Model-Based Fleet Deployment of Edge Computing Applications

Hui Song
Rustem Dautov
Nicolas Ferry
first.last@sintef.no
SINTEF Digital
Forskningsveien 1, 0314 Oslo, Norway

Arnor Solberg
Franck Fleurey
first.last@tellu.no
Tellu IoT AS
Lensmannslia 4, 1386 Asker, Norway

ABSTRACT

Edge computing brings software in close proximity to end users and IoT devices. Given the increasing number of distributed Edge devices with various contexts, as well as the widely adopted continuous delivery practices, software developers need to maintain multiple application versions and frequently (re-)deploy them to a fleet of many devices with respect to their contexts. Doing this correctly and efficiently goes beyond manual capabilities and requires employing an intelligent and reliable automated approach. Accordingly this paper describes a joint research with a Smart Healthcare application provider on a model-based approach to automatically assigning multiple software deployments to hundreds of Edge gateways. From a Platform-Specific Model obtained from the existing Edge computing platform, we extract a Platform-Independent Model that describes a list of target devices and a pool of available deployments. Next, we use constraint solving to automatically assign deployments to devices at once, given their specific contexts. The resulting solution is transformed back to the PSM as to proceed with software deployment accordingly. We validate the approach with a Fleet Deployment prototype integrated into the DevOps toolchain currently used by the application provider. Initial experiments demonstrate the viability of the approach and its usefulness in supporting DevOps in Edge computing applications.

CCS CONCEPTS

• **Software and its engineering** → **System modeling languages**; **Agile software development**; • **Hardware** → **Sensor applications and deployments**.

KEYWORDS

Software deployment, IoT, Model-based software engineering, device fleet, DevOps

ACM Reference Format:
Hui Song, Rustem Dautov, Nicolas Ferry, Arnor Solberg, and Franck Fleurey. 2020. Model-Based Fleet Deployment of Edge Computing Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410951

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410951

1 INTRODUCTION

With the rapid growth of the Internet of Things (IoT), extreme amounts of data are collected from the physical world. Due to network limitation and strict latency requirements, more and more data are processed close to where the data are generated[1]. This led to the emergence of *Edge computing*, where software components are deployed on devices at the *Edge* of the network, such as gateways, routers, small base stations, etc. An Edge computing application normally comprises tens to thousands of distributed Edge devices, collectively referred to as a device *fleet*.

Edge application providers are essentially software developers, and typically follow the modern DevOps practice to continuously add new features to their software running both on the cloud and at the Edge. In this context, a practical challenge is how to automatically deploy the updated software to the many distributed Edge devices after each DevOps iteration. Unlike the centralized cloud model, where computing resources are relatively homogeneous, an Edge fleet consists of distributed and heterogeneous devices, which have different contexts in terms of hardware capacity, network connection, user preferences, etc. Developers need to maintain multiple variants of the software to fit such different device contexts. This raises the main problem for the automatic deployment of Edge computing applications, i.e., how to assign m deployments (each "deployment" is a unique composition of variant software components) to n devices, so that each device is assigned with a proper deployment that matches its context, and at the same time, the whole system meets its global goals, e.g., keeping the software diversity within the fleet, and selecting a designated number of devices to trial a preview version.

To the best of our knowledge, there is no effective solution to this *fleet deployment* problem. The start-of-the-art Infrastructure as Code (IaC) tools automate the deployment of one application on one device, or a predefined set of devices. The mainstream IoT/Edge fleet management platforms offer to maintain multiple deployments, the fleet of devices, and their contexts, but developers still need to manually designate which deployment goes to which devices.

Aiming at this challenge, this paper presents an industrial research that applies model-based techniques to achieve automatic fleet deployment, by assigning multiple deployments to many devices in the fleet, without human interaction. The approach takes as input a Platform-Specific Model (PSM) from a fleet management platform, and transforms it into a Platform-Independent Model (PIM) that represents deployments, devices and their context. After that, the approach applies constraint solving techniques to the PIM

[1]Gartner estimates that more than 50% of data will be processed outside the cloud by 2022. https://cutt.ly/xyPv0jf

to map deployments to devices, guided by a set of hard and soft constraints based on the domain knowledge. The resulting PIM that includes the device-deployment mapping is then transformed back into the PSM which will be used by the platform to actually deploy the software.

We implement the approach as a fleet deployment tool, and integrate it into the DevOps toolset currently adopted by an e-Health provider. An experimental scenario using the tool through a series of DevOps iterations shows that the approach is able to produce valid assignments, exempting developers from heavy manual effort, and to provide valuable feedback for planning the subsequent development.

Accordingly, the contribution of this paper is threefold:

- A novel approach to automatically deploy multiple software variants on many Edge devices, with respect to device contexts and global goals of the whole fleet.
- A set of hard and soft constraints to achieve correct assignment and even distribution of software variants
- A research pilot on an industrial use case, showing that a model-based approach with meta-modeling and constraint solving can automate key DevOps activities and increase the development productivity.

The rest of the paper is organized as follows. Section 2 describes the fleet deployment problem via a motivating example. Section 3 presents our model-based approach and Section 4 details the use of constraint solving for fleet assignment. Section 5 presents the tool implementation and demonstrates its viability in an established DevOps process. Section 6 compares the approach to the existing related works, and Section 7 concludes the paper and discusses some limitations of the current work and the future steps.

2 PROBLEM STATEMENT

2.1 Motivating example

Throughout this paper, we will focus on an e-Health service provider named Tellu, which two coauthors of this papers work with. Tellu provides Remote Patient Monitoring (RPM) services. For each of their customers (typically elderly people living at their own residences), they provide a healthcare gateway – a small single-board computer similar to Raspberry Pi – together with a set of medical sensors, cameras and wearable emergency beepers. Each gateway collects measurements such as blood pressure, glucose and oxygen levels, etc., via Bluetooth, processes and aggregates the data, and sends them to the back-end cloud services. The patients and their nurses have access to the data via a Web interface and a mobile App. For some patients, Tellu sends their technicians to mount the gateway on the wall, while other patients can opt for having the gateway delivered from the manufacturer, and installing it themselves. Some patients choose battery-powered portable gateways to have a possibility to carry them with the essential sensors whenever they are outside their houses, e.g., for walking, taking medical examinations, or travelling. A recent feature under development by Tellu is *fall risk detection* based on machine learning, which combines live gesture detection via cameras together with other real-time physiological and environmental data to continuously assess the risk of patient falling down in a short term.

