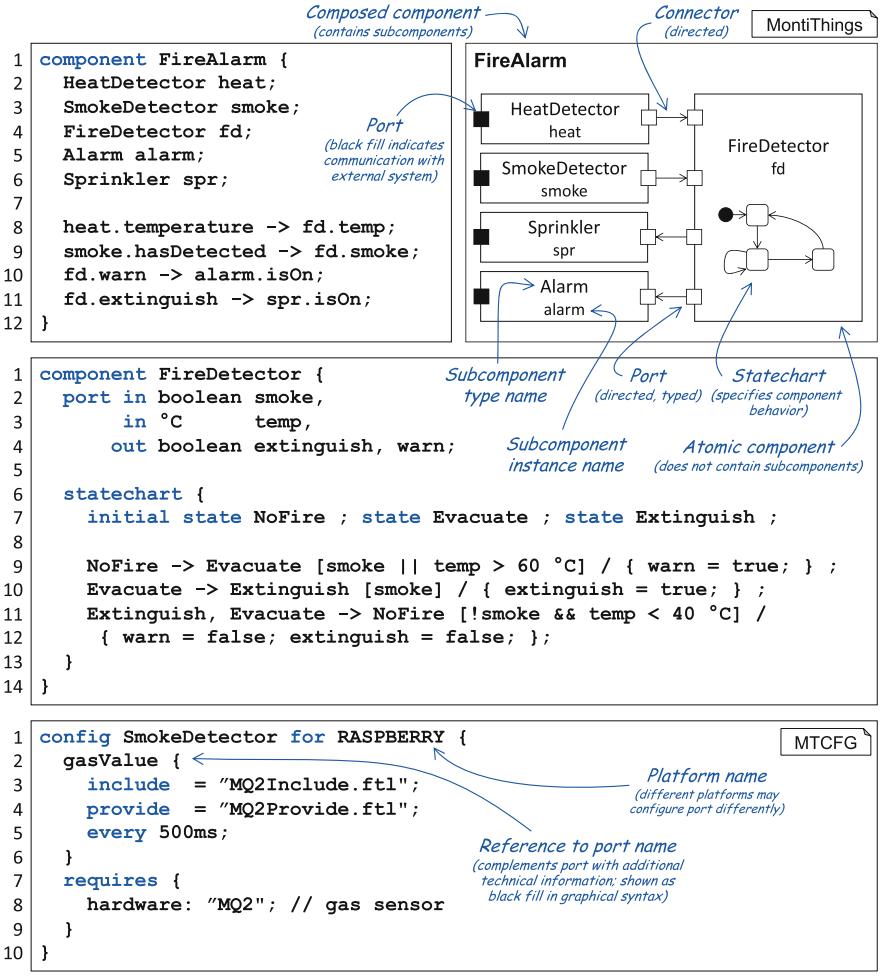

Fig. 2 An example of the graphical and textual syntax of MontiThings. The graphical syntax is only for better comprehensibility. Only the textual version is parsed. Figure adapted from [20]

Components that define their behavior through subcomponents are also called composed components. Components that describe their behavior via one of the other three methods are also called atomic components.

In the graphical syntax, one can see a difference between black and white ports. White ports represent a port that exchanges data with other components. Black ports represent a port for which the IoT developers have stored handwritten code. This handwritten code enables the port to access the hardware (e.g., a sensor). In the textual syntax, however, there is no difference between black and white ports. This makes it possible to use an override mechanism similar to that used by object-oriented languages to override base class methods in subclasses. If a port for which

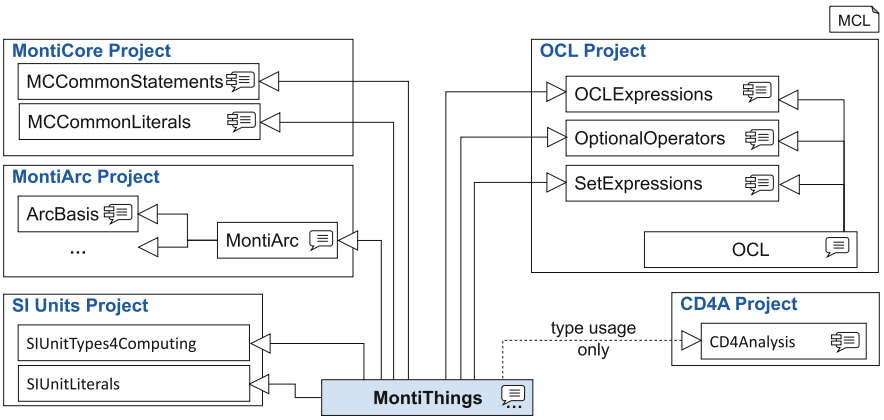

Fig. 3 Overview of languages from the MontiVerse incorporated by MontiThings' core language. Figure taken from [19]

handwritten code exists is connected using a connector, the handwritten code is automatically ignored, and only the connector is considered. This mechanism makes it easier to reuse components in different contexts. For example, a component that accesses hardware can be connected in the context of a test case with mock components that take on the role of the real hardware for the test.

MontiThings also serves as an example of how the MontiCore language workbench [15] can be used to build large languages. In total, MontiThings combines 46 grammars from the MontiCore language library in addition to its own grammars. An overview can be found in Fig. 3. Besides MontiArc, which is the basis for MontiThings, the type system and the expressions are especially worth mentioning. MontiThings reuses the primitive types of MontiCore. They are extended by the types of the International System of Units (SI) Units language. Hereby, it is possible to use SI Units like primitive data types. This can be seen, for example, in the middle model of Fig. 2, where `°C` is used like a normal data type. If two compatible but different types are to be converted into each other (e.g., `km/h` and `m/s`), MontiThings can automatically convert the values into each other in the background. This makes components more flexible to use, since the types of connected ports do not have to match but only have to be compatible to each other. If more complex data types are to be used, they can be defined via class diagrams of the Class diagrams for analysis (CD4A) project. MontiThings can import the symbols of such class diagrams and thus make them available to the components. These types can be instantiated using an object diagram-like syntax similar to Go's composite literals.[1]

Furthermore, MontiThings uses the Object constraint language (OCL) for expressions. The main use case here is to enable IoT developers to describe pre- and postconditions for component behavior. If an error is detected, the execution can

[1] https://go.dev/ref/spec.

either be aborted at this point or a behavior can be defined to handle the exception. For example, a default value or the last measured value could be used if a sensor value deviates too much from the expected range. Parts of the OCL can also be used within the Java-like behavioral language at points where Boolean expressions are provided. For example, an if condition can be specified using the OCL.

Further language features of MontiThings' core language such as the definition of initial behavior, periodic behavior, or dynamics can be found in [16].

Besides the C&C language, the MontiThings project consists of other languages. The MontiThings Configuration Language (MTCFG, bottom model of Fig. 2) is a tagging language that can be used to customize components depending on their target platform. For example, different code templates can be selected for different platforms (e.g., Arduino vs. Raspberry Pi). Technical requirements can also be specified here (cf. Sec. 3).

