SEMIoTICS

Deliverable D4.1
SEMIoTICS SPDI Patterns (first draft)

Deliverable release date	Initial 22.04.2019, revised 25.11.2019
Authors	1. Konstantinos Fysarakis, Jason Somarakis, Konstantina Koloutsou (STS) 2. Nikolaos Petroulakis, Manos Papoutsakis, Othonas Soultatos (FORTH) 3. Tobias Marktscheffel (UP) 4. Łukasz Ciechomski, Karolina Walędzik (BS) 5. Arne Bröring (SAG)
Responsible person	Konstantinos Fysarakis (STS)
Reviewed by	Nikolaos Petroulakis (FORTH), Georgios Spanoudakis (STS), Tobias Marktscheffel (UP), Karolina Walędzik (BS), Arne Bröring (SAG)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Verikoukis Christos) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0 revised
Dissemination level	Public

Table of Contents

Acronyms Table	
Acronym	**Definition**
SPDI	Security, Privacy, Dependability, Interoperability
IoT	Internet of Things
SDN	Software-defined networking
QoS	Quality of Service
API	Application Programming Interface
E2E	End to End
IIoT	Industrial Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
GDPR	General Data Protection Regulation
NIS Directive	Directive on security of network and information systems
UC	Use Case
SARA	Service Availability and Readiness Assessment
HIPAA	Health Insurance Portability and Accountability
HHSa	Health and Human Services Agency
PHR	Personal Health Record
CRC	Cyclic Redundancy Check
DoS	Denial of Service
HTTP	Hypertext Transfer Protocol
CoAP	Constrained Application Protocol
XMP	eXtensible Multimedia Protocol
MQTT	Message Queuing Telemetry Transport
DDS	Data Distribution Service
DPWS	Devices Profile for Web Services
UPnP	Universal Plug and Play
OSGi	Open Services Gateway initiative
TCP	Transmission Control Protocol
M2M	Machine to machine
VLAN	Virtual LAN
VXLAN	Virtual Extensible LAN

GRE	Generic Routing Encapsulation
IDS	Intrusion Detection System
CPU	Central Processing Unit
VNF	Virtual Network Function
ACS	AREAS Cloud Service
EMF	Eclipse Modelling Framework
UML	Unified Modelling Language
WF	Workflow

1 INTRODUCTION

This deliverable is the first output of Task 4.1 ("Architectural SPDI Patterns") and provides the initial version of the language for specifying SPDI patterns, referred to as pattern-language in the rest of this deliverable, and the initial set of SPDI patterns developed in SEMIoTICS. This is directly targeting the first key overarching objective of WP4, which is to: *"Define a language for specifying machine interpretable SPDI patterns and develop patterns encoding horizontal and vertical ways of composing parts of IoT applications that can evidently guarantee SPDI properties across heterogeneous smart objects and components from all layers of the IoT application implementation stack."*

In more detail, Task 4.1 activities focus on defining a language for specifying machine interpretable SPDI patterns and then develop and specify, 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. Such properties may apply across heterogeneous smart objects and components from all layers of the implementation stack of an IoT application. Thus, this deliverable presents the initial outcomes of Task 4.1. More specifically, it presents the requirements, design process and the first version of the definition of the pattern-language that will be used for the specification of the SEMIoTICS SPDI patterns. It also presents a first set of SPDI patterns. The final versions of all of these aspects will be documented in the final version of this deliverable, i.e. on D4.8 "SEMIoTICS SPDI Patterns (final)", to be delivered in month 28 of the project.

The pattern language itself is based on a system model defined and presented within this deliverable. 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., backend services), and the associated SPDI and QoS 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 to allow for automated verification of the properties and the triggering of adaptations is presented as well.

Moreover, a set of preliminary SPDI patterns have been defined in the deliverable, covering each of the key properties (i.e., Confidentiality, Integrity, Availability, Dependability and Interoperability) and the different data states (data in transit, at rest, and in process). A representation of these in machine-processable format is also included, covering an important implementation aspect that will enable the automated processing, verification and adaptation driven by the patterns.

In addition to the above, the deliverable also presents some pointers and examples of the envisioned integration of the above pattern-driven elements with the Recipes approach. The latter allows the definition of abstract IoT orchestrations, hiding the implementation details from the end user. The user (e.g., IoT service provider or application developer) does not need to have expertise in configuring the network and physical connections between the involved IoT devices, can use the Recipe definition tool to define this intended application and the required SPDI and QoS at a high level. Then, these can be automatically instantiated (choosing specific implementations of the included elements), via by the tool and the underlying technologies. Thus, the integration of Recipes and Patterns will enable the user-friendly, abstract definition of IoT orchestrations (through Recipes) with SPDI and QoS guarantees for said orchestrations, both at design and runtime (through Patterns).

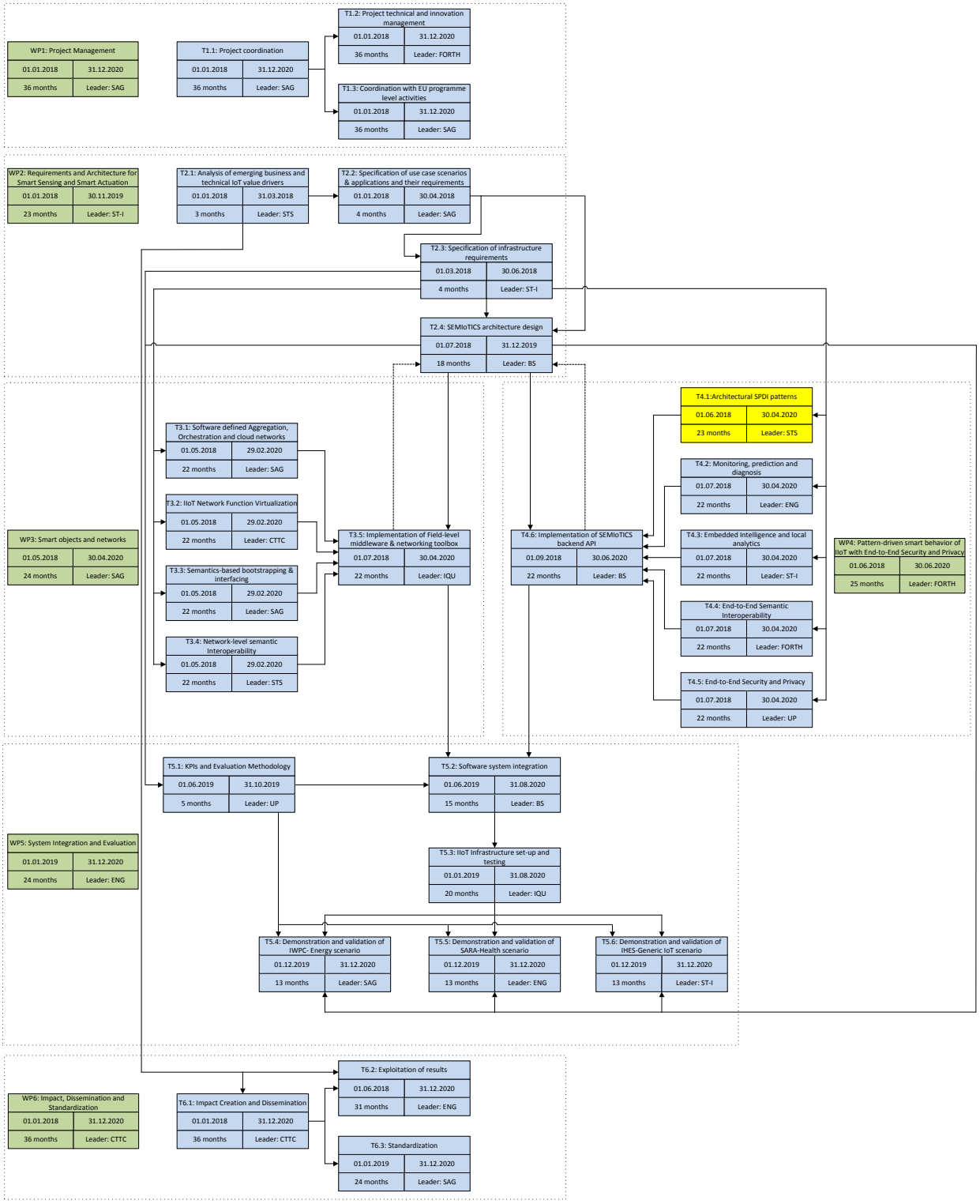
In more detail, and considering the above, the deliverable is structured as follows:

- Chapter 2 summarises the pattern language requirements that will have to be considered when defining the language and the patterns.
- Chapter 3 features the pattern language definition, including the presentation of the SEMIoTICS system model derived, the grammar of the language and the way this will be translated into a machine-processable format to enable the automated SPDI-driven processing and adaptation.
- Chapter 4 provides an initial set of patterns that have been specified based on the language defined in Chapter 3, covering all core property types, different data states and connectivity/interaction types
- Chapter 5 and 6 present the concept of Recipes, leveraged to define IoT orchestrations in a usable manner and pointers to the integration approach to be followed for integrating Recipes with the

SEMIoTICS pattern-driven SPDI monitoring and adaptation approach, along with some examples of its use.

- Finally, Section 7 features the concluding remarks of the deliverable, with pointers to the next steps and the planned update to the content provided herein.

1.1 PERT chart of SEMIoTICS



Please note that the PERT chart is kept on task level for better readability.

2 SPDI PATTERN REQUIREMENTS

An important first step in the development of the SEMIoTICS pattern language is to define the requirements that will guide the development of said language.

In this context, it is essential to consider how the pattern language will be used to specify machine interpretable SPDI patterns supporting:

- the **composition structure** of the IoT applications and platform components;
- the **end-to-end SPDI properties** guaranteed by the pattern;
- the **smart object/component/activity level SPDI properties** required for the end-to-end SPDI properties to hold;
- conditions about pattern components that need to be **monitored** at runtime to ensure
- end-to-end SPDI properties; and
- ways of **adapting and/or replacing** individual IoT application smart objects/components that instantiate the pattern if it becomes necessary at runtime (e.g., when some components become unavailable).

Moreover, the SPDI language will need to be able to support the definition of all SPDI properties, including the six core property types, namely*:*

- Security (S), i.e. Confidentiality, integrity and availability,
- Privacy (P),
- Dependability (D) and
- Interoperability (I).

The above will be considered in all three data states:

- Data-in-transit,
- Data-at-rest, and
- Data-in-processing.

...and two cases of IoT platform connectivity:

- Within the SEMIoTICS platform
- Across IoT platforms

Considering these aspects, the detailed requirements are analysed in the subsections below, organized per SPDI property.

2.1 Security

Security is generally composed of the three properties of confidentiality, integrity, and availability, sometimes also abbreviated as CIA. In more detail:

- **Confidentiality:** the disclosure of information happens only in an authorised manner; i.e. non-authorised access to information should not be possible.
- **Integrity:** maintenance and assurance of the accuracy and consistency of data.
- **Availability:** the invocation of an operation to access some information or use a resource leads to a correct response to the request.

Therefore, for the pattern language, we will also develop patterns covering these three aspects, at the component as well as at the end-to-end and workflow level.

In terms of the **composition** structures of IoT applications and platform components, the following must be considered:

- **Confidentiality:** End-to-end confidentiality can be composed as confidentiality of each link, of each platform handling the data, and of each platform processing the data. If one link or one platform fails to achieve the property, then the property is broken end-to-end.
- **Integrity:** End-to-end integrity can be composed as integrity of each link and of each platform handling the data. If one link or one platform fails to achieve the property, then the property is broken end-to-

end. For data-in-processing, integrity is typically irrelevant, as in most changes said processing changes data; though there are cases where integrity of the processing would need to me monitored (e.g. through internal checks in the processing functions). Data links in this context are logical links and not network links. In particular, some SDN nodes may not be endpoints of a data link. Instead, there may be a direct logical link between gateway and backend, preserving confidentiality and/or integrity from gateway to backend.

– **Availability:** For availability, we consider mainly availability of network connections. Sensors/actuators, gateways, switches and backend are usually singular components existing only once, i.e. if one of these devices or platforms fails, then overall availability is lost. Thus, as there are no alternatives in these cases, a pattern has no means of ensuring availability. In contrast, on the network layer, SEMIoTICS generally assumes that there are several redundant network connections available: The software defined network (SDN), interconnected by various SDN switches, connects the gateway to the backend. In this case, a connection from gateway to backend is assumed to be available if each intermediate hop on the connection is available. Should at least one intermediate hop from or to an SDN switch become non-available, then the pattern can reroute the connection from gateway to backend (or vice versa) to use a different intermediate route which is available.

In addition to the above, smart object/component/activity level SPDI properties required for the end-to-end properties to hold. All components must provide standardized APIs for security functions which are mandatory to be used, i.e. applications or virtual network functions must not use their own cryptography libraries. This is necessary to be able to monitor use of cryptographic functions in order to enforce patterns.

Monitored conditions about pattern components to ensure above-mentioned E2E properties are needed. These could include, e.g., encryption enforcement monitors, checks that traffic is encrypted, integrity checks on stored data, or network components, such as SDN controllers and nodes, that are monitored for availability.

An example of such a monitor for the **Availability** property would be simple to devise, as a component is considered to be available if it can be reached via the network and is able to perform specified services. Non-availability can be due, e.g., to loss of network connectivity or the hardware running a network component failing.

For **Confidentiality**, some examples for each state of data could include:

– Data in transit: At least one of the endpoints needs to be monitored. If there is a standard system-wide API for cryptography functions, behavioural monitoring can be used: Before data can be transferred, it has to be encrypted, i.e. a call to a sufficiently strong encryption function must be observed. Depending on the scenario, a call to a key generation function can also be required to generate keys for encryption (and also to distribute them). If these calls, as required by the pattern, are missing, then a warning can be logged and/or the network transmission can be stopped.
– Data at rest: Similarly, to data in transit, behaviour monitoring can be used to determine whether confidentiality of data is ensured using sufficiently strong encryption. Before data is written to a file, there must be a corresponding call to an encryption function. After data is read from a file, there must be a corresponding call to a decryption function.
– Data in processing: For data processing, data at rest must be decrypted. During processing, it must be monitored that there are no unexpected network connections by the data processing process(es).

Furthermore, patterns can also define to which recipients' data may be sent in order to protect confidentiality of data.