Since all the devices can be purchased off the shell, Tellu is essentially a software vendor. Their main effort and focus is on developing the front-end and back-end software components, running on the gateway and the cloud, respectively. The development team continuously adds new features to the software and patches issues, following a DevOps practice. During the process, they produce and simultaneously maintain multiple variants of the front-end software, which conceptually may be separated into *vertical* and *horizontal* ones. Vertically, they have *development versions* with the most recent features running on staging devices, and *released versions* with mature features running on production-ready devices, already used by patients. In between of them, they regularly deploy *preview* versions with new features to a small set of selected patients, in order to collect feedback from real users. Horizontally, they maintain software variants that fit gateways with different setups and contexts. For example, if a variant has the machine learning (ML) module for fall detection running in the back-end, then it only fits those gateways that are connected to WiFi network, not 4G, since they need to send images to the cloud. Alternatively, the variant with the ML module running fully on the edge only fits the gateways that are mounted on the wall, since the heavy computation load implies risks of overheating the self-installed gateways (which usually do not have proper cooling/ventilation facilities). As an option, some patients may choose to buy an additional hardware accelerator (typically in the form of a pluggable USB dongle, such as Google Coral [16]) which can reduce the computation load on the gateway, but it requires an application variant with a re-compiled ML module). Since no variant fits every device, Tellu has to maintain multiple variants (deployments[2]), and assign them to the devices according to their contexts.

In summary, Tellu maintains a fleet of about 500 gateways for their patients, and additional 10 local installations on their own premises. The gateways have different contexts, in terms of network, mounting, accelerator, etc. The development team maintains around 3-10 different variants of the front-end software. The Tellu team expect to conduct DevOps cycles on a daily basis, retiring parts or all of the active variants and introducing new ones. However, since there is a mass re-deployment after each cycle involving many or sometimes all the devices, and it is not fully automatic at the moment, they cannot achieve such high frequent DevOps cycles. Automatic fleet deployment becomes a bottle net for the development team.

2.2 Fleet deployment: concepts and challenges

Fleet deployment is a problem of automatically deploying software on many functionally similar devices, while the actual software deployed may vary from device to device, based on the context.

Figure 1 illustrates the conceptual architecture of a fleet deployment framework, with three main components:

- *Fleet monitoring*, which manages the life-cycle of all the devices, and collects the context for each device;
- *Fleet assignment*, which maintains active candidate deployments, and assigns a deployment plan to each device.

[2]Each software variant represents a unique way of deploying the software. Therefore, throughout the paper, we use *software variants* and *deployments* interchangeably.

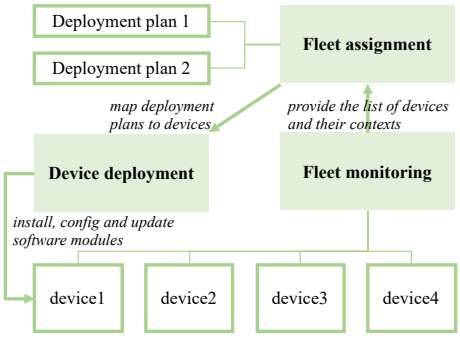

Figure 1: Main components for managing a fleet of Edge devices.

- *Device deployment*, which enacts the assigned deployment to each device, by installing software modules according to the deployment plan.

There are several commercial IoT/Edge development platforms providing *fleet monitoring* support, such as Azure IoT Edge [19]. At the same time, *device deployment* is essentially an Infrastructure as Code (IaC) problem – i.e., developers provide a model specifying where to obtain the software modules their libraries, and the corresponding engine will deploy them accordingly on the Edge device. Mainstream IaC solutions, such as Ansible [18], also supports remote deployment. However, *fleet assignment* is still an open question.

This paper focuses on *fleet assignment*, which can be abstracted as a problem of finding the function d:

$$d : Dv \rightarrow Dp \cup \{\perp\}$$

where Dv is the set of all devices and Dp is the set of active deployments. If no proper deployment can be found for a certain device $dv \in Dv$, then $d(dv) = \perp$. This function d is the main output of fleet assignment, indicating what software should be deployed to each device. In addition, the assignment may also involve a set of functions for the configuration of deployments on each device:

$$c_x : Dv \rightarrow Vx$$

which passes some additional parameters to each assigned deployment. For example, $c_{ie}(dv_1) = \top$ means the device dv_1 is configured in the way that machine learning is running on the gateway (where *ie* means *Intelligence on Edge*)

Fleet assignment is challenging, because it must satisfy the following requirements for all devices, at the same time:

R1. Match hardware capacity. The software modules that are assigned to a device must match the particular device capabilities, including CPU architecture, memory volume, hardware accelerator, etc.

R2. Fit software development pipeline. A deployment that is still under development can be only deployed to staging devices. At the same time, developers may want to deploy the preview version on a designated number of production-ready devices for direct feedback.

R3. Align the resource usage. The deployment assigned to a device should have reasonable resource consumption, e.g, low CPU usage for a device powered by battery. Note that the same deployment may have different consumption on different devices, influenced by offloading, hardware acceleration, etc.

R4. Achieve global objectives for deployment distribution. A typical objective is software diversity, i.e., even distribution of deployment plans among the devices for the sake of resilience, robustness and security.

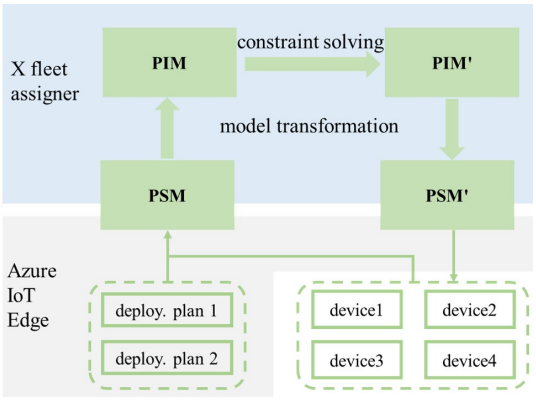

Figure 2: Overall architecture for model-based fleet deployment.

In practice, an Edge computing application may involve many devices that may run with various contexts, and thus can be deployed in multiple alternative ways. In such case, fleet assignment must handle multi-dimensional constraints for a large collection of objects. Research approaches to the automatic searching of solutions under complex constraints have resulted in the emergence of multiple theories and tools, collectively referred to as *Boolean Satisfiability Problem* (SAT) [26], and more recently *Satisfiability Modulo Theories* (SMT) [4]. Given the various constraints and conditions, employing constraint solving theories and tools is a promising approach towards a fully automated solution for fleet management. However, a main barrier is the gap between the abstract SAT or SMT models and the concrete representation of devices and deployments obtained from the existing platforms for fleet monitoring and device software deployment.

3 MODEL-BASED FLEET DEPLOYMENT

We employ a model-based approach to automate the fleet deployment problem, using meta-modeling and model transformation to bridge the gap between the abstract constraint solving theory and the concrete Edge computing platforms. Figure 2 illustrates the overall architecture of the approach.