Furthermore, MontiThings includes a language for specifying test cases based on [14]. Figure 4 gives an example of this language. Again, the graphical syntax is only for easier comprehension. MontiThings only parses textual models. Technically, MontiThings uses the test models to generate C++ tests written against the GoogleTest framework. Based on MontiCore's sequence diagram language, the

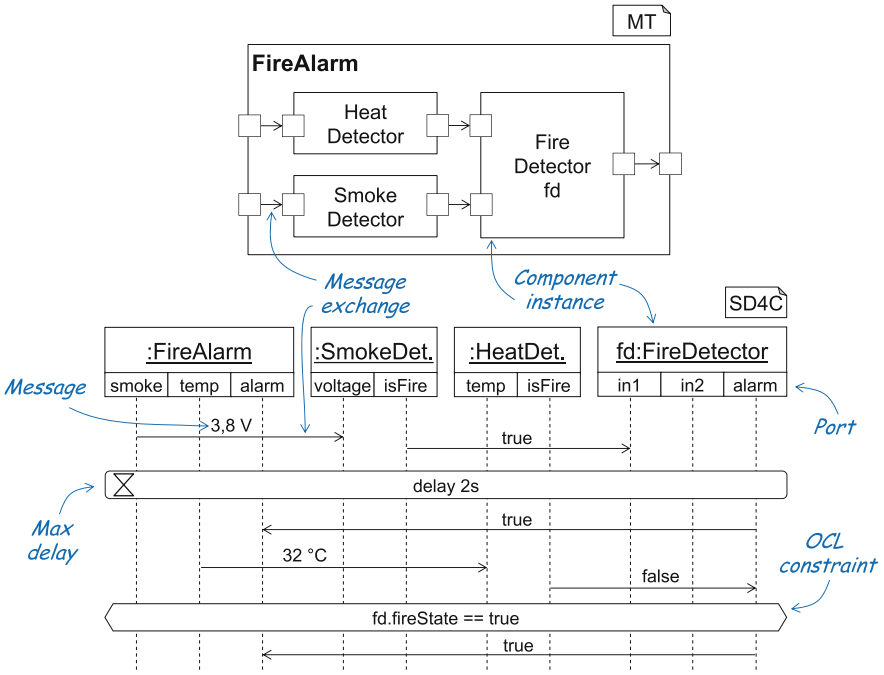

Fig. 4 White box test cases can be specified in the form of sequence diagrams that describe the message exchange between component instances. The graphical syntax of placing ports below components is taken from [14]. Figure taken from [16]

desired interaction between subcomponent instances of a composed component is represented in a sequence diagram. Of course, it is also possible to omit the specification of the inner workings and define a pure blackbox test where only the inputs and outputs are specified. In the depicted example, a smoke detector senses a voltage of 3.8 V and decides based on this voltage that there is a fire and informs the `FireDetector`'s `in1` port about it. After a maximum delay of 2 s, the `FireDetector` must have sent a message to the `alarm` port of the `FireAlarm` component. Then the temperature sensor detects a temperature of 32 °C and informs the `FireDetector` about this. Nevertheless, the `FireDetector` does not change its decision as it still has sufficient evidence of a fire based on the `SmokeDetector`'s earlier message.

3 Requirement-Based Self-Adaptive Deployment

IoT applications are often distributed applications. Partial applications must be deployed to a large number of IoT devices. In the same way, parts of applications can also be deployed to a cloud. The interaction of the IoT devices and the cloud results in the overall business logic. Furthermore, IoT applications can also include user interfaces via which the data of the application can be viewed or commands can be sent to the application. Such graphical user interfaces are not considered in this chapter.

IoT devices can be very different from each other. In addition to different computing power, they can also have different sensors and actuators. Consequently, the sub-applications cannot be deployed arbitrarily on the IoT devices. Instead, deployment requires precise planning of which devices should run which software. In addition to the purely technical framework conditions, the personal wishes of the device owners also play a role. For example, a device owner may wish not to install camera software provided by a social network on the devices in his bathroom. Legal requirements can also play a role. For example, in some countries, it is necessary to install a fire alarm in certain living spaces. A requirement could therefore be to install a fire alarm in every room, for example.

Furthermore, the deployment of IoT applications is not necessarily static. One reason for this is that IoT devices—unlike a television, for example, which is sold as a complete product—are often sold in extensible form. Many people initially buy a small number of IoT devices. If these devices prove successful, more devices are purchased. In this way, the IoT system is continuously expanded. The deployment of the software must adapt to these changes in the hardware accordingly. On the other hand, IoT devices can also fail. IoT devices often consist of inexpensive hardware and are often exposed to harsh environmental conditions. These and other factors favor a failure of the devices. Furthermore, IoT devices can of course also be deliberately removed by their owners. If an IoT device leaves the system, the deployment may have to be adjusted accordingly.

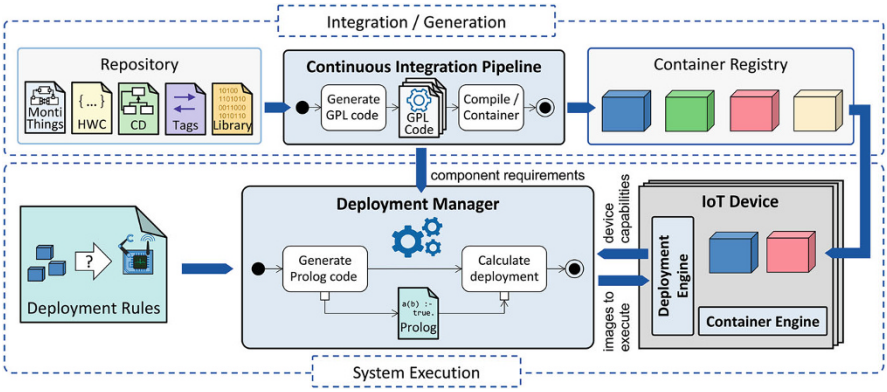

Fig. 5 The deployment manager generates Prolog code that calculates which IoT devices execute which images from the container registry based on technical requirements of the components, requirements of the device owners, and information about the IoT devices. Figure taken from [22]

MontiThings relies on a requirement-based deployment process. Figure 5 gives an overview of this deployment process. MontiThings distinguishes between technical requirements and local requirements. Technical requirements define the properties that a component must technically fulfill in order to be able to execute a component. They are defined by the IoT developers at design time. Local requirements, on the other hand, refer to the locality in which a component is executed. These requirements can be different for each instance of an application. They are defined by the device owners.

The technical and local requirements are merged in the Deployment Manager. In addition, the Deployment Manager receives information about the devices used in an IoT system. The Deployment Manager uses all this information to generate Prolog code that can be used to calculate a distribution of the software components to the IoT devices. In the process, Prolog facts are generated from the information about the IoT devices, and queries are generated from the requirements. A special feature here is that the generated Prolog code can not only calculate a distribution of the software components to the IoT devices but can also make counterproposals in the case of unfulfillable requirements. In particular, the purchase of new IoT devices and the modification (i.e., weakening) of the requirements can be suggested. Once a deployment is agreed upon with the device owner, the Deployment Manager communicates it to the IoT devices, which then download the (Docker) containers assigned to them according to the deployment.

Figure 6 shows the deployment process in more detail. First, IoT developers model their IoT components using MontiThings. In particular, they also specify the technical requirements of the components. If necessary, they also implement handwritten code to implement the behavior of the components. The IoT developers then upload all these artifacts to an online repository. There, a CI pipeline distributes the uploaded artifacts. First, the artifacts are checked for validity. If errors are

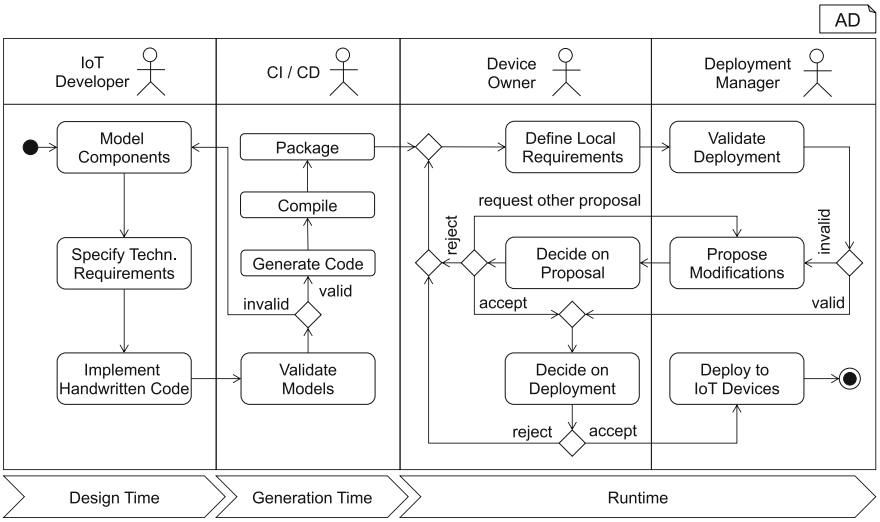

Fig. 6 Deployment process. The artifacts of the IoT developers are checked and provided by a CI/CD pipeline. The device owners negotiate with the deployment manager which devices should run which software. Figure taken from [20]

found, the IoT developers are asked to correct the errors with a corresponding error message. If the artifacts are accepted as valid, they are then used for code generation. The generated code is compiled and packaged into containers.

