A pattern–driven adaptation in IoT orchestrations to guarantee SPDI properties

Papoutsakis Manos[1,2][0000−0002−5145−5537], Fysarakis Konstantinos[3][0000−0002−6871−8102], Michalodimitrakis Emmanouil[1][0000−0002−1704−454X], Lakka Eftychia[1][0000−0002−0512−6150], Petroulakis Nikolaos[1][0000−0002−3489−7763], Spanoudakis George[2,4][0000−0002−0037−2600], and Ioannidis Sotiris[4][0000−0001−9340−2241]

[1] Institute of Computer Science, Foundation for Research and Technology – Hellas, Heraklion, Greece
{paputsak,manmix,elakka,npetro,sotiris}@ics.forth.gr
[2] Department of Computer Science, City University of London, London, UK
{Emmanouil.Papoutsakis, g.e.spanoudakis}@city.ac.uk
[3] Sphynx Technology Solutions AG, Zug, Switzerland
{fysarakis, spanoudakis}@sphynx.ch

Abstract. The orchestration of heterogeneous IoT devices to enable the provision of IoT applications and services poses numerous challenges, especially in contexts where end-to-end security and privacy guarantees are needed. To tackle these challenges, this paper presents a pattern–driven approach for interacting with IoT systems, whereby the required properties are guaranteed. Patterns are leveraged to represent the relationship between security, privacy, dependability and interoperability (SPDI) properties of specific smart objects and corresponding properties of orchestrations that include said objects. In this way, patterns allow the verification that certain SPDI properties hold for an IoT orchestration, while also enabling the adaptation of IoT orchestrations in ways that allow the given properties to hold.

Keywords: Security Patterns · IoT orchestrations · SPDI properties · Adaptation · Node–RED · Drools Rule Engine.

1 Introduction

The term "Internet-of-Things" (IoT) has been introduced by the expansion of the Internet from the usual devices such as desktop pc, laptops etc, to different machines and smart objects, in an effort to describe the new environment which represents new ways of working, interacting entertainment and living by enabling the creation of new applications [1]. Admittedly, there are numerous challenges of IoT orchestrations due to their heterogeneity of the IoT devices including end-to-end security and privacy and that is evident from various studies in the field [2–4].

Motivated by the above, this paper presents a pattern driven approach for composing IoT systems where security properties are guaranteed. Security patterns are intended to capture security expertise in the form of worked solutions to recurring problems. Consequently the security patterns provide the developers a comprehensive tool that enables them to utilise security concepts without having to be a security professional [5].

The role of patterns into our approach is to represent the relationship between SPDI properties of specific smart objects and corresponding properties of orchestrations that include said objects. The adoption of patterns allows for (i) the verification that a smart object orchestration satisfies certain SPDI properties, and (ii) the generation (and adaptation) of orchestrations in ways that are guaranteed to satisfy required SPDI properties. The proposed approach is based on the development and the architecture that is carried out in SEMIoTICS[4], a horizon 2020 project.

The remainder of this paper is organized as follows. In Section 2 an overview of related work is presented. Section 3 presents a background to the underlying technologies and concepts used. Section 4 outlines the architecture of our approach. In Section 5 we present an implementation of our approach where the mechanisms of an IoT orchestration adaptation are detailed. Finally, Section 6 provides concluding remarks.

2 Related Work

It is common knowledge that the IoT is consisted by physical or virtual object/devices (Things), equipped with sensing, accumulating and transferring data over the internet automatically. In parallel, billions of sensors and actuators have been already deployed and should be combined into a number of domain-specific platforms. Taking to account said issues, a widespread interest of both industry and academy was increased to overcome several challenges such as the dynamicity, scalability, heterogeneity and end-to-end security and privacy requirements of such environments. This implies that the system should have the ability to adapt (semi)-itself in order to continue offering the above requirements. Dynamic proactive adaptation in particular is demanded to provide adjustments at runtime [6].

The huge number of dimensions over which such a big arena spans, makes the investigation of the adaptation domain in IoT very hard and subject to entropy [7]. Thus, the first attempts that are presented in this section, are focused on the description of IoT service compositions ([8], [9], [10]) taking to account the energy consumption of the involved IoT devices; the latter pays attention to QoS properties reducing the services search space and the composition time. Although these works are interesting, none of them takes into consideration possible security properties of the individual IoT services or the whole composition.

