MontiThings: Model-driven Development and Deployment of Reliable IoT Applications

Jörg Christian Kirchhof[a,*], Bernhard Rumpe[a], David Schmalzing[a] and Andreas Wortmann[a,b]

[a]Software Engineering, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany, https://se-rwth.de
[b]Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Seidenstraße 36, 70174 Stuttgart, Germany

ARTICLE INFO

Keywords:
Internet of Things
Model-Driven Engineering
Architecture Modeling
Code Generation
Deployment

Abstract

Internet of Things (IoT) applications are exposed to harsh conditions due to factors such as device failure, network problems, or implausible sensor values. We investigate how the inherent encapsulation of component and connector (C&C) architectures can be used to develop and deploy reliable IoT applications. Existing C&C languages for the development of IoT applications mainly focus on the description of architectures and the distribution of components to IoT devices. Furthermore, related approaches often pollute the models with low-level implementation details, tying the models to a particular platform and making them harder to understand. In this paper, we introduce MontiThings, a C&C language offering automatic error handling capabilities and a clear separation between business logic and implementation details. The error-handling methods presented in this paper can make C&C-based IoT applications more reliable without cluttering the business logic with error-handling code that is time-consuming to develop and makes the models hard to understand, especially for non-experts.

1. Introduction

The Internet of Things (IoT) consists of billions of interconnected cyber-physical devices around the globe that collect and share data. It is the quintessential enabler of various initiatives that will change our world [45], from the interconnected production systems of Industry 4.0 [104], to (automated) vehicles communicating with each other and their environment [94], to collaborating fleets of robots in agriculture [30] or rescue missions [6], to smart homes offering their service over the Internet [86]. Connecting all these different devices will provide new insights about the world around us as well as new means to improve it to our benefit.

From a software engineering point of view, IoT applications are distributed over heterogeneous devices that possibly share their hardware among multiple applications [70], operate in dynamic and uncertain environments, and, in the worst case, can fail to provide their services without notice [95, 68]. Consequently, the development of IoT applications differs from the development of traditional applications [95]. While traditional development tools and methods can also be used to develop IoT applications, these neither provide the abstraction to properly separate business logic and error-handling concerns nor do they provide means to automatically deploy logical IoT applications to heterogeneous devices. The focus of traditional development methods on single, homogeneous platforms that are assumed to operate reliably makes developing and deploying an IoT application complex [74], inefficient [95], and thus error-prone and expensive.

Model-driven engineering (MDE) [39] can mitigate these challenges by lifting more models to primary development artifacts that can abstract from implementation details of the platforms and devices the IoT applications are deployed to and separate concerns into different modeling languages whose models can be integrated and analyzed automatically. To make models machine-processable and, thus, accessible to automated analyses, syntheses, integration, and evolution, these models must conform to explicit modeling languages [55], such as UML [84], SysML [41], or AADL [34].

Languages for the MDE of IoT applications provide little abstraction (*e.g.,* Eclipse Mita[1]) or offer no or only limited mechanisms for handling error situations (*e.g.,* ThingML [70, 51]). Hence, developers are required to specify error handling and business logic in an integrated fashion. This lack of separation of concerns hampers model comprehension, complicating debugging, evolving, and reusing IoT application models [95]. Overall, the question of whether "MDE [can] play a key role in the future of IoT" is still considered unanswered [13].

We present the novel MontiThings modeling infrastructure for the systematic engineering of IoT applications that features (1) integrated modeling languages for architectures of IoT applications, their deployment, and error-handling that lift the level of abstraction in IoT system engineering and separate these concerns properly; (2) a model-driven toolchain for the automated synthesis of executable IoT containers; (3) automated deployment planning, featuring deployment suggestions, for the generated containers; and (4) support to suggest changes to deployment goals based on deployment planning results. The architecture modeling language of MontiThings extends the tried and tested MontiArc [15, 16] architecture description language with new modeling elements for error handling that are logically

*Corresponding author
kirchhof@se-rwth.de (J.C. Kirchhof); rumpe@se-rwth.de (B. Rumpe); schmalzing@se-rwth.de (D. Schmalzing); andreas.wortmann@isw.uni-stuttgart.de (A. Wortmann)

[1]Eclipse Mita: https://www.eclipse.org/mita/

separated from the architecture's business logic, whereas its toolchain for model checking, synthesis of executable containers, and deployment was conceived specifically to support the engineering of comprehensible, reliable, and robust IoT applications. A 2020 survey on IoT reliability [69] divides the problem space of IoT reliability into the categories *device reliability*, *communication and network reliability*, and *application layer reliability*. MontiThings addresses these problem categories either by providing its own mechanisms or by reusing state-of-the-art technologies such as message brokers. To avoid human errors, MontiThings tries whenever possible to apply the used reliability mechanisms without the developer's involvement.

Consequently, the contributions presented in this paper are

1. the MontiThings modeling language for reliable IoT applications, which separates the concerns of behavior modeling and error handling,
2. a model-driven infrastructure that produces executable containers from MontiThings models,
3. a logics-based, automated deployment of those containers, and
4. a runtime environment for MontiThings component realizations that automates operating with unreliable devices.

Overall, the modeling method presented in this paper can facilitate the engineering of IoT applications by increasing abstraction, separating concerns, and facilitating their deployment to heterogeneous devices.

In the remainder, Section 2 introduces preliminaries and Section 3 discusses requirements for reliable and robust IoT applications. Section 4 presents the MontiThings C&C language, and Section 5 introduces its features for automatic deployment and recovery. Afterward, Section 6 presents a twofold evaluation featuring a case study and observations on scalability. Section 7 debates related work, and Section 8 discusses other approaches to IoT modeling. Section 9 concludes.

2. Background

The modeling technique and infrastructure for IoT systems presented in this paper leverage principles from MDE and software language engineering (SLE). For realization of the infrastructure, we exploit the MontiCore language workbench and extend the MontiArc architecture description language. This section introduces these preliminaries.

2.1. Model-Driven Engineering

There is a conceptual gap [39] in software engineering: software generally should support domain experts (accountants, biologists, lawyers, mechanical engineers, medical doctors, *etc.*) to solve challenges from their specific problem domain. Yet, software is developed by software engineering experts, which rarely have the deep problem domain understanding of the respective experts. Consequently, domain issues are less well understood and reflected in their software solutions. Domain experts rarely are professionally trained software engineers. Consequently, if domain experts are able to contribute software solutions at all, these rarely leverage state-of-the-art methods for their maintainability, modularity, verifiability, *etc.*

This gap can be reduced by using domain-specific models, which are well understood by the domain experts, as primary development artifacts and translating these to software using automated analyses and syntheses that leverage state-of-the-art solutions in the process. To make these domain-specific models machine-processable, *i.e.,* applicable to automated analyses and syntheses, MDE relies on domain-specific languages (DSLs)[2], which describe the syntax and semantics of these languages such that meaningful [39] analyses and translations into program code can be conduced automatically. Examples of popular DSLs are MATLAB Simulink [9] for the engineering of cyber-physical systems, HTML + CSS for web design, SQL [25] for database interaction, the Icam Definition for Function Modeling (IDEF0) [76] for functionally modeling the behavior of cyber-physical systems, or SysML [34] for interdisciplinary systems engineering.

Such DSLs can be textual [102, 100] or graphical [34, 66] and they can be defined via grammars [54, 8], metamodels [93, 24], or projectional editors [20]. Moreover, they can feature operational behavior realizations by means of interpreters [24] or translational behavior realizations through model transformations [65, 59]. They also can leverage logics, such as OCL [52] or general-purpose languages (GPLs) [54, 8] to restrict the set of valid models.

From the need for DSLs to reduce the conceptual gap inherent in software engineering and the wealth of ad-hoc solutions to create such languages, the field of software language engineering has arisen.

2.2. Software Language Engineering

SLE aims to systematize the design, engineering, maintenance, evolution, and operations of DSLs realized in software. Generally, a language can be considered as the set of sentences it comprises. Yet, for sophisticated reasoning about languages and, for a constructive application, a more detailed, intensional, definition is required. Research in computer science produced various characterizations for DSLs, such as: "A domain-specific language is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [26] or "By focusing on a problem domain's idioms and jargon, DSLs avoid the notational noise required when using overly general constructs of a general-purpose language to express the same thing. Moreover,

[2] We use the terms domain-specific language and modeling language interchangeably as all modeling languages, even as generic as UML or SysML, target a domain.

DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem." [103]. Notwithstanding the distinction between modeling languages and DSLs, employing such languages promises benefits regarding productivity and quality as DSLs may provide a platform-independent "thinking and communication tool" [101] that can leverage the concepts and terminology the experts using the DSL are familiar with. Thus, the availability of appropriate DSLs can greatly facilitate the systematic engineering of cyber-physical systems.

As software languages are software too [33], these are subject to the same complexities as general software engineering and impose additional complexities through the various meta-levels and metamodeling technologies language engineers have to operate with. Hence, various language workbenches [32], software specific for the purpose of engineering DSLs, have been produced. These language workbenches provide sophisticated (meta-)modeling languages that enable describing the syntax or semantics of DSLs. For syntax, metamodels and grammars are widely applied to language engineering: For instance, ECore of the Eclipse Modeling Framework [92] is a modeling language for the specification of metamodels. A metamodel of a DSL governs the structure of the DSL's abstract syntax in terms of classes, their properties, and relations. Models conforming to these metamodels are considered valid elements of the DSL. The GEMOC Studio [24] language workbench and others use ECore for the definition of abstract syntaxes and provide features, such as debugging, editing, or interpretation, based on it. Other language workbenches, such as MontiCore [54], Neverlang [99], or Xtext [8] employ grammars as syntax metamodeling languages instead. These grammars support prescribing the integrated concrete and abstract syntax of languages through their productions as the set of derivable sentences. Often, these syntax modeling languages are context-free [22], *i.e.,* they cannot express arbitrary well-formedness restrictions. To define well-formedness rules, language workbenches often employ OCL constraints [52] or GPL context conditions [54, 99]. For semantics, language workbenches often focus on model transformations leveraging mature transformation frameworks, such as ATL [58], epsilon [65], or MOLA [60], to define translations of models into other formalisms (including GPL code)

Leveraging research in SLE and powerful language workbenches, a variety of DSLs have been produced for, *e.g.,* automotive software engineering [56, 27], avionics [29], business processes [46], Internet of Things applications [42], manufacturing systems and processes [104] software engineering [48, 67], robotics [98, 73], and many more.

2.3. MontiCore Language Workbench

MontiCore [54] is a language workbench for the efficient engineering of textual DSLs. To this end, it employs grammars for the integrated definition of abstract syntax and concrete syntax of DSLs that define which models are principally valid. MontiCore generates a comprehensive model processing infrastructure from each grammar, including the abstract syntax classes representing models in machine-processable form, parser, lexer, a model checking framework, and a template-based code generation framework. Java well-formedness rules can be integrated into the model checking framework. This enables restricting the set of valid models of a MontiCore DSL further. MontiCore's template-based code generation framework supports translating valid models to other formalisms, including executable GPL artifacts. To facilitate the efficient engineering of DSLs, MontiCore supports compositional language integration in form of language extension, language embedding, and language aggregation [53, 49].

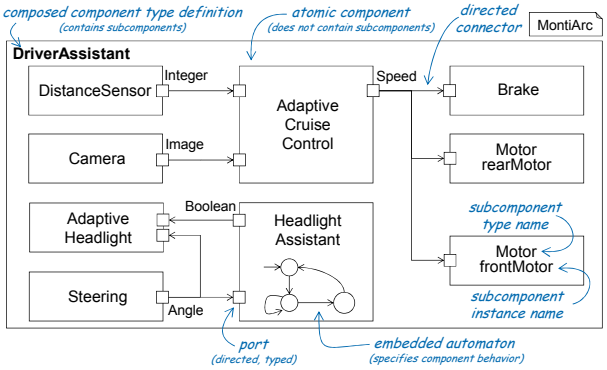


Figure 1: MontiArc architecture describing driving assistants of a car.

2.4. MontiArc

MontiArc [15, 50] is an architecture description language (ADL), for the MDE of cyber-physical systems.

MontiArc specifies architectures as networks of components that exchange data with each other via typed and directed ports. Hence, each component is a black box that can only exchange data via its ports. Composed components yield a behavior by instantiating other components as subcomponents and connecting these. Atomic components do not contain other components and instead define their behavior using embedded behavior descriptions. MontiArc models can be used to generate Java code [82] for simulation purposes or to generate Python code [1] for robotics applications.