The device owners who want to deploy the application on their infrastructure must first specify their local requirements. MontiThings currently supports the following four types of local requirements:

1. A component shall (not) be deployed at a specific location,
2. A location requires a (minimum, maximum, or exact) number of components to be deployed there,
3. Two components may not be deployed to the same device,
4. A component requires a certain number of components (optionally in a similar location, i.e., the same room, floor, or building).

The Deployment Manager first validates these local requirements. If a valid deployment can be found taking into account the requirements, the device owners can decide whether they want to install this deployment on their devices. If no deployment can be found, the Deployment Manager suggests changes to the device owners. It is always possible to reject the proposed changes. In this case, the Deployment Manager calculates another proposal. In order not to overload the device owners with very similar proposals, the proposals are filtered so that the rejection of a proposal automatically counts as a rejection of all supersets of this proposal. For example, if the device owners refuse to buy a new fire alarm for the bathroom, a theoretically possible proposal to buy one fire alarm for the bathroom and one for the kitchen is automatically rejected.

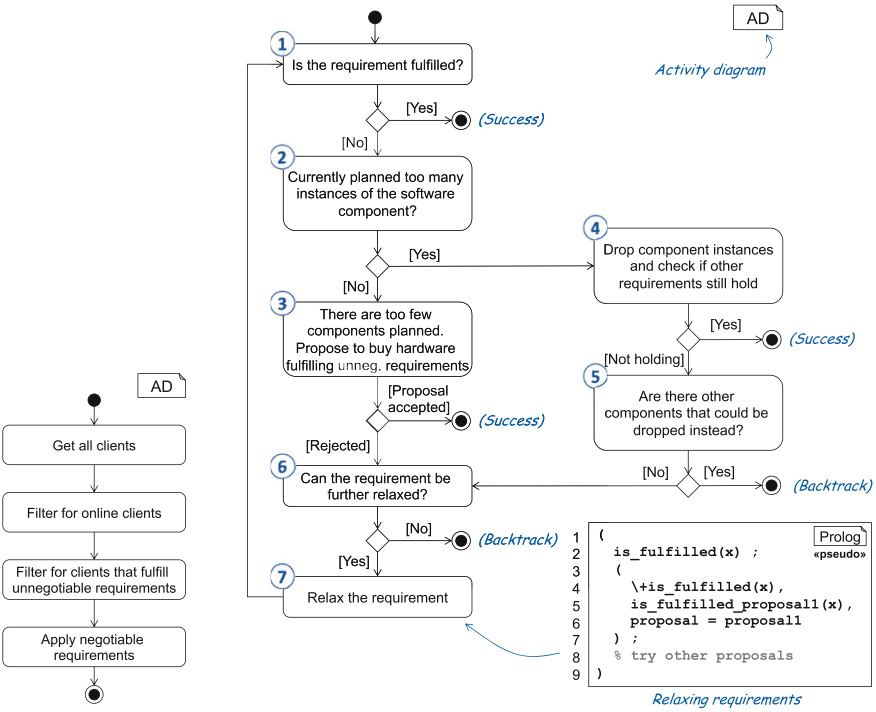

Fig. 7 Overview of the Prolog code generated by Deployment Manager. Left: high-level workflow. Right: applying a single negotiable requirement. Figure taken from [20]

The process of how the automatically generated Prolog code processes the requirements is shown in more detail in Fig. 7. First, Prolog searches the list of all known IoT devices for the devices that are currently online and thus available for deployment. Based on this list, it then identifies the devices that meet the technical requirements. Since the IoT developers are not involved in the deployment process, their technical requirements are considered non-negotiable in the process. If a device does not meet the technical requirements of a component, it cannot execute the component. Device owners cannot overrule this decision. After applying the non-negotiable requirements, the local requirements are checked. These are assumed to be negotiable because the equipment owners are involved in the deployment process and can respond to counterproposals. The reaction includes in particular the possibility to reject all counterproposals and to cancel the deployment, i.e., to consider the local requirements as non-negotiable as well.

When Prolog considers a local requirement, it first checks whether the requirement is already satisfied by the current allocation of components to IoT devices (1 in Fig. 7). If this is the case, one can proceed to the next requirement. If the requirement is not fulfilled, it is first checked whether too many IoT devices are currently executing the corresponding component (2). This can occur, for example,

if device owners require a particular component to be deployed *a maximum of 5 times*. If this is the case, components are removed from IoT devices using backtracking, and it is checked whether the requirement can be fulfilled in this way while complying with the previously processed requirements (4 and 5). If the component is not scheduled too often, it is handled that a component is not yet scheduled frequently enough. In this case, it is first checked whether the requirement can be met by purchasing more hardware (3). Only if this is not the case is it suggested that requirements be reduced (6 and 7). In order not to overload the device owners with requests that may not have any influence on the ultimate fulfillment of the deployment later in the process, the modification proposals are first collected before they are presented to the device owners until a theoretically valid deployment is found. The Deployment Manager implicitly assumes that all change requests are accepted. Should a valid deployment be found with this, the proposed changes will be offered to the device owners in a bundle.

Creating local requirements requires some knowledge of the software components of the IoT system. This is not desirable in some cases. On the one hand, because it requires IoT developers to disclose their software architecture to a certain extent and, on the other hand, because it requires training from device owners. Therefore, to increase the level of abstraction, MontiThings also offers an approach based on feature diagrams. Here, IoT developers create a feature diagram that models the features they envision in their application and their dependencies on each other. They use tagging to relate the features to the software components. This is illustrated in Fig. 8. This enables device owners to select the desired features based on the abstract feature diagram. Furthermore, device owners can run automatic analyses through which feature configurations are automatically calculated. For example, the largest possible feature configuration can be calculated or the largest possible feature configuration that can be deployed with the existing IoT devices. Behind the feature analyses lies the previously described requirements-based mechanism, which generates Prolog code from requirements.

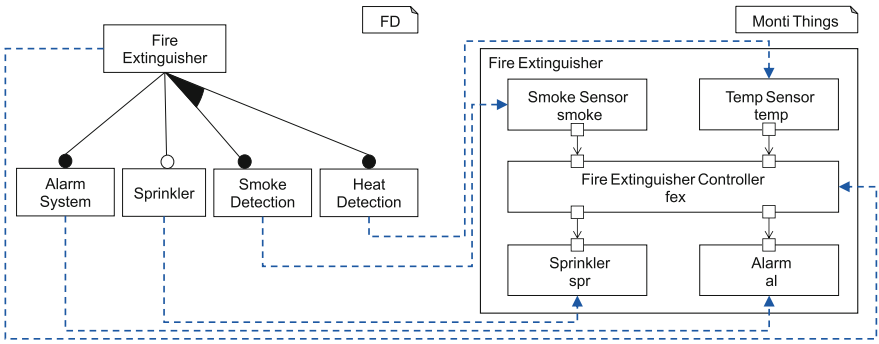

Fig. 8 Feature Diagrams can be used to tag architecture models. In this way, multiple components can be combined into a common feature. Device owners can thus select the desired features at a higher level of abstraction. Figure taken from [6]

For this purpose, requirements are generated from the feature configurations, e.g., that a certain component must be deployed in the system so that a certain feature is fulfilled.

