Chapter 4

Model-based Continuous Deployment of SIS

By Nicolas Ferry, Hui Song, Rustem Dautov, Phu Nguyen and Franck Chauvel

DOI: 10.1561/9781680838251.ch4

Published in *DevOps for Trustworthy Smart IoT Systems* by Nicolas Ferry, Hui Song, Andreas Metzger and Erkuden Rios (eds.). 2021. ISBN 978-1-68083-824-4. E-ISBN 978-1-68083-825-1.

Suggested citation: Nicolas Ferry, Hui Song, Rustem Dautov, Phu Nguyen and Franck Chauvel. 2021. "Model-based Continuous Deployment of SIS" in *DevOps for Trustworthy Smart IoT Systems*. Edited by Nicolas Ferry, Hui Song, Andreas Metzger and Erkuden Rios. pp. 59–93. Now Publishers. DOI: 10.1561/9781680838251.ch4.

now
the essence of knowledge

4.1 Introduction

Smart IoT Systems (SIS) are characterized by the presence of software deployed along the entire IoT-Edge-Cloud continuum. Software defines the behaviour of the SIS, and such behavior keeps evolving during the entire system life cycle following the ever-changing system context. This evolution may be realized as self-adaptation (such as the use of online learning for dynamic adaptation, as elaborated in Chapter 6) or manual reconfiguration of the existing software, but, very often, it requires releasing new versions of software. On the one hand, this evolution characteristic of SIS is typically conflicting with the traditional IoT systems vision consisting of devices with immutable code, once deployed at the factories. There must be new approaches for supporting the evolution of SIS. DevOps, on the other hand, promotes the idea of continuously delivering new software updates. Indeed, the DevOps movement promotes an iterative and incremental approach enabling the continuous evolution of software systems. Embracing DevOps can support the continuous evolution of SIS and improve their trustworthiness (e.g., security). As an evolution of the DevOps movement, DevSecOps [30] promotes security as an aspect that must be carefully considered in all the development and operation phases for the continuous evolution of systems to be secure. However, how

to effectively deploy the software update to the computing continuum is a main obstacle to enable DevOps or DevSecOps for SIS as it requires the capability of continuous deployment of software at all levels.

Continuous and automatic software deployment is still an open question for SIS, especially at the Edge and IoT ends. The state-of-the-art Infrastructure as Code (IaC) solutions are established on a clear specification about which part of the software goes to which types of resources. This is based on the assumption that in the Cloud it is easy to obtain the exact computing resources as required. However, this assumption is not valid on the Edge and IoT levels. A typical SIS in production often contains hundreds or thousands of heterogeneous and distributed devices (also known as a *fleet of IoT/Edge devices*[1]), each of which has a unique context, while their connectivity and quality are not always guaranteed. The major challenges are as follows:

- How to automate the deployment of software on heterogeneous devices possibly with limited or no direct Internet access?
- How to manage variants of the software which fit different types or contexts of Edge or IoT devices in the fleet?
- How to ensure the trustworthiness of the deployed software whilst the quality of the underlying resources are not guaranteed?

In the ENACT project, we focus on the problem of automatic software development for SIS, and our research attempts to address these challenges resulted in two complementary prototype tools for the deployment of SIS at two different layers: (i) GeneSIS targets at the device layer, providing a unified way to deploy software on heterogeneous devices, including those without direct internet connection; (ii) DivEnact targets at the fleet layer, allowing developers to deploy software into the abstract fleet as a whole instead of focusing on concrete individual devices. The tool maintains the software variants and assigns them automatically to the devices according to their contexts. With trustworthiness (e.g., security) being a concern cross-cutting both layers, our tools provide solutions that contribute making the deployment and the SIS trustworthy. At the device layer, we support the specification and deployment of security and privacy mechanisms together with the SIS software in a DevSecOps fashion. Moreover, we provide the novel *rolling deployment* method to guarantee the availability of the deployed software as well as to handle errors during a deployment, i.e., in addition to the main software, we also deploy a

1. Similar to a fleet of vehicles in a transportation company, a fleet of devices are owned by the same application providers and distributed to different places or users. Devices in a fleet conduct relatively independent tasks, whilst coordinated by the application provider from a global perspective.

backup copy which will replace the main one when necessary without delay. At the fleet level, we maintain the software diversity within the fleet for security purposes.

Model-Driven Engineering (MDE) is the scientific basis underlying both GeneSIS and DivEnact. MDE is a branch of software engineering that aims at improving the productivity and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. It has shown to be effective in supporting design activities [42]. This approach, which is commonly summarised as "model once, generate anywhere", is particularly relevant to tame the complexity of developing heterogeneous systems such as SIS. Models and modelling languages as the main artefacts of the development process enable developers to work at a high level of abstraction by focusing on deployment concerns rather than implementation details.

This chapter is organized as follows. Section 4.2 provides an overview of the current state of the art and of the practice for the automatic deployment of SIS. Section 4.3 introduces our solutions for the automatic deployment of SIS, first describing how they can be integrated in order to form a coherent deployment bundle and then detailing each our two enablers: GeneSIS and DivENACT. Section 4.4 focus on the support offered by our solutions to ensure the trustworthiness deployment of SIS. Finally, Section 4.5 draws some conclusions.

4.2 The State of the Art

Software deployment has been evolving from deployment of component-based commercial desktop software [39], deployment of component-based distributed applications [25], to deployment on Cloud resources, and more recently deployment for IoT systems along the entire IoT-Edge-Cloud continuum. Even though some core concepts from deployment of component-based applications such as *capability*, *port* in [25] can be inherited for deployment on Cloud or IoT resources, they need to be tailored and customized to fully address the specificities of these environments.

4.2.1 On the Deployment at the Device Layer

For some years now, multiple tools have been available on the market to support the deployment and configuration of software systems, e.g., Puppet,[2] Chef.[3] These tools were first defined as configuration management tools aiming at automating

2. https://puppet.com/
3. https://www.chef.io/chef/

the installation and configuration of software systems on traditional IT infrastructure. Recently, they have been extended to offer specific support for deployment on Cloud resources. Meanwhile, new tools emerged and were designed for deployment of Cloud-based systems or even multi-Cloud systems (i.e., systems deployed across multiple Clouds from different providers) such as CloudMF [19], OpenTOSCA [43], Cloudify,[4] and Brooklyn.[5] Those are tailored to provision and manage virtual machines or PaaS solutions. In addition, similar tools focus on the management and orchestration of containers, e.g., Docker Compose,[6] Kubernetes.[7] As opposed to hypervisor virtual machines, containers leverage lightweight virtualization technology, which executes directly on the operating system of the host. As a result, the container engine shares and exploits a lot of resources offered by the operating system thus reducing containers' footprint. These characteristics make container technologies suitable not only for the Cloud, but also for Edge devices [13].

Besides, a few tools, such as Resin.io (Balena)[8] and ioFog,[9] are specifically designed for the IoT. In particular, Resin.io provides mechanisms for (i) the automated deployment of code on devices, (ii) the management of a fleet of devices, and (iii) the monitoring of the status of these devices. Resin.io supports the following continuous deployment process. Once the code of the software component is pushed to the Git server of the Resin.io Cloud, it is built in an environment that matches the targeted hosting device(s) (e.g., ARM for a Raspberry Pi) and a Docker image is created before being deployed on the hosting device(s). However, Resin.io offers limited support for the deployment and management of software components on tiny devices that cannot host containers.

Regarding the deployment of elements of hardware and software that are to operate in harmony within a networked system, the Software Communications Architecture (SCA) [1] and IoT deployment share some basic concepts. The SCA is an open architecture that specifies a standardized infrastructure for a software-defined radio (SDR). However, the SDR SCA specification requires an SCA-compliant system for elements of hardware and software to operate within. In other words, the SCA is tightly tied to the specific needs for standardizing the development of SDRs, which is much less heterogeneous than the IoT domain in terms of

4. http://cloudify.co/
5. https://brooklyn.apache.org
6. https://docs.docker.com/compose/
7. https://kubernetes.io
8. https://www.balena.io/
9. https://iofog.org/

communication means, systems of systems, which may span all the layers of Cloud, Edge, IoT devices. Moreover, the SCA does not have any concept about supporting the deployment on devices not directly accessible.

In [34], we conducted a systematic literature review (SLR) to systematically study a set of 17 primary studies of orchestration and deployment specifically for the IoT. We found a sharp increase in the number of primary studies published in two-three recent years. We also found that most approaches do not really support the IoT deployment and orchestration at low-level IoT devices. As for the continuous deployment tools mentioned before, these approaches mainly focus on the deployment of software systems over edge and Cloud infrastructures whilst little support is offered for the IoT space. When this feature is available, it is often assumed that a specific bootstrap is installed and running on the IoT device. A bootstrap is a basic executable program on a device, or a run-time environment, which the system in charge of the deployment rely on (e.g., Docker engine). Approaches such as Calvin run-time [28], WComp [27], or D-LITE [10], D-NR [24] all rely on their specific run-time environment where mechanisms such as dynamic component loading or class loading are typically used. There is a lack of addressing the trustworthy aspects and advanced support in the deployment and orchestration of the IoT.

To the best of our knowledge, none of the approaches and tools aforementioned have specifically been designed for supporting deployment over the whole IoT, Edge, and Cloud infrastructure. In particular, they do not provide support for deploying software components on IoT devices with no direct or limited access to internet. In addition, we also identified they do not offer support for including security concerns as core concepts in the tool and/or language.

4.2.2 On the Deployment at the Fleet Layer

While all these solutions discussed above are focusing on the deployment of a software system, they typically do not offer specific support for the management of a fleet of devices or a fleet of systems, which basically consists in managing large set of deployments with those solutions.

