Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Continuous Deployment of Trustworthy Smart IoT Systems

Nicolas Ferry[a] Phu H. Nguyen[a] Hui Song[a] Erkuden Rios[b]
Eider Iturbe[b] Satur Martinez[b] Angel Rego[b]

a. SINTEF, Oslo, Norway

b. TECNALIA, Basque Research and Technology Alliance, Spain

Abstract While the next generation of IoT systems need to perform distributed processing and coordinated behaviour across IoT, Edge and Cloud infrastructures, their development and operation are still challenging. A major challenge is the high heterogeneity of their infrastructure, which broadens the surface for security attacks and increases the complexity of maintaining and evolving such complex systems. In this paper, we present our approach for Generation and Deployment of Smart IoT Systems (GeneSIS) to tame this complexity. GeneSIS leverages model-driven engineering to support the DevSecOps of Smart IoT Systems (SIS). More precisely, GeneSIS includes: *(i)* a domain specific modelling language to specify the deployment of SIS over IoT, Edge and Cloud infrastructure with the necessary concepts for security and privacy; and *(ii)* a models@run.time engine to enact the orchestration, deployment, and adaptation of these SIS. The results from our smart building case study have shown that GeneSIS can support security by design from the development (via deployment) to the operation of IoT systems and back again in a DevSecOps loop. In other words, GeneSIS enables IoT systems to keep up security and adapt to evolving conditions and threats while maintaining their trustworthiness.

Keywords Deployment; MDE; DSL; models@run.time; IoT; DecSecOps.

1 Introduction

The next generation of IoT systems often need to perform distributed processing and coordinated behaviour across IoT, Edge and Cloud infrastructures. Such systems, denoted in this paper as smart IoT systems (SIS), typically rely on heterogeneous infrastructures ranging from virtual machines running in the Cloud to gateways and micro-controllers. They typically expose a broad attack surface and their security must not be an afterthought. The ability to continuously evolve and adapt these systems to their new environment is decisive to ensure and increase their trustworthiness, quality and user experience. This includes security mechanisms, which must evolve

Nicolas Ferry, Phu H. Nguyen, Hui Song, Erkuden Rios, Eider Iturbe, Satur Martinez, Angel Rego. *Continuous Deployment of Trustworthy Smart IoT Systems*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 16:1–23.
doi:10.5381/jot.2020.19.2.a16

along with the smart IoT systems, continuously fixing security defects and dealing with new security threats. In particular, following the DevSecOps principles [MCP17], there is an urgent need for supporting the continuous deployment of IoT systems, including security mechanisms, over IoT, Edge, and Cloud infrastructures [ea15]. As an evolution of the DevOps movement, which calls for better collaborations between developers and operators and advocates to further automate deployment to improve the flexibility and efficiency of the delivery process. DevSecOps promotes security as an aspect that must be carefully considered in all the development and operation phases. However, in practice the DevSecOps of a SIS typically face two key obstacles: *(i)* the high heterogeneity across Cloud, Edge, and IoT resources and *(ii)* complex maintenance and evolution management of the system and its security mechanisms.

In the past years, multiple tools have emerged to support the building as well as the automated and continuous deployment of software systems with a specific focus on cloud infrastructures (*e.g.*, Puppet, Chef, Ansible, Vagrant, Brooklyn, OpenTOSCA, CloudML, etc.). However, very little effort has been spent on providing solutions tailored to the delivery and deployment of applications across the whole IoT, Edge, and Cloud space [NFE+19, ea15]. In particular, Cloud and Edge solutions typically lack languages and abstractions that can be used to support the orchestration of software services and their deployment on heterogeneous IoT devices possibly with limited or no direct access to the Internet [NFE+19]. Moreover, there has not been any IoT-specific deployment method dedicated to support continuous deployment of security mechanisms together with IoT applications [NFE+19].

In this paper, we present our approach for Generation and Deployment of Smart IoT Systems (GeneSIS)[1]. GeneSIS aims to support the continuous deployment of smart IoT systems over IoT, Edge and Cloud infrastructures. Moreover, our approach aims to support DevSecOps [MCP17] to promote security-by-design along with agility, enabling dynamism in security protections. Thus, GeneSIS includes: *(i)* a domain specific modelling language to specify the deployment of SIS over IoT, Edge and Cloud infrastructure with the necessary concepts for security and privacy; and *(ii)* a models@run.time deployment engine to enact the orchestration, deployment, and adaptation of these SIS. More precisely, the main contributions of GeneSIS are:

1. By relying on a model-driven approach and the principle of "model once, generate anywhere", it enables to cope with the vast heterogeneity of IoT, Edge and Cloud infrastructures and control the orchestration and continuous deployment of SIS that span across this space. Particular focus has been to tackle challenges imposed by IoT infrastructures that typically include devices with no or limited access to the Internet.

2. It enables to cope with security and privacy concerns of SIS as it offers necessary concepts to specify security and privacy requirements and to support the automatic deployment of the associated security and privacy mechanisms.

3. By leveraging the Models@run.time approach, the same language and tool are used for the continuous deployment of SIS (including the monitoring of the deployment progress - *i.e.*, monitoring if hosts are still reachable and if software component are still running, and the dynamic adaptation of a deployment - *i.e.*,

[1] A short position paper about the GeneSIS concept was published at COMPSAC'19 [FNS+19]. This paper elaborate the language and models@run.time engine, extending it in particular with support for security concerns.

modifying how a SIS is deployed), providing a unique model-based representation of the SIS for both design- and run-time activities.

The remainder of the paper is organized as follows. Section 2 introduces a motivating example used throughout the paper. Section 3 presents the GeneSIS Modeling language while Section 4 details the supporting execution engine. In Section 5, we summarize how GeneSIS fulfills the identified requirements and report on the usage of GeneSIS in the context of an industrial use case. Section 6 presents our analyses on the state of the art of deployment and orchestration approaches for IoT Systems. Finally, Section 7 concludes the paper.

2 Motivating Example

In this section, we describe a motivating example inspired by the smart building case study from the ENACT H2020 project [FSS+18, FDG+20], which needs support for the continuous deployment of its SIS together with security mechanisms. The SIS aims at improving user comfort and energy efficiency in the building and is formed by three different systems as illustrated by the informal diagram shown in Figure 1.