4 Synthesizing Digital Twins

Once the components are deployed to the target infrastructure, the next challenge is to observe or influence the system. For this purpose, digital twins can be created. In this chapter, we will refer to the definition of digital twins that the Chair of Software Engineering has developed through several years of discussions and a systematic literature review [8]:

Definition 1 "Digital Twin, V2.1
A *digital twin* of a system consists of

- a set of models of the system and
- a set of digital shadows, both of which are purposefully updated on a regular basis, and
- provides a set of services to use both purposefully with respect to the original system.

The digital twin interacts with the original system by

- providing useful information about the system's context and
- sending it control commands." [30]

MontiThings offers the possibility to create digital twins based on class diagrams and C&C architecture models. The class diagram represents the data structure of a Digital twin information system (DTIS). In the actual implementation, the business logic of the system is created as usual with MontiThings. The information system is created with the help of MontiGem [1, 12], a tool for the model-driven creation of web applications. Figure 9 gives an overview of the process of synthesizing digital twins. After the IoT applications and the web application, and thus MontiThings models and class diagrams, have been developed (step 1 and 2 in Fig. 9), a system integrator connects the models together (step 3). For this purpose, he connects attributes of the class diagram with ports of the MontiThings architecture by means of tagging.

For this purpose, let's look at exemplary models of a fire extinguishing system in Fig. 10. The associated tagging model that the system integrator uses to connect the two models is shown in Fig. 11. First, the integrator uses the `identify` keyword to distinguish the different IoT devices from the web system. This can be done either by an entry in the database (especially if the digital twin is created before the real system) (ll. 1–5) or by the system automatically assigning identifiers to the IoT devices and storing them in the database (ll. 6–8). After that, the ports of the architecture models are connected to the attributes of the class diagram. The direction plays a role here. On the one hand, the real system can have data sent to its

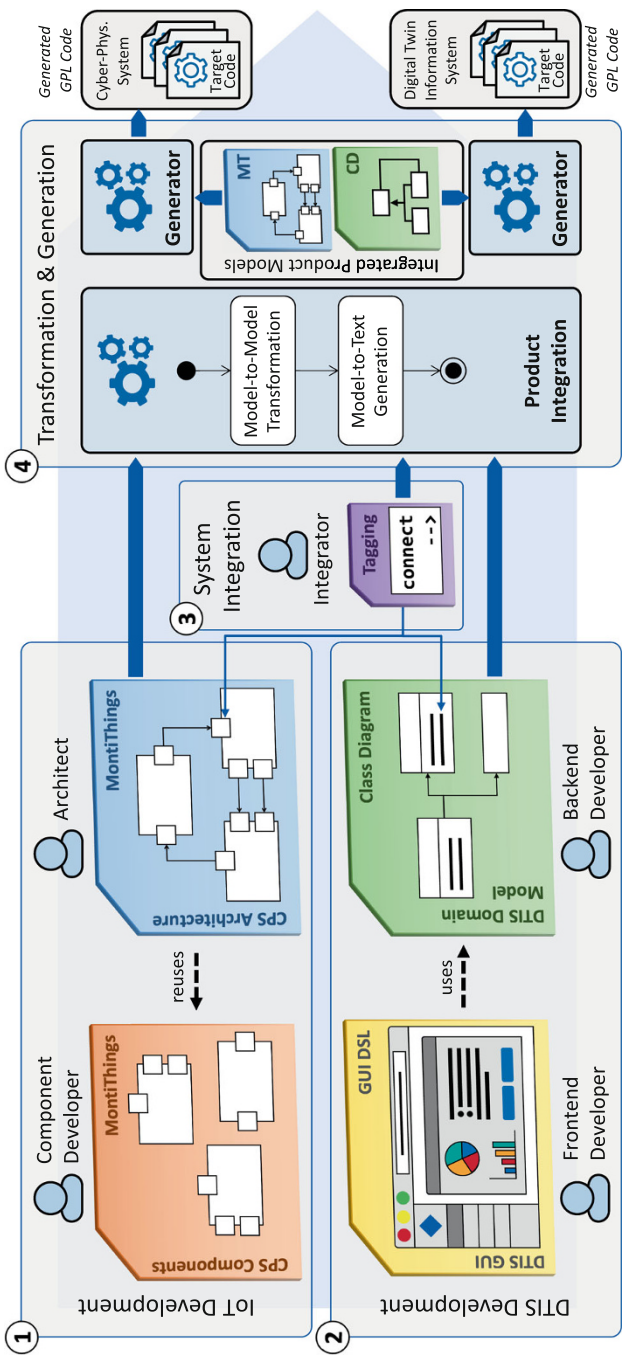

Fig. 9 Tagging allows MontiThings models to be associated with class diagrams that define the data structure of a web application. Via model-to-model transformation, the necessary model elements are added that keep the web system as a digital twin in sync with the IoT system. Figure adapted from [17]

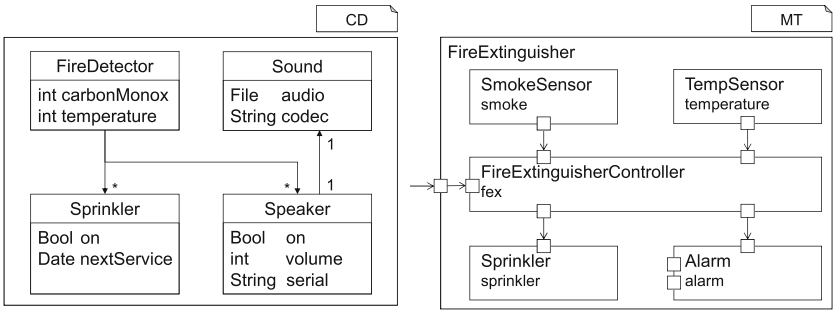

Fig. 10 An example of a fire alarm application. Left: the data structure of the web application. Right: the model of the IoT application. Figure taken from [17]

Tagging Model

```
1  // Objects of the Sound and Speaker classes serve as
2  // digital twins for CPS devices that use the value of
3  // Speaker.serial as identifier
4  identify Sound    by attr Speaker.serial
5  identify Speaker  by attr Speaker.serial
                     class from domain model
                     class from domain model. Same as before or reachable via to-1-association.
                     attribute of the class before the dot. Stores device identifier.

6  // Automatically create and link a digital twin when
7  // a device first connects to the IS
8  auto identify FireDetector
                 class name from domain model.

9  // Send data from the CPS architecture to the IS
10 connect port smoke.value
11     --> attr FireDetector.carbonMonox
           architecture component instance   outgoing port
12 connect port temperature.value
13     --> attr FireDetector.temperature
                class name from domain model   attribute name
           "Use messages from temperatureSensor's port 'value' to update the temperature attribute of the FireDetector class"

14 // Send data from the IS to the CPS architecture
15 connect attr Sprinkler.on          --> port sprinkler.on
16 connect attr Speaker.on            --> port alarm.on
17 connect attr Speaker.volume        --> port alarm.volume
           "If 'Speaker.volume' changes, inject the new value into alarm's 'volume' port"

18 connect attr Speaker.sound.audio --> port alarm.sound
                class name from domain model   (association +) attribute   component instance   port
```

Fig. 11 The tagging language associates attributes of a class diagram with ports of a C&C architecture (ll. 9-18). Additionally, it defines how the IoT devices identify themselves to the web application (ll. 1-8). Figure taken from [17]