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, but lack the support for distributing multiple variants across a large fleet. They also do not provide sufficient automated support for updating devices with constrained resources and limited (or none) Internet connectivity [34]. Such embedded and microcontroller-enabled devices traditionally have been flashed with 'one-off' firmware not intended to be updated in the future, but they are not often seen as active contributors to the common pool of shared computing resources, which

can be iteratively assigned and deployed with updated firmware. This has also led to the so-called concept of IoT-edge-cloud computing continuum, where computing and storage tasks are distributed across all three levels. On the other hand, the mainstream IoT/Edge fleet management platforms offer tools to maintain multiple deployments, the fleet of devices, and their contexts, but developers still need to manually designate which deployment goes to which device.

Another relevant reference architecture for deploying component-based applications into heterogeneous distributed target systems is described in [37]. 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. Software diversity is a new dimension of architecture-level properties, which is both a result of the hardware heterogeneity and a method toward more secure system. The fleet deployment approach provides a implementation-level support to our theoretical approaches towards a more diverse software [29, 45].

The assignment problem (such as assigning software components to the devices in an IoT fleet) frequently appears in ICT scenarios, where some resources need to be allocated to available nodes, often taking into consideration various context-specific characteristics [38, 40]. 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, 7]. The approaches based on Satisfiability Modulo Theories (SMT) are specifically popular and efficient due to their expressively and rich modelling language [8]. In this respect, a relevant approach that also makes use of SMT and Z3 Solver is described by Pradhan *et al.* [41]. 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 [46], 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 [47]. A main obstacle for using SMT in practice is the gap between real platforms and the mathematical model.

Model-based techniques are often used to support DevOps. Combemale *et al.* [12] 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.* [9] use multi-level modelling to automate the deployment of gaming systems. In [16, 17], 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.

4.3 Overview of the ENACT Deployment Bundle

The ENACT approach to automatic software deployment is implemented as a prototype deployment bundle with two enablers, i.e., GeneSIS and DivEnact, supporting automatic deployment at the device and fleet layers, respectively.

Figure 4.1 illustrates how the ENACT deployment bundle is used in a typical SIS. The illustrative SIS has six subsystems, each of which is in charge of a particular business task, such as serving a user, monitoring a room, etc. Such a subsystem is usually composed by at least one edge device and several IoT devices such as sensors and actuators. For the sake of simplicity, we do not show all the IoT devices. These subsystems form the fleet of this SIS. Since each subsystem contains one main edge device as the main contact point, or gateway with the back-end service, we also refer to such fleet as an *edge fleet*. A fleet is normally distributed, with the edge devices (together with its IoT devices) serving different customers or tenants, and deployed in different locations. The developers often maintain one or several edge devices at their own premises for testing or trial purposes.

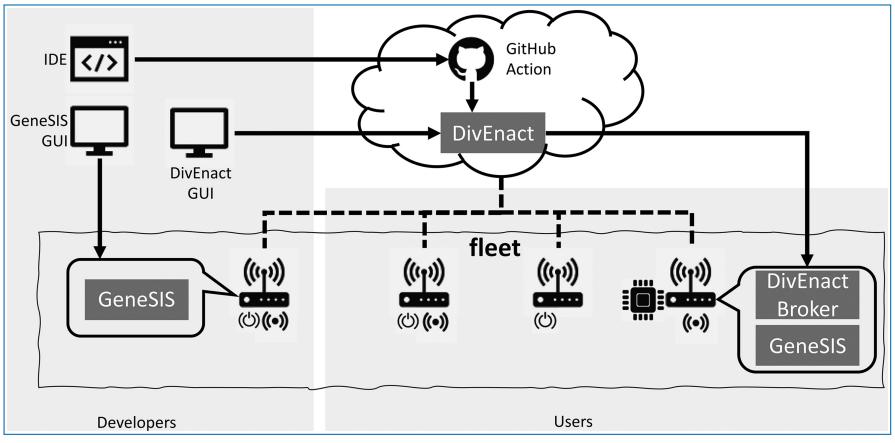

Figure 4.1. The ENACT deployment bundle.

GeneSIS supports the automatic deployment within a local subsystem, for example the deployment on the devices located on the developers' side. In such case, the developers can directly interact with the GeneSIS engine hosted on the local edge device, and use it as the bridge to further deploy required code to the associated IoT devices. In the development phase, developers define a *deployment model* in the GeneSIS modelling language specifying which software artefacts should be deployed onto which devices. Once the development phase completed, in the deployment phase, the same deployment model, or a slightly modified one, will be provided to the GeneSIS deployment engine, running either on a local machine or the edge device. The engine will install or update the software artefacts according to the deployment model.

DivEnact handles a different automatic deployment problem at the fleet level. When the developers want to release the new version of their application to production, they need to deploy software artefacts to all the devices on the users' sites. They cannot extend the deployment model to include every device in the fleet, because such a huge model is not maintainable, especially when the devices keep joining and exiting the fleet. Instead, since each user has a subsystem similar to the one at the developers' side, the developers can provide the deployment models they developed in the previous phase for the local subsystem to DivEnact. The latter maintains the list of all subsystems, and sends the deployment model to the devices before invoking the GeneSIS engine running on the edge device of the subsystem, to eventually deploy the software artefacts according to the deployment model. Within a fleet, the subsystems have different contexts, such as the device capacity, the connectivity, the user preferences, etc., and developers need multiple variants of their software to fit different contexts. DivEnact accepts multiple deployment models representing different software variants and configurations, coming for a series of releases, and automatically assign them to the proper subsystems. For the sake of availability, we recommend running the main service of DivEnact in the Cloud, with a light-weight DivEnact broker running on edge devices of each subsystem.

Next, we present GeneSIS and DivEnact, detailing their main innovations as well as how they contribute ensuring the trustworthiness of SIS.

4.3.1 GeneSIS

GeneSIS enables the continuous orchestration and deployment of Smart IoT Systems throughout the IoT-Edge-Cloud continuum. Given a description of a deployment topology, GeneSIS deploys and configures the needed software components, by connecting to the hardware (or software) nodes. This topology, the so-called deployment model, only prescribes what components must be deployed, how a single component can be deployed, and how they connect to each other. GeneSIS

automatically derives how to deploy them. Therefore, GeneSIS is composed of two key components: (i) a domain-specific modelling language for specifying deployment models, and (ii) an execution engine to enact the provisioning, deployment and adaptation of a SIS. We refer the reader to [20] for more details about the GeneSIS modelling language.

The target user groups of GeneSIS are mainly DevOps engineers, software developers, and software architects. The GeneSIS modelling language has been conceived so the deployment model can act as a touch point between development and operation activities. DevOps teams can use it to deploy either in development, staging or production environments. It is also worth noting a deployment model written using the GeneSIS modelling language is independent of the underlying technologies, i.e., GeneSIS can deploy components anywhere in the IoT-Edge-Cloud continuum: from microcontrollers without direct Internet access to virtual machines running in the Cloud.

The main task of the GeneSIS deployment engine is to reconcile two views of the system: the deployment model given by the user, and the current state of the running infrastructure, assuming that software components may already be running on the infrastructure, for example, as a result of a system upgrade. To reconcile these two views, the GeneSIS deployment engine adheres to the "models@runtime" architectural pattern [6]. It compares these two views and deduces what changes the adaptation engine must carry out on the running infrastructure to align it with the prescription, i.e., the deployment model given by the user. After the deployment, the engine synchronizes the current GeneSIS model with the actual deployment result. Such synchronization will ensure that all the tools in future DevOps cycles will leverage an up-to-date deployment model.

The GeneSIS deployment engine is non-invasive, meaning it does not require any GeneSIS bootstrap or agent running on a target device to deploy software on it. However, when decided by the DevOps engineer, the GeneSIS deployment engine can deploy on a target device a monitoring agent. This agent is an instance of netdata[10] and provides information about the performance and health status of a device, including data about software components it hosts.

Finally, the deployment engine can delegate parts of its activities to deployment agents running in the field. It is not always possible for the GeneSIS deployment engine to directly deploy software on all hosts. For instances, tiny devices do not always have direct access to the Internet or even the necessary facilities for remote access (in such case, the access to the Internet is typically granted via a gateway) or for specific reasons (e.g., security) the deployment of software components can

10. https://github.com/netdata/netdata

only be performed via a local connection (e.g., a physical connection via a serial port). In such case, the actual deployment of the software on the device has to be delegated to the gateway locally connected to the device. The GeneSIS deployment agent aims at addressing this issue. It is generated dynamically by GeneSIS based on the artefact to be deployed and its target host, and is implemented as a Node-RED application. We refer the reader to [20, 21] for more details.

GeneSIS comes with a set of predefined component types that can be seamlessly instantiated in deployment models. In addition, GeneSIS embeds a plugin mechanism that enables the dynamic loading of new component types. A components type repository is scanned by the GeneSIS execution engine before each deployment ensuring all available types are loaded before a deployment model is analyzed and deployed.

4.3.2 DivEnact

While GeneSIS focus on the deployment of a single system, DivEnact, the diversity-oriented fleet deployment enabler, is an implementation of our concept of *fleet deployment*. Fleet deployment is an automatic software deployment support for IoT/Edge applications, which allows developers to deploy software artefacts onto a fleet of devices as an abstract whole, without concerning about the concrete devices and their contexts in the fleet. The automatic fleet deployment tool, such as DivEnact, will maintain the devices and their contexts in the fleet, the software variants, and assign the variants to the appropriate devices depending on their contexts.

DivEnact utilizes Azure IoT Edge to maintain a list of edge devices, together with their contexts and run-time status. Developers provide DivEnact with a set of deployment models (typically GeneSIS models), each of which specifies a particular software artefact, together with the specification about how to configure and deploy it on an Edge device. In order to facilitate the definition of similar deployment models, we also introduce the concept of deployment *templates* and *variants*. A template defines the common parts among a number of deployment models, and a variant further instantiates the template as a deployment model. A common use case for this is to define a deployment model for a particular software, and then use variants to represent the different versions of this software. After receiving all the deployment models, DivEnact automatically assigns them to the list of edge devices, and enacts the deployment model on each edge devices to finalize the local deployment.