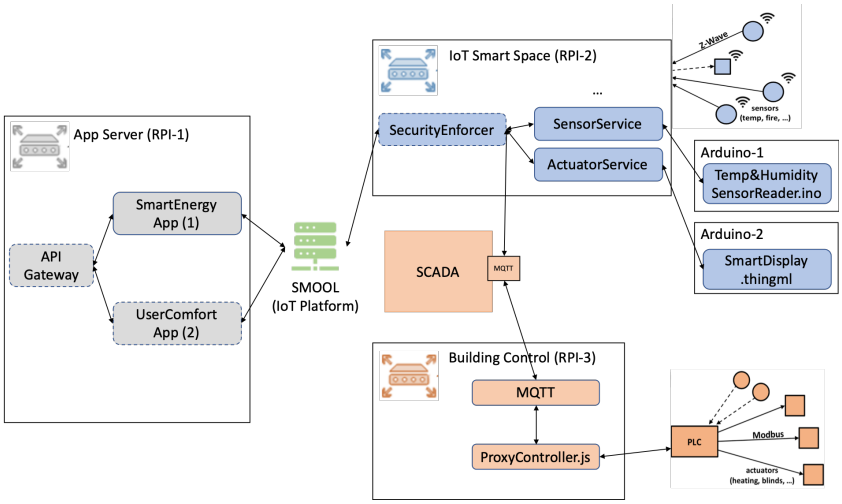


Figure 1 – Architecture of the SIS in the Smart Building example

The first system (namely IoT Smart Space in Figure 1) includes a gateway (RPI-2) that connects to *(i)* a wireless sensor/actuator network using the Z-Wave protocol and *(ii)* smart devices (Arduinos) physically connected to the gateway via serial port. The second system (namely Building Control in Figure 1) is a proprietary building control system that consists of a gateway (RPI-3) and a PLC (Programmable Logic Controller) that uses building automation protocols (KNX, DALI, PROFIBUS, etc.) and direct control over the devices through relays or analog/digital outputs. These two systems communicate using a MQTT broker of the SCADA. Finally, the third system (namely App-Server) hosts applications that commands the sensors and actuators available in the other two systems. The App-Server interoperates with the IoT Smart Space and the building control using a cloud-based instance of the Interoperability SMOOL IoT middleware [NRS14] that has a semantic broker for connecting heterogeneous devices or sources of information. More precisely, the App-Server only communicates with the IoT Smart Space, which in turn forwards authorized and secured messages to

the Building Control system. For security reason, all the messages between the App-Server and the IoT Smart Space are checked and controlled by the SecurityEnforcer component deployed on RPI-2. In other words, the local services on RPI-2 are deployed together with security mechanisms ensuring that applications are allowed to access and send authorized actuation commands to the actuators.

For the deployment of the SIS, we consider the following two-stage scenario. In the first stage, all the software components depicted in Figure 1 with plain border are deployed. This includes: *(i)* the security and privacy mechanisms, *(ii)* the software components on the Arduino boards, and *(iii)* the SmartEnergy application. The SmartEnergy application 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 IoT Energy Efficiency application may need to close the window blinds to prevent sunlight heating the room.

In the second stage, a second application (*i.e.*, user comfort) is deployed on the App-Server. The SecurityEnforcer component needs to be updated with a new security policy to provide this application with access to sensors and actuators. In addition, because both applications can be accessed by external services, a generic and secured API gateway is placed in front of them, making them accessible from external services via authorised API calls.

This example motivates for the following requirements that are addressed by GeneSIS:

- **Separation of concerns and reusability (R_1)**: A modular, loosely-coupled specification of the data flow and its deployment is required so that the modules can be seamlessly substituted and reused. Elements or tasks should be reusable across scenarios.

- **Abstraction and infrastructure independence (R_2)**: It is a need to be able to specify the orchestration and deployment of SIS over IoT, edge, and cloud infrastructures in both a device- and platform-independent and -specific way. In addition, a continuously up-to-date, abstract representation of the running system is required to facilitate the reasoning, simulation, and validation of operation activities.

- **White- and black-box infrastructure (R_3)**: Support for white- and black-box devices is required to cope with various degrees of delegation of control over underlying infrastructures and platforms.

- **Automation and adaptation (R_4)**: A fully automated deployment of SIS over IoT, edge, and cloud resources is required. In addition, the deployment of a system should be dynamically adaptable with minimal impact over the running system (*i.e.*, only the necessary part of the system should be adapted).

- **Specify security and privacy requirements (R_5)**: GeneSIS should support the specification of the security mechanisms required and offered by the different components that form the SIS.

- **Automatic deployment and enforcement of generic security mechanisms (R_6)**: Support for continuously deploying security mechanisms should

be offered. In addition to deploying security mechanisms as any software components, generic off-the-shelf security components that can be deployed in different scenarios and context are required.

- **Deployment on devices with no Internet connection (R_7)**: Deployment of software on tiny devices that do not always have direct access to the Internet or even the necessary facilities for remote access is required.

Abstraction and infrastructure independence (R_2) and automation (R_4) are justified by the need for deploying the system on an infrastructure leveraging the IoT (*i.e.*, Arduino board), Edge (*i.e.*, Raspberry PI), and Cloud (*i.e.*, Amazon EC2) spaces. Separation of concerns (R_1), automation and adaptation (R_4), and automatic deployment of generic security mechanisms (R_5) are justified by the need for dynamically *(i)* adding a new software component to manage the access to the applications and *(ii)* updating security policies, with minimal impact on the already running system. The support for white- and black-box infrastructure (R_3) is justified by the need to use, in the same system, a black-box device (*i.e.*, the Z-Wave transceiver) and white-box devices (*e.g.*, Raspberry PI). The support for security and privacy requirements (R_5) and enforcement (R_6) is justified by the involvement in the system of actuators whose access should be controlled, and private data must be protected. Finally, support for deploying software on devices with no Internet connection is justified by the need to deploy software on the two Arduinos (R_7).

In the next sections, we present GeneSIS and how it addresses these requirements.

3 The GeneSIS Modelling Language

The objective of GeneSIS is to support the orchestration and deployment of IoT systems whose software components can be deployed over IoT, Edge, and Cloud infrastructures. The target user groups for our framework are thus mainly DevOps engineers, software developers, and architects.

To deploy an application on the selected target environment, its application components need to be allocated on host services and infrastructure. More precisely, what needs to be allocated is the implementations of those components. This is often referred as deployable artefact [BBF+18]. Some examples of deployable artefacts are binaries, scripts, etc. A deployable artefact can be physically allocated independently to multiple hosts (*e.g.*, a Jar file can be uploaded and executed on different Java runtime environments).

Where and how these deployable artefacts are allocated is specified in a deployment model. Deployment approaches typically rely on the logical concept of software artefacts or components [Dea07]. A deployment model is thus a connected graph that describes software components along with targets and relationships between them from a structural perspective [BBF+18].

GeneSIS includes a domain-specific modelling language to specify deployment model – *i.e.*, the orchestration and deployment of Smart IoT Systems (SIS) across the IoT, Edge, and Cloud spaces. In the following, we provide a description of the most important classes and corresponding properties in the GeneSIS metamodel as well as examples in the associated textual syntax. The textual syntax better illustrates the various concepts and properties that can be involved in a deployment model, and that can be hidden in the graphical syntax.

3.1 Deployment Models for IoT, Edge and Cloud-based Systems

One of the objectives when we developed the GeneSIS Modelling language was to keep it with minimal set of concepts, but still easily extensible. Our language follows a component-based approach in order to facilitate separation of concerns and reusability (for R_1). Here, deployment models can be regarded as assemblies of components.

It is worth noting that we applied the type-instance pattern [AK02] to facilitate the definition and reuse of generic type of components (addressing R_1). As a result, components can remain device- and platform-independent or specialized into device- or platform-specific components (for R_2). The type part of the GeneSIS modelling language metamodel is depicted in Figure 2.