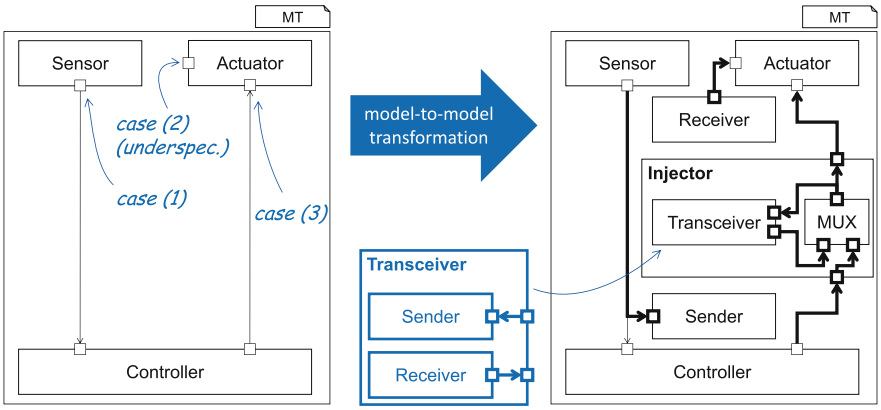

Fig. 12 Model-to-model transformations add components to the C&C architecture that synchronize with the digital twin. Elements created by model-to-model transformations are shown in bold. Figure adapted from [17]

digital twin by sending data from the port to the attribute in the class diagram (and thus to the database generated from it) (ll. 9–13). On the other hand, the digital twin can send data to its real counterpart by defining the reverse direction in the tagging (ll. 14–18).

Once the models are connected, the next step is to process them through model-to-model transformations (step 4 in Fig. 9). The transformations give the models additional elements that keep the real system and its twin in sync with each other. In the following, we will look at the transformations of the architecture. Interested readers can find a more detailed explanation of the method and the transformations of the web system in [17]. Figure 12 gives an overview of the architecture transformations. We distinguish three cases:

1. Connecting an outgoing port
2. Connecting an incoming port that currently has no incoming connectors
3. Connecting an incoming port that already has an incoming connector

In the first case, we add a new component via transformation that receives all data sent through the port and forwards it to the digital twin. In the second case, we do the reverse and add a component that receives data from the digital twin and forwards it to the port. In the third case, the already-existing connector must be resolved. We replace it with a new `Injector` component. On the one hand, this contains a `Transceiver` component that can both forward data to the digital twin and receive data from it. The situation can arise here that the value of the digital twin does not correspond to the value that the component receives via the connector replaced by the transformation. To resolve this situation, the `Injector` component includes a `MUX` that decides whether to use the data from the digital twin or from the real system. Users can control this `MUX` in the web interface. It enables them to prevent their desired values from being overwritten by the real system in the next

moment. For example, in our fire alarm, a test alarm can be triggered even if the sensors report that there is no fire, and the alarm should therefore be switched off.

5 IoT App Store Concept

When IoT devices are sold today, they are usually sold as a single product consisting of hardware and software. This gives the provider a great degree of control over the IoT devices. Users are usually not free to install new software on their IoT devices. If the manufacturer of the IoT devices now decides to change the rules of the game after the devices have been purchased, e.g., to introduce a subscription model, the user usually has little recourse against this. If the device manufacturer decides to shut down the cloud services required to operate the devices or simply goes bankrupt, the devices can become electronic waste. This practice is neither economically nor ecologically sustainable.

One way to solve this problem is to introduce an app store that would allow software to be installed independently of the hardware manufacturer. Such an IoT app store has already been proposed by various scientists, e.g., [2, 5, 25]. Consequently, MontiThings also includes a concept for an app store. Figure 13 shows an overview of MontiThings' app store concept. This concept is mainly based on the deployment algorithm already presented. A key feature of the concept is the clear separation between hardware and software development. The software developer specifies his application as previously introduced by C&C architecture models. In addition, their hardware requirements are specified for each component. The hardware requirements are specified thereby with the help of OCL. Thus, for example, also ranges of hardware requirements can be defined, e.g., a camera with at least 4 megapixels (instead of *exactly* 4 megapixels). Optionally, other models such as a feature diagram can be used to define high-level features. The applications specified in this way are transformed into executable container images by a CI/Continuous deployment (CD) pipeline.

On the hardware side, device developers develop their IoT devices and the corresponding drivers to access their devices. In addition, they specify the properties of their IoT devices in the form of an object diagram. On the software side, the IoT devices have the following software stack: A container engine executes the containers of the actual IoT application as specified by the deployment algorithm. A message broker enables the device-internal communication between the application containers and the hardware drivers. The hardware access manager coordinates which application containers access which sensors and actuators. This is particularly relevant if there is more than one instance of a hardware component, e.g., *four weight sensors*. It ensures that the application containers do not conflict with each other. The hardware access manager tells the application containers on which topics they can communicate with the requested hardware. The hardware access manager is then no longer involved in the subsequent data exchange.

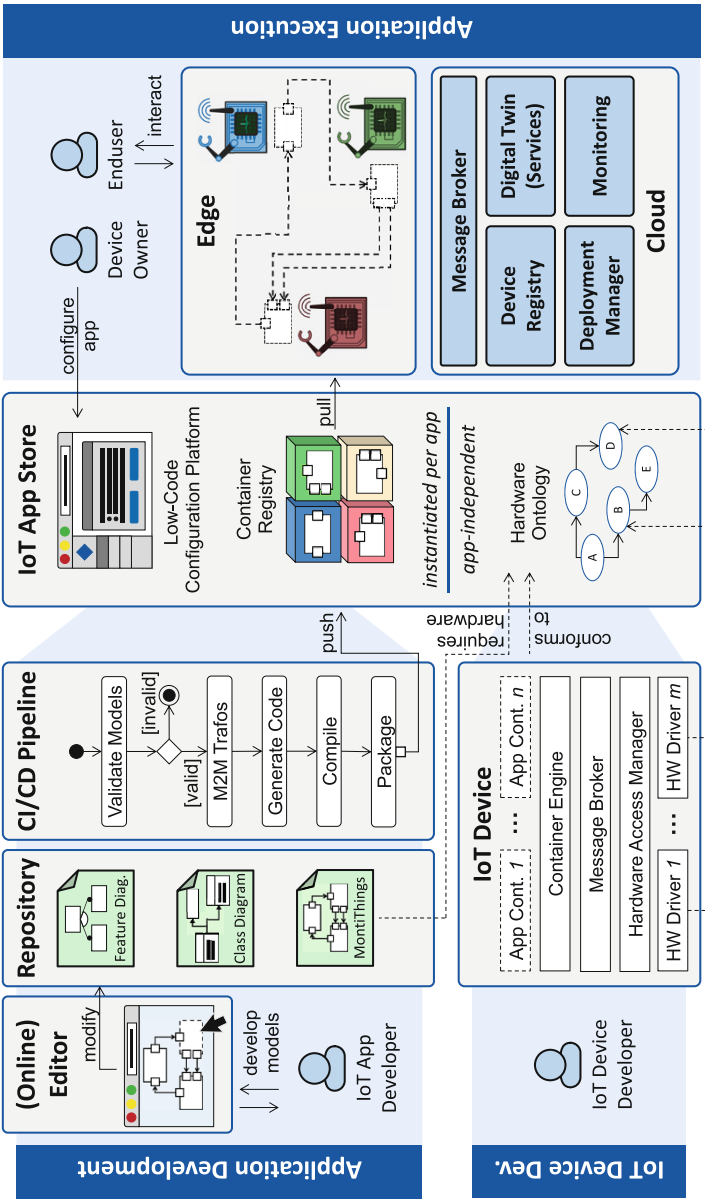

Fig. 13 MontiThings' app store concept decouples hardware and software development. Software developers specify technical requirements for components in OCL. Hardware developers describe technical properties of their devices via object diagrams. A hardware ontology provided by the app store harmonizes the requirements and device properties. Figure taken from [6]

To determine which hardware can execute which software components, the deployment algorithm must now check the OCL requirements of the software against the object diagrams that describe the hardware. For integration into our deployment algorithm, both are transformed into Prolog. The details can be found in [6]. To enable the software and hardware developers to match OCL and object diagrams in the end, even if the developers do not know each other, the app store provides a hardware ontology in the form of a class diagram. This class diagram specifies which types of hardware the app store expects in principle and which properties must be defined for such hardware. For example, it can be defined that cameras are a type of sensor and a width and height in pixels must be specified for each of the images shot. The rest of the deployment process then takes place as usual, i.e., device owners can specify additional local rules in a web interface. The deployment algorithm then decides which IoT devices should execute which software components and the IoT devices download the software accordingly from a container registry.