Similarly, for **Integrity**:

– Data in transit: Many network protocols provide integrity protection. Thus, if data integrity is required, it must be monitored that protocols meeting this requirement are used.
– Data at rest: Data at rest is usually integrity protected at the hardware level and/or at the file system level.

Finally, runtime adaptations will be needed to ensure the required (and monitored) security properties are maintained, i.e. ways of adapting and/or replacing concrete IoT application smart objects/components that

instantiate the pattern if it becomes necessary at runtime. Thus, these will also need to be encoded in the developed language.

2.2 Privacy

There have been plenty attempts to define privacy over the years but so far, no universal definition could be created. Despite the fact that the claim for privacy is universal, its concrete form differs according to the current era and context (technical landscape) [1]. In any case, IIoT devices generate, process, exchange and store vast amounts of security add safety-critical data as well as privacy-sensitive information hence careful handling is needed, both from an ethical as well as a regulatory perspective (esp. in cases where medical data is involved).

It is important to understand that information collected in a system becomes personal if identity can be correlated with an activity [2]. Such identification can be direct or indirect. The identifier can be a name, an identification number, location data or an online identifier (such as IP address). It may also be specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person [3]. This is why data protect law does not apply to anonymous data (i.e., data in which the data subjects are no longer identifiable). However, if the risk of identification is reasonably high, then the information should be regarded as personal data [4], experience shows that the risks may be quite high [5].

2.2.1 REGULATORY REQUIREMENTS

An important aspect when considering privacy is the compliance with regulations (such as the General Data Protection Regulation of European Union – Regulation (EC) 2016/679 (European Parliament 2016)) [3] and several standards, like the ISO/IEC standards 27018 (ISO/IEC 2014) [6] and 29100 (ISO/IEC 2011) [7]. Some key aspects to be considered in the pattern language design (and the SEMIoTICS approach as a whole) are analysed below.

Under the GDPR[1], data controllers and processors need to ‘’implement appropriate technical and organizational measures’’ (GDPR, Article 32). Such measures shall take into account the following elements:

- State-of-the-art;
- Cost of implementation;
- Nature, scope, context and purposes of the processing; and
- Risk of varying likelihood and severity of the rights and freedoms of natural persons

Nevertheless, the security measures to be implemented should be ‘’appropriate to the risk’’

- the pseudonymization and encryption of personal data;
- the ability to ensure the on-going confidentiality, integrity, availability and resilience of processing systems and services;
- the ability to restore the availability and access to personal data in a timely manner in the event of a physical or technical incident; and
- a process for regularly testing, assessing and evaluating the effectiveness of technical and organizational measures for ensuring the security of the processing.

When considering the NIS directive, the following should be considered:

Essential Service *"a service essential for the maintenance of critical societal and/or economic activities depending on network & information systems, an incident to which would have significant disruptive effects on the service provision ", defined in article 5.*

EU Member States have to identify the operators of essential services established on their territory by 27 months after entry into force of the Directive. Operators active in the following sectors may be included: energy, transport, banking, stock exchange, healthcare, utilities, and digital infrastructure (NIS Directive, Annex II)[2].

[1] https://eugdpr.org/
[2] https://www.enisa.europa.eu/topics/nis-directive

When determining the significance of a disruptive effect in order to identify operators of essential services, the EU Member States must consider the following factors:

- the number of users relying on the service concerned;
    - For health sector, the number of patients under the provider's care per year.
- the dependency of (one of) the sectors mentioned above on the service concerned;
- the impact incidents could have on economic and societal activities or public safety;
- the market shares of the entity concerned;
- the geographic spread of the area that could be affected by an incident;
- the importance of the entity to maintain a sufficient level of the service, taking into account the availability of alternative means for the provision of that service;
    - With regard to energy suppliers, we should be considered circumstances where an incident would have significant disruptive effect on the provision of an essential services. Such factors could include the volume or proportion of national power generated;
- and any other appropriate sector-specific factor (NIS Directive, art 6).

Digital Service "*any service normally provided for remuneration, at a distance, by electronic means and at the individual request of a recipient of services"* (NIS Directive, art 4(5)). The NIS covers three different types of digital services, Online Marketplace, Online search engine and Online computing service.
For our cases we need to consider the Online computing service which is defined as "*services that allow access to a scalable and elastic pool of shareable computing resources".*

2.2.2 PILOT-SPECIFIC PRIVACY ASPECTS

Since the 1st and 2nd pilot of SEMIoTICS focus on specific vertical domains, the intrinsic requirements of each of those verticals has to be considered. The Industrial environment of UC1 has quite different requirements to the healthcare environment of UC2, while the horizontal pilot (UC3) has to be able to consider to these and other vertical domains and their intricacies. Especially for UC2 and the healthcare domain, special care will need to be taken to monitor and safeguard the Privacy properties of components and their orchestrations.

2.2.2.1 HEALTHCARE-SPECIFIC PRIVACY CONSIDERATIONS

Additionally, specific requirements must be met for the demonstration of SEMIoTICS framework in the healthcare pilot, the SARA-Health scenario. Following the example of HIPAA Privacy Rule (Health and Human Services Privacy Rule and Public Health) [8], the definition of protected health information is needed" (HHSa, 2003, p. 1); the definition and limitation of the circumstances in which an individual's protected health information may be used or disclosed (HHSa, 2003, p. 4); the goal is to strike a balance that permits important uses of information, while protecting the privacy of people who seek care and healing (HHSa, 2003, p. 1).

Additionally, End-user consent is crucial in the healthcare sector, patients should have control over their data (e.g. Personal Health Record – PHR); who can access, for how long and under what conditions. Failure to ensure the above may result in significant physical, financial and emotional harm to the patient. Slamanig and Stringle [8] present certain mechanisms for prevent disclosure related attacks;

– **Unlinkability**

  A system containing n users provides unlinkability if the relation of a document Di and a user Uj exists with probability p= 1/n. Hence, an insider or attacker cannot gain any information on links between users and documents by means of solely observing the system.

– **Anonymity**

  It is the state of being not identifiable within a set of subjects X. The degree of anonymity can be measured by the size of anonymity set |X|. For example, anonymity is provided when anonymous user in a set U′⊆U can access document Dj

– **Identity Management**
  A user's identity can be managed by dividing the identity of a person into sub-identities I = {Ipublic, I1,.Ik }, where each sub-identity is a user-chosen pseudonym. A user can assign any sub-identity for any subset

of his PHR/HER records. This allows the hiding of sensitive data via a sub-identity, this protecting them from disclosure attacks.

Data that can be used to traced back the identity or the location of the patient [9] should be carefully handled and be protected, meaning that mechanisms such as encryption [10], HL75 protocols and anonymization/pseydo-anonymization, should be examined. Considering mechanisms like this, we can hide the real identity that is tied to the stored data so that is not directly associable to the patient. Even if that that data somehow ends on an unauthorized user, he will not be able to leverage (e.g. sell, modify it) it, without the encryption key. Although invoking unlinkability/anonymity is very important we also need to consider ways of achieving it so that we don't disturb our data handling, in terms of data incorrectly ascribed (at any point in the System's processing of that data) to another patient. Since this will not only disclosure confidential data of a current patient but also may cause medical problems due to false data for the assessment.

Other than sensor data during the SARA program, audio and video transmissions are captured during tele-presence, SARA through SEMIoTICS must ensure that these communications remain private and follow the same principles as we mentioned above. Storage of the said data, should be limited to what's necessary and accomplished in a secure backend database.