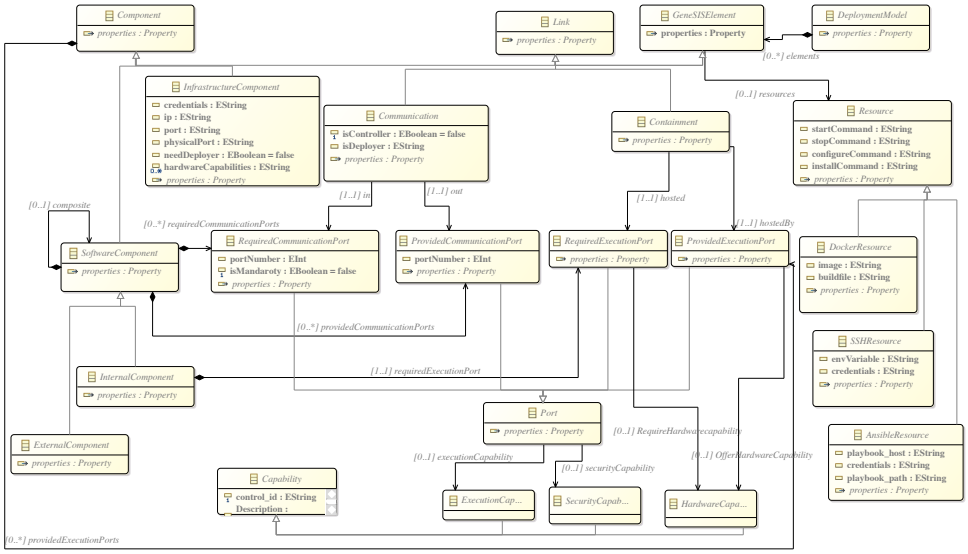


Figure 2 – Type part of the GeneSIS language metamodel

A Deployment Model consists of `GeneSISElements`. Each `GeneSISElement` inherits from `NamedElement`[2] and thus has a unique name. In addition, they can all be associated with a list of properties in the form of key-value pairs. The two main types of `GeneSISElements` are `Components` and `Links`.

A `Component` represents a reusable type of node that will compose a Deployment-Model. A `Component` can be a `SoftwareComponent` representing a piece of software to be deployed on a host (*e.g.*, the Temp&HumiditySensorReader Arduino sketch to be deployed on Arduino-1). A `SoftwareComponent` can be an `InternalComponent` meaning that it is managed by GeneSIS (*e.g.*, the instance of MQTT to be deployed on RPI-3), or an `ExternalComponent` meaning that it is either managed by an external provider (*e.g.*, the SMOOL IoT middleware offered as a service) or hosted on a blackbox device (*e.g.*, Z-Wave transceiver) (addressing R_3). A `SoftwareComponent` can be associated with `Resources` (*e.g.*, scripts, configuration files) adopted to manage its deployment life-cycle (*i.e.*, download, configure, install, start, and stop). In particular, there are three main predefined types of resources: `Docker-Resource` (see Listing 1), `SSH-Resources`, and `AnsibleResources`.

```
{
  "_type": "/internal/mqtt",
```

[2]Due to space limitation, the NamedElement class is not represented in Figure 2

```
  "name": "MQTTBroker",
  "properties": [],
  "version": "0.0.1",
  "provided_execution_port": [{
      "name": "6c4b4d0d-d9d7-4e70-98e8-0e9f3b302285"
   }],
   "docker_resource": {
      "name": "f3e3feba-056e26a7-9225-5b9edf5f1820",
      "image": "eclipse-mosquitto:1.6.8",
      "command": "",
      "port_bindings": {
          "1883": "1883",
          "9001": "9001"
   }},
   "required_execution_port": {
      "name": "80c0d0cb-421f-411d-8c75-bd01e56d29fd",
      "needDeployer": false
   },
   "provided_communication_port": [{
      "name": "bd3provf-f691-4a46-b9cd-14bab0f9a8e0",
      "port_number": "1883",
   }],
  "required_communication_port": [{
      "name": "bd3f34af-f691-4a46-b9cd-14bab0f9a8e0",
      "port_number": "1883",
      "isMandatory": false
   }]
}
```


Listing 1 – An example of Internal component

An `InfrastructureComponent` provides hosting facilities (*i.e.*, it provides an execution environment) to `SoftwareComponents`. The properties `IP` and `port` represent the IP address and port that can be used to reach the `InfrastructureComponent` (see Listing 2). The property `needDeployer` depicts that a local connection is required to deploy a `SoftwareComponent` on an `InfrastructureComponent` via a `Physical-Port` (*e.g.*, the Arduino board can only be accessed locally via serial port, see Listing 2). This property is typically used for devices with no direct access to Internet, which can only be reached indirectly via other devices (*e.g.*, a RaspberryPi gateway), configured by GeneSIS (addressing R_7).


```
{
     "_type": "/infra/device",
     "name": "arduino1",
     "properties": [],
     "version": "0.0.1",
     "provided_execution_port": [{
        "name": "a0bfe966-a952-47e4-8580-c36dc288e57d",
        "capabilities": [{
          "_type": "/capability/hardware_capability",
          "name": "bus_i2c",
          "control_id": "I2C",
          "description": "Temperature and Humidity sensors plugged I2C"
        }]
      }],
     "port": [],
     "physical_port": "/dev/ttyACM0",
     "device_type": "arduino",
     "needDeployer": true
}
```


Listing 2 – An example of Infrastructure component

`Components` are connected through two kinds of `Ports`: communication ports and execution ports. A communication port represents a communication interface of a component. A `ProvidedCommunicationPort` provides a feature to another component (*e.g.*, MQTT provides an interface on port 1883, see Listing 1), while a

`RequiredCommunicationPort` consumes a feature from another component. The property `isMandatory` of `RequiredCommunicationPort` represents that the `InternalComponent` depends on this feature (*e.g.*, the ProxyController component hosted on RPI-3 will not work if the communication with the PLC is not properly set up, see Listing 3). The property `portNumber` represents the logical port that can be used to interact with the component.

```
{
    "name": "Proxy_to_TempAndHumidity",
    "properties": [],
    "src": "/ProxyController/53a937cd-492c-a1d8-dc2ea16d8154",
    "target": "/TempAndHumidityReader/16826ee6-4349-894b-66cbfbbe5284",
    "isControl": true,
    "isDeployer": true,
    "isMandatory": false
}
```

Listing 3 – An example of Communication

An execution port represents the execution interface of a component (*i.e.*, the execution environment offered by a component to other components). A `ProvidedExecutionPort` represents that the component provides execution environment facilities (*e.g.*, Arduino-1 provides an execution environment for the Temp&HumiditySensorReader sketch, see Listing 2), while a `RequiredExecutionPort` represents that the internal component requires an execution environment from another component (*e.g.*, the SmartEnergyApp requires hosting from a Docker engine). Only internal components can have a `RequiredExecutionPort` since they are managed by GeneSIS.