Additionally, there is the work of [11], in which a contExt Aware web Service dEscription Language (wEASEL) is introduced. wEASEL is an abstract service

[4] https://www.semiotics-project.eu/

model to represent services and user tasks in Ambient Assisted Living (AAL) environments. Attention is paid to data-flow and context-flow constraints for the service composition but the authors do not mention any security properties. A work that includes security aspects is that of [12]. BPMN 2.0 is used for the description of the service choreographies that the built platform can synthesize and execute. Regarding the security aspect, the enforcement of security properties is exclusively done by the existing communication protocols since the security filter component is able to filter these protocols and keep only those that conform to the specified security requirements.

Finally, [13] provide a mechanism that manages IoT choreographies at runtime dynamically. According to their approach IoT service compositions are described by templates called Recipes, which consist of Ingredients and their Interactions. Furthermore, there are more requirements described by Offering Selection Rules (OSRs) that make the reconfiguration of the system possible during runtime. The authors' service composition approach is semi-automated to avoid the complexity of the semantic models and the inefficiency of the reasoner due to the large number of available devices and services. We consider this approach closer to the way we envision a pattern language. Their way of IoT representation with the notions of Ingredients and Interactions, and the fact that the OSRs allow for requirement description inspired the creation of the language described in the next section.

3 Background

The core of our approach is pattern based reasoning that can deliver verification of SPDI properties or adaptation of IoT orchestrations in a way that guarantees such properties. In the next subsections innovative building blocks of the pattern based reasoning concept are presented.

3.1 Pattern Language

The proposed approach infers the development of a framework for managing IoT applications based on patterns. A language for specifying the components that constitute such applications along with their interfaces and interactions has been created. Moreover, the said language is able to encode various functional and non-functional properties of such components and their orchestrations. The corresponding model was derived using the Eclipse Modeling Framework (EMF), visualizing the Ecore part of the EMF metamodel, which contains the information about the defined classes. The defined model in conjunction with patterns allows for the verification of SPDI patterns in IoT applications and possibly enables different adaptation actions.

The main classes of the model, which are used in the pattern specification, include Placeholders, Orchestrations, OrchestrationActivities and Properties. An orchestration of activities may be of different types depending on the order in which the different activities involved in it must be executed. As a result, an

orchestration can be defined as Sequence, Merge, Choice or Split. An orchestration involves OrchestrationActivities. The implementation of an activity in an IoT application orchestration may be provided by: i) a software component, ii) a software service, iii) a network component, iv) an IoT sensor, v) an IoT actuator or vi) an IoT gateway. Orchestrations and OrchestrationActivities are grouped under the general concept (class) of a Placeholder. Placeholders may be characterized by SPDI properties. A Property can be verified at runtime, through the verification process, which can be done through monitoring, testing, a certificate or via a pattern. The first two cases require the existence of a monitoring service or a testing tool allows the verification of the SPDI property of a placeholder activity. The third case refers to a service allowing the verification of validity of certificates verifying that a placeholder satisfies a certain property. Thus, while in the case of a pattern, the verification points to a specific pattern rule, in all the other cases the verification must point to the interface of a monitoring tool, testing service or certificate repository. Moving on, a category is assigned to each Property. A category can refer to confidentiality, integrity, availability (covering the Security property), privacy, dependability, interoperability or even QoS. In this way a classification of the properties is achieved.

Finally, the set of all the SPDI properties that are inferred for the different placeholders of an orchestration by a pattern are aggregated into a PropertyPlan object. A detailed representation of the pattern language and model is presented in our previous work [14].

