GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems

Nicolas Ferry, Phu H. Nguyen, Hui Song
SINTEF
Oslo, Norway
Email: name.surname@sintef.no

Pierre-Emmanuel Novac,
Stéphane Lavirotte, Jean-Yves Tigli
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
Email: name.surname@unice.fr

Arnor Solberg
Tellu IoT AS
Asker, Norway
Email: name.surname@tellu.no

***Abstract*—Multiple tools have emerged to support the development as well as the continuous deployment of cloud-based software systems. However, currently, there is a lack of proper tool support for the continuous orchestration and deployment of software systems spanning across the IoT, edge, and cloud space. In particular, there is a lack of languages and abstractions that can support the orchestration and deployment of software services across vastly heterogeneous IoT infrastructures. In this paper, we present a tool supported framework for the continuous orchestration and deployment of IoT systems, named GENESIS. In particular, GENESIS enables to cope with the heterogeneity at each of the IoT, edge, and cloud levels and allows to control the orchestration and continuous deployment of software systems that executes across IoT, edge, and cloud infrastructures.**

***Index Terms*—Deployment, model-driven engineering, domain-specific modelling language, models@run-time**

I. INTRODUCTION

Gartner envisions that 21 billion Internet-of-Things (IoT) endpoints will be in use by 2020[1], representing great business opportunities. Until recently, IoT system innovations have mainly been concerned with sensors, device management, and connectivity, with the mission to gather data for processing and analysis in the cloud to aggregate information and knowledge [1]. This approach has conveyed significant added value in many application domains but does not unleash the full potential of the IoT[2]. The next generation IoT systems need to perform distributed processing and coordinated behaviour across IoT, edge, and cloud infrastructures [2], manage the closed loop from sensing to actuation[3], and cope with vast heterogeneity, scalability, and dynamicity of IoT systems and their environments. This paper addresses them as Smart IoT Systems (SIS). SIS typically operate in a changing and often unpredictable environment. The ability of these systems to continuously evolve and adapt to their new environment is decisive to ensure and increase their trustworthiness, quality, and user experience. In particular, there is an urgent need for supporting the continuous orchestration and deployment of SIS over IoT, edge, and cloud infrastructures.

[1]Gartner (January 2017) - http://www.gartner.com/newsroom/id/3598917

[2]https://ec.europa.eu/digital-single-market/en/internet-of-things

[3]implying that IoT devices are not only applied for sensing the physical world, but are also exploited for processing and actuation

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, CloudML). However, very little effort has been spent on providing solution tailored for the delivery and deployment of applications across the whole IoT, edge, and cloud space [3]. 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 Internet [3].

To address these challenges, we have developed a framework for the continuous deployment of SIS. In this paper, we present our Generation and Deployment of Smart IoT Systems (GENESIS) Framework, allowing decentralized processing across heterogeneous IoT, edge, and cloud infrastructures. GENESIS includes: *(i)* a domain-specific modelling language to model the orchestration and deployment of SIS; and *(ii)* an execution engine that supports the orchestration of IoT, edge, and cloud services as well as their automatic deployment across IoT, edge, and cloud infrastructure resources. The main contributions of GENESIS that we present in this paper are the following:

1) It enables to cope with the heterogeneity across the 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 Internet.
2) Same language and tool are used for the continuous deployment of SIS (including the monitoring and dynamic adaptation) providing a unique model-based representation of the SIS for both design- and run-time activities

In the remainder, Section II describes the overall approach of the GENESIS Framework. Section III presents the GENESIS Modeling language while Section IV details the supporting execution engine. Section V presents our analyses on the state of the art of deployment and orchestration approaches for IoT Systems. Finally, Section VI highlights future research directions and Section VII concludes our presented work.

II. Overall approach

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 group for our framework is thus mainly software developers and architects. Figure 1 depicts the overall GeneSIS approach.