There are two main types of `Link`s: `Hosting`s and `Communication`s. A `Hosting` depicts that an `InternalComponent` will execute on a specific host. This host can be any component, meaning that it is possible to describe the whole software stack required to run an `InternalComponent`. A `Hosting` can be associated with `Resource`s specifying how to configure the components so that the contained component can be deployed on the container component. A `Communication` represents a communication binding between two `SoftwareComponent`s. A `Communication` can be associated with `Resources` specifying how to configure the components so that they can communicate with each other. Finally, the property `isDeployer` specifies that the `InternalComponent` (one of the endpoint of the `Communication`) hosted on an `InfrastructureComponent` with the `needDeployer` property should be deployed from the host of the other `SoftwareComponent` (the other endpoint of the `Communication`) (*e.g.*, the artefact to be executed on Arduino-1 will be deployed from the RPI-2). This property is important as several host may have a local access to the host with limited Internet access but only one should run the deployment agent. The property `isLocal` indicates that the source and target of the communication have to be deployed on the same host.

GeneSIS is an extensible language. Subtypes of `SoftwareComponents` can be added to the GeneSIS modelling language in the form of plugins, in order to support user-defined elements for deployment. These plugins can be dynamically loaded in the supporting execution engine.

3.2 Specific Support for Security

The concept of `Capability` can be used to specify that a component provides or requires a specific feature. Capabilities are used to validate that one component is fulfilling the requirements from another one. A `Capability` is defined by a

`description` and a `controlId`, which is a unique identifier for different type of capabilities. `Capabilities` are attached to `Ports`. A required `Port` may require `SecurityCapabilities` (*i.e.*, the SmartEnergyApplication can only be accessed with proper authorizations, see Listing 4) (addressing R_5) and `ExecutionCapabilities` (*i.e.*, a specific execution environment or a feature is required for the component to execute). By contrast, a provided `Port` may offer `SecurityCapabilities` and `ExecutionCapabilities`. For a deployment model to be valid, all the required capabilities must match a provided capability.

```
{
    "name": "fe1fdedf-2736-4c1e-97ee-a4584852bf4d",
    "capabilities": {
        "_type": "/capability/security_capability",
        "name": "Access_control",
        "control_id": "AC1/OAuth",
        "description": "External services can only access the app when authenticated"
    },
    "port_number": "1880"
}
```

Listing 4 – An example of Port with SecurityCapabilities

The ports attached to an `InfrastructureComponent` can expose a set of `hardwareCapabilities`, which represents the interfaces toward specific hardware facilities attached to the component (*i.e.*, the temperature and humidity sensors attached to Arduino-1 are accessible via I2C, see Listing 2). This is important as *(i)* the software component that will use the hardware facility must know how to access it and *(ii)* in case a software component is using a specific interface for accessing a hardware facility we must ensure that the required interface matches what is offered by the `InfrastructureComponent`.

Finally, the property `isController` depicts that the `SoftwareComponent` associated to the attribute is controlled by the other (*e.g.*, all messages going to the Arduino should pass through the service hosted on RaspberryPi, see 3).

4 The GeneSIS Deployment Engine

From a deployment model specified using the GeneSIS Modelling language, the GeneSIS deployment engine is responsible for: *(i)* provisioning cloud resources, *(ii)* deploying the `SoftwareComponents`, *(iii)* setting up communication between them, and *(iv)* monitoring the status of the deployment.

4.1 Overall Architecture

The GeneSIS deployment engine implements the Models@run.time pattern to support the dynamic adaptation of a deployment with minimal impact on the running system (addressing R_4). Models@run.time [BBF09] is an architectural pattern for dynamic adaptive systems that leverage models as executable artefacts that can be applied to support the execution of the system. Models@run.time provides abstract representations of the underlying running system, which facilitates reasoning, analysis, simulation, and adaptation. A change in the running system is automatically reflected in the model of the current system. Similarly, a modification to this model is enacted on the running system on demand. This causal connection enables the continuous evolution of the system with no strict boundaries between design- and run-time activities (addressing R_2).

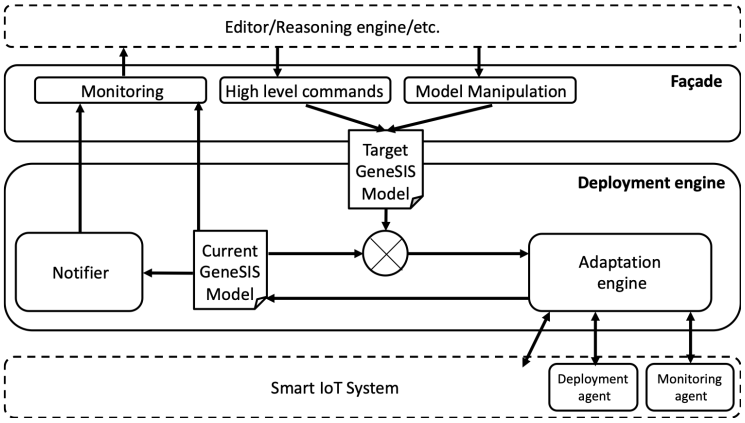


Figure 3 – The GeneSIS models@run.time architecture

Our engine is a typical implementation of the Models@run.time pattern. When a target model is fed to the deployment engine, it is compared (see Diff in Figure 3) with the GeneSIS model representing the running system. Finally, the adaptation engine enacts the adaptation (*i.e.*, the deployment) by translating the difference between the current and the target models into one or several of the following deployment steps. After the deployment, the engine synchronize the current GeneSIS model with the actual deployment result.

Overall a deployment process typically consists in the following steps:

1. **Check infrastructure**: This step consists in checking if the hosts specified in the deployment model are reachable (*e.g.*, is the docker remote API accessible at the address specified in the deployment model).
2. **Provision and instantiate resource**: In the case of cloud solutions, this step consists in provisioning the cloud resources and running the proper execution environment as specified in the deployment model. For container technologies, this step consists in pulling the image of the container and running it with the set up specified in the deployment model (*e.g.*, access to file system, specifying open ports).
3. **Set up host environment**: This step consists in preparing the environment for installation and configuration. In particular, environment variables can be exported to expose specific data or deployment information (*e.g.*, IP address of a virtual machine provisioned during deployment, ports numbers).
4. **Installation and configuration**: This step consists in running scripts and commands to configure and install software on the host. This includes ensuring that the software components in the deployment topology can communicate with each other.
5. **Start**: This step consists in starting the deployed software artefacts.

When decided by the end-user, the GeneSIS deployment engine can deploy on a target `InfrastructureComponent` a monitoring agent. This agent is an instance

of netdata[3] and provide information about the performance and health status of the `InfrastructureComponent`, including data about the `SoftwareComponents` it hosts.

Finally, the deployment engine can delegate part of its activities to deployment agents running on the field (see Section 4.2.2 for more details).

In the following subsections, we detail the specific support offered by GeneSIS for *(i)* deployment on Edge and IoT infrastructures, including devices with no or limited access to the Internet, and *(ii)* security.

4.2 Support for Continuous Deployment on IoT and Edge Infrastructures

GeneSIS provides the following support for the continuous deployment of Software components on IoT and Edge devices.

4.2.1 ThingML Components for Platform and Hardware Independent Deployable Artefacts

Contrary to the Cloud, IoT and Edge resources are typically very heterogeneous and cannot always benefit from the abstraction offered by virtualization techniques as they potentially provide limited computing and storage capacities. The software components to be deployed on such resources are typically tailored for them (in term of optimization and programming language, etc.), making their reuse difficult. From the GeneSIS perspective, this dramatically reduces the possibilities to adapt a deployment (*e.g.* migration of component from one host to another) and to reuse `SoftwareComponents` across deployments, leading to an explosion in the combinatory of component type (potentially one component type per IoT and Edge device).