3.2 Pattern Definition

SPDI patterns encode proven dependencies between SPDI properties of individual placeholders implementing activities in IoT applications orchestrations (i.e. activity-level SPDI properties) and SPDI properties of these orchestrations (i.e. workflow-level SPDI properties). The specification of an SPDI pattern consists of the following parts:

1. **Activity Properties (AP)** part, which defines the SPDI properties required of the activity placeholders present in the workflow of the pattern, to allow for the guarantee of the OP properties detailed in the corresponding part of the pattern.
2. **Orchestration (ORCH)** part, which defines the abstract form of the orchestration that the pattern applies to.
3. **Conditions** part, which defines the functional requirements, the states or the constraints that a system should define, or what a system must do, and how it reacts on specific inputs or situations.
4. **Orchestration Properties (OP)** part, which defines the SPDI properties that the pattern guarantees for the orchestration that is specified in its ORCH part.

The semantic interpretation of an SPDI pattern, as described above, is that if the AP properties that have been specified for the activity placeholders, part of the

orchestration of the pattern, and the Conditions of the pattern hold (verified as True), then the specified OP property also holds for the whole ORCH. Formally, this can be expressed as:

$$APs \wedge ORCH \wedge Conditions \models OP \tag{1}$$

where APs are materialized using the Property class described above. Each Property is uniquely identified by its name, while the attribute PropertySubject the orchestration component for which the property verifiable. ORCH is an Orchestration object including Placeholders. Finally, OP is an orchestration-wide Property object.

3.3 Pattern Rule

Towards the automated processing of the SPDI patterns, the latter are expressed as Drools [15] business production rules, and the associated rule engine. A Drools production rule has the following generic structure:

```
rulename
    <attributes>*
when
    <conditional element>*
then
    <action>*
end
```

In the *when* part of the rule the ORCH part of the pattern, the conditions regarding the inputs and outputs of orchestration placeholders, and the OP property guaranteed by the pattern for the specific ORCH are described. Additionally, in the *then* part, the AP properties are described, which will guarantee the OP property, if they are satisfied by the orchestration placeholders. The above can be expressed formally as:

$$ORCH \wedge Conditions \wedge OP \Rightarrow APi \quad (i = 1, ..., n) \tag{2}$$

where APi are the AP properties required by the orchestration placeholders of the SPDI pattern. As we can see, this is the opposite of the dependency relation proven in the equation (1) defined above. Thus, this encoding allows the inference of the APi properties which, if satisfied, guarantee the satisfaction of the ORCH-level SPDI property of it, as encoded in the pattern. In this way, Drools Rules can be used to allow the automated design or adaptation of the ORCH in order to preserve the ORCH-level SPDI property defined in the pattern.

4 Architecture and core building blocks

In this section we describe the components that constitute our architecture which are depicted in Fig. 1 The main build blocks are (i) Node–RED, (ii) Pattern Orchestrator and (iii) Pattern Engine instances. The interaction between these blocks is achieved through RESTful APIs.

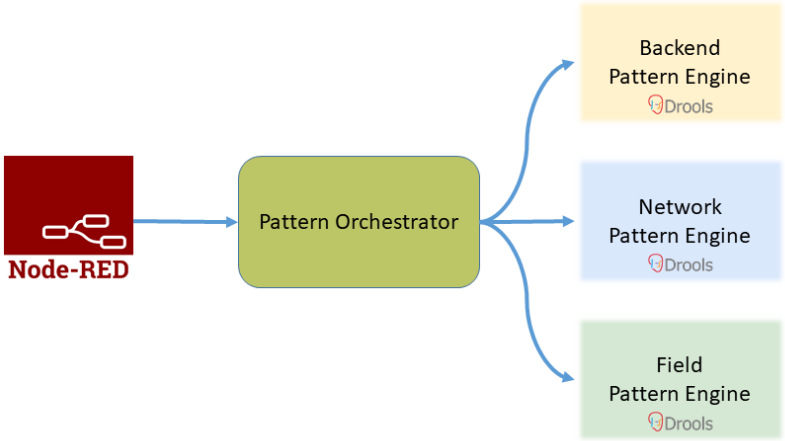

Fig. 1. Architecture

4.1 Node–RED

Node–RED is a tool that helps programmers to visually create their applications, inspired by the flow-based programming paradigm [16]. The underlying framework is NodeJS[5] which enables the programmer to develop an IoT system with concepts such as nodes and flows. A node reflects a service offered by the system. When nodes are used in cooperation, constitute a flow that is able to perform a task and provide functionalities to the system while creating most of the code automatically.

