Towards IoT Orchestrations with Security, Privacy, Dependability and Interoperability Guarantees

Konstantinos Fysarakis*, Manos Papoutsakis†, Nikolaos Petroulakis†, and George Spanoudakis*
*Sphynx Technology Solutions AG, Zug, Switzerland
Email: {fysarakis, spanoudakis}@sphynx.ch
†Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece
Email: {paputsak, npetro}@ics.forth.gr

***Abstract*—The advent of the Internet of Things opens a plethora of possibilities, provided the research and industry communities are able to overcome a number of challenges such as the dynamicity, scalability, heterogeneity and end-to-end security and privacy requirements of such environments. Motivated by these challenges, this paper proposes leveraging architectural patterns to provide, in an integrated manner, security, dependability, privacy, and interoperability guarantees, across horizontal and vertical compositional structures of IoT applications. The pattern language design process and definition is presented, along with an implementation enabling the automated, pattern-driven property verification and adaptation of IoT orchestrations.**

***Index Terms*—internet of things, pattern-based engineering, security, privacy, dependability, interoperability**

I. Introduction

As we approach towards the realisation of the Internet of Things (IoT) vision, the enormous potential for new generations of IoT applications is becoming more evident, enabled by leveraging synergies arising through the convergence of consumer, business and industrial Internet, and the creation of an open, global network connecting people, data, and things [1] [2] [3]. Nevertheless, early adopters are faced with numerous challenges [4] [5] [6] stemming from the intricacies of the IoT environment, such as their dynamicity, scalability, and heterogeneity, as well as the end-to-end security, privacy and Quality of Service (QoS) requirements of each of these applications domains.

Motivated by the above, project SEMIoTICS (https://www.semiotics-project.eu/) aims to enable and guarantee secure and dependable actuation and semi-autonomic behaviour in IoT/IIoT applications, through a pattern-driven approach. Patterns are re-usable solutions to common problems and building blocks to architectures. In SEMIoTICS, patterns are used to encode proven dependencies between security, privacy, dependability and interoperability (SPDI) properties of individual smart objects and corresponding properties of orchestrations (composition) involving them. The encoding of such dependencies will enable: (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.

In this context, the work presented herein focuses on presenting a core enabling element of the above approach: the definition of a language for specifying machine-interpretable SPDI patterns and the development, using this language, patterns encoding horizontal and vertical ways of composing parts of or end-to-end IoT applications that can evidently guarantee SPDI properties. The pattern language itself is based on a system model defined and presented within this paper. Said system model is encompassing smart objects in the field layer (IoT sensors, actuators and gateways), the network layers (e.g., SDN controllers) and at the backend (e.g., cloud services), and the associated SPDI properties, as well as their orchestrations. This model forms the basis of the language definition, while a grammar is also defined to specify the exact structure of the language. The translation from this language to a machine-processable format is also presented, along with a preliminary proof of concept, to validate the feasibility of the automated verification of properties and the triggering of relevant adaptations

II. Related Works

The pattern-driven approach of SEMIoTICS follows the *security-by-design* concept, which aims to guarantee system-wide security properties by virtue of the design of the involved systems and their subsystems. This is leveraged to provide orchestration-level SPDI guarantees, while encompassing all involved components and entities which are composed to create the orchestrations (e.g., physical devices and software). A key capability required in security-by-design is the ability to verify the desired security properties as part of the design process. A typical way to achieve this is using model-based techniques [7]–[9], whereby software component and service compositions are modelled using formal languages and the required security properties are expressed as properties on the model [10]. The satisfiability of the required properties is based on model checking [11], [12]. Other approaches focus on software service workflows using business process modelling languages (e.g., Sec-MoSC [13]). Pino et al. [15] use secure service orchestration (SSO) patterns to support the design of service workflows with required security properties, leveraging pattern-based analysis to verify security properties. This avoids full model checking that is computationally expensive and non-scalable to larger systems, such as the IoT. Moreover, some model-based approaches (e.g., [15]) support the transformation of security requirements to code for

automated checking of the required properties, both at design and at runtime.

The pattern-driven approach presented herein is inspired from similar pattern-based approaches used in service-oriented systems [14], [16], cyber-physical systems [17], and networks [18], [19], while covering the intricacies of IoT deployments and more properties in addition to Security, and also providing guarantees and verification capabilities that span both the service orchestration and deployment perspectives.

