Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Concept and Implementation of a TOSCA Orchestration Engine for Edge and IoT Infrastructures

Leon Kiefer

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Dr. Uwe Breitenbücher
Commenced:	October 21, 2020
Completed:	April 21, 2021

Abstract

Reliable and automated management technologies are essential to support the fast growth of Internet of Things (IoT) applications and infrastructures. Manually deploying IoT applications on thousands of devices in a heterogeneous environment is complex, time-consuming, and error-prone. IoT devices are mostly embedded systems which are deployed as edge devices at specific physical locations where they provide their service by interacting with the physical environment and each other. For example, outdoor temperature sensors, traffic sensors on highways, or remote controlled lights. From a technical perspective, this cyber-physical nature of IoT applications is their most valuable but also their most challenging characteristic. To keep up with the proliferation of IoT technologies, as well as the fast growing needs of IoT applications, their development and deployment speed must increase accordingly. Techniques such as DevOps and continuous delivery, which are well-known in the context of cloud applications, are slowly adapted for IoT applications. One challenge of this process is to automate the deployment and management of IoT applications on edge infrastructures. The Topology and Orchestration Specification for Cloud Applications (TOSCA) enables the automated provisioning and management of various kinds of applications. However, its general-purpose modeling language makes it difficult to capture the cyber-physical nature of IoT applications. Existing TOSCA orchestration engines do not account for the low reliability, size, and heterogeneity of IoT infrastructures.

To tackle these issues, this work introduces the Reconciliation-based IoT Application Management (RITAM) approach to manage IoT application deployments on IoT and edge infrastructures. It combines domain-specific modeling of IoT infrastructures and general-purpose modeling using TOSCA. To apply the RITAM approach, this work formalizes the Controller and Reconciler Pattern which replaces imperative management workflows with eventually consistent reconciliation. Moreover, the practical feasibility of RITAM is validated using a prototypical implementation.

Contents

List of Figures

Listings

Acronyms

ACID Atomicity Consistency Isolation Durability. 30

API Application Programming Interface. 18

BASE Basically Available Soft State Eventually Consistent. 31

CLI Command-line interface. 18

CRUD create, read, update, and delete. 57

EDMM essential deployment metamodel. 24

HTTP Hypertext Transfer Protocol. 18

IaC Infrastructure as Code. 13

IoT Internet of Things. 13

JSON JavaScript Object Notation. 41

JVM Java virtual machine. 81

REST Representational state transfer. 18

RITAM Reconciliation-based IoT Application Management. 14, 53

SDK Software Development Kit. 18

TLS Transport Layer Security. 51

TOSCA Topology and Orchestration Specification for Cloud Applications. 14, 23

URI Uniform Resource Identifier. 63

URL Uniform Resource Locator. 62

VM virtual machine. 18

YAML YAML Ain't Markup Language. 9

1 Introduction

In recent years, the Internet of Things (IoT) market has grown exponentially, with about 8 billion devices in 2019 and a forecast of more than 41 billion IoT devices in 2027 [New20]. IoT devices are now of ubiquitous presence in our daily life in the form of smart devices, smart homes, and smart cities [FRS+13]. The Fourth Industrial Revolution, also called Industry 4.0, postulates interconnectivity of devices to optimize manufacturing processes and technologies [Jaz14]. It also enables the implementation of new manufacturing processes faster and more flexible. Emerging technologies enable developers and engineers to build smaller, more efficient, and more powerful connected devices for the Internet of Things. The trend to move services to the cloud and the high availability of high bandwidth internet connections support the growth of the IoT market.

IoT devices are sensors and actuators which interact with their physical environment and are connected to a network. The network allows to access and control the physical environment remotely by software systems. IoT devices are mostly embedded systems which are deployed as edge devices at physical locations where they provide their service by interacting with the physical environment. For example, outdoor temperature sensors, traffic sensors on highways, or remote controlled lights. From a technical perspective, this cyber-physical nature of IoT applications is their most valuable but also their most challenging characteristic [NHRR18]. Often thousands of devices are deployed for IoT applications.

To integrate and connect so many devices, the Message Queuing Telemetry Transport (MQTT) OASIS standard [OAS19] can be used. MQTT is a lightweight publish/subscribe messaging protocol for the Internet of Things. IoT applications often use the cloud to host central services which integrate and control edge devices. For example, MQTT brokers and management services of IoT applications are hosted in the cloud to provide high availability, scalability, and durability.

The management and deployment of IoT applications is complex, because of different technologies, hardware limitations and the huge amount of devices. It is not uncommon that a software update has to be rolled out to thousands or millions of devices. As a result, manually deploying IoT applications is time-consuming, error-prone, and costly [BKLW17]. The automation of orchestration and management processes for IoT application is therefore crucial. The infrastructure of IoT applications is composed of many different edge devices, technologies, cloud services, and networks, resulting in a heterogeneous infrastructure [HQ21; SBH+17].

To operate applications and the infrastructure they are running on, concepts such as DevOps and Infrastructure as Code (IaC) emerged [BSK20]. According to Bass [Bas15], DevOps is “a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality”. Its concepts can also be used for IoT applications, to automate deployment processes and deploy more frequently [AS18; HF10; Mal19]. With IaC, software development techniques are used to management the infrastructure,

e. g., using version control systems and continuous delivery pipelines [Mor21]. These techniques also allow to update applications and infrastructure, deploy new software versions, and quickly adapt to changing requirements.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) standard enables automated provisioning and management of various kinds of applications. It enables modeling of heterogeneous components of an application as TOSCA Service Templates. An application modeled as TOSCA Service Template can be deployed using an orchestration engine, which is either provided by a cloud provider or as a standalone runtime. These orchestration engines enable the deployment and management of applications across multiple technologies, machines, and networks. As a result, TOSCA can be used to orchestrate IoT applications.

However, deploying on heterogeneous infrastructure requires additional configuration and custom orchestration logic, such as addressing devices at physical locations and executing commands on remote devices. Either special plugins for the orchestration engine are required or the TOSCA Service Templates must be adapted to handle the heterogeneous infrastructure. Furthermore, TOSCA does not define an instance model which describes the current state of the infrastructure and allows to make changes to it[1]. As a result, TOSCA is limited to the initial provisioning of applications. To modify applications an operator must perform additional manual steps, which is time-consuming, error-prone, and costly. TOSCA also ignores the domain characteristics of IoT applications, such as the unique physical environment of devices and the low reliability of devices and their network connections.

To tackle these issues, this work introduces the Reconciliation-based IoT Application Management (RITAM) approach. RITAM splits the management into a central manager component running in the cloud and a lightweight TOSCA orchestration engine running on the edge and IoT devices. The Controller and Reconciler Pattern is used, which enables the resilient, reliable and self-healing behavior of RITAM. Although, the use case is IoT and edge infrastructures, this works aims for independent domain beyond the scope of IoT and edge infrastructures. Therefore, the concepts and TOSCA orchestration engine introduced in this work are also very well suited for other domains.

1.1 Structure of this work

The remainder of this work is structured as follows. First the the basics of application management and related work is discussed in Chapter 2. The used concepts and patterns are introduced and formalized in Chapter 3. Chapter 4 presents the RITAM approach. The RITAM Device and Application Manager is explained in Chapter 5. The RITAM Orchestrator is described in Chapter 6. RITAM is evaluated based on a prototypical implementation and different management scenarios in Chapter 7. Lastly, in Chapter 8 the results of this work are concluded.

[1]There is a specification draft for an instance model for TOSCA [OAS17].

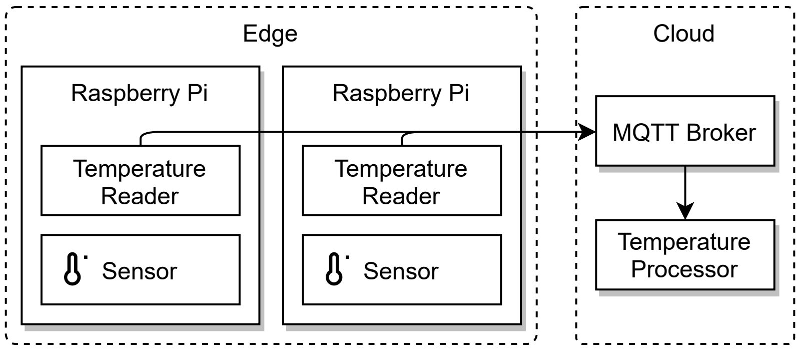

Figure 1.1: Temperature IoT application as motivating scenario

1.2 Contributions

This work can be divided into three main contributions:

1. Formalization of the Controller and Reconciler Pattern[2] in the context of general application management. The definitions and components are generalized for use in domain independent application management.
2. Concept and implementation of a scalable, highly available, and resilient multi-device application orchestrator using the Controller and Reconciler Pattern, to manage IoT devices and applications hosted on them using an IaC approach.
3. Concept and implementation of a TOSCA extension and orchestration agent using the Controller and Reconciler Pattern. The TOSCA extension enables individual nodes to implement custom reconcilers and allows resilient and fault tolerant application orchestration and management using the TOSCA standard.

1.3 Motivating Scenario

To explain the RITAM approach, a motivating scenario is introduced. Imagine an IoT application, which uses IoT edge devices to measure and collect the temperature from different locations and processes them in the cloud. Figure 1.1 shows all the relevant components of the example IoT application. The edge infrastructure of the IoT application consists of *Raspberry Pis*[3], which have the required temperature sensor hardware and are connected the internet. In the cloud environment a *MQTT Broker* is deployed, which acts as a gateway for all edge devices. On the *Raspberry Pis*, the *Temperature Reader* software reads the temperature values from the hardware sensor and sends them to the *MQTT Broker*. From the *MQTT Broker* the temperature values are sent to the *Temperature Processor*.

[2]The Controller and Reconciler Pattern is known from Kubernetes [BGO+16].

[3]https://www.raspberrypi.org/products/

This work focuses on the modeling, deployment, and management of the IoT application part which is deployed on the edge infrastructure, see the left part of Figure 1.1. The deployment of cloud services is out of scope of this work and already covered in the literature. In this work, the resilient, robust, and self-healing RITAM approach is presented to managed IoT applications in unreliable and dynamic environments. It covers the provisioning, updating, decommissioning, and monitoring of IoT applications and edge infrastructures.

2 Background and Related Work

In this chapter the required background for this work is presented. Also, relevant scientific literature of this work is provided. Additionally, some background about TOSCA is given.

First, the basics of application management are presented in Section 2.1. Then, an overview of TOSCA and related work is provided in Section 2.2. Related work of edge and IoT application management is discussed in Section 2.3. Section 2.4 introduces and defines the term level-based API.

2.1 Basics of Application Management

In this section application management models, approaches, and terms are presented. Application management refers to the operation of an application, which includes the whole lifecycle of an application, except its development [Bre16]. Applications are typically managed by human operators. These operators deploy, update, and decommission applications. Manual application management is complex, time-consuming, error-prone, and costly [Bre16; DJ07; FLR+14]. Therefore many tools, approaches, models, and technologies were developed to support human operators and automate application management.

2.1.1 Application Management Systems

Application management systems or short management systems are software systems that support or enable an operator to manage applications or parts of applications. The following list shows some categories of management systems:

- Configuration management tools (Ansible[1], Puppet[2], Chef[3])
- Infrastructure tools (AWS CloudFormation[4], Terraform[5])
- Container orchestration tools (Docker[6], Kubernetes[7])
- Operating systems (File systems, Scripts, Systemd[8])

[1]https://www.ansible.com/
[2]https://puppet.com/
[3]https://www.chef.io/chef/
[4]https://aws.amazon.com/cloudformation/
[5]https://www.terraform.io/
[6]https://www.docker.com/
[7]https://kubernetes.io/
[8]https://www.freedesktop.org/wiki/Software/systemd/

- On-premise platforms (OpenStack[9], OpenShift[10], vSphere[11])
- Cloud-based services (AWS[12], Google Cloud[13], Azure[14])

Management systems manage a wide range of things, from applications, to components, to infrastructures. Because many aspects depend on the concrete management system it is hard to talk about management systems in general. Therefore, in this work the term *component* will be used to denote the “things” managed by a management system. Examples of components: Software components, services, virtual machines (VMs), networks, operating system processes, containers, files, databases, applications, backups.

Management systems provide an Application Programming Interface (API) for operators and other systems to manipulate components and to call management operations. These interfaces are called *management APIs*. Many different technologies are used by management system to provide management APIs. Commonly used technologies are for example, Representational state transfer (REST) based Hypertext Transfer Protocol (HTTP) APIs, Software Development Kits (SDKs), Command-line interface (CLI) tools, and web-based interfaces [Bre16].

2.1.2 Declarative Application Management

According to Breitenbücher [Bre16], declarative application management describes the management of applications and components based on declarative descriptions. These descriptions define management goals which should be realized by a management system. The declarative descriptions only describe *what* should be achieved but do not describe management operations or *how* these goals can be achieved [Bre16; EBF+17].

For declarative application management, *declarative deployment models* are used to describe the application, components, and structure [EBF+17]. The models are either created by an operator or automatically generated before they are deployed in a management system. Declarative deployment models only describe what should be deployed, e. g., the components, their composition, structure, and configuration. They do not describe how a management system can realize the described application. Different deployment metamodels and languages exist to express them.

Deployment models do not have to be declarative, *imperative deployment models* are often used for customized deployments or deployments of legacy components [EBF+17]. Imperative deployment models describe how to deploy an application by providing all operations and their order in which they should be executed. Imperative and declarative deployment models can also be combined [BBK+14] to form hybrid deployment models. Hybrid deployment models express the high-level structure declaratively and allow to customize the deployment process by providing imperative operations at specific extension points. Deployment models only describe the initial deployment process of an application [BBK+14; EBF+17]. For other management operations after the initial deployment, such as adding, updating, and removing components of a running application, other

[9]https://www.openstack.org/
[10]https://www.openshift.com/
[11]https://www.vmware.com/products/vsphere.html
[12]https://aws.amazon.com/
[13]https://cloud.google.com/
[14]https://azure.microsoft.com/

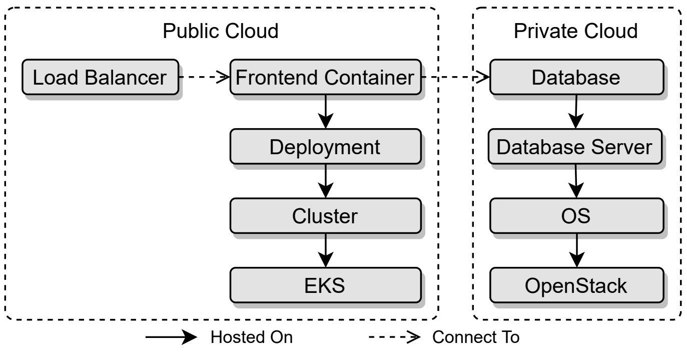

Figure 2.1: Example application topology model

models have to be used. For example, imperative management models, desired state models, or declarative management models can be used to models these management operations [Bre16]. However, declarative deployment models can be used to derive other models to perform management operations, such as desired state models or instance models.

To describe the composition and structure of an application, application topologies are used. They model the application structure as a graph structure which follows the application architecture or deployment view, see Figure 2.1. The graph consists of nodes and relationships between nodes. The nodes represent components which are connected via relationships. Application topologies can be used in declarative models. The graph-based structure allows management systems to derive plans on how to deploy and manage the application [BBK+14].

2.1.3 Desired State Model

Desired state models or goal state models are prescriptive models which declaratively describe the desired structure and individual state of components [Bre16]. The term “state” comprises the environment, configuration, and runtime information of a component. Desired state models often have an one to one relationship to the component instance they model. In contrast to deployment models, they describe the goal state independently of the current state. As a result, desired state models enable idempotent management and are a form of IaC [Mor21].

2.1.4 Instance Model

Instance models are used to model the configuration and runtime information of running components. They can be used descriptive, to describe the current state of a running application, or prescriptive, to define the desired state or desired configuration of an application [Bre16]. The manual creation of such models by an operator is a time-consuming task [Bre16], which can be automated. For example, as the result of a deployment operation or loaded from an instance model database. However, these instance models can get outdated quickly. Over time the state of the component may have been changed without also updating the stored instance models, leading to inconsistencies, called *configuration drift* [Mor16]. These inconsistencies can lead to management operation failures or broken application configurations.

To tackle these issues, the RITAM approach uses IaC combined with a continues synchronization process to ensure the instance models are consistent with the state of the components. The synchronization process keeps the desired state in sync with the deployed component and automatically updates instance information in the model.

2.1.5 Actual State

The previous sections describe different management models of applications and components. Models can be descriptive or prescriptive, describe deployment information or runtime information, and model the desired state or the actual state. This section gives a definition of actual state and how it is different from desired state.

To manage components, it is important to know the state of each component. Two states have to be distinguished, the *actual state* and the *desired state* [GN17]. The desired state exists in the form of a desired state model (Section 2.1.3), defined by an operator. The actual state is the state which can be observed in reality from the environment and the component itself. Therefore, the desired state only exists in the context of application management, while the actual state is an intrinsic property of the component itself. For example, the desired number of instances of an elastic application is part of its desired state, it must be stored in the management system. The actual number of instances is part of its actual state, it is an intrinsic property of the running application and can be measured by counting the instances.

The actual state has to be distinguished from instance models, which may only be a model of the actual state. An instance model may also be generated without actually observing the state of the component. For example, by creating an instance model from the expected result of a management operation. When the state of a component is measured, e. g., by a management system, the resulting information is only a snapshot of the actual state. The snapshot can be used by the management system to derive an instance model or to compare it to the desired state model.

In general, the actual state of a component can not be set to a specific value directly, often it takes time to change the actual state to a specific value. For example, it takes time to start instances of an application and reach a specific number of running instances. When the actual state of the component changed, either by a management operation or any other means, the snapshot and the information about the actual state becomes stale. Because state changes can happen at any time, keeping the actual state information (instance model), which is processed by a management system, in sync with the actual state of the component is difficult. In general, management systems can not know about state changes without creating a new snapshot. Even if there are some kind of change listeners for the actual state of the component, there is a delay between the state change and processing the change notification. It follows, that a management system can not immediately know about state changes and therefore it can not know if the last observed snapshot is already stale or not.

This limitation makes it difficult to build a consistent management system. However, different approaches were developed to mitigate this limitation. In this work, two approaches are discussed: workflow-based management systems [Bre16] and management systems which accept inconsistencies.

Workflow-based management systems make two assumptions: (i) the initial state of a component is known and (ii) all state changes are either controlled or immediately known by the management system [OAS17]. This allows workflow-based management systems to define the state of a component as a model which is modified based on pre- and postconditions of management operations. These models allow to plan management operations and generate workflows, see Section 2.1.7. However, the assumptions only hold for transactional systems with strict consistency. Using a workflow-based management system to manage a component which does not support transactional management can leave the component in a broken state from which the workflow-based management system may not be able to recover from, because the used compensations might fail as well.

The second approach to manage applications and components is to minimize the discrepancy of the observed actual state information and the actual state of the component. This can be done by always measuring the actual state before performing a management operation. After the management operation, the state is measured again to confirm the operation was successful. Inconsistencies are minimized by repeatedly measuring the actual state and retrying failed operations. Compared to the strict consistency requirement of workflow-based management systems, this approach does make less assumptions about the components state consistency, but comes with the limitation that under certain circumstances the components end up in a broken state. However, the implementation of this approach is simpler because no transactional support is required by any involved management system.

This work uses the Controller and Reconciler Pattern, which is based on the second approach. The Controller and Reconciler Pattern accepts the limitation that the observed actual state information may be stale but still guarantees eventual consistency.

2.1.6 Declarative Management Model

Breitenbücher [Bre16] defines a *declarative management model* ("deklaratives Managementmodell") which (i) declaratively describes abstract management tasks, (ii) the current state of the application, and (iii) the desired state of the application. The declarative management model only describes *what* management task should be executed, but does not provide *how* it can be executed technically. The abstract description of management tasks in combination with the instance model allows to generate imperative management models and reduces the effort of manually creating these models. Declarative management models are not limited to state-changing management tasks, because the abstract management tasks are able to model state-preserving management tasks which are incorporated into the generated imperative management models.

As a concrete realization of a modeling language for declarative management models, Breitenbücher [Bre16] introduces *DMMN* (Declarative Application Management Modelling and Notation). The DMMN-metamodel can represent the application topology, annotate topology elements with management annotations, and specify technical management tasks on different layers. DMMN models can be visualized and modeled using a graphical notation which is based on *Vino4TOSCA* [BBK+12]. Further, DMMN is used in the *PALMA* framework [Bre16] which implements an end-to-end application management process based on declarative management models. As a result, the application management process can be declaratively modeled and executed by application manager.

However, the PALMA framework does not prevent configuration drift and does not provide tooling for IaC, which is an essential part of DevOps [Mor21]. This work focuses on IaC in the context of DevOps as its main use case scenario.

2.1.7 Workflow-based Application Management

Workflows are imperative models which describe the control- and data-flow of management operations. They can be used to automate management tasks, which require complex application specific logic. Manually creating workflows is a complex, time-consuming, costly, and error-prone task [BBK+14].

Workflows themselves are imperative, but they can be generated from declarative models, such as deployment models [BBK+14; Bre16]. When generating workflows, a management system interprets a declarative model and adds required orchestration steps to create a workflow. The steps of a workflow have pre- and postconditions and are composed together, that the precondition of the workflow matches the current state of the application and the post condition matches the desired state. A workflow can be executed by a workflow management system [Ley00] and after the execution finished successfully, the application should be in the desired state according to the postcondition of the workflow.

Workflows are not limited to state-changing management tasks, they can also model state-preserving management tasks [Bre16]. In general, not all state-preserving management tasks can be modeled using declarative deployment models. However, there are some approaches which enable modeling of state-preserving management tasks using other declarative models. For example, the PALMA-Method [Bre16], which is presented in the previous section, or the *management feature enrichment and workflow generation* approach of Harzenetter et al. [HBL+19], which enriches deployment models with additional management operations and then generates a workflow based on provided plugins.

In this work, another approach is used, which extends the scope of the “state” definition, so that state-preserving management tasks become state-changing management tasks in the extended scope. For example, a backup management task preserves the state of the system it is exported from (source). However, it changes the state of the system where the backup is stored (target). By extending the scope of the backup management task to also describe the state of the target system, it becomes a state-changing management tasks. The separation of state-changing and state-preserving management tasks is only a matter of defining the scope of the managed system. When using IaC, all parts of a system should be specified in the definition files. Therefore, all management tasks based on a holistic IaC definition can be considered state-changing management tasks.

Workflows provide strict consistency. If any operation during the execution of the workflow fails, all changes done by the workflow so far will be rolled back. If an operation was already committed, compensation operations are executed and all outstanding operations are rolled back. This means, already a temporary failure in one operation of the workflow leads to a rollback of the complete workflow. Temporary failure can happen at any time, e. g., network failures, timeouts, or software failures in distributed systems [WBB+20b]. Therefore, workflow-based management systems are not suited for unreliable environments, such as IoT and edge infrastructure where network partitioning is not uncommon.