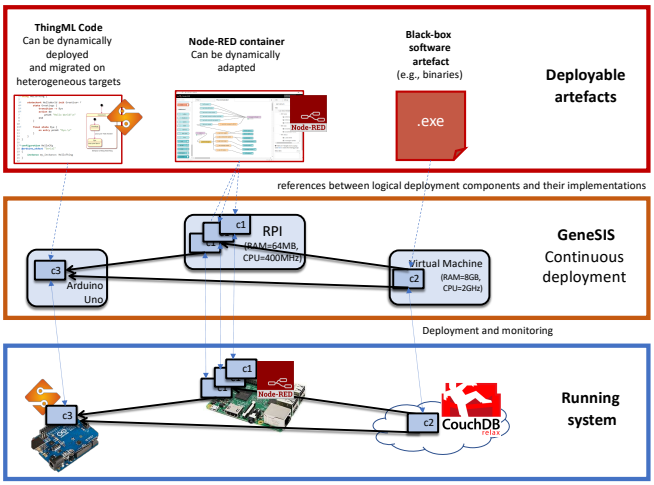

Fig. 1. Overall GeneSIS approach.

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 are the implementations of those components. This is often referred as deployable artefact [4]. 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). As depicted in the top layer of Figure 1, at the current moment, GeneSIS consumes as input three types of deployable artefacts:

- **Blackbox deployable artefact:** This refers to deployable artefacts that cannot be modified by GeneSIS (*e.g.*, GeneSIS can deploy a binary but cannot modify it). Our framework is agnostic to any development paradigm and technology, meaning that the developers can design and implement their blackbox deployable artefact based on their preferred paradigms and technologies. GeneSIS is also agnostic to any specific business domain.
- **ThingML source code:** ThingML [5] is a domain specific language for modelling distributed IoT systems including the behavior of the distributed components in a platform-specific or -independent way. From a ThingML code, the ThingML compiler can generate code in different languages, targeting around 10 different target platforms (ranging from tiny 8 bit microcontrollers to servers). This is particularly interesting for GeneSIS as, from a deployment model, the GeneSIS execution engine can identify the host to which a ThingML source code should be allocated and thus generate code in the relevant language before compiling and deploying it on the host. *This also provides* GeneSIS *with the ability to seamlessly migrate or deploy a ThingML program from one host to another.*
- **Node-RED container:** Using Node-RED, one can build an application as assembly of components executed in a Node-RED container, which can be dynamically adapted. *This provides* GeneSIS *with the ability to dynamically tailor an application to best fit its deployment.*

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 [6]. A deployment model is thus a connected graph that describes software components along with targets and relationships between them from a structural perspective [4]. A deployment configuration or deployment model typically includes the description of how its software components are integrated and communicate with each other. This is often referred to as software composition or orchestration. Software components represent either the deployable artefact and the resources on top of which these are deployed.

GeneSIS includes: *(i)* a domain-specific modelling language to specify deployment model – *i.e.*, the orchestration and deployment of SIS across the IoT, edge, and cloud spaces; and *(ii)* an execution engine to enact their actual orchestration and deployment.

Because it is not always possible for the GeneSIS execution engine to directly deploy software on all hosts (tiny devices do not always have direct access to Internet or even the necessary facilities for remote access), the actual action of deploying the software on the device has to be delegated to a host (*e.g.*, a gateway) locally connected to the device. The GeneSIS execution environment handles this problem by *(i)* generating a deployment agent responsible for deploying the software on the device with limited connectivity and *(ii)* deploying it on the host locally connected to the device with limited connectivity.

In the following sections we present the GeneSIS Modelling language before we detail its supporting execution engine.

III. The GeneSIS Modelling Language

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 is inspired by component-based approaches in order to facilitate separation of concerns and reusability. In this respect, deployment models can be regarded as assemblies of components. The type part of the GeneSIS modelling language metamodel is depicted in Figure 2.

In the following, we provide a description of the most important classes and corresponding properties in the GeneSIS metamodel as well as sample models in the associated textual syntax. The textual syntax better illustrates the various