Figure 4.2 illustrates the technical architecture of the DivEnact tool. The DivEnact tool is designed and implemented following the established Model-View-Controller (MVC) design pattern for client-server application systems.

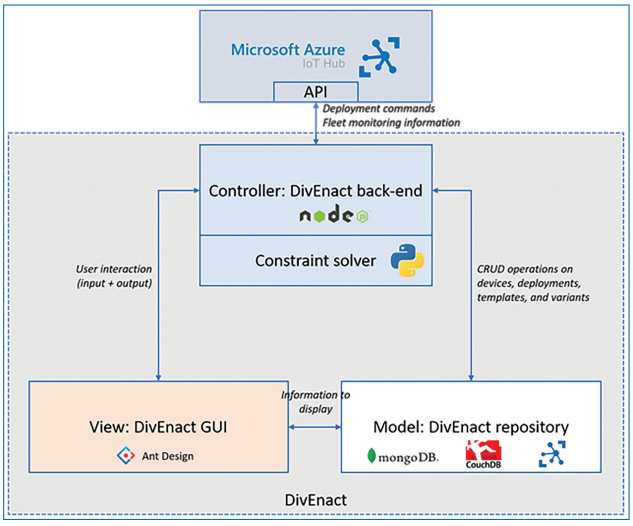

Figure 4.2. The architecture of DivEnact.

The DivEnact knowledge base stores various deployment- and fleet-related artefacts and is modified by the DivEnact back-end upon the user input received through the graphical user interface. The modification actions (CRUD – create, read, update, delete) are implemented on top of standard APIs and libraries. The model itself is spread across the following three repositories. MongoDB database is installed locally, along with a DivEnact instance, and serves to store information about templates and variants unique to each application system. There is a centralised repository in CouchDB for storing various ENACT artefacts, including deployment models used by DivEnact. In particular, the CouchDB database stores previously designed GeneSIS deployment models that are to be enacted on low-level IoT devices as part of the "last mile deployment". Azure IoT Hub Cloud portal keeps track of registered devices in the fleet and existing deployments. The information obtained from the hub reflects the current state of all the devices through continuously updated digital twins, as well as deployments applied to these devices (i.e., software modules currently deployed and running on each device).

The DivEnact graphical user interface remains the main point of interaction with the user. The main functionality is structured across several functions, i.e., the editing and maintenance of templates, variants, deployment models, devices and the assignment.

The back end of the DivEnact tool is implemented in Node.js. It receives RESTful requests originating from the user's graphical interface and manipulates the data model accordingly. It also interacts with the Azure IoT Hub API to update some information about the devices in the fleet and trigger deployments. The back

end also implements the actual diversification functionality (described in the next subsections) by receiving the input model from the user and passing it to the underlying Python script. Upon execution, the calculated solution is passed back to the user for the final approval.

The main function of the back end is the automatic assignment of deployment models into the list of devices, considering the constraints, the deployment preferences, the resource optimization, etc. We have implemented two experimental assignment approaches, using constraint solving and resource assignment theories as the back end mechanisms. The details of these two approaches can be found in our recent publications [15, 44].

4.4 Trustworthy Deployment

As explained before, ensuring the trustworthiness of the deployment and of the SIS is critical and challenging. It is a concern that crosscuts both the device or at fleet layers. In the following we detail how GeneSIS and DivEnact help addressing this challenge. Our effort is concentrated on three complementary directions, i.e., how to increase the availability of deployed system; how to automatically deploy the required security mechanisms together with the application; and how to maintain the software diversity across the whole fleet.

4.4.1 Deploying Availability Mechanisms

Availability refers to "the ability of the system to mask or repair faults such as the cumulative service outage period does not exceed a required value over a specified time interval" [4, p. 174]. Availability is a primary concern for business stakeholders because service interruptions often translate into money loss. The failure of an electricity meter for instance may affect the capacity of the electricity company to properly bill its customers.

Availability, as any extra-functional requirements, does not affect the system function, but rather affects its architecture. Building high-availability systems requires additional components to detect, repair, or even prevent faults, such as monitors, watchdogs, replicas, or voting mechanisms to name a few. Availability tactics are now well documented, so we refer the reader to [4, Chap. 5] for an introduction.

Many things can go wrong in Smart IoT Systems, including incorrect algorithms, network failure and delays, hardware failure, etc. In the following, we focus on scheduled outages, which are interruptions of service needed because of software upgrade, and internal faults, which are faults that occur because of defects

in the source code of the components. In other words, the availability support we present hereafter contributes (i) improving trustworthiness of a SIS by maximising its availability (by minimizing downtime during upgrades); and (ii) improving deployment trustworthiness by modifying a system and its deployment only if the deployment process is successful (i.e., old version of a software is removed only if the new version is up and running).

We extended GeneSIS with the ability to deploy mechanisms that cope with these two kinds of fault. To mask internal faults, GeneSIS deploys multiple instances of the same service/component (so called replicas) behind a proxy. When one replica fails, the proxy can query another replica. To mask scheduled outages, GeneSIS provides zero-downtime upgrades. We leverage the same architecture and deploys the new version (behind the proxy) before to decommission the older one. That way, there is always a replica available and upgrades do not affect availability.

Modern execution platforms such as Docker or AWS already implement various availability mechanisms, and, they have strategies for both scheduled outages and fault-tolerance. Docker Swarm for instance performs zero-downtime upgrades by deploying new services instances before to decommission the older ones. The challenge is that, from a deployment perspective, the availability tactics are tightly coupled to the underlying execution platform. Changing platform requires changing the deployment configuration.

To decouple availability from deployment platform, GeneSIS captures these deployment tactics independently of the underlying platform. If the platform already provides mechanisms (such as Docker Swarm), GeneSIS uses those, otherwise it deploys built-in components to implement the selected tactics. In the following, we illustrate three scenarios that show how GeneSIS copes with scheduled outages and internal faults.

1. Initial Deployment: GeneSIS deploys the system following the availability tactics selected by the user.
2. Internal Fault: A fault occurs in the system and we explain how the mechanisms that GeneSIS has deployed deal with that fault, so that it is not visible to the end-user.
3. Zero-downtime Upgrades: The user requests the deployment of a new version of the system and we illustrate how GeneSIS leverage the underlying mechanisms to minimize service disruption.

4.4.1.1 Using built-in components on top of docker

By default, GeneSIS does not make any assumption of the capability of the platform where it should install a component. It could be a very fully featured platform such as Docker Swarm (see Section 4.4.1.2) or simply an operating system offering

remote access (through SSH, Telnet, etc.). We detail here the later case, that is when the host is a bare OS. Recall that GeneSIS uses two strategies to improve availability: Replication to deal with internal faults, and zero-downtime deployment to deal with scheduled outages. To implement these two strategies, we need three capabilities that are provided by additional components:

- *Routing*, that is, the capability of redirecting incoming traffic to a selected replica. Network proxies provide this and in GeneSIS, we selected Nginx.
- *Error detection*, that is, the capability to proactively detect replicas that have failed (for whatever reasons). We used a watchdog, that is a component that periodically connects to the replicas and runs a so-called "health check". The health check is an application specific behaviour that confirms that the replicas is up and running. It could be requesting a predefined resource using HTTP, checking the status or OS-level services, or any other "quick-check". In GeneSIS, we have implemented simple watchdogs using Shell scripts and CRON tasks.
- *Spatial isolation*, that is, the capability to deploy multiple instances of the same application with guarantees that they can access external resources (network port, files on disks, etc.) without stepping on each other. GeneSIS uses containers (i.e. Docker in the current implementation) to ensure spatial isolation of replicas, but other container technologies such as LXC apply.

Scenario 1: Initial Deployment

The first step is for GeneSIS to ensure that the underlying host offers "spatial isolation" guarantees. To do so, GeneSIS first installs Docker as container offer such guarantees. Figure 4.3 below illustrates how GeneSIS interacts with the host to install docker and to create a "replicable image" of the software stack.

Given a component to deploy, GeneSIS first connects to the host through SSH and installs Docker (Step 1). Then GeneSIS configures Docker in remote mode so that other components (including itself) can access it through the network. Then, GeneSIS creates a new temporary container (by default, using the image "debian:10-slim") and installs the underlying software stack. To do this, GeneSIS traverses the underlying software stack and installs all underlying components by triggering the associated SSH commands into the container (Step 6, 7 and 8). Once the stack is installed and configured, GeneSIS converts it to a separate Docker image, that it later uses to install multiple replicas (Step 9). Finally, GeneSIS destroy the temporary container. At this stage, GeneSIS has enforced spatial isolation, and can then proceeds with replication and zero-downtime upgrades.