III. Pattern Language Definition

A. System Modeling

The overall objective of this work is to develop a framework that will be capable of managing the IoT applications based on patterns. Therefore, it is necessary to develop a language for specifying the components that constitute such applications along with their interfaces and interactions. To enable this, the definition of the various functional and non-functional properties of such components and their orchestrations is required. A model with such characteristics will effectively serve as a general "architecture and workflow model" of the IoT application. Once defined, this model can be used in conjunction with patterns to enable the reasoning required for verifying SPDI patterns in specific IoT applications and subsequently enable different types of adaptations. Working towards this goal, the system model appearing in Fig. 1 was defined; from an implementation perspective this was derived using the Eclipse Modeling Framework (EMF), visualising the Ecore part of the EMF metamodel, which contains the information about the defined classes.

The language for defining IoT application models adopts an orchestration-based approach. Orchestrations are modelled by the class *Orchestration* in Fig. 1. An orchestration of activities may be of different types depending on the order in which the different activities involved in it must be executed. According to this criterion, an orchestration may be defined as follows:

- *Sequence* refers to several activities executed in sequence under a single thread of control.
- *Parallel* refers to two or more activity instances executed in parallel within the workflow, giving rise to multiple threads of control.
- *Merge* is a point in the workflow where two or more parallel executing/alternative activities converge into a single common thread of control.
- *Choice* is a point within the workflow where a single thread of control makes a decision upon which branch to take when encountered with multiple alternative workflow branches, based on a choice condition.
- *Iterates* is a workflow cycle involving the repetitive execution of one (or more) activity(s) until a condition is met.

Moreover, an orchestration involves orchestration activities (class *OrchestrationActivity* in Fig. 1). At any instance of time, these activities may have a known implementation or not. In the former case, the activity will be a linked activity (class *LinkedActivity* in Fig. 1). Such an activity requires a ThingDescription (following W3C Thing Descriptions), which provides the details on how the activity is implemented, the characteristics of the underlying devices and relevant parameters (e.g., IP address, exposed endpoints, available resources), the corresponding SDPI properties, etc. In the latter, the activity will be an unassigned activity (see class *UnassignedActivity* in Fig. 1). Unassigned activities in an IoT application orchestration may exist during the design of the IoT application, when the exact implementation of a specific orchestration activity might not have been decided yet or at runtime when the particular component that used to provide the implementation of the activity can no longer be used (because, for example, it might be unavailable or because it no longer fulfils the properties required of it) and must be replaced. The implementation of an activity in an IoT application orchestration may be provided by:

i) A *software component*, i.e., a software module with an available and modifiable implementation that encapsulates a set of functions and data and makes them available through a programmatic interface.
ii) A *software service*, i.e., a software module that encapsulates a set of functions and data and makes them available through a programmatic interface, accessible remotely over a network, whose implementation is neither available to the owner nor modifiable.
iii) A *network component*, such as software defined network controllers, software switches/vSwitches, and potentially legacy networking components.
iv) An *IoT sensor*, i.e., a device that collects data from the environment or object under measurement and turns it into useful data.
v) An *IoT actuator*, i.e., a device that takes electrical input and transforms the input into tangible action.
vi) An *IoT gateway*, i.e., a physical device or software program that serves as the connection point between the field devices and the backend, via the software-defined network layer.
vii) A *(sub)orchestration* of IoT application activity implementers of types (i) to (vi).

The above types of IoT application activity implementers are grouped under the general concept of a placeholder (see the class *Placeholder* in Fig. 1). A placeholder is accessible through a set of interfaces. An interface is a named set of operations through which the functions and the data of the placeholder can be accessed from any element outside it; this is represented by the class *Interface* in Fig. 1. The interfaces through which a placeholder can be accessed are linked to the placeholder as the interfaces that it provides (see *provides* association end between the class *Placeholder* and *Interface* in Fig. 1). In addition, placeholders may require additional interfaces provided by other placeholders for them to function properly. A placeholder `P1` that provides access to a set of data may, for example, authenticate data access requests by relying to another placeholder `P2` with responsibility for