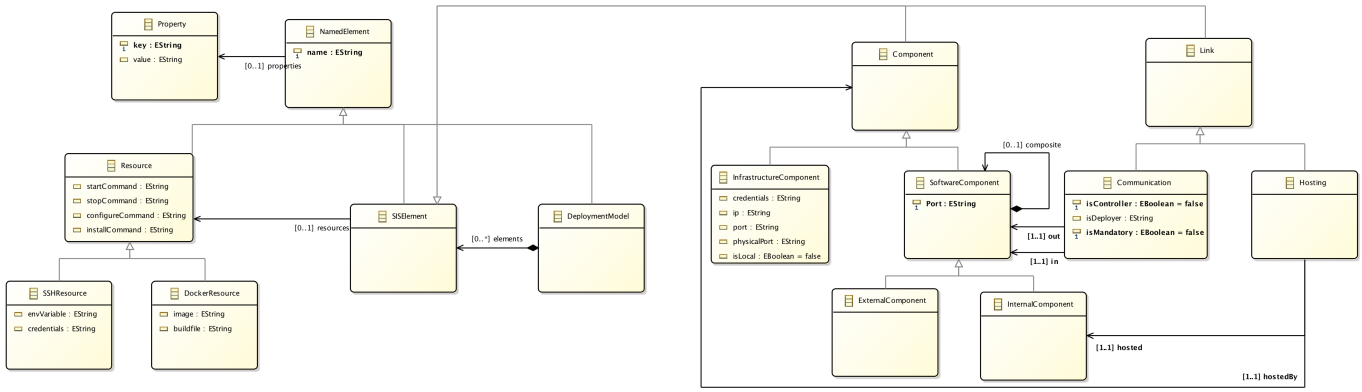

Fig. 2. Type part of the GENESIS language metamodel.

concepts and properties that can be involved in a deployment model, and that can be hidden in the graphical syntax.

A `DeploymentModel` consists of `SISElements`. All `SISElements` have a `name` and a unique `id`entifier. In addition, they can all be associated with a list of `properties` in the form of key-value pairs. The two main types of `SISElements` are `Components` and `Links`.

A `Component` represents a reusable type of node that will compose a `DeploymentModel`. A `Component` can be a `SoftwareComponent` representing a piece of software to be deployed on an host (*e.g.*, an Arduino sketch can be deployed on an Arduino board). A `SoftwareComponent` can be an `InternalComponent` meaning that it is managed by GENESIS (*e.g.*, an instance of Node-RED to be deployed on a Raspberry Pi), or an `ExternalComponent` meaning that it is either managed by an external provider (*e.g.*, a database offered as a service) or hosted on a blackbox device (*e.g.*, RFXCom transceiver). The property `port` of a `SoftwareComponent` represents logical ports (*e.g.*, port 1880 for Node-RED as depicted in Listing 1). 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 two main predefined types of resources: `DockerResources` (see Listing 1), and `SSHResources`.

Listing 1. An example of Internal component

```
{
   "_type": "node_red",
   "name": "ControlTemp",
   "properties": [],
   "id": "controltemp",
   "id_host": "RaspberryPi2",
   "docker_resource": {
      "name": "a resource",
      "image": "default",
      "command": "",
      "port_bindings": {
          "1880": "1880"
      },
      "devices": {
         "PathOnHost": "/dev/ttyUSB0",
         "PathInContainer": "/dev/ttyUSB0",
         "CgroupPermissions": "rwm"
      }
   },
   "port": ["1880"]
}
```

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 `isLocal` depicts that a local connection is required to deploy a `SoftwareComponent` on a `InfrastructureComponent` via a `PhysicalPort` (*e.g.*, the Arduino board can only be accessed locally via serial port, see Listing 2).

Listing 2. An example of Infrastructure component

```
{
   "_type": "device",
   "name": "Arduino",
   "properties": [],
   "id": "arduino",
   "ip": "127.0.0.1",
   "port": [],
   "physical_port": "/dev/ttyACM0",
   "device_type": "arduino",
   "isLocal": true
}
```

There are two main types of `Links`. 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 `Communication` can be associated with `Resources` specifying how to configure the components so that they can communicate with each other. The property `isMandatory` of `Communication` represents that the `SoftwareComponents` depend on this feature (*e.g.*, the service hosted on RaspberryPi2 will not work if the communication with RFXtrx433E is not properly set up). The property `isController` depicts that the `SoftwareComponent` associated to the `in` attribute is controlled by the other (*e.g.*, all messages going to the Arduino should pass through the service hosted on RaspberryPi). Finally,

the property `isDeployer` specifies that the `InternalComponent` hosted on the `InfrastructureComponent` with the `isLocal` property should be deployed from the host of the other `SoftwareComponent` (*e.g.*, the artefact to be executed on the Arduino will be deployed from the RaspberryPi). 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.

IV. The GeneSIS execution engine

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

A. Overall architecture

The GeneSIS execution engine can be divided into two main elements: *(i)* the facade and *(ii)* the deployment engine.

The facade provides a common way to programmatically interact with the GeneSIS execution engine.