Due to the fact that Node–RED is open source with an active community, new nodes with up-to-date technologies are constantly available, thus making it feasible to complete a variety of tasks like storing values to database, encrypting traffic between nodes, establish communication using the MQTT protocol and more.

4.2 Pattern Orchestrator

The Pattern Orchestrator module features an underlying semantic reasoner able to understand the internal components of IoT Service orchestrations expressed using the pattern language, received from the Node–RED module and transform them into architectural patterns. The Pattern Orchestrator is then responsible to pass said patterns to the corresponding Pattern Engines (as defined in the Back-end, Network, and Field layers), after translating them to a machine-processable format (in Drools), selecting for each of them the subset of patterns that refer to components under their control (e.g. passing Network-specific patterns to the

[5] https://nodejs.org/en/

Pattern Engine present in the SDN controller). Through the above functions, the module achieves automated configuration, coordination, and management of the patterns across different layers and service orchestrations.

4.3 Pattern Engine

The Pattern Engine (PE) module enables the capability to insert, modify, execute and retract patterns at design or at run-time. PE is based on a rule engine which is able to express design patterns as production rules. Enabling reasoning, driven by production rules, appeared to be an efficient way to represent patterns. For that reason, a rule engine is required to support backward and forward chaining inference and verification. Drools rule engine [17] appeared to be a suitable solution to support design patterns by applying and extending the Rete algorithm [18] and later the Phreak algorithm [19]. Finally, the PE integrates different sub-components required by the rule engine such as the knowledge base, the core engine and the compiler.

The Drools Engine in the PE is not running continuously. On the contrary, it is started only when needed, i.e. when a Fact or a Rule is added/updated/removed. To prevent the loss of the facts when the Drools Engine stops, they are also saved in the memory of the PE. Additionally they are stored locally in the file system where the PE is installed for debugging purposes. The PE also has the Rules stored locally and loads them every time the engine is required to run. These Rules may be pre-installed, but they can also be added during run-time through the corresponding endpoint "insertRule". Facts are inserted from the Pattern Orchestrator by using the "addFact" endpoint as well as from internal monitoring mechanism.

5 Implementation details

We have described in previous work [21] the way an SPDI property is verified in orchestration-level, using our pattern-driven approach. In this section, we focus on describing the mechanisms that will be engaged in the form of adaptation actions, when a given SPDI property referring to an orchestration, does not hold. The result of the said adaptation actions is an updated orchestration, that is communicated back to Node–RED in order to be deployed again. This updated version of orchestration may differ from the original in terms of adding new components and/or replacing others. The said adaptation actions are driven by a specific pattern, chosen from the literature and in our case it is the Encrypted Storage Pattern [20]. The next two sub-sections describe in detail the chosen pattern and the information flow among the components of our architecture.

Encrypted Storage Pattern: This pattern constitutes a second layer line of defense against the theft of data on systems. Even if the data is stolen, the most sensitive data will remain safe, since it cannot be decrypted.

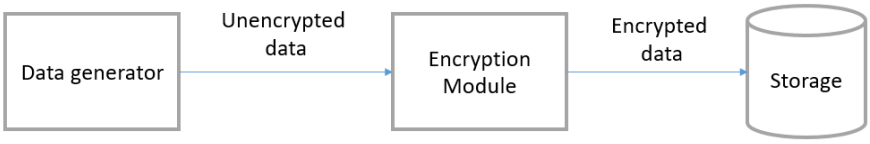

Fig. 2. Encrypted Storage Pattern

Well-known examples of the Encrypted Storage Pattern usage include: a) the UNIX password file that hashes each user's password and stores only the hashed form and b) web sites that use encryption to protect the most sensitive data that are stored on the website server.

An implementation of the pattern is depicted in Fig. 2. As it can be seen, the specific implementation imposes the existence of an encryption module between the data generator and the storage unit. Our implementation is based on this pattern, therefore the absence of the said module will trigger adaptation actions that will result in the addition of the encryption module where it is needed.