We obtain from the platform a PSM (Platform-Specific Model) that represents the available devices and deployments. We transform a PSM into a PIM (Plantform-Independent Model), which represents the essential information, such as devices, deployments, and their attributes, using a simple and standard notation, agnostic with regard to the platform. This original PIM, essentially representing the information from fleet monitoring, is not a complete model yet, since the mapping between devices and deployment

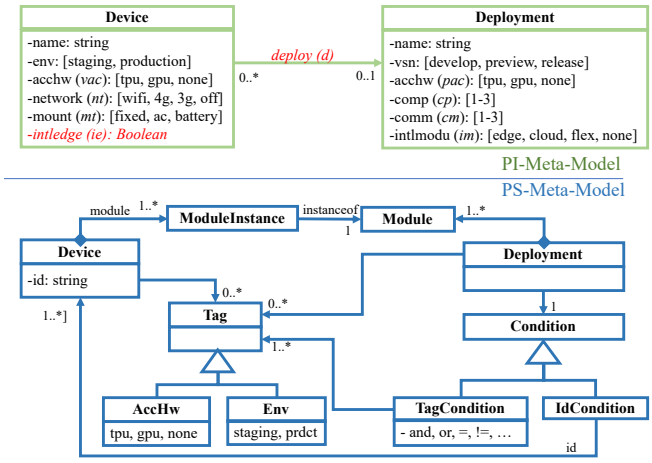

Figure 3: The meta-models.

plans is missing, together with some run-time configurations. We apply *constraint solving* on this PIM to automatically search for the missing part. The solution provided by the constraint solver is a new PIM′ with the missing part. We then transform PIM′ into a new PSM′, which will be finally used by the platform to install the software modules. Transforming back to PSM is necessary since it is the format that the fleet management platform understands.

The approach itself is generic across multiple Edge computing platforms – i.e., for different platforms, we need to define different PSMs and the model transformation, while the PIMs and the constraint solving approach remain the same. As reported below, in the context of this paper, we use Azure IoT Edge as the target platform to implement a Fleet Deployment tool and demonstrate the viability of the proposed approach.

3.1 The meta-models

Figure 3 shows the meta-models we defined for the RPM use case (Section 2.1).

The Platform-Specific Meta-Model (PSMM) in the lower half of Figure 3 reflects how the Azure IoT Edge platform manages the device fleet and deploys software modules. The platform maintains a set of Devices and a set of Deployments. Both devices and deployments are accompanied by a set of attached Tags. A tag can carry numeric, Boolean or enumerated values, which are either manually added by the application developers, or automatically collected by the platform from the environment at run-time. Each deployment contains a set of Modules. A module is a software artefact, such as a Docker image. If a device is assigned with a deployment, then for each module in the deployment, the platform will create one Docker container using the image. This container is represented as a ModuleInstance.

The mapping between devices and deployments are defined indirectly through the *Conditions* of the deployment. Each deployment contains one condition, defining to which devices the deployment applies. Azure IoT Edge supports two types of conditions. Developers can define a condition as an expression on the tags, which means all the devices that are attached by a specific group of tags. Alternatively, developers can directly refer to the device IDs in the condition. In this paper, we use the ID-based conditions to represent the solving result.

The Platform-Independent Meta-Model (PIMM) in the upper half of Figure 3) contains two simple classes, i.e., Device and Deployment. Each class contains a set of attributes, and there is one relationship, deploy, between the two classes. What attributes are needed depends on the problem domain, and in this paper, we define them according to the RPM example. For a Device, env specifies whether the it is used for staging or production; acchw specifies what hardware accelerator is used for the device; mount specifies whether the device is mounted to the wall, simply plugged to an A/C socket (installed by the patient themselves), or powered by batteries; and network specifies the network used primarily by the device. These attributes are used as input for fleet assignment. In addition, intlege tells whether the machine learning modules will be executed on the Edge device, which is used to carry the fleet assignment result. On the deployment side, vsn specifies the development phase of the software behind this deployment; acchw specifies the type of hardware accelerator, against which the machine learning module was compiled; comp and comm estimates the levels of computation and communication resources required by the software; intlmodu defines where the machine learning model can be executed. In Figure 3, some attribute names are followed by shortcuts, which will be used for defining constraints in the next section.

3.2 Model transformations between PIM and PSM

The approach employs *forward* (from a PSM to a PIM) and *backward* (from a PIM to a PSM) transformations, main aspects of which are described below.

In the forward transformation, PSM elements of types Device and Deployment are mapped bijectively to homonymous PIM elements of devices and deployments, respectively. Tags on the PSM elements are transformed to attributes, where tag names are mapped to attribute names, and tag values to attribute values. If an element does not carry an optional tag, such as acchw, the attribute value will be assigned with a default value, such as none. Attribute intledge and Reference deploy in PIM′ are left.

The backward transformation takes PIM′ and the original PSM as input and yields a new PSM′. For each Device in PIM′ (e.g., dv1), we find the homonymous Device in PSM, and propagate the values of intlege and deploy: For intlege, we create a new tag for the PSM device. For deploy, we find the corresponding deployment plan in the PSM, and append "or id = dv1" to its IdCondition (or create a new IdCondition if there is none).

3.3 Sample models

Figure 4 illustrates sample PSM and PIM in the context of considered example. According to the PIM, the system comprises of 6 devices. The second device (dv2) has a Tensor Processing Unit (TPU) as accelerator, connected by 4G and powered by A/C. Together with dv1, the two devices are used for staging. The other four devices are used in production. We have 4 deployments under maintenance:

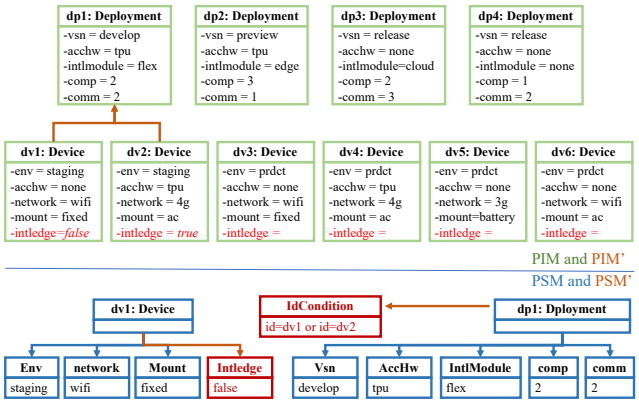

Figure 4: Sample PIM and PSM for the motivating example.

dp1 is still under development, and supports TPU and flexible offloading. The worst-case computation and communication levels are both medium (2 out of 3). dp2 is a preview version, which hosts the machine learning module only on the Edge, and supports TPU. The worst-case computation requirement is high, but communication requirement is low, since the analytics is done on the Edge. Attributes (intledge) and reference (deploy) highlighted in red do not exist in PIM, but will be added in PIM′. In the figure, we only show the partial assignment for the first two devices.