GeneSIS follows a declarative deployment approach. From the specification of the desired system state, which captures the needed system topology, the deployment engine computes how to reach this state. It is worth noting that the deployment engine may not always compute optimal plans.

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. Models@Run-time [7] enables to provide 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.

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 with the GeneSIS model representing the running system. Finally, the adaptation engine enacts the adaptation (*i.e.*, the deployment) by modifying only the parts of the system necessary to account for the difference and the target GeneSIS model becomes the current GeneSIS model. The deployment engine can delegate part of its activities to deployment agents running on the field (see Section IV-B for more details).

In the following subsections, we detail the specific deployment support that is offered for two `InternalComponents` natively supported by GeneSIS: the Node-RED and ThingML components, namely. These are the deployable artefact presented in Section II, for which classical deployment approaches do not offer specific support. It is worth noting that *(i)* these nodes are represented as regular `InternalComponent` and *(ii)* GeneSIS is not bound to any of them (*i.e.*, GeneSIS can be used without these nodes).

1) Node-RED Components – dynamic adaptation of the application behavior: Node-RED[4], an open source project by IBM, uses a dataflow programming model for building IoT applications and services. Provided with a visual tool, Node-RED facilitates the tasks of orchestrating IoT devices, wiring them up to form an IoT application. More precisely, a Node-RED application takes the form of one or more *flows*, which are composed of a set of *nodes* and *wires*. A *node* is a piece of software written in JavaScript that typically executes when a message is received from a *wire*. Node-RED can run at the edge of the network because of its light footprint. Thanks to the large community behind Node-RED, a large set of Node-RED nodes are available off-the-shelf making it easy to implement new applications.

The Node-RED Admin API can be used to remotely administer an instance of Node-RED[5]. In particular, it enables the dynamic loading of a flow or the dynamic modification of the running flow. Node-RED also implements the Models@Run-time pattern, it is thus possible to add or remove nodes without modifying the rest of the flow.

When deploying a Node-RED `InternalComponent`, GeneSIS leverages the Node-RED Admin API in order to dynamically instantiate the necessary nodes within a flow to ensure the communications with the rest of the components in the GeneSIS model.

2) ThingML Components – deployment across heterogeneous platforms: 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 at the IoT and edge end. 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 [5]. 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 use 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

[4]https://nodered.org

[5]https://nodered.org/docs/api/admin/methods/

necessarily relying on a virtualization technology for lower footprint.

B. The GeneSIS Deployment Agent

It is not always possible for the GENESIS execution engine to directly deploy software on all hosts. Indeed, tiny devices, for instance, do not always have direct access to Internet or even the necessary facilities for remote access (in such case, the access to 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. It is implemented as a Node-RED application. We decomposed the deployment procedure into four steps resulting in four groups of Node-RED nodes.

a) 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 sketche 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 should 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.

b) Deployment configuration nodes: This type of node aims at preparing the actual deployment of a software component (being generated by 1. or not). This typically consists in generating configuration files. For instance, we created a ThingML 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.

c) Deployment nodes: This type of node aims at enacting the deployment of a software component on a specific target. In the context of our motivating example, we created an Arduino deployment node that *(i)* build and upload an Arduino sketch on the Arduino board using the Arduino CLI[6] and *(ii)* install the libraries required for its proper execution. Similarly, we created a Docker deployment node. These nodes typically consume messages from the configuration nodes and do not produce any output. Removing an instance of a deployment node results in the termination of the deployed software (*e.g.*, killing a docker container, deploying a dummy Arduino sketch).

d) Communication nodes: After deployment, it can be important to communicate with the deployed software artefacts, for instance to monitor the status of a deployment. Communications nodes are regular Node-RED I/O nodes such as serial port for Arduino board or HTTP requests for REST services.

The flow of components (a.k.a. nodes) that will form the deployment agent is dynamically generated and deployed by the GENESIS deployment engine based on the target host. It is worth noting that only the deployment node is mandatory and needs to be deployed on the device locally connected to the target host. Indeed, compilation and communications are activities that could be run anywhere on the IoT, edge, and cloud space. In addition, the other components can be distributed across different instances of Node-RED.

Thanks to this modularity, components from each of these groups can be seamlessly and dynamically composed for different types of deployments.