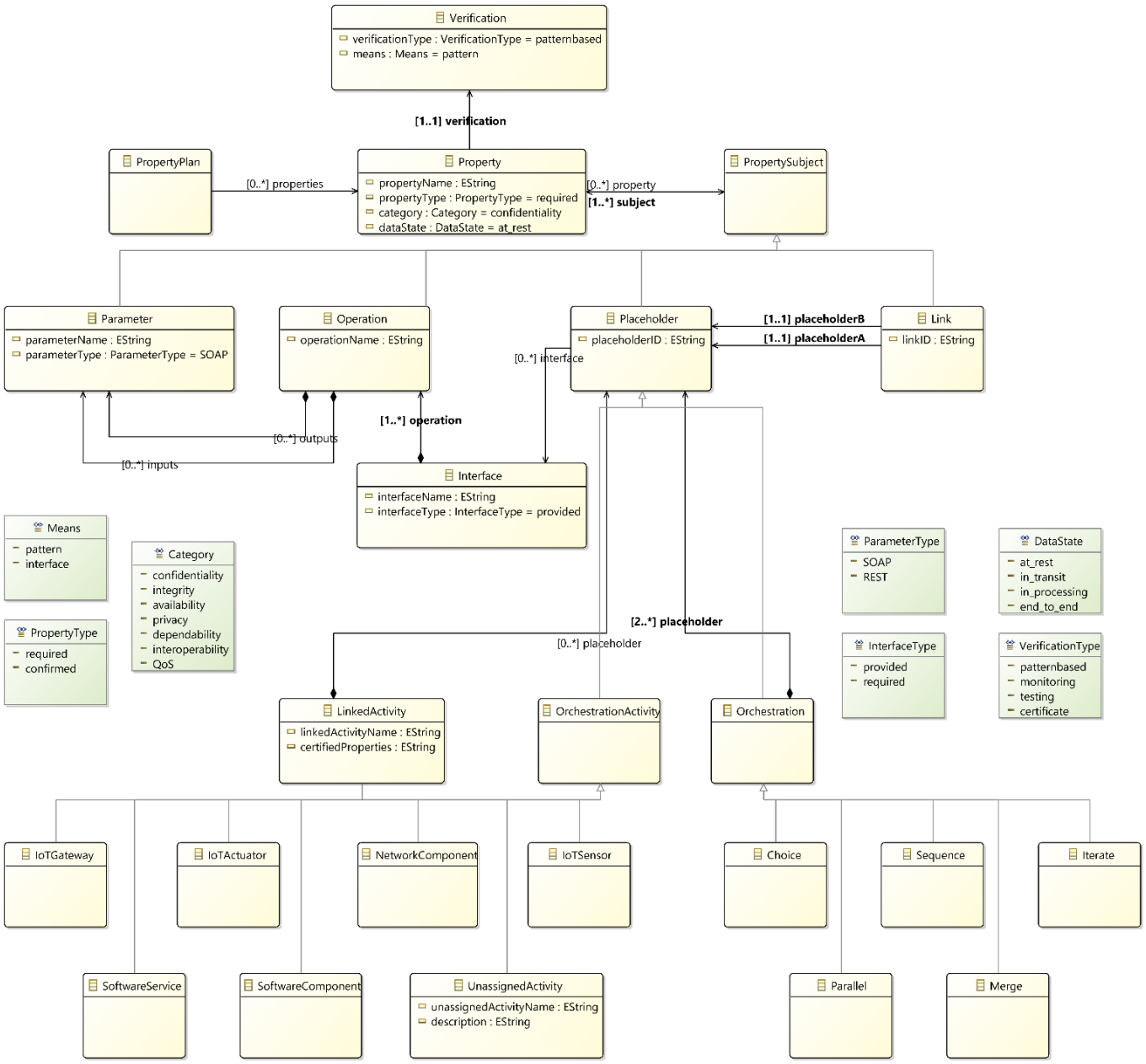

Fig. 1. IoT Orchestrations System Model

authentication and authorisation checks over users. In this case, `P2` would be modelled as a placeholder that provides two interfaces, i.e., an authentication and an authorisation interface, and `P1` as a placeholder that requires these two interfaces. Requires relations between placeholders and interfaces are modelled through the *requires* association end between the class *Placeholder* and *Interface* in Fig. 1. The individual operations that constitute the interface of a placeholder are represented by the class *Operation* in Fig. 1. As shown in the figure an operation has a set of parameters: i) *name*; ii) *input*, and; iii) *output*. *Name* is used as an identifier for the *Operation* and the *input* and *output* are a set of *Parameters*. If we assume that an activity `PaymentService` is to be invoked, the name of the operation could be `payment` and the input/output could be the items to be purchased, the number of the credit card and the address for the items to be delivered. *Placeholders* may also be characterized by their SPDI and QoS properties. A property (class *Property* in Fig. 1) has: i) *name*; ii) *type*; iii) *verification*; iv) *category*, and; v) *dataState*. The attribute *type* refers to the state of the property, which can be required or confirmed. A required property is a property that a placeholder must hold to be included in an orchestration. For example, if the required property of an orchestration is Confidentiality, then all placeholder activities involved in the orchestration and the links between them may be required to have the Confidentiality property. On the other hand, a

confirmed property is a property that is verified at runtime, through a specific means as defined in the Verification. *Verification* is a class that describes the way a *Property* of a *Placeholder* is verified. The verification process 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 the 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 with category attribute, the *Category* enumerator in Fig. 1, shows the different categories. A Property can refer to *confidentiality*, *integrity*, *availability* (covering the Security property), *privacy*, *dependability*, *interoperability* or *QoS*. In this way a classification of the properties is achieved. The final attribute, *dataState*, is referred to state of the data of a *Placeholder* (see enumerator *DataState* in Fig. 1). In this work all three data states are considered, i.e. data in transit, at rest or in