Once Docker is operational and the component to install is available as a Docker image, GeneSIS proceeds with the two remaining capabilities, namely

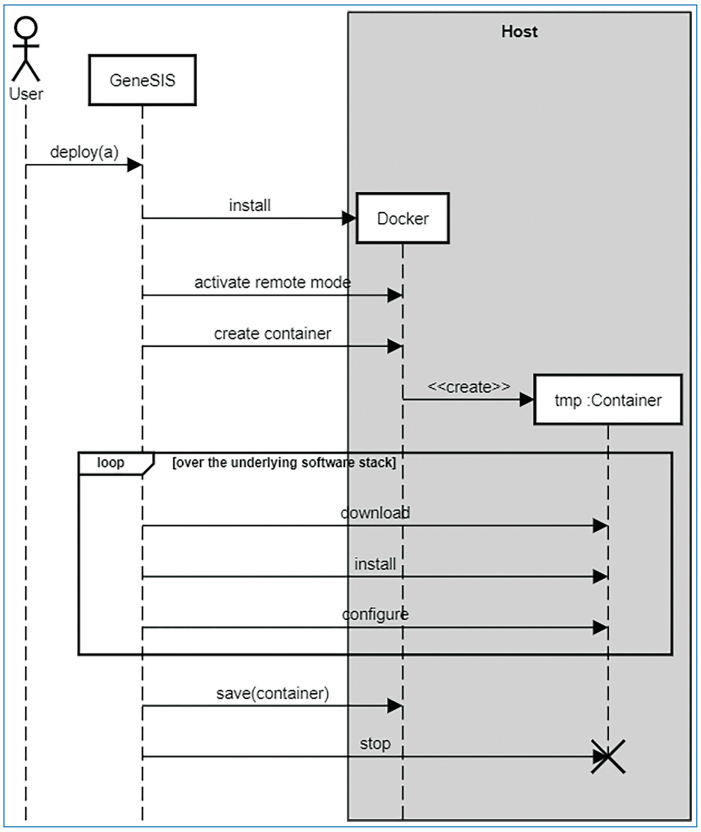

Figure 4.3. Automatically converting SSH resources into a Docker image. GeneSIS connects to the host, and execute all SSH commands into a new container, which it then saves as a new "ready to use" image.

detecting errors and routing as shown on Figure 4.4. First GeneSIS installs the proxy component through Docker (Step 1 and 2). Then, it installs the watchdog and configures it with the endpoints of the Docker host, the proxy and with the number of replicas to maintain (Step 2 and 3). For each missing replica, the watchdog requests Docker to provision a new instance of the image built in Scenario 1 and then the watchdog start checking the health of each replica periodically (Step 6 and 7). As soon as a replica is detected as healthy, the watchdog registers it to the proxy (Step 8), which uses it process user requests (Step 9 and 10).

Scenario 2: Fault tolerance

We now turn to the second scenario where one replica fails and we explain on Figure 4.5 how the watchdog detects and reacts to such a failure. The main mechanism to detect failure is the health check. Since a health check is an application-dependent behaviour, the user must provide it as a script to be executed periodically

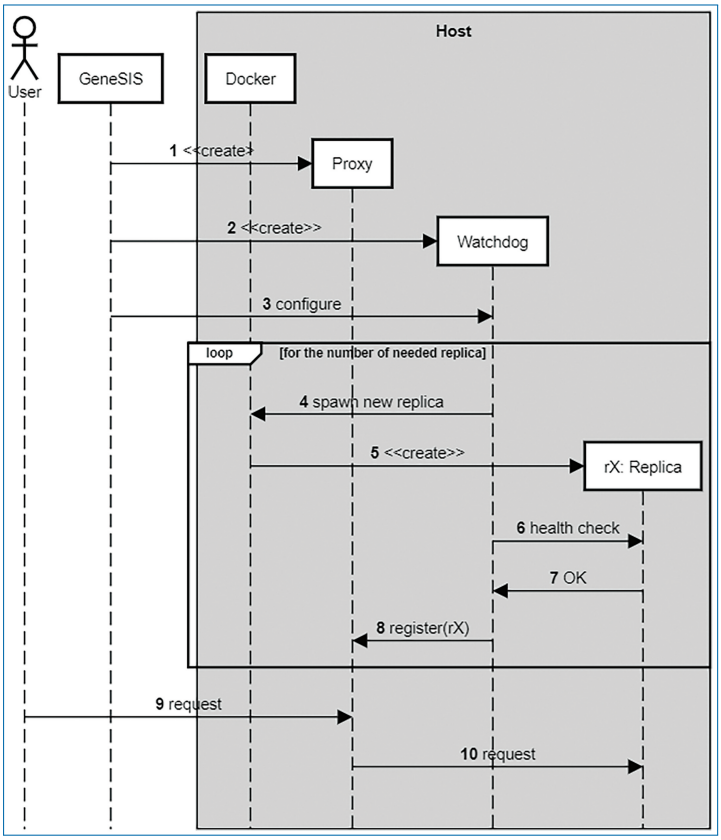

Figure 4.4. Configuring watchdogs and proxies to improve availability.

by the watchdog. GeneSIS defines the interface of the health check script as follows. The health check must accept the endpoint of the replica to query as its sole input parameter and must output the replica status in return through its exit code: Zero if the replica is healthy and any other value otherwise. This gives the user the capability to integrate any application-specific health check logic. The listing below shows one such health check script based on the HTTP status code, returned by a service.

```
#!/bin/bash
ENDPOINT="${1}"
response=$(curl --write-out '%{http_code}' --silent --output /dev/null "
    ${ENDPOINT}")
if [ "${response}" != 200 ]
then
    exit 1
fi
```

Listing 4.1. A Sample health-check script.

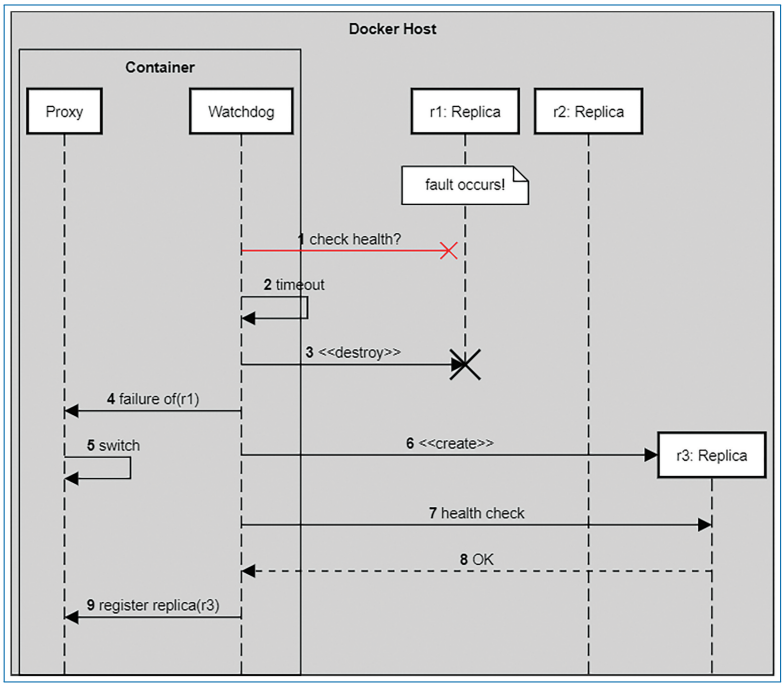

Figure 4.5. Masking internal faults to improve availability.

Note that this architecture can only detect replicas' failure as fast as the watchdog waits between two health checks. Besides, for the failure to be invisible to the user, there must at least two replicas, for the proxy to switch between them as soon as a delegation fails. Finally, transient phenomena such as network delays may be mistaken for replica failures and lead to unnecessary starts and stops of the container.

Scenario 3: Zero-downtime Upgrades

Finally, GeneSIS leverages these proxy and watchdog to guarantee zero-downtime upgrades, as shown on Figure 4.6.

When the user requests an upgrade, that is the deployment of a new version, GeneSIS first builds a new docker image of the software stack, including this new version. We described this process in Figure 15. Once this new image is ready, GeneSIS request the watchdog to perform the upgrade (Step 2). The watchdog thus provisions new instance of the new version (Steps 3 and 4) and, once these new replicas are operational, the watchdog registers them to the proxy (Steps 4, 5, 6 and 7). At that stage, incoming requests from the user are still delegated to the older version (Steps 8 and 9). Only once all replicas of the new version is operational, then the watchdog starts to decommission the older versions (Steps 10 and 11).

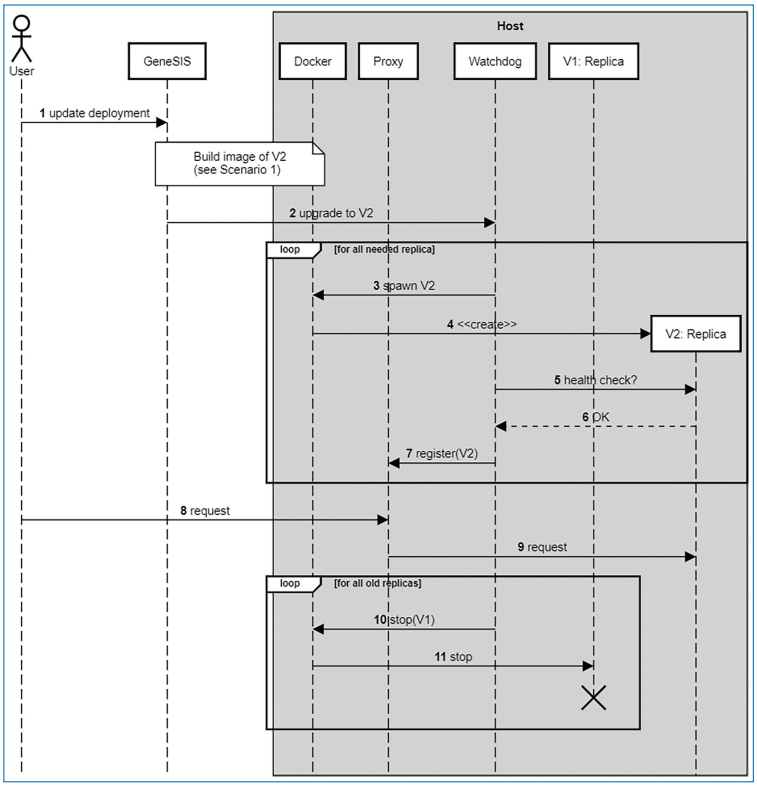

Figure 4.6. Using proxy and watchdog to guarantee zero-downtime upgrades.