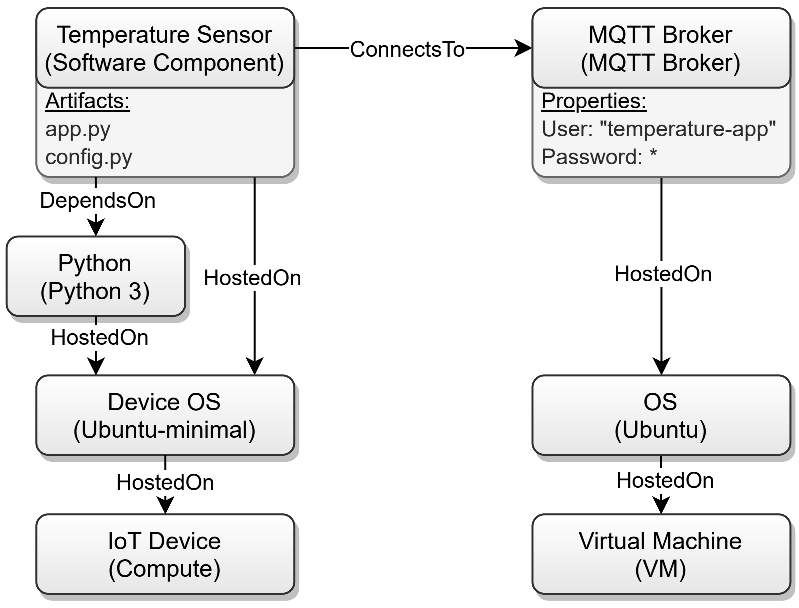

Figure 2.2: Example TOSCA Topology

This effect is amplified by the fact that IoT applications often involve thousands of devices. Let's assume an IoT device has a network availability of 99.9%, if we need to provision 1000 devices the probability that all devices are available at the same time is only $0.999^{1000} = 36.8\%$. This means the provisioning workflow would fail with a probability of 63.2%. Workflows could be adapted to handle temporary failures, but this requires extra effort, introduces complexity, and creates new sources of failures into the workflow.

2.2 TOSCA Fundamentals and Related Work

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an OASIS standard [OAS20] which specifies how cloud applications can be modeled and deployed [BBKL13; LSC20]. Applications are modeled as TOSCA topologies which contain TOSCA Nodes and TOSCA Relationships, as shown in Figure 2.2. TOSCA Nodes describe components of an application. Each node defines operations, e. g., lifecycle operations to start and stop the component described by the TOSCA Node. The implementations of the operations are defined by TOSCA Node Types, which are shown in Figure 2.2 in the brackets. TOSCA Relationships express a relationship between two TOSCA Nodes, they can also define operations, e. g., to configure a connection between two components. A TOSCA Topology Template combines all this information and forms an application template which can be deployed by a TOSCA orchestrator.

The TOSCA standard does only specify the deployment model and orchestration procedure, but does not define an instance model for deployed models. A specification draft was published by OASIS which should define an instance model for TOSCA [OAS17]. However, the draft is in a very early stage and is currently not usable in practice. Therefore, in this work an instance model for TOSCA applications is introduced which models nodes and their attributes and allow operators to use them.

2.2.1 Essential Deployment Metamodel

Similar to TOSCA there is the essential deployment metamodel (EDMM) [WBF+19], which uses a simplified model for applications and components. EDMM covers all essential model components to enable the modeling and deployment of applications. As a result, deployment models are convertible between multiple heterogeneous deployment automation formats using EDMM as an intermediary format, which is also shown by Wurster et al. [WBB+20a]. However, for mapping a deployment model to EDMM, it has to comply with the set of requirements imposed by EDMM [WBH+20]. In general, this limits the convertibility to only a small subset of deployment models which only use very basic features of the modeling language and platform.

To mitigate the issue of inconvertible deployment models, Wurster et al. [WBH+20] introduce TOSCA Light, an EDMM-compliant subset of TOSCA. TOSCA Light defines rules which can be used by operators to model their applications with TOSCA but still be able to automatically convert them to other formats. In combination with the EDMM Modeling and Transformation System [WBB+20a], TOSCA Light can be used to enable DevOps while using the vendor agnostic TOSCA modeling language. For example, the TOSCA Light model of an application can be stored as a YAML file in a git repository and an automated workflow converts these to Kubernetes definitions and deploys them, when a commit is pushed to the repository.

In this work TOSCA is used as modeling language for IoT applications, which is not TOSCA Light compliant, because it uses a custom lifecycle interface. However, RITAM enables the use of DevOps and IaC without transforming TOSCA deployment models.

2.2.2 TOSCA Orchestrator Implementations

TOSCA is an open standard and many different orchestration engine implementations exist [LSC20]. Some implementations define vendor specific extensions to the TOSCA standard to support platform specific features. Other implementations focus on a smaller subset of the TOSCA standard to reduce the complexity for operators.

Cloudify[15] is an open source platform based on TOSCA. The Cloudify DSL is derived from TOSCA YAML profile and is used to define application blueprints. Cloudify provides a support for multiple cloud providers, orchestration plugins and a CLI to automate the deployments.

OpenTOSCA is a research prototype developed by University of Stuttgart [BEK+16]. It is part of an ecosystem of tools that enable management of many different use cases and applications. As part of active research it provides many functional extensions to the TOSCA standard.

xOpera[16] is a TOSCA orchestrator which supports TOSCA YAML v1.3. It uses Ansible playbooks to model TOSCA Nodes and their operations. This allows operators to describe their application using the declarative domain-specific language of Ansible[17].

[15]https://cloudify.co/

[16]https://github.com/xlab-si/xopera-opera

[17]https://www.ansible.com/

2.2.3 IoT Application Management with TOSCA

Hirmer et al. [HBS+16] present an approach for automated provisioning and configuration of devices. The approach distinguishes domain experts, which model *smart environments*, and hardware experts, which manage the technical device integration. The domain experts model the IoT environment using a so called *digital twin blueprint* which defines specific information about the devices and their environment. When the blueprint is applied, a binding between the models in the blueprint and the physical devices is performed and device adapters, which are manged by hardware experts, are deployed in the physical environment. The digital twin is the digital representation of the physical environment and is used for monitoring of the environment. Therefore, the digital twin must constantly synchronize with the physical environment to reflect all changes. However, the synchronization process is not explained in detail. Also, the digital twin and monitoring is not implemented and evaluated in their prototype. This work introduces a well-defined synchronization process and instance model, which is evaluated based on a prototypical implementation in different management scenarios.

TOSCA is also used to automate the deployment of IoT applications. Da Silva et al. [SBH+17] use TOSCA to model the complete IoT application, including physical hardware components, IoT middleware, and IoT software components. In [SBK+16] they also introduce new TOSCA Node Types to model IoT devices and middleware, such as Raspberry Pis and Mosquitto MQTT brokers[18]. The resulting TOSCA model allows to setup the entire IoT environment out-of-the-box. However, their approach does not cover the automated deployment of applications onto thousands of devices, which is not uncommon in IoT environments. As a result, modeling entire IoT environments in one TOSCA Topology Template results in thousands of nodes in the topology which has to be managed by operators for each application individually. This work tackles these issues, by allowing to define the devices independently of the applications, so the topology is small and does not change when devices are added or removed from the infrastructure.

The management and automation of many edge devices is tackled by the work of Schmid [Sch18]. He introduces a *Device Manager* component which is modeled as part of the IoT application topology and manages all devices and related software components. The Device Manager component implements management operations for devices, which allows all devices to be modeled as homogenous TOSCA Node Types which only define different properties. To add a new device or update a software component the approach uses imperative management operations, e. g., `updateDevice` and `uploadBinary`. Therefore, the approach allows to reuse artifacts for many devices and operations, but still requires to model all devices and all software components in the same topology. To manage the devices, imperative management plans are used, which add and remove devices from the topology. As a result, updates to the topology can not be managed using IaC, because the declarative deployment model is only used for the initial deployment. This work tackles these issues, by forbidding imperative workflows for management operations and enforcing IaC as the single source of truth. As a result, developers and operators can use DevOps and GitOps [BKH21] to manage devices and applications using IaC.

[18]https://www.mosquitto.org/

2.3 Edge and IoT Application Management

Nastic et al. [NTD15] collect design principles for software-defined IoT cloud systems. Their findings of design principles include: everything as code, scalable development, API encapsulation, declarative provisioning, central point of operations, and automation. Based on these principles they introduce SDG-Pro (Software-Defined Gateways Programing framework) a programming framework for software-defined IoT cloud systems. SDG-Pro is comprised of multiple microservices, which include the *APIManager* and *IoT units management layer*. The APIManager exposes *data and control channels* of devices and the *IoT units management layer* provides agents to support the provisioning of software-defined gateways which are an abstraction of the components deployed in the IoT cloud system. SDG-Pro also supports programming for business logic services which can directly interact with the data and control channels provided by the framework. As a result, SDG-Pro supports developers and operators during the whole lifecycle of an IoT application.

The RITAM approach, introduced in this work, also follows the main design principles of software-defined IoT cloud systems. However, this work focus on the management of the IoT applications and devices with a high degree of flexibility and reliability.

To manage edge and IoT applications and their infrastructure, configuration management systems can be used. Configuration management systems allow to specify the configuration of devices using a configuration language. Configurations can be parameterized and deployed to many devices and reduce the manual management effort. For example, Ansible is used in practice to manage thousands of low power devices using a pull-based eventual consistent approach [Ann19]. Hassan and Qasha [HQ21] use Ansible to model and deploy IoT systems on heterogeneous IoT environments including public cloud. For small environments, such as smart home scenarios, Perumal et al. [PDB15] present a framework which accounts for the heterogeneity of IoT devices by introducing a *proxy layer*. The framework also provides the *service enablement layer* which provides the configuration management and device discovery for applications.

IoT cloud applications are comprised of cloud services and IoT devices, such as sensors, actuators, and gateways [NTD15]. They utilize different infrastructure resources to benefit from scalability, availability, and durability of cloud services and the locality and physical environment of IoT devices, enabling pervasive computing [FHH+10]. Heterogeneous infrastructure, which includes different cloud services and thousands of physically distributed IoT devices, imposes a management challenge for such applications. To manage IoT applications, general-purpose modeling languages, such as TOSCA, can be used, because they are technology agnostic, flexible, and extensible. However, general-purpose modeling languages lack specialized features, abstractions, and non-functional properties for the IoT and edge infrastructure domain. For example, the physical environment of an IoT device is an important property of the IoT infrastructure, however general-purpose models often try to abstract from these heterogeneous physical properties to allow homogenous modeling. As a result, modeling IoT applications, which make use of the heterogeneity of the infrastructure, using general-purpose languages is difficult and involves additional effort to overcome the impractical abstractions of these modeling languages. In contrast, a domain-specific modeling language incorporates appropriate constructs and abstractions, and encodes common domain knowledge into the modeling language [MHS05]. In this work a domain-specific language is used to model IoT devices and application templates and a general-purpose language is used to describe the topologies and components of the applications.

2.4 Background of Level-based and Edge-based Control Loops

The terms *level-based* and *edge-based* originate from hardware interrupts, which can be triggered by either of both [Spa19; Yad13]. The terms are used to categorize the program logic of a system. Level-based logic only depends on the current state. While edge-based logic also depends on the transitions of the state, e. g., state changes.

Level-based and edge-based systems have different properties. A level-based system is more resilient because if it crashes, it can recover by just reading the current state [Bow20; Spa19]. When an edge-based system crashes, it can not recover from just reading the current state. It depends on other systems to reliably store a history of all state transitions. If an edge-based system misses some of the state changes it will produce wrong results [Bow20]. On the other hand, advantages of edge-based triggered systems are that not the whole current state must be reevaluated and actions are only performed if something changed.

The term control loop refers to the logic which is triggered by a specific state level or change using a hardware interrupt. Here it refers to the overall concept of having a level-based loop which evaluates the current state and makes changes to the state so it matches a goal state. Multiple control loops can be integrated using a *level-based API*. Such an API implements the concept of only allowing idempotent operations (get or set) the level (state) of objects. The edges or state changes of the objects are opaque to the clients of the level-based API.

Informally a level-based API can be thought of as a database which only allows idempotent operations. The REST architectural style can be used to implement a level-based API. Either by forbidding POST requests or by implementing idempotent POST requests which create resource with deterministic identifiers.

A level-based API is data-driven. This means the API is built around the data and not built around functionality of a system. The focus of the API is to manipulate, manage, and maintain the state. The functionality of the system is implicitly provided through the state stored in the level-based API.

A level-based APIs enables the implementation of IaC and DevOps processes [Mor16]. One of the base practices of IaC, according to Morris [Mor16], is to use definition files. All configuration is defined in "executable" configuration definition files, executable here means that the configuration definition files can be interpreted by some management system. These configuration definition files can be applied to a level-based API and interpreted by a controller, as described in Section 3.3.

A commonly used system with a level-based API is Kubernetes [Kub20]. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications[19]. Kubernetes provides a level-based REST API to define the desired state of all application components, which are then managed by a control loop based on the Controller and Reconciler Pattern.

[19]https://kubernetes.io/

The goal seeking behavior of the control loop is very stable. This has been proven in Kubernetes where we have had bugs that have gone unnoticed because the control loop is fundamentally stable and will correct itself over time.

If you are edge triggered you run risk of compromising your state and never being able to re-create the state. If you are level triggered the pattern is very forgiving, and allows room for components not behaving as they should to be rectified. This is what makes Kubernetes work so well.

- Joe Beda, in [GN17]

3 Fundamentals

In the following sections the definitions, concepts, and patterns used in the RITAM approach are described. First, important definitions of data consistency of distributed systems is provided in Section 3.1. In Section 3.2, different approaches to correlate components and models are discussed. An overview of the Controller and Reconciler Pattern is provided in Section 3.3. The pattern is then formalized in Section 3.4. In Section 3.5, the data model of the pattern is presented. Section 3.6 describes the storage, which is the level-based API that stores the data models. The reconciler and controller components and their behavior are presented in Section 3.7 and Section 3.8. Section 3.9 introduces the Operator Pattern, which is used to automate tasks of human operators.

3.1 Data Consistency of Distributed Systems

Management systems have to interact and integrate with many different systems and technologies to manage all kinds of infrastructures and applications. The goal of these interactions is to change the state of the infrastructure and application, e. g., starting a process or open a port in a firewall. Each management system and technology may store its state in some way and expose it via a Management API.

When building and integrating management systems, one must be aware of the data consistency guarantees and requirements of these systems. For example, some systems use workflows and rollbacks (AWS CloudFormation, OpenTOSCA), some provide idempotent operations (Puppet, CFEngine[1]), and others provide eventual consistency (Kubernetes). In the following, important consistency models are explained in detail and an overview of their guarantees and constraints will be provided.

3.1.1 Data Consistency Definitions

The following data consistency definitions are based on the work of Vogels [Vog09]. Only the client point of view is of interest for the integration of management systems, because a management system interacts with other systems as a client.

Terminology used in the following definitions:

- Storage System: A system a client can read from or write data to.
- Client A: A client that writes to or reads from a storage system.

[1]https://cfengine.com/

- Client B and C: Independent clients from client A. They also write to and read from the storage system.
- We assume client A made an update to a data object.

Definition 3.1.1 (Strict consistency)
After the update completes, any subsequent access (by A, B, or C) will return the updated value.

Definition 3.1.2 (Weak consistency)
The storage system does not guarantee that subsequent accesses will return the update made by client A. There is a time period between the update and the moment the storage systems guarantees that all clients will always see the updated value. The period is called inconsistency window. During that period clients can observe different values for the same data object.

Definition 3.1.3 (Eventual Consistency)
Eventual Consistency is a specific form of weak consistency. The storage system guarantees that if no new updates are made to the data object, eventually all clients will see the update.

Definition 3.1.4 (Causal consistency)
If client A has communicated to client B that it has updated a data item, a subsequent access by client B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by client C that has no causal relationship to client A is subject to the normal eventual consistency rules.

Definition 3.1.5 (Read-your-writes consistency)
After client A has updated a data item, it will always accesses the updated value and will never see an older value. Client B and C may not see the updated value immediately. This is a special case of the causal consistency model.

Definition 3.1.6 (Read-your-creations consistency)
When client A creates a new data object, any subsequent access by client A will see the data object, unless it is deleted. Client B and C may not see the created data object immediately. This is a special case of the causal consistency model.

Definition 3.1.7 (Monotonic Read consistency)
If a client has seen a particular value for the object, any subsequent accesses will never return any previous values.

3.1.2 Inconsistent View

When multiple systems are involved in the management of an application or component, the actual state is also distributed across all these systems. The CAP theorem presented by Brewer [Bre00] states that a distributed system can only achieve two of the three properties (data consistency, system availability, and tolerance to network partitioning) at any given time. This means if the network is partitioned a system can not be available and provide consistent data at the same time. Therefore, systems which provide the Atomicity Consistency Isolation Durability (ACID) properties can not be available in case of a network partitioning [Pri08]. Instead of ACID properties eventual consistent systems provide the Basically Available Soft State Eventually Consistent (BASE) properties [Pri08]. It turns the choice between availability and consistency into a trade-off [Bre12]. Availability is preferred over a precisely-known state at all times, and data consistency is only assured over time.

When related data is stored in different systems which are connected via a network, they form a distributed system. If the distributed system provides the BASE properties, a client that reads from the distributed system may see inconsistent data. Because the data is distributed, the client has to read different parts from different systems. As a consequence of the BASE properties, updates to the data may not be visible in all systems at the same time and therefore a combination of old and new data parts is retrieved be the client. This behavior is called inconsistent view. Clients of such distributed systems must handle them appropriately.

Management systems can be viewed as clients of distributed systems, because they need to interact with many different systems to manage applications and their components. Most management systems support the BASE properties, which make it easy to integrate them. The Controller and Reconciler Pattern presented in this work makes use of the BASE properties to enable scalability, resilience, and robustness.

3.2 Component Identity Correlation

Application management systems work with instance models, which are abstract representations of real components. The relation between the instance model and a component is known as “model of” relationship.

This relation imposes some technical challenges, when the instance model and the component are managed by two different software systems. In the following these two systems are differentiated and called *management system* and *external system*. The management system implements high-level management operations and keeps track of “managed” components by storing component instance information in form of instance models. The external system provides a management interface to mange the real components. Note that the external system can also be a management system, for unambiguity they are referred to as external systems in this section. The management system interacts with the management interface of the external system to perform management operations on components.

The management system uses the stored instance information to invoke the management interface with the correct arguments. The stored instance information must contain information about the identity of the real component. The management system must know the identity of the real component for all its instance models. If the management system does not know the identity of the real component it can not perform any management operation on it. In general, the correlation must only be made from the instance model to the component but not the other way round.

For example, a management system contains an instance model of a VM which is running on a hypervisor. Here the hypervisor represents the external system. The instance model of the VM must be correlated with the VM running on the hypervisor. The management system must identify the correct VM on the hypervisor at runtime to be able to start, stop, update, and delete it.

There exists different approaches and patterns for management systems to correlate instance models with components in the external systems. Each with its own requirements, constraints, and guarantees. In the following some of these correlation patterns are discussed from the perspective of a management system.

3.2.1 Deterministic Unique Identifier

One approach is to generate a deterministic unique identifier for the component in the external system. The external identifier is generated based on the unique identifier of the instance model, which already has a unique identifier in the scope of the management system. A naive approach would be to use the identifier of the instance model as the identifer of the component in the external system. While this approach works for isolated systems, it does not guarantee that the identifiers are globally unique when multiple systems are involved, especially when the identifier is an increasing number.

A better approach to generate a deterministic unique identifier is to add a unique element to the identifier, such as the name or namespace of the management system. The namespace can be added as prefix or postfix to the identifier. The only two requirements for the namespace are, that it is only used by one management system and that it will not changed. When the external identifier has size or format constraints, a hash function can be used to shorten the identifier. Practically any collision-free deterministic algorithm can be use generate the identifier for the component.

For example, assumed that the instance model of the VM has a unique identifier in the management system. A new unique identifier can be generated from that by appending the reversed domain name of the management system as a prefix. The new unique identifier is then used to create the VM on the hypervisor with that generated identifier. Because the identifier of the VM on the hypervisor was generated by a deterministic algorithm, it can always be recomputed if needed.

The limitation of this approach is that it only works if the external system allows the management system to define the identifier when creating components. If the external systems automatically generates random or increasing identifier for components this approach can not be used. It is also required that the unique identifier of the instance model and the component do not change. Using this approach, only a one-to-one correlation can be maintained.

3.2.2 Selectors

Instance models not always correlate to exactly one component in an external system. For example, the instance model could describe an elastic application with multiple component replicas. When one instance model correlates to multiple components, the management of these correlations becomes more complex.

The simplest approach in terms of keeping track of all related components is to use selectors. Instead of managing all individual references to components of an instance model, the components are all labeled with the same value. The instance model only stores the label selector which is a collection of labels which are used to filter components. Components match the label selector, if they have all the labels defined in the label selector with the same value.

When a new instance model is created a unique label selector must be defined, which does not match components of other instance models. This could be done by generating a unique random value for the used label. Every component created for the instance model is then labeled with the unique value, which is matched by the label selector of the instance model. When the management system needs to perform a management operation, all related components can be queried from the external system using the selector.

Looking at an elastic application as an example, the instance model of that application contains a template for component instances, the label selector, and the number of desired replicas. When the application should be scaled up, the desired replicas number is increased and the management system creates a new component using the template stored in the instance model. The component template also defines the label, e. g., `application: 21bab4d9ccae`, so that all components have the same labels and are matched by the selector. When the application should be scaled down, the desired replicas number is decreased and the management systems queries all component instances using the selector, it chooses one component instance and terminates it.

The selector approach can only be used when the external system supports labels and selectors. New components can be added or removed without changing the instance model. The selector approach becomes very valuable when many systems should be integrated in a loosely coupled manner. Components can have many different labels, to model different groups or to enable integration and discovery of other systems. Selectors may be used to select components not only based on labels, but also based on other metadata, e. g., names, namespaces, and types of the components.

3.2.3 Correlation Identifier

An alternative to the deterministic unique identifier is to use a correlation identifier [HW04]. Instead of defining the unique identifier of the external component, the identifier information is stored in a special attribute designed to keep the correlation identifier. Except from using the special correlation identifier field of the component, this approach is very similar to the deterministic unique identifier approach. The external system must support this approach by allowing to set a correlation identifier and retrieve a component based on the correlation identifier.

3.2.4 Stored identifier

A naive approach to correlate a component to the instance model is to store the unique identifier of the component in the instance model. It can then be used by the management system to perform subsequent management operation on the component. The unique identifer of the component is stored in the instance model after the component was created in the external system.

Typically this approach requires a transaction between the management system and the external system. Thats because when the component is created it must be guaranteed that its identifer is stored in the instance model. If the component is created and its identifer is not set in the instance model, because of a network failure or a failure in one of the systems, the created component will be orphaned. Without a transaction, the management system can be left in a broken state, from which it can not recover, e. g., the component of the instance model is already created but the identifier of the component is not known by the management system.

When the stored identifier approach is used, first a two-phase commit protocol has to be started including the management system and the external system. Then the component is created, the resulting identifer is stored in the instance model and the transaction is then committed. If anything goes wrong everything is rolled back and the management system can try to create the component again.

The limitation of this approach is that a two-phase commit is required involving the management system and the external system. When the stored identifier approach is used without a transaction, a failure of the management system potentially results in an inconsistent state and orphaned components. The requirement to support transactions renders the stored identifier approach unsuitable for the Controller and Reconciler Pattern which is based on eventual consistency. Instead the deterministic unique identifier and the selectors approach are better fits for the eventual consistency.