Figure 1 shows an example of a MontiArc architecture describing driving assistants of a car. A `DistanceSensor` and a `Camera` continuously provide the distance to the front and an image to the `AdaptiveCruiseControl` component. The `AdaptiveCruiseControl` uses this information to calculate new target speeds and inform the `Brake` and `Motors` about the new target speed. An `AdaptiveHeadlight` uses the current steering angle provided by the `Steering` component and instructions on whether to turn on the headlights provided by the `HeadlightAssistant` component. Whether these components are composed or atomic is transparent to their use and, hence, not depicted

here. As MontiArc components can be hierarchically composed, and of these might feature arbitrary (but finitely) many subcomponents on arbitrary (but finitely) many hierarchy levels. The `HeadlightAssistant`, in contrast, is an atomic component that uses an automaton to model the different states of the headlight switch of the car.

MontiArc is based on the FOCUS [12] semantics for stream processing functions. Hence, each component realizes a stream processing function and composed components are actually composite functions. This semantic basis enables powerful analyses rarely supported by other ADLs, such as automated semantics-aware decomposition [63] and refinement [17], that have been applied to architecture modeling for various domains, including automotive [27], cloud [71], and robotics [18].

3. Requirements

Engineering cyber-physical systems, such as IoT applications, is non-trivial using traditional programming languages. Besides the inherent complexity of programming multi-device applications for heterogeneous platforms, developers often fall for invalid assumptions such as the assumption that devices never fail [95]. To support the development of IoT applications, IoT development languages need to sufficiently abstract from low-level details and provide developers the necessary mechanisms to tame the complexity of IoT application development. Based on an analysis of related work and existing commercial solutions (*cf.* Sec. 7) and the differences between the development of IoT applications and traditional applications in [95], we derived the following set of requirements for the C&C language:

(R1) Error handling. *The modeling infrastructure shall enable error handling without polluting the business logic*. As IoT applications have to handle a wide variety of errors ranging from hardware errors to programming errors to network outages, the language needs to support handling these errors. When the errors are handled within the application's business logic, the business logic quickly becomes incomprehensible because error handling makes up a large part of the code [95].

(R2) Heterogenous hardware. *The modeling infrastructure shall facilitate modeling applications for different platforms.* IoT applications involve heterogeneous devices with different capabilities [95, 68]. Any assumptions that the generator makes about the target platform, *e.g.,* the operating system, restrict the devices on which the components run. In some cases such assumptions are unavoidable. For example, a component that uses a temperature sensor may only be executed on devices that are actually equipped with a compatible temperature sensor. The modeling infrastructure should allow the modelers to express such requirements to prevent invalid deployment. Furthermore, the language needs to abstract from platform-specific details wherever possible and offer means to select different implementations for different platforms.

(R3) Dynamic deployment. *The system should automatically (re)deploy (software) components when devices are added or removed from the system.* Besides the fact that users may wish to add new devices to the system at runtime and that devices may fail, this also enables users to take their software with them, given that the necessary devices are available. For example, this would enable hotel guests to use the smart home applications they use at home in the hotel, provided that the hotel room has the necessary equipment. This idea of moving software components between devices is sometimes referred to as *liquid software* [97, 96].

(R4) Resilience to temporary failures. *The system shall adapt to temporary device outages*. IoT devices are often battery-powered [38] and have to rely on unreliable network connections [95]. As a result, the system may not assume that two devices are constantly able to communicate with each other. Especially, some devices may be deliberately shut down to save energy, and two devices that might need to communicate might not be turned on at the same time. The system must be able to handle such temporary unavailabilities.

(R5) Resilience to permanent failures. *The system shall support (semi-)automatically adapting to hard- and software failures*. IoT systems need to be resilient to failures [95, 68, 57]. Since systems may contain essential equipment without which the system cannot function, the system should facilitate recovery from failures.

Overall our requirements reflect the industries' need for reliability in IoT applications [78, 3].

4. The MontiThings Modeling Infrastructure

From a structural point of view, C&C ADLs essentially model connections, *i.e.,* data exchange, between components. Similarly, IoT applications need to design data exchange between everyday objects. Unsurprisingly, many existing approaches for the model-based development of IoT applications, *e.g.,* ThingML [70, 51] and Calvin [74], are C&C ADLs. We introduce MontiThings, a C&C ADL for modeling IoT. In contrast to existing languages, MontiThings focuses especially on error resistance. Therefore, we decided not to base MontiThings on an existing IoT language but instead extend MontiArc. As MontiArc is based on the FOCUS calculus, MontiArc models can be verified [62], systematically refined [85], and semantics-aware refactored [17].

MontiThings' extensive modeling infrastructure supports the development of IoT applications from the implementation of individual components to managing redeployment and services during system execution. The process for

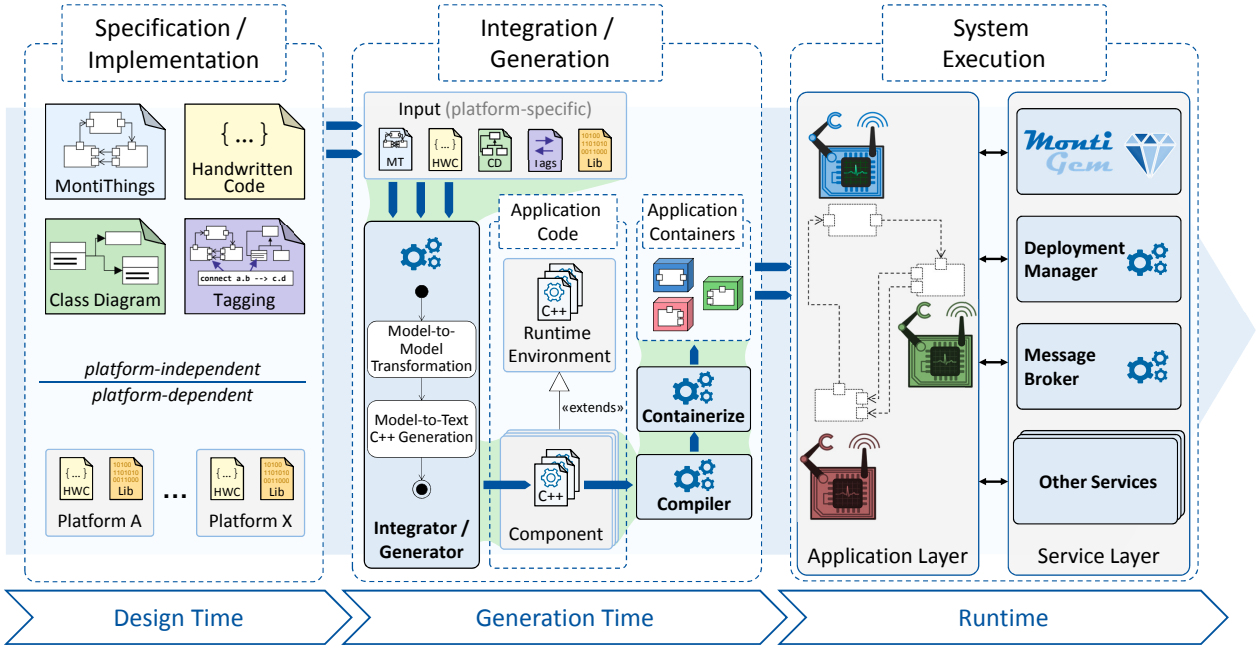
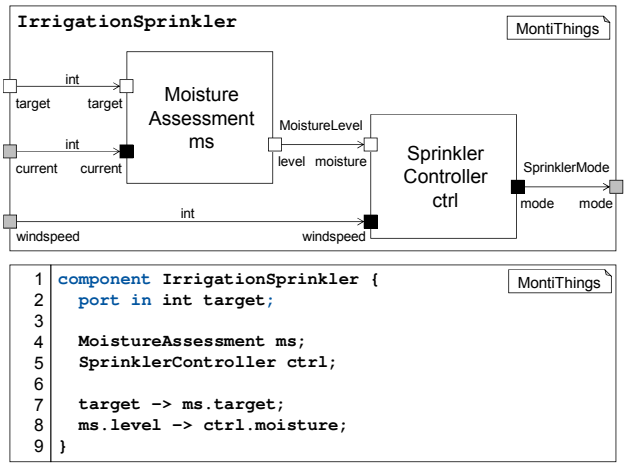


Figure 2: Overall development process from implementing systems through models and handwritten code at design time, to generating application code from these, to deploying and executing distributed systems.

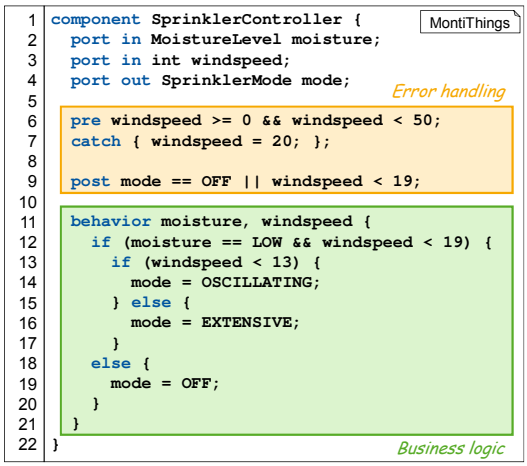
developing IoT applications with MontiThings is visualized in Fig. 2. At design time, architects model MontiThings components that represent the platform-independent, logical architecture of IoT systems. To this end, MontiThings employs four different types of artifacts, namely component models, class diagrams, tagging models, and handwritten code. Each component model defines a system respectively component type, including its interface and behavior. Here, classes from a class diagram provide possible types to be used in the interface of components, therefore defining possible messages between components. The behavior of a component is defined through its decomposition into subcomponents, or, in case of atomic components, through embedded behavior descriptions. Furthermore, component behavior can be implemented, or overwritten through handwritten code in the target language of the used code generator. Finally, tagging models can be used to add additional properties to components. While component models in general can be defined independently of any platform, platform-dependent code may be needed to access specific services. Therefore, handwritten code in component models can be exchanged with respect to the target platform, providing platform-dependent code accessing respective libraries and services.

The development artifacts at design time are input to code generation that generates the target code for the individual components, integrates it with the handwritten code, compiles the code, and finally packages the components in application containers that can then be executed on IoT devices. During generation, component models are preprocessed by transformations, adding explicitly derivable properties and preparing the models for code generation before they are then translated into the target language. Since C/C++ is one of the most popular languages for developing IoT applications [28], and platform-specific and other handwritten code should be easily integrated with the models, MontiThings translates the models to C++ code. The generated code furthermore extends an existing runtime environment that addresses problems common to any component, such as management of incoming messages. After integration with handwritten code, the application code is then compiled though a selected compiler and packaged into shippable and deployable units of software during containerization.

To execute the applications, the containers are distributed to individual IoT devices, considering hardware requirements (*cf.* Sec. 5). For this purpose, hardware requirements, distribution requirements, and the available devices are modeled. If a valid distribution is found, the application containers are directly deployed accordingly. At runtime, these then communicate given the previously defined architecture to provide their services. In addition, the applications can access other predefined services, such as MontiGem [2, 43], to store and visualize application specifics, or a deployment manager that handles redeployment of components in case of device failures. We implemented all aspects presented in Figure 2 and Figure 6 in at least a prototype implementation and provide an overview of the design time aspects in the following and present in more detail deployment and error recovery in Section 5.1.



(a) Component model of an irrigation sprinkler in graphical and textual notation.



(b) Atomic component model of a sprinkler system containing a behavior description and error handling.

Figure 3: Irrigation sprinkler that commands the sprinkler mode based on the current wind-speed and moisture. Assesses the current level of moisture based on the desired target moisture. Black ports mark interaction with the environment using code templates (*cf.* Sec. 4.4). Grey ports and their connectors can be added automatically (if desired) (*cf.* Sec. 4.4).