4.4.1.2 Using docker swarm

In many cases, developers do not choose the platform on which their software runs: It may result from organizations' policies, customer requirements, etc. Platform such as Kubernetes, Docker Swarm or Rancher for instance all implement availability tactics, including replication and zero-downtime releases. Should the user use such platform, GeneSIS can exploit these native features to ensure fault-tolerance and zero-downtime upgrades. Docker swarm already implements routing among multiple replicas and fault detection. GeneSIS therefore delegates these features to Docker Swarm. We briefly example how GeneSIS handles our three scenarios using Docker Swarm.

Scenario 1: Initial deployment

This step is the simplest as we assume here that the host already runs Docker Swarm and that it therefore already guarantees spatial isolation. Here, GeneSIS simply requests Docker Swarm to deploy the given number of replicas of a given Docker image.

Scenario 2: Fault tolerance

This is also fully transparent from the GeneSIS standpoint. When GeneSIS delegates the deployment to Docker Swarm it specifies the number of replicas and the health check script to be used to detect faults. It is docker swarm that periodically checks the replicas status, provisions new ones if some have failed and route incoming requests accordingly.

Scenario 3: Zero-downtime redeployment

Further Docker Swarm also offers various strategies to upgrade a service (i.e., the set of replicas in Docker Swarm parlance). Among many options, Docker Swarm lets the user specify the "update order". If this order is "stop-first", then Docker Swarm first stops all the replicas of the older version, and only then starts provisioning replica for the new version. By contrast, if the update-order is "start-first", then Docker Swarm provisions all new replicas before to decommissions, as GeneSIS would do without Docker Swarm (see Figure 18). This "start-first" option let Docker Swarm minimize service interruptions.

4.4.1.3 Limitations

The support for availability tactics in GeneSIS is limited to Docker platform, although the general principle applies regardless of the underlying technology and other can extend GeneSIS and support other technologies. In addition, there are other types of faults that the current tactics cannot deal with. Hardware failure for instance would take down the whole host and therefore all the replicas at once. To tackle hardware failure, replication would have encompassed hardware, but this goes beyond GeneSIS whose mission is to provide platform agnostic deployment. Nevertheless, using the ENACT framework, the Root Cause Analysis enabler can be used to monitor and identify such failures and DevOps engineers can use GeneSIS to migrate the software components on a new host, benefiting from its platform independence. Programming faults are also not dealt with. Because GeneSIS is oblivious to the inner working of the components it deploys, all replicas are similar and fail in a similar manner. For instance, if a defect in the code lead to a fault of one replica (say because of invalid user input), then all replicas will exhibit this fault. Only diversification techniques [5] could help having replicas whose behaviours differ from one another, and that exhibit different failure profiles.

Improving availability from a pure deployment perspective, as GeneSIS provides, is bound to stateless components that can be easily replicated. Replicating a component that persists state requires some modification of its code. Either we separate its state from its application logic (using a local database, for instance) and

we ensure that all replica can access this single data source. Alternatively, each replicas also have its own local copy but we must now define a strategy to ensure the correct and timely synchronization of the multiple copies of the state, and possible conflicts. Modern database engines offer such mechanisms and would need to be integrated with GeneSIS in an ad-hoc manner. Edge platforms however often only pass data further on to Cloud services, and are thus likely to be stateless, or can simply leverage a local database as a cache, a strategy that the GeneSIS availability mechanisms handles.

Finally, on an Edge platform there are resources that cannot be replicated and that would require further investigation. A serial link for instance cannot be shared between replicas and, in this case, dedicated, application-specific logic must be in place to ensure consistent behaviour between all the replicas.

4.4.2 GeneSIS for Continuous Deployment Supporting DevSecOps

GeneSIS empowers a DevOps team to cope with security and privacy concerns of SIS as it natively offers support for including, as part of the deployment models, concepts to express security and privacy requirements and for the automatic deployment of the associated security mechanisms [20]. More importantly, GeneSIS enables the continuous enhancement of security controls in a DevSecOps cycle to keep security mechanisms up-to-date and well-aligned with the evolution of SIS, as well as addressing IoT security risks that are always evolving.

In this sub-section, we present the latest development of GeneSIS for better supporting the continuous deployment and enhancement of security controls that can refine or override the associated (default) security and privacy mechanisms of the IoT platforms such as SMOOL [36] or FIWARE [11]. Such security mechanisms are further elaborated in Chapter 7. More specifically, GeneSIS provides a generic way for a DevOps team to extend such existing security mechanisms with other (third-party) security mechanisms to provide enhanced security controls in a DevSecOps fashion.

4.4.2.1 GeneSIS for the specification and deployment of security components

To better support DevSecOps, GeneSIS promotes specifying security mechanisms as explicit elements in the deployment model, instead of hidden (and thus tightly coupled) in the source code, so that developers can see and change the security mechanisms in the deployment model level. This includes specifying security requirements and capabilities, and supporting the deployment of security mechanisms as components reusable in different scenarios. Compared to the

previous GeneSIS version reported in [21], we have built a new library of off-the-shelf security components that can be selected for instantiating in the deployment model. More importantly, we provide DevOps teams with mechanisms to configure security components and inject fine-grained security policies into deployment components (without modifying their business logic), enabling their seamless integration with third party security mechanisms (services, libraries, etc.). These supports can ease the development, integration, and deployment of SIS with continuously enhanced security mechanisms (see Section 4.4.2.2).

GeneSIS supports the deployment of security components as any other software components in the way that their deployment and configuration can be defined via exposed APIs and configuration files. A security component to be deployed together with an IoT application can be declared in GeneSIS with "security capabilities" in a provided port. A required port of a software component that requires a matching security capability can be bound with the provided port of the security component that provides such security capability. Before enacting a deployment, the GeneSIS deployment engine validates the correctness of the provided deployment model. In particular, it ensures that the required "security capabilities" match the provided ones.

GeneSIS allows specifying the deployment of security mechanisms and policies built on top of IoT platforms. We present here its application to the SMOOL IoT platform, which is used in our ENACT project. Similar approach can be applied to other IoT platforms. At the development phase (as well as at the deployment phase presented below), GeneSIS provides the support to relieve developers from manually specifying and maintaining security monitoring and control mechanisms in the code of a SMOOL client. Instead, a developer can define its own SMOOL client, focusing on its business logic. We integrated the SMOOL client wizard with ThingML[11]. As a result, a single Eclipse IDE can be used to generate the code of a SMOOL client, which can then be directly used as part of a ThingML program. The proper Maven manifests are automatically created facilitating the building and release of the desired application. This means that the DevOps team can quickly develop the business logic of the SIS based on the SMOOL platform, including necessary security mechanisms. DevOps teams can define SMOOL clients that leverage built-in security properties to check and enforce security concepts on messages requiring security controls.

The SMOOL's default security enforcement can be done with the SMOOL clients built-in security metadata checker to verify messages exchanged

11. https://github.com/TelluIoT/ThingML

among them. In cases where a deeper control is needed, a specialised security metadata checker can be included in SMOOL clients, with additional privileges to watch and process the security metadata in messages exchanged, in the same way it is done with business logic concepts such as sensed temperature or gas values. This provides a fine-grained control on critical messages that may have a significant security impact in the IoT system such as orders to actuators. More precisely, a client code can conduct security checks based on policies to be fulfilled by ontology concepts by using any of these options: (i) the default security metadata checker (for minimal configuration), (ii) a custom security metadata checker implemented in the development phase (for full control of security), and (iii) a custom security metadata checker for integration with external security services. Whatever security options, GeneSIS provides support for easily configuring the security mechanisms and how they should be integrated and deployed with the SIS. Thanks to ThingML, GeneSIS provides advanced support for the three options.

To support the first option, GeneSIS enables the DevOps team to specify explicitly the default security policy that must be enforced by the Security checker. To support the second option, where the DevOps team can implement its own ad-hoc security checker, GeneSIS provides the means to automatically inject this security checker into the code of the component to be deployed and to rebuild the component automatically. More precisely, when deploying this security component, the GeneSIS deployment engine injects the security policy into the ThingML code of the SMOOL client. This code injection is done before GeneSIS triggers the compilation of this code to generate the actual implementation of the SMOOL client with the corresponding security policy.

To support the third option, GeneSIS not only injects the security checker code that integrates with a third party security solution (e.g., Casbin[12] or the Context-aware Access Control mechanism [23], or a "gatekeeper" in [33]) but it can also deploy the latter. At the deployment phase, a SMOOL client can be deployed by GeneSIS as any other software components. Once the SMOOL client has been developed, the developer can specify how to deploy it together with the security and monitoring mechanisms that should apply to its SMOOL client. GeneSIS will then inject within the SMOOL client the necessary code to perform the security checks before actually deploying it. To do so, we created a generic security component that represents a SMOOL client as a deployable artefact. This client can follow any of the security check options discussed above and is implemented with ThingML code, which integrates (i) the necessary SMOOL libraries, (ii) the SMOOL client business logic, (iii) and the security logic. The main rationale behind this choice is

12. https://casbin.org/

the following. ThingML offers an extra abstraction layer that provides the ability to wrap the code and dependencies that compose a SMOOL client and to inject into it the necessary security code. In addition, it provides GeneSIS with a standard and platform-independent procedure to generate, compile, configure, and deploy the implementation of the security mechanisms. A similar approach could be applied to other IoT platforms. In this way, GeneSIS allows DevOps teams to reconfigure and update security mechanisms by design, in line with the evolution of IoT applications and the development of security and privacy risks. In the next section, we present more details on the DevSecOps support.

4.4.2.2 The DevSecOps support for the continuous enhancement of security mechanisms

SIS typically expose a broad attack surface and their security must not be an afterthought [22]. The ability to continuously evolve and adapt these systems to their dynamic environment is decisive to ensure and increase their trustworthiness, quality, and user experience. This includes security mechanisms, which must evolve along with the SIS, continuously fixing security defects and dealing with new security threats [32, 35]. Following the DevSecOps principles [30], there is an urgent need for supporting the continuous deployment of SIS, including security mechanisms, over IoT, Edge, and Cloud infrastructures [31]. The DevOps movement promotes an iterative and incremental approach enabling the continuous evolution of software systems. As an evolution of the DevOps movement, DevSecOps promotes security as an aspect that must be carefully considered in all the development and operation phases for the continuous evolution of systems to be secure.