3.3 Overview of Controller and Reconciler Pattern

The Controller and Reconciler Pattern is a cloud native approach to manage applications, components, and infrastructure. A level-based API is used to describe the desired state of the components in the management system database. The controller and reconciler are the active management components of the system.

The controller and reconciler together implement the control loop, which continuously reconciles the component's states. Reconciliation is the process during which the actual state gradually approaches the desired state. Garrison and Nova [GN17] describe the controller as an endless loop through the following steps:

1. `GetExpected()`: read from the level-based API the desired state of the components.
2. `GetActual()`: read from the environment to get the actual state of the components.
3. `Reconcile()`: reconcile the states.

These steps are implemented by a program called *reconciler*, it is invoked by the control loop implemented by the controller program. The advantage of the pattern becomes immediately evident, since the program of the controller is very small and the structure is simple and generic [GN17]. Furthermore, managing applications, components, and infrastructure is as simple as mutating the state in the level-based API. The controller will read the change the next time `GetExpected()` is called and trigger a reconciliation.

The Controller and Reconciler Pattern is used by Kubernetes in practice to manage containerized applications and cloud native components [BGO+16; Kub20], such as load balancers, storage volumes, and cluster nodes. Kubernetes is a container orchestration engine, which evolved from the Borg and Omega container-management systems at Google [BGO+16]. All documentation related to the Controller and Reconciler Pattern is coupled to the terminology of Kubernetes. In Kubernetes the terms "controller" and "reconciler" are often used as synonyms. In the following the Controller and Reconciler Pattern is defined and explained in the context of general application management in a technology agnostic way.

3.4 Formalization of the Controller and Reconciler Pattern

The Controller and Reconciler Pattern is based on a level-based API (Definition 3.4.1) which stores the state descriptions of the managed components. These state descriptions are modeled using *Controller and Reconciler Component Models (CRC Models)* (Definition 3.4.2), in Kubernetes they are called “Kubernetes Objects” or “API Resources”. CRC Models describe the desired and actual state of managed components and are synchronized by a controller (Definition 3.4.4) and reconciler (Definition 3.4.3) with the managed components. Reconcilers are stateless functions which use the level-based API to compare the desired states with the actual states, and perform actions to bring them closer to each other. The controller calls the reconciler periodically or when the desired state or the actual state of a component has changed. The Controller and Reconciler Pattern guarantees that eventually the actual state of a component matches the desired state defined in the CRC Model.

Definition 3.4.1 (Level-based API)
An idempotent interface to declaratively describe state. It provides durability, availability, and eventual consistency.

Definition 3.4.2 (Controller and Reconciler Component Model (CRC Model))
A dynamic state model which describes the desired state of a managed component and the observed actual state of a managed component. A reconciliation process enforces the desired state of the managed component and updates the actual state in the model.

Definition 3.4.3 (Reconciler)
A stateless function which implements the reconciliation process of a CRC Model. The logic of a reconciler is level-based or idempotent.

Definition 3.4.4 (Controller)
A stateless component which manages the reconciliation of a CRC Model. The controller invokes the reconciler.

Definition 3.4.5 (Storage)
A component which implements the level-based API.

Definition 3.4.6 (Desired State Information)
The part of the CRC Model which defines the desired state of the component.

Definition 3.4.7 (Actual State Information)
The part of the CRC Model which describes the actual state and instance information of the component.

State is managed on a per component basis. A component can be anything: a software component, infrastructure, a cloud service, or a group of components forming an application. The state is stored in a storage component, which implements a level-based API. The components and their states are prescribed by the Desired State Information (Definition 3.4.6) and described by the Actual State Information (Definition 3.4.7) of their CRC Model.

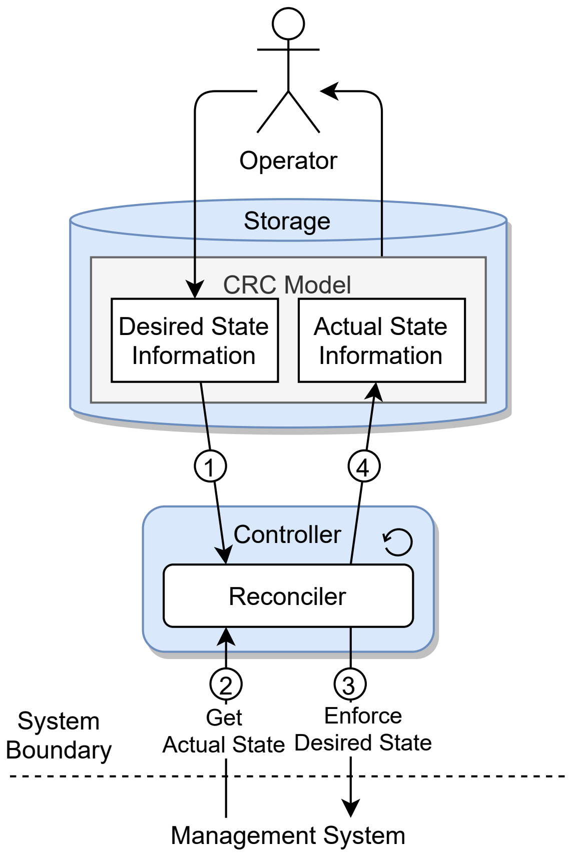

Figure 3.1: Data flow of the Controller and Reconciler Pattern

Figure 3.1 gives an overview of the involved systems and their interactions. *CRC Models* are created by *Operators*, which are either humans or other systems. They specify the *Desired State Information* of the *CRC Models*. The *Controller* invokes the *Reconciler* for each *CRC Model* individually. The *Reconciler* interacts with the *Storage*, which stores the *CRC Models*, and a *Management System* which can be any external system that performs management operations. It reads the *Desired State Information* of the *CRC Model* (1) and the actual state from the *Management System* (2). If they do not match the *Reconciler* enforces the desired state in the *Management System* (3). Finally, the *Reconciler* writes the *Actual State Information* observed from the *Management System* to the *CRC Model* (4). The *Reconciler* is called periodically by the *Controller*, or when the *Desired State Information* has changed. The *Operator* can then read the *Actual State Information* of the *CRC Model* to get information about the actual state of the component.

3.5 Controller and Reconciler Component Model

The CRC Model is a dynamic state model of a component, see Definition 3.4.2. It pre-scribes the desired state of the component and is defined by an operator. It also describes the actual state of the component, which is updated based on observations of the actual state. The goal of the model is to support the eventual consistent reconciliation process of the actual state with the desired state.

The dynamic nature of the CRC Model, allows it to describe the state of a component throughout the complete lifecycle of a component. The CRC Model is not just a model to describe pre- and postconditions of management operations. Rather, it allows to think about application management as a continuous process. The continuous management process starts when the CRC Model for a component is created in the storage and ends when the CRC Model and the component are destroyed and removed from the storage. Over the lifetime of the CRC Model, it is updated to reflect the latest observed actual state of the component or when the operator changes the desired state. In other words, the CRC Model is synchronized with the component.

A CRC Model can be created by an operator from a deployment model. A deployment model defines the desired state of one or multiple components, which is mapped to the Desired State Information of one or multiple CRC Models. The Actual State Information of the newly created CRC Models is either not set or in its initial state, because at this point the components do not exist. The desired state of the CRC Model can be defined and managed using the IaC approach. After the first reconciliation, an operator can see the Actual State Information of the component via the level-based API.

The semantic of a CRC Model is defined by its type. The CRC Model Type defines the type of component it models, what parameters these components have as part of their Desired State Information and what Actual State Information they provide. For example, an IoT Device CRC Model Type can used to define the characteristics of IoT devices and a Container CRC Model Type can be used to define docker containers.

3.5.1 Desired and Actual State Information

Desired and Actual State Information about the component are distinguished in the CRC Model, to separate and enforce the responsibilities of operators and controllers. Operators describe and define the desired state they want. They are allowed to modify the Desired State Information of the CRC Model. Controllers can only read the Desired State Information and are not allowed to change the desired state of the component they manage. The controllers are responsible for managing the actual state and interact with the management systems.

There are multiple reasons for the separation of the Actual and Desired State Information and the continuous management process:

1. Operator and controller can run concurrently without lost updates of the state information in the storage.
2. An operator can change the desired state while a component is in an intermediate state.
3. An operator can change the desired state without knowing the actual state.
4. A continuous management process can prevent configuration drift.
5. Temporary failures of controllers, management systems, and the components are tolerated.
6. Temporary failures do not cause rollbacks of the desired state changes made by an operator.
7. Desired state and actual state are eventual consistent.
8. From the perspective of an operator, the management system is idempotent.

```yaml
softwarePackage: tempReader-1.2.1
interval: 10s
location: entrance
connection:
  url: mqtt://mqtt.example.org
  user: device-26
  password: QHdE6mFIBiTCdiqmnE5h
```

Listing 3.1: Example desired state definition for the motivating scenario IoT application in YAML format

Because of the strong synchronization between the model and the component, a component can not exist without their CRC Model. As a result, the set of all CRC Models provides a complete definition of the applications, infrastructures, and components. This conceptual model is also known as IaC [Mor16; Mor21]. The CRC Model is represented by a data structure which can be controlled with a version control system and applied to the level-based API, which stores it in the storage system. In the level-based API, a CRC Model is an atomic data structure. Which means creating, updating, and deleting a single CRC Model is an atomic and idempotent operation.

3.5.2 Desired State Information

The Desired State Information is the part of the CRC Model which contains all information about the goal state of the component. The desired state is defined by an operator of the component. The CRC Model Type defines a metamodel for this information, so the CRC Model contains only relevant information for the specific type of component. Based on the Desired State Information a desired state model of the component can be created.

When deploying a new component only the desired state of the component needs to be defined by an operator. The operator can use the IaC approach and create a component definition file. The definition file contains the serialized CRC Model with only the Desired State Information. This file can be sent to the level-based API to create the component. To modify the component, the operator changes the desired state in the definition file and sends it to the level-based API again.

Listing 3.1 shows an example of the Desired State Information. The desired state is given in YAML format similar to how it would be defined in an IaC component definition file.

3.5.3 Actual State Information

The state of a component is represented by the Actual State Information. It is a snapshot of the actual state of that component for the operator. The Actual State Information describes the last observed state of the component. It is written by the reconciler and read by the operator, as shown in Figure 3.1. The purpose of the Actual State Information is to give the operator the ability to monitor the actual state of a component. The Actual State Information is CRC Model Type specific. It contains all information needed by the operator to reason about the actual state of the component.

```yaml
installed: true
configured: true
running: true
connected: false
message: "Connecting to broker 'mqtt://mqtt.example.org'"
```

Listing 3.2: Example Actual State Information of the motivating scenario IoT application in YAML format

The Actual State Information is a side product of the reconciliation process of the CRC Model. After each reconciliation the reconciler updates the Actual State Information based on the observed actual state used during reconciliation. The Actual State Information is disposable, this means it can always be recreated by observing the actual state. Therefore, it is not read by the controller or reconciler during reconciliation, but instead the real actual state of the component is observed. This limitation on the data flow is important to guarantee eventual consistency, because it guarantees that there are no loops in the data flow.

As a consequence, it is guaranteed that a reconciler always uses the latest observed actual state and does not use a stored instance model which might be stale. However, the Actual State Information may not be up to date when it is read by the operator. But at least, monotonic read consistency should be guaranteed by the storage, so an operator will never see an older version of the Actual State Information.

Listing 3.2 shows the corresponding Actual State Information of the example Desired State Information shown in Listing 3.1. They give an operator only a very abstract view on the actual state of the application, such as simple boolean values indicating if the application is installed, configured, and running. The shown Actual State Information also contain reconciliation information, such as the `message` property, which indicate the last performed step of the reconciler.

3.6 Basics of the Storage and the Level-based API

The storage is a potentially distributed database which stores the CRC Models, see Definition 3.4.5. It provides a level-based API used in the Controller and Reconciler Pattern. The storage is only accessed via the level-based API and therefore they are used as synonyms. As a level-based API, the storage only supports idempotent operations. The storage decouples operators and controllers. It enables resilience, scalability, reliability, and availability.

The storage supports the following basic operations:

1. Create a new CRC Model in the storage.
2. Get an existing CRC Model from the storage.
3. Update an existing CRC Model in the storage.
4. Delete an existing CRC Model in the storage.

```yaml
name: temperature-app-temp-reader
labels:
  application: temperature-app
  component: temp-reader
  creator: application-controller
finalizers:
- component-controller
deletion: null
parentRef: temperature-app
```

Listing 3.3: Example of CRC Model metadata in YAML format

The storage is a central component in the Controller and Reconciler Pattern. It stores data durably and manages the CRC Models. The storage is the only stateful component because it stores persistent data.

The storage does not interpret or process the Desired or Actual State Information of the CRC Models it stores. From the prescriptive of the storage this information is opaque. The operations that the storage supports are therefore CRC Model Type agnostic and any type of CRC Model can be stored in it. Which increases the reusability and loose coupling. However, the storage need some metadata about the CRC Models to manage them.

3.6.1 Metadata for the CRC Models

Besides the already discussed Desired State Information and Actual State Information, a CRC Model also has metadata attached to it. The metadata is required to hold all the non CRC Model Type specific information, which are needed by the different components of the Controller and Reconciler Pattern. For example, the metadata contains the deletion flag, the parent reference, labels, and the finalizers of the component. Most importantly, they contain the unique name of the CRC Model, which defines the identity and makes it possible for CRC Models to reference each other. Listing 3.3 shows example metadata, which is explained in this section.

Accessing CRC Models from the storage can be done by name or labels. Labels are key value pairs of strings, which can be used to represent arbitrary information. The labels are stored in the `labels` attribute as a map of strings to strings. The storage allows to filter the list of CRC Models based on the value of the labels, using a label selector.

For lifecycle management of components, which is explained in Section 3.6.3, finalizers and a deletion flag are stored in the metadata. The finalizers of a component are stored as a list of unique strings which identify the finalizers in the `finalizers` attribute. A finalizer registers itself by adding its unique name to the list of finalizers. The `deletion` attribute holds the deletion status of the component. It can be realized as a deletion flag, a boolean attribute which is set to true if the component is marked for deletion. Alternatively, the `deletion` attribute can be unset until deletion and then set to the timestamp of the deletion, as shown in Listing 3.3. Only the storage is allowed to set the value of the deletion attribute, but operators and controllers can read the value.

When using the Operator Pattern, which is explained in Section 3.9, CRC Models may have parent-child relationships, which should also be expressed in the CRC Models. The parent CRC Model often contains references to its children in its Desired State Information, because it is type-specific which CRC Model are referenced as children. However, any CRC Model may be a child of another CRC Model, therefore the child role is not type-specific. Therefore, the metadata contains an optional `parentRef` attribute, which is the name of the parent CRC Model or null if the CRC Model has no parent. This attribute is used by the controller to trigger a reconciliation of the parent when the child CRC Model has changed.

3.6.2 Level-based API

The term level-based API, as defined by Definition 3.4.1, refers to idempotent data-driven systems and the APIs provided by these systems. In this section the API of the storage is described. The storage handles the CRC Models as data objects. A data object is a data structure which can be serialized and deserialized in different formats, such as JavaScript Object Notation (JSON)[2] and YAML[3]. The data objects represent CRC Models with a concrete syntax and structure, so that they can be serialized and deserialized. The data objects are stored as documents in a database.

The CRC Model Type defines what the concrete syntax and structure of the data object looks like. In general, the structure of the metadata is not affected by a CRC Model Type. A data object has the following three attributes: `metadata`, `spec`, `status`. The `metadata` attribute contains the data structure which defines all the metadata of the CRC Model. The metadata data structure is the same for all CRC Model Types. The `spec` attribute holds the type-specific data structure representing the Desired State Information. The Actual State Information is stored in the `status` attribute, which is also a type-specific data structure. The CRC Model Type may also be stored in a serialized form in the data object, to let the storage handle different types at runtime.

The key of designing a CRC Model Type data structure is to keep it coherent, simple, and concise. The data structure used for the spec and the status should be domain-specific and contain the Desired and Actual State Information in a form that makes sense to a human operator.

3.6.3 Lifecycle of CRC Models

The storage uses garbage collection to handle the deletion of CRC Models. An operator can not delete a CRC Model from the storage without giving the controller and reconciler a chance to clean up the created components and allocated resources. Therefore, an operator can only mark the CRC Model for deletion, but it is not immediately removed from the storage. In combination with finalizers, the garbage collection allows to implement an eventually consistent deletion and cleanup process for CRC Models.

In the context of garbage collection, a finalizer is a function which is executed by the garbage collector before an object is garbage collected. A finalizer cleans up allocated resources and handles the destruction of an object, this is called finalization. The storage allows to register finalizers

[2]https://www.json.org/

[3]https://yaml.org/spec/1.2/spec.html

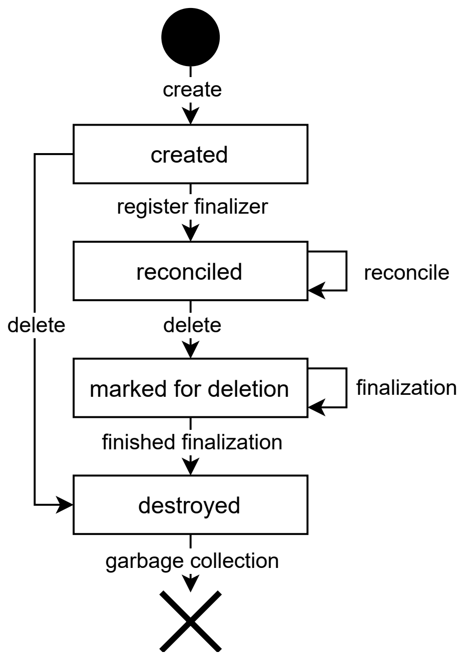

Figure 3.2: Lifecycle and garbage collection process of a CRC Model in the storage

for CRC Models, so that reconcilers can delete the component and clean up resources they have allocated during reconciliation of the CRC Model. The finalization process is asynchronous and eventually consistent. This also means that the deletion of CRC Models is asynchronous and eventually consistent.

Figure 3.2 shows the lifecycle of CRC Models in the storage. After a CRC Model was created, the reconciler registers itself as finalizer, to handle the cleanup before deletion of the CRC Model. Only after the finalizer is registered it starts to reconcile the CRC Model. When an operator wants to remove a component he marks the CRC Model for deletion. The reconciler will stop reconciling the CRC Model and performs the finalization. When the finalization is complete, the reconciler notifies the storage, by deregistering the finalizer from the CRC Model. Eventually the destroyed CRC Model is garbage collected by the storage. If an operator marks an CRC Model for deletion before a finalizer is added, it is immediately in the destroyed state.

3.6.4 Consistency Constraints and Guarantees

The storage is an potentially distributed eventually consistent database. In this section the guarantees and constraints are explained.

The storage guarantees monotonic read consistency for CRC Models. This means the storage guarantees that the retrieved CRC Model is not older than the previous retrieved CRC Model. This is guaranteed for both, the operator and the controller. The storage also prevents lost updates, by rejecting conflicting write operations. Operators and controller treat such rejected write operations as temporary failure and try again later.

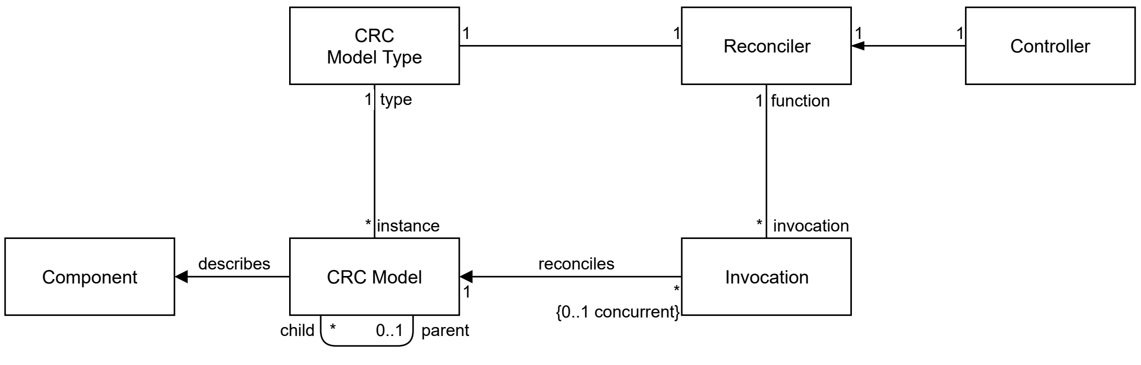

Figure 3.3: Domain model of Reconciler, Invocation, and CRC Model

When reading multiple CRC Models from the storage at once, it does not guarantee a strictly consistent view. Instead the view of multiple CRC Models is only eventually consistent. This means a change to multiple CRC Models, may not be observed at the same time, resulting in an inconsistent view.

The storage should guarantee read-your-creation consistency. This means that an operator will immediately see a CRC Model he created. This is an addition to the monotonic read consistency and should prevent edge cases where a created child CRC Model is not visible during finalization.

3.7 Concept of a Reconciler

A reconciler is a function which processes a single CRC Model at a time and executes management logic to synchronize the desired state of a component with the actual state. Reconcilers are stateless functions, each CRC Model Type has its own reconciler function. When invoked by the controller, the reconciler function processes exactly one CRC Model, see Figure 3.3.

The reconciler can be scaled horizontally to reconcile multiple CRC Models concurrently. But the same CRC Model must not be reconciled concurrently by multiple invocations. Previous reconciliation invocations for that CRC Model must end before a new reconciliation is started for the CRC Model. Therefore, a special work queue is used in the controllers, which is explained in Section 3.8.1.

Garrison and Nova [GN17] separate the reconciler in four separated functions: `GetActual()`, `GetExpected()`, `Reconcile()`, and `Destroy()`. In this work all functionality is combined into a single function, which performs the following tasks:

1. Get the desired state from the storage.
2. Observe the actual state from the management system.
3. Compare and reconcile the states by executing a single step based on the difference between the actual state and the desired state.
4. If the CRC Model is marked for deletion, the reconciler cleans up all related components which were created previously by the reconciler. This is also done in a step by step fashion.

5. After the execution of each step the Actual State Information of the CRC Model is updated to propagate the actual state to the operator.

A reconciler is level-based, it only uses the desired and actual state. State changes and events are opaque to the reconciler, even when triggered by an event, it reads from the level-based API. As a result, the reconciliation process is eventually consistent and resilient to any temporary failures and missed events.

Its very basic consistency constraints allows a reconciler to integrate basically all management systems using common APIs and interfaces. This makes the Reconciler Pattern suitable for infrastructure and application management use cases which require integration of existing management systems and tools. Although, only eventual consistency is guaranteed, systems which support strict consistency can be integrated. In the following section the behavior of a reconciler is described in detail.

3.7.1 Behavior of a Reconciler

Figure 3.4 shows the behavior of a reconciler on a logical and abstract level. First the reconciler gets the desired state and observes the actual state of the component. If the CRC Model is not marked for deletion, the reconciler checks if it is registered as finalizer for the CRC Model in the storage, see Section 3.6.3. If not, it registers itself as finalizer, else the CRC Model Type specific reconciliation logic is executed. The implementation of the component management logic follows the chain structure shown in the right dashed area of Figure 3.4.