4.1. MontiThings ADL

In MontiThings, a system is modeled by hierarchically composed components that expose their behavior through their interfaces, consisting of typed, directed ports. Components are composed through connectors between their interfaces, describing the interactions of these. The strong type system of MontiThings is divided into component types, which are defined by component models and can be instantiated as subcomponents, and communication data types defined in class diagrams. The definition of component types in MontiThings follows the formal definition of MontiArc component types [80] and is realized through the systematic language extension mechanism of MontiArc [15]. Furthermore, MontiThings builds on Focus [12], a mathematical framework of stream processing functions. As such, each component defines a time-discrete stream processing function [81], or, in case of underspecification or variability, a set of stream processing functions. A component's behavior is a time-discrete relation of incoming and outgoing messages, *i.e.,* messages the component receives via its incoming ports and messages the component emits via its outgoing ports. For composed components, the component's behavior is defined by its composition of subcomponents, whereas for atomic components, behavior is defined through embedded behavior descriptions [19], such as embedded I/O^* automata [80, 83].

While components are mostly represented in a graphical form for better visualization, component types in MontiThings are implemented in textual descriptions. The textual syntax of component descriptions and the mapping to the otherwise used graphical representation is exemplified by the irrigation sprinkler shown in Fig. 3. The example furthermore contains the textual description of a sprinkler controller implemented as an atomic component. Definitions of component types may include port declarations (lines 2-4), pre- and postconditions (lines 6-9), and a behavior description (lines 11-21). Besides processing locally defined variables, statements in behavior descriptions may read and write the component's variables, read from the components' input ports (lines 12, 13), and write on the component's output ports (lines 14, 16, 19). Behavior blocks are executed in the event of new incoming messages. Reading from input ports always accesses the message of the last event on that port and writing on an output port implies emitting a new message over the respective port. Besides behavior descriptions, components can contain pre- and postconditions that define boolean expressions over inputs and outputs, respectively.

4.2. Error Detection and Handling (R1)

The problem space of IoT reliability can be categorized into *device reliability*, *communication and network reliability*, and *application layer reliability* [69]. Local problems such as a failing sensor can be detected, and in some cases solved, by a single device. More complex errors can affect multiple devices, *e.g.,* a network failure or a missing communication partner due to a device failure. This section discusses device and application layer reliability issues that are solved by the device that detects them. More complex failures are considered in Sec. 5.2 and Sec. 5.3.

Sensors may produce erroneous values [69, 61, 40] for numerous reasons, including wrong calibration, harsh environmental conditions, degradation over time [69, 77], and programming errors. Therefore, a modeling language for specifying IoT systems that often contain sensors must offer appropriate countermeasures. MontiThings provides multiple mechanisms to detect and handle errors **(R1)**. Pre- and postconditions can be used to detect invalid inputs and

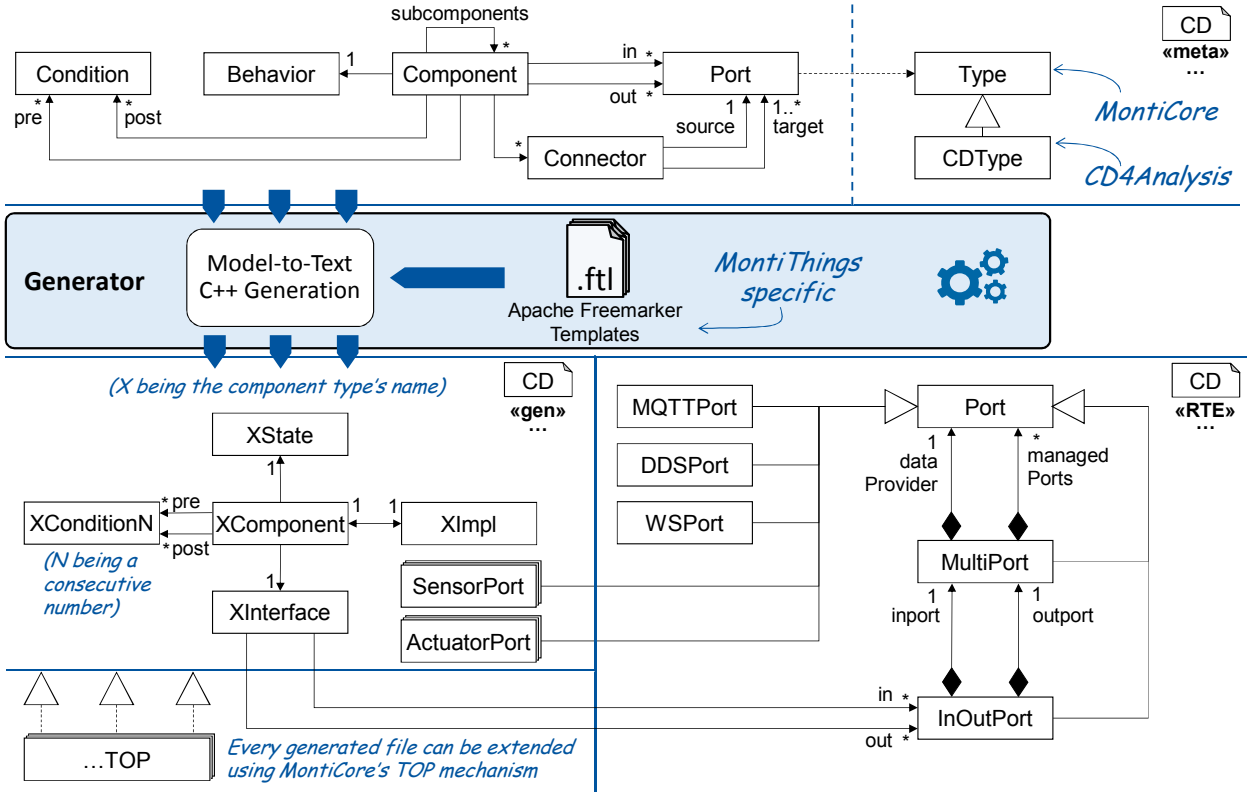


Figure 4: Simplified overview of MontiThings' metamodel, the generated code, and the runtime environment. After parsing, we generate C++ code using Apache Freemarker templates. Each generated file can be extended by hand-written code using the TOP mechanism described in [54].

programming errors before they cause hard-to-locate errors in other parts of the system. Preconditions check that a component's inputs are valid, whereas postconditions inspect the outputs of a component. At runtime, violating pre- or postconditions can be handled using `catch` statements, as known from exception handling in many object-oriented languages. These catch statements can be used to, for example, use a default sensor value if the actually measured sensor value seems corrupted. Unhandled violated pre- and postconditions lead to error messages.

Fig. 3(b) shows a sprinkler controller modeled using MontiThings that demonstrates pre- and postconditions. The precondition in line 6 is checked before the input of the incoming port `windspeed` is forwarded to the behavior implementation. If `windspeed` has a negative value or a value 50 or more, the catch statement in line 7 sets it to 20. After executing the behavior (lines 11-21), the postcondition in line 9 is checked before forwarding the output to the outgoing port. By validating the results after the computation, the postcondition in this example also checks that the sprinkler is turned off in cases where `windspeed` exceeds a threshold.

From the users' perspective, stopping (parts of) a system due to violated pre- and postconditions is a strong intrusion that should only be used as a last resort, *i.e.,* to prevent errors from accumulating. The occasional occurrence of an implausible sensor value should not require such steps. Instead, the system should use reasonable default values as a fallback. What "reasonable" means in this context depends on the particular application. Turning on the hazard warning lights in a vehicle whose sensors incorrectly indicate that the brakes are not working will not cause much damage. Switching on the sprinklers in a building where a single sensor falsely reports a fire can cause considerable material damage. For these reasons, it is up to the modeler to decide which default values are reasonable.

4.3. Runtime Environment and Code Generation

As an extension of MontiArc, MontiThings is implemented using the language workbench MontiCore [54] and the template engine Freemarker. Abstract and concrete syntax are realized through an EBNF-like grammar with MontiCore providing additional infrastructure for language realization, including symbol table, context-conditions, and code generators. A corresponding but simplified metamodel of MontiThings component models is shown in the upper part of Fig. 4. Noteworthy here is that MontiThings relies on MontiCore's language extension mechanism to include already existing language constituents, for example

MontiCore's library of types, and expressions [14]. These enrich MontiThings with capabilities for stating expressions in behavior blocks or types of ports and also provide adequate type checks with models from an aggregated class diagram language (called CD4Analysis) providing corresponding type definitions.

Processing MontiThings component models from textual files to generated code is a multi-step process. First, the model is parsed and translated into an abstract syntax tree (MontiCore provides a fulled fledge parser for a given grammar). Afterwards, the models' symbol table is created, model transformations are executed, and well-formedness is checked. Finally, the models are translated into code through MontiThings specific code generator employing Freemarker templates and accessing information stored in the abstract syntax trees and symbol table. Model-specific code is generated against and directly linked to an existing and predefined runtime environment (RTE), as shown in the bottom part of Fig. 4. The RTE provides common implementations and interfaces for ports, components, and the like, and technology-specific implementations that may be reused in the generated code. For ports, the RTE provides a common interface and common implementations of ports for specific network technologies, such as MQTT. In cases where ports should access hardware or an operating system not supported through the RTE, developers may need to provide respective templates, which are then used during code generation instead. Furthermore, a port may be read using multiple communication technologies, thus managing multiple ports that could be realized using different technologies. Finally, ports in the implementation support both sending and receiving data, especially for composed components where ports may forward data to subcomponents' ports.

During code generation, each component type definition is translated into a set of classes, namely classes for the component's pre- and postconditions, its implementation, its interface, its current state, and their aggregation. Here, the interface of a component bundles incoming and outgoing ports whereas the components' implementation class realizes the component logic either derived from atomic components' descriptions or the composition of subcomponents. Objects of the state class may represent the components state during runtime and furthermore provide functionality for automatic serialization. Employing MontiCore's TOP mechanism [54], developers may extend and override any generated file, such as the components implementation, without losing changes on re-execution of the code generator.

4.4. Integrating the System with Its Environment and Other Systems (R2)

IoT applications require access to sensors and actuators or, more generally, access to hardware. The system boundary of MontiThings is exactly this hardware access. Consequently, hardware access is realized via ports, as these are the interface of components for communication with the outside world. To include implementations for these, developers can provide code templates for ports, which inject

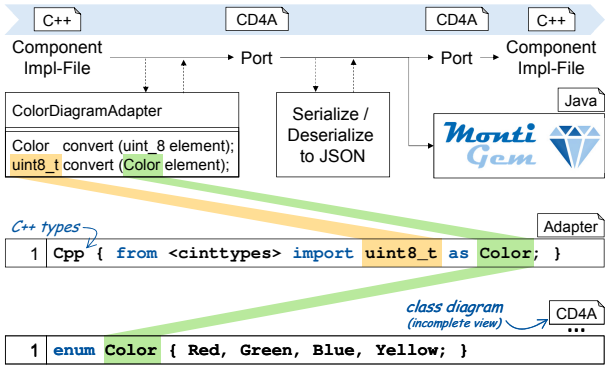


Figure 5: Class diagram adapters allow developers to use custom C++ types in their handwritten code while enabling MontiThings to serialize the data in a standardized way to exchange it with other components or services (here: MontiGem).

behavior into the ports during code generation. The code within these templates must be able to provide data to the architecture (in the case of an incoming port) or process data provided by the architecture for hardware access (in the case of an outgoing port). Different platforms may use different templates to realize the same hardware access **(R2)**. Therefore components are by-design easily testable. As the hardware access only happens within templates, these can be easily exchanged by mocks providing test inputs instead of real sensor values.

To prevent this approach from limiting the composability of components, we use a priority mechanism: Similar to how subclasses may overwrite methods of their superclasses in many object-oriented languages, the composed components that instantiate a component may overwrite the port templates of their subcomponents by connecting the port using a connector. If a port is connected with a connector, only the values provided by this connector will be processed by the component. If there is no such connector, the port uses its code templates. Having unconnected ports without code templates is forbidden. This priority mechanism makes it possible to reuse components whose ports have code templates even in contexts where hardware access is not desired.