Information flow: Node–RED is used to describe the functionality of the IoT system which will later on will be distributed to the Pattern Orchestrator in a translated version of the flow. This translation will encode the flow from Node–RED to pattern language thus enabling the Pattern Orchestrator to create a Drools fact, i.e. an instance of the corresponding Java class of the IoT application model, for every orchestration activity, control flow operation or property. For a more detailed view regarding the creation of a Drools Fact inside Pattern Orchestrator please refer to our previous work [21, 22].

As soon as the Pattern Orchestrator receives the translated flow, has to decide based on each of the orchestration components involved, which Pattern Engine will be the recipient of the said components. Not all components may arrive at the same Pattern Engine as it may refer to different layers e.g. components that represent services will be forward to the Pattern Engine residing at the Backend Layer, whereas components that represent field devices will be forwarded to the Pattern Engine residing at the Field Layer.

Upon arrival at the corresponding Pattern Engine, each orchestration component is inserted into knowledge session of Drools Engine as Drools facts. These Drools facts are used by Drools rules, which are fired when their conditions are met. A Pattern Engine equipped with the said rules, is capable to verify whether an SPDI property holds for a given orchestration. Additionally when such a property does not hold, appropriate rules can be used to trigger adaptation actions which will result to the satisfaction of the property.

The adaptation actions that are presented in the current implementation include the addition of a new component in the original orchestration, which corresponds to a new node in the Node–RED editor. The said addition means that new connections between the latter and the pre-existing nodes will be created.

Node–RED exposes an API that allows such flow updates and in this particular case a PUT method request is used at the following URL: http://"nodeRedIP": "nodeRedPort"/flow/"ID". The updated flow is passed to the request body in JSON format which represents a single flow configuration object in Node–RED. Upon successful consumption of the said request, the status code returned from Node–RED is 204 combined with the id of the new flow.

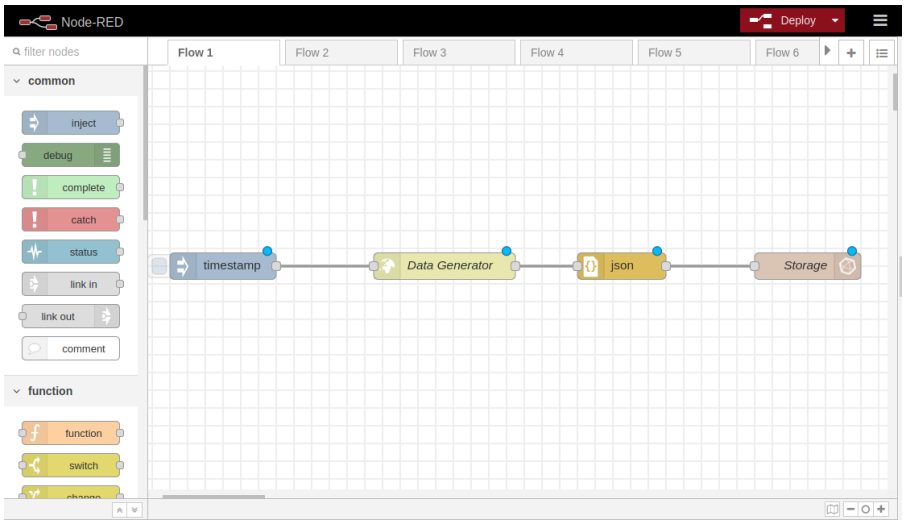

Fig. 3. Service orchestration depicted as Node–RED flow

The orchestration that has been used in the current implementation is depicted in Fig. 3, in the Node–RED editor. Its main components are a data generator and a storage unit where the data are driven. The former is depicted as a node named Data Generator, which makes an HTTP request whenever triggered by the timestamp node. The output of the Data Generator is essentially the response of the HTTP request, which is in JSON format and parsed by a simple JSON node.

The storage unit is represented by a node named Storage, which sends data to an InfluxDB, a time-series database, to be stored. The described orchestration along with the SPDI property (Encrypted Storage Property) to be verified, is translated in the pattern language and sent to Pattern Orchestrator. Having such a property hold for a certain orchestration component, it states that the said component stores data in an encrypted format. All the orchestration components and the desired SDPI property are sent to Pattern Engine and added to Drools working memory as Drools Facts.