The content of the dashed areas in Figure 3.4 is specific to an individual reconciler and the diagram only shows an example. The structure of the reconciliation logic is a chain of decision nodes, where the alternative branch is followed by exactly one action. The example shows the following reconciliation behavior: it installs the `temp-reader` if it does not exist, updates the `temp-reader` configuration if it does not match the desired configuration, and starts the `temp-reader` if it is not running. Also more complex management behavior is possible, e. g., creating and starting sub components in a specific order, waiting for components to become ready, implementing different update strategies such as rolling updates, restarting failed components, and performing periodic tasks. The reconciliation chain performs actions based on the actual state, rather than using a predefined control flow. This means a reconciler will not create a component and immediately start it, but instead always checks the actual state before performing an action.

As a result, typically the execution time of the reconciler function is short. Instead of waiting and blocking the invocation, the reconciler should return immediately and let the controller requeue the CRC Model for a later reconciliation. For example, to wait for some long running external process to finish, the reconciler returns the control flow to the controller and specifies a rescheduling delay, e. g., 100 milliseconds. This non-blocking step by step behavior is important, because management operations often take some time and may fail. A reconciler must be able to recover from failures and react to changes made by an operator. This is not possible if it blocks the control flow while waiting for long running management operations or fails during a sequence of operations.

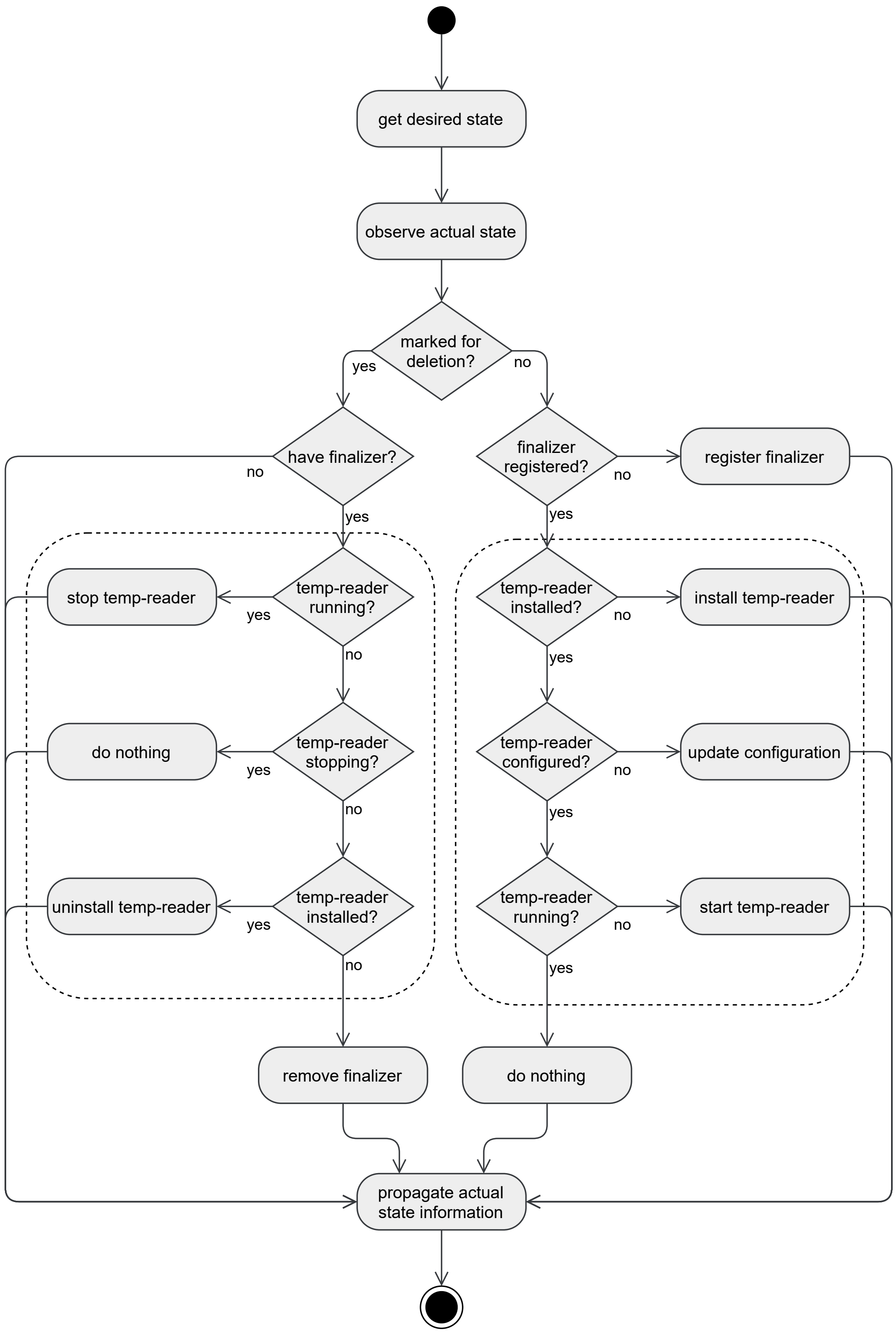

Figure 3.4: Activity diagram of a reconciler

The reconciler is level driven, this means it only needs to know the current state to perform its job. It does not depend on the state of the control flow and therefore it can easily recover from temporary failures. As a result, a reconciler does not need additional recovery logic. The normal control flow of a reconciler transparently handles the recovery from failures and long running tasks.

On the left side of Figure 3.4 the finalization process is shown. It is executed when the CRC Model is marked for deletion and the reconciler is registered as finalizer. The content of the left dashed area shows an exemplary finalization process. When the finalization process completes the reconciler removes the finalizer from the CRC Model, so it is garbage collected by the storage.

The reconciliation process guarantees that eventually the actual state and the desired state are reconciled. And the finalization process guarantees that eventually all resources and components are cleaned up.

3.8 Role of the Controller

On a high-level, controllers and reconcilers can be considered one component and often the term “controllers” is used to refer to both. But on a detailed level there is a clear separation. For each reconciler there is one controller, which invokes the reconciler function. The controller uses an internal work queue to queue and schedule reconciliations of CRC Models. The work queue guarantees that a CRC Model is never reconciled by multiple invocations concurrently.

The reconciler is a level-based component. It does not require a history or information about changes. This makes it very simple for the controller to invoke the reconciler. The controller can invoke the reconciler when ever it needs to reconcile a CRC Model. When invoked, the controller passes the name of the CRC Model, which should be reconciled, as an argument to the reconciler. Typically, the controller schedules the invocation on some interval.

3.8.1 Controller Work Queue

Each controller has an internal work queue which contains reconciliation requests. It contains the name of the CRC Model which should be reconciled. The work queue does not contain duplicate requests, this means when a CRC Model should be reconciled, but the queue already contains a request for that CRC Model, no new requests are added to the queue. The work queue is only stored in-memory and during startup of the controller all CRC Models are added to the queue. As a result, the controller is stateless and can be scaled horizontally by partitioning the CRC Models.

To avoid consistency issues in the reconcilers, the work queue uses an algorithm to ensure that a CRC Model is not reconciled concurrently [Guo19]. However, different CRC Models are independent and are reconciled concurrently. Therefore, the work queue tracks which CRC Models are currently being processed, and only add requests into the queue if they are currently not being processed.

The controller schedules the reconciliation of all CRC Models of the CRC Model Type handled by the reconciler. The scheduling interval is called *sync period* and its value is defined by the controller, e. g., 5 minutes. After this duration the controller will requeue the CRC Model for reconciliation. This guarantees the eventual consistent behavior of the Controller and Reconciler Pattern.

3.8.2 Efficient Detection of Changes

While scheduled reconciliation is enough to guarantee eventual consistency, it results in a slow and inefficient reconciliation process. To increase the reconciliation speed and prevent unnecessary reconciler invocation, notifications and the Observer Pattern are used, while not violating eventual consistency. Change notifications do not replace the scheduled invocations of the reconciler, neither do they change that the whole system is level-based. The Observer Pattern is used to trigger a reconciliation when a CRC Model itself or a related one has been changed. This significantly improves the reconciliation speed and results in performance comparable to workflow-based systems.

The Observer Pattern is implemented using *watches*. They are a concept of Kubernetes [Kub20], to efficiently detect and propagate changes in resources. In the context of this work this means, that changes to CRC Models in the storage are detected and the observers are notified. This allows a controller to create a watch in the storage, to be notified about any changes to CRC Models of a specific type. The controller can then filter and create reconciliation requests from the received notifications and put them into their work queue.

Watches are only used to optimize speed, delays, and overall efficiency of the Controller and Reconciler Pattern. They may not provide guaranteed delivery, notifications may be dropped, and may only be stored in-memory or not at all. This makes it possible to scale the notification system horizontally to create a distributed, highly available, resilient, and performant change notification system. To take full advantage of watches, a controller (i) must watch for notifications of the CRC Model Type of the reconciler, (ii) should watch for notifications of child CRC Models the reconciler created, and (iii) may watch for notifications of CRC Models that are related, e. g., referenced via label selector [Kub20].

However, state changes of the environment and the component itself are opaque to the controller. It can not be assumed that a controller can subscribe to state changes of a component in an external management system. Therefore, to decrease the delay between reconciler invocations, in scenarios where the reconciler needs to wait for some external process to finish, the reconciler can inform the controller about an optimal rescheduling delay. The controller will add the CRC Component to the work queue after the delay.

3.9 Operator Pattern

The controller can function as an operator and use the storage as management system to create and manage child CRC Models. This is known as Operator Pattern from Kubernetes [The21]. It is used to automate tasks of human operators. For example, scaling application instances, creating backups, and managing complex systems. In the following, terms and scenarios are introduced to refine the possibilities given by the Controller and Reconciler Pattern.

An operator is a person or system that creates CRC Models and defines their desired state. When a controller functions as an operator, it delegates some management work to other controllers, by creating CRC Models for them, as illustrated in Figure 3.5. In the figure, *Controller 1* is the operator of the *CRC Model 2*. In such a constellation, the controller which acts as an operator interacts with multiple CRC Models. To distinguish these CRC Models, the terms “parent” and “child” CRC

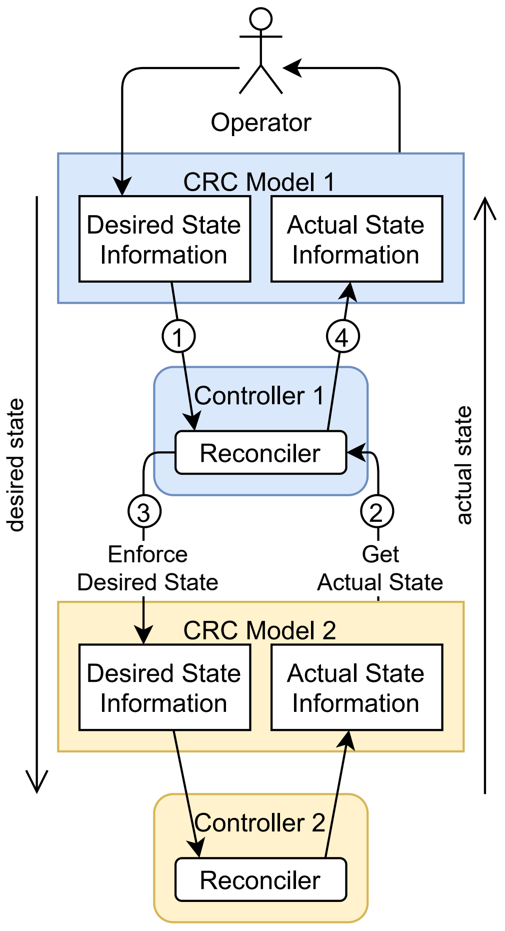

Figure 3.5: Operator Pattern data flow

Models are introduced. When a controller act as an operator, the CRC Model which is reconciled by the controller is called the parent (marked blue in Figure 3.5), while the CRC Model which is created by the controller is called the child (marked yellow in Figure 3.5). When a CRC Model is created by a human operator, it has no parent. Child CRC Models are always created at runtime by a controller after the parent CRC Models has been created.

The concept of type inheritance is not used with the Controller and Reconciler Pattern. Instead the Composite Reuse Principle [Kno02, p. 17] is applied, which prefers delegation and composition over inheritance [GHJV95]. Delegation and composition is an alternative to CRC Model Type Inheritance [GHJV95], which enables reusability without introducing complex inheritance hierarchies.

For the Operator Pattern three basic scenarios can be distinguished: delegation scenarios, template scenarios, and composition scenarios. The template scenarios and the composition scenarios can be seen as special cases of the delegation scenarios. The following sections explain how the Operator Pattern can be used in these scenarios.

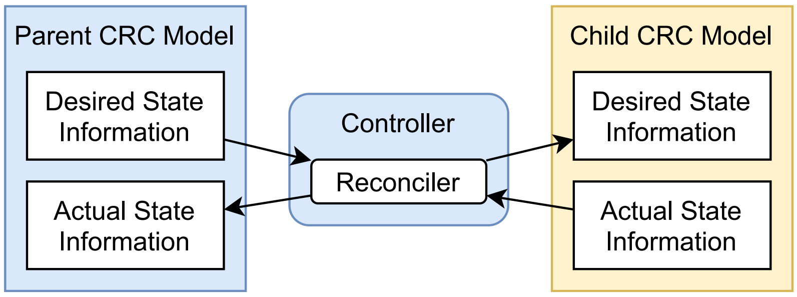

Figure 3.6: Delegation scenario

3.9.1 Delegation Scenario

In the delegation scenario an existing CRC Model Type is reused in a specialized context. For example, a database reuses a VM CRC Model Type or a backup reuses the Cron Job CRC Model Type. In the delegation scenario, it is possible to maintain a collection of common high-level components that handle the low-level integration. These common components are then used by domain-specific CRC Model Types as a template. Domain-specific CRC Model Types are only virtual components, because they are only an abstraction layer on top of existing component types.

Figure 3.6 illustrates the delegation scenario with a parent and a child CRC Model. The *Reconciler* of the *Parent CRC Model* creates the *Child CRC Model*. The *Desired State Information* of the parent is transformed to *Desired State Information* of the child by the *Reconciler*, using a domain-specific mapping and default values. For the operator of the parent CRC Model the transformation is transparent. The *Controller* of the *Parent CRC Model* is the operator of the child component. It creates, updates, and deletes the child component depending on the Desired State Information of the parent CRC Model. This allows a human operator to focus on the management of domain-specific virtual components.

3.9.2 Template Scenario

The template scenario is a special form of the delegation scenario. It focuses on the management of multiple similar instances of a component template. An operator defines the component template in the parent CRC Model. The template contains the Desired State Information of the child CRC Models and additionally information about the desired number of child component instances.

When the *Reconciler* of the *Parent CRC Model* is invoked, it acts as an operator and checks how many *Child CRC Models* already exist and creates new ones based on the template or deletes them, see Figure 3.7. The *Reconciler* ensures that the *Child CRC Models* matches the template, and if necessary, updates them. The *Reconciler* also monitors and aggregates the *Actual State Information* of the *Child CRC Models*. As a result, the operator of the parent CRC Model gets an overview of all running instances. Labels and label selectors are used to get all existing child CRC Models from the storage. Therefore, a reconciler does not need to actively maintain a list of all children.

When an operator changes the template in the parent CRC Model, all child CRC Models must be updated as well. For example, if a new software version should be deployed, all instances running the old version must be replaced with instances running the new version. In the template scenario

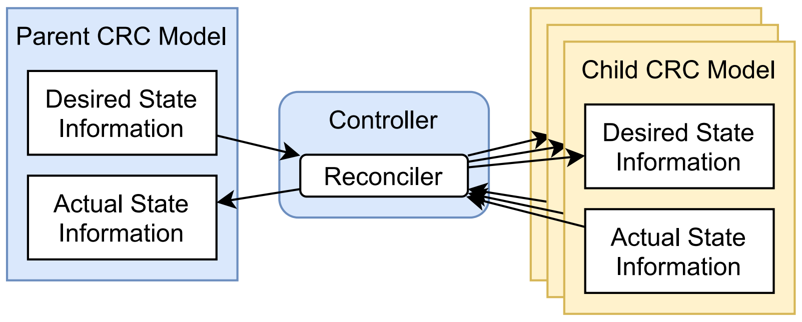

Figure 3.7: Template scenario

the Operator Pattern automates the update process. A reconciler can implement different update strategies, such as blue-green deployments[4], rolling upgrades, and canary deployments[5]. A human operator would only select an update strategy, which is automatically executed by the reconciler when needed.

In the template scenario the Operator Pattern also supports elastic scaling of applications. For example, a horizontal auto-scaler could monitor the utilization of the instances or use other metrics to calculate the required number of instances. This number is then set in the parent CRC Model and the reconciler will automatically create or remove instances.

For example, the ReplicaSet Kubernetes Controller[6] uses the Operator Pattern in the context of a template scenario to enable elastic scalability.

3.9.3 Composition Scenario

In the composition scenario virtual parent components are decomposed into multiple child components by the *Reconciler* at runtime, see Figure 3.8. In contrast to the template scenario the child components may not be similar and have different types. The Operator Pattern allows to reuse existing components and arrange them in a new way, to form a new component. The composition is managed by the *Parent CRC Model*, which defines a domain-specific interface and hides the complexity of the underlying composition.

The Operator Pattern allows to package complex system setups and configurations of multiple components into a single virtual component. Human operators only have to deploy and manage the virtual component. The composition is transparent to the operator of a parent CRC Model. The *Reconciler* handles the decomposition and aggregation into the underlying child components during reconciliation. The *Parent CRC Model* can also be used as a child in another composition, which enables reusability on all layers, from low-level infrastructure components, to application components, to distributed applications.

[4]https://martinfowler.com/bliki/BlueGreenDeployment.html
[5]https://martinfowler.com/bliki/CanaryRelease.html
[6]https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

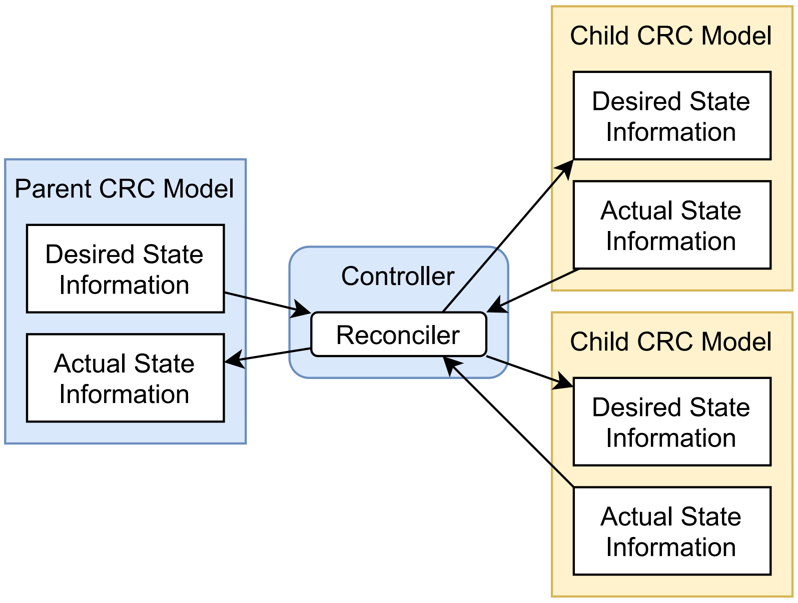

Figure 3.8: Composition scenario

In the context of Kubernetes, the Operator Patten is used to extend its functionality, integrate other cloud native technologies, and automate management tasks. For example, Stash[7] automates backups, Prometheus Operator[8] manages Prometheus clusters, and cert-manager[9] manages Transport Layer Security (TLS) certificates.

3.9.4 Lifecycle of Child CRC Models

The lifecycle of the parent CRC Model always includes the lifecycle of its children. This is also true for transitive child CRC Models. The child CRC Models are created by the reconciler of the parent CRC Model, so children are always created after their parents. When a parent CRC Model is marked for deletion the reconciler also marks all children for deletion during finalization. The finalization of the parent is only completed after the children have been garbage collected by the storage. This ensures that the deletion of a parent CRC Model will also delete all transitive children.

[7]https://stash.run/
[8]https://github.com/prometheus-operator/prometheus-operator
[9]https://cert-manager.io/

4 Concept of Reconciliation-based IoT Application Management

IoT applications are deployed on heterogeneous infrastructures with thousands of devices, which are connected via a potentially unreliable network and are developed and updated rapidly [AS18; Mal19]. To support the operations and management of such applications an automated application management system is required which supports the DevOps practice. The TOSCA standard provides a technology agnostic modeling language for heterogeneous applications and is already used by da Silva et al. [SBH+17] and Schmid [Sch18] to automate the deployment of IoT applications. However, the approach of da Silva et al. [SBH+17] does not automate the management of multiple devices and non of these approaches provide a resilient level-based API to enable the use of IaC and DevOps.

To tackle these issues, this work introduces Reconciliation-based IoT Application Management (RITAM), which consists of (i) the central RITAM Device and Application Manager, (ii) the RITAM Orchestrator which runs as an agent on each device. Both components implement the Controller and Reconciler Pattern and use CRC Models to manage state. The RITAM Device and Application Manager provides a level-based API to manage individual devices and applications, see Figure 4.1. Application templates declaratively define rules, to select devices they should be deployed on. A reconciliation process deploys and updates the applications on the selected physical devices. Therefore, the TOSCA Topology Templates of the applications are sent to the RITAM Orchestrators of the selected devices. The RITAM Orchestrator also implements a level-based API and stores the application definitions as CRC Models in a database on the device. The controllers and reconcilers of the RITAM Orchestrator decompose the applications into their components and invoke the custom reconciliation logic to deploy, update, manage, and delete the individual components on the device. The instance information of components and applications are automatically updated in the CRC Models and sent back to the RITAM Device and Application Manager. As a result, a human operator can see an up to date instance model of the deployed applications and manage the application using IaC.

The next section gives an overview of the architecture, which is explained in detail in Chapter 5 and Chapter 6. The concept is discussed and compared to other approaches in Section 4.2.