In this section, we present how GeneSIS can enable the continuous enhancement of security controls in a DevSecOps cycle: from development to operation. GeneSIS also supports the adaptation of the system having enhanced security mechanisms or updated security policies with minimal impact on the already delivered and under operation. Our approach [18, 20, 21] for the continuous deployment of SIS with enhanced security mechanisms can serve the DevOps team in both adaptation and evolution of the SIS. First, GeneSIS supports for evolving SIS with updated security mechanisms according to a new development cycle. Second, GeneSIS supports for adapting security enforcement to improve how the IoT system operates securely. This DevSecOps support leverages the GeneSIS ' necessary mechanisms, interfaces, and abstractions to dynamically adapt the deployment and configuration of a SIS as presented earlier. We elaborate more on the two kinds of DevSecOps support in the following paragraphs.

First, GeneSIS supports for evolving SIS with updated security mechanisms according to a new development cycle. In this line of adaptation, the SIS in operation is evolving with new business logic components or even new physical devices

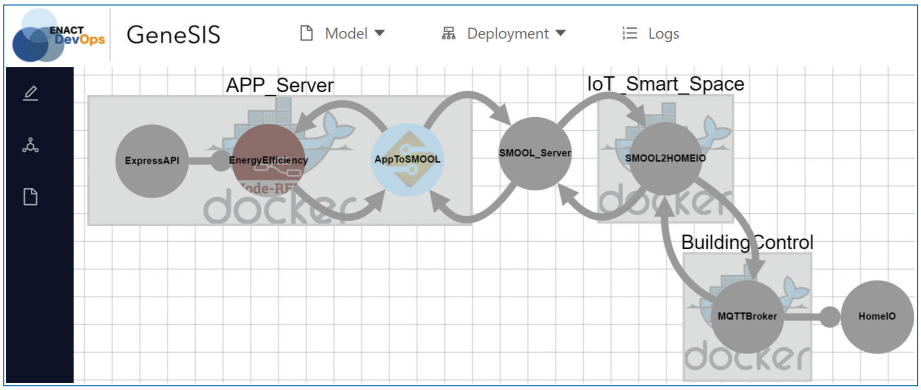

Figure 4.7. An initial version of a smart home system (deployment view).

being added resulting in the need for enhancing security mechanisms accordingly. We demonstrate this support using a smart home simulation called HomeIO.[13] More details on the deployment demo using the HomeIO simulation can be found in this video.[14] In the smart home system, there are IoT applications (e.g., *UserComfortApp*) that get access to sensors' data (e.g., temperature) from the smart home to make decisions and send commands to control the actuators, e.g., window blinds. The applications interact with the smart home devices and services via the SMOOL platform (in the middle of Fig. 4.7). GeneSIS can easily support for the deployment of components that are either built on top of the existing IoT platforms like SMOOL or are independent of any IoT platform because of its generic approach for specifying deployment components. However, to make GeneSIS even more useful in practice, we have developed GeneSIS to ease the integration of IoT platform-specific components (e.g., SMOOL clients) and IoT platform-independent components (e.g., third-party security mechanisms like Casbin presented below) from development to operation.

In the initial version of the smart home system, there is the *EnergyEfficiency* application, which gets access to sensors' data to make decisions for energy efficiency and send commands to control the actuators, e.g., window blinds. In particular, it maximizes the exploitation of daylights and regulates the in-door temperature whilst minimizing the energy consumption. If the room is bright because of daylight, it will switch off the LED-lights, and vice versa. On the other hand, if the room temperature is high, the application may need to close the window blinds to

13. https://realgames.co/home-io/

14. https://youtu.be/yQ9XYWu-EZM

prevent sunlight heating the room. The *EnergyEfficiency* application interacts with the smart home devices via the SMOOL platform. There are two notable security mechanisms associated in this first version of the smart home. The first one is a secure API gateway (Express Gateway[15]) that allows secure remote API access to the *EnergyEfficiency* application. The second one is a *SecurityEnforcer* by default of the SMOOL middleware that enforcing the security check for the data passing through, e.g., only allowing genuine actuation commands to be sent to the actuators of the smart home. The latest version of GeneSIS has provided a built-in support to ease the specification of the Express API Gateway in the deployment model. Adding a new instance of Express API Gateway is easy. The remaining work for the DevOps team is to specify the configuration files of the API gateway, which define how the API of the *EnergyEfficiency* application can be securely accessed.

In IoT platforms like SMOOL, there are often default security enforcements. For example, the actuation orders must be checked before they are actually sent to the actuators. This check (embedded in the *SMOOL2HOMEIO* component, Fig. 4.7) makes sure only genuine actuation commands can be sent to the actuators. In other words, the SMOOL platform allows to check for actuation commands with valid security tokens. All the IoT apps must send actuation commands with valid security tokens.

However, during the evolution of the smart building system, new applications can be added, and new physical devices can also be added. In the subsequent development cycle, another application called *UserComfortApp* has been added to the smart home system. Moreover, the smart home system can also have new IoT devices such as *AirQualitySensor* or *SmartDisplay* as shown in Fig. 4.8.

New security requirements come up because the smart building system must control which apps can access which actuators. This means that more fine-grained security control must be introduced, which may not be available in the IoT platform. GeneSIS should support for seamlessly integrating new (third-party) security mechanisms into the IoT platforms. In this new development cycle, not only that the secure API gateway must be updated with a new configuration file, but also the DevOps team needs to introduce a new security mechanism that can enhance the fine-grained control of how different applications can access to the sensors and actuators of the smart home system. GeneSIS has a generic support for seamlessly integrating and deploying any advanced security mechanism together with the IoT platform in use, e.g., the SMOOL platform. More specifically, in this example, the DevOps team develop an access control mechanism based on an open source

15. https://www.express-gateway.io/

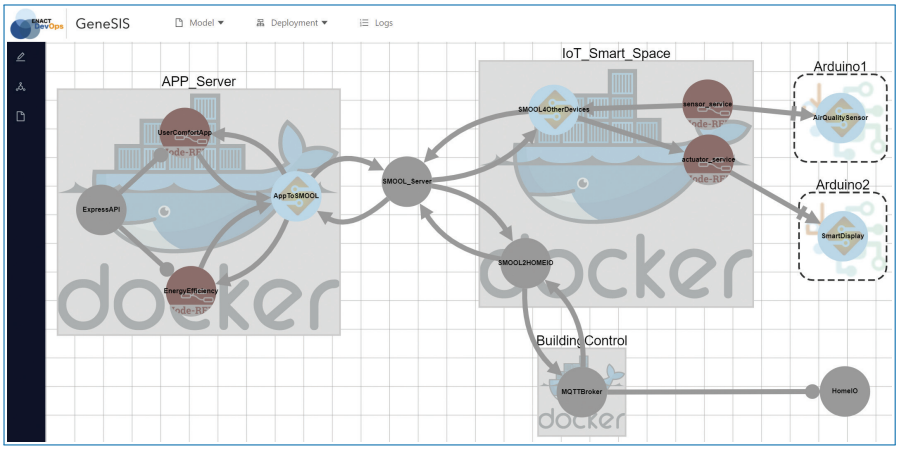

Figure 4.8. New applications and new IoT devices can be added in a development cycle.

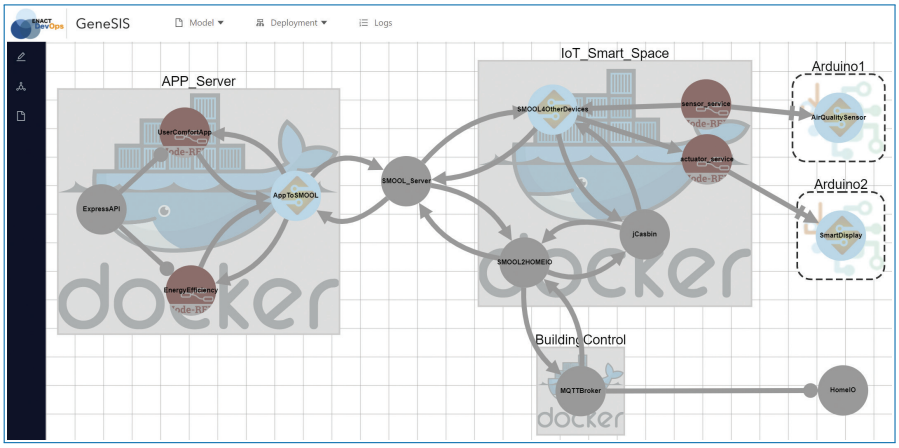

Figure 4.9. An enhanced security control has been added.

framework called jCasbin,[16] and then specify the integration point with the IoT platform in use (with GeneSIS support, see Fig. 4.9). During the deployment process, GeneSIS compiles the integration code before orchestrating the deployment of the integrated components.