Data types of ports, variables, and parameters are defined through classes of UML/P class diagrams [21], a variant of UML class diagrams. Each type defined by a class diagram is translated into an equivalent definition of the desired target language through the use of code generators, with C++ being the target language of MontiThings. While we encourage modeling of data structures to keep models independent of target code, we acknowledge that developers sometimes want to use types from libraries of the specific target GPL. Therefore, MontiThings allows replacing the class diagram types with concrete C++ types in the generated code using tags [47]. Tags are externally defined elements that enrich

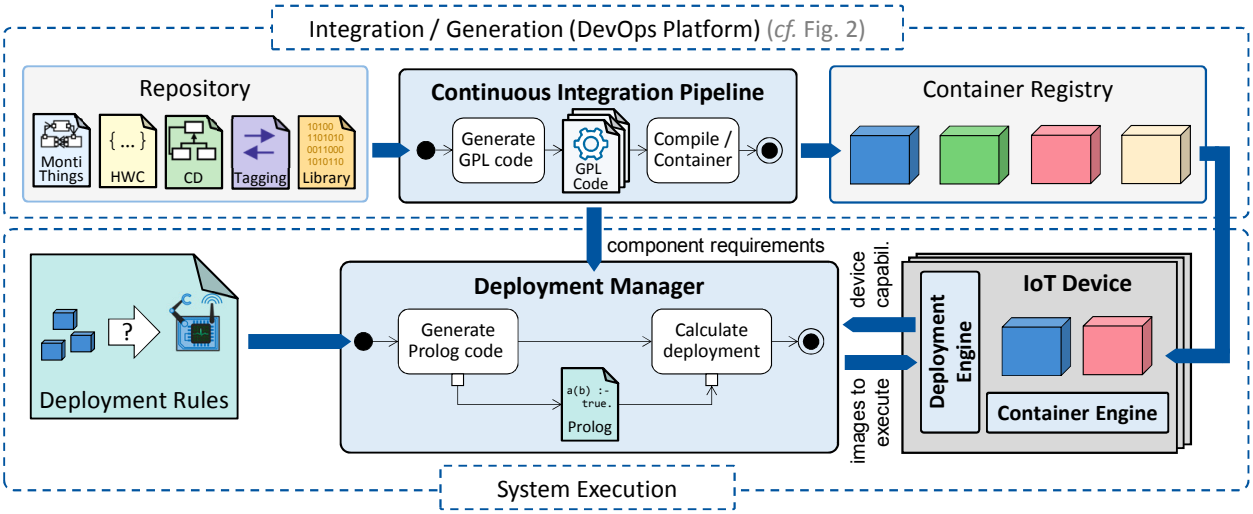


Figure 6: Overall deployment process. Architects push MontiThings architectures to an online repository on a DevOps platform. The DevOps platform generates GPL code from the architecture which it provides to the IoT devices using containers. IoT devices ask a central deployment manager which container images to execute based on the rules defined by the deployer **(R3)**.

a model with additional information. In the case of MontiThings, they target elements of class diagrams and enrich them with information about the generated code. That is, they tag an element (*e.g.,* a class) in a class diagram by referencing its name and list a corresponding element of the target language it is converted to. Fig. 5 exemplifies this. In this example, a class named `Color` from a class diagram is converted into `uint8_t` in the generated code. For each of these tags, the user has to implement two `convert` methods for converting between the GPL type and the class diagram type. When handing data to handwritten code, MontiThings will convert the class diagram types to the GPL types. When exchanging data within the system or with other systems, MontiThings uses the types based on class diagrams.

This approach has multiple advantages. First, the class diagrams for analysis (CD4A)-based types can be schematically serialized and deserialized to, in our case, JavaScript object notation (JSON). This allows us to send the data between ports of different components. Moreover, the serialized data can also be exchanged with other systems. In Fig. 5, the data is exchanged with MontiGem, a tool for the MDE of web-applications. We exchange data between MontiGem and MontiThings to synthesize digital twins. More information on how we keep the physical devices synchronized to digital twins is out of this article's scope and can be found in [64]. Additionally, this approach also fosters underspecification by enabling architects to work with very abstract types (*e.g.,* `Image` or `Temperature`) while programmers can use concrete types. Using abstract types instead of very technical types makes the models less dependent on concrete GPL libraries and makes the models easier to understand by non-programmers.

As mentioned in the introduction of this section, we decided to base our language on MontiArc instead of an IoT-focused C&C language because MontiArc models can be verified. For this, however, the interface of the components needs to reflect all in- and outputs of the system, including those of (unconnected) ports with code templates. To solve this, the unconnected ports of each subcomponent need to be connected to the ports of the composed components that instantiates the subcomponent. To avoid having to forward ports through hierarchy levels manually, MontiThings offers automated model transformations. Through these, all ports of the interfaces of subcomponent instances that are not wired internally in the topology of the composed component are forwarded and become part of the interfaces of the composed components that instantiate them. Ports that do mark interactions with the environment, and those that are automatically forwarded to the composed component, are marked in the graphical representations through black and gray ports, respectively. Furthermore, MontiThings enables automatically creating connectors based on the ports' types and names to support the definition of larger system topologies.

5. Automatic Deployment and Recovery

Traditionally, the decision which hardware executes which software is made by humans, often in advance of writing the software. In future IoT systems, this strategy is no longer feasible. On the one hand, hardware made for specific software may become useless once the developer of the software decides to no longer support the servers needed to execute the software. This is neither economically nor ecologically sustainable. On the other hand, coupling soft- and hardware requires that customers need to unnecessarily

replace their devices with devices similar to those they already own, just because they want to get software from a different vendor. A more sustainable strategy is to monitor and analyze the available hardware continuously and then decide which software to execute on which device. Similar concepts have been proposed under various names, including, for example, liquid software [97] or self-adaptation [23, 7].

In MontiThings, containerized software is deployed to general-purpose hardware based on the software's requirements and the capabilities of the hardware. In addition to that, MontiThings leverages the fact that components in C&C architectures communicate only via clear interfaces to ensure that software components can communicate with each other even if the devices on which they are deployed do not have synchronized sleep cycles and thus are not online at the same time. To bridge short outages caused by battery-saving modes, MontiThings also automatically serializes the states of components. Once they come back online, they deserialize the saved state and can pick up their work where they left off before entering battery-saving mode. In case of an unexpected failure of a device, MontiThings can automatically transfer the software state of its components to other devices.

5.1. Deployment (R3)

Fig. 6 gives an overview of MontiThings' deployment process. The process starts by developing a C&C architecture using the MontiThings language (*cf.* Sec. 4). Each time developers push artifacts into an online repository, a DevOps tool splits the architecture into single components. After splitting the architecture, the DevOps platform generates C++ code from these components. This code is compiled and packed into container images that can be received and executed by the IoT devices. We choose to use GitLab as DevOps platform because it offers Git repositories, continuous integration pipelines (GitLab Runner), and a container registry in one place.

To find out which container images to execute, IoT devices communicate with a central deployment manager. Fig. 7 gives an overview of the deployment manager. Each IoT device informs the deployment manager about its capabilities and characteristics, *e.g.,* available sensors, the device's location, or its operating system. Additionally, a human deployer provides a set of rules the deployment manager has to fulfill during the automatic deployment. The rules are encoded in JSON, similar to the format presented for Calvin in [4].

The deployment manager parses the information provided by the IoT devices and the JSON file containing the rules. Being a language for logic programming, Prolog can be used to deterministically find a valid deployment that fulfills all rules or state that a valid deployment is not possible given the available IoT devices. The generated Prolog program consists of three parts: 1) facts about the current state of the devices, 2) functions for finding distributions that fulfill the deployment rules, and 3) a set of helper functions.

Figure 7: Deployment manager behavior and code generation. Based on the status information received from the devices, facts are generated (1). These facts are used in queries (2) to calculate valid deployments. If a valid deployment is found, the manager tells the devices what to execute by sending them `docker-compose.yml` files (3) corresponding to the deployment.

Using the devices' state information, the manager generates a set of Prolog facts which represent its knowledge about the world (① in Fig. 7). Thereby, each generated prolog fact consists of a fact kind, which specifies what the fact is about, the fact content, *i.e.,* the actual information, and the name of the device to which the fact applies. Further, the manager uses the human-readable JSON rules and generates a set of prolog rules for finding valid deployments (② in Fig. 7). If there is no solution, Prolog considers the deployment query `false`. Finding the distribution consists of two steps: First, the program identifies all devices that fulfill the requirements of a component. Then the program tries to find a set of devices that fulfill all of the constraints given by the user. Besides hardware requirements, these requirements can also consist of location requirements, *e.g.,* require having a fire detector in each room, or arbitrary user-defined properties. Since the deployment manager has global knowledge of the system, numerical requirements can also be expressed, *e.g.,* require having at least three fire detectors in each room. It

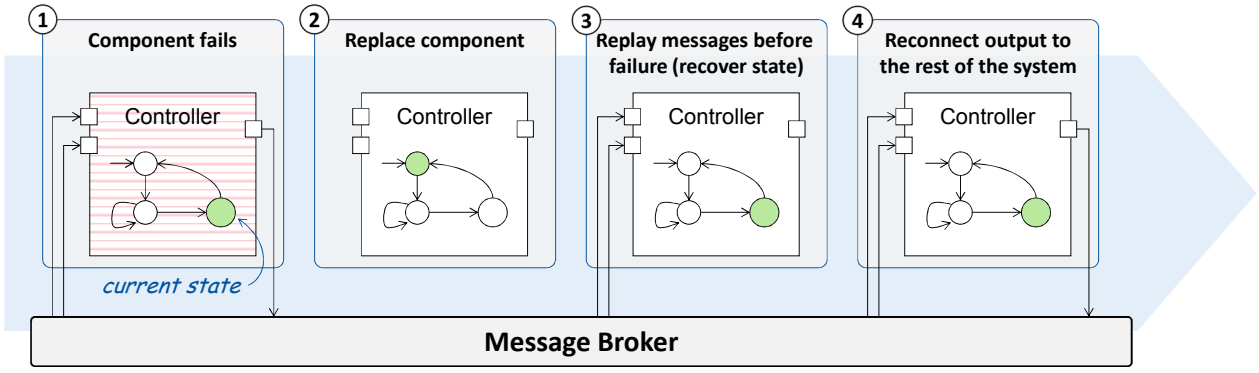


Figure 8: Failure recovery **(R5)**: If a component fails and gets replaced by a new component, MontiThings can automatically recover the state of the failed component by replaying its messages to the new component. To not disturb the rest of the system with the messages generated while replaying old messages, the new component's output port are only connected to the system after the old state has been fully restored.

is also possible to specify that components must not run on the same device. This is sometimes desirable for security reasons.

Additionally, Prolog's backtracking mechanism can also be used to make suggestions on how to repair a set of unsatisfiable rules. This is done by accepting the unsatisfiable rules and documenting what rules are dropped. Moreover, more relaxed versions of rules can be added to make suggestions on how to modify the rules. By sorting them such that the original rule comes before the alternative or dropped rules, Prolog's backtracking mechanism will always choose the original queries over the modified or dropped ones. For example, if a rule requests ten temperature sensors from a system that only has eight, Prolog cannot fulfill the query. Because the query cannot be fulfilled, Prolog then tries to find a deployment with nine and, after that, eight temperature sensors.

If Prolog can find a valid deployment, the deployment manager informs each IoT device which container images to execute. To do so it generates `docker-compose.yml` files telling each device which Docker containers to execute and where to pull the images from (③ in Fig. 7). The IoT devices must report back to the deployment manager regularly. If there is no contact to the deployment manager for too long, the deployment manager will try to deploy the lost components on other devices.

5.2. Fault Tolerance—Temporary Failures (R4)

The fact that IoT devices are often battery-powered is often neglected when designing C&C languages. Since the devices are battery-powered, they frequently need to go into sleep mode to prolong their battery life. This, however, makes them unable to receive or send messages. Systems based on the assumption that all devices are always online could, therefore, suffer from complicated errors. As discussed in [95], another extreme is that developers have to write large amounts of error-handling code. This lack of separation of concerns can make the business logic of an application harder to understand.

MontiThings, in contrast, tries to make the communication between the devices as reliable as possible while, on the other hand, hiding as much of the work required for that from the developers and architects. A key characteristic of C&C architectures is that components only communicate via defined interfaces, *i.e.,* their ports. This characteristic is an advantage for us to make communication reliable.

To ensure that components always have a reliable communication partner, we introduce a central message broker to the system. Cloud providers such as Microsoft Azure or Amazon web services (AWS) can provide such message brokers with high availability. Therefore, we do not consider the broker to be a single point of failure. For efficiency, MontiThings normally moves messages between ports of components deployed within a single container only in memory. However, it replaces the ports of the outermost component within each container by communication with a message broker. In our prototypical implementation, we utilize an Eclipse Mosquitto MQTT broker. A message broker persisting messages between the devices can bridge temporary device outages **(R4)** by allowing them to receive messages once they come back online. Conceptually, this can be considered an application of the device shadow pattern [79]. Normal MQTT brokers neither persist messages nor answer requests from the devices. Thus, we add a service in the form of an MQTT client that can handle such requests from the IoT devices.