To overcome this issue, GeneSIS offers specific support for the deployment of ThingML components. It is worth noting that *(i)* a ThingML component is a regular `InternalComponent` (it inherits from `InternalComponent`) and *(ii)* GeneSIS is not bound to ThingML components (*i.e.*, GeneSIS can be used without).

ThingML is an open source IoT framework that includes a language and a set of generators to support the modelling of system behaviours and their automatic derivation across heterogeneous and distributed devices on the IoT and edge spaces. The ThingML code generation framework has been used to generate code in different languages, targeting around 10 different target platforms (ranging from tiny 8 bit microcontrollers to servers) and 10 different communication protocols [HFMH16]. ThingML models can be platform specific, meaning that they can only be used to generate code for a specific platform (for instance to exploit some specificities of the platform); or they can be platform independent, meaning that they can be used to generate code in different languages.

The deployment of a ThingML `InternalComponent` by GeneSIS, not only consists in the deployment of the code generated by ThingML on a specific platform, but also in the actual generation of this code. The GeneSIS deployment engine proceed as follows. It first identifies the platform on which the ThingML `InternalComponent` should be deployed. Then it consumes the ThingML models attached to the component and uses ThingML to generate the code for the identified platform. If required, the generated code is further built and packaged before being deployed. Thanks to this mechanism, a ThingML `InternalComponent` can easily be migrated from one host to another. In other words, this means that the same ThingML code can be dynamically migrated from one device and platform to another without necessarily relying on a virtualization technology for lower footprint.

[3] https://github.com/netdata/netdata

4.2.2 Deployment Agents

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) and for specific reasons (*e.g.*, security) the deployment of software components can only be performed via a local connection (*e.g.*, a physical connection via a serial port). In such case, the actual action of deploying 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 (addressing R_7). It is generated dynamically by GeneSIS based on the artefact to be deployed and its target host. It is implemented as a Node-RED application. The typical architecture of a deployment agent is depicted in Figure 4.

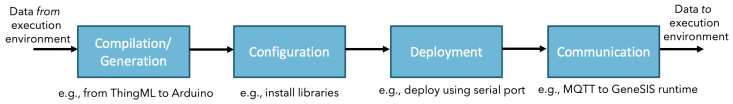


Figure 4 – Architecture of a deployment agent

This architecture is inspired by the four main steps of a deployment procedure and is implemented with Node-RED nodes from the following four groups.

Code generation nodes. The aim of this type of node is to generate, from source code or specification languages, the code or artefact to be deployed on a target device. In the context of our motivating example, we created a ThingML compilation node, which consumes ThingML models and generates code in a specific language. The desired language is specified as a property of the node (*e.g.*, Arduino sketch in our example). The code generation is achieved by using the ThingML compiler. In order to trigger a compilation, code generation nodes consume as input a `start compilation` message. Once the compilation is successfully completed, they send a `generation success` message that includes the location of the generated code. Finally, a `compile on start` property can be set to true enabling to trigger the compilation when the node is instantiated. By contrast, the deletion of an instance of the node results in the deletion of the generated code.

Deployment configuration nodes. This type of node aims at preparing the actual deployment of a software component (being generated by code generation nodes or not). This typically consists in generating configuration files. For instance, we created a Docker deployment configuration node that generates a “docker-compose" file as well as the relevant Dockerfile files depending on the target device. These nodes typically consume messages from the code generation nodes – *i.e.*, `generation success` messages that include details about the location of the artefact to be deployed. The retrieval of such a message triggers the actual generation of the configuration file. Once this process is completed, it generates a message containing the location of both the artefact to deploy and the configuration files. Removing an instance of configuration nodes results in the deletion of all the configuration files it has generated.

Deployment nodes. These nodes aim at enacting the deployment of a software component on a specific target. In our motivating example, we created an Arduino deployment node that *(i)* builds and uploads an Arduino sketch on the Arduino board

using the Arduino CLI[4] and *(ii)* installs the libraries required for its proper execution. These nodes typically consume messages from the configuration nodes. Removing an instance of a deployment node results in the termination of the deployed software (*e.g.*, killing a docker container, or deploying a dummy Arduino sketch).

Communication nodes. After deployment, it can be important to communicate with the deployed software artefacts, for instance to monitor the status of a deployment. Communication nodes are regular Node-RED I/O nodes such as serial port for Arduino board or HTTP requests for REST services. In any case, a MQTT node is added to the agent and is used to provide GeneSIS with information about the status of a deployment (*i.e.*, the agent inform the GeneSIS deployment engine about the state and result of a deployment). Thanks to this modularity, components from each of these groups can be seamlessly and dynamically composed for different types of deployments. Once a deployment agent has successfully completed its activities (*i.e.*, the deployment is completed) the GeneSIS deployment engine automatically terminates the agent.

4.3 Support for Security

The GeneSIS deployment engine provides not only general support for deploying security components as any other components but also customised support (for R_6).

4.3.1 General Support for Security Components

GeneSIS supports the deployment of security components as any other software components. Their deployment and configuration are managed using `Resource`s, meaning they can be configured via exposed APIs and configuration files. GeneSIS offers a library of off-the-shelf security components that can be selected for instantiation in the deployment model. For example, built-in cryptography components can be selected and configured to provide secure communication between IoT components using SSL/TLS. Another example is the security API gateway presented in our motivating example (see Section 2), which is configured to secure the access to the smart energy and user comfort applications. A security component to be deployed together with an IoT application can be declared in GeneSIS with `SecurityCapabilities` in a provided port. A required port of a `SoftwareComponent` that requires a matching `SecurityCapabilities` can be bound with the provided port of the security component that provides such `SecurityCapabilities`. Before enacting a deployment, the GeneSIS deployment engine validates the correctness of the provided deployment model. In particular, it ensures that all the required `SecurityCapabilities` match a provided `SecurityCapability`.

4.3.2 Reusable Security Components for IoT Platforms

IoT systems are typically built on top of IoT platforms such as SMOOL or FIWARE[5], which often act as an intermediary for the communications between the things within the IoT environment. They provide a proper ground for building mechanisms, in different applications and scenarios, to control and monitor these communications.

GeneSIS provides a generic sub-type of `InternalComponent` for the deployment of security mechanisms and policies built on top of such IoT platforms. In this paper, we present its application to the SMOOL IoT platform, which is used in our motivating

[4] `https://playground.arduino.cc/Learning/CommandLine`
[5] `https://www.fiware.org`

example. SMOOL [NRS14] is an open-source middleware for IoT smart spaces, developed and maintained by Tecnalia. This platform consists of a server (Semantic Information Broker - SIB) and tools for creating clients (or Knowledge Processors - KPs) in Java, C++ and other languages. The created clients can communicate by using different protocols such as TCP, Bluetooth or WebSockets. The structure of the messages exchanged among clients follows the SMOOL OWL ontology.