To enable such DevSecOps adaptation support, GeneSIS not only provides the modelling language embedded in a web UI for specifying the components of such IoT platforms, but also the reconfiguration and rebuild of these components (for integrating new security mechanisms with the IoT platform) before deployment (for adaptation or for a new development cycle). For example, in the SMOOL

16. https://casbin.org/

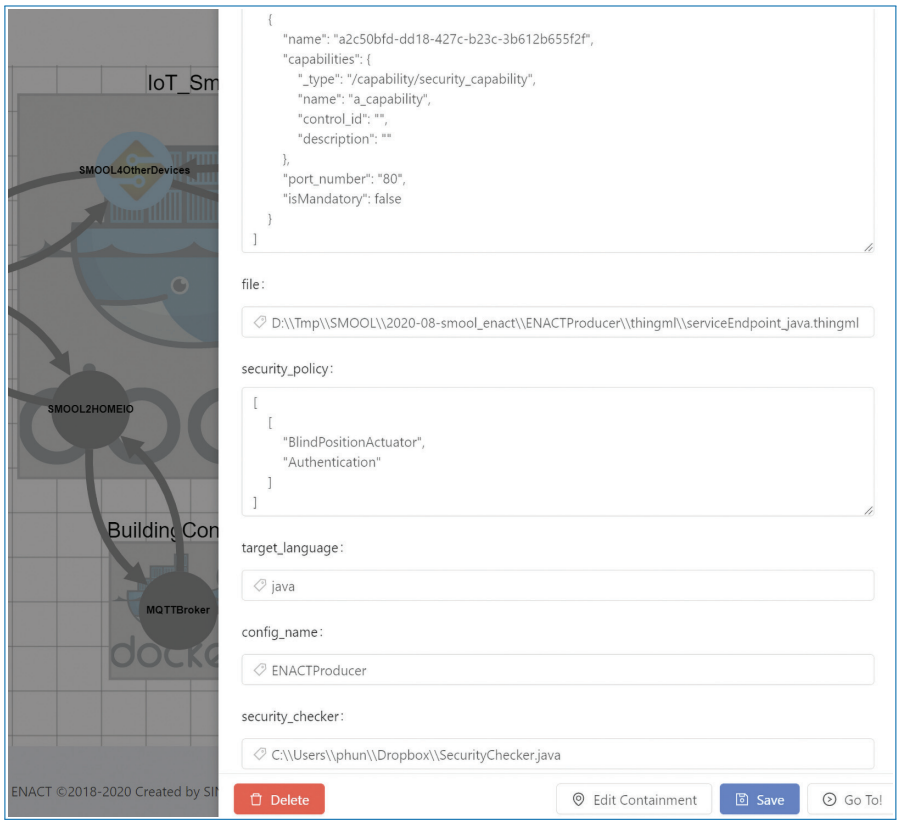

Figure 4.10. An enhanced security control has been added.

platform, each SMOOL producer or consumer is associated with a security checker for checking the security key of sensor data or actuation commands. GeneSIS allows updating the configuration of the security checker (e.g., by injecting new configuration to overwrite the default one), and automatically rebuilding the SMOOL producer or consumer including the reconfigured security checker. By doing so, GeneSIS enables the DevOps team to make reconfiguration or redevelopment and redeployment easily for the evolution of SMOOL producers or consumers including security checkers. Figure 4.10 shows an example of the GeneSIS's UI for extending the *SecurityChecker* (in *SMOOL2HOMEIO*) to become a security enforcement point of the external access control service. Thanks to ThingML support within GeneSIS, the extended *SecurityChecker* is compiled in the *SMOOL2HOMEIO* component for a new version of *SMOOL2HOMEIO* to be deployed that works as a security enforcement point of the external access control service.

This approach is what we call the DevSecOps adaptation support for the co-evolution of business logic components and the security mechanisms. This means

that when new business logic components require security mechanisms to evolve, GeneSIS can support for the adaptation, even including the integration of the IoT platform with other (third-party) security mechanisms.

After the successful deployment, GeneSIS allows dynamic adaptations that can be triggered at any point, manually or automatically for adapting security enforcement to improve how the IoT system operates securely. In this line of adaptation, new security policies or configurations can be updated dynamically for the security mechanisms that are in operation. For example, the role-based access control policy can be easily updated according to new requirements. The trigger of such adaptation can be manually, but also can be automatically from a risk assessment process or after a reasoning process of actuation conflict management.

In summary, with the support from GeneSIS, the DevOps team can develop a new version of the smart home system together with enhanced security mechanisms according to its evolution. The deployment of this new development cycle can be triggered manually from GeneSIS's GUI. After the successful deployment, GeneSIS also allows dynamic adaptations that can be triggered at any point, manually or automatically for adapting security enforcement to improve how the IoT system operates securely. In both ways presented so far, GeneSIS allows DevSecOps teams to reconfigure and update security mechanisms by design, in line with the evolution of IoT applications and the development of security and privacy risks.

It is important to note that in this chapter we have not addressed the security of the build and deployment pipeline itself. The security of this pipeline is critical to protect the integrity of the code and the systems being deployed. For the production environment, GeneSIS must adhere to the secure deployment practice.[17] One of the main principles in secure deployment is to support automatic testing as part of the deployments to gain confidence in the security of the code (see Section 8.2 for Test and Simulation).

4.4.3 Software Diversity Within IoT Fleet

Software diversity in an IoT fleet, i.e., deploying variants of software on different devices, creates a moving target for malicious attacks, and therefore improves the overall security of the system. The DivEnact tool assigns the available variants to the fleet of devices and maintains the balance between the variants. The remaining questions is how to obtain functionally-equivalent variants.

The ENACT IoT diversity-by-design tool takes as input a single deployment or behaviour specification and generates multiple diverse specifications. Within a DevOps context, it is important and necessary to keep the diversity generation fully

17. https://owaspsamm.org/model/implementation/secure-deployment/

automatic, instead of relying on developer's manual effort to diversify systems (such as the traditional N-Version Programming approach). Developers can focus on a single line of code to achieve frequent iteration, and the diversification tool, as part of automatic building step, will generate diversified versions automatically [14].

Automated diversity is a promising means of mitigating the consequences of a security breach. However, current automated diversity techniques operate on individual processes, leveraging mechanisms available at the lower levels of the software stack (in operating systems and compilers), yielding a limited amount of diversity. In this section, we present a novel approach for the automated synthesis of diversified protocols between processes. This approach builds on (i) abstraction, where the original protocol is modelled by a set of communicating state machines, (ii) automated synthesis, applying mutation operators onto those protocols, which produces semantically-equivalent, yet phenotypically-different protocols, and (iii) automated implementation of these protocols through code generation.

The tool is currently in an experimental stage. Automatic diversity of communication protocols is a novel technology, yet without convincing implementation and applications, to the best of our knowledge. Therefore, our focus is currently on the theoretical feasibility of the idea and the experimental evaluation of its effects. In the next step, we will improve the user experience of the tool and its applicability to practical scenarios.

Mass-produced software applications denote clonal applications, with thousands or millions of identical siblings. Think of, for example, a popular mobile application installed on millions of mobile phones, or software embedded into a widely-used connected device. To mitigate the risks of such large mono-cultures, diversity is typically automatically introduced either in a generic way, typically at the OS level, oblivious of the actual logic and semantics of the software, or in some very specific places, typically low-level libraries reused across applications, in order to improve security. This leaves most of the actual business logic unchanged, unaffected by the diversity. In addition, diversity often affects individual processes, but leaves the communication between processes intact.

A more holistic approach to diversity is challenging. Consider a typical client-server application, where multiple clients interact with a server, and where each client has a different implementation, and a different way of communicating with the server. This would significantly hinder a hacker, be it a human being or a machine, when attempting to generalize an attack through all possible protocols. This would make large-scale exploits a time-consuming and costly endeavour for hackers. Yet, the engineering, e.g., the production, maintenance and integration, of such levels of diversity raises several challenges. How to ensure that each implementation still behaves as specified? How to ensure that each client is still able to communicate with the server, without information loss or distortion? How to

ensure that different clients are fundamentally (i.e., sufficiently) different, and not merely cosmetically different? How to keep the development and operation costs of a diversified system significantly lower than the cost of mitigating large scale attacks?

We have seen that abstraction, synthesis and automated implementation can yield a convincing solution to introduce a wide diversity into protocols, for example between a device and a gateway, or a web/mobile app and a server. This approach:

- abstracts protocols into (i) a structural view describing the messages to be exchanged, and (ii) a behavioral view based on state machines describing how those messages are exchanged between the participants, including sequencing and timing.
- combines and applies a number of atomic mutations to this protocol model, yielding a large number of diversified protocols, which operate differently, still with the same semantics.
- automatically implements protocols, diversified or not, by generating fully operational code targeting C, Go, Java and JavaScript, able to run on a wide range of platforms.

Our empirical assessment indicated that this approach implies a reasonable overhead in terms of execution time, memory consumption and bandwidth, fully compatible with the requirements of mass-produced software. We also showed that this approach could generate a significant amount of diversity. Our assumption was that this diversity would contribute to the diversity-stability hypothesis, i.e., this would make the whole ecosystem more robust by making it less likely for an exploit to propagate to the whole population. In other words, if the protocol between a specific client and the server could be observed, analysed and eventually understood, this would not systematically imply that all other diversified protocols could be understood following the very same procedure. In this section, we briefly describe the mechanisms and the corresponding tools we developed to automatically generate the diverse protocols. Technical details and the experiment results can be found in our conference paper [29].

Our approach relies on ThingML [26] for the specification of protocols. ThingML provides a way to formalize the messages involved in protocols, in a comparable way to what Protocol Buffer proposes. In addition, ThingML provides a mean to formalize the behavior of protocols through state machines. ThingML specifications are both human-readable and machine-readable, which makes it possible to analyse protocols at a high-level of abstraction and to fully automate the implementation of those protocols through code generation. In the next-subsection, we present relevant aspects of ThingML on our motivating example.

We model communication protocols as a set of communicating state-machines, encapsulated into components. A protocol typically involves two roles: (i) a client, i.e., a device, a web-browser or a mobile app, and (ii) a server, i.e., a gateway or a Cloud back-end. The clients and the server need to agree on a common API. Since communication is typically asynchronous in a distributed system, the common API is specified as a set of messages. Next, this API is imported by the client component and the server component, and the messages are organized into ports.

The ultimate goal of our approach is to diversify the wire image of protocols. Diversifying the wire image of protocols basically means shuffling the sequence of bytes exchanged over the network e.g., turning the payloads while ensuring the interoperability between the client and the server.