5.3. Fault Tolerance—Permanent Failures (R5)

Besides handling temporarily unavailable devices, MontiThings also leverages its service added to the message broker to handle failing components. As components in C&C architectures may only communicate with the outside world via their ports, we can use these messages to restore the states of devices. Such persisting of state-changing events is sometimes referred to as *event sourcing*. The possibility of

reconstructing the states of actors in actor-based programming using event sourcing has previously been discussed, *e.g.,* in [31]. Usually, such event sourcing requires developers to define state-changing events and provide them to the event sourcing system. As components in C&C architectures already have a defined interface, MontiThings can do this automatically.

Fig. 8 gives an overview of how MontiThings handles failing components. When a component fails (1), the deployment manager will notice that the component does not respond anymore and request a different device to execute the failed component (2). Then, MontiThings replays the messages of the failed component that were persisted by the service added to the message broker (3). During replaying the messages, only the input ports of the new component are connected to the message broker. This is done so that the output produced by the new component during this state recovery phase is not forwarded to components that already received and processed the output when the failed component sent it. After all messages have been replayed and the state of the failed component has been restored, the outgoing ports of the new component are connected to the message broker (4).

While this approach is conceptually simple, it gets slower as more and more messages are processed by a component, *i.e.,* $\mathcal{O}(n)$, where n is the number of messages to be replayed. To mitigate this, MontiThings generates functions to serialize the current state of a component. Storing the serialized states comes at the price of causing network traffic and requiring storage on the server that stores the states. However, it reduces the effort for recovering from a failure to $\mathcal{O}(1)$ if the state is serialized every m messages, as at most the (constant number of) m messages since the last serialization are required to recover the state.

6. Evaluation

This section evaluates the MontiThings modeling infrastructure. First, we model a smart home application based on academic and commercial components. Then, we deploy that application using our Prolog-based approach. Lastly, we evaluate the scalability of the deployment.

6.1. Modeling Case Study

To demonstrate our architecture language, we chose to model an exemplary smart home as smart homes are a popular use case for IoT systems [78]. Similar to the smart home case study presented in [5], our smart home manages the home's temperature and air quality and handles fires. Fig. 9 gives an overview of the most important components of the smart home. Overall, the case study consisted of 17 component types. As shown in Fig. 9(a), our smart home consists of thermostats (Fig. 9(f)), which homeowners can use to set their temperature preferences. Air conditioning components (Fig. 9(c)) open the windows if either it is too warm or the air quality drops below a certain level. If it is too cold, the heating will turn on. Additionally, our smart home also contains fire extinguisher components. This component uses a fire detector as a subcomponent (Fig. 9(e)). If it detects a fire, it will tell the sprinklers of the home to turn on until the fire is extinguished.

The thermostat and the fire detector are based on commercial solutions: The *Rock und Roll UT522*[3] thermostat and the Google *Nest Protect*[4]. For easier comprehensibility, we reduced the functionality to the core functionality of the devices. The thermostat shows users the current temperature on a display. It has three different temperature modes: Eco, Comfort, and Auto. Eco and Comfort use different temperatures. Auto selects automatically switches between Eco and Comfort using a user-defined schedule. For example, users can choose to use a lower temperature at night when they are asleep (Eco mode) and a higher temperature during the day when they are awake (Comfort mode). The *Nest Protect* serves as a basis for our fire detector. Like the *Nest Protect*, the fire detector has an LED and speaker to read information to the homeowners. The fire detector issues warnings using an LED and a speaker that reads messages to the user. Additionally, the color of the LED is influenced by the battery level of the device and can be used to illuminate the home at night if it detects movement. If smoke or carbon monoxide is detected, the fire detector warns the homeowners using its LED and speaker. For comprehensibility, we, however, omitted more complex functionalities such as communication with a web server that can be used to, *e.g.,* silence warnings. Our approach for such communication with web servers is described in [64].

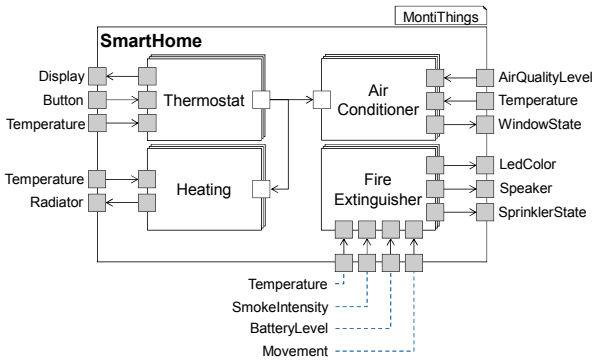
The clear separation between business logic and technical details makes it possible to specify the architecture before knowing which hardware will be used. It keeps the models easier to understand for non-experts. For example, the `WindowController` component (Fig. 9(b)) does not require architects to understand technical details of how to read out the air quality or how to control the actuator of the window, and the technical experts are not required to understand the models and their modeling language but can focus on the technical challenges of interacting with the hardware.

Compared to other ADLs such as MontiArc, Calvin, or ThingML, shifting the technical details of hardware access to ports leads to an overall lower number of required components. Other languages often model hardware access using components whose only purpose is to access hardware and make it available to the architecture (*cf.* Fig. 1). By shifting hardware access into ports, we do not require such components. Forwarding ports automatically (gray ports in Fig. 9) can bring ports to the system's outermost component. Thereby, composed components get more compact as visible by the difference between Fig. 9(c) and Fig. 9(d), where the textual representation only defines one port.

Furthermore, our pre- and postconditions (*cf.* Sec. 4.2) are limited to single components. Nevertheless, some error

[3] https://rockundroll.de/media/pdf_dateien/UT522vorabversion_Horst_Rock_GmbH.pdf

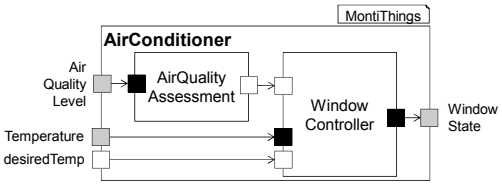
[4] https://store.google.com/us/product/nest_protect_2nd_gen



(a) SmartHome component. Its *Temperature* ports refer to the same port and are only split up for layout reasons.

```
1  component WindowController {
2    port in AirQuality airQuality;
3    port in int temperature, desiredTemp;
4    port out WindowState windowState;
5
6    pre temperature > -20 && temperature < 60;
7    catch { temperature = 22; };
8
9    behavior {
10     if (temperature > (desiredTemp + 10)
11         || airQuality == BAD) {
12       windowState = OPEN;
13     } else if (temperature > desiredTemp) {
14       windowState = HALFOPEN;
15     } else {
16       windowState = CLOSED;
17     }
18   }
19 }
```

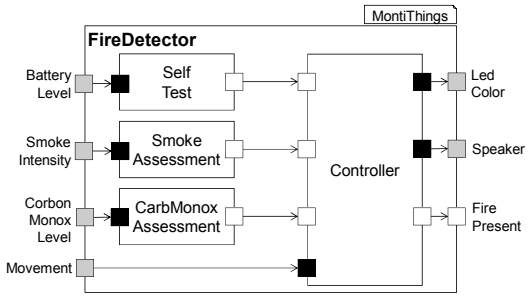
(b) Textual representation of temperature and air-quality-based window opener (based on [5]).



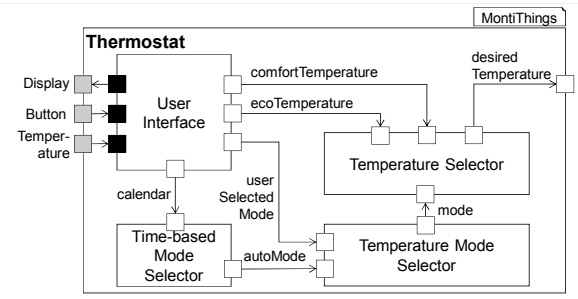
(c) Air quality-based window opener based on [5].

```
1  component AirConditioner {
2    port in int desiredTemperature;
3
4    AirQualityAssessment aqa;
5    WindowController wc;
6
7    desiredTemperature -> wc.desiredTemperature;
8    aqa.airQuality -> wc.airQuality;
9  }
```

(d) Air quality-based window opener based on [5].



(e) FireDetector component based on Google's *Nest Protect*.



(f) Thermostat based on *Rock und Roll UT522*.

Figure 9: Smart home system used as our case study. The components were inspired by either commercial or academic smart home systems. Black and gray ports have the same meaning as those in Fig. 3

situations require a broader view of the system. For example, our case study controls the heating and windows of a smart home. The rules to open the window when the air quality drops below a certain level and to heat the building when it gets too cold individually make sense. However, if the two rules are applied together, this can lead to the windows being opened while the heating is on. Such situations should be avoided but cannot always be identified from the perspective of a single component.

6.2. Deployment Case Study

After generating C++ code from the architecture, building the code, and containerizing the resulting applications, the next step is to deploy these containerized applications. Fig. 10 shows the ground plan of the office to which our distributed application should be deployed. Each blue letter in this figure stands for one hardware device. There are four different hardware devices: Air quality devices (A) contain sensors for measuring temperature and air quality, and an actuator for opening or closing a window. Fire detector devices (F) are modeled after Google's Nest Protect and contain smoke, carbon monoxide, and movement sensors. Further, they have an LED and a speaker. Heating devices (H) contain a display, buttons, a temperature sensor, and a radiator. Sprinklers (S) only contain the sprinkler as their only actuator.

Without any further constraints, our Prolog-based deployment calculates a deployment where each component with a hardware requirement is deployed to the device containing at least the requested hardware. One possible constraint could be to require a `FireDetector` component (Fig. 9(e)) in each room. In this case, Prolog evaluates to `false`, *i.e.,* the request could not be fulfilled because the kitchen and other rooms do not have the necessary hardware to deploy the component. Now, we assume that office six

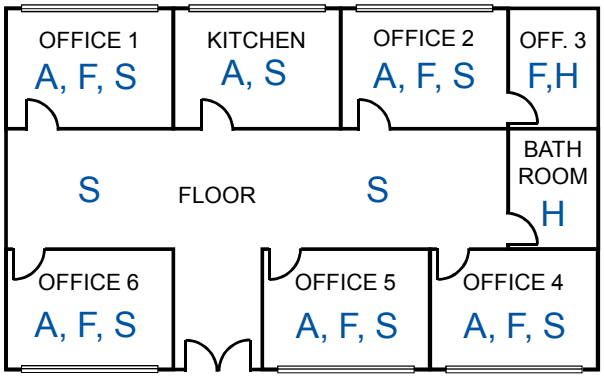


Figure 10: Floor plan for the devices in our case study. A = Airquality, F= Fire detector, H = Heating, S = Sprinkler.

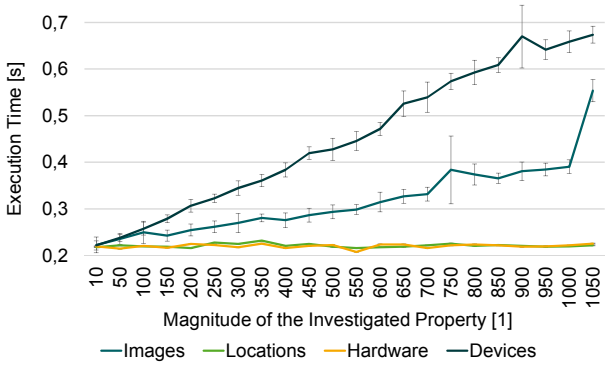


Figure 11: Mean execution time in seconds for finding a valid distribution or falsifying that one exists based on the magnitude of the investigated properties. Error bars denote standard deviation.

contains chemical equipment that might trigger a false positive fire alarm. To prevent false alarms, we set a constraint that requires the number of fire detector components in room `office6` to be zero. Now, our deployment chooses to deploy the `FireDetector` to all rooms with a fire detector device except for the device in `office6`. If we further require at least six fire detector components, Prolog informs us that this request cannot be fulfilled but could be fulfilled if we relax that rule to require only five. Here we can see one limitation of the backtracking mechanism: As the rule to not deploy a fire detector component to office six is fulfilled, Prolog does not suggest dropping this rule to be able to fulfill the other rules. Prolog's backtracking only suggests modifying violated rules to get their corresponding queries to evaluate to `true`. To prevent cooling down the building during winter by opening multiple windows, another constraint requires to deploy at most three `WindowControllers` (Fig. 9(e)). With this constraint, the solution is ambiguous. Prolog found 156 possible solutions in 0.73 s. Theoretically, there are $\sum_{k=1}^{3}\binom{6}{k} = 41$ different solutions for drawing three or less out of six possible elements. The additional solutions found by Prolog come from unnecessarily respecting the order in which the components are deployed (*i.e.,* $\sum_{k=1}^{3}\frac{6!}{(6-k)!}$) as Prolog does not support sets but only lists. As Prolog is based on backtracking, it uses the first solution it finds. In this case, Prolog's first solution is to deploy a `WindowController` *only* in the kitchen, as our requirement only says to deploy *at most* three `WindowControllers`. While this solution is correct, users may consider it unintuitive.