Since no encryption takes place between the two main components of our orchestration, the corresponding adaptation Drools rule that will be triggered

will respond that the Encrypted Storage Property is not satisfied and adaptation actions will be initiated.

The Pattern Engine will send a request to an appropriate API of the Pattern Orchestrator with the SPDI Property that is not satisfied. As soon as, the Pattern Orchestrator receives the said request, adapts the JSON representation of the original orchestration, based on the presented implementation of the Encrypted Storage pattern, and sends it to Node–RED. These changes include the addition of an EncryptionModule node along with the appropriate connections, in front of Storage node, which is able to encrypt the transferred message using any of the most well-known encryption algorithms and a provided secret key.

The code snippet below depicts what is added by Pattern Orchestrator to the JSON representation of the flow in particular the EncryptionModule node. As we can see, its id, name and type are defined and the z attribute represents the flow to which this new node is added. Moreover, we see that the encryption algorithm and the secret key are defined. Finally, the wires attribute includes the ids of the nodes that use the output of the node in question as their input. In our case, it includes the id of the Storage node.

```
{
    "id": "6757fce7.108a94",
    "type": "encrypt",
    "z": "99783291.272d2",
    "name": "EncryptionModule",
    "algorithm": "AES",
    "key": "semiotics",
    "x": 720,
    "y": 260,
    "wires": [
        [
            "eae7b2ac.b98fc"
        ]
    ]
}
```

The new, updated orchestration, after the adaptation actions, is depicted in the form of a flow inside the editor of Node–RED, in Fig. 4.

6 Conclusion

This work presented a pattern–driven approach able to rectify an IoT orchestration in a way that an SDPI property to hold. A proof of concept is presented including an exemplary IoT orchestration that is consisted of mainly two components, a data generator and a data storage. The SPDI property that needs to be enforced to this orchestration requires the addition of an encryption node between them, according to the Encrypted Storage pattern.

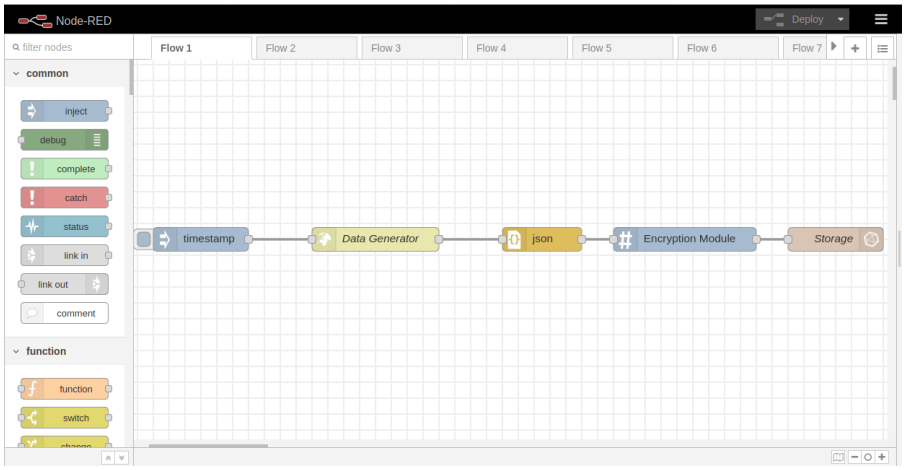

Fig. 4. Updated service orchestration depicted as Node–RED flow

Currently, only a small number of patterns has been used for verification of SPDI properties and adaptation of IoT orchestrations. Our plans for the future are to incorporate even more patterns that will allow a wider range of adaptation actions that will enable the application of a plethora of SPDI properties.

7 Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 780315 (SEMIoTICS).