The accessibility of such data should be limited to authorized personnel that registered through a strict (e.g. two factor) authentication process. Furthermore, the principle of least privilege should be used to minimize the exposure of such data on irrelevant parties. For instance, the patient's General Practitioner should have access to all Patient medical records whereas the technician should only have access to technical system configuration information. Using this approach, when an incident occurs we can already narrow our search.

As mentioned above, consent is crucial on nowadays technical landscape and more importantly to sensitive health related data gathered by IoT devices. We must consider mechanisms that ensure that the patient (or close relatives) should always be properly informed (e.g. by the RA) prior of using the service. This includes notifications to the user, whenever a tele-presence session is about to begin & to end, and the clarification of the identity of the person responsible for that session (e.g. remote operator)

When considering the **composition structures** of IoT applications and platform components, the following aspects should be highlighted:

- *Data pseudonymization in-transit:* Data's identifier should not be able to be tracked directly during transit.
- *Data pseudonymization at rest:* Data's identifier won't be able to be tracked directly during rest.
- *Data pseudonymization in-transformation:* Pseudonymization should be reversible and should still remain difficult to track directly during transformation. Transformation should be reversible while keeping the pseudonymization.

In the privacy context, the **E2E properties** that must be guaranteed by the patterns and their protection mechanisms should include

- Data collection
  - Consent
  - Opt-in
  - Fairness
- Data access
  - Identifiability
  - Notification
  - Auditability
  - Challenge compliance (Accountability)
- Data usage
  - Retention
  - Disposal
  - Report
  - Break or incident

For the E2E properties to hold, additional **component-level** properties must be identified and guaranteed by the patterns. Regarding sensors, initially, proper configuration must be attained via certain pattern mechanisms; these mechanisms need to be easily repeatable in order for re-configuration on runtime to apply. Appropriate, authentication must be achieved through identifiable attributes that can be integrated in the sensor's hardware such as Trusted Platform Module [11]. Additionally, it is important that the sensors have enough resources (e.g. computational power) to complete as much processing of the data as possible at their end, to effectively avoid leaking identifiable or user related data to interested parties; if they do need to send such data for the purposes of SEMIoTICS, then operations to encrypt and anonymize the data must be supported by the computational environment and performed prior sending them. Malfunctioning sensors, that can no longer guarantee the properties above should be detected by the Sensing unit's dedicated firewall and reported to the IoT gateway.

For both **E2E** and **component-level** properties to endure through the pattern language mechanisms, SEMIoTICS will **monitor** certain conditions and make necessary **runtime adaptations**.

Authentication and authorization services throughout the framework (cross-layer) and between all components (cross-platform) must be observed carefully to ensure only approved and appropriate (e.g. privilege wise) interactions occur. In the case that an abuse is detected, specific pattern-based mechanisms must engage to notify related components, such as an IoT gateway, to compel certain actions (e.g. shutdown a sensing unit). Moreover, the patterns must track the procedures that interact with sensitive data and intend to secure them; including protection of sensitive data at rest and at transit operations (e.g. encryption); mechanisms that ensure only the necessary data is aggregated, stored, processed and send (minimization, under GDPR); mechanisms that offer secure disposition/deletion of unimportant, no longer relevant or personal data (under GDPR's right to be forgotten). In the event of misuse of such mechanisms, due to misconfiguration/malfunction/malicious activity, specific privacy-pattern-driven operations will be used in runtime to tackle this.

2.3 Dependability

Dependability is the ability of a system to deliver its intended level of service to its users [12]. The main attributes which constitute dependability are reliability, availability, safety and maintainability. Dependable systems impose the necessity to provide higher fault and intrusion tolerance. The satisfaction of these attributes can avoid threats such as faults, errors and failures offering fault prevention, fault tolerance and fault detection. More specifically, dependability in SEMIoTICS is focused on three major attributes such as reliability, availability, and fault tolerance as follows:

- **Reliability** is the ability of a system to perform a required function under stated conditions for a specified period of time [13]. It is an attribute of system dependability and it is also correlated with availability. For hardware components, the property is usually provided by the manufacturer. This is calculated based on the complexity and the age of the component. Reliability can be classified into two main categories the deterministic models and the probabilistic ones.
- **Availability** guarantees that information is available when it is needed [13]. The lack of availability in network transmissions has a severe influence on both the security and the dependability of network. More specifically, network availability is the ability of a system to be operational and accessible when required for use. Moreover, availability in networks is the probability of successful packet reception [14]. Other factors which affect the availability of a link are the transmission range of the signal strength, noise, fading effects, interference, modulation method, and frequency.
- **Fault Tolerance** is the ability of a system or component to continue normal operation despite the presence of hardware or software faults [13]. Network fault tolerance appears to be a critical topic for research [15]. The most common solutions to guarantee fault tolerance and avoid single point of failure, include the replication of paths forwarding traffic in parallel, the use of redundant paths and the ability to switch in case of failure (failover) and traffic diversity. Fault tolerance mechanisms exists in all layers of field, network and backend/cloud. In the field layer, failures involve the drop of sensors or actuators and the gateways. More specifically, fault tolerance in network architectures requires the design of a network able to guarantee avoidance of single or multiple link failures, faulty end hosts and switches, or attacks. The key technical solution of the problem includes the creation of a fault tolerance mechanism to provide open-flexible design where existing fault tolerance solutions are not effective.

Dependability analysis of an IoT system includes whether non-functional requirements such as availability, reliability, safety and maintainability are preserved. The conditions depend on the respective dependability property that the system guarantee. The satisfiability of a property can be defined by a Boolean value (i.e. true, false), an arithmetic measure (i.e. delay) or probability measure (i.e. reliability/uptime availability).

More specifically, for the SEMIoTICS framework a number of different dependability requirements have been defined as follows:

- **Use Case 1 (Wind Energy)** defines that network resource isolation must be performed for guaranteed service properties – i.e. reliability, delay and bandwidth constraints. Furthermore, fail-over and highly available network management shall be performed in the face of either controller or data-plane failures. Finally, decisions made by unreliable, i.e. faulty or malicious SDN controllers, shall be identified and excluded.
- **Use case 2 (SARA)** defines that SEMIoTICS platform should support time- and safety- critical requirements by allowing SARA application logic to be deployed on resource-constrained edge gateways (e.g. smartphones, vehicles, mobile robots). SEMIoTICS platform functionalities should be locally available even in case of failure of communication with the SEMIoTICS cloud nodes. Furthermore, the SEMIoTICS platform should support the SARA solution to manage the trade-off between different requirements (e.g. reliability, power consumption, latency, fault-tolerance) by allowing both SARA application logic and platform features to be distributed over a cluster of gateways (SARA Hubs). Finally, the connectivity should keep track of the field device connectivity state (e.g. to detect anomalies, but also required for higher-level (cognitive) control algorithms).
- In **Use Case 3 (Smart Sensing)**, there are not defined any specific dependability requirements. However, the previously described properties such as reliability, safety and availability requirements should be also satisfied for the component in the field layer as involved for this scenario.