V. RELATED WORK

For some years now, multiple tools have been available on the market to support the deployment and configuration of software systems, *e.g.*, Puppet[7], Chef[8], CFEngine[9]. 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 [8] (*i.e.*, systems deployed across multiple cloud solutions from different providers) such as CloudMF [9], OpenTOSCA [10], Cloudify[10], and Brooklyn[11]. 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[12], Kubernetes[13]. 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

[6]https://playground.arduino.cc/Learning/CommandLine
[7]https://puppet.com/
[8]https://www.chef.io/chef/
[9]https://cfengine.com/
[10]http://cloudify.co/
[11]https://brooklyn.apache.org
[12]https://docs.docker.com/compose/
[13]https://kubernetes.io

containers' footprint. Thanks to these characteristics, container technologies are not only relevant for cloud infrastructure but can also be used on edge devices.

On the other side, 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 for the software component to be deployed 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.*, ARMv6 for a Raspberry Pi) and a Docker image is created before being deployed on the target hosting device(s). However, Resin.io offers limited support for the deployment and management of software components on tiny devices that cannot host containers.

In [3], we have conducted an 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). 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.

VI. Future work

The development and operation of applications running on IoT devices such as Arduino boards is typically challenging as it is not always possible to access to the logs or to the systems output. As future work we plan to extend ThingML and the deployment of ThingML programs via GENESIS with the necessary mechanisms to enable the remote debugging of ThingML programs as well as the runtime monitoring of their execution flow. One of the requirements should be to minimize the impact of such monitoring facilities on the performances of the host device, possibly delegating part of the work to more powerful resources.

Finally, GENESIS is currently evolving as part of the ENACT H2020 project [11]. During the project, our framework is being evaluated in the context of three use cases: smart building, eHealth, and intelligent transport system. In this context, we will need to extend our framework, enabling the assimilation of the proper mechanisms for trustworthy execution of the SIS (*e.g.*, security mechanisms, quality assurance, robustness). The language will be extended with the necessary trustworthiness concepts whilst the execution engine will enact the deployment of the trustworthiness mechanisms when necessary. As detailed in Section V, approaches for the deployment and orchestration of IoT systems typically offer very little, or even no, support for ensuring or improving the trustworthiness of the system deployed.

VII. Conclusion

In this paper, we have presented how GENESIS leverages upon model-driven techniques and methods to support the continuous deployment and orchestration of SIS. The GENESIS Modelling language support the specification of the deployment of SIS over IoT, edge, and cloud infrastructure in a platform-independent or -specific way. The associated execution engine provides the mechanisms to enact this deployment and support its dynamic adaptation. Particular focus has been on supporting deployment on IoT infrastructure, which can include devices with no or limited access to Internet.

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

References

[1] IoT 2020 project team in the IEC Market Strategy Board, "IoT 2020: Smart and secure IoT platform," IEC White paper, 2016.

[2] A. M. (Ed.), "Cyber physical systems: Opportunities and challenges for software, services, cloud and data," NESSI White paper, 2015.

[3] P. H. Nguyen, N. Ferry, G. Erdogan, H. Song, S. Lavirotte, J.-Y. Tigli, and A. Solberg, "Advances in deployment and orchestration approaches for iot - a systematic review," in *IEEE International Congress On Internet of Things (ICIOT)*, ser. ICIOT'19. IEEE, 2019.

[4] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A systematic review of cloud modeling languages," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 22:1–22:38, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3150227

[5] B. Morin, F. Fleurey, K.-E. Husa, and O. Barais, "A generative middleware for heterogeneous and distributed services," in *19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. IEEE, 2016, pp. 107–116.

[6] A. Dearie, "Software deployment, past, present and future," in *Future of Software Engineering, 2007. FOSE'07*. IEEE, 2007, pp. 269–284.

[7] G. Blair, N. Bencomo, and R. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[8] D. Petcu, "Multi-cloud: expectations and current approaches," in *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*. ACM, 2013, pp. 1–6.

[9] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "Cloudmf: Model-driven management of multi-cloud applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 18, no. 2, p. 16, 2018.

[10] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, and F. 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*. ACM, 2016, pp. 181–182.

[11] N. Ferry, A. Solberg, H. Song, S. Lavirotte, J.-Y. Tigli, T. Winter, V. Muntés-Mulero, A. Metzger, E. R. Velasco, and A. C. Aguirre, "Enact: Development, operation, and quality assurance of trustworthy smart iot systems," in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, 2018, pp. 112–127.