processing. If the property requirement is referred to a link between activities, then the state of the data will be *"in_transit"*. If we have to do with an *OrchestrationActivity* bound to a storage service, *dataSate* could be *at_rest*. If the *OrchestrationActivity* is bound to a service or device that processes and transforms data, then *dataSate* could be *in_processing*. Complex orchestrations can involve data in all three data states. Finally, the set of all the SPDI properties that are inferred for the different placeholders of an orchestrator by a pattern are aggregated into a *PropertyPlan* object.

B. Language Constructs

The orchestration-based model view presented in Fig. 1 can be used to define activities, as well as basic control flow operations enabling their composition into complex orchestrations, and to define the associated individual and composition properties. Upon instantiating the orchestration, the abstract definition of the orchestration structure is replaced with actual components (e.g., specific endpoints, network addresses, and functions). In this context, language constructs need to be used to define an orchestration pattern. A textual representation of the model in Fig. 1 in the form of an EBNF [20] grammar is used as input to the Eclipse ANTLR4 [21] plugin for the creation of lexer and parser. In this way, any input can be checked for compliance with the defined grammar. For the sake of brevity the whole grammar is , a sample for the definition of a *Placeholder* is presented below and not the whole grammar:

```
placeholder : placeholdertitle OPEN_PAREN
placeholderid COMMA interfacename (COMMA
interfacename) COMMA propertyname (COMMA
propertyname) CLOSE_PAREN | orchestration |
orchestrationactivity ;
```

C. SPDI Patterns Specification

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 four parts:

I. The ***Activity Properties (AP)*** part; defines the activity-level 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.
II. The ***Orchestration (ORCH)*** part; defines the abstract form of the orchestration that the pattern applies to, thus the ORCH is specified as an orchestration of abstract activity placeholders. When the pattern is matched to a specific orchestration, the placeholders in its ORCH may be bound to specific activities or sub-orchestrations of it.
III. The ***Conditions*** part; 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.
IV. The ***Orchestration Properties (OP)*** part; defines the orchestration-level SPDI properties that the pattern guarantees for the orchestration specified in its ORCH part.