In terms of the **Composition structures** of IoT applications and platform components, the following aspects should be noted: The definition of the composition includes also a set of constraints as requirements that should be satisfied by the individual components composed or by the component composition as a whole. These constraints may represent functional requirements such as connectivity and reachability. Considering the functional requirements, the connectivity between the different components is one of mode crucial requirements of the component composition. Different parameters such as the distance between network nodes that is a topological constraint for a network may also be expressed through patterns constraints. For instance, in wired networks this connectivity can be satisfied using suitable interfaces and cables.

However, in IoT devices such as wireless sensors, the connectivity is based on the coverage of each IoT device and it can be classified into deterministic and probabilistic models. But for a wireless link the following can be assumed: either a communication link can be characterized as a component having specific properties (propagation, length, interference, noise, etc.) or a link can be a connector which connects two components i.e. two wireless sensors. Other constraints which may be expressed may refer to the quantity and type of nodes, interfaces per nodes, cost and energy consumption. Furthermore, the applications and services in the that make use of the network are crucial factors on the design of a network as they can affect the available resources such as computational power, available memory, storage and networking capabilities.

Based on the above, as components, we may consider the different elements of SEMIoTICS architecture. That includes applications and clouds in the backend cloud, controllers, switches in the network level, and finally, sensors, gateways and actuators in the field devices.

In terms of the **E2E properties** that must be guaranteed by the patterns, the need for end-to-end dependability between the heterogeneous IoT devices (at the field level), the heterogeneous IoT Platforms (at the backend cloud level) and the network level include high adaptation capability to accommodate different dependability needs such as reliable communication, availability and low latency.

Monitored Conditions about pattern components to ensure above-mentioned E2E properties should include

failure monitoring and detection, which is required to discover link failures and packet losses in order to identify lack of network availability. To do so, a suitable mechanism is able to dynamically monitoring path and component conditions. For instance, in network layer, when there are dropped packets between two nodes or

the link is down, the monitoring mechanism detects it as failure. This can be done also by the use of node connector statistics as fetched by switches such as receive/transmit packets, errors, drops CRC errors and collisions in the SDN components. Furthermore, these statistics can be used as an intrusion detection mechanism to forward traffic to different secure paths. In SEMIoTICS, suitable mechanism should be defined for monitoring the components on the different layers of SEMIoTICS architecture.

When the monitors detect unwanted alterations of the dependability state, ways of adapting and/or replacing concrete IoT application smart objects/components that instantiate the pattern should be present, if it becomes necessary, at runtime. **Dependability-driven adaptations** can be used at runtime when the property is violated in case of DoS attacks. In case of a network fall, new alternative network paths or components must be found. However, the most important factor for runtime adaptation appear to be after the detection and the identification of an attack/ failure, the required adaptation time for restoration. Apart from the proactive definition of the respective paths, a reactive mechanism should exist to dynamically allocate paths for fast fault detection and restoration, which is required to detect link failures and packet losses in order to restore network availability. The abstract form of the fault detection and restoration of network faults or attacks and can be applied first locally and then globally. In SEMIoTICS, suitable mechanism should be defined to replace, re-instantiate, or reroute traffic at runtime adaptation.

2.4 Interoperability

Desired interoperability characteristic imposes special requirements on the designed SEMIoTICS framework. Interoperability gives an ability to a system or a product to connect and work with other systems or products. Interoperability is defined as *a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, in either implementation or access, without any restrictions [16].*

The following types of interoperability can be distinguished and will be covered by SEMIoTICS:

- **Technological interoperability** – enables seamless operation and cooperation on heterogeneous devices that utilize different communication protocols
- **Syntactic interoperability** – establishes clearly defined formats for data, interfaces and encoding
- **Semantic interoperability** – settles commonly agreed information models and ontologies for the used terms that are processed by the interfaces or are included in exchanged data
  Organizational interoperability – cross-domain service integration and orchestration through common semantic and programming interfaces.

Considering the composition structures of IoT applications and platform components, and from the perspective of semantic operability, an information about every entity should be available through dedicated interface. There should be interfaces for exchanging information about entities, values of their attributes, metadata and availability status. Similarly, IoT platforms services should be available to use via dedicated interface. Some key components to be considered in the regard, are highlighted below.

Semantic Broker: Although, a common interpretation of exchanged information is a desired characteristic of designed SEMIoTICS solution, shared ontology between local systems is not always available. Thus, it is necessary to enable interaction in indirect way. The Semantic Information Broker proposed in [17] can be used for this purpose. This component is responsible for correlating required information and enabling the interoperability of systems with different semantics as well as cross-domain interaction.

The metadata interoperability is a prerequisite for uniform access to objects in multiple autonomous and heterogeneous systems. Domain experts must establish the metadata interoperability model before uniform access can be achieved [18]. *Metadata interoperability can be defined as a qualitative property of metadata information objects that enables systems and applications to work with or use these objects across system boundaries* [18].

Mechanisms that can be used to reconciling heterogeneities among models are: language mapping, schema mapping, instance transformation and metadata mapping [18]. For example, temperature units can be Fahrenheit, Celsius or Kelvin, but they express the same information which can be obtained after proper

instance transformation (Figure 1) using the correct mathematical formula. Semantic ontology for each domain (healthcare, smart sensing, renewable energy) should be established first by domain experts.

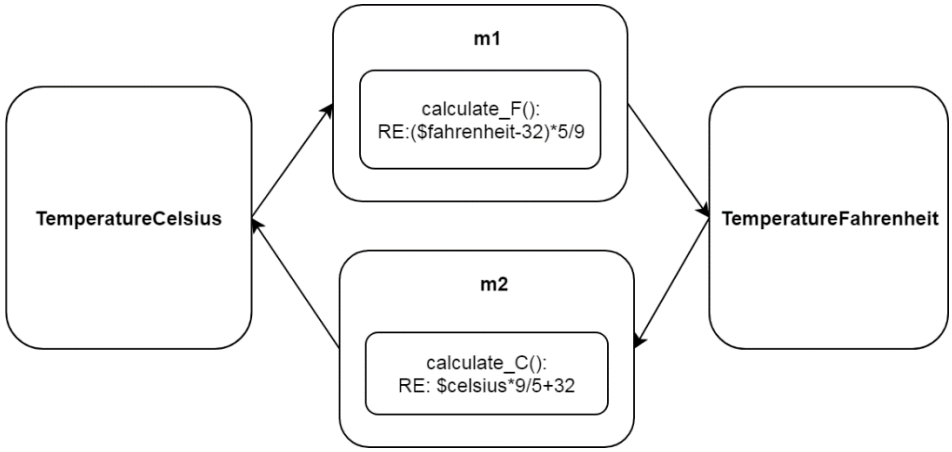


FIGURE 1 ACHIEVING METADATA INTEROPERABILITY BY INSTANCE TRANSFORMATION MAPPING ON THE EXAMPLE OF TEMPERATURE

Common Programming Interface: Furthermore, a common Application Programming Interface (API) is established by EU funded project BIG IoT between different IoT middleware platforms. This approach will ease the development of software services and applications for different platforms.

Functionalities provided by such an API can also implement interoperability on device-, fog-, and cloud-level. The main functionalities of API:

- Identity management and registration to resources
- Resource discovery based on user-defined criteria
- Access to data or metadata (publish/subscribe streams)
- Command forwarding to things enabling smart actuation
- Vocabulary management of semantic information
- Security management (authentication, authorization, key management)

Charging and billing management for using providing assets.