4.1 Architecture of RITAM

The architecture of RITAM is shown in Figure 4.2. The central RITAM Device and Application Manager is deployed as a service in the cloud so it can be accessed by operators and can be scaled on-demand. It stores *Device CRC Models* and *Application Template CRC Models*, which are created by the operators. A Device CRC Model defines the properties of a device and information required to connect to the RITAM Orchestrator of the device. The device properties can be used by operators

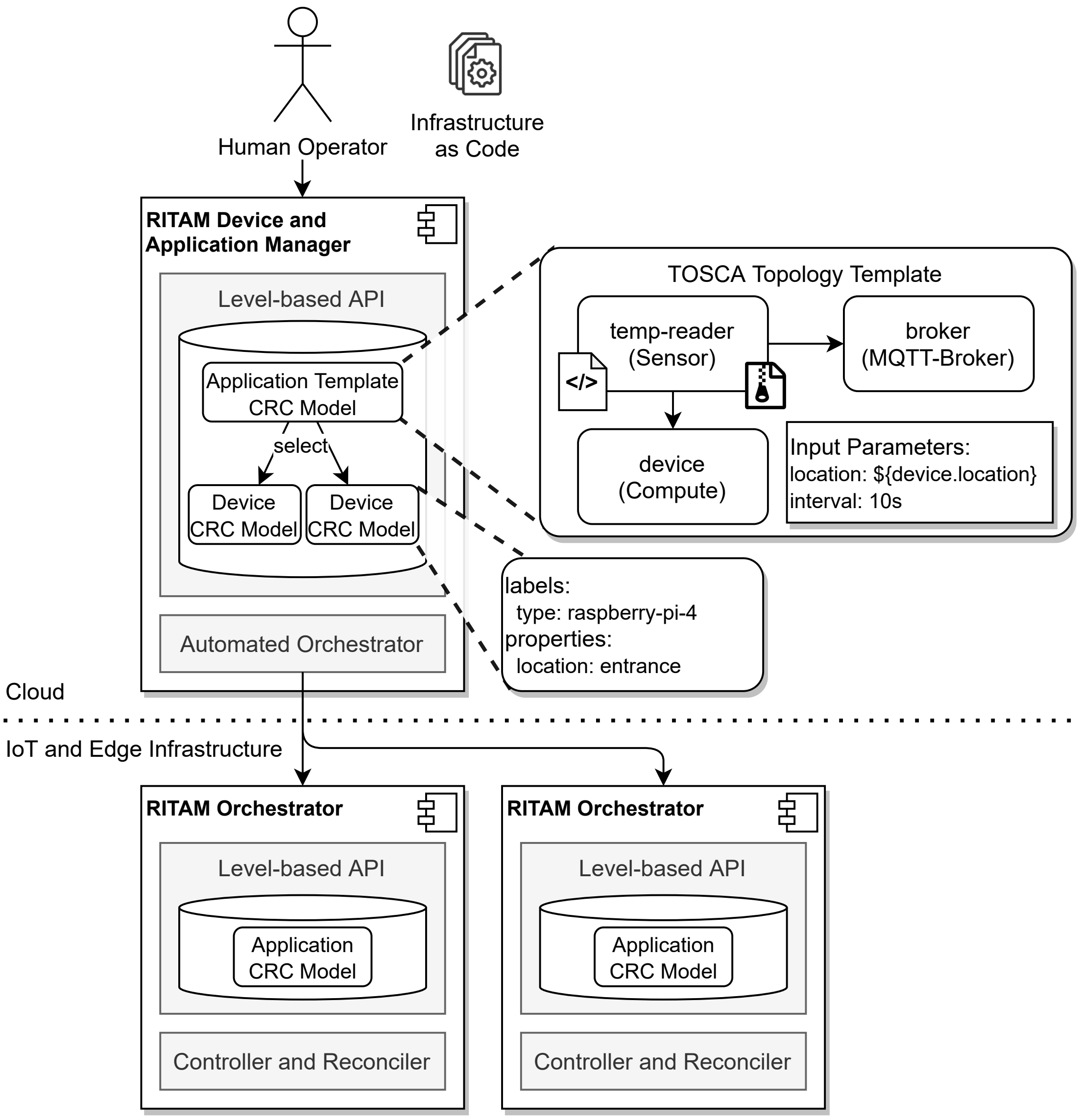

Figure 4.1: RITAM approach

to store custom data for each device, which can be used as parameters for the application templates deployed on the devices. The Device CRC Model metadata defines the name and labels of the device, which are used in the application templates to select devices.

The Application Template CRC Model specifies a TOSCA Topology Template, input parameters, and a device selector. The TOSCA Topology Template is deployed to all devices matching the device selector. The input parameters are used to instantiate the TOSCA Topology Template and can contain references to device properties to customize the application on each device. In Figure 4.1 for example, the devices have the `location` property defined, which is used in the shown TOSCA Topology Template as input parameter.

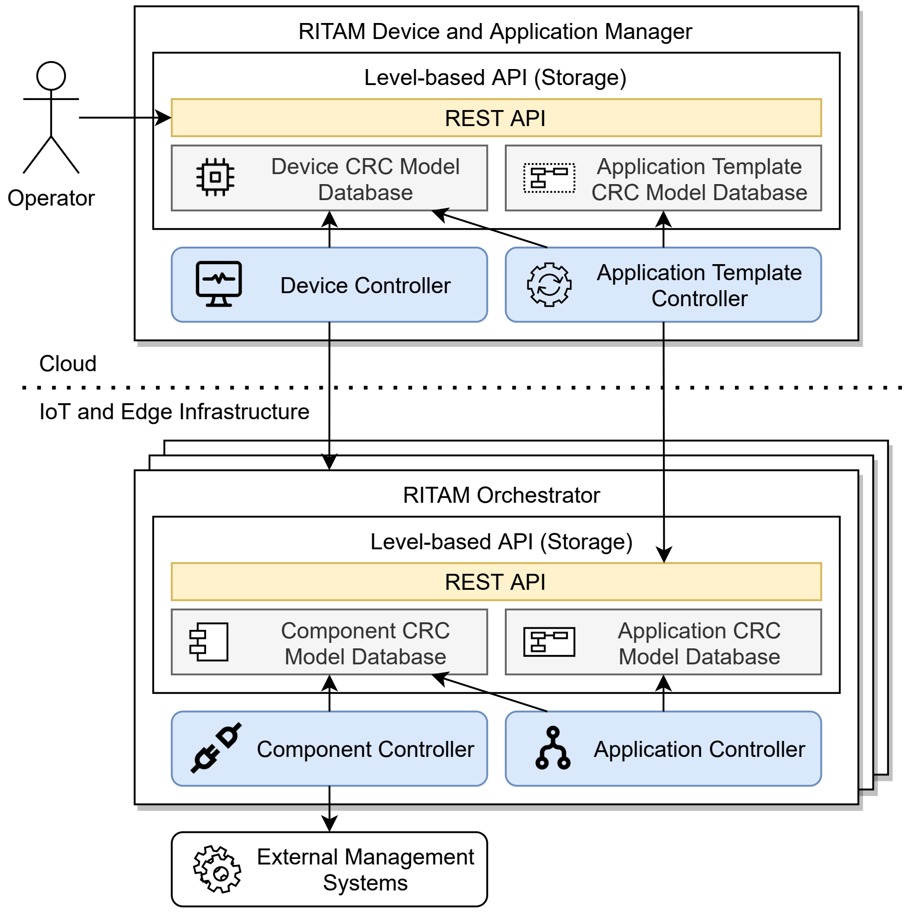
RITAM Device and Application Manager
Level-based API (Storage)
REST API
Device CRC Model Database
Application Template CRC Model Database
Device Controller
Application Template Controller
Operator
Cloud
IoT and Edge Infrastructure
RITAM Orchestrator
Level-based API (Storage)
REST API
Component CRC Model Database
Application CRC Model Database
Component Controller
Application Controller
External Management Systems

Figure 4.2: Architecture of RITAM

The RITAM Device and Application Manager contains an Application Template Controller, which implements the Operator Pattern as defined in Section 3.9. The controller deploys, updates, and deletes the application instances on the selected devices using the level-based API of the RITAM Orchestrator. The RITAM Orchestrator is an agent which runs on each device of the IoT and edge infrastructure. It implements a level-based API, which allows to deploy application instances and to synchronize the desired state and actual state with the RITAM Device and Application Manager. As shown in Figure 4.2 the level-based API is implemented as REST API. The RITAM Orchestrator implements the Controller and Reconciler Pattern and stores the Application CRC Models and Component CRC Models in a database on the device.

In a reconciliation process the application is decomposed into its components by the Application Controller. The Component Controller of the RITAM Orchestrator periodically calls the custom reconciliation operation of each component to reconcile the desired and actual state. This is the equivalent of executing the TOSCA lifecycle operations of each TOSCA Node in a workflow. The custom reconciliation operations can interact with external management systems to perform all necessary actions to install, configure, start, update, and delete the components. During reconciliation, the instance and runtime information of the components and applications are updated and propagated back to the RITAM Device and Application Manager, where they can be observed by the human operator.

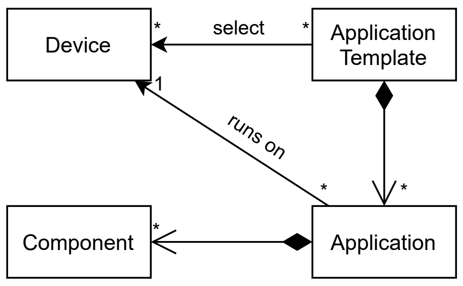

Figure 4.3: Domain model with all CRC Model Types

The complete domain model of RITAM is shown in Figure 4.3. Chapter 5 describes the RITAM Device and Application Manager including the data model and runtime behavior in more detail. The details of the RITAM Orchestrator are explained in Chapter 6.

4.2 Discussion

An essential difference of RITAM to other application management approaches is that it uses reconciliation. The Controller and Reconciler Pattern enables a reliable and resilient reconciliation process. The automatic synchronization between model and component, eliminates many sources of errors and reduces the effort of human operators. RITAM provides a new conceptual model for how operators think about operating applications, infrastructures, and components. From the perspective of an operator, inconsistencies between model and component are transparent, so that model and component can be considered a single thing.

4.2.1 Summary

The RITAM approach combines domain-specific modeling (Device CRC Model and Application Template CRC Model) and general-purpose modeling (TOSCA) to provide domain-specific structures and abstractions while maintaining flexibility. It allows independent management of devices and application templates. Application templates use device selectors which can select thousands of devices using a single label. Applications are automatically deployed on new devices when they match the selectors of the application template. The essential characteristics of IoT devices, namely their physical environment, is reflected by labels and properties, which are used to customize the application instances on the devices. The resilient synchronization process accounts for the unreliability of network connections and devices. As a result, RITAM is a reliable and robust management approach for IoT applications.

RITAM dynamically growths with the size of the IoT applications and infrastructure. Not only does the RITAM Device and Application Manager scale, but also the management concepts and tools. IaC allows to manage constantly changing and growing IoT applications, in terms of requirements, number of applications, number of devices, and complexity. If the manual management of IaC becomes a bottleneck, tools can be introduced for automation and higher level composition, e. g., templating tools and hierarchical decomposition. Large growth of infrastructure can be handled using the Operator Pattern on top of the provided level-based API. For example, an operator may automate the synchronization of devices from an existing database or automate discovery of devices. All these growth scenarios are made possible by the extensibility and eventual consistency of the Controller and Reconciler Pattern.

4.2.2 Error handling

An important aspect of a management system is how it handles errors. A management system is often an integration point of many different systems, which results in many possible sources of errors in the configuration, the software, the network connections, and the combined behavior. Error handling is a cross-cutting concern, because errors can happen anywhere and must be handled appropriately by all components.

The Controller and Reconciler Pattern, which is used by RITAM, is built for failures, this means producing and handling errors is part of its normal operation. For example, a CRC Model update conflict produces an error, which is handled by retrying the operation later. The reconciliation process guarantees that errors are handled in the next invocation. Also, the chain like control flow of reconcilers encourage programmers to write self-healing management operations.

The storage is a generic create, read, update, and delete (CRUD) component, which can be deployed as a highly available distributed database[1] for the central RITAM Device and Application Manager. It decouples the operators and controllers of the CRC Models and creates isolated failure environments. If a controller fails or is unavailable, an operator is still able to perform all operations with the storage. For example, if the Application Template Controller fails to connect to the IoT or edge devices, human operators are still able to see the last observed actual state and can make changes to the desired state of the Application Template CRC Models. A controller isolates the invocation of the reconciler on a per component basis, which means if the reconciliation of a CRC Model fails, the reconciliation of the other CRC Models is not affected.

RITAM does not use workflows and rollbacks as opposed by the TOSCA standard. Temporary errors are handled by retrying the operations. However, persistent errors, such as configuration errors or software bugs, require a human operator to fix them. For deploying the configuration fix or bug fix, only the Application Template CRC Model must be updated by the operator. The reconciliation process will handle the rollout of the update regardless of the current state of the components.

4.2.3 Comparing to Workflow-based Application Management

Compared to workflow-based application management, RITAM has advantages regarding interoperability, scalability, resilience, and durability. First of all, using the Controller and Reconciler Pattern puts less restrictions on data consistency and therefore allows integration with most management system. Reconcilers are not limited to manage the deployment of components, but can perform periodic tasks, such as auto scaling and creating backups. Modeling period task with workflows requires external triggers and additional configuration.

[1]The storage could be realized using etcd (https://etcd.io/) for example.

Workflow-based application management handles failures using rollbacks. However, not always it is possible to rollback a management operation, therefore compensations are used, which can introduce edge cases where the state of the application is unknown [OAS17]. Workflows guarantee consistency on a per management operation level, this means a management operation is atomic and can span multiple components. When multiple components are involved, workflows are slow, because they have to guarantee an all or nothing semantic across a distributed system [Ley00]. The Controller and Reconciler Pattern guarantees the consistency on a per component level, this means a management operation is eventually consistent across multiple components. Failures on a component level do not affect the management of other components.

Workflows are edge triggered, which means a workflow performs actions based on an event triggered by another action. An example of such an event is the completion of an action, which starts the next action in a workflow. A reconciler on the other hand is level triggered, which means it performs an action if the state has a specific value. For example, if a software component is installed and not running, it is started. This also means a reconciler is idempotent.

Workflow-based application management relies on strict ordering of actions and preconditions which have to be met at specific points in the control flow. For example, a backup workflow requires the application to be running. With the Controller and Reconciler Pattern there are no order or time dependencies. For an operator this means, he can change the desired state of different components in any order and at any time without difference in the end result. With the Controller and Reconciler Pattern the dependencies are resolved by retrying the operation at a later point. For example, if a backup should be created at 8:00 am but the application has a failure, the reconciler will retry every five minutes until the application is up again and the backup was successfully created.

4.2.4 Comparing to Component-based Management Systems

The concept of component-based application management where models define the configuration and runtime information of components is not new. Many management systems use some form of model to represent components they manage, e. g., component templates and instance models. These models represent desired state or runtime information about the deployed components. However, these management systems lack a continuous synchronization process for models and components. This results in configuration-drift and can lead to failures, which require a human operator to manually debug and fix the inconsistencies. For example, if a VM was created using AWS CloudFormation but later changed manually by an operator, then the CloudFormation template does not represent the current state of the VM.

The reconciliation process of RITAM solves this problem by (i) tolerating temporal inconsistencies in the context of eventual consistency and (ii) automatically correcting the inconsistency over time by reconciling the desired state of the component model with the actual state of the component. The CRC Model is the single source of truth of the desired state of the component at any time.

4.2.5 Comparing to Configuration Management Systems

RITAM can be compared to configuration management systems. Configuration management systems such as CFEngine, Puppet, and Chef are convergent. According to Barbour and Rhett [BR18], convergence means, “the configuration management tool is invoked over and over: each time the tool is invoked, the system approaches a converged state in which all changes defined in the configuration language have been applied, and no more changes take place." The Reconciler Pattern is very similar to convergence, because a reconciler is also called over and over again by the controller and each time it reconciles the actual state with the desired state.

The main difference is, that configuration management systems define the desired configuration using multiple small modules. The modules provide a model to express the desired state and the logic which enforces it. But in contrast to reconcilers, modules are not isolated and define low-level configuration of the environment.

Most convergence configuration management systems are node[2] based. This means, they evaluate the desired state on a per node basis and not on a component basis. For each node the desired state is defined by multiple modules. The desired state of a node is only implicitly defined by the desired configuration of the different modules.

RITAM provides a similar workflow for operators compared to configuration management systems, but it is application and component oriented. It uses the TOSCA standard to model applications instead of a vendor- or technology-specific modeling language. RITAM also provides the actual state information and an instance model of the applications and components, so operators only have to interact with one system to operate all applications.

4.2.6 Limitations of Reconciliation-based IoT Application Management

RITAM is built on top of the Controller and Reconciler Pattern, which has many advantages over edge-based management systems such as workflow-based systems or script engines. However, its eventual consistency is also a limitation. Management processes in the Controller and Reconciler Pattern are always asynchronous, the desired state changes are immediately visible in the storage but only applied in the background reconciliation process.

With RITAM operators manage applications by creating, updating, and deleting Application Template CRC Models. However, after the creation of an Application Template CRC Model some properties of the Desired State Information should not be changed by an operator. Either because of limitations of the underlying management system or to maintain the component correlation (see Section 3.2). For example, operators should not change correlation identifier in the TOSCA Topology Template, because this would change the identity of a component and the reference to the old component would be lost. Depending on the external management system operators should not change desired state properties that would require a destructive operation and recreation of a component. For such changes, the operator should explicitly delete the Application Template CRC Model and create a new one with the new Desired State Information.

These kinds of operations are very common in cloud migration scenarios, where VMs and databases are migrated from one data center or cloud to another. These scenarios are not covered by the RITAM approach. In future work, concepts for automatic migration of components without recreation of the CRC Model can be investigated.

[2]Nodes refer to compute resources, such as physical or virtual machines on which software can be executed.

5 RITAM Device and Application Manager

The RITAM Device and Application Manager is the central component of the RITAM approach. It is used by operators to define all devices and application templates. All RITAM Orchestrators are controlled by a RITAM Device and Application Manager, which deploys applications on them.

The basic principle of the RITAM Device and Application Manager is to manage devices and application templates independently and then to allow the definition of *n* to *m* mappings between them. For this mapping labels and selectors are used, as shown in Figure 5.1. Devices have labels and application templates define device selectors. This allows an application template to select zero, one, or many devices. Devices can be selected by zero, one, or many application templates, resulting in a loosely coupled *n* to *m* mapping.

In the following the device and application template data models are described in Section 5.1 and Section 5.2. The behavior of the Application Template Controller is defined in Section 5.3.

5.1 Device CRC Model

IoT and edge devices are modeled using the Device CRC Model Type. In the context of RITAM all devices have to run a RITAM Orchestrator agent, which handles the orchestration locally on the devices and synchronizes with the RITAM Device and Application Manager. Listing 5.1 shows an example Device CRC Model. The important attributes are `labels`, `properties`, and `url`.

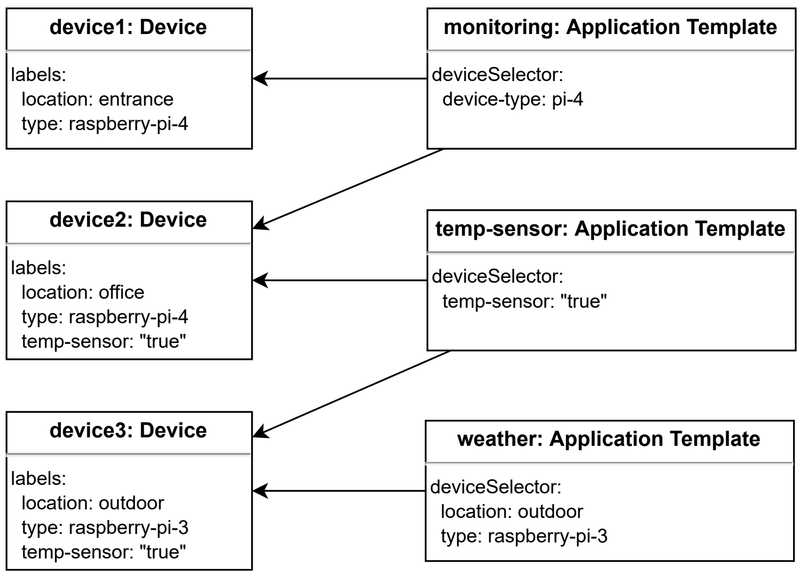

Figure 5.1: Example of devices with labels and application templates with device selectors

```yaml
metadata:
  name: device1
  labels:
    location: office
    type: raspberry-pi
  finalizers:
  - device-controller
  deletion: false
  resourceVersion: 851ba1d0-3d49-4953-8f98-3523f8f5967f
spec:
  properties:
    location: office
  url: http://device1:8080
status:
  message: UP
  lastReconciled: "2021-04-13T18:50:46.818152Z"
```

Listing 5.1: Example of a Device CRC Model in YAML format

Labels are a key-value map of strings. They can store arbitrary information and characteristics of IoT devices, such as location, special hardware, and group assignments. In Listing 5.1 the location and device type is given by a label.

The properties of a device can be used to store device specific information which can be used to customize the applications running on the device. The given example defines the `location` property, to indicate that the device is located in the office. An application deployed on that device may use this location value and include it in MQTT messages, so that messages from different devices can be distinguished and correlated to a location.

The difference between labels and properties is, that labels can be used to select devices and properties can be used as parameters of topology templates. For example, when devices should be selected based on their location, the location should be a label, if applications should be deployed on multiple devices, but customized based on their location, the location should be a property. Typically, label values represent a fixed number of categories while property values are unique to that device.

A Device CRC Model also defines the Uniform Resource Locator (URL) of RITAM Orchestrator agents running on the device. The URL points to the level-based REST API of the RITAM Orchestrator, which is used by the RITAM Device and Application Manager to apply the desired state to the device. It is also used by the Device Controller to periodically check the health status of IoT devices and to update the status in the Device CRC Models.

```yaml
metadata:
  name: temperature-app
  labels: {}
  finalizers:
  - application-template-controller
  deletion: false
  resourceVersion: 8f8731b6-cedf-4afd-9ab9-9430bddd9d7a
spec:
  deviceSelector:
    type: raspberry-pi
  inputs:
    MQTT_URL: mqtt://broker
    interval: 10s
    location: "${device.location}"
  serviceTemplate: file:///resources/temperature-sensor-service-template.yaml
status:
  message: Successfully reconciled
  lastReconciled: "2021-04-13T11:36:36.018574Z"
  instances: 3
  devices:
    device1: up-to-date - Successfully Reconciled
    device2: up-to-date - Successfully Reconciled
    device4: up-to-date - Successfully Reconciled
```

Listing 5.2: Example of an Application Template CRC Model in YAML format

5.2 Application Template CRC Model

IoT applications are modeled using the Application Template CRC Model Type. It represents a template for application instances on individual IoT devices. Listing 5.2 shows an example Application Template CRC Model. The application template defines the desired state of the IoT application, which consist of the device selector, the TOSCA Service Template, and the input parameters.

The device selector defines on which devices the application should be deployed. As shown in Figure 5.1, devices selectors are flexible and support many different uses cases. For example, they can be used to select devices of a specific type, at a specific location, or with specific hardware. Labels and label selectors are defined by the operators and those can be adapted to the specific needs or use cases.

The applications on the IoT devices are modeled using TOSCA, more specifically using the RITAM TOSCA extension, which will be defined in Section 6.2. The Application Templates CRC Model contains a reference to the TOSCA Service Template of the IoT application. The reference is a Uniform Resource Identifier (URI) which uniquely identifies the service template, see `serviceTemplate` in Listing 5.2. Typically, this is a URL where the service template is stored and can be downloaded.

TOSCA Service Templates can be parameterized, reused, and instantiated multiple times with different configurations. To specify values for these input parameters, the `inputs` attribute of the Application Template CRC Model can be used. To customize the individual application deployed by the same application template, the inputs may use the properties of the devices the application is deployed on. For example, the `location` input value in Listing 5.2 uses the `location` property of the device, with this information the temperature values sent by the application can be annotated with the location where the temperature was measured.

Application templates can deploy an IoT application to thousands of devices. To get an overview of the status of all the applications, Application Template CRC Models contain the number of application instances created by the template. Additionally, for each selected device the reconciliation status is contained in the application templates status. This allows to quickly identify problems of individual devices.