Due to the space limitation, we also only show part of PSM, with one device and one deployment with attached attributes represented as tags. In PSM′ (the original elements in PSM plus the additional ones marked in red in the same figure), one new tag is created, and the deploy reference is represented as IdCondition.

4 FLEET ASSIGNMENT USING CONSTRAINT SOLVING

We employ SMT constraint solving and an existing implementation, Z3 solver [12], to achieve automatic fleet assignment. To do so, we represent the PIM as an SMT problem, and define a set of hard and soft constraints according to the domain knowledge. The solver will automatically search for the missing values in the PIM, according to the constraints.

4.1 Representing the PIM as an SMT problem

An SMT problem is a decision problem for logical formulas on a combination of background theories. A basic built-in theory for SMT is the *uninterpreted function theory* [7]. An uninterpreted function is a function that is defined with only domain and codomain, without an interpretation on how the domain is mapped to the codomain. The building blocks of an SMT problem are a series of such uninterpreted functions, on which we can define first-order logic formulas as constraints.

We transform a PIM into an uninterpreted function theory problem based on the following rules, explained using the sample PIM in Figure 4.

- All devices form a set $Dv = \{dv_1, ..., dv_6\}$, and all deployments form a set $Dp = \{dp_1, ..., dp_4\}$.
- The deploy relation is represented by a function $d : Dv \rightarrow Dp \cup \{\perp\}$, as discussed in Section 2.
- Attributes in primitive types are represented as functions to the corresponding types, e.g., $cp : Dp \rightarrow \mathbb{N}$ and $ie : Dv \rightarrow \mathbb{B}$, for comp and intledge, respectively.
- The attributes whose values are within a particular set, are represented by functions with codomains as the set of all the valid choices. For example, the version attribute is a function $vsn : Dp \rightarrow \{\text{develop, preview, release}\}$.

A constraint solver can automatically search for an interpretation for each of the uninterpreted functions that we defined above. The interpretation of function d (i.e., giving a dp for every $dv \in Dv$) and the interpretation of ie (assigning either true or false for every $dv \in Dv$) are the result for fleet assignment. The searching process will be guided by a set of constraints from: (i) the current model status, i.e. the existing attribute values; (ii) the domain-specific knowledge; (iii) global optimization objectives. In the rest of this section, we will present the three types of constraints, respectively.

4.2 Constraints representing the current model state

In the current PIM, the devices and deployments already have existing attribute values, which serve as the basis for fleet assignment. Therefore, we translate all the current attribute values into hard constraints. The example below shows how we generate the constraints from the current state of dp1:

$$vsn(\text{dp1}) = \text{develop} \ \wedge \ pac(\text{dp1}) = \text{tpu} \ \wedge \\ ie(\text{dp1}) = \text{flex} \ \wedge \ cp(\text{dp1}) = 2 \ \wedge \ cm(\text{dp1}) = 2$$

The shortened function names are defined in the meta-model in Figure 3. We generate the constraints automatically by iterating over all the devices and deployments in the PIM, reading the attribute values, and generating the equations accordingly.

4.3 Domain-specific constraints

We define a set of hard constraints based on the domain knowledge about software deployment on Edge devices. These constraints are related to the requirements **R1** to **R3** in Section 2. To simplify the constraint definition, we first introduce 3 auxiliary functions.

$$acc : Dv \rightarrow \mathbb{B}; \ vcp : Dv \rightarrow \mathbb{N}; \ vcm : Dv \rightarrow \mathbb{N}$$

Here acc indicates whether the hardware accelerator will be used on a device, and vcp and vcm are the actual levels of computation and communication resource consumption on the device, which is determined not only by the levels defined in the deployment, but also by the configuration of the device, such as whether the machine learning module is executed on the Edge device or on the cloud (ie) and whether the accelerator is used (acc).

We start from the constraint related to the development stage (**R2**): any deployment with software under development should be only deployed to a staging device:

$$\forall dv \in Dv, dp \in Dp : d(dv) = dp \ \wedge \ vsn(dp) = \text{develop} \\ \implies env(dv) = \text{staging}$$

The next set of constraints requires that the resource consumption of a deployment should match the capacities of its target device (**R1** and **R3**).

The first step is to define how the actual consumption of each device is determined by the theoretically worst-case consumption defined by the developers.

$$\forall dv \in Dv, dp \in Dp : d(dv) = dp \implies$$

$$vcm(dv) = \begin{cases} cm(dp) - 1 & im(dp) = \text{flex} \wedge ie(dv) \\ cm(dp) & \text{otherwise} \end{cases}$$

$$vcp(dv) = \begin{cases} cp(dp) - 1 & im(dp) = \text{flex} \wedge \neg ie(dv) \\ cp(dp) - 1 & acc(dv) \\ cp(dp) & \text{otherwise} \end{cases}$$

For communication, if a deployment allows flexible offloading of the ML model, then running the model on the Edge device would reduce the requirement on communication, since the raw data will be consumed locally without being transferred to the cloud. For computation, there are two cases that the actual computation requirement on the gateway will be reduced: (i) When offloading is supported, and the model is not executed on the Edge, the computation requirement will be reduced, since the computation task is offloaded to the cloud; (ii) When hardware accelerator is used, the computation requirement is reduced, since the task is offloaded to the accelerator.

The second step is to regulate when we can offload the ML module to the cloud or the hardware accelerator.

$$acc(dv) \iff ie(dv) \wedge vac(dv) = pac(dp) \neq \text{none}$$
$$ie(dv) \implies im(dp) \in \{\text{flex}, \text{edge}\}$$
$$ie(dv) \impliedby im(dp) = \text{edge}$$

Offloading to accelerator happens (i.e., $acc(dv) = \top$), *if and only if* the ML module is allocated to the device (i.e., $ie(dv)$) and the type of accelerator supported by the deployment match the accelerator attached to the device. The equivalence statement means that whenever it is possible, we will opt for acceleration, which is the most reasonable for our use case. The ML module is executed at the Edge (i.e., $ie(dv) = \top$), only if in the deployment the ML module is defined as flexible or running on the Edge. On the other hand, if in the deployment the ML module is defined to be only running on the Edge, then on the device, $ie(dv)$ must be on. It is worth noting that when $im(dp) = \text{flex}$, $ie(dv)$ can be either true or false, which is a decision to be made by fleet assignment.