In terms of the corresponding **E2E properties** that must be guaranteed by the patterns, and from the perspective of IoT landscape, interoperability means that every smart object should be seamlessly plugged into a system without additional effort while the whole process of establishing meaningful connection should be as transparent as possible. The data collected by smart objects should be sent automatically in a way that ensures user requirements and this data should be completely understood in the destination place without any loss of data (or with acceptable minimal one). Established connection must have confidentiality and integrity properties, but other patterns will be responsible for that. From the destination perspective it should be also possible to seamlessly interact with smart objects like actuators to enable actions in response to corresponding events generated by analytics backend. To fulfil interoperability pattern requirements SEMIoTICS framework should have an ability not only to recognize and balance the heterogeneous capabilities and constraints of smart objects and to correctly interpret data generated by this objects, but also to establish meaningful connections between different IoT platforms.

IoT ecosystem's end-to-end interoperability features that should be guaranteed by service orchestration-focused patterns, referred to as Recipes, introduced in the context of the BIG IoT project[3]. These patterns are listed below [19].

- **Cross platform** – applications or services access resources from multiple platform though the common interfaces.

[3] http://big-iot.eu/

- **Platform-scale independence** – integrates the resources from platforms at different scale in the way that application can uniformly aggregate information for different scale platforms (cloud-, fog-, device-level).
- **Platform independence** – refers to distinct platforms that implement the same functionality in the way that ensures that a single driver application can interoperate with both platforms in a uniform manner without requiring any changes.
- **Cross application domain** – refers to uniform access to information from platforms that process data from different domains.
- **Higher-level service facades** – services can also interact themselves through common API. Therefore, a single application can interact with two platforms to create value-added operations.

Smart object/component level interoperability properties are required in this case as well, for the end-to-end properties to hold. A table with component-level interoperability properties is presented below.

TABLE 1 COMPONENT-LEVEL INTEROPERABILITY PROPERTIES

Component	Requirements for vertical interoperability	Requirements for horizontal interoperability
Smart object	***Technological interoperability*** - device should have a protocol which enables to operate with framework	***Technological interoperability*** – Two device should have the same communicate interface and thus be able to work with each other
Gateway	***Technological interoperability*** – gateways should implement multimode radios and support different technologies (Wi-Fi, Bluetooth, ZigBee) ***Syntactic interoperability*** – gateway proxies for messaging protocols converting messages from one messaging protocol to compatible format of another protocol (RESTful HTTP, CoAP, XMP, MQTT, DDS, platform specific protocols <DPWS, UPnP, OSGi> or other protocols)	
Network	***Technological interoperability*** – network elements should support the same technologies (e.g., wired Ethernet)	***Technological interoperability*** – network elements should support the same technologies (e.g., wired Ethernet) ***Syntactic interoperability*** – Considering the presence of SDN, network elements should support the same protocol for control flows (OpenFlow)

SEMIoTICS Backend/Cloud	***Semantic interoperability*** and ***Cross-domain/organizational interoperability*** - usage of some services like Semantic Information Brokers correlating required information and enabling interoperability of the system ***Semantic interoperability*** and ***Cross-domain/organizational interoperability*** - common Application Programming Interface (API) for connecting platforms and objects to a SEMIoTICS framework	
IoT Platform / services	***Technological interoperability*** - service should have an API which enables to operate with framework	***Semantic interoperability*** and ***Cross-domain/organizational interoperability*** – common programming interface between different IoT platforms/services to establish meaningful connection and give ability to work with each other

When considering **Monitored Conditions** about pattern components to ensure above-mentioned E2E properties, verifying whether interoperability requirements are satisfied will be possible by testing if devices are able to communicate with SEMIoTICS components without compatibility issues in different scenarios. When any new device is plugged, ensure that data type mappings exist for this smart object, ensure that semantic mechanism exists and the desired IoT platform/service is available.

Interoperability properties such us cross platform property, platform-scale independence, platform independence, cross platform domain and higher-level service facades will be tested to ensure interoperability condition. It should be noted that in the pattern scheme, these properties can also be achieved by the presence of specific certifications that the devices/applications hold, and which, while valid, provide guarantees about the interoperability/compatibility properties of the entity.

In terms of ways of **adapting and/or replacing** concrete IoT application smart objects/components that instantiate the interoperability pattern if it becomes necessary at runtime, the interoperability between each related component should be checked again in case of any observed change in connection occurs, before checking integrity conditions. An any change in connection with the device/smart object, especially disconnection, should be handled by network protocol.

2.5 Requirements Specification considerations

The table below highlights some key pertinent requirements, as defined in deliverable D2.3 (“Requirements specification of SEMIoTICS framework”), also including non-SPDI specific requirements, such as underlying/global requirements, functional requirements etc., that directly or indirectly affect the design of SPDI patterns and which will need to be considered during the pattern language definition.

TABLE 2. PATTERN LANGUAGE REQUIREMENTS

SEMIoTICS Requirement		Pattern language considerations	Reference
Req. ID	**Description**		
R.BC.18	The backend layer must feature SPDI pattern reasoning embedded intelligence capabilities	This is a core set of requirements for the SPDI capabilities that must be covered within the pattern-driven approach developed within T4.1. Individual Pattern reasoning components should be developed and deployed at all layers, while the backend should feature global reasoning capabilities. All reasoning engines should aggregate (through interfacing with monitoring) relevant information needed for said reasoning.	The system model and associated pattern language developed are tailored to the multi-layer approach of SEMIoTICS, also anticipating intra- and cross- layer reasoning. Furthermore, Pattern reasoning components (referred to as Pattern Engines) are embedded at all layers; see subsection 3.6.2.2. The real-time reasoning will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning.
R.BC.19	The backend layer should feature pattern-driven cross-layer orchestration capabilities		
R.BC.20	The backend layer must aggregate intra-layer as well as inter-layer SPDI status information to enable local and global intelligence reasoning and adaptation		
R.NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities		
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation		
R.FD.14	The field layer must feature SPDI pattern reasoning local embedded intelligence capabilities		
R.FD.15	The field layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation		
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	While an indirect set of requirements, the various cross platform and cross layer interactions (including E2E between field and backend) with heterogeneous components will need to be supported and their SPDI properties monitored accordingly.	As can be seen in subsections 3.2 (Language Model) and 3.3 (Language Constructs), instances of Java class *Link* allows Pattern Engines to monitor and verify connectivity among IoT service orchestration

			components. This also encompasses the pattern-driven interoperability mechanisms developed in the context T3.4 (and which are further described in D3.4), which leverage the language and pattern definitions.
R.UC1.1	Automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders		
R.UC2.3	The SEMIoTICS platform SHOULD guarantee proper connectivity between the various components of the SARA distributed application. The SARA solution is a distributed application not only because it uses different cloud services (e.g. AREAS Cloud services, AI services) from different remote computational nodes, but also because the SARA application logic itself is distributed across various edge nodes (SARA Hubs).		
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication)	Other than the aspects of availability and dependability (and associated concepts; e.g. fault tolerance) that are already integral in the SPDI properties, other QoS-related parameters (e.g. latency) can also be accommodated by the pattern language adopted. Moreover, the pattern language must be able to leverage appropriate monitors and interface with the necessary mechanisms to act as an enabler for configuring the network and triggering network updates / reconfigurations, as needed (e.g. for fault tolerance or QoS).	As can be seen in subsections 3.2 (Language Model) and 3.3 (Language Constructs), Java class *Property* owns an attribute *Category,* allowing Pattern Engines to monitor QoS properties of the components of an IoT service orchestration. Moreover, the properties associated with the *Link* class directly affect the requirements relayed to the network layer (with
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration needed by SPDI pattern		
R.GP.7	SDN controller giving feedback for a future generation of SPDI patterns to avoid using the same pattern in case of failure		
R.UC1.5	Fail-over and highly available network management SHALL be performed in the face of either controller or data-plane failures.		
R.UC1.3	There MUST be enabled the definition of network QoS on application-level and automated translation into SDN controller configurations.		