5.3 Application Template Controller

The Application Template Controller and the Device Controller are part of the RITAM Device and Application Manager. Each controller executes its respective reconciler, for brevity the term "controller" will be used as synonym for reconciler. The RITAM Device and Application Manager is the central component which is used by human operators to manage IoT devices and applications. The distribution of the applications on individual devices is automated by the Application Template Controller.

The Application Template Controller reconciles the Application Template CRC Models. It uses the device selectors to select devices from the device storage. For each selected device, it reads the actual state from the RITAM Orchestrators and applies the desired state. The desired state for each device is computed from the application template, which contains a TOSCA Service Template and input parameters. The Application Template Controller resolves template variables in the input parameters with the values from the device properties. The resolved template is then applied as Application CRC Model to the RITAM Orchestrator. This is known as the Operator Pattern, see Section 3.9.

Before the Application Template Controller begins to apply the applications to the devices, it adds a finalizer to the Application Template CRC Model. This is required to ensure that the applications on the devices can be cleaned up when the application template is marked for deletion. Here it is also important, that the device selector of an application template is not changed after creation, else the stateless Application Template Controller loses track of device it has deployed the application to. The same holds for removing labels from devices, which are in use by application templates.

After the reconciliation of an Application Template CRC Model, the controller updates the status with the number of deployed application instances and a status message for each individual instance. The Controller and Reconciler Pattern guarantees that IoT applications are eventually deployed and up to date on all selected devices.

6 RITAM Orchestrator

In this chapter the RITAM Orchestrator is introduced. It integrates the Controller and Reconciler Pattern into the TOSCA modeling language and orchestration process. The Controller and Reconciler Pattern is not just used internally in the implementation of the RITAM Orchestrator, but the Reconciler Pattern is applied to the individual TOSCA lifecycle operations of TOSCA Nodes. The integration allows to model TOSCA applications with a resilient, reliable, and self-healing management behavior. It enables the use of TOSCA beyond the initial deployment of an application and makes it possible to apply modifications to deployed applications.

However, a TOSCA Topology Template is a deployment model and TOSCA does not define an instance model for deployed TOSCA topologies. But instance and actual state information are required to enable the integration of the Controller and Reconciler Pattern. Therefore, the Component and Application CRC Model Type are defined. The CRC Models combine the deployment model (desired state) and the instance model (actual state).

The RITAM Orchestrator enables management operations of TOSCA Nodes to use the Reconciler Pattern. This means operations are level-based and eventually consistent. The new reconciler operations are part of a new TOSCA lifecycle interface which is introduced to support the Reconciler Pattern. The RITAM Orchestrator implements two controllers: the Application Controller and the Component Controller. The Component Controller handles the Component CRC Models. It implements the control loop which invokes the reconciler operations of the TOSCA Nodes. The Component Controller operates on the level of a single component and its outgoing relationships. The management of a whole application topology with multiple components and relationships is the responsibility of the Application Controller. Its purpose is to create, update, and delete the Component CRC Models. The Application Controller applies the Operator Pattern (see Section 3.9) to automate the management of the application components.

In the following sections the RITAM Orchestrator is explained in more detail. First, an overview of the orchestration flow is given in Section 6.1. Section 6.2 defines an extension to the TOSCA standard to support the integration of the Controller and Reconciler Pattern. Then in Section 6.3, the CRC Model Types and their relations to the TOSCA templates are described. Details on how the controllers work in the RITAM Orchestrator and how the reconciler operations are executed is specified in Section 6.4.

6.1 RITAM Orchestrator Overview

The RITAM Orchestrator is deployed on the IoT and edge devices and operated by the central RITAM Device and Application Manager. However, the RITAM Orchestrator can also be used standalone, therefore in the following the term “operator” refers to a person or a system which

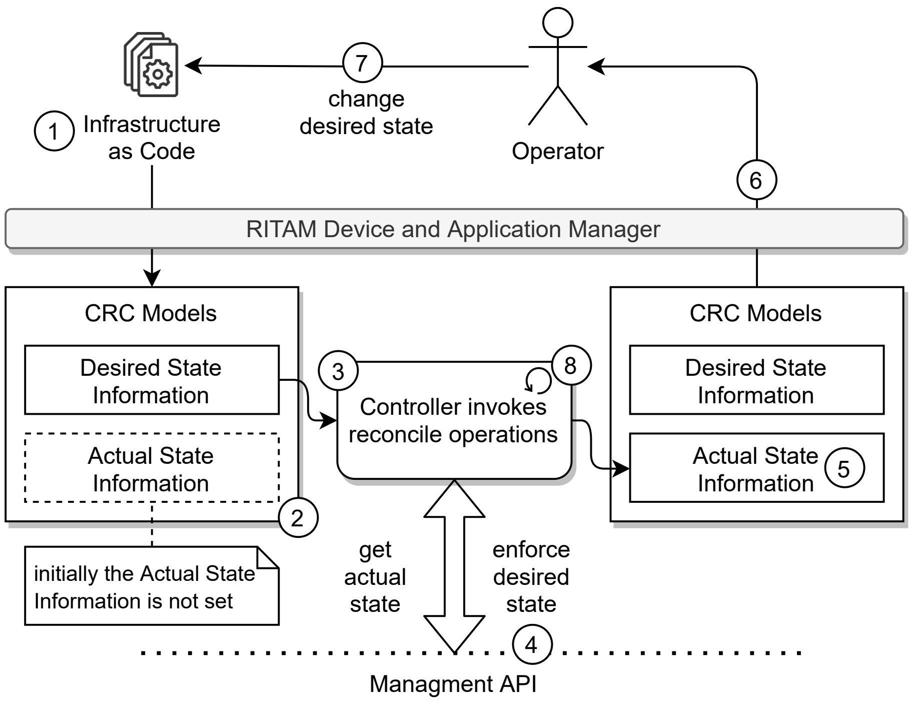

Figure 6.1: Overview of the orchestration flow using the RITAM Orchestrator

defines the desired state of an application. Figure 6.1 shows an overview of the orchestration process. The orchestration flow can be described by the following steps, which are also shown in the diagram:

1. An operator defines the application using IaC and applies it to the RITAM Device and Application Manager.
2. Eventually, an Application CRC Model is created in the level-based API of the RITAM Orchestrator. Internally it is decomposed into Component CRC Models.
3. The Component Controller calls the reconcile operation of each individual component.
4. The reconciliation operation invocations create, configure, and start the components.
5. The Component and Application CRC Models of the application are updated with the current instance information.
6. The Actual State Information is propagated through the RITAM Device and Application Manager to the operator.
7. The operator can update the desired state in the IaC definitions and apply the changes using the level-based API.
8. The reconciliation is a continuous process, which repeats until the CRC Models are deleted.

6.2 RITAM TOSCA Extension

In this section, an extension to the TOSCA Simple Profile in YAML Version 1.3 [OAS20] standard is defined. The RITAM TOSCA Extension defines how the Reconciler Pattern can be used to model management operations for TOSCA applications. The extension also defines a runtime model and semantics for deployed TOSCA Topology Templates.

The runtime model is based on the three TOSCA Entities: Node Template, Relationship Template, and Topology Template. Other TOSCA entities, such as TOSCA Artifacts and TOSCA Types, have no corresponding runtime model, because they represent static information. The purpose of the runtime models is to hold dynamic and static runtime information about instances of TOSCA Node Templates, Relationship Templates, and Topology Templates. Instances of TOSCA Node Templates are described by Component CRC Models. The term “component” is used instead of “node”, which is used in TOSCA, to keep it consistent with the rest of this work. Instances of TOSCA Relationship Templates are modeled as part of the Component CRC Models, because they are closely related and relationships can not exist without the components they connect. A Component CRC Model therefore contains all information about its outgoing relationships. Instances of TOSCA Topology Templates are described by Application CRC Models.

The RITAM TOSCA Extension introduces a new TOSCA Interface which is described in Section 6.2.1. It also puts additional constraints on some parts of the TOSCA specification, see Section 6.2.2. In Section 6.2.3 the limitations of the RITAM TOSCA Extension are described.

6.2.1 Reconciler Interface

To enable the use of the Reconciler Pattern in TOSCA, the RITAM TOSCA Extension defines the new *Reconciler lifecycle interface*, which can be implemented by TOSCA Nodes. The Reconciler lifecycle interface is a TOSCA Interface Type with two operations: `reconcile` and `delete`. Both represent stateless reconciler functions, while `reconcile` creates, installs, starts, updates, and reconciles the component, `delete` does the opposite and stops, deletes, destroys, and cleans up the component. In contrast to the TOSCA Standard lifecycle interface, which are only triggered by predefined events (create, configure, start, stop, delete) and thereby limiting the customizability [BKLW17], the Reconciler lifecycle interface is triggered by any change to the desired state.

As a result, it removes the implicit state transitions between the TOSCA Standard lifecycle operations and enables a customized state handling in the `reconcile` operation. Without the restriction of predefined management steps, the Reconciler lifecycle interface is more flexible and enables new use cases. Most importantly, it enables to execute and re-execute management tasks based on the actual state of the component. It also allows to adapt the execution of management task to changes of the desired state made by an operator. Scheduled management tasks, such as automatic backups and health probes, may be implemented by making use of the scheduled re-execution of the reconciler.

The Reconciler lifecycle interface replaces the TOSCA Standard lifecycle interface for TOSCA Nodes. This means that TOSCA Nodes implement their own reconciler functions or inherit them from a TOSCA Node Type. In contrast to the TOSCA Standard lifecycle interface, the operations of the Reconciler lifecycle interface are level-based or idempotent and are executed multiple times during the lifecycle of a component. The RITAM Orchestrator implements the controller and executes the reconciler operations. The operations of the Reconciler lifecycle interface are executed locally on the host of the RITAM Orchestrator. Nevertheless, for RITAM they are executed on the IoT devices, because the RITAM Orchestrator runs locally on the IoT devices.

Reconciler Input and Output Mapping

The operations of the Reconciler lifecycle interface use TOSCA input and output definitions [OAS20, s. 3.6.17]. The desired state of a component is defined, amongst other things, by the properties of the TOSCA Node. For example, properties can define the version number of the software component, a network port, a user name, or a URL of a remote service. The reconcile operation can access the value of properties by defining an operation input parameter with the `get_property` function [OAS20, s. 4.4.2].

Properties are part of the Desired State Information and can only be changed by an operator of the application. The reconciler operation has read-only access to the desired state. On the other hand, the reconcile operation may need access to runtime information of another components. For example, the IP address, a callback URL, or the status of a component. This information is only available at runtime and represent Actual State Information. In TOSCA it is described by attributes, which represent the runtime information of a component. Like properties, attributes can be accessed by using the `get_attribute` function [OAS20, s. 4.5.1].

Attributes expose the actual state of a component and can change over the lifetime of a component. For example, the number of running instances or the timestamp of the last backup. A reconcile operation updates the attributes of the component with the Actual State Information it has observed. In TOSCA, this is done using an Attribute Mapping [OAS20, s. 3.6.15], where the output of the reconcile operation is mapped to the attributes of the TOSCA Node. Note that a reconcile operation can not read its own attributes, a reconciler must always observe the actual state of the component it reconciles. Only the attributes of other components, it has a direct relationship to, can be read by the reconcile operation. For example, a reconciler can read the attributes of a component it is hosted on, or it can read the attributes of a component it is connected to.

Listing 6.1 shows the reconciler interface definition for a TOSCA Node Type used in the motivating scenario IoT application. It defines the inputs and outputs of the `reconcile` and `delete` operation. The inputs include properties of the TOSCA Node, e. g., `interval` and `location`, and deployment artifacts such as `index.mjs`. The complete Reconciler Interface Type definition and TOSCA Service Template can be found in the Listing A.1 and Listing A.2.

To conclude, a reconcile operation can read the desired state defined by properties and propagate the actual state using attributes. The RITAM TOSCA Extension enforces a strict separation of Desired and Actual State Information.

Reconcile Operation

The `reconcile` operation of the Reconciler Interface Type should implement the reconciliation logic of the component. The Component Controller invokes the `reconcile` operation continuously, with the defined input parameters. By default, the invocation is triggered every few minutes, or when some input properties or attributes of the operation have been changed.

```
# rest omitted for brevity
interfaces:
  Reconciler:
    type: ritam:ritam.interfaces.Reconciler
    inputs:
      interval:
        type: string
        default: { get_property: [SELF, interval] }
      location:
        type: string
        default: { get_property: [SELF, location] }
      MQTT_URL:
        type: string
      DA_INDEX:
        type: string
        default: { get_artifact: [SELF, index.mjs]}
      # rest omitted for brevity
    operations:
      reconcile:
        implementation:
          primary: scripts/reconcileSensor.mjs
          dependencies:
            # omitted for brevity
          timeout: 15
        outputs:
          status: [SELF, status]
      delete:
        implementation:
          primary: scripts/deleteSensor.mjs
          dependencies:
            # omitted for brevity
          timeout: 10
        outputs:
          status: [SELF, status]
          deleted: [SELF, deleted]
```

Listing 6.1: Reconciler interface definition of a TOSCA Node Type with `reconcile` and `delete` operation

The order of invocation of the `reconcile` operations is not constrained by the topology. The invocations happen multiple times, in any order, and may run concurrently. As a result, a `reconcile` operation of a software component can be invoked before or during the `reconcile` operation of the component it is hosted on is invoked. The Reconciler Pattern does not guarantee any strict consistency, therefore the `reconcile` operations must tolerate inconsistent views of attribute values. Also, they must tolerate temporary failures of itself, other components, and the controllers. All these requirements are satisfied if the *reconcile* operation is level-based, because then the controllers may reinvoke the `reconcile` operation after failures.

Listing 6.1 shows the definition of a `reconcile` operation of a TOSCA Node Type. The `implementation` property defines the artifact which is invoked by the Component Reconciler. The artifact contains the custom reconciliation logic for the component defined by the TOSCA Node Type.

Deletion Operation

The `delete` operation of the Reconciler Interface Type is the inverse to the `reconcile` operation. It is invoked to delete a component. The operation is also level-based and invoked multiple times until the component is finally deleted, which is indicated by the `deleted` output parameter. The `delete` operation is a clean up operation, it should undo any management changes done by the `reconcile` operation.

The RITAM Orchestrator guarantees that the `delete` operation is invoked again after failures. When the `delete` operation returns `true` for the `deleted` output parameter, it must have cleaned up all resources. After that, the Component CRC Model is garbage collected by the level-based API.

6.2.2 Additional Constraints

In addition to the new Reconciler Interface Type, the RITAM TOSCA Extension also defines constraints on the existing TOSCA standard. The TOSCA standard [OAS20] defines in 3.6.10.1 and 3.6.12.1 that properties should automatically be reflected by attributes with the same name. The RITAM TOSCA Extension removes this behavior, so that attributes are not implicitly reflected from properties. Attributes must either explicitly define a reference to the value of a property or must be set by an output mapping from an operation, see Section 6.2.1.

Topology constraints

The RITAM TOSCA Extension restricts TOSCA topologies to have no loops. References between components are only allowed if there is a relationship between these components. These constraints are required to manage components individually. With only local dependencies and dependencies to adjacent nodes of the topology, the management of a component becomes a local problem, that can be solved without knowing the state of the whole application. It also enables operators to make local changes to the desired state of the application without redeploying the whole application.

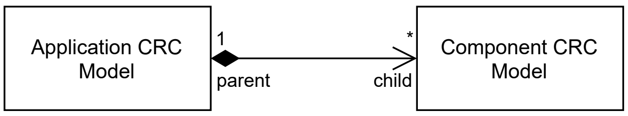

Figure 6.2: Domain model of the RITAM TOSCA Extension CRC Model Types

These constraints improve the TOSCA standard, which currently does not take the property and attribute references into account when ordering the management operations. This can lead to undefined behavior when a operation reads an attribute before it is set by another operation which is executed later, because the dependency between these are not expressed in the model.

Consistency Constraints

The RITAM TOSCA Extension does not guarantee strict consistency, ordering, or consistent views. The consistency model of the Controller and Reconciler Pattern is eventual consistency. This means there is no central management concept, such as workflows, that strictly manages the order and state transitions of all components. Instead, each component manages itself and makes only loose assumptions about the state of other components. It is very common for a reconcile function to fail and retry later if it encounters an intermediate or inconsistent state.

6.2.3 Limitations

While the RITAM TOSCA Extension brings new functional and non-functional properties to the TOSCA orchestration process and lifecycle handling of TOSCA Nodes, it also has its limitations. When modeling a TOSCA topology, the Reconciler lifecycle interface and the Standard lifecycle interface can not be used at the same time. The whole topology should either support the Standard lifecycle interface or Reconciler lifecycle interface. Mixing the two concepts in one topology is not possible because one requires strict consistency and the other only provides eventual consistency.

With the RITAM TOSCA Extension, TOSCA Relationships can not have a configuration interface, the configuration behavior must be part of the reconciler operations of the components. This is to ensure that only one reconciler is responsible for reconciling the state of the component. The relationships in the topology are only used to declare dependencies, so TOSCA intrinsic functions can be used to access properties and attributes of connected components.

6.3 TOSCA Runtime Models

In this section the runtime models of the RITAM Orchestrator are described. The CRC Models are used, because they combine the desires state with the instance information. Figure 6.2 shows the Application and Component CRC Model Types. An Application CRC Model functions as a parent for several Component CRC Models. The decomposition into child Component CRC Models is managed by the Application Controller, which implements the Operator Pattern.

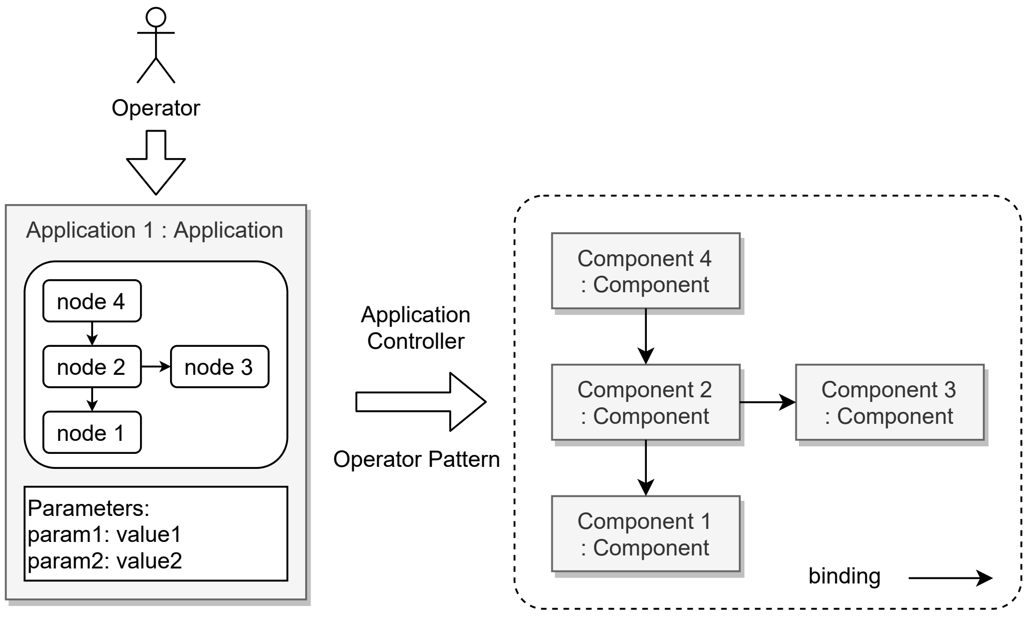

Figure 6.3: Example Application CRC Model with its child Component CRC Models

The Operator Pattern enables the deployment and management of TOSCA topologies created from TOSCA Topology Templates. Here these topology instances are called applications, they define the TOSCA Topology Template and input parameters used to instantiate the template. The Application CRC Model also aggregates the Actual State Information of all TOSCA Nodes of the instantiated template.

Figure 6.3 shows an example of model instances at runtime. On the left the Application CRC Model of Application 1 is shown, which defines a TOSCA Topology Template and input values for its parameters, created by an operator. On the right Component CRC Models are shown, which were created by the Application Controller. Each TOSCA Node Template is instantiated into a Component CRC Model.

In the following sections the CRC Model Types of the RITAM TOSCA Extension are described in detail.

6.3.1 Metadata

As discussed in Section 3.6.1, CRC Models also have metadata attached to them, which allows the storage and controllers to manage the models. The most important metadata of a CRC Model is its name in the storage, it is used to reference the model in other CRC Models. The RITAM TOSCA Extension uses deterministic unique identifier as names for CRC Models, see Section 3.2.1. The identifier for Component CRC Models are generated using the name of the Application CRC Model as prefix. The Component CRC Models are also labeled so a human operator can easily identify the application they belong to.

```yaml
metadata:
  name: temperature-app
  labels:
    creator: application-template-controller
  finalizers:
  - application-controller
  deletion: false
  resourceVersion: 79397dba-4202-47b5-9c04-fec767dd012c
spec:
  serviceTemplate: file:///resources/temperature-sensor-service-template.yaml
  inputs:
    MQTT_URL: mqtt://broker
    interval: 10s
    location: office
status:
  message: Successfully Reconciled
  lastReconciled: "2021-04-13T11:36:33.218153Z"
  nodes:
    temp-reader: Successfully reconciled
    device: Successfully reconciled
    broker: Successfully reconciled
  outputs: {}
```

Listing 6.2: Example Application CRC Model of the motivating scenario IoT application in YAML format

6.3.2 Application CRC Model

An Application CRC Model is the single interaction point between an operator and the RITAM Orchestrator. To deploy an application, an operator creates an Application CRC Model and specifies a TOSCA Topology Template and values for its input parameters. With RITAM the deployment and management of the applications on the individual devices is automated by the RITAM Device and Application Manager. This information is stored as desired state in a newly created Application CRC Model.

Each Application CRC Model represents a self-contained and isolated instance of a TOSCA Topology Template. The Application CRC Model Desired State Information contains the topology, which defines the desired state of the components and relationships. The Actual State Information aggregates the status of all components. It also contains the output parameter of the TOSCA Topology Template [OAS20, s. 3.9.2.4].

Listing 6.2 shows an example of such an Application CRC Model. The shown Application CRC Model is in the form how an operator would retrieve it from the level-based API. The `status` attribute indicates that the reconciliation of the application and all components was successful.

6.3.3 Component CRC Model

The Component CRC Model represents an instance of a single component of an application. It is a child of the Application CRC Model and created by the Application Controller. The model is self-contained and defines its outgoing relationships, which are used by the Component Controller to get attribute values at runtime.

On the right in Figure 6.3 a topology of four Component CRC Models is shown. A component *A* is called a *dependency* of a component *B* if there is a relationship where *A* is the target and *B* the source. Component *B* is then called *dependent* of component *A*. In Figure 6.3, *Component 1* and *Component 3* are dependencies of *Component 2*. *Component 4* is a dependent of *Component 2*. A *binding* is a deletion lock established at runtime between a dependent and dependency. It guarantees that the dependency is not deleted before the dependent. A binding is a similar concept to finalizers, but instead of a reconciler, another Component CRC Model owns the deletion lock.

Listing 6.3 shows an example of a Component CRC Model from the level-based API. In the `metadata` the `parentRef` attribute defines the name of the parent Application CRC Model. The Desired State Information contains only information of that single component. Including the reconciler operations definition with their implementation artifacts and inputs. Only outgoing relationships of that component are included in the Desired State Information, as part of the `dependencies` property. Incoming relationships are added at runtime by the source component of the relationship and are stored in the `bindings` property. The `temperature-app-temp-reader` Component CRC Model defined in Listing 6.3, has two outgoing and no incoming relationships.