The SMOOL ontology embeds the necessary concepts to attach metadata to message payloads, in particular related to the following security concepts: Authentication, Authorization, Confidentiality, Integrity, and Non-repudiation. For each of these, a security type and a security data can be defined (*e.g.*, for Authorization, OAuth 2.0 can be a security type and the access token can be the security data). By using security properties as semantic data, SMOOL provides a mean to easily embed security monitoring and control to existing IoT environments (security-by-design). DevOps teams can define SMOOL clients that leverage these security properties to check and enforce security concepts on messages requiring security controls. This can be done with the SMOOL clients built-in security metadata checker to verify messages exchanged 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 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.

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. Once the client is ready, s/he can specify how to deploy it also indicating 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 as a subtype of `Internal-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 and *(ii)* the SMOOL client business logic. The main rationale behind this choice is 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. Similar approach could be applied to other IoT platform. When deploying this component, the GeneSIS deployment engine injects the security policy into the ThingML code of the SMOOL client, before it compiles this code to generate the actual implementation of the SMOOL client with the corresponding security policy. Finally, the application is built and deployed. This component exposes a property `securityPolicy` that represents the security policy to be implemented by the security

mechanism (*i.e.*, the policy to be adopted by the SMOOL security metadata checker). In the case of our motivating example, we control the actuation orders to change the position of the blinds. Every actuation order must be accompanied by a valid security key before executing the order as depicted in Listing 5. 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.

```
{
   "_type": "/internal/SMOOLSecurity",
   "name": "SecurityEnforcer",
    ...
    "security_Policy": [["BlindPositionActuator", "Authorization"]],
    ...
}
```

Listing 5 – Security policy

5 Synthesis

In this section, we discuss how our approach addresses the requirements defined in Section 2 and we report on the use of GeneSIS in the context of the Kubik laboratory[6]. The Kubik test facility is a three floors smart building owned by Tecnalia and designed for testing and research. This infrastructure allows to develop and validate innovative products and systems to optimize energy efficiency in buildings, from its conceptualization to its implementation. The IoT system deployed in the laboratory is similar to the one presented in our motivating example. The GeneSIS deployment model of the system is depicted in Figure 5 using the graphical syntax of GeneSIS. Excerpt of the deployment model in the textual syntax are presented in Section 3, whilst the whole deployment model can be found in the GeneSIS code repository[7]. In the following, we provide details about the actual implementation of the smart building's components.

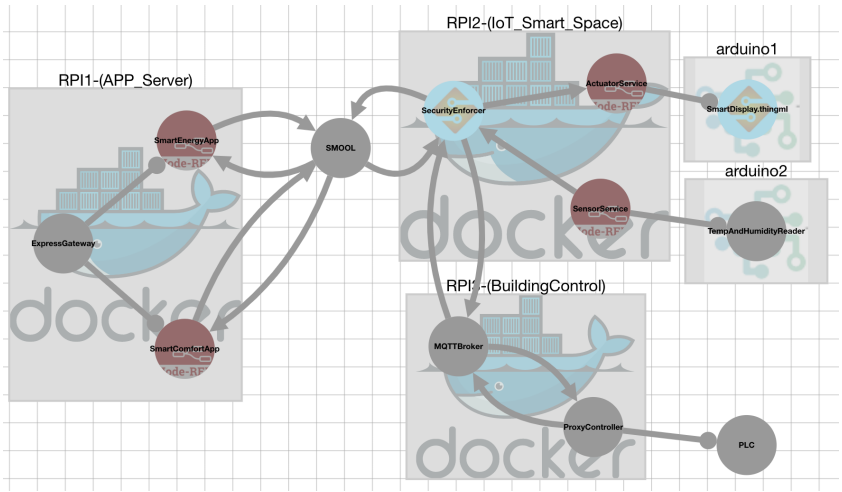


Figure 5 – GeneSIS deployment model

[6] https://www.tecnalia.com/images/stories/Catalogos/CAT_KUBIK_EN_dobles.pdf
[7] https://gitlab.com/enact/GeneSIS/docs/examples

The `Temp&HumiditySensorReader.ino` is an Arduino program that reads the temperature and humidity values from the Grove - Temperature&Humidity Sensor Pro and sends the data to the Sensor Service via serial port. The `SmartDisplay.thingml` is a ThingML program to display messages on the TFT LCD Arduino display. The Sensor Service is a NodeJS application that receives the temperature and humidity data. Similarly, the `ProxyController` component is implemented in NodeJS. The user comfort and smart energy applications are implemented using Node-RED, whilst the `ExpressGateway` component is an instance of the NodeJS Express API Gateway framework[8]. The SMOOL server is deployed in the Cloud and is implemented in Java. The SCADA exposes an MQTT interface and is represented as the MQTT software component. Finally, the Security Enforcer is an application generated by using ThingML for deployment of security behaviour, and SMOOL for connectivity and application logic.

Separation of concerns and reusability (R_1):

As depicted in Figure 5, several component types were reused (*e.g.*, Node-RED components) in the deployment model. In addition, new component types were created and integrated to the GeneSIS component repository (*e.g.*, ExpressGateway, MQTTBroker). These components are typical candidates for reuse in other scenarios. The component-based design of the GeneSIS modelling language ensures that the provisioning and deployment models are modular and loosely-coupled. In addition, the type-object pattern in the metamodel of the GeneSIS modelling language ensures that component types can be reused across several models.

Abstraction and Infrastructure independence (R_2):

In the context of Kubik we demonstrated the deployment of software component over the whole IoT, Edge and Cloud space. By leveraging MDE techniques, the GeneSIS modelling language offers a single domain-specific modelling language and abstraction that enables the management of application deployed on IoT, Edge, and Cloud infrastructure. Independently of IoT layers, these resources as well as the software components can be abstracted in a homogeneous way as components. In addition, by applying the Models@run.time pattern, the GeneSIS execution environment provides an abstract and up-to-date representation of the running system that can be dynamically manipulated.

White- and black-box infrastructure (R_3):

In the context of Kubik, both the SMOOL server and the PLC are considered as `ExternalComponent` not managed by GeneSIS. Nevertheless, it is possible to seamlessly orchestrate these components with all the other `InternalComponents`. The GeneSIS modelling language embeds the necessary concepts for the GeneSIS execution environment to distinguish and orchestrate white-box (*i.e.*, resources on top of which GeneSIS can manage a software stack) and black-box resources (*i.e.*, resources coming with a software stack that cannot be manipulated). More specifically, we refer here to the concept of `InternalComponent` and `ExternalComponent`, respectively.

Automation and adaptation (R_4):

From a deployment model, GeneSIS supports the fully automated deployment of a SIS. By applying the Models@Run-time pattern GeneSIS provides developers

[8] `https://www.express-gateway.io/`

with the means to adapt the deployment of a SIS. In our evaluation we updated the deployment by adding the User Comfort application, the ExpressGateway, and the SecurityEnforcer component. Thanks to the adoption of the Models@run.time pattern, only these parts of the application were adapted.

Specify Security Requirements (R_5):
As depicted in Section 3, when modelling the deployment of the Kubik smart IoT system, several required and provided `SecurityCapabilities` were specified. In particular related to the need for the `ExpressGateway` and the `SecurityEnforcer`, GeneSIS provides mechanisms for specifying security and privacy capabilities provided and required by a component.