Based on the above, a semantic interpretation of an SPDI pattern having the above structure is that if the *AP* properties that have been specified for the activity placeholders in the orchestration of the pattern and the *Conditions* of the pattern hold (verified as `True`), then the *OP* property specified in the pattern also holds for the whole *ORCH*. Formally, this can be expressed as:

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

where $\models$ denotes the entailment relation that has been established by the proof of the pattern. *AP*s are materialized using the *Property* class in Fig. 1; Property *name* identifies uniquely the SPDI property and the *PropertySubject* depicts the *Placeholder* that implements the activity for which the property is required or verifiable (*PropertyType*). In the latter case, *PropertyVerification* depicts how the verification takes place. *PropertyCategory* classifies the SPDI property, while *DataState* show that state of the data used by the *Placeholder*. *ORCH* is an Orchestration object including *Placeholders* of type *UnassignedActivity*, making our model parametric since it does not have to refer to exact placeholders. *Conditions* are materialized using the *Operation* and *Parameters* classes. Inputs and outputs of the activity placeholders of the SPDI pattern are defined in the objects of those two classes. Finally, *OP* is an orchestration-wide *Property* object. That means that values of some of its attributes are pre-defined, such as the *PropertySubject*, which is the *ORCH* described above, and the *DataState* that is set to `end-to-end` to indicate it refers to an orchestration-wide property.

IV. Implementation Approach

A. Machine-processable Pattern Encoding

An important requirement for implementing the SPDI pattern-driven management and adaptation of the IoT infrastructure is to support the automated processing of developed patterns. To achieve this, the SPDI patterns can be expressed as *Drools* [23] business production rules, and the associated rule engine, by applying and extending the Rete algorithm [24]. The latter is an efficient pattern-matching algorithm known to scale well for large numbers of rules and data sets of facts, thus allowing for an efficient implementation of the pattern-based reasoning process. A Drools production rule has the following generic structure: **`rule`** `name <attributes>*` **`when`** `<conditional element>*` **`then`** `<action>*` **`end`**. Thus, herein Drools are leveraged to encode the relation between *AP* and *OP* properties in SPDI patterns in a way that allows the inference of the *AP* properties required of the activity placeholders present in the *ORCH* of said pattern in order for the *ORCH* to have the SPDI property guaranteed by the pattern. In more detail, the `when` part of the rule encodes the *ORCH* part of the pattern, conditions regarding the inputs and outputs of activities within the *ORCH*, as well as the OP property guaranteed by the patterns for the specific *ORCH*; the `then` part encodes the *AP* (i.e. activity-level) properties which, if satisfied by the *ORCH*s activity placeholders will guarantee the OP property. Leveraging the above, a Drools rule expressing an SPDI pattern encodes $ORCH \wedge Conditions \wedge OP \Rightarrow AP_i(i = 1, , n)$, where AP_i are the AP properties required of the individual nodes bound to the activity placeholders of the SPDI pattern. This is the opposite of the dependency relation proven in the pattern equation (1) defined above. Thus, this encoding allows the inference of the AP_i properties which, if satisfied by the individual activities participating in the *ORCH*, guarantee the satisfaction of the *ORCH*-level SPDI property of it, as encoded in the pattern. This satisfaction of the *OP* property allows for the design (but also the adaptation at runtime) of the *ORCH* in a manner that preserves the *ORCH*-level SPDI property defined in the pattern.

B. Pattern Rule Example - Confidentiality

The preservation of Confidentiality requires that the disclosure of information happens only in an authorised manner; i.e. non-authorised access to information should not be possible. The Perfect Security Property (PSP, [25]) requires low-level users (i.e. a user with restricted access, in contrast to high-level users having full access) who are only allowed to view public information, should not be able to determine anything concerning high-level (confidential) information. Considering the above, let us consider a sequential orchestration P with two activity placeholders, A and B, whereby B is executed after A, and that for each x in $\{P, A, B\}$ the following hold:

– IN^x and OUT^x are the sets of inputs and outputs of x, and $E^x = IN^x \cup OUT^x$
– V^x and C^X are two disjoint subsets of E^x, portioning into public parts and confidential parts respectively.
– The inputs of A are the inputs of the orchestration P
– The inputs of B are the outputs of A
– The outputs of the orchestration P are the outputs of B

Based on the above, the SPDI pattern for preserving *PSP* (i.e. confidentiality) on the service orchestration P can be defined as follows:

AP: – $PSP(A, V^A, C^A)$ and $V^A \subseteq V^P$ and $C^A \cap V^P = \varnothing$
– $PSP(B, V^B, C^B)$ and $V^B \subseteq V^P$ and $C^B \cap V^P = \varnothing$
OP: – $SecReq^P = PSP(P, V^P, C^P)$

Interpreting the pattern above, and as proven in [26], *PSP* then holds on the orchestration P if, for all activity placeholders x in $\{A, B\}$, the following are true:

– $V^X \subseteq V^P$; i.e. the actions of x that reveal public information are part of the actions of P that reveal public information, and
– $C^X \cap V^P = \varnothing$; i.e. the actions of x that reveal confidential information do not include any action of P that reveal public information.