The Actual State Information contains the attribute values set by the reconciler of that component. The attributes itself are defined in the desired state. In the given example the `reconcile` operation defines the `status` output parameter, which is mapped to the `status` attribute of the component. The Actual State Information also has a `message` property which is set by the Component Reconciler and describes the reconciliation status.

6.4 RITAM Orchestrator

The RITAM Orchestrator is comprised of a level-based API, the Application Controller, and the Component Controller. Figure 6.4 shows the architecture of the RITAM Orchestrator. The level-based API of the orchestrator consists of a REST API and two Databases for the CRC Models. Operators and other systems use the REST API to interact with the orchestrator. The Application Controller and Component Controller interact with the level-based API internally. The Component Controller invokes the Reconciler lifecycle interface of the TOSCA Nodes, which then interacts with external management systems to bring the components to the desired state.

In the following sections the behavior of the Application and Component Controller is described. Section 6.4.1 discusses the role of the Application Controller in the RITAM Orchestrator. The Component Controller and how it invokes the reconciler operations of the TOSCA Nodes is described in Section 6.4.2.

```yaml
metadata:
  name: temperature-app-temp-reader
  labels:
    application: temperature-app
    component: temp-reader
    creator: application-controller
  finalizers:
  - component-controller
  deletion: false
  parentRef: temperature-app
  resourceVersion: efabfa38-7c82-4364-bdfd-5f26f93f2a33
spec:
  reconciler:
    reconcile:
      artifact:
        type: https://legion2.github.io/ritam/ritam.artifacts.Implementation.
JavaScript
        file: file:/resources/scripts/reconcileSensor.mjs
      inputs:
        interval: 10s
        location: office
        MQTT_URL: mqtt://broker
        # rest omitted for brevity
      outputs:
        status: status
      timeout: 15.0
      dependencies:
        # omitted for brevity
    deletion:
      # omitted for brevity
  dependencies:
  - temperature-app-device
  - temperature-app-broker
  bindings: {}
  operationFinalizer: true
status:
  message: Successfully reconciled
  lastReconciled: "2021-04-13T15:40:03.739614Z"
  attributes:
    status:
      value: updated - online
```

Listing 6.3: Example Component CRC Model of the *temp-reader* component of the motivating scenario IoT application in YAML format

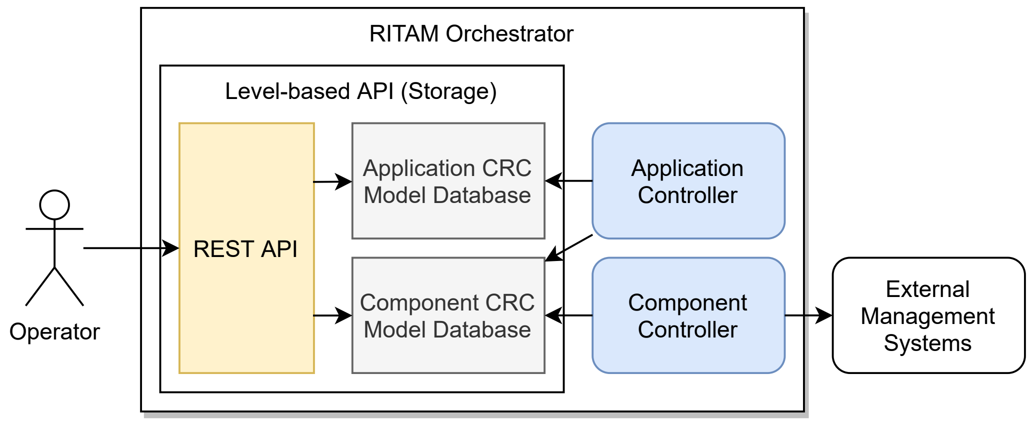
RITAM Orchestrator
Level-based API (Storage)
REST API
Application CRC Model Database
Component CRC Model Database
Application Controller
Component Controller
External Management Systems
Operator

Figure 6.4: Architecture of the RITAM Orchestrator

6.4.1 Application Controller

The Application Controller decomposes Application CRC Models into self-containing Component CRC Models. The Desired State Information of an Application CRC Model contains a TOSCA Topology Template, which contains all the nodes which should be instantiated. However, extracting a single TOSCA Node Template from a TOSCA Topology Template, does not provide enough information to orchestrate the TOSCA Node. Therefore, the Application Controller will also collect all outgoing relationships of that Node and include them in the Component CRC Model of that Node.

The Application Controller maps between TOSCA Node Templates and Component CRC Models, by using the name of the TOSCA Node prefixed with the name of the Application CRC Model. To make the Component CRC Model self-contained, the Application Controller will evaluate intrinsic function expressions in properties and input parameters of the TOSCA Nodes. Expressions that contain the `get_attribute` function can not be evaluated by the Application Controller, because the value depends on the actual state of the components. These dynamic expressions will be evaluated by the Component Controller on-demand.

Besides the initial creation of all Component CRC Models based on the TOSCA Topology Template, the Application Controller also handles modifications, updates, and inconsistencies. This means if an operator updates the Application CRC Model, e. g., changes an input parameter value, the Application Controller updates all Component CRC Models which are affected by this change. If new TOSCA Nodes are added or removed from the topology, the Application Controller will create or delete the respective Component CRC Models. When an operator decommissions an application, the Application Controller will delete all related Component CRC Models.

The Application CRC Model reconciliation process does not only enforce the desired state, but also reports the Actual State Information of all components back to to operator. Therefore it aggregates the status of all Component CRC Models and set the status of the Application CRC Model.

The Application Controller does not perform any application or component specific management operations. This is the task of the Component Controller and the Reconciler lifecycle interface operations, described in the next section.

6.4.2 Component Controller

The Component Controller is responsible for reconciling individual Component CRC Models. The reconciliation process of a component requires component specific logic and management operations. The RITAM Orchestrator does not implement this component or application specific orchestration logic. Instead, the components themselves provide the custom logic using TOSCA Node Reconciler lifecycle interface operations. The Component Reconciler, which is part of the RITAM Orchestrator, just invokes the custom reconciler logic of the individual components.

However, before the reconciler of a TOSCA Node is invoked, the Component Reconciler has to ensure that all dependencies of the component are available. The dependencies of a component are the outgoing relationships. The Component Reconciler performs a two phase binding for all dependencies. The bindings are deletion locks to prevent the deletion of a component while it is being used by another component as a dependency. To establish a binding, the dependent component requests a binding. During the reconciliation of the dependency component the binding is accepted or ignored if the component is currently being deleted. The Component Reconciler guarantees that a component will not be deleted while it has accepted bindings.

In Figure 6.5, the behavior of the Component Reconciler is shown as activity diagram, which is a refined version of Figure 3.4. On the right side of the activity diagram, the reconciliation steps are modeled. The dashed boxes mark the activities calling the TOSCA Node `reconcile` and `delete` operations. Before the `reconcile` operation of the TOSCA Node is executed, the following steps are performed by the Component Reconciler:

1. The garbage collection finalizer is added to the Component CRC Model.
2. For all dependencies bindings are requested.
3. Wait for all bindings to be accepted.
4. All binding requests of dependent components are accepted.
5. The reconcile operation cleanup finalizer is set in the Component CRC Model.

After these steps have been performed, the Component Reconciler invokes the custom reconcile operation of the TOSCA Node.

When the `reconcile` or `delete` operation of the TOSCA Node is invoked, the input parameters are evaluated. The input parameter can be defined as expressions containing the `get_attribute` function, which can reference attribute values of other components at runtime. The references in the `get_attribute` function are limited to the references of the component dependencies. After the invocation the output parameter of the `reconcile` or `delete` operation are mapped to attribute values of the component, see Section 6.2.1. The last step of the Component Reconciler is to update the Component CRC Model with the new Actual State Information, in the form of the component's attribute values.

The Component Controller periodically calls the Component Reconciler, e. g., every five minutes. It also triggers a reconciliation for changes to the desired state of the component. To reduce the reconciliation delay, the Component Controller also watches for changes of the component dependencies and triggers a reconciliation if their attributes change. The reconciliation process of the components are independent of each other and changes are only eventually observed by other components.

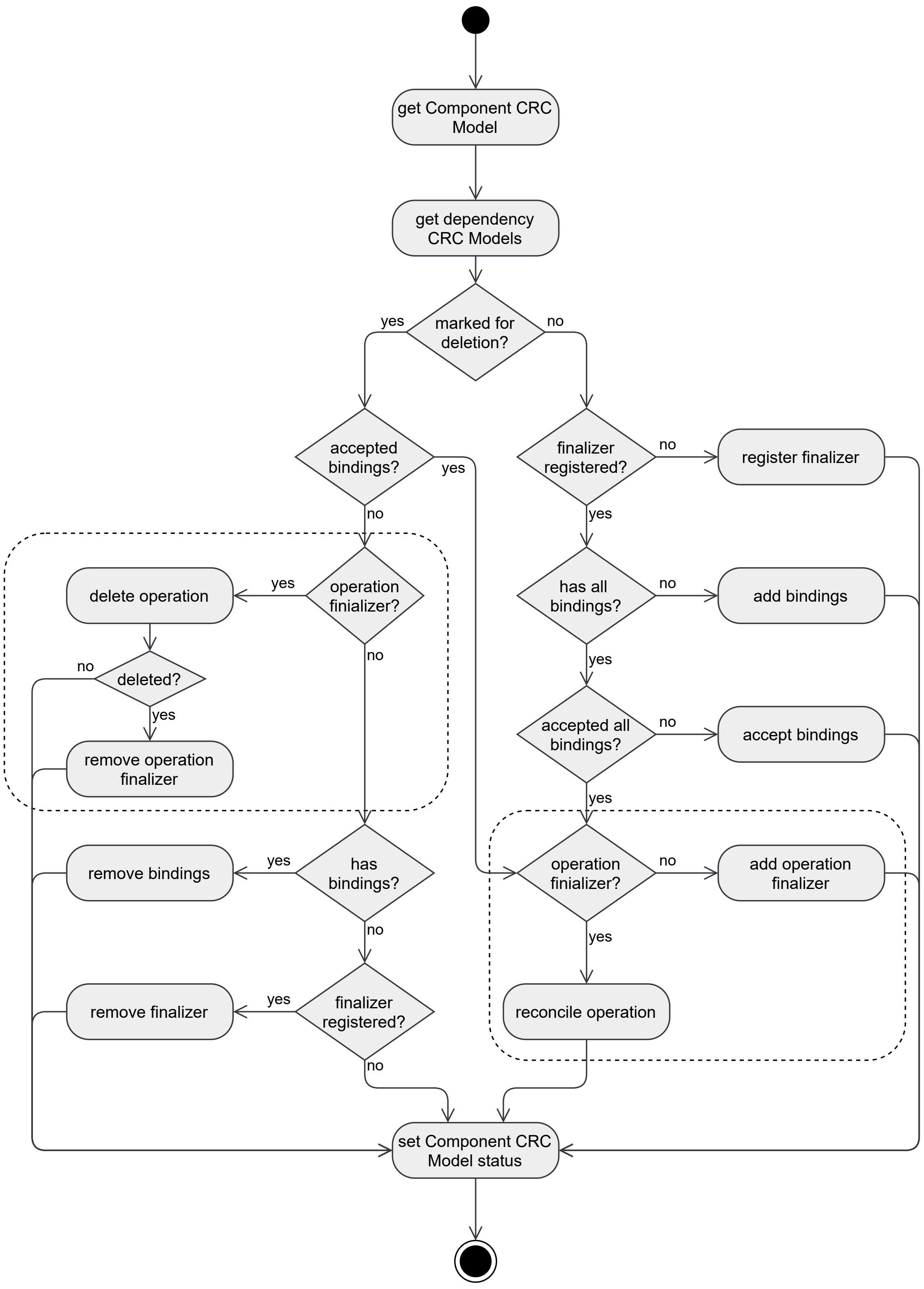

Figure 6.5: Component Reconciler activity diagram

When a TOSCA Node is removed from an application, the Component CRC Model will be marked for deletion. However, the Component Reconciler will not delete the component while other components have an accepted binding to it. As shown by the connection from the deletion process (left) to the reconciliation process (right) in Figure 6.5. The Component Reconciler will continue to call the reconcile operation when there are accepted bindings. When all components of an application topology should be deleted, there is always at lease one component with no incoming relationships which can be deleted, because the topology has no cycles, and eventually all components are deleted.

Only if all dependent components have been deleted, the Component Reconciler will invoke the `delete` operation of the TOSCA Node. The `delete` operation must ensure to stop, delete, destroy, and cleanup all resources and components created by the `reconcile` operation. The `delete` operation uses the `deleted` output parameter to inform the Component Reconciler about the deletion status. Its value must be `true`, if everything was cleaned up, else `false`. Then the Component Reconciler will remove the operation cleanup finalizer. Finally, all requested bindings are removed and the garbage collection finalizer is removed. Eventually the storage garbage collection will remove the Component CRC Model.

7 Evaluation

In this chapter the RITAM approach is evaluated using a prototypical implementation. In Section 7.1, the implementation of the prototype is described. The concepts are evaluated based on the motivating scenario from the introduction in Section 7.2. Finally, the limitations of the prototype are summarized in Section 7.3.

7.1 Implementation of the Prototype

The following sections describe the prototype implementation of the different building blocks of the RITAM approach. This includes important considerations, design decisions, technologies, and concrete implementation details of the prototype. The prototype is open source and available on GitHub (`https://github.com/Legion2/ritam`). It is implemented in Kotlin[1] a modern programming language, which can be executed on the Java virtual machine (JVM) and is interoperable with Java. This allows the usage of the Java ecosystem within the Kotlin programming language. Additionally, Kotlin provides null safety, immutable data classes, concise syntax, and multiplatform support.

The prototype consists of the RITAM Device and Application Manager and the RITAM Orchestrator. Both use the Controller and Reconciler Pattern which is implemented as a framework to support its reusability in both components. In the following, first the controller and reconciler framework is described in Section 7.1.1. Then in Section 7.1.2, the RITAM Device and Application Manager prototype is explained. Finally, the RITAM Orchestrator is presented in Section 7.1.3.

7.1.1 Controller and Reconciler Framework

The prototype has multiple controllers and reconcilers. To enable easy development, a framework for developing controllers and reconcilers was created. It provides abstract base implementations and utilities to write the reconciler logic and create controllers. The framework also allows to define the CRC Model data objects and generates the level-based API for them.

The framework provides a generic controller implementation, which can be instantiated and configured for different reconcilers. For the instantiated controller, watches can be created to trigger the reconciliation of CRC Models based on notifications. For each watch the sync period and a filter function can be configured. Internally, the controller uses the work queue implementation of the Kubernetes extended java client library[2].

[1]https://kotlinlang.org/

[2]https://github.com/kubernetes-client/java/

To create a reconciler with the framework, a function which takes a `Request` as input and returns a `Result` as output must be created. The `Request` contains the name of the CRC Model which is reconciled and the returned `Result` informs the controller if the reconciliation was successful or failed. Failed reconciliations are automatically requeued by the controller, so that the reconciler can retry the reconciliation. The `Result` also allows to set a duration after which the `Request` should be requeued. When an exception is thrown by the reconciler function, the controller catches it and requeues the `Request` object automatically.

The framework also provides a generic storage implementation. The storage is implemented as an in-memory database with a REST API, which supports JSON and YAML formats and also provides an OpenAPI Document [Lin20] to generate client code. The storage prototype is idempotent and all operations are atomic. For the controller and reconciler the storage prototype also provides a Kotlin interface which provides the same functionality as the REST API. Using the interface is more convenient and faster than making a HTTP request, because in the prototype all components are running in the same process.

7.1.2 RITAM Device and Application Manager Prototype

The RITAM Device and Application Manager prototype uses the controller and reconciler framework. It defines the Device and Application Template CRC Models, which can be created, updated, and deleted using the `ritam` CLI. The CLI allows operators to manage CRC Models using IaC. The Application Template Controller evaluates, based on labels, on which devices an application template should be deployed and then sends a POST or PUT request to the respective RITAM Orchestrator. When an application template is marked for deletion, it sends a DELETE request to the RITAM Orchestrator. The URL of the RITAM Orchestrator is stored in the Device CRC Models.

7.1.3 RITAM Orchestrator Prototype

The RITAM Orchestrator prototype implements a TOSCA parser which is compatible with the TOSCA YAML 1.3 standard [OAS20], excluding parts related to workflows, which are not used in the RITAM approach. The TOSCA metamodel is defined using Kotlin data classes, to ensure type safety and correct processing of the TOSCA Service Templates in the orchestrator. The orchestrator can execute Bash and Javascript[3] implementation artifacts and allows to add new artifact processors using a plugin system.

Via the REST API, Application CRC Models can be created, updated, and deleted. The Application CRC Model contains a URI of a TOSCA Service Template, which is parsed and split into Component CRC Models during reconciliation. The Component Reconciler implements the logic as shown in Figure 6.5 and calls the respective reconcile and delete operations of the Component CRC Models. The `reconcile` and `delete` operations are specified in the Component CRC Models as implementation artifacts which can be executed by one of the artifact processors of the orchestrator.

[3]https://nodejs.org/

7.2 Orchestration Scenarios

In this section different use case scenarios of the RITAM approach are presented. For each scenario the interactions and activities of the different components are explained. Four scenarios are presented:

1. Orchestration of a new IoT application
2. Modification of a deployed IoT application
3. Decommission of a deployed IoT application
4. Monitoring of the deployed IoT application

All scenarios use the motivating scenario IoT application as example application.

7.2.1 Orchestration of a new Application

A common use case is to deploy and orchestrate new IoT applications. In the following the motivating scenario IoT application should be deployed. Because an existing MQTT broker is used only the Temperature Reader software component has to be deployed on the edge device which has the temperature sensor hardware.

First, the operator models the application and uses the RITAM TOSCA Extension, because a resilient, robust, and durable management of the application is required on the edge device. The operator creates a reusable TOSCA Service Template, which can be deployed with different parameters on different devices. The modeled application is shown on the left in Figure 7.1. The TOSCA Service Template consists of three TOSCA Nodes. The *device* node represents the edge devices with the temperature sensor. The *temp-reader* node is the software component which reads and sends the temperature values to the MQTT broker. The running MQTT broker, where the temperature values are sent to, is represented by the *broker* node. A custom reconciler is created and attached to the *temp-reader* node together with the deployment artifact.

The next step is to create the application template which references the created TOSCA Service Template and specifies a device selector. As shown in Figure 7.1 there are already two devices with the label `type: raspberry-pi`. The operator uses this label in the label selector to deploy onto both devices. After the operator has created the application template in the RITAM Device and Application Manager, the Application Template Controller reconciliation processes deploys the temperature IoT application to all selected devices. Therefore, it uses the URL stored in the Device CRC Models to send the rendered application template to the RITAM Orchestrators running on the devices.

On the devices, the Application Controller starts the reconciliation of the new Application CRC Model and imports the service template of the temperature IoT application. For each TOSCA Node Template it creates a Component CRC Model, which is then reconciled by the Component Controller. Only the *temp-reader* node implements the Reconciler lifecycle interface. The *temp-reader* reconcile operation installs, configures, and starts the deployment artifact.

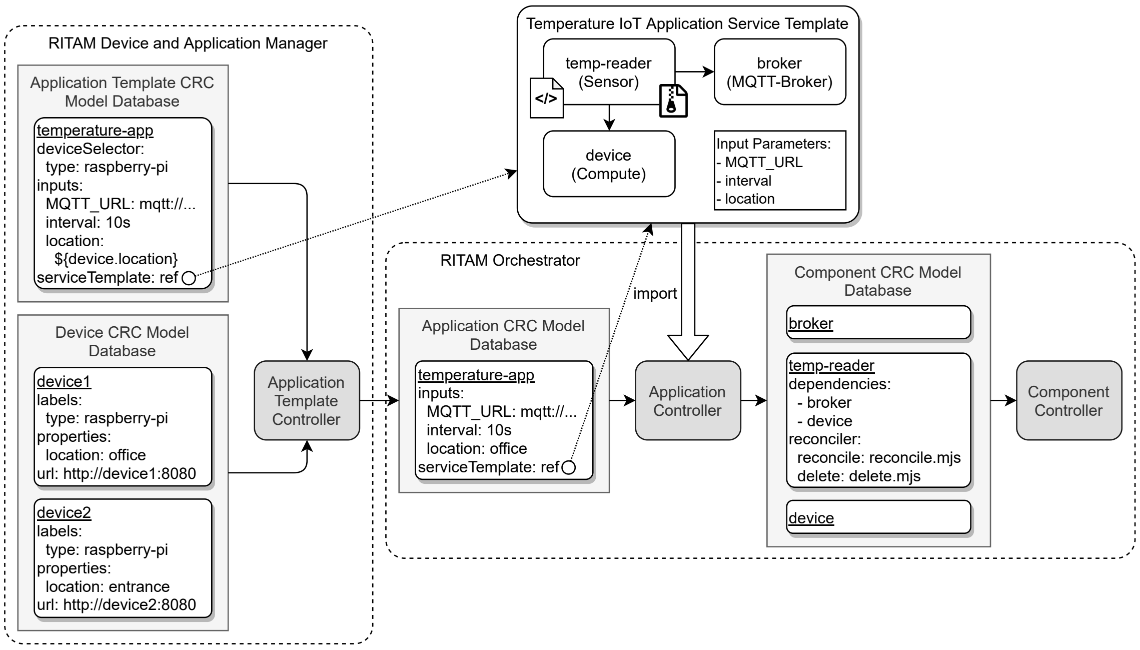

Figure 7.1: Deployment process of motivating scenario IoT application

If there is a failure at any point in the deployment process, e. g., a network timeout or a power outage, the reconciler is executed again and recovers by observing the actual state. Because there are no dependencies between application instances, the deployment process is executed concurrently on all devices. The periodic reconciliation guarantees that eventually the IoT application is deployed on all selected devices.

7.2.2 Modification of a deployed application

Another common scenario is the modification of an already deployed application. The scenario can be split into two sub scenarios. One for changing the configuration of an existing component in the application and another for adding a new component to the topology of the existing application.

For the first scenario imagine the temperature measurement interval should be changed from $10s$ to $30s$. To do this, the operator updates the `interval` property of the application template to the new value in the IaC repository. Then he applies the updated application template to the level-based API. The Application Template Controller will then apply these changes to all devices.

On the devices, the Application Controller detects that the *temp-reader* Component CRC Model does not match the desired state and updates it with the new polling interval. Then the Component Reconciler calls the `reconcile` operation with the new configuration as input. The `reconcile` operation checks if the deployment artifact is installed and if the configuration is correct. It detects that the configuration is not correct and stops the running *temp-reader*. After the `reconcile` operations has updated the configuration it restarts the *temp-reader*.

Another scenario is when a new component should be added to the IoT application. For example, the humidity value should be measured as well and sent to the MQTT broker. The humidity reader is implemented in a separate deployment artifact and modeled by the operator as a new TOSCA Node which is also hosted on the *device* and connected to the *broker*. The operator publishes the new TOSCA Service Template and updates the reference in the application template. The Application Template Controller will then apply this change to all devices.