The last set of constraints describes how the device contexts determines what level of resource assumption is allowed.

$$\begin{aligned} \forall dv \in Dv : & \\ nw(dv) = 4\text{g} &\implies vcm(dv) < 3 \\ nw(dv) = 3\text{g} &\implies vcm(dv) < 2 \\ mt(dv) = \text{ac} &\implies vcp(dv) < 3 \\ mt(dv) = \text{battery} &\implies cvp(dv) = vcm(dv) = 1 \end{aligned}$$

The rationale behind these constraints in the sample domain is threefold. First, the worse network a device has, the lower communication level it can support. Second, if the device is not mounted professionally, it should not run software modules with the highest computation requirement. Finally, if a device is powered by a battery, it should opt for the lowest computation and communication consumption.

4.4 Soft constraints to optimize the distribution

In addition to the hard constraints that must be satisfied by the assignment, we also define a set of soft constraints to guide the assignment towards optimized global distribution (**R2** and **R4**). The solver can choose to violate a soft constraint, with an agreed penalty (a.k.a the *weight* of the constraint), and it aims at the minimal total penalty for the solving result. In our case, we used the following three groups of soft constraints.

The first group of software constraints suggests each device to be assigned with a deployment. For each device, we generate a soft constraint for its deployment to be not empty, with a penalty of 50.

$$(d(dv_1) \neq \perp\ (p : 50),\ \ldots,\ d(dv_6) \neq \perp)_{penalty=50}$$

The second group consists of only one soft constraint, which instructs the solver to select 20% of all production devices to trial the preview deployments (see **R2** in Section 2). The left side of the equation counts all the devices assigned with a preview deployment, while the right side counts all the devices in production and multiplies it by 0.2.

$$(|\{dv \in Dv | vsn(d(dv)) = \text{preview}\}| = \lceil 0.2 \cdot |\{dv \in Dv | env(dv) = \text{production}\}| \rceil)_{penalty=100}$$

The third group of soft constraints aims to evenly distribute the deployments in order to maintain the diversity of deployments in the whole group (R4). In the two constraints defined below, we count the number of devices that are assigned with each deployment, and suggest the number to be close to the average number of devices per deployment (i.e., within the scope of ±20% of the average number).

$$(|\{dv \in Dv | d(dv) = dp_1\}| > 0.8 \cdot |Dv|/|Dp|)_{penalty=20}$$
$$(|\{dv \in Dv | d(dv) = dp_1\}| < 1.2 \cdot |Dv|/|Dp|)_{penalty=20}$$

This is a relatively low-cost way to achieve even distribution, compared to other complex methods, such as minimizing the total variance (i.e., $\min \sum(\tau_i - \overline{\tau})^2/n$, where τ_i is number of devices for dp_i).

4.5 Implementation in the Z3 constraint solver

We use the Z3 SMT solver [12] to automatically search for a solution for the uninterpreted functions under the constraints. We define the SMT problem in Z3Py, the Python interface of Z3, feed the defined constraints into the solver as input, and extract the results.

Listing 1 demonstrates a simplified example to illustrate the whole process. We first define an enumerated type Dp to represent the set of all deployments (Line 1), followed by an uninterpreted function d for deploy (Line 3). The definition for Dv is similar, and is therefore omitted in this example. Function env links Device to another enumerated type representing the environment choices (Lines 4-5), and cp is a function from Deployment to integer. After defining the functions, we instantiate a solver (Line 7), and add constraints to it. We give two examples to show how to program the constraints that we presented earlier. The first one is a hard

Listing 1: Sample SMT problem in Z3Py.

```
1 Dp, dps =
2   EnumSort('Dp', ['dp1', 'dp2', 'dp3', 'dp4', 'nodp'])
3 d = Function('deploy', Dv, Dp)
4 Env, [staging, prod] = EnumSort('Env', ['staging', 'prod'])
5 env = Function('env', Dv, Env)
6 cp = Function('comp', Dp, IntSort())
7 solver = Optimize()
8 solver.add(ForAll([dv, dp], Implies(
9   And(d(dv) == dp, vsn(dp) == develop),
10  env(dv) == staging)))
11 for i in dvs:
12   solver.add_soft(Sum([If(d(j)==i, 1, 0) for j in dps]))
13     < len(dv)*1.2/len(dp), 20)
14 solver.check()
15 assert dp2 == solver.model().eval(d(dv4))
```

constraint indicating that a "develop" deployment can be only deployed to a "staging" device (Lines 8-10). The second example is a group of soft constraints: For each deployment, we generate a constraint stating that the total number of devices assigned with it should not be more than 120% of the average number (Lines 11-13). After feeding the solver with all constraints, we launch the solving procedure (Line 14), and then, as an example, check for a deployment assigned to Device No.4.

5 APPLICATION AND EVALUATION

This section puts theory into practice by presenting how the approach was applied to the use case of Tellu and demonstrating its effectiveness in their DevOps routine.

5.1 Implementation and integration

The proposed approach was implemented as an automatic Fleet Assignment prototype tool[3], and integrated into the current DevOps toolset of Tellu.

Figure 5 shows the new DevOps toolchain for Tellu. Except for the fleet assignment tool (noted as green box), all the other tools are the same as the ones they use currently in their production. At the present, the team uses Ansible for device deployment, and use Azure IoT Edge for fleet monitoring. The process is driven by a Jenkins pipeline. In each *Dev* loop, after coding and testing the new changes, Tellu releases a new version of their application, and creates a number of new deployments which contain the new version. These deployments differ from each other in terms of configurations and the connection with other components, libraries, and/or services. They label the new deployments with different tags, occasionally update the tags of existing deployments, and retire some old deployments. With the current tools, after each release, they need to manually update the mapping between devices and deployments, in the form of several Ansible Inventory files, each of which records a list of IP addresses for the devices that apply for one deployment plan (which in turn is in the form of an Ansible Playbook).

[3]All source code available in the open source repository: https://github.com/SINTEF-9012/divenact

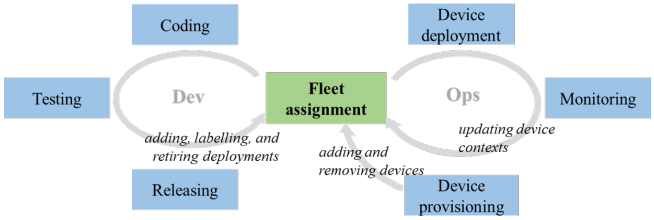

Figure 5: The DevOps process enhanced by automatic fleet assignment.

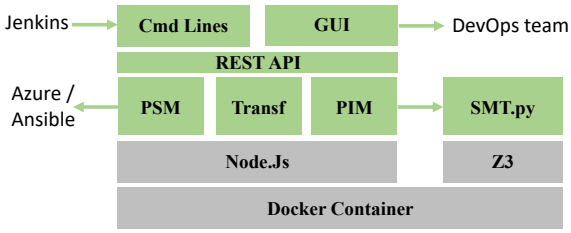

Figure 6: Technical architecture of the tool implementation.