6 Failure Handling in MontiThings Applications

IoT devices are often based on low-cost hardware. One disadvantage of this hardware is that it is not particularly protected against failures or errors. IoT software must therefore be able to deal with the fact that errors occur. Such errors range from incorrect sensor values to completely failing devices.

MontiThings' C&C models describe the business logic of IoT applications. Technical details are not visible at this level of abstraction. Figure 14 shows an example of this. Even if the application has been modeled correctly in itself, various errors can occur at runtime due to unreliable hardware. Sensors can provide incorrect readings, affecting the flow of the system. Similarly, software errors such as incorrectly set clocks can affect the system. Network problems can delay or completely prevent the delivery of messages. This is especially noticeable on mobile devices that must rely on a cellular connection.

Frameworks for developing IoT applications must therefore be able to handle errors. Thus, MontiThings provides several mechanisms for analyzing and handling errors, which are summarized in the remainder of this section.

6.1 Record and Replay for Handling Failing Devices

The strongest form of failure of an IoT device is its complete failure. MontiThings deployment algorithm can detect failing devices by missing heartbeat messages. When a device fails, the components that the device was executing before its failure (if possible) are reassigned to another IoT device. However, this new device is not in the same state as the failed device before its failure.

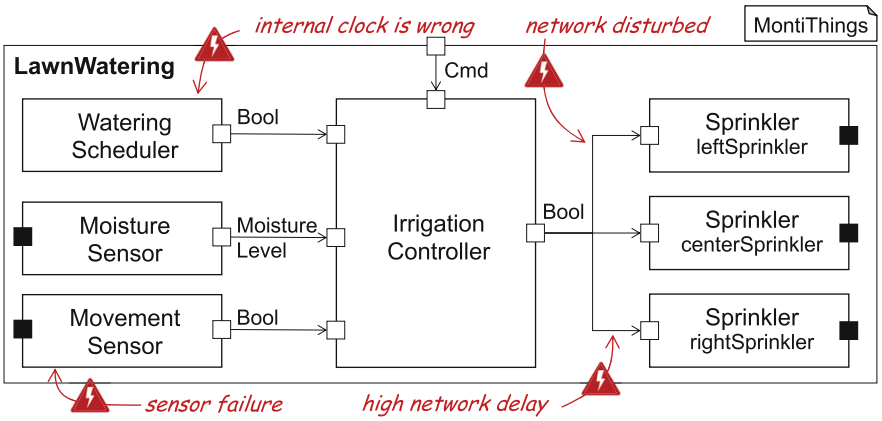

Fig. 14 (Hardware) errors that are not caused by the business logic may not be detected directly in the C&C architecture. Figure taken from [18]

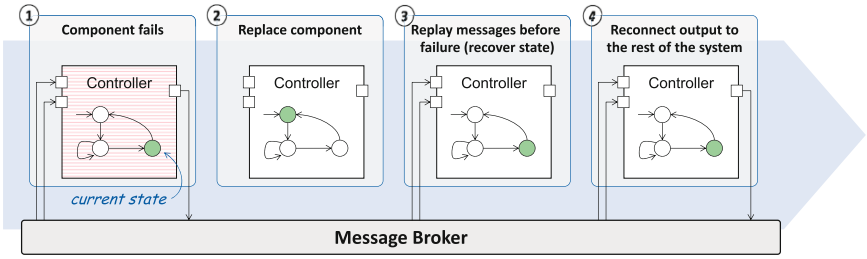

Fig. 15 If components fail (due to hardware defects), the components that replace them are not necessarily in the same state. MontiThings restores the state of the failed component by resending messages sent to the failed component to the new component. Figure taken from [22]

To address the issue of complete hardware failure, MontiThings uses record-and-replay. Figure 15 shows an overview of this. MontiThings continuously records the messages exchanged between the devices during runtime. If one device fails, the deployment algorithm starts the component on another device. When the new component is launched, incoming connectors are first connected to a replayer. The recorded messages are then used to put the new component in the state of the failed component. The replayer plays back the recorded messages. Once the messages are replayed and thus the state is restored, the ports are connected to the rest. In particular, the outgoing ports are connected only now, so that the messages sent as a by-product during state recovery do not affect the rest of the system.

This procedure has a complexity of $\mathcal{O}(n)$, where n is the number of messages. To improve this, components can periodically serialize their state and store it in the record-and-replay system. If a component fails, only the constant number of messages since the last state serialization has to be replayed. Thus the complexity sinks to $\mathcal{O}(1)$.

6.2 Recording and Transformation-Based Replaying

In less severe failure cases, only parts of the IoT system fail or misbehave. One problem in analyzing such errors is that they often cannot be reproduced under laboratory conditions. To analyze such faults, MontiThings therefore offers the possibility to record the behavior of the system and reproduce it later under deterministic conditions. Figure 16 shows an overview of this procedure. The procedure consists of the following steps:

1. IoT developers model their application through a C&C architecture as usual.
2. the developers' models are used to generate (C++) code that is executed by the IoT devices.
3. during the runtime of the system, a recorder records all messages exchanged between the devices. Metadata is also recorded. This includes, for example, what time elapsed between sending and receiving a message. As a result, system traces are created that contain the recorded system behavior.
4. a transformation engine uses the architecture models originally used by the generator and the system traces to create a new architecture model, the reproduction model. This model is a modified form of the original model that allows the replay of the system traces.
5. from the reproduction model, a new (non-distributed) application and (C++) code are created. Unlike the original version of the application, this is not a distributed application but a single binary. We call this application Reproduction Executable.
6. the Reproduction Executable can now be analyzed by the IoT developer using the usual debugging tools such as *gdb*. In particular, he now also has the possibility, for example, to set breakpoints and thus stop the entire system. Inspecting the global state of the system like this is not easily possible in a distributed system [31].

In step 4, the reproduction model was created from the architecture and system traces. Figure 17 gives a detailed insight into the relationship between the original model and the reproduction model. The transformation engine looks for places in the original model where the hardware or the environment affects the execution of the IoT system. At these points, the corresponding model elements are replaced or extended in such a way that the influences are removed and deterministically reproduced for the reproduction. In particular, sensors and actuators are replaced by components. By the mechanism described in Sec. 2, it is sufficient to insert new components and connect their ports to the black ports for this purpose. The new components then mock the real hardware by, for example, replaying recorded sensor values at the right time. Where components are connected, new components are introduced that simulate the recorded network properties. This means in particular delaying or losing messages. Where components execute a computation (atomic components), a wrapper is introduced around the components, which maps the