The above conditions are expressed as *AP* properties of the pattern and entail the *PSP* property on P, as expressed in the *OP* part of the pattern. Based on the above, the confidentiality (PSP) pattern can be represented in Drools as shown in Table I. The **`when`** part of the rule specifies: the two activity placeholders *A* and *B*, the order in which *A* and *B* are executed and the conditions between the outputs of *A* and the inputs of *B*, as required by the *PSP* pattern (lines 3-9), and; the *OP* property that can be guaranteed by applying said pattern (lines 10-11). The **`then`** part of the rule generates a security plan that includes the AP security properties that (if satisfied by the activity placeholders selected) would lead to a *ORCH* satisfying the *OP* (i.e. the PSP property). Based on the proof of the *PSP* property detailed earlier, *PSP* is defined as the *AP* property that both placeholders should satisfy (lines 17 and 22, respectively). Moreover, the additional conditions defined are also added to the corresponding *AP*, as can be seen in lines 18-19 and 23-24, respectively.

C. Experimental Results

As an early verification of the feasibility of the proposed approach, a proof of concept environment has been setup based JBoss Drools 7.15, and gRPC (https://www.grpc.io/) with Protocol Buffers Version 3 (https://developers.google.com/protocol-buffers/). A gRPC server is created loading the Pattern Engine with a basic set of Drools on a desktop system (Core i7, 8GB RAM), while a test client is used to make gRPC calls to the server to request verification of the confidentiality pattern rule presented above. Based on the complexity of the modelled IoT environment, i.e. the number of placeholders stored as facts within the Drool knowledge base, the execution time ranges from 19ms for 10 placeholders to 82ms for 100 placeholders.

TABLE I
SPECIFICATION OF PSP PROPERTY VIA DROOLS

```
rule "PSP on Cascade"
when
$A: Placeholder($input : operation.inputs,
$intData : parameters.outputs)
$B: Placeholder(parameters.inputs == $intData,
$output : parameters.outputs)
$ORCH: Sequence(parameters.inputs == $inputs,
parameters.outputs == $outputs,
firstActivity == $A, secondActivity == $B)
$OP: Property( propertyName == "PSP",
subject == $ORCH, satisfied == false)
$SP: PropertyPlan (properties contains $OP)
then
PropertyPlan newPropertyPlan = new
 ↪  newPropertyPlan ($SP);
newPropertyPlan.removeProperty($OP);
Set V_P = $OP.getAttributesMap().get("V");
Property AP_A = new Property($OP, "PSP", $A);
AP_A.getAttributesMap().put("V", new
 ↪  Operation("subset", V_P));
AP_A.getAttributesMap().put("C", new
 ↪  Operation("subset", new
 ↪  Operation("complement",V_P)));
newPropertyPlan.getProperty().add(AP_A);
insert(AP_A);
Property AP_B = new Property($OP, "PSP", $B);
AP_B.getAttributesMap().put("V", new
 ↪  Operation("subset", V_P));
AP_B.getAttributesMap().put("C", new
 ↪  Operation("subset", new
 ↪  Operation("complement",V_P)));
newPropertyPlan.getProperties().add(AP_B);
insert(AP_B);
insert(newPropertyPlan);
end
```

While a more detailed performance evaluation will follow, investigating in more detail the performance impact of modeling more complex environments and supporting and evaluation a larger set of pattern rules, these initial results validate the feasibility of real-time SPDI property verification and the timely triggering of needed adaptations.