6.3. Deployment Scalability

With the redistribution of systems at runtime, it is particularly relevant how quickly changes can be applied since the system's execution and thus the provision of functionality is interrupted. To investigate the execution time of our implementation of the deployment, we conducted scalability experiments. In these, we consider the runtime for finding a valid deployment from generated prolog facts and queries along three dimensions:

1. increasing number of components (container images) that need to be redistributed
2. increasing number of distribution constraints (hardware or location constraints)
3. and increased number of devices that could be suitable targets for components to be deployed.

The investigation subject is the deployment of components in a single, possible larger office building consisting of multiple locations (a room on some floor). Each location can be equipped with arbitrarily many devices providing various kinds of hardware. The components are to be distributed across the various locations and have certain hardware requirements that may or may not be fulfilled by some device in some room. Furthermore, there are constraints on distributing components across locations, *e.g.,* at least one fire sprinkler per floor or at most 2 smoke detectors per room. Possible characteristics of the restrictions are changes in the components' hardware requirements and the number of locations to which the components are distributed. During the investigations, we examined each dimension in a separate experiment for an increasing magnitude of the investigated property, while the other properties remained constant. That is, to measure the impact of an increase in the number of images to be distributed, several experiments were performed for 10 up to 1050 images to be distributed across 10 devices, each image having up to 10 hardware and location requirements. For each value of a property, ten tests were performed with randomized input. The generated prolog facts and queries were consulted via SWI-Prolog[5], and goals were executed until the first binding succeeding the query was reported or until the goal fails. The reported runtimes indicate the mean waiting time from sending the input to receiving the response.

The results of the experiments are shown in Fig. 11. The mean execution times are reported in seconds against

[5] `https://www.swi-prolog.org/`

an increasing number of components, available hardware, devices, or locations. The plot furthermore shows the standard derivation of the observed execution times for the ten tests performed for each value. The results show that the execution time is mostly affected by an increase in the number of devices, providing more possible targets to choose from. Here, the execution time increases linearly with the number of devices. Execution time for an increasing number of images to be distributed similarly increased linearly; however, a higher number of images to be distributed is less impactful than an increase in the number of devices. An increase in the number of constraints on hardware or locations does not noticeably increase execution time. The standard deviations of the observations are generally reasonable and are due to minor differences in the complexity of the generated inputs as well as measurement errors. However, both the tests for an increasing number of devices and for an increasing number of images show two clearly stronger fluctuations. This suggests that the worst execution time for more complex examples could be noticeably worse than the mean execution time. Since we are not yet able to determine the worst execution time exactly, it would be necessary to implement appropriate safety mechanisms for time-critical systems, which, for example, fall back to appropriate defaults if the execution time is too long.

The experiments show that systems with more than 1000 components or restrictions can still be redistributed within a reasonable time for application areas requiring response times in the millisecond range. As such, our solution performs significantly better than the approach for fleet deployment shown in [91], in which the execution time of deployment assignment for more than two hundred devices is in the double-digit seconds range. For larger, more complex, and time-critical systems, our solution may not be used without adjustments. However, these limitations are not a major limitation for the application of component redeployment. On the one hand, the number of such large and complex systems is probably limited. For large, time-critical systems, the problem could also be broken down into smaller problems through suitable architectural decomposition. This solution would have to be applied mainly when more than 1024 components need to be distributed, as this is the current technical upper limit of our implementation inherited from the underlying technologies.

7. Related Work

Over the last few years, multiple IoT-focused C&C languages have been proposed. ThingML [70, 51] provides a C&C modeling language that allows defining components using statecharts enriched with handwritten code. ThingML uses these models to generate C, Java, or JavaScript code. The handwritten code is either written in one of the target languages or in ThingML's own C-like language. ThingML works on a relatively low level of abstraction where fragments of target GPL code may be included in the model. This makes developing applications using ThingsML easy for developers who already have experiences in the target GPL. The other side of the coin is that the models are often platform-specific and may be impossible to understand for users who do not have experience with the target GPLs[6]. Moreover, coupling models and target languages in the same file fosters creating redundant models for different target languages that only differ in the GPL code. MontiThings instead tries to completely separate the model and the target GPL (*cf.* Sec. 8). Also, ThingML is mainly used to generate application code but does not provide means for the automatic (re)deployment of its components.

In contrast, Calvin [74, 4, 75] is a Python-based C&C language that automatically deploys its components. Calvin's components have requirements, and its devices have capabilities. By matching requirements and capabilities, Calvin decides which devices can execute which components. Additional rules may be used to guide further the deployment, *e.g.,* by specifying that a certain component should be executed not by one but by all compatible actors. MontiThings was influenced by this general idea but moved it to a centralized system to increase the rules' expressiveness and reduce communication between devices (*cf.* Sec. 8). For example, the decentralized approach makes it impossible to request *at least three devices* to run a certain software as devices lack global knowledge about the network.

Similar to Calvin, Distributed Node-RED [10, 44] also uses the approach of matching device capabilities against component requirements. Developers model IoT applications using a web-application. Whenever the application changes, the system informs the IoT devices using MQTT, which will then execute the matching components. As with Calvin, MontiThings allows for more expressive deployment rules, such as requesting at least a specific number of devices to execute a component.

OpenTOSCA for IoT [90] is a deployment tool for MQTT-based IoT applications. In contrast to MontiThings, this approach requires a low-level specification of IP addresses and other technical details of the IoT devices. The benefit of not relying on a container engine is that their approach enables targeting more resource-constraint devices. MontiThings' deployment process requires the IoT devices to have the necessary resources to execute Docker images.

GeneSIS [37, 35, 36] is a model-based tool for IoT application deployment. It supports three types of artifact: ThingML components, Node-RED containers, and blackbox artifacts, *i.e.,* binaries not further inspected by GeneSIS. A deployment model specifies in a graph 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, 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. It also supports deploying software to devices

[6] Example: https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/core/_linux/mqtt/mqtt_client.thingml

not directly connected to the Internet. In comparison, MontiThings currently only supports deploying Docker containers to devices directly connected to the internet. However, GeneSIS does not provide its own reasoner to decide which software to deploy on which device. A future extension point could be to add MontiThings' deployment logic as input to GeneSIS and let GeneSIS manage the technical aspects of the deployment.

CapeCode [11] is a Ptolomy-based development environment that uses that extends Ptolomy by actors using the *accessor* pattern. Accessors are (in summary) actors that access non-modeled software. They serve as a kind of proxy for this software. Similar to MontiThings and Calvin, CapeCope accessors have (abstract) requirements. If a platform meets all requirements of an accessor, it can execute the accessor.

The idea of connecting to hardware through a variant of ports has also been proposed by real-time object-oriented modeling (ROOM) [89, 87]. 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) and other layers of the system. These service access and provision points are connected. Besides that, ports represent communication channels between components. MontiThings differs from that in that there is a single concept of ports. This makes MontiThings components more flexible as their ports (with code templates) do not strictly depend on lower layers or hardware to be available but can also be reused in a different context by attaching a connector to them.

Similarly, Node-RED[7] is a graphical JavaScript-based C&C development tool that connects components that often access external software. For example, a Twitter component can be used to monitor new messages containing a particular hashtag on Twitter. AWS IoT Things Graph[8] provides a graphical C&C modeling tool similar to Node-RED. After modeling the data flow using the C&C modeling tool, users define which components should be deployed to which devices. The deployment can then be carried out automatically by AWS. Overall, AWS provides a whole ecosystem for IoT development, including operating systems and clients for IoT devices (Amazon FreeRTOS and AWS Greengrass), device management (AWS IoT Device Management), and connectivity (AWS IoT Core). Microsoft Azure offers a similar infrastructure, including over-the-air deployment, under the name "Azure IoT Hub". Since most of the projects mentioned above projects are research projects, obviously none of these (including this paper) offers a competitive ecosystem to AWS. However, if we only compare the effort required to generate an executable application from a C&C model, MontiThings can generate large parts of the code for IoT devices that have to be handwritten with AWS. Moreover, MontiThings achieves a clearer separation of business logic and infrastructure of the handwritten code.

[7] https://nodered.org/
[8] https://aws.amazon.com/iot-things-graph/

Eclipse Mita[9] is a programming language for IoT applications that was originally developed by Bosch and itemis as a development tool for the Bosch Cross Domain Development Kit (XDK). Mita generates C code from high-level behavioral descriptions on how to react to certain events. To support a new platform, an Xtend-based code generator for that particular platform has to be provided. The Mita language is, however, very close to its target language C, and mainly sets itself apart from C/C++ by providing a convenient syntax for writing event-handling callbacks. While the language does provide support for MQTT to exchange messages with other systems, the language itself is rather limited and targeted more at generating software for single low-powered devices such as Arduinos or the XDK. Compared to Mita, MontiThings is focused on more complex applications that require, *e.g.,* failure tolerance.

Regarding the deployment of IoT applications, Balena is closest to MontiThings as MontiThings' deployment is based on Balena. Balena[10] is a tool for managing the deployment of IoT applications. It mainly provides an operating system for various popular platforms, called *balenaOS*, a web application for managing devices, called *balenaCloud*, and a failure-resistant, less resource-demanding variant of Docker, called *balenaEngine*. Developers can use Balena to deploy one or multiple Docker containers to multiple IoT devices. To do so, developers push their code to a Git repository provided by Balena. Balena's continuous integration pipeline then creates Docker images from the containers. These images are then pulled by all registered IoT devices using the *supervisor* that handles communication with the cloud. This supervisor is part of balenaOS and thus installed on all devices registered in the balenaCloud. As a commercial service, Balena naturally provides a more mature ecosystem than research projects like MontiThings. MontiThings, however, offers more expressive deployment rules. While Balena deploys the same containers to all devices, MontiThings follows a rule-based approach like Calvin. Since Balena does not foster MDE but mainly focuses on the deployment, MontiThings is better at supporting developers in creating the applications to be deployed. For example, MontiThings automates many error-handling and connectivity tasks that have to be implemented with handwritten code using Balena and thereby also achieves a clearer separation of concerns.

Regarding our requirements (Sec. 3), MontiThings is the only tool mentioned above that fulfills all requirements out-of-the-box. Of course, features such as error-handling or resilience to failures can be implemented using most of the other approaches by integrating handwritten code. This, however, makes the applications more time-consuming (and hence expensive) to implement and test, often leads to a worse separation of concerns, and (in some cases) makes the models harder to understand. By promoting a clear separation of business logic and implementation details, MontiThings models are easy to understand and can be

[9] Eclipse Mita: https://www.eclipse.org/mita/
[10] https://www.balena.io/

used to discuss the product under development between developers and non-developers.

8. Discussion

Language designers can mainly be tempted in two ways to undermine the platform independence of their language: When designing the type system or when designing the behavior specification. Having a certain target GPL in mind, a language designer might consider adding types of the target GPL to the modeling language to enable modelers to use more complex types. Moreover, self-developed behavioral languages are often considered as too inexpressive compared to GPLs. Especially if low-level hardware functionalities are to be implemented, accessing GPL libraries in the model can seem attractive. Both of these pitfalls come at the cost of making it harder to write code generators for different target GPLs. They also require modelers to understand the target GPL and, as more generators are added, to add more and more annotations to the model that make the model more difficult to comprehend.