Automatic Deployment and Enforcement of Generic Security Mechanisms (R_6):
GeneSIS offers support for the deployment of security mechanisms. In particular, a specific component type can be used for the generic deployment of security monitoring and control mechanisms on IoT platforms. In the context of Kubik, the RPI3 (Building Control) and the two Arduinos depicted in Figure 5 cannot be accessed without proper authorization. The smart building apps are built on top of the SMOOL platform and uses clients implemented in Java. The GeneSIS-SMOOL security component was used to deploy the `SecurityEnforcer` component, automatically injecting this security policy into the SMOOL client. The `SecurityEnforcer` will allow passing only messages of type "BlindPositionActuator" containing some specific data in the Authorization concept. The `security_policy` property of the `SecurityEnforcer InternalComponent` was thus specified as indicated in Listing 5. The security policy was injected in the ThingML code of the SMOOL client and compiled to Java, in turn. The proper Maven manifest was automatically created and used to build the application before deployment.

Devices with no Internet Connection (R_7):
By leveraging the concept of deployment agent, the GeneSIS execution environment can be extended to Edge devices, which, in turn, can deploy software components on devices only accessible locally. In the context of Kubik, the two Arduinos in the building do not offer remote access and can only be accessed via their serial connections with RPI-2. The deployment of the software components on these devices must be enacted by a deployment agent. Figure 6 depicts the deployment agent generated to deploy the `SmartDisplay.thingml` component. From the left to the right, the first node is a code generation node, which generates Arduino code from a ThingML program. The second is a deployment node responsible for deploying the generated Arduino code onto the Arduino board. The third node is used to format the status message, which is sent back to the GeneSIS execution engine by the last node.

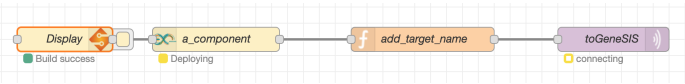


Figure 6 – Deployment agent generated by GeneSIS

By addressing all these requirements, and in particular R_6 and R_4, GeneSIS provides support for the DevSecOps of SIS. In the context of the smart building, once the User Comfort application has been added, its security policy has also been enforced in the `SecurityEnforcer`. Later, to enable remote accesses via authorised calls to some services of the Smart Energy and the User Comfort applications, an

Express Gateway has been added and deployed by GeneSIS. These function and security updates show how GeneSIS supports DevSecOps for the evolution of the smart building.

6 Related Work

Software deployment has been evolving from deployment of component-based commercial desktop software [PDC01], deployment of component-based distributed applications [Gro06], to deployment on Cloud resources, and more recently deployment for IoT systems over IoT, Edge, and Cloud infrastructure. Even though some core concepts from deployment of component-based applications such as *capability*, *port* in [Gro06] 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. For some years now, multiple tools have been available on the market to support the deployment and configuration of software systems, *e.g.*, Puppet[9], Chef[10]. These tools were first defined as configuration management tools aiming at automating 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 cloud solutions from different providers) such as CloudMF [FCS+18], OpenTOSCA [dSBK+16], Cloudify[11], and Brooklyn[12]. 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[13], Kubernetes[14]. Opposed to hypervisor virtual machines, containers such as Docker containers leverage lightweight virtualization technology, which executes directly on the operating system of the host. As a result, Docker shares and exploits a lot of the resources offered by the operating system thus reducing containers' footprint. These characteristics mean container technologies are not only relevant for Cloud but can also be used on Edge devices.

Besides, few tools such as Resin.io and ioFog 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) [ADR09] and the IoT deployment model of GeneSIS share some basic concepts. The SCA is an open architecture that specifies a standardized infrastructure for a software-defined radio (SDR). The SDR SCA specification has commonalities with our

[9] `https://puppet.com/`
[10] `https://www.chef.io/chef/`
[11] `http://cloudify.co/`
[12] `https://brooklyn.apache.org`
[13] `https://docs.docker.com/compose/`
[14] `https://kubernetes.io`

GeneSIS approach in particular the concepts of components, ports, and resources [ADR09]. 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 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 (*i.e.*, what GeneSIS is able to do with deployment agents).

In [NFE+19], we conducted a systematic literature review (SLR) to systematically reach a set of 17 primary studies of orchestration and deployment for IoT. 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 [MBS+17], WComp [LRRT15], or D-LITE [CGDLR11], D-NR [GBLL15] all rely on their specific run-time environment where mechanisms such as dynamic component loading or class loading are typically used. Contrary to these approaches, GeneSIS does not rely on a specific bootstrap but instead leverage common run-time environments such as Docker, Node.js, SSH.

To the best of our knowledge, none of the approaches and tools aforementioned have specifically been designed for supporting deployment over 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.

7 Conclusion

In this paper, we have presented the GeneSIS modelling language and framework. GeneSIS enables the continuous deployment and orchestration of Smart IoT Systems (SIS) by leveraging upon model-driven techniques and methods. The GeneSIS modelling language supports the platform-independent specification of SIS deployment including software pieces installation and configuration over IoT, Edge, and Cloud resources, which are modelled in an endogenous way through the concept of component. The language expressiveness has been optimised to specify: IoT components, security concerns, dependencies between deployable artefacts, as well as how and from where they should be deployed. The associated Models@Run.time deployment environment provides mechanisms for the dynamic provisioning, deployment, and adaptation of SIS, including specific support for the deployment of security mechanisms. This environment can delegate part of its duty to deployment agents, that are dynamically generated and can execute on the field when devices cannot be accessed remotely. Our case study shows how GeneSIS enables the security-by-design of SIS that are able to modulate a variety of security features at operation depending on real needs. The system requirements may change during its lifetime and still the re-deployment by GeneSIS of enhanced security features into already running applications makes it possible that SIS adapt to evolving conditions and threats while keeping their trustworthiness.

Acknowledgments The research leading to these results has received funding from the European Commission's H2020 Programme under grant agreement numbers 780351 (ENACT).

References

[ADR09] C. R. Aguayo Gonzalez, C. B. Dietrich, and J. H. Reed. Understanding the software communications architecture. *IEEE Communications Magazine*, 47(9):50–57, 2009.

[AK02] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321, 2002.

[BBF09] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[BBF+18] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. A systematic review of cloud modeling languages. *ACM Comput. Surv.*, 51(1):22:1–22:38, February 2018.

[CGDLR11] Sylvain Cherrier, Yacine M Ghamri-Doudane, Stéphane Lohier, and Gilles Roussel. D-lite: Distributed logic for internet of things services. In *2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 16–24. IEEE, 2011.

[Dea07] Alan Dearie. Software deployment, past, present and future. In *Future of Software Engineering, 2007. FOSE'07*, pages 269–284. IEEE, 2007.

[dSBK+16] Ana C Franco da Silva, Uwe Breitenbücher, Kálmán Képes, Oliver Kopp, and Frank Leymann. 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*, pages 181–182. ACM, 2016.

[ea15] Andreas Metzger et al. Cyber physical systems: Opportunities and challenges for software, services, cloud and data. NESSI white paper, 2015.

[FCS+18] Nicolas Ferry, Franck Chauvel, Hui Song, Alessandro Rossini, Maksym Lushpenko, and Arnor Solberg. Cloudmf: Model-driven management of multi-cloud applications. *ACM Transactions on Internet Technology (TOIT)*, 18(2):16, 2018.