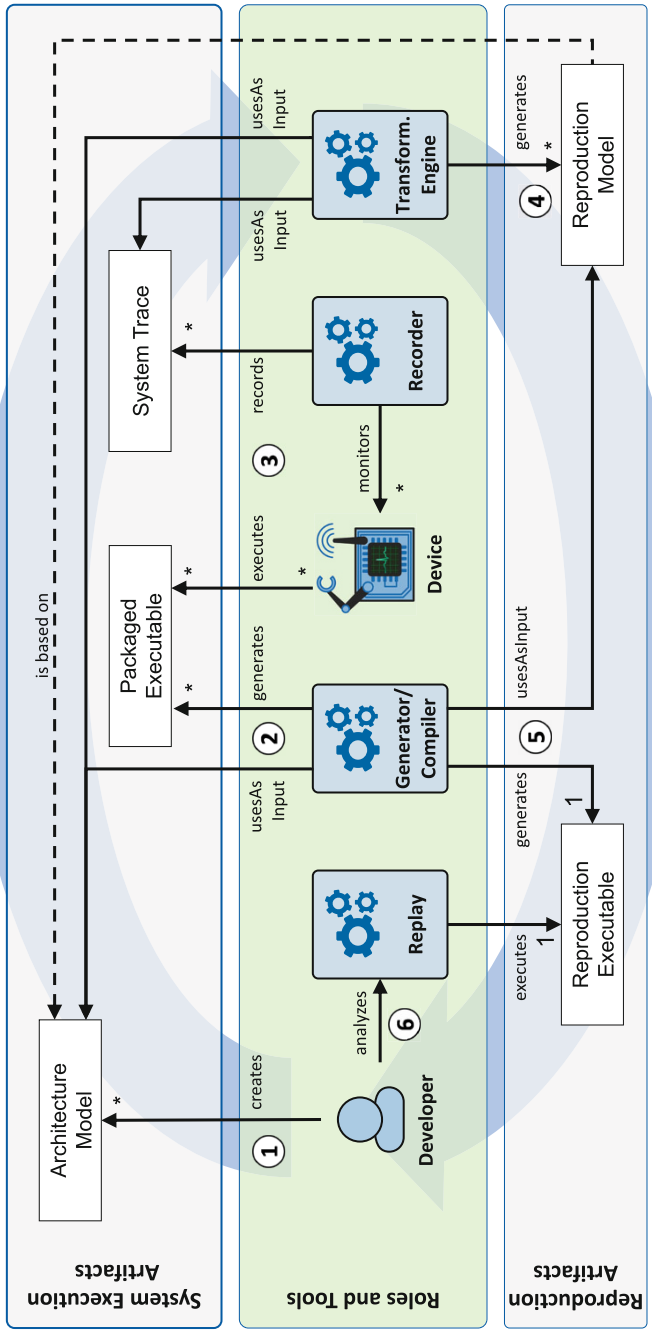

Fig. 16 Developers can recreate recorded errors in a reproduction. During runtime, exchanged data and metadata are recorded. These are used to create a reproduction model. This reproduction model can be analyzed by the developer. Figure taken from [18]

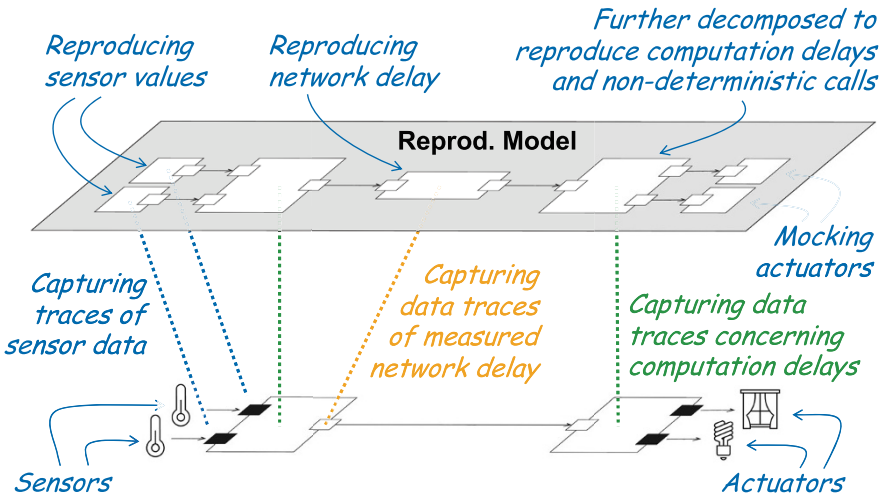

Fig. 17 The reproduction model (top) replaces hardware- or environment-dependent model elements in the original model (bottom) with elements that replay the recorded data. Figure adapted from [18]

delay by the computations of the processor. Further details like the handling of non-deterministic computations can be read in [18].

6.3 Web-Based Failure Tracing

The method presented in the previous section analyzes faults in an environment separated from the real system. Another popular option for debugging is the analysis of logs. The difficulty with IoT devices is that they are distributed applications. The logs of the individual IoT devices are therefore not necessarily available in a coherent form. If errors occur, such as clocks not being perfectly synchronized, the logs can be misleading. In order to analyze errors, a large amount of additional information must be logged that may not be relevant to the analysis of the problem at hand. These log messages further complicate troubleshooting by distracting from the relevant messages.

In practice, error analysis often takes the form of noticing misbehavior at a certain point. In the best case, this misbehavior can be detected in the logs. From this point, the developers perform a reverse search and try to identify how the error occurred. If the application is modeled in the form of a C&C application, the modeled data flow yields additional information that can narrow down the error search: by knowing which component exchanges data with which other components, log messages can be filtered.

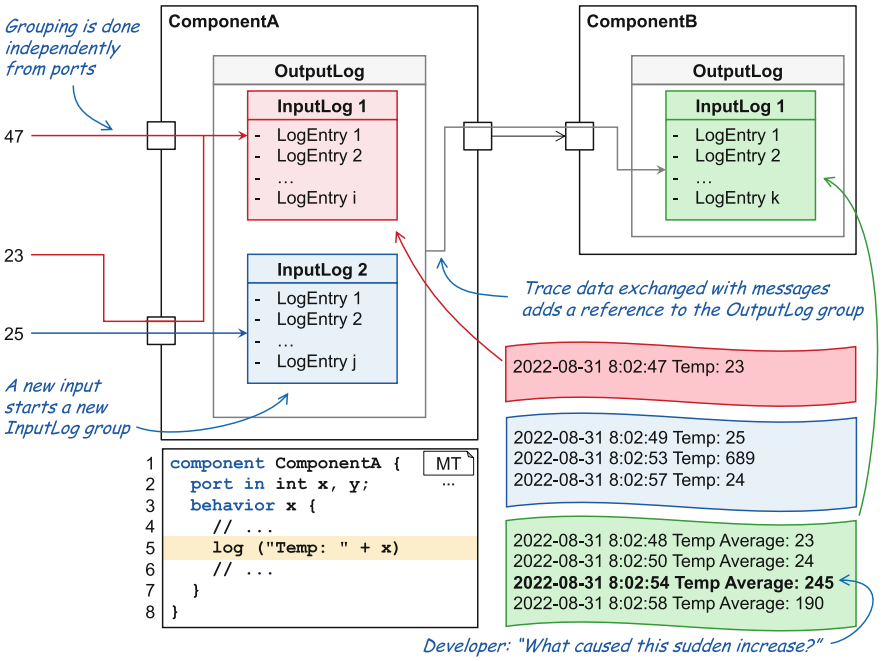

Fig. 18 MontiThings correlates log messages from interacting components. Thus, in large logs, it is possible to trace which logs have led to the generation of a log message. Figure taken from [21] and based on [23]

MontiThings offers a tool for this that lets developers interact with the real system at runtime. The logs of each individual component are displayed. If a developer clicks on a log message, it is displayed which other log messages are related to this log message. For this purpose, a graph is built that graphically represents the architecture, reducing it to the relevant communication paths.

Technically, this works as shown in Fig. 18. When a message arrives at a port, MontiThings starts to bundle log messages. A unique ID is assigned for each bundle. If the component now sends a message on a port in response to the incoming message, the ID of the current bundle of log messages is also sent. In this way, a graph structure of bundles of log messages can be created. When a developer asks for the origin of a particular log message, the log system communicates with the components to get the log messages associated with the IDs. Details about this process can be found in [21].