The fleet assignment tool replaces this manual task. After each release, the tool will be automatically triggered, and will provide a mapping between the current devices and the active deployments (the Ansible inventory files). The device deployment tool receives the new assignment and enforces re-deployment of each individual device at a pre-defined timeslot (e.g., the nearest available midnight when the device is online). During runtime, the monitoring system will keep track of the device context and update the list of devices maintained by the Fleet Assignment tool. The device provisioning tool continuously adds new devices into the device list, after new patients entering the service, or existing patients switching to new devices. Such changes will also trigger fleet assignment action.

The fleet assignment tool itself is implemented as a Web Service, with the main components described in Figure 6. The main backend service is composed of the three modules for PSM, PIM and the transformation between them, as described in Section 3. The PSM module invokes the external fleet management and device deployment tools through REST API, while the PIM module invokes the constraint solving logic, as presented in Section 4, implemented as a single Python script SMT.py. The backend service also exposes its own REST API for other tools to check the status of deployments and devices, edit the labels, add additional constraints and trigger the assignment process. Based on the API, we provide two interfaces: The command line interface is used by Jenkins to automatically trigger assignment; The web-based GUI allows the developers to manually check the device and deployment status and control the assignment. More details about the GUI can be found in our earlier publication [11], which describes the general fleet management support without the assignment approach based on constraint solving.

Table 1: Application version after each DevOps iteration.

Dep.	Summary of features
A	Base version, no ML (comm:1, comp: 1)
B	ML running on cloud (comm: 3, comp: 1)
C	ML running on gateways (comm: 1, comp: 3)
D	Flexible ML offloading (comm: 3, comp: 3)
E	Improve C by supporting accelerator (comm: 1, comp: 3)
F	Improve D, offloading + accelerator (comm: 3, comp: 3)
G	Improve E with lighter model (comm: 1, comp: 2)

Table 2: Assignment in Demonstration scenarios.

#	deploy ments	staging devices				production devices																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	A||	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
1	A|*B*	A	B	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
2	A|B|C	C	A	B	A	A	B	A	A	A	B	A	B	A	A	B	A	A	A	A	A	A	A	A	A	A
3	AB|C|D	D	A	B	A	A	B	A	C	C	B	A	B	C	A	B	A	C	A	A	A	A	A	A	C	A
4	ABC|D|E	C	E	B	A	D	D	A	D	D	B	A	B	D	C	B	A	C	A	A	A	A	A	A	C	A
5	ABCD|E|F	F	E	D	A	D	C	A	E	D	A	A	D	E	C	E	A	C	E	E	A	A	A	A	C	A
6	ABDEF||	D	E	B	A	D	B	A	D	F	B	A	F	F	E	B	A	E	E	F	A	A	A	A	D	A
7	BDEF||	B	F	E	-	D	E	-	D	D	F	-	F	D	E	D	-	E	E	F	-	-	-	-	F	-
8	DEF||	D	F	E	-	F	E	-	E	E	F	-	E	F	E	F	-	E	E	F	-	-	-	-	F	-
9	DEFG||	D	E	F	-	D	F	G	D	D	F	G	D	E	E	D	-	E	E	F	-	G	G	G	F	-

5.2 Application scenarios

We evaluate the effectiveness of the approach by applying it to a DevOps scenario that simulates the real development activity at Tellu, and observing how it assigns the deployments at different stages during the development process. The sample scenario demonstrates that:

(1) Automatic fleet assignment is useful for developers to accelerate the DevOps process.
(2) The constraint solving approach, and the sample constraints presented in Section 4, can produce valid assignment for developers.

The scenario consists of a series of DevOps iterations, each of which yields a new version of the application, changes the status of existing versions, and/or retires some old versions. Through these iterations, the developers gradually create, test and release 6 versions of the RPM application, in order to introduce and improve the new fall detection feature with a machine learning module. Table 1 summarizes these versions, and their resource consumption. Each version corresponds to one deployment.

Table 2 summarizes the automatic assignment result after each iteration. The first column lists the set of deployments maintained by the developers after each iteration. We use the vertical bar sign | to divide the deployments into three groups, according to the different development phases (or *vsn* as we named it in the model). For example, "AB|C|D" means deployment "A" and "B" are released to production, "C" is under preview and "D" is under development. As iterations go on, more deployments are released and go into preview or deployment phases.

The rest of the table schematically illustrates how the proposed tool automatically assigns the deployments after each iteration to the 25 devices, including 4 staging devices and 21 production devices. The devices have different contexts in terms of mounting, network, and accelerator, based on sampling with the actual devices maintained by the company. We omit the actual profile of each device, since the important part is to evaluate how the approach provides valid assignments as a whole, rather than to check if each single assignment satisfies all the constraints (we do not need to validate the correctness of Z3 solver).

We go through the assignment results during the entire scenario, step by step, to show how the automatic fleet deployment benefits developers. For each step, we give a brief description of the development purpose, followed by how the automatic assignment results support this purpose.

1. Baseline deployment. In the beginning, A is the only valid deployment and has the lowest resource requirements, and therefore all devices are assigned with this deployment.

2. First attempt with machine learning (ML). Developers first implement B with the ML module running in the cloud, which requires devices with WiFi due to the high communication consumption. Two staging devices were selected after iteration 1 to test B internally, but afterwards only 3 were picked for preview in #2 (expected to be 20%, or 4 devices) and the same 3 after B is released (#3), since WiFi is not widely used by patients – most of them would skip configuring WiFi and keep using the default 4G network. The developers get the feedback that running ML only in the cloud has limited usage for their users.

3. Migrating ML to the Edge. Deployment C with ML running on the Edge (tested, previewed and released through #2-#4) is complementary to B: It requires fixed mounted devices but not necessary WiFi. Comparing iteration 2 and 3, C is deployed to devices that B was not assigned to, which is in line with the developers' expectations.

4. Support Cloud-Edge offloading. The next version D allows flexible offloading. Due to the networking overheads, it has some extra cost[4], and is therefore not a complete substitution to B and C. The developers choose to keep all the three deployments. They would expect an even distribution of A-D among the production devices at 4, but the assignment result has too many A's (10 devices in total), which indicates that there are still many devices that cannot run B, C or D.

5. Utilize hardware accelerators. Deployment E and F add accelerator support to C and D, respectively, by adding an ML module compiled for Edge TPU. E and F still keep the original ML module, so that they can also run on devices that do not have a TPU (in that case, the computation consumption would not be lowered), This means that E can completely replace C. Therefore, when releasing F (#6), the developers also retire C. The new assignment result does not increase the number of A's, which leads the developers to hypothesize that the remaining 10 devices with A could not run any of the current deployments with ML.