References

1. Miorandi, D., Sicari, S., De Pellegrini, F., & Chlamtac, I.: Internet of things: Vision, applications and research challenges. Ad hoc networks **10**(7), 1497–1516 (2012)
2. Čolaković, A., Hadžialić, M. : Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues. Computer Networks, 144, 17-39 (2018)
3. Singh, S., Singh, N.: Internet of Things (IoT): Security challenges, business opportunities & reference architecture for E-commerce. In: 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), Noida, pp. 1577–1581. IEEE, (2015). https://doi.org/ 10.1109/ICGCIoT.2015.7380718
4. Xu, T., Wendt, J. B., Potkonjak, M.: Security of IoT systems: Design challenges and opportunities. In: 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, pp. 417–423, IEEE, (2014). https://doi.org/10.1109/ICCAD.2014.7001385
5. Kienzle, D.M. and Elder, M.C.: Final Technical Report: Security Patterns for web Application Development. DARPA, Washington DC (2005)

6. Achtaich, A., Souissi, N., Mazo, R., Salinesi, C., & Roudies, O.: Designing a Framework for Smart IoT Adaptations. In International Conference on Emerging Technologies for Developing Countries, pp. 57-66. Springer, Cham (2017).
7. Arcelli, D.: Exploiting Queuing Networks to Model and Assess the Performance of Self-Adaptive Software Systems: A Survey. Procedia Computer Science, 170, pp 498-505 (2020).
8. Baker, T., Asim, M., Tawfik, H., Aldawsari, B., & Buyya, R.: An energy-aware service composition algorithm for multiple cloud-based IoT applications. Journal of Network and Computer Applications, 89, pp 96-108 (2017).
9. Zhou, Z., Zhao, D., Liu, L., & Hung, P. C.: Energy-aware composition for wireless sensor networks as a service. Future Generation Computer Systems, 80, pp 299-310 (2018).
10. Alsaryrah, O., Mashal, I., & Chung, T. Y.: Energy-aware services composition for Internet of Things. In 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), pp. 604-608. IEEE (2018).
11. Urbieta, A., González-Beltrán, A., Mokhtar, S. B., Hossain, M. A., & Capra, L.: Adaptive and context-aware service composition for IoT-based smart cities. Future Generation Computer Systems, 76, pp 262-274 (2017).
12. Chen, L., & Englund, C.: Choreographing services for smart cities: Smart traffic demonstration. In 2017 IEEE 85th Vehicular Technology Conference (VTC Spring), pp. 1-5. IEEE (2017).
13. Seeger, J., Deshmukh, R. A., & Broring, A.: Running distributed and dynamic IOT choreographies. In 2018 Global Internet of Things Summit (GIoTS), pp. 1-6. IEEE (2018).
14. Fysarakis, K., Papoutsakis, M., Petroulakis, N., & Spanoudakis, G.: Towards IoT Orchestrations with Security, Privacy, Dependability and Interoperability Guarantees. In: Conference: 2019 IEEE Global Communications Conference (GLOBECOM), At Waikoloa, HI, USA (2019). https://doi.org/10.1109/GLOBECOM38437.2019.9013275
15. Business Rules Management System (BRMS), https://www.drools.org. Last accessed 5 Aug 2020
16. Morrison, J. Paul.: Flow-Based Programming: A new approach to application development. CreateSpace, (2010)
17. Drools - Business Rules Management System (Java™, Open Source), https://www.drools.org. Last accessed 5 Aug 2020
18. Forgy, C. L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In Readings in Artificial Intelligence and Databases pp. 547–559. Morgan Kaufmann (1989)
19. Chapter 5. Hybrid Reasoning https://docs.jboss.org/drools/release/6.4.0.Final/drools-docs/html/ch05.htmlLast accessed 5 Aug 2020
20. Kienzle, D. M., Elder, M. C., Tyree, D., & Edwards-Hewitt, J.: Security patterns repository version 1.0. DARPA, Washington DC (2002)
21. Bröring, A., Seeger, J., Papoutsakis, M., Fysarakis, K. & Caracalli. Networking-Aware IoT Application Development. Sensors, vol. 20, pp. 897, (2020). https://doi.org/10.3390/s20030897
22. Soultatos, O., Papoutsakis, M., Fysarakis, K., Hatzivasilis, G., Michalodimitrakis, M., Spanoudakis, G., Ioannidis, S. Pattern-Driven Security, Privacy, Dependability and Interoperability Management of IoT Environments. In 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Limassol, Cyprus, pp. 1-6, (2019). https://doi.org/10.1109/CAMAD.2019.8858429.