7 Conclusion

Developing distributed IoT applications based on heterogeneous, error-prone IoT devices is complex. GPLs are not designed for this task. Model-driven approaches promise to make this problem manageable through abstraction. In this chapter, we presented *MontiThings*, a model-driven ecosystem for developing, deploying, and analyzing IoT applications. MontiThings also outlines an app store concept that decouples hardware and software development. Overall, MontiThings' deployment algorithm and app store concept help give device owners more control over their devices. By negotiating deployment with device owners, the deployment algorithm increases the flexibility of IoT systems. Possible future work includes more automated exploitation of cloud services, integration of user-defined behavior (including, e.g., through Large Language Models), and generation of user-understandable explanations for system behavior.

Source Code

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

Acronyms

C&C	Component and connector 45
CD	Continuous deployment 61
CD4A	Class diagrams for analysis 50
CI	Continuous integration 170
DTIS	Digital twin information system 57
GPL	General-purpose programming language 46
IoT	Internet of Things 45
OCL	Object constraint language 50
SI	International System of Units 50

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

References

1. Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S.: 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). Lecture Notes in Informatics, vol. P-304, pp. 59–66. Gesellschaft für Informatik e.V., Bonn (2020)

2. Ahmad, S., Mehmood, F., Mehmood, A., Kim, D.H.: Design and Implementation of Decoupled IoT Application Store: A Novel Prototype for Virtual Objects Sharing and Discovery. Electronics **8**(3) (2019)
3. Angelsmark, O., Persson, P.: Requirement-Based Deployment of Applications in Calvin. In: Žarko, I.P., Broering, A., Soursos, S., Serrano, M., (eds.) Interoperability and Open-Source Solutions for the Internet of Things, pp. 72–87. Springer International Publishing, Cham (2017)
4. Brooks, C., Jerad, C., Kim, H., Lee, E.A., Lohstroh, M., Nouvelletz, V., Osyk, B., Weber, M.: A Component Architecture for the Internet of Things. Proc. IEEE **106**(9), 1527–1542 (2018)
5. Bröring, A., Schmid, S., Schindhelm, C.K., Khelil, A., Käbisch, S., Kramer, D., Le Phuoc, D., Mitic, J., Anicic, D., Teniente, E.: Enabling IoT Ecosystems through Platform Interoperability. IEEE Softw. **34**(1), 54–61 (2017)
6. Butting, A., Kirchhof, J.C., Kleiss, A., Michael, J., Orlov, R., Rumpe, B.: Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In: Proceedings of the 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22), pp. 108–121. ACM, New York (2022)
7. Corno, F., De Russis, L., Sáenz, J.P.: How is Open Source Software Development Different in Popular IoT Projects? IEEE Access **8**, 28337–28348 (2020)
8. Dalibor, M., Jansen, N., Rumpe, B., Schmalzing, D., Wachtmeister, L., Wimmer, M., Wortmann, A.: A cross-domain systematic mapping study on software engineering for Digital Twins. J. Syst. Softw. **193** (2022)
9. Dias, J.P., Restivo, A., Ferreira, H.S.: Designing and constructing internet-of-Things systems: An overview of the ecosystem. Internet of Things **19**, 100529 (2022)
10. Eclipse Foundation: IoT Developer Survey 2020. [Online]. Available: https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020 (2020). Last accessed 20 June 2021
11. Eclipse Mita Project Website: [Online]. Available: https://www.eclipse.org/mita/. Last accessed: 13 April 2023
12. Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., Rumpe, B., Varga, S.: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Michael, J., Bork, D., (eds.) Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, pp. 22–30. CEUR Workshop Proceedings (2020)
13. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: 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, pp. 125–135. ACM, New York (2016)
14. Hermerschmidt, L., Perez, A.N., Rumpe, B.: A Model-based Software Development Kit for the SensorCloud Platform. In: Workshop Wissenschaftliche Ergebnisse der Trusted Cloud Initiative, pp. 125–140. Springer, Schweiz (2013)
15. Hölldobler, K., Kautz, O., Rumpe, B.: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, Herzogenrath (2021)
16. Kirchhof, J.C.: Model-Driven Development, Deployment, and Analysis of Internet of Things Applications. Aachener Informatik-Berichte, Software Engineering, Band 54. Shaker Verlag, Herzogenrath (2023)
17. Kirchhof, J.C., Michael, J., Rumpe, B., Varga, S., Wortmann, A.: Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 90–101. ACM, New York (2020)
18. Kirchhof, J.C., Malcher, L., Rumpe, B.: Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Tilevich, E., De Roover, C. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), pp. 197–209. ACM SIGPLAN, New York (2021)
19. Kirchhof, J.C., Kleiss, A., Michael, J., Rumpe, B., Wortmann, A.: Efficiently Engineering IoT Architecture Languages—An Experience Report (Poster). STAF 2022 Workshop Proceedings:

10th International Workshop on Bidirectional Transformations (BX 2022), 2nd International Workshop on Foundations and Practice of Visual Modeling (FPVM 2022) and 2nd International Workshop on MDE for Smart IoT Systems (MeSS 2022) (co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2022)) (2022)
20. Kirchhof, J.C., Kleiss, A., Rumpe, B., Schmalzing, D., Schneider, P., Wortmann, A.: Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. ACM Trans. Internet of Things **3**(4) (2022)
21. Kirchhof, J.C., Malcher, L., Michael, J., Rumpe, B., Wortmann, A.: Web-Based Tracing for Model-Driven Applications. In: Proceedings of the 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA'22). In Press (2022)
22. Kirchhof, J.C., Rumpe, B., Schmalzing, D., Wortmann, A.: MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. J. Syst. Softw. **183**, 111087 (2022)
23. Malcher, L.: Reconstructing the Behavior of Cyber-Physical Systems through Digital Shadows and Deterministic Replay in Component & Connector Architectures. Master Thesis. RWTH Aachen University. Software Engineering Group (2021)
24. Morin, B., Harrand, N., Fleurey, F.: Model-Based Software Engineering to Tame the IoT Jungle. IEEE Softw. **34**(1), 30–36 (2017)
25. Munjin, D., Morin, J.-H.: Toward Internet of Things Application Markets. In: IEEE International Conference on Green Computing and Communications, pp. 156–162 (2012)
26. Nguyen, X.T., Tran, H.T., Baraki, H., Geihs, K.: FRASAD: A framework for model-driven IoT Application Development. In: IEEE 2nd World Forum on Internet of Things (WF-IoT), pp. 387–392 (2015)
27. Node-RED—Low-code programming for event-driven applications: [Online]. Available: https://nodered.org. Last accessed 13 April 2023
28. Persson, P., Angelsmark, O.: Calvin – Merging Cloud and IoT. Procedia Comput. Sci. **52**, 210–217 (2015). 6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015)
29. Persson, P., Angelsmark, O.: Kappa: Serverless IoT deployment. In: Proceedings of the 2nd International Workshop on Serverless Computing, WoSC '17, pp. 16–21. Association for Computing Machinery, New York (2017)
30. Rumpe, B., Michael, J.: Digital Twins 2.1. [Online]. Available: https://www.se-rwth.de/essay/Digital-Twin-Definition/. Last accessed 05 April 2023
31. Serror, M., Kirchhof, J.C., Stoffers, M., Wehrle, K., Gross J.: Code-Transparent Discrete Event Simulation for Time-Accurate Wireless Prototyping. In: Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS '17, pp. 161–172. Association for Computing Machinery, New York (2017)
32. Taivalsaari, A., Mikkonen, T.: A Roadmap to the Programmable World: Software Challenges in the IoT Era. IEEE Softw. **34**(1), 72–80 (2017)