In an earlier version of the language, MontiThings allowed importing and directly using C++ types as port types. ThingML approaches this problem by including annotations in the model that define how certain types should be handled by the respective code generators (*e.g.,* `@c_type "const char*"`) and which import statements are necessary (*e.g.,* `@c_header "#include <stdio.h>"`). Moreover, ThingML allows tagging components, called "thing" in ThingML, to be specific to a particular target GPL (`@platform "javascript"`). This is mostly done for components that include code of the respective target GPL. For example, a within a ThingML statechart may execute code when triggering a transition. A major drawback is that it can easily introduce code duplication if several platform-specific things need to be maintained that differ only in the included target code GPL.

[95] discusses that developers who create IoT systems often come from a different background (*e.g.,* smartphone app development) and thus make mistakes when developing distributed (IoT) systems that are inherently different. We argue that MDE and IoT languages can help to further abstract from the challenges of developing distributed systems. For example, MontiThings abstracts from device failure by providing means to resolve such situations automatically. However, such abstractions are currently still very limited. Safety-critical applications may require more fine-grained control over latency, jitter, and other network-related parameters than other applications. Such fine-grained control over network parameters is currently out-of-scope for MontiThings. While MontiThings addresses multiple of the problem categories from [69], we want to emphasize that MontiThings is not intended to solve all possible reliability issues. Instead, MontiThings aims at highlighting how model-driven development can facilitate the utilization of reliability mechanisms. Other high-level reliability issues such as anomaly detection [69], which are currently not covered by MontiThings could be added as a service (*cf.* Fig. 2). Low-level mechanisms, such as those affecting communication protocols, could be added by extending the RTE on which the generated code is built.

For MDE platforms to be successful in the IoT domain, more research is needed on MDE platforms to find the *right* level of abstraction that both gives developers fine-grained control when needed but also does not overwhelm them with long lists of parameters they do not need for their specific project. On code-level, MontiThings separates low-level details from the business logic by enabling developers to use handwritten C++ code to define the behavior of components if the developer considers MontiThings' Java-like behavior language not expressive enough. Furthermore, generated code can be overwritten using handwritten code. For the deployment, however, the user does not have such fine-grained options but has to rely on MontiThings' mechanisms.

Moreover, our method currently relies on employing (Docker) containers, which are automatically generated from the models as environments for the component implementations. Yet, it is generally possible to deploy the generated component implementations manually, as we see cases where this would be beneficial, such as operating with very computationally constrained hardware. For such systems, containers might produce undesired computation overhead. Thus, we are investigating lightweight alternatives to containers to provide alternative deployment techniques within MontiThings. Furthermore, our approach does currently not have means of minimizing possible redeployments, *i.e.,* minimizing the number of components that have to be redeployed.

Components in C&C models are black boxes that should only be accessed using their interface, *i.e.,* their ports. This fact enables architects to compose components independently of each other. Also, this fact makes it possible to deploy components to devices independently. While it is technically possible to distribute the components to devices independently of each other, this is often not reasonable. In other words, one has to differentiate between *platform independence* and *platform ignorance* [88]. Similar to Modeling and Analysis of Real-time and Embedded systems (MARTE) [88] and Calvin [4], MontiThings allows to specify requirements to the platform but hides to the developer how these requirements are fulfilled. Also, we give architects the option to limit MontiThings from splitting certain parts of the architecture (*cf.* Sec. 5.1). Overall, however, we conclude that deployment aspects should not be included in C&C models if possible and instead be configured separately.

In practice, however, we noticed that architects, developers, and other stakeholders—even in the early stages of the development—may have specific devices in mind when designing the architecture. This may be due to the fact that IoT products, unlike smartphone applications, as of today are often sold bundled as hardware and software. Standardized

platforms like Apple HomeKit[11] aim at providing standardized access to IoT devices. Nevertheless, many IoT products still exist in their own ecosystems. To be sustainable, future developers will need to make use of the devices their customers already own. This requires that their way of thinking changes from developing software for specific devices to developing software for a specified environment.

This has implications for the design of C&C languages. For C&C architectures to be successfully deployed automatically, architects must be given the ability to specify requirements to their environment. To interact with components not provided by the code generator, other languages often use components that act as substitutes for components not provided by the generator [4, 11]. However, in order to ensure a certain reliability of the resulting product during development, C&C languages should enable architects to specify such substitutes further. If such a specification exists, the compatibility of the self-developed components with the components provided by third parties can be tested. MontiThings uses the mechanisms shown in Sec. 4.2 for this purpose.

Overall, research on IoT deployment is currently "still in its infancy" [72]. MontiThings uses a centralized deployment system in contrast to Calvin's peer-to-peer-based deployment approach [4, 75]. After evaluating the peer-to-peer-based approach, we came to the conclusion that a centralized approach is better fitted for IoT use cases for multiple reasons:

Firstly, IoT devices are often battery-powered. Battery-powered devices enter sleep modes to save energy. Therefore, they can be unavailable for communication for extended periods of time. Furthermore, communication causes high energy consumption. If possible, communication should be avoided. Cloud providers such as Amazon AWS and Microsoft's Azure nowadays provide centralized infrastructure with high availability. Thus one can reasonably expect them to provide central entities for IoT systems. By handling deployment-related tasks using a central entity, we can reduce communication between the devices and thus allow them to enter sleep mode more often.

Secondly, IoT devices are often based on low-performance hardware. While we expect this to become less and less of a factor in the future, devices should not be required to keep parts of the software they do not need available for other devices. This allows IoT devices to allocate more resources to running the actual application.

Thirdly, in a full peer-to-peer approach, no IoT device knows the full topology of available devices. This restricts the expressiveness of the rules used for deploying the components. Without knowledge of the whole system, the requirements can be only based on properties of individual devices, but not on properties of the system as a whole. Since MontiThings' deployment is based on a centralized approach, it can be used to express constraints that require global knowledge about the system, *e.g.,* that a certain number of devices should run a particular component.

[11] https://developer.apple.com/homekit/

Technically, one of the most challenging tasks was creating connectivity between the devices. Many middlewares rely on knowing the IP addresses and port numbers of their communication partners. While this requirement is easily fulfilled in lab environments where devices even can have statically assigned IP addresses, it fails in IoT projects where devices are behind a NAT or can connect via cellular networks and thus have IP addresses that change regularly. Hence, middlewares such OpenDDS explicitly state their discovery protocol is not suited for IoT projects[12], and container orchestration tools such as Kubernetes explicitly require the absence of a network address translation (NAT)[13]. To circumvent this problem, we use a central message broker (Eclipse Mosquitto[14]) in such environments. We think this is reasonable as many IoT applications do parts of their processing in a cloud [78]. While this causes a higher latency compared to decentralized approaches, many IoT applications that communicate over the Internet and not via a local network do not have hard time constraints that would require the slightly lower latency. We also provide decentralized communication between the components, but ultimately, the decision on which networking model to use must be made based on the application's requirements and should be made in advance by the code generator. By extending our `Port` class, developers can, of course, also provide other means of communication.

9. Conclusion

We presented the MontiThings modeling infrastructure for the systematic MDE of IoT applications that supports separation of error-handling from the development of business logic, features a comprehensive model-driven toolchain for generating executable containers, and an efficient deployment even for large IoT systems. With this in place, developers of IoT systems can focus on implementing the business logic first, before they make error handling explicit (instead of merging it into the business parts), and can leverage the power of logic programming to compute optimal deployments for their architectures.

Based on this separation of concerns, we presented a systematic modeling method for reliable IoT applications that leverages sophisticated language, code, and model integration techniques. Overall, this can facilitate engineering IoT systems by increasing abstraction, separating concerns, and facilitating their deployment to heterogeneous devices.

Source Code

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

[12] http://download.objectcomputing.com/OpenDDS/OpenDDS-latest.pdf

[13] https://kubernetes.io/docs/concepts/cluster-administration/networking/

[14] https://mosquitto.org/

Acknowledgements

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

References

[1] Adam, K., Butting, A., Heim, R., Kautz, O., Rumpe, B., Wortmann, A., 2016. Model-Driven Separation of Concerns for Service Robotics, in: International Workshop on Domain-Specific Modeling (DSM'16), ACM. pp. 22–27.

[2] Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S., 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), Gesellschaft für Informatik e.V.. pp. 59–66.

[3] Alkhabbas, F., Spalazzese, R., Cerioli, M., Leotta, M., Reggio, G., 2020. On the Deployment of IoT Systems: An Industrial Survey, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE Computer Society, Los Alamitos, CA, USA. pp. 17–24.

[4] Angelsmark, O., Persson, P., 2017. Requirement-Based Deployment of Applications in Calvin, in: Podnar Žarko, I., Broering, A., Soursos, S., Serrano, M. (Eds.), Interoperability and Open-Source Solutions for the Internet of Things, Springer International Publishing, Cham. pp. 72–87.

[5] Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2020. Msl: A pattern language for engineering self-adaptive systems. Journal of Systems and Software 164, 110558.

[6] Balta, H., Bedkowski, J., Govindaraj, S., Majek, K., Musialik, P., Serrano, D., Alexis, K., Siegwart, R., De Cubber, G., 2017. Integrated data management for a fleet of search-and-rescue robots. Journal of Field Robotics 34, 539–582.

[7] Berrocal, J., Garcia-Alonso, J., Galán-Jiménez, J., Murillo, J.M., Mäkitalo, N., Mikkonen, T., Canal, C., 2017. Situational Context in the Programmable World, in: 2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), pp. 1–8.

[8] Bettini, L., 2016. Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd.

[9] Bishop, R.H., 1996. Modern control systems analysis and design using MATLAB and SIMULINK. Addison-Wesley Longman Publishing Co., Inc.

[10] Blackstock, M., Lea, R., 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, Association for Computing Machinery, New York, NY, USA. pp. 34–39.

[11] Brooks, C., Jerad, C., Kim, H., Lee, E.A., Lohstroh, M., Nouvelletz, V., Osyk, B., Weber, M., 2018. A Component Architecture for the Internet of Things. Proc. of the IEEE 106, 1527–1542.

[12] Broy, M., Stølen, K., 2001. Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Heidelberg.

[13] Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A., 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling 19, 5–13.

[14] Butting, A., Eikermann, R., Hölldobler, K., Jansen, N., Rumpe, B., Wortmann, A., 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology 19, 3:1–16. Special Issue dedicated to Martin Gogolla on his 65th Birthday.

[15] Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A., 2017a. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language, in: European Conference on Modelling Foundations and Applications (ECMFA'17), Springer. pp. 53–70.

[16] Butting, A., Kautz, O., Rumpe, B., Wortmann, A., 2017b. Architectural Programming with MontiArcAutomaton, in: In 12th International Conference on Software Engineering Advances (ICSEA 2017), IARIA XPS Press. pp. 213–218.

[17] Butting, A., Kautz, O., Rumpe, B., Wortmann, A., 2019. Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. Journal of Systems and Software 149, 437–461.

[18] Butting, A., Rumpe, B., Schulze, C., Thomas, U., Wortmann, A., 2015. Modeling Reusable, Platform-Independent Robot Assembly Processes, in: International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015).

[19] Butting, A., Rumpe, B., Wortmann, A., 2016. Embedding Component Behavior DSLs into the MontiArcAutomaton ADL, in: Globalization of Modeling Languages Workshop (GEMOC'16).

[20] Campagne, F., 2014. The MPS language workbench: volume I. volume 1. Fabien Campagne.

[21] Cengarle, M.V., Grönniger, H., Rumpe, B., 2008. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05. TU Braunschweig. Germany.

[22] Chomsky, N., 1956. Three models for the description of language. IRE Transactions on information theory 2, 113–124.

[23] Ciccozzi, F., Spalazzese, R., 2016. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering, in: 10th Int. Symp. on Intelligent Distributed Computing.

[24] Combemale, B., Barais, O., Wortmann, A., 2017. Language engineering with the GEMOC studio, in: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE. pp. 189–191.

[25] Date, C.J., Darwen, H., 1996. A Guide to SQL Standard. Addison-Wesley Professional.

[26] van Deursen, A., Klint, P., Visser, J., 2000. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices 35, 26–36.

[27] Drave, I., Greifenberg, T., Hillemacher, S., Kriebel, S., Kusmenko, E., Markthaler, M., Orth, P., Salman, K.S., Richenhagen, J., Rumpe, B., Schulze, C., Wenckstern, M., Wortmann, A., 2019. SMArDT modeling for automotive software testing. Software: Practice and Experience 49, 301–328.