			the associated properties reasoned by the Pattern Engine embedded at the SDN controller; see subsection 3.6.2.2).
R.UC1.4	Network resource isolation MUST be performed for guaranteed Service properties – i.e. reliability, delay and bandwidth constraints.		
R.UC2.15	The SEMIoTICS platform SHOULD provide low latency connectivity between the SARA hubs and cloud services (i.e. AREAS cloud services and AI services) to allow offloading of near real-time computation intensive tasks to the cloud. Therefore, SARA hubs need to send with minimal delay: • raw range data (e.g. from Lidar sensors) to identify proximal objects/objects, • real-time audio stream for speech analysis, and real-time raw video stream (object/people recognition, gesture recognition, posture analysis).		
R.GSP.1	The Intrusion Detection System (IDS) MUST capture and process suspicious traffic.	Considerations regarding any sensitive data that is generated, processed, stored and exchanged at all layers must be considered, enforcing and monitoring the corresponding security mechanisms, especially when different trust domains are involved. Proper authentication and authorization services are a necessity when trying to safeguard the security and privacy of data and services. These aspects must be defined in the pattern language, monitored and enforced, considering the different types of devices (e.g. sensors, network controllers, backend servers), actors (e.g. humans, machines/applications) and interaction types (e.g. maintenance or medical staff, simple users). These, along with cryptographic	Security-related properties (such as Confidentiality) are at the core of the properties covered in the SEMIoTICS system model (subsection 3.2) and associated language (subsection 3.3). Moreover, a first version of security-related pattern rules can be seen in subsection 4.1, while a first set of Privacy Patterns can be seen in subsection 4.2. Moreover, using the pattern language, different verification types can be declared for each of the properties (see subsection 3.2); this can be exploited to define interfaces
R.NL.11	Secure communication with the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules), as well as the communication between VIM, SDN Controller, and MANO, with data paths acting as computing nodes for VNF spinoff.		
R.S.7	The negotiation interface of the SDN Controller SHALL be secure against network-based attacks		
R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.		
R.S.6	Sensors SHALL be able to encrypt the data they generate, i.e. their CPU and memory SHALL be sufficient to perform these cryptographic operations.		

		mechanisms, will need to be used to establish trust within and across domains. Moreover, privacy considerations will have to be included (e.g. protection of private data at rest and in transit, data anonymization and minimisation, data retention; see section 2.2.1 above). In addition to the above, patterns can also be leveraged to monitor and enforce the presence of security mechanisms in different IoT orchestrations.	with the various security mechanisms which will allow the verification of the different SPDI properties associated with them (e.g., monitoring encryption mechanisms that provide the property of Confidentiality). This will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning on relevant security and privacy -related aspects, such as secure deletion of unnecessary data, limitation of sampling via a variant of the mechanisms used to ensure QoS parameters, etc.
R.S.2	Authentication and authorization of the stakeholders MUST be enforced by the Network controller, e.g. through access and role-based lists for different levels of function granularities (overlay, customized access to service, QoS manipulation, etc.)		
R.S.3	Sensors SHALL be identifiable (e.g. by a TPM module/smartcard) and authenticated by the gateway.		

R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.		
R.S.5	Before sensitive data is being transmitted, the respective components SHALL be authenticated as defined by requirements R.S.3 and R.S.4		
R.S.17	There MUST be an interface between the network controller and the network administrators for the designation of the applications' permissions.		
R.S.18	All network functions SHALL be mapped to application permissions		
R.GSP.4	Platforms, e.g. cloud platform and sensor, SHALL be trusted.		
R.GSP.9	The SARA system SHALL provide robust mechanisms to protect Patient-related data…		
R.GSP.10	The SARA system MUST fully comply with all relevant Italian laws governing the privacy, security and storage of sensitive Patient health-related data.		
R.P.1	The collection of raw data MUST be minimized.		
R.P.3	Storage of data MUST be minimized.		
R.P.4	A short data retention period MUST be enforced and maintaining data for longer than necessary avoided.		
R.P.6	Data MUST be anonymized wherever possible by removing the personally identifiable information in order to decrease the risk of unintended disclosure.		
R.P.8	Data MUST be stored in encrypted form.		
R.P.9	Repeated querying for specific data by applications, services, or users that are not intent to act in this manner SHALL be blocked.		
R.UC1.6	Decisions made by unreliable, i.e. faulty or malicious SDN controllers, SHALL be identified and excluded.		
R.GSP.7	The cloud platform SHALL to be able to monitor the execution of an app, in particular its interactions with other apps, the network interface, and APIs.	Events received from monitoring critical aspects of the systems' and subsystems' operation, as highlighted by the pattern	As can be seen in section 3.2 (Language Model) and 3.3 (Language Constructs), the

		language, will need to be aggregated and evaluated by the pattern engine. These will need to encompass SPDI and other parameters (e.g. QoS related), as well as anomalies, indicators of malicious actions, malfunction, resource depletion, failures etc., across the different layers and (physical & logical) components of the SEMIoTICS deployment. Pattern-driven interoperability mechanisms will ensure that these connections can be established, further explored in D3.4. In cases of privacy-sensitive monitoring data (e.g. location of the device), the necessary privacy provisions will need to be enforced.	pattern language that has been created can declare Properties that their verification type is *Monitoring*. That allows for capturing the monitoring critical aspects and enabling the reasoning on parameters related to availability, reliability, . As above, the necessary inputs will be aggregated from the monitoring framework of SEMIoTICS (T4.2/D4.2).
R.UC2.7	The SEMIoTICS platform SHOULD notify periodically the SARA solution about the state of the resources hosted by registered IoT field devices.		
R.UC2.8	The SEMIoTICS connectivity SHOULD keep track of the field device connectivity state (e.g. to detect anomalies, but also required for higher-level (cognitive) control algorithms).		
R.UC3.7	MCU IoT Sensing unit shall be able to send change detection and signal local changes / anomalies to IoT Sensing gateway.		

R.UC3.16	Each registered sensing unit should send to the sensing gateway a keep alive signal on a specified period (e.g. few seconds) to notify the gateway it is correctly working. The sensing gateway should detect by this mean any non-working sensing unit and reconfigure the system accordingly.		
R.UC3.18	Sensing units may be equipped with dedicated FW to detect relevant sensors malfunctioning and report that to the gateway		
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised	Logging is an integral part of security, enabling auditing functions and providing accountability. Moreover, regulatory drivers also necessitate it (e.g. transparency through logging is essential under GDPR). This must be considered in the definition of the pattern language, the associated engine and its monitors, enabling the provision of reliable and trustworthy logging mechanisms both for the various actors as well as the events and reasoning of the pattern engine itself.	All pattern engine components (see subsection 3.6.2.2) feature integrated logging mechanisms that allow for auditing on all pattern-driven reasoning and adaptation actions triggered. In other parts of the SEMIoTICS framework and protected infrastructure, the deployment and monitoring of the proper operation of the logging functions can be introduced as with any other mechanism (see subgroups of requirements above).