[FDG+20] Nicolas Ferry, Jacek Dominiak, Anne Gallon, Elena González, Eider Iturbe, Stéphane Lavirotte, Saturnino Martinez, Andreas Metzger, Victor Muntés-Mulero, Phu H. Nguyen, Alexander Palm, Angel Rego, Erkuden Rios, Diego Riviera, Arnor Solberg, Hui Song, Jean-Yves Tigli, and Thierry Winter. Development and operation of trustworthy smart IoT systems: The ENACT framework. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 121–138, Cham, 2020. Springer International Publishing.

[FNS+19] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Genesis: Continuous orchestration and deployment of smart iot systems. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875. IEEE, 2019.

[FSS+18] Nicolas Ferry, Arnor Solberg, Hui Song, Stéphane Lavirotte, Jean-Yves Tigli, Thierry Winter, Victor Muntés-Mulero, Andreas Metzger, Erkuden Rios Velasco, and Amaia Castelruiz Aguirre. Enact: Development, operation, and quality assurance of trustworthy smart iot systems. In *DevOps'18 International workshop*, pages 112–127. Springer, 2018.

[GBLL15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. Developing iot applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 155–162. IEEE, 2015.

[Gro06] Object Management Group. Deployment and configuration of component-based distributed applications specification. *OMG Available Specification Version 4.0 formal/06-04-02*, 2006.

[HFMH16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 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, page 125–135, New York, NY, USA, 2016. Association for Computing Machinery.

[LRRT15] Stéphane Lavirotte, Gaëtan Rey, Gérald Rocher, and Jean-Yves Tigli. A generic service oriented software platform to design ambient intelligent systems. In *Proceedings of the 2015 ACM International Conference on Pervasive and Ubiquitous Computing*, pages 281–284. ACM, 2015.

[MBS+17] Amardeep Mehta, Rami Baddour, Fredrik Svensson, Harald Gustafsson, and Erik Elmroth. Calvin constrained-a framework for iot applications in heterogeneous environments. In *37th International Conference on Distributed Computing Systems*, pages 1063–1073. IEEE, 2017.

[MCP17] Håvard Myrbakken and Ricardo Colomo-Palacios. Devsecops: A multivocal literature review. In Antonia Mas, Antoni Mesquida, Rory V. O'Connor, Terry Rout, and Alec Dorling, editors, *Software Process Improvement and Capability Determination*, pages 17–29, Cham, 2017. Springer International Publishing.

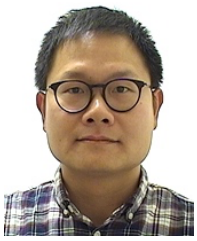
[NFE+19] Phu H. Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Advances in deployment and orchestration approaches for IoT - a systematic review. In *2019 IEEE International Congress On Internet of Things (ICIOT)*, pages 53–60. IEEE, 2019.

[NRS14] Adrian Noguero, Angel Rego, and Stefan Schuster. Towards a smart applications development framework. *Social Media and Publicity*, 27, 2014. URL: `https://bitbucket.org/jasonjxm/smool,2011-2020`.

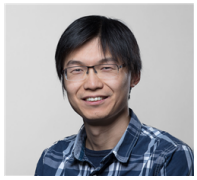
[PDC01] Allen Parrish, Brandon Dixon, and David Cordes. A conceptual foundation for component-based software deployment. *Journal of Systems and Software*, 57(3):193 – 200, 2001.

About the authors

Nicolas Ferry is a Research Scientist at the Secure IoT Software group, SINTEF Digital. He holds a Ph.D. degree from the University of Nice. His research interest includes model-driven engineering, domain-specific languages, Internet of Things, cloud-computing, self-adaptive systems, and dynamic adaptive systems. He has actively contributed to various national and international research projects such as the REMICS, CITI-SENSE, MC-Suite and MODAClouds EU projects, and is Technical Manager of H2020 ENACT project. He has also served as a program committee member of international conferences and workshops. Contact him at `nicolas.ferry@sintef.no`.



Phu H. Nguyen is a Research Scientist at SINTEF, as a member of the Secure IoT Software group with a focus on tools and methodologies for software development and operation of heterogeneous and autonomous, yet secure and privacy-aware systems spanning across the Cloud, the Edge, and the IoT. He has experience from working in international research projects as well as research and development projects with industry in Norway. He has a very international education and research background, from Vietnam (BSc) to the Netherlands (MSc), Luxembourg (Ph.D.), and Norway. He is also an active reviewer and PC member of high-impact journals, conferences, and workshops. Contact him at `phu.nguyen@sintef.no`.



Hui Song is Research Scientist with the Secure IoT Software group, SINTEF Digital. His research interests are focused on the software engineering practices and tools, and the application of them on Cloud and IoT. He received his PhD from Peking University. Before joining SINTEF, he has working experience in the National Institute of Informatics (NII), Japan, and Trinity College Dublin, Ireland. Contact him at `hui.song@sintef.no`.

Erkuden Rios , after working six years for Ericsson Spain, currently she is senior scientist of Cybersecurity research team at TECNALIA, Spain. She is currently the coordinator of the Security WP in the H2020 ENACT project on Secure and Privacy-aware Smart IoT Systems as well as in the H2020 SPEAR project on Secure Smart Grids. Previously, she was the coordinator of the H2020 MUSA project on Multi-cloud Security, successfully ended in 2017, as well as the chair of the Data Protection, Security and Privacy in Cloud Cluster of EU-funded research projects, launched by DG-CNECT in April 2015. Furthermore, she has worked in multiple large European and Spanish projects on cybersecurity and trust such as POSEIDON, PDP4E, TACIT, RISC, ANIKETOS, SWEPT, CIPHER and SHIELDS. Her main research interests include Trust and Security, Risk Management, and AI for Cybersecurity. Contact him at `erkuden.rios@tecnalia.com`.

Eider Iturbe is a senior researcher of Cybersecurity team at TECNALIA, Spain. Eider graduated in Telecommunication Engineering from the University of the Basque Country (Spain) and in the European Master of of Project Management at the same university. Before joining Tecnalia in 2009, she worked for software consultancy firms where she acquired management skills and a great technical expertise in the cyber security field. She has worked in multiple large European and Spanish projects on cybersecurity, privacy and trust such as SPEAR, ENACT, SPARTA, POSEIDON, PDP4E, TACIT, and MUSA. Her main research interests include Cyber Security, Privacy, Risk Management, and AI for Cybersecurity. Contact him at `eider.iturbe@tecnalia.com`.

Satur Martinez is a Cybersecurity Researcher at TECNALIA, Spain. His research interests are focused on Cybersecurity in all its aspects, Machine Learning and Big Data where he accumulates more than 20 years of experience. Before joining to Tecnalia, he worked at Panda Security as cybersecurity researcher and threat hunter. Contact him at `satur.martinez@tecnalia.com`.

Angel Rego works as a senior scientist for the Cybersecurity research team at TECNALIA, Spain. He has been involved in many EU-funded research and development projects related with real time data collecting, analysis and managing, web applications, microservices, Internet of Things and Cyber Physical Systems. Mr. Rego is co-author of the patent EP3144841A1 (System, method and device for preventing cyber attacks). Contact him at `angel.rego@tecnalia.com`.