6. Try to retire the base version. To confirm the hypothesis, the developers attempts to retire A (#7), and, as expected, 8 production devices end up with no deployments, and so is one staging device (Device 4). They go on to retire B, based on an observation back in #5 that when the assignment tool tried to recruit a sufficient number of devices to preview E, it removed B completely, which

[4]E.g., if ML is offloaded to the cloud, the computation cost is down to 2 comparing to 1 in B.

Table 3: Time spent (in seconds) on assigning deployments.

#	Dep.	Number of devices				
		25	50	100	200	400
2	A|B|C	0.32	0.61	2.53	10.69	59.52
4	ABC|D|E	0.33	0.94	4.62	31.17	121.31
5	ABCD|E|F	0.77	1.91	8.44	92.52	606.63
6	ABDEF||	0.19	1.28	4.79	16.60	101.55
9	DEFG||	0.16	0.78	3.14	15.22	170.50

means B is replaceable by the subsequent versions. The result at #8 also confirms this, as the number of un-deployed devices is the same. As trial steps in #7 and #8, the developers run assignment without executing the subsequent device deployment, to avoid production devices having no deployments.

7. Optimize ML module. By examining the profile of un-deployed devices, the developers realize that the computation power is the key problem, and therefore they simplified the ML module and produce G with lower computation requirement. By releasing G (#9, we skip the iterations for testing and previewing G, for the sake of space limitation), only four devices remained un-deployed. This can be further solved by sending accelerators to these devices, so that eventually all the devices can be equipped with the new ML feature.

In summary, the scenario demonstrates the following benefits for developers.

- **Automation**. After each iteration, developers do not need to manually select matching devices for preview and release.
- **Handling complexity**. Maintaining multiple deployments in parallel often needs to re-assign existing deployments to make space for new ones. Taking the transition from #4 to #5 as a simple example, the assigner has to change Device 12 from B to D, in order to release Device 13 to preview E. When many devices are involved in such re-arrangement, it would be too complex for manual work.
- **Testing**. Running fleet assignment without the device deployment step is an efficient way to test the composition of deployments, as is shown by the attempt to retire A and B.
- **Feedback**. Knowing that the assignment tool will satisfy the constraints and will aim at a balanced distribution, developers can use the results to understand the missing parts of the current deployments and plan for subsequent development.

5.3 Performance and scalability

As the first step, the performance requirement to the approach is that it is feasible to be used in the current setup of Tellu. We conducted a series of experiments to test this, by selecting 5 out of the 10 compositions of deployments in Table 2, and trying to assign them to different numbers of devices (from 25 to 400) with generated profiles. We conducted the experiments on a MacBook Pro laptop with 3.1 GHz Intel Core i5 processor and 16GB Memory. It is worth noting that the experiments only reveals the time for fleet assignment. The time used for actual deployment may vary significantly depending on the size of deployment, the device and the internet connection.

Table 3 lists the average time spent on fleet assignment after each iteration to different numbers of devices. For each iteration, we start with the same 25 devices as used in the last section, run assignment for 10 times and record the average time in seconds. After that, we double the number of devices and repeat it until reaching 400 devices, which is currently the number of devices managed by Tellu. The time for assignment increases significantly as the number of devices or deployments increases. The composition of deployments also has an impact: Comparing #4 and #6, the one with simpler composition is marginally faster to assign. The experiments confirm the feasibility of the current approach: Assigning 6 deployments to 400 devices in 10 minutes, albeit not perfect, is still acceptable for developers, since other automated DevOps steps, such as building, testing and device deployment takes comparable time, from several minutes up to an hour.

The experiments also show the limitation in terms of scalability. Consider one hour as the up limit, when the application system scales out to more devices, we would need to divide the whole fleet into multiple sub-fleets and conduct the assignment separately on each sub-group. We also foresee that developers may add new attributes to the meta-model and new constraints, making it somewhat more computationally complex. In such cases, we may need to carefully design the constraints, limit the size of sub-fleets, etc. Further experiments and guidelines for performance optimization is part of the future work.

6 RELATED WORK

Managing software updates across a large fleet of devices has been a key challenge for the leading mobile OS providers, such as Apple and Google. At present, their adopted over-the-air update mechanism implements a publish-subscribe model, where mobile devices first get notified of and then fetch available updates through the centralised marketplace. Since smartphone fleets are rather homogeneous (i.e., modern smartphones are equipped with more or less the same resources and capabilities), there is no challenge of multi-criteria software assignment as such, and the compatibility check is performed only once, upon the initial installation. Furthermore, in a fleet of smartphones there are typically no global system goals, such as even distribution of components or A/B testing.

Deploying software applications on hardware platforms with respect to heterogeneous contexts is not a new problem, and has been studied for several decades now. In particular, the Software Communication Architecture (SCA) [15] describes how properties of waveform software components are mapped to heterogeneous device characteristics, thus enabling Software-Defined Radio (SDR) systems. Another relevant reference architecture for deploying component-based applications into heterogeneous distributed target systems is described in [20]. In particular, the proposed architecture includes the concept of *Planner* – a component responsible for matching software requirements to available platform resources and deciding whether a component is compatible with a device. These existing specifications remain implementation-agnostic and only describe the high-level concepts.

Automatic deployment is a key challenge to adopt the DevOps practices. Infrastructure as Code (IaC) tools, such as Ansible [17] and Chef [9] can be used to support the installation of software

modules on devices, while Edge computing frameworks such as Azure IoT Edge [19] and AWS IoT Greengrass [1] provide online services to maintain device fleets throughout the whole device lifecycle. Our approach is complementary to these tools, and provides the missing capability of assigning multiple deployment models to many devices in a reliable automated manner.

In general, the assignment problem frequently appears in ICT scenarios, where some resources need to be allocated to available nodes, often taking into consideration various context-specific characteristics [21, 22]. The research community has come up with multiple algorithms, ranging in their computational complexity, completeness, preciseness, etc. Many of these approaches treat assignment as a collection of constraints, which need to be satisfied in order to find an optimal solution in the given circumstances [2, 5]. The SMT-based approaches are specifically popular and efficient due to their expressively and rich modelling language [6]. In this respect, a relevant approach that also makes use of SMT and Z3 Solver is described by Pradhan et al. [23]. The authors introduce orchestration middleware, which continuously evaluates available resources on Edge nodes and re-deploys software accordingly. Similar goal is pursued by Vogler et al. in [24], where authors report on a workload balancer for distributing software components at the Edge. Multiple approaches specifically focus on the autonomic and wireless nature of IoT devices and contribute to energy-efficient resource allocation, where the primary criterion for software deployment is energy efficiency [25]. A main obstacle for using SMT in practice is the gap between real platforms and the mathematical model, and our approach uses model-based techniques to bridge this gap.