On the device, the Application Controller imports the new service templates and detects that the Component CRC Model for the new TOSCA Node is missing and creates it. The Component Reconciler will then handle the reconciliation of the new *humidity-reader* component. Eventually, the new component is in the desired state and the humidity values are sent to the broker as well.

7.2.3 Decommission of a deployed application

Applications are decommissioned at the end of their lifecycle. All components of the application have to be stopped, uninstalled, deleted, and cleaned up. For the motivating scenario IoT application this means, the software package has to be uninstalled from the IoT devices. The MQTT broker itself should not be uninstalled or stopped, because it is not managed by RITAM.

To decommission the IoT application, the operator deletes the application template in the IaC repository and applies this change also to the RITAM Device and Application Manager. The level-based API marks the Application Template CRC Model for deletion. The Application Template Controller will then also mark the application on all selected devices for deletion.

On the device, the Application Controller checks if an Application CRC Model is marked for deletion. If so, it identifies all created Component CRC Models and also marks them for deletion. The Component Reconciler will then start to delete the components with no incoming relationships. As a result, first the *temp-reader* component will be deleted.

The Component Reconciler invokes the `delete` operation to delete the component. After successful deletion the Component Reconciler removes the finalizer from the Component CRC Model, so it is garbage collected. The Application Controller waits until the all child Component CRC Models are deleted from the storage and then removes the garbage collection finalizer from the Application CRC Model. The Application CRC Model is eventually garbage collected.

After the application was removed from all devices, the application template is garbage collected as well. If there are any failures during decommissioning, the reconcilers will retry until everything is cleaned up. As a result, RITAM guarantees that the IoT application is removed from all devices without leaving any orphan resources in the environment.

7.2.4 Monitoring of the components of an application

When tasks are automated, operators have to rely on the correct execution of these task. However, automated tasks are created by humans and therefore may contain errors. To detect failures in automated and asynchronous systems, they have to be monitored.

In case of the motivating scenario IoT application, the operator wants to know if the application is successfully deployed on all selected devices or if there are failures during reconciliation. For example, the operator specified the wrong URL for the MQTT broker and therefore the *temp-reader* component can not connect to the broker. Because the failure happens during the asynchronous reconciliation process on the IoT devices, the operator can not be informed directly about the problem. Instead the failure is propagated in the Actual State Information of the Component CRC Model.

The Application CRC Model contains the aggregated status of all its components. The Application Template Controller also aggregates the status of the applications from all devices. Therefore, to monitor an application and its component, the operator can use the level-based API of the RITAM Device and Application Manager. The REST API provides access to the eventually consistent Actual State Information.

If the Actual State Information of the application template is not enough for the operator, he can connect to the RITAM Orchestrator directly. The Component CRC Models contains more detailed information about the components. Most importantly they contain all attributes defined in the TOSCA Node, which are filled with the output values of the `reconcile` operation. As a result, the attributes can provide all kind of component specific runtime information to the operator. Because of the continuous reconciliation process the runtime information is kept up to date, so that also performance and usage monitoring information can be stored in the attributes.

7.3 Limitations of the Prototype

The prototype only implements an in-memory storage system and therefore does not provide durability. However, because of the loose coupling and the well-defined CRUD API of the storage, the implementation can be replaced with a persistent implementation. For communication between the RITAM Device and Application Manager and the RITAM Orchestrator the prototype only implements the push model via the level-based REST API. For the evaluation of the approach, security aspects such as authentication and authorization were not covered, but have to be considered for production use. The `ritam` CLI uses JSON Merge Patch[4] to apply the changes of the local CRC Models to the CRC Models stored in the storage. JSON Merge Patch has some limitations, it would be better to use a service-side apply mechanism. The Actual State Information exposed by the prototype are very basic and unstructured, and should be refined in future versions to provide more information to make it possible for other systems to parse the information. The prototype and scenarios were tested in a virtualized IoT environment using docker-compose[5], which allows to create a virtual network and multiple container instances to test and run distributed applications.

[4]https://tools.ietf.org/html/rfc7386

[5]https://docs.docker.com/compose/

8 Conclusion and Outlook

This work presented RITAM, a resilient management approach for IoT applications and edge infrastructures. The management components of the RITAM approach use the Controller and Reconciler Pattern which was also formalized in this work (Section 3.4). RITAM enables the usage of IaC and DevOps workflows for the management of IoT applications. The work also formalized the Operator Pattern to automate human tasks even further with adaptive model reuse, templating, and composition (Section 3.9).

The TOSCA modeling language is used to model the structure and components of IoT applications, as already proposed by [HBS+16; SBH+17; Sch18], with the difference, that only a single device topology is modeled. The devices and the distribution of applications onto the devices is modeled using a domain-specific modeling language. The RITAM approach combines the domain-specific modeling (Device CRC Model and Application Template CRC Model) and the general-purpose modeling (TOSCA) to provide domain-specific structures and abstractions while maintaining flexibility.

RITAM is driven by the level-based behavior and declarative modeling of the Controller and Reconciler Pattern. As a result, individual management operations such as installing and updating applications on the devices are hidden from the operators. The management system uses the desired state given by the operator and the actual state observed from the environment to decide on the next action. The declarative modeling of RITAM supports maintenance operations and general management operations, e. g., updating applications, adding components, and scheduling backups. The reconciliation process guarantees that unexpected changes to the actual state, such as failures, will be handled as part of the next reconciliation, resulting in a self-healing behavior.

RITAM reflects the essential characteristics of IoT devices, such as their physical environment and location. It allows the customization of application instances on the individual devices. RITAM also supports the growth of IoT applications and infrastructures, in the sense of elastic resources and adaptive tooling for the growing complexity of applications and infrastructures. The practical feasibility of the approach was shown by the evaluation of multiple IoT application management scenarios based on a prototypical implementation. Nevertheless, the RITAM approach is not limited to IoT applications, the concept of reconciliation-based management can be applied in many other domains.

Outlook

The benefits of reconciliation-based management should be investigated in domains other than the IoT. This work only discussed the minimal required set of features for a reconciliation-based management system. Instead of TOSCA as general-purpose modeling language, the use EDMM or TOSCA Light can be analysed in future works. The possibility to replace the general-purpose modeling language with a domain-specific one in certain domains should be considered as well.

The lack of literature about the Controller and Reconciler Pattern shows, that fundamental research is required in the field of reconciliation-based management. This work could be used as entry point in this field. Because of its loose coupling, flexibility, and modularity, reconciliation-based management can be applied incrementally to solve real world problems.

For the transition from workflow-based management to the reconciliation-based management, imperative workflows must be turned into reconciler logic. For basic workflows, which are a chain of actions, this transformation is trivial, but for more complex workflows the mapping also becomes more complex. In future works mappings, transformations, and automated generation of reconciler logic from workflows could be investigated.

Another interesting topic is the modeling of migration scenarios, where resources must be migrated from one platform to another. The RITAM approach requires the manual recreation of the CRC Models in migration scenarios. Concepts for automatic migrations of components without the recreation of the CRC Model could be analysed in future works.

Bibliography

[Ann19] B. Annis. *Scaling Ansible for IoT*. Oct. 12, 2019. URL: https://www.ansible.com/hubfs//AnsibleFest%20ATL%20Slide%20Decks/AnsibleFest%202019%20-%20Scaling%20Ansible%20for%20IoT%20Deployments.pdf (cit. on p. 26).

[AS18] F. Ahmadighohandizi, K. Systä. “Application Development and Deployment for IoT Devices”. In: *Communications in Computer and Information Science*. Springer International Publishing, 2018, pp. 74–85. DOI: 10.1007/978-3-319-72125-5_6 (cit. on pp. 13, 53).

[Bas15] L. Bass. *DevOps : a software architect’s perspective*. Old Tappan, NJ: Addison-Wesley, 2015. ISBN: 9780134049847 (cit. on p. 13).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. “Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA”. In: *On the Move to Meaningful Internet Systems: OTM 2012*. Springer Berlin Heidelberg, 2012, pp. 416–424. DOI: 10.1007/978-3-642-33606-5_25 (cit. on p. 21).

[BBK+14] U. Breitenbucher, T. Binz, K. Kepes, O. Kopp, F. Leymann, J. Wettinger. “Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA”. In: *2014 IEEE International Conference on Cloud Engineering*. IEEE, Mar. 2014. DOI: 10.1109/ic2e.2014.56 (cit. on pp. 18, 19, 22).

[BBKL13] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “TOSCA: Portable Automated Deployment and Management of Cloud Applications”. In: *Advanced Web Services*. Springer New York, Aug. 2013, pp. 527–549. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 23).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. “The OpenTOSCA Ecosystem - Concepts & Tools”. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*. SCITEPRESS - Science and Technology Publications, 2016. DOI: 10.5220/0007903201120130 (cit. on p. 24).

[BGO+16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes. “Borg, Omega, and Kubernetes”. In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444 (cit. on pp. 15, 34).

[BKH21] F. Beetz, A. Kammer, S. Harrer. “GitOps. Cloud-native Continuous Deployment”. Mar. 10, 2021. URL: https://www.gitops.tech/ (cit. on p. 25).

[BKLW17] U. Breitenbüher, K. á. Képes, F. Leymann, M. Wurster. “Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?” In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2017, pp. 18–27 (cit. on pp. 13, 67).

[Bow20] J. Bowes. *Level Triggering and Reconciliation in Kubernetes*. Mar. 20, 2020. URL: https://hackernoon.com/level-triggering-and-reconciliation-in-kubernetes-1f17fe30333d (cit. on p. 27).

[BR18] C. Barbour, J. Rhett. *Puppet Best Practices*. O'Reilly Media, Inc, USA, Sept. 7, 2018. 294 pp. ISBN: 1491923008. URL: https://www.ebook.de/de/product/23781875/chris_barbour_jo_rhett_puppet_best_practices.html (cit. on p. 58).

[Bre00] E. A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*. ACM Press, 2000. DOI: 10.1145/343477.343502 (cit. on p. 30).

[Bre12] E. Brewer. "CAP twelve years later: How the "rules"have changed". In: *Computer* 45.2 (Feb. 2012), pp. 23–29. DOI: 10.1109/mc.2012.37 (cit. on p. 31).

[Bre16] U. Breitenbücher. *Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements*. de. 2016. DOI: 10.18419/OPUS-8764 (cit. on pp. 17–22).

[BSK20] A. Brown, M. Stahnke, N. Kersten. *State of DevOps Report 2020*. Tech. rep. Puppet and CircleCI, Nov. 11, 2020. URL: https://puppet.com/resources/report/2020-state-of-devops-report/ (visited on 04/03/2021) (cit. on p. 13).

[DJ07] T. Delaet, W. Joosen. "PoDIM: A Language for High-Level Configuration Management." In: Jan. 2007, pp. 261–273 (cit. on p. 17).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27 (cit. on p. 18).

[FHH+10] A. Ferscha, C. Holzmann, M. Hechinger, B. Emsenhuber, S. Resmerita, S. Vogl, B. Wally. "Pervasive Computing". In: *Hagenberg Research*. Springer Berlin Heidelberg, 2010, pp. 379–431. DOI: 10.1007/978-3-642-02127-5_9 (cit. on p. 26).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. Springer Vienna, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on p. 17).

[FRS+13] N. Fantana, T. Riedel, J. Schlick, S. Ferber, J. Hupp, S. Miles, F. Michahelles, S. Svensson. "Internet of Things - Converging Technologies for Smart Environments and Integrated Ecosystems". In: Jan. 2013, pp. 153–204. ISBN: ISBN 978-87-92982-73-5 (print) ISBN 978-87-9282-96-4 (ebook) (cit. on p. 13).

[GHJV95] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides. *Design Patterns*. Prentice Hall, Dec. 1, 1995. ISBN: 0201633612. URL: https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html (cit. on p. 48).

[GN17] J. Garrison, K. Nova. *Cloud Native Infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment*. O'Reilly Media, 2017. Chap. Chapter 4: Designing Infrastructure Applications. ISBN: 9781491984253. URL: https://books.google.de/books?id=1Fk7DwAAQBAJ (cit. on pp. 20, 28, 34, 43).

[Guo19] F. Guo. *Learning Concurrent Reconciling*. Nov. 10, 2019. URL: https://openkruise.io/en-us/blog/blog2.html (cit. on p. 46).

[HBL+19] L. Harzenetter, U. Breitenbucher, F. Leymann, K. Saatkamp, B. Weder, M. Wurster. “Automated Generation of Management Workflows for Applications Based on Deployment Models”. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, Oct. 2019. DOI: 10.1109/edoc.2019.00034 (cit. on p. 22).

[HBS+16] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, M. Wieland. “Automating the Provisioning and Configuration of Devices in the Internet of Things”. In: *Complex Systems Informatics and Modeling Quarterly* 9 (Dec. 2016), pp. 28–43. DOI: 10.7250/csimq.2016-9.02 (cit. on pp. 25, 87).

[HF10] J. Humble, D. Farley. *Continuous Delivery*. Addison Wesley, July 1, 2010. 512 pp. ISBN: 0321601912. URL: https://www.ebook.de/de/product/9446498/jez_humble_david_farley_continuous_delivery.html (cit. on p. 13).

[HQ21] H. A. Hassan, R. P. Qasha. “Toward the generation of deployable distributed IoT system on the cloud”. In: *IOP Conference Series: Materials Science and Engineering* 1088.1 (Feb. 2021), p. 012078. DOI: 10.1088/1757-899x/1088/1/012078 (cit. on pp. 13, 26).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns*. Addison Wesley, Jan. 1, 2004. 480 pp. ISBN: 0321200683. URL: https://www.ebook.de/de/product/2779126/gregor_hohpe_bobby_woolf_enterprise_integration_patterns.html (cit. on p. 33).

[Jaz14] N. Jazdi. “Cyber physical systems in the context of Industry 4.0”. In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. IEEE, May 2014. DOI: 10.1109/aqtr.2014.6857843 (cit. on p. 13).

[Kno02] K. Knoernschild. *Java design : objects, UML, and process*. Boston, MA: Addison-Wesley, 2002. ISBN: 9780201750447 (cit. on p. 48).

[Kub20] Kubernetes SIGs. *What is a Controller*. 2020. URL: https://book-v1.book.kubebuilder.io/basics/what_is_a_controller.html (cit. on pp. 27, 34, 47).

[Ley00] F. Leymann. *Production workflow : concepts and techniques*. Upper Saddle River, N.J: Prentice Hall PTR, 2000. ISBN: 9780130217530 (cit. on pp. 22, 58).

[Lin20] Linux Foundation. *OpenAPI Specification*. Ed. by D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, U. Sarid. Version 3.0.3. Feb. 20, 2020. URL: http://spec.openapis.org/oas/v3.0.3 (visited on 03/23/2021) (cit. on p. 82).

[LSC20] A. Luzar, S. Stanovnik, M. Cankar. “Examination and Comparison of TOSCA Orchestration Tools”. In: *Communications in Computer and Information Science*. Springer International Publishing, 2020, pp. 247–259. DOI: 10.1007/978-3-030-59155-7_19 (cit. on pp. 23, 24).

[Mal19] D. Mala. *Integrating the internet of things into software engineering practices*. Hershey, PA: Engineering Science Reference, 2019. ISBN: 9781522577911 (cit. on pp. 13, 53).

[MHS05] M. Mernik, J. Heering, A. M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. DOI: 10.1145/1118890.1118892 (cit. on p. 26).

[Mor16] K. Morris. *Infrastructure as code : managing servers in the cloud*. Sebastopol, CA: O’Reilly Media, 2016. ISBN: 9781491924358 (cit. on pp. 19, 27, 38).

[Mor21] K. Morris. *Infrastructure as Code*. O'Reilly UK Ltd., Feb. 1, 2021. 430 pp. ISBN: 1098114671. URL: https://www.ebook.de/de/product/39207381/kief_morris_infrastructure_as_code.html (cit. on pp. 14, 19, 22, 38).

[New20] P. Newman. *The Internet of Things Report*. Tech. rep. Business Insider Intelligence, Mar. 1, 2020. URL: https://store.businessinsider.com/products/the-internet-of-things-report (cit. on p. 13).

[NHRR18] R. Nicolescu, M. Huth, P. Radanliev, D. D. Roure. "Mapping the Values of IoT". In: *Journal of Information Technology* 33.4 (Dec. 2018), pp. 345–360. DOI: 10.1057/s41265-018-0054-1 (cit. on p. 13).

[NTD15] S. Nastic, H.-L. Truong, S. Dustdar. "SDG-Pro: a programming framework for software-defined IoT cloud gateways". In: *Journal of Internet Services and Applications* 6.1 (Aug. 2015). DOI: 10.1186/s13174-015-0037-1 (cit. on p. 26).

[OAS17] OASIS. *Instance Model for TOSCA Version 1.0*. Ed. by P. Lipton, J. Crandall, D. Palma. Nov. 16, 2017 (cit. on pp. 14, 21, 23, 58).

[OAS19] OASIS Standard. *MQTT Version 5.0*. Ed. by R. Coppen, A. Banks, E. Briggs, K. Borgendale, R. Gupta. Mar. 7, 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (cit. on p. 13).

[OAS20] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Ed. by M. Rutkowski, C. Lauwers, C. Noshpitz, C. Curescu. Feb. 26, 2020. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html (cit. on pp. 23, 67, 68, 70, 73, 82).

[PDB15] T. Perumal, S. K. Datta, C. Bonnet. "IoT device management framework for smart home scenarios". In: *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*. IEEE, Oct. 2015. DOI: 10.1109/gcce.2015.7398711 (cit. on p. 26).

[Pri08] D. Pritchett. "BASE: An Acid Alternative". In: *Queue* 6.3 (May 2008), pp. 48–55. DOI: 10.1145/1394127.1394128 (cit. on pp. 30, 31).

[SBH+17] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. "Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments". In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2017. DOI: 10.5220/0006243303580367 (cit. on pp. 13, 25, 53, 87).

[SBK+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. "OpenTOSCA for IoT". In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM, Nov. 2016. DOI: 10.1145/2991561.2998464 (cit. on p. 25).

[Sch18] M. A. Schmid. *Provisionierung und Management von heterogenen IoT-Geräten mit TOSCA*. de. 2018. DOI: 10.18419/OPUS-10107 (cit. on pp. 25, 53, 87).

[Spa19] R. Spazzoli. *Kubernetes Operators Best Practices*. June 11, 2019. URL: https://www.openshift.com/blog/kubernetes-operators-best-practices (cit. on p. 27).

[The21] The Kubernetes Authors. *Operator pattern*. Jan. 3, 2021. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (cit. on p. 47).

[Vog09] W. Vogels. "Eventually consistent". In: *Communications of the ACM* 52.1 (Jan. 2009), pp. 40–44. DOI: 10.1145/1435417.1435432 (cit. on p. 29).

[WBB+20a] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “The EDMM Modeling and Transformation System”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 294–298. DOI: 10.1007/978-3-030-45989-5_26 (cit. on p. 24).

[WBB+20b] M. Wurster, U. Breitenbücher, A. Brogi, F. Leymann, J. Soldani. “Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020. DOI: 10.5220/0009571001710180 (cit. on p. 22).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The essential deployment metamodel: a systematic review of deployment automation technologies”. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Aug. 2019), pp. 63–75. DOI: 10.1007/s00450-019-00412-x (cit. on p. 24).

[WBH+20] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020. DOI: 10.5220/0009794302160226 (cit. on p. 24).

[Yad13] V. Yadav. *Edge Triggered Vs Level Triggered interrupts*. Mar. 5, 2013. URL: http://venkateshabbarapu.blogspot.com/2013/03/edge-triggered-vs-level-triggered.html (cit. on p. 27).

All links were last followed on April 4, 2021.

A Appendix

```
tosca_definitions_version: tosca_simple_yaml_1_3

namespace: https://legion2.github.io/ritam

interface_types:
  ritam.interfaces.Reconciler:
    derived_from: tosca.interfaces.Root
    description: The Reconciler interface, implement the Reconciler Pattern with TOSCA
    operations:
      reconcile:
        description: Reconcile function, used to reconcile the state of a Node.
      delete:
        description: Deletion function, used to delete the component
```

Listing A.1: RITAM Reconciler interface type definition

```
tosca_definitions_version: tosca_simple_yaml_1_3

imports:
  - file: https://legion2.github.io/ritam/ritam.yaml
    namespace_prefix: ritam

capability_types:
  MQTT-Broker-Endpoint:
    derived_from: tosca.capabilities.Endpoint
    valid_source_types: [tosca.nodes.SoftwareComponent]

node_types:
  MQTT-Broker:
    derived_from: tosca.nodes.Root
    properties:
      MQTT_URL:
        type: string
    capabilities:
      broker: MQTT-Broker-Endpoint
  Sensor:
    derived_from: SoftwareComponent
    properties:
      interval:
        type: string
      location:
        type: string
    requirements:
      - broker:
          capability: MQTT-Broker-Endpoint
          relationship: tosca.relationships.ConnectsTo
          occurrences: [1, 1]
    artifacts:
      index.mjs:
        type: ritam:ritam.artifacts.Implementation.JavaScript
        file: sensor/index.mjs
        deploy_path: /opt/temp-reader/index.mjs
      package.json:
        type: File
        file: sensor/package.json
        deploy_path: /opt/temp-reader/package.json
      package-lock.json:
        type: File
        file: sensor/package-lock.json
        deploy_path: /opt/temp-reader/package-lock.json
    interfaces:
      Reconciler:
        type: ritam:ritam.interfaces.Reconciler
        inputs:
          interval:
            type: string
            default: { get_property: [SELF, interval] }
          location:
            type: string
            default: { get_property: [SELF, location] }
          MQTT_URL:
            type: string
          DA_INDEX:
            type: string
            default: { get_artifact: [SELF, index.mjs]}
          DA_PACKAGE:
            type: string
            default: { get_artifact: [SELF, package.json]}
          DA_PACKAGE_LOCK:
            type: string
            default: { get_artifact: [SELF, package-lock.json]}
        operations:
          reconcile:
            implementation:
              primary: scripts/reconcileSensor.mjs
              dependencies:
                - type: File
```

```yaml
                  file: scripts/package.json
                  deploy_path: package.json
                - type: File
                  file: scripts/package-lock.json
                  deploy_path: package-lock.json
              timeout: 15
            outputs:
              status: [SELF, status]
          delete:
            implementation:
              primary: scripts/deleteSensor.mjs
              dependencies:
                - type: File
                  file: scripts/package.json
                  deploy_path: package.json
                - type: File
                  file: scripts/package-lock.json
                  deploy_path: package-lock.json
              timeout: 10
            outputs:
              status: [SELF, status]
              deleted: [SELF, deleted]

topology_template:
  inputs:
    interval:
      type: string
      description: Temperature polling interval
    location:
      type: string
      description: Location of the temperature sensor
    MQTT_URL:
      type: string

  node_templates:
    temp-reader:
      type: Sensor
      properties:
        interval: { get_input: [interval] }
        location: { get_input: [location] }
      interfaces:
        Reconciler:
          inputs:
            MQTT_URL: { get_property: [broker, MQTT_URL] }

    device:
      type: Compute
    broker:
      type: MQTT-Broker
      properties:
        MQTT_URL: { get_input: [MQTT_URL] }
```

Listing A.2: TOSCA Service Template of the motivating scenario IoT application

Declaration