[28] Eclipse Foundation, . IoT Developer Survey 2019. [Online]. Available: https://outreach.eclipse.foundation/download-the-eclipse-iot-developer-survey-results. Last checked 04. June 2020.

[29] Efkemann, C., Peleska, J., 2011. Model-based testing for the second generation of integrated modular avionics, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE. pp. 55–62.

[30] Emmi, L., Gonzalez-de Soto, M., Pajares, G., Gonzalez-de Santos, P., 2014. New trends in robotics for agriculture: integration and assessment of a real fleet of robots. The Scientific World Journal 2014.

[31] Erb, B., Habiger, G., Hauck, F.J., 2016. On the Potential of Event Sourcing for Retroactive Actor-Based Programming, in: First Workshop on Programming Models and Languages for Distributed Computing, Association for Computing Machinery, New York, NY, USA.

[32] Erdweg, S., Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G.H., van der Woning, J., 2013. The State of the Art in Language Workbenches, in: Erwig, M., Paige, R.F., Van Wyk, E. (Eds.), Software Language Engineering. Springer International Publishing.

volume 8225 of *Lecture Notes in Computer Science*, pp. 197–217.

[33] Favre, J.M., Gasevic, D., Lämmel, R., Pek, E., 2010. Empirical Language Analysis in Software Linguistics, in: SLE, Springer. pp. 316–326.

[34] Feiler, P.H., Gluch, D.P., 2012. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley.

[35] Ferry, N., Nguyen, P., Song, H., Novac, P.E., Lavirotte, S., Tigli, J.Y., Solberg, A., 2019. GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems, in: IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), pp. 870–875.

[36] Ferry, N., Nguyen, P., Song, H., Rios, E., Iturbe, E., Martinez, S., Rego, A., 2020. Continuous Deployment of Trustworthy Smart IoT Systems. Journal of Object Technology 19, 16:1–23.

[37] Ferry, N., Nguyen, P.H., 2019. Towards Model-Based Continuous Deployment of Secure IoT Systems, in: ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 613–618.

[38] Fleurey, F., Morin, B., Solberg, A., Barais, O., 2011. MDE to Manage Communications with and between Resource-Constrained Systems, in: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems (MODELS), Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 349–363.

[39] France, R., Rumpe, B., 2007. Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering (FOSE '07) , 37–54.

[40] Franke, C., Gertz, M., 2008. Detection and Exploration of Outlier Regions in Sensor Data Streams, in: IEEE International Conference on Data Mining Workshops, pp. 375–384.

[41] Friedenthal, S., Moore, A., Steiner, R., 2014. A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann.

[42] García, C.G., Espada, J.P., Valdez, E.R.N., Díaz, V.G., 2014. Midgar: Domain-specific language to generate smart objects for an internet of things platform, in: 2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IEEE. pp. 352–357.

[43] Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., Rumpe, B., Varga, S., 2020. 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, CEUR Workshop Proceedings. pp. 22–30.

[44] Giang, N.K., Blackstock, M., Lea, R., Leung, V.C.M., 2015. Developing IoT applications in the Fog: A Distributed Dataflow approach, in: 5th International Conference on the Internet of Things (IOT), pp. 155–162.

[45] Gilchrist, A., 2016. Industry 4.0: the industrial internet of things. Springer.

[46] González, O., Casallas, R., Deridder, D., 2009. MCM-BPM: A domain-specific language for business processes analysis, in: International Conference on Business Information Systems, Springer. pp. 157–168.

[47] Greifenberg, T., Look, M., Roidl, S., Rumpe, B., 2015. Engineering Tagging Languages for DSLs, in: Conference on Model Driven Engineering Languages and Systems (MODELS'15), ACM/IEEE. pp. 34–43.

[48] Group, O.M., 2010. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03).

[49] Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A., 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components, in: Model-Driven Engineering and Software Development Conference (MODELSWARD'15), SciTePress. pp. 19–31.

[50] Haber, A., Ringert, J.O., Rumpe, B., 2012. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03. RWTH Aachen University.

[51] Harrand, N., Fleurey, F., Morin, B., Husa, K.E., 2016. ThingML: A Language and Code Generation Framework for Heterogeneous Targets, in: Proc. of the ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems, ACM, New York, NY, USA. pp. 125–135.

[52] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C., 2010. Integrating OCL and textual modelling languages, in: International Conference on Model Driven Engineering Languages and Systems, Springer. pp. 349–363.

[53] Heim, R., Mir Seyed Nazari, P., Rumpe, B., Wortmann, A., 2016. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors, in: Conference on Modelling Foundations and Applications (ECMFA), Springer. pp. 67–82.

[54] Hölldobler, K., Rumpe, B., 2017. MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag.

[55] Hölldobler, K., Rumpe, B., Wortmann, A., 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. Computer Languages, Systems & Structures 54, 386–405.

[56] Hölzl, F., Feilkas, M., 2007. 13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems, in: Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems, Springer. pp. 317–322.

[57] Hussein, M., Li, S., Radermacher, A., 2017. Model-driven Development of Adaptive IoT Systems, in: Proceedings of MODELS 2017. Workshop ModComp, CEUR, Austin, United States. pp. 17–23.

[58] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Science of computer programming 72, 31–39.

[59] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P., 2006. ATL: a QVT-like Transformation Language, in: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, ACM. pp. 719–720.

[60] Kalnins, A., Barzdins, J., Celms, E., 2004. Model transformation language MOLA, in: Model Driven Architecture. Springer, pp. 62–76.

[61] Karkouch, A., Mousannif, H., Moatassime, H.A., Noel, T., 2016. A model-driven architecture-based data quality management framework for the internet of Things, in: 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech), pp. 252–259.

[62] Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., 2020. MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety, in: AIAA Scitech 2020 Forum, American Institute of Aeronautics and Astronautics.

[63] Kautz, O., Rumpe, B., Wortmann, A., 2020. Automated semantics-preserving parallel decomposition of finite component and connector architectures. Automated Software Engineering 27, 119–151.

[64] Kirchhof, J.C., Michael, J., Rumpe, B., Varga, S., Wortmann, A., 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 (MODELS), ACM. pp. 90–101.

[65] Kolovos, D.S., Paige, R.F., Polack, F.A.C., 2008. The epsilon transformation language, in: International Conference on Theory and Practice of Model Transformations, Springer. pp. 46–60.

[66] Kolovos, D.S., Rose, L.M., Paige, R.F., Polack, F.A., 2009. Raising the level of abstraction in the development of gmf-based graphical model editors, in: 2009 ICSE Workshop on Modeling in Software Engineering, IEEE. pp. 13–19.

[67] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A., 2013. What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering 39, 869–891.

[68] Miorandi, D., Sicari, S., Pellegrini, F.D., Chlamtac, I., 2012. Internet of things: Vision, applications and research challenges. Ad Hoc Networks 10, 1497 – 1516.

[69] Moore, S.J., Nugent, C.D., Zhang, S., Cleland, I., 2020. IoT reliability: a review leading to 5 key research directions. CCF Transactions on Pervasive Computing and Interaction 2, 147–163.

[70] Morin, B., Harrand, N., Fleurey, F., 2017. Model-Based Software Engineering to Tame the IoT Jungle. IEEE Software 34, 30–36.

[71] Navarro Pérez, A., Rumpe, B., 2013. Modeling Cloud Architectures as Interactive Systems, in: Model-Driven Engineering for High Performance and Cloud Computing Workshop, pp. 15–24.

[72] Nguyen, P., Ferry, N., Erdogan, G., Song, H., Lavirotte, S., Tigli, J., Solberg, A., 2019. Advances in Deployment and Orchestration Approaches for IoT - A Systematic Review, in: IEEE International Congress on Internet of Things (ICIOT), pp. 53–60.

[73] Nordmann, A., Hochgeschwender, N., Wrede, S., 2014. A survey on domain-specific languages in robotics, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer. pp. 195–206.

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

[75] Persson, P., Angelsmark, O., 2017. Kappa: Serverless iot deployment, in: Proceedings of the 2nd International Workshop on Serverless Computing, Association for Computing Machinery, New York, NY, USA. pp. 16–21.

[76] Presley, A., Liles, D.H., 1995. The use of IDEF0 for the design and specification of methodologies, in: Proceedings of the 4th industrial engineering research conference.

[77] Rayes, A., Salam, S., 2018. Internet of Things-from Hype to Reality: The Road to Digitization. 2nd ed., Springer, Cham.

[78] Reggio, G., Leotta, M., Cerioli, M., Spalazzese, R., Alkhabbas, F., 2020. What are IoT systems for real? An experts' survey on software engineering aspects. Internet of Things 12, 100313.

[79] Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., Riegg, A., 2016. Internet of Things Patterns, in: Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP), ACM.

[80] Ringert, J.O., 2014. Analysis and Synthesis of Interactive Component and Connector Systems. Aachener Informatik-Berichte, Software Engineering, Band 19, Shaker Verlag.

[81] Ringert, J.O., Rumpe, B., 2011. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics .

[82] Ringert, J.O., Rumpe, B., Schulze, C., Wortmann, A., 2017. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems, in: International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17), IEEE. pp. 127–136.

[83] Rumpe, B., 1996. Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Verlag Wissenschaft, München, Deutschland.

[84] Rumpe, B., 2016. Modeling with UML: Language, Concepts, Methods. Springer International.

[85] Rumpe, B., Wortmann, A., 2018. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures, in: Lohstroh, Marten and Derler, Patricia Sirjani, Marjan (Ed.), Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Springer. LNCS 10760, pp. 383–406.

[86] Santoso, F.K., Vun, N.C., 2015. Securing IoT for smart home system, in: 2015 International Symposium on Consumer Electronics (ISCE), IEEE. pp. 1–2.

[87] Selic, B., 1996. Tutorial: Real-Time Object-Oriented Modeling (ROOM), in: Proceedings Real-Time Technology and Applications, pp. 214–217.

[88] Selić, B., Gérard, S., 2013. Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Elsevier Science & Technology.

[89] Selic, B., Gullekson, G., McGee, J., Engelberg, I., 1992. ROOM: An Object-Oriented Methodology for Developing Real-Time Systems, in: [1992] Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering, pp. 230–240.

[90] da Silva, A.C.F., Breitenbücher, U., Képes, K., Kopp, O., Leymann, F., 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, Association for Computing Machinery, New York, NY, USA. pp. 181–182.

[91] Song, H., Dautov, R., Ferry, N., Solberg, A., Fleurey, F., 2020. Model-based fleet deployment of edge computing applications, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Association for Computing Machinery, New York, NY, USA. pp. 132–142.

[92] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework. Pearson Education.

[93] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF: Eclipse Modeling Framework. 2. ed., Addison-Wesley, Boston, MA.

[94] Stübing, H., Bechler, M., Heussner, D., May, T., Radusch, I., Rechner, H., Vogel, P., 2010. simTD: a car-to-X system architecture for field operational tests [Topics in Automotive Networking]. IEEE Communications Magazine 48, 148–154.

[95] Taivalsaari, A., Mikkonen, T., 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. IEEE Software 34, 72–80.

[96] Taivalsaari, A., Mikkonen, T., 2017. Beyond the next 700 lot platforms, in: 2017 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC), pp. 3529–3534.

[97] Taivalsaari, A., Mikkonen, T., Systä, K., 2014. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, pp. 338–343.

[98] Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A., 2013. A New Skill Based Robot Programming Language Using UML/P Statecharts, in: Conference on Robotics and Automation (ICRA'13), IEEE. pp. 461–466.

[99] Vacchi, E., Cazzola, W., 2015. Neverlang: A framework for feature-oriented language development. Computer Languages, Systems & Structures 43, 1–40.

[100] Vacchi, E., Olivares, D.M., Shaqiri, A., Cazzola, W., 2014. Neverlang 2: a framework for modular language implementation, in: Proceedings of the companion publication of the 13th international conference on Modularity, pp. 29–32.

[101] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G., 2013. DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org. URL: http://www.dslbook.org.

[102] Wachsmuth, G.H., Konat, G.D., Visser, E., 2014. Language design with the spoofax language workbench. IEEE Software 31, 35–43.

[103] Wile, D.S., 2001. Supporting the DSL Spectrum. Computing and Information Technology 4, 263–287.

[104] Wortmann, A., Barais, O., Combemale, B., Wimmer, M., 2020. Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. Software and Systems Modeling 19, 67–94.