V. CONCLUSIONS

This paper presented a novel approach enabling pattern-driven IoT orchestrations with guarantees for SPDI properties across horizontal and vertical compositional structures of IoT deployments. The developed solution, along with the early proof of concept validation results of its feasibility, pave the way for the creation of reasoning engines supporting the execution of patterns at runtime to realize the overall process of monitoring, forming, adapting and managing smart object orchestrations in IoT applications. As future work, said pattern engines will be developed and deployed in the various layers of IoT deployments (field, network, backend), enabling their semi-autonomous operation as well as the centralized (from the backend) definition of IoT orchestrations, along with additional pattern rules covering more SPDI properties. Moreover, approaches for user-friendly definition of orchestrations will be explored, while a comprehensive performance evaluation of the pattern engines will be carried out on heterogeneous platforms.

ACKNOWLEDGMENT

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

REFERENCES

[1] G. Hatzivasilis, K. Fysarakis, O. Soultatos, I. Askoxylakis, I. Papaefstathiou, and G. Demetriou, The Industrial Internet of Things as an enabler for a Circular Economy Hy-LP: A novel IIoT protocol, evaluated on a wind parks SDN/NFV-enabled 5G industrial network, Comput. Commun., vol. 119, pp. 127137, Apr. 2018.
[2] S. Katsikeas et al., Lightweight & secure industrial IoT communications via the MQ telemetry transport protocol, in 2017 IEEE Symposium on Computers and Communications (ISCC), 2017, pp. 11931200.
[3] S. Tennina et al., WSN4QoL: WSNs for remote patient monitoring in e-Health applications, in 2016 IEEE International Conference on Communications (ICC), 2016, pp. 16.
[4] A. Botta et al., Integration of Cloud computing and Internet of Things: A survey, Futur. Gener. Comput. Syst., 2016.
[5] I. Lee and K. Lee, The Internet of Things (IoT): Applications, investments, and challenges for enterprises, Bus. Horiz., vol. 58, no. 4, pp. 431440, Jul. 2015.
[6] E. Kartsakli et al., A survey on M2M systems for mhealth: a wireless communications perspective, Sensors (Switzerland). 2014.
[7] M. Bartoletti et al., Semantics-based design for secure web services, IEEE Trans. Softw. Eng., 2008.
[8] Deubler M., et al. "Sound development of secure service-based systems." In Proc. of the 2nd Int. Conf. on Service oriented computing. ACM, 2004.
[9] G. Georg et al., Verification and trade-off analysis of security properties in UML system models, IEEE Trans. Softw. Eng., 2010.
[10] J. Dong, T. Peng, and Y. Zhao, Automated verification of security pattern compositions, Inf. Softw. Technol., 2010.
[11] I. Siveroni, A. Zisman, and G. Spanoudakis, A UML-based static verification framework for security, Requir. Eng., 2010.
[12] S. Rossi, Model checking adaptive multilevel service compositions, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2012.
[13] A. R. R. Souza et al., Incorporating security requirements into service composition: From modelling to execution, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2009.
[14] L. Pino et al., Discovering secure service compositions, in CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science, 2014.
[15] L. Pino et al., Designing secure service workflows in BPEL, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2014.
[16] L. Pino et al., Pattern Based Design and Verification of Secure Service Compositions, IEEE Trans. Serv. Comput., 2017.
[17] A. Maa et al., Extensions to Pattern Formats for Cyber Physical Systems, in Proceedings of the 21st Conference on Pattern Languages of Programs, 2014, pp. 15:1–15:8.
[18] N. E. Petroulakis et al., Patterns for the design of secure and dependable software defined networks, Comput. Networks, 2016.
[19] N. E. Petroulakis et al., Fault Tolerance Using an SDN Pattern Framework, in 2017 IEEE Global Communications Conference, GLOBECOM 2017 - Proceedings, 2018.
[20] Extended Backus-Naur Form, https://tomassetti.me/ebnf
[21] ANother Tool for Language Recognition, https://www.antlr.org
[22] G. Eason et al., On Certain Integrals of Lipschitz-Hankel Type Involving Products of Bessel Functions, Philos. Trans. R. Soc. A Math. Phys. Eng. Sci., 1955.
[23] Business Rules Management System (BRMS), https://www.drools.org
[24] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artif. Intell., vol. 19, no. 1, pp. 1737, Sep. 1982.
[25] A. Zakinthinos and E. S. Lee, General theory of security properties, in Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, 1997.
[26] M. Maidl, Common fragment of CTL and LTL, in Annual Symposium on Foundations of Computer Science - Proceedings, 2000.