4.5 Conclusions

This chapter summarizes our effort in the ENACT project towards automatic software deployment for Smart IoT Systems. Automatic deployment is a cornerstone of DevOps, as it connects development with operation, and ensures that changes on the software will be placed into the production in a correct and prompt way.

Although there are already mature deployment solutions for Cloud computing in the market, automatic deployment for smart IoT systems is still an open problem. The main challenges are from two fundamental characters of smart IoT systems: First, an IoT application involves software running at all types of resources along the Cloud-Edge-IoT continuum, and it is difficult to provide a consistent way to support the deployment on all those different types of resources. Second, an IoT application in the production stage usually contains many subsystems of Edge and IoT devices, each of which serves a particular user or manages a particular part of the physical world. It is difficult to deploy a new change on the software to all those subsystems regardless of the different contexts and status among them.

During the ENACT project, we conducted research aiming at these two challenges, resulting in two ENACT enablers, namely GeneSIS and DivEnact. We briefly introduced how these enablers work, both as individual tools and as an integrated deployment bundle for the automatic deployment of SIS. More details about the theories, implementations and use cases can be founded in our recent publications [15, 44]. In this chapter, we focused on the mechanisms and practices of using these tools to ensure the trustworthiness of the deployment software, including the availability of software components on unstable resources, the deployment support of security and privacy mechanisms, and the automatic generation and maintenance of software diversity towards a more secure systems.

In the next step, we will extend the concepts and implementation of automatic deployment into the more general edge computing domain, providing an engineering solution for the core problem of edge computing, i.e., the distribution and offloading of computation among the complex and dynamic resources. Currently, the deployment is driven by manually define deployment models which embeds the resource allocation and the constraints about software-device mapping. An important future plan is to introduce intelligence into automatic deployment, which learns from historical deployments and their effects to automatically assign software parts to the proper resources.

References

[1] C. R. Aguayo Gonzalez, C. B. Dietrich, and J. H. Reed. "Understanding the software communications architecture". In: *IEEE Communications Magazine* 47.9 (2009), pp. 50–57.

[2] Carlos Ansótegui *et al.* "Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem". In: *Ninth Symposium of Abstraction, Reformulation, and Approximation*, 2011.

[3] Matej Arta *et al.* "Model-driven continuous deployment for quality devops". In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. 2016, pp. 40–41.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. Addison-Wesley Professional, 2012. ISBN: 0321815734.

[5] Benoit Baudry and Martin Monperrus. "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond". In: *ACM Comput. Surv.* 48.1 (Sept. 2015). ISSN: 0360-0300. DOI: 10.1145/2807593. URL: https://doi.org/10.1145/2807593.

[6] Gordon S. Blair, Nelly Bencomo, and Robert B. France. "Models@run.time". In: *IEEE Computer* 42.10 (2009), pp. 22–27.

[7] Miquel Bofill *et al.* "Solving constraint satisfaction problems with SAT modulo theories". In: *Constraints* 17.3 (2012), pp. 273–303.

[8] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. "Satisfiability modulo theories and assignments". In: *International Conference on Automated Deduction*. Springer. 2017, pp. 42–59.

[9] Antonio Bucchiarone, Antonio Cicchetti, and Annapaola Marconi. "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. 2019, pp. 34–44.

[10] Sylvain Cherrier *et al.* "D-lite: Distributed logic for internet of things services". In: *2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*. IEEE. 2011, pp. 16–24.

[11] F. Cirillo *et al.* "A Standard-Based Open Source IoT Platform: FIWARE. In: *IEEE Internet of Things Magazine* 2.3 (2019), pp. 12–18. DOI: 10.1109/IOTM.0001.1800022.

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

[13] Rustem Dautov and Hui Song. "Towards Agile Management of Containerised Software at the Edge". In: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1. IEEE. 2020, pp. 263–268.

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

[15] Rustem Dautov, Hui Song, and Nicolas Ferry. "A Light-Weight Approach to Software Assignment at the Edge". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2020, pp. 380–385.

[16] Johannes Eder *et al.* "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. 2017, pp. 270–280.

[17] Johannes Eder *et al.* "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*. 2018, pp. 438–446.

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

[19] Nicolas Ferry *et al.* "CloudMF: Model-Driven Management of Multi-Cloud Applications". In: *ACM Transactions on Internet Technology (TOIT)* 18.2 (2018), p. 16.

[20] Nicolas Ferry *et al.* "Continuous Deployment of Trustworthy Smart IoT Systems". In: *The Journal of Object Technology* (2020).

[21] Nicolas Ferry *et al.* "Genesis: Continuous orchestration and deployment of smart IoT systems". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE. 2019, pp. 870–875.

[22] M. Frustaci *et al.* "Evaluating Critical Security Issues of the IoT World: Present and Future Challenges". In: *IEEE Internet of Things Journal* 5.4 (2018), pp. 2483–2495. DOI: 10.1109/JIOT.2017.2767291.

[23] Anne Gallon *et al.* "Making the Internet of Things More Reliable Thanks to Dynamic Access Control". In: *Security and Privacy in the Internet of Things: Challenges and Solutions* 27 (2020), p. 61.

[24] Nam Ky Giang *et al.* "Developing IoT applications in the fog: a distributed dataflow approach". In: *Internet of Things (IoT), 2015 5th International Conference on the*. IEEE. 2015, pp. 155–162.

[25] Object Management Group. "Deployment and Configuration of Component-based Distributed Applications Specification". In: *OMG Available Specification Version 4.0 formal/06-04-02* (2006).

[26] Nicolas Harrand *et al.* "ThingML: A Language and Code Generation Framework for Heterogeneous Targets". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. MODELS 16. Saint-malo, France: Association for Computing Machinery, 2016, pp. 125–135. ISBN: 9781450343213.

[27] Stéphane Lavirotte *et al.* "A generic service oriented software platform to design ambient intelligent systems". In: *Proceedings of the 2015 ACM International Conference on Pervasive and Ubiquitous Computing*. ACM. 2015, pp. 281–284.

[28] Amardeep Mehta *et al.* "Calvin Constrained-A Framework for IoT Applications in Heterogeneous Environments". In: *37th International Conference on Distributed Computing Systems*. IEEE. 2017, pp. 1063–1073.

[29] Brice Morin *et al.* "Engineering software diversity: A model-based approach to systematically diversify communications". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2018, pp. 155–165.

[30] Håvard Myrbakken and Ricardo Colomo-Palacios. "DevSecOps: A Multivocal Literature Review". In: *Software Process Improvement and Capability Determination*. Ed. by Antonia Mas *et al.* Cham: Springer International Publishing, 2017, pp. 17–29. ISBN: 978-3-319-67383-7.

[31] NESSI. *Cyber physical systems: Opportunities and challenges for soft- ware, services, cloud and data*. NESSI White paper. 2015.

[32] P. H. Nguyen *et al.* "SoSPa: A system of Security design Patterns for Systematically engineering secure systems". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015, pp. 246–255. DOI: 10.1109/MODELS.2015.7338255.

[33] Phu H. Nguyen, Phu H. Phung, and Hong-Linh Truong. "A Security Policy Enforcement Framework for Controlling IoT Tenant Applications in the Edge". In: *Proceedings of the 8th International Conference on the Internet of Things*, IOT 18. Santa Barbara, California, USA: ACM, 2018. ISBN: 9781450365642.

[34] Phu H. Nguyen *et al.* "Advances in deployment and orchestration approaches for IoT – A systematic review". In: *2019 IEEE International Congress On Internet of Things (ICIOT)*. Milan, Italy: IEEE, 2019, pp. 53–60.

[35] Phu H. Nguyen *et al.* "An extensive systematic review on the Model-Driven Development of Secure Systems". In: *Information and Software Technology* 68 (2015), pp. 62–81. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.08.006. URL: http://www.sciencedirect.com/science/article/pii/S0950584915001482.

[36] Adrian Noguero, Angel Rego, and Stefan Schuster. "Towards a Smart Applications Development Framework". In: *Social Media and Publicity* 27 (2014). URL: https://bitbucket.org/jasonjxm/smool,%202011-2020.

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

[38] Temel Öncan. "A survey of the generalized assignment problem and its applications". In: *INFOR: Information Systems and Operational Research* 45.3 (2007), pp. 123–141.

[39] Allen Parrish, Brandon Dixon, and David Cordes. "A conceptual foundation for component-based software deployment". In: *Journal of Systems and Software* 57.3 (2001), pp. 193–200. ISSN: 0164-1212.

[40] David W Pentico. "Assignment problems: A golden anniversary survey". In: *European Journal of Operational Research* 176.2 (2007), pp. 774–793.

[41] Subhav Pradhan *et al.* "Chariot: Goal-driven orchestration middleware for resilient IoT systems". In: *ACM Transactions on Cyber-Physical Systems* 2.3 (2018), pp. 1–37.

[42] Davide Di Ruscio, Richard F. Paige, and Alfonso Pierantonio, eds. *Special issue on Success Stories in Model Driven Engineering*. Vol. 89, Part B. Elsevier, 2014.

[43] Ana C Franco da Silva *et al.* "OpenTOSCA for IoT: automating the deployment of IoT applications based on the mosquitto message broker". In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM. 2016, pp. 181–182.

[44] Hui Song *et al.* "Model-based fleet deployment of edge computing applications". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2020, pp. 132–142.

[45] Hui Song *et al.* "On architectural diversity of dynamic adaptive systems". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 595–598.

[46] Michael Vögler *et al.* "A scalable framework for provisioning large-scale IoT deployments". In: *ACM Transactions on Internet Technology (TOIT)* 16.2 (2016) pp. 1–20.

[47] Changsheng You *et al.* "Energy-efficient resource allocation for mobile-edge computation offloading". In: *IEEE Transactions on Wireless Communications* 16.3 (2016) 1397–1411.