Model-based techniques are often used to support DevOps. Combemale et al. [10] present an approach to use a continuum of models from design to runtime to accelerate the DevOps processes in the context of cyber-physical systems. Artavc et al. [3] uses deployment models on multiple cloud environments, which is a promising way to support the smooth transition of software from testing to production environments. Looking at approaches targeted at particular application domains, Bucchiarone et al. [8] use multi-level modelling to automate the deployment of gaming systems. In [13, 14], the authors apply model-driven design space exploration techniques to the automotive domain and demonstrate how different variants of embedded software are identified as more beneficial in different contexts, depending on the optimisation objective and subject to multiple constraints in place. To solve this optimisation problem, the authors also employ the SMT techniques and the Z3 solver implementation. Our approach focuses on a particular problem in the DevOps of IoT applications, and it goes beyond merely providing models as an effective interface for developers to conduct manual work, but looking at the full automation of a particular task, i.e., fleet deployment.

7 CONCLUSION AND FUTURE WORK

This paper described an industry-driven research aiming to enable the automatic fleet deployment of Edge computing systems. This is a joint research, and co-authors of this paper are from both a research institute and the industrial Edge application provider Tellu. We address the problem using model-based techniques including meta-modeling, transformation and constraint solving, and integrate the prototype tool into the DevOps toolchain currently used in Tellu. An initial demonstration with simulated DevOps iterations showed that the approach is useful in the DevOps process, by generating correct assignment results, and providing valuable feedback.

As an industry-driven research, the approach is specific to the concrete problem faced by Tellu: the meta-models and model transformation are designed for their Edge computing application and its underlying platform, and the constraints are specifically defined to reflect the company's requirements. Therefore, one of the main directions for future work is to generalize the approach across multiple Edge application providers and facilitate other companies in adopting it, through generic guidelines, templates, and tools for creating the meta-models, transformations and constraints.

Another future work direction is to introduce other techniques into fleet deployment. Under the current setting, developers need to know clearly what every context means to the assignment, in order to design the effective constraints. However, under some circumstances, the potential effect of certain characteristics on the assignment may not be so clear in the beginning, or this influence is fuzzy. For example, if our objective is to maximize the chance to receive good feedback on the preview version, we may need to consider a series of contexts, such as how often the patient uses the application, how active they are in providing feedback, how useful the feedback has been so far, etc. It is difficult to define black-white constraints on such context characteristics. We foresee two directions to address this issue. One possibility is to apply Semantic Web techniques to the meta-models, by providing a common unambiguous ontology about the contexts and the standard established way to define the semantics of contexts. Another direction is to combine constraint solving with machine learning. We use constraint solving only to ensure the assignment does not violate hard and soft constraints currently being in place. At the same time, we can also allow the tool to learn from historical records (i.e., what assignments have already been computed, and what feedback has been collected) in order to lead to a good assignment.

ACKNOWLEDGMENTS

This work is partially funded by the European Commission's H2020 Programme under grant agreement 780351 (ENACT) and the Norwegian Research Council's BIA-IPN programme no. 309700 (FLEET)

REFERENCES

[1] Amazon. 2020. AWS IoT Greengrass. https://aws.amazon.com/greengrass/. Accessed: 2020-05-20.
[2] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. 2011. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Ninth Symposium of Abstraction, Reformulation, and Approximation*.
[3] Matej Artač, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian A Tamburri. 2016. Model-driven continuous deployment for quality devops. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. 40–41.
[4] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.
[5] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. 2012. Solving constraint satisfaction problems with SAT modulo theories. *Constraints* 17, 3 (2012), 273–303.
[6] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. 2017. Satisfiability modulo theories and assignments. In *International Conference on*

Automated Deduction. Springer, 42–59.

[7] Randal E Bryant, Shuvendu K Lahiri, and Sanjit A Seshia. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *International Conference on Computer Aided Verification*. Springer, 78–92.

[8] Antonio Bucchiarone, Antonio Cicchetti, and Annapaola Marconi. 2019. Exploiting multi-level modelling for designing and deploying gameful systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 34–44.

[9] CHEF. 2020. CHEF INFRA. https://www.chef.io/products/chef-infra/. Accessed: 2020-05-20.

[10] Benoit Combemale and Manuel Wimmer. 2019. Towards a Model-Based DevOps for Cyber-Physical Systems. In *Software Engineering Aspects of Continuous Development*.

[11] Rustem Dautov and Hui Song. 2019. Towards IoT Diversity via Automated Fleet Management.. In *MDE4IoT/ModComp@ MoDELS*. 47–54.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[13] Johannes Eder, Andreas Bahya, Sebastian Voss, Alexandru Ipatiov, and Maged Khalil. 2018. From deployment to platform exploration: automatic synthesis of distributed automotive hardware architectures. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 438–446.

[14] Johannes Eder, Sergey Zverlov, Sebastian Voss, Maged Khalil, and Alexandru Ipatiov. 2017. Bringing DSE to life: exploring the design space of an industrial automotive use case. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 270–280.

[15] Carlos R Aguayo González, Carl B Dietrich, and Jeffrey H Reed. 2009. Understanding the software communications architecture. *IEEE Communications Magazine* 47, 9 (2009), 50–57.

[16] Google. 2020. Coral Edge TPU. https://coral.ai/. Accessed:2020-05-20.

[17] Daniel Hall. 2013. *Ansible configuration management*. Packt Publishing Ltd.

[18] Lorin Hochstein and Rene Moser. 2017. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, Inc.

[19] Microsoft. 2020. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/. Accessed: 2020-05-20.

[20] OMG. 2006. *Deployment and Configuration of Component-based Distributed Applications Specification, v4.0*. Technical Report. Object Management Group, Inc. https://www.omg.org/spec/DEPL/4.0/PDF

[21] Temel Öncan. 2007. A survey of the generalized assignment problem and its applications. *INFOR: Information Systems and Operational Research* 45, 3 (2007), 123–141.

[22] David W Pentico. 2007. Assignment problems: A golden anniversary survey. *European Journal of Operational Research* 176, 2 (2007), 774–793.

[23] Subhav Pradhan, Abhishek Dubey, Shweta Khare, Saideep Nannapaneni, Aniruddha Gokhale, Sankaran Mahadevan, Douglas C Schmidt, and Martin Lehofer. 2018. Chariot: Goal-driven orchestration middleware for resilient IoT systems. *ACM Transactions on Cyber-Physical Systems* 2, 3 (2018), 1–37.

[24] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. 2016. A scalable framework for provisioning large-scale IoT deployments. *ACM Transactions on Internet Technology (TOIT)* 16, 2 (2016), 1–20.

[25] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. 2016. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications* 16, 3 (2016), 1397–1411.

[26] Lintao Zhang and Sharad Malik. 2002. The quest for efficient boolean satisfiability solvers. In *International Conference on Computer Aided Verification*. Springer, 17–36.