2.6 Project KPIs considerations

In addition to the requirements stemming from the project's concept and approach (as described in subsections 2.1 to 2.4), as well as the formally defined project requirements (subsection 2.5), an additional aspect considered are the overarching Objectives and associated KPIs. These are detailed in Table 3 and Table 4.

TABLE 3. PATTERN-SPECIFIC KPIS

Objective		KPI		Relation and Status
#	Description	ID	Description	
1	Development of patterns for	KPI-1.1	Delivery of 36 verified SPDI	Pattern-driven SPDI management is at the core of the SEMIoTICS security-by-design approach. The

	orchestration of smart objects and IoT platform enablers in IoT applications with guaranteed security, privacy, dependability and interoperability (SPDI) properties.		patterns covering the 6 core property types for 3 data states and 2 cases	first set of patterns is defined herein (see Section 4). As aggregated in subsection 4.5, 10 out of 36 patterns are presented, while this first set of patterns covers all key SPDI properties and different data states and cases of platform connectivity.
		KPI-1.2	Machine-processable pattern language	First version of the pattern language is defined and presented in detail herein (see Section 3). This will be updated and refined as the implementation of the SEMIoTICS components and the use case scenarios progresses, and the final version of the language will be presented in D4.8, which updates D4.1.

TABLE 4. CONSIDERATIONS AND RELATION TO OTHER PROJECT OBJECTIVES AND ASSOCIATED KPIS

Objective		KPI		Relation and Status
#	Description	ID	Description	
2	Development of semantic interoperability mechanisms for smart objects, networks and IoT platforms.	KPI-2.1	Semantic descriptions for 6 types of smart objects	The pattern-driven approach of SEMIoTICS has been integrated with the usable IoT orchestrations framework leveraging Thing Descriptions (Task 3.3), as shown in Sections 5 and 6 below.
3	Development of dynamically and self-adaptable monitoring mechanisms supporting integrated and predictive monitoring of smart objects of all layers of the IoT implementation stack in a scalable manner.	KPI-3.1	Delivery of a monitoring management layer for generating monitoring strategies for different checks and configurations of monitors available in the targeted IoT platforms	The initial semantics of a monitoring language, derived in the context of Task 4.2, are presented in D4.2, while the associated SPDI monitoring properties are foreseen in the design of the SPDI model and associated pattern language (see subsections 3.2 and 3.3, respectively).
		KPI-3.2	Delivery of a monitoring language capable of defining platform agnostic monitoring conditions (as part of SPDI patterns), correlations of different IoT platform events that are necessary for this, and predictive monitoring checks	
4	Development of core mechanisms for multi-layered	KPI-4.2	Delivery of adaptation mechanisms that support proactive and	Pattern rules defined herein (see section 4), along with the associated pattern components able to reason on said pattern rules and facts

	embedded intelligence, IoT application adaptation, learning and evolution, and end-to-end security, privacy, accountability and user control.		reactive, as well as horizontal and vertical adaptation actions, related to network, smart objects and IoT platforms with an adaptation time of 15ms	(see subsection 3.6.2), will be key drivers behind the SEMIoTICS adaptations. This is in tandem with the proactive mechanisms (defined within D4.2), backend orchestration for management/adaptation (see D4.6).
		KPI-4.6	Development of 3 new security mechanisms/controls enabling the secure management of smart devices and sensors over programmable industrial networks	The SPDI-focused pattern-driven monitoring and adaptation that is at the heart of the SEMIoTICS concept and is presented herein is one of the three core security-related innovations of the project and a key enabler of the multi-layered embedded intelligence of the platform.
6	Development of a reference prototype of the SEMIoTICS open architecture, demonstrated and evaluated in both IIoT (renewable energy) and IoT (healthcare), as well as in a horizontal use case bridging the two landscapes (smart sensing), and delivery of the respective open API.	KPI-6.1	Reduce Required Manual Interventions.	The semi-autonomous operation of the IoT deployment, through the multi-layered embedded intelligence capabilities that, among others, the pattern-driven approach presented herein provides, aims to reduce manual interventions and also effectively and efficiently mitigate the SPDI-related risks stemming from the faults introduced from erroneous or malicious human actions.

3 PATTERN-LANGUAGE DEFINITION

3.1 Overview

This section defines the Pattern Language. Overall, this language:

- provides constructs for expressing/encoding dependencies between SPDI properties at the component and at the composition/orchestration level.
- is structural; It does not prescribe exactly how the functions should be executed nor, e.g., how the ports ensure communication.
- Supports the static and dynamic verification of SPDI properties.
- It is automatically processable by the SEMIoTICS framework so that IoT applications can be adapted at runtime.

Patterns expressed in this language will enable the pattern-based IoT application management process followed in SEMIoTICS, in which patterns are used to

- design IoT applications that satisfy required SPDI properties
- verify that existing IoT applications satisfy required SPDI properties at design time, prior to the deployment of the application
- enable the adaptation of IoT applications or partial orchestrations of components within them at runtime in a manner that guarantees the satisfaction of required SPDI properties

To fulfil the above, SPDI patterns encode proven dependencies between security, privacy, dependability and interoperability (SPDI) properties of individual components of IoT applications and corresponding properties of orchestrations of such components. More specifically, a pattern encodes relationships of the form

$$P_1 \text{ and } P_2 \text{ and } \ldots \text{ and } P_n \rightarrow P_{n+1}$$

where P_i (i=1,…,n) are properties of individual components and P_{n+1} is a property of the orchestration of these components. The relation encoded by a pattern is an entailment relation.

The runtime adaptations that can be enabled by SPDI patterns may take three forms:

(1) to replace particular components of an orchestration
(2) to change the structure of an orchestration, and
(3) a combination of (1) and (2).

The above types of adaptations are exemplified in Figure 2, where we consider nodes (these could represent any atomic component; e.g., a physical device or a service) which are composed through links (e.g., a network connection) into more complex orchestrations (e.g., a complex workflow involving several services), forming a graph. As shown in the figure at the node level *Node 1* is replaced by *Node 1'*. This adaptation may, for example, become necessary as a component of the role and type of Node 1 may be required (by a pattern) to have a security certificate to prove a property (e.g., encryption of data with a 256 algorithm) that is needed of the component for the overall system to fulfil another property (confidentiality). If at some point during runtime the (encryption) certificate of Node1 expires, the active pattern will trigger the replacement the component. Adaptations may also be at the link level. More specifically, a network link between *Node 1* and *Node 2* may be replaced with two links (two alternative networks) to offer increased redundancy (e.g. to achieve the required levels of dependability). Finally, adaptations may be triggered at graph level. In Figure 2, the graph containing *Node 3*, *Node 4*, *Node 5* and their links is replaced completely with a new graph, containing *Node 3'*, *Node 4'* and *Node 5'* and their new links (e.g. to satisfy the need for a certain end-to-end security property).

SPDI patterns cover the different and heterogeneous orchestration models required for IoT and Industrial IoT (IIoT) applications, covering aspects of both the high-level service orchestration view as well as the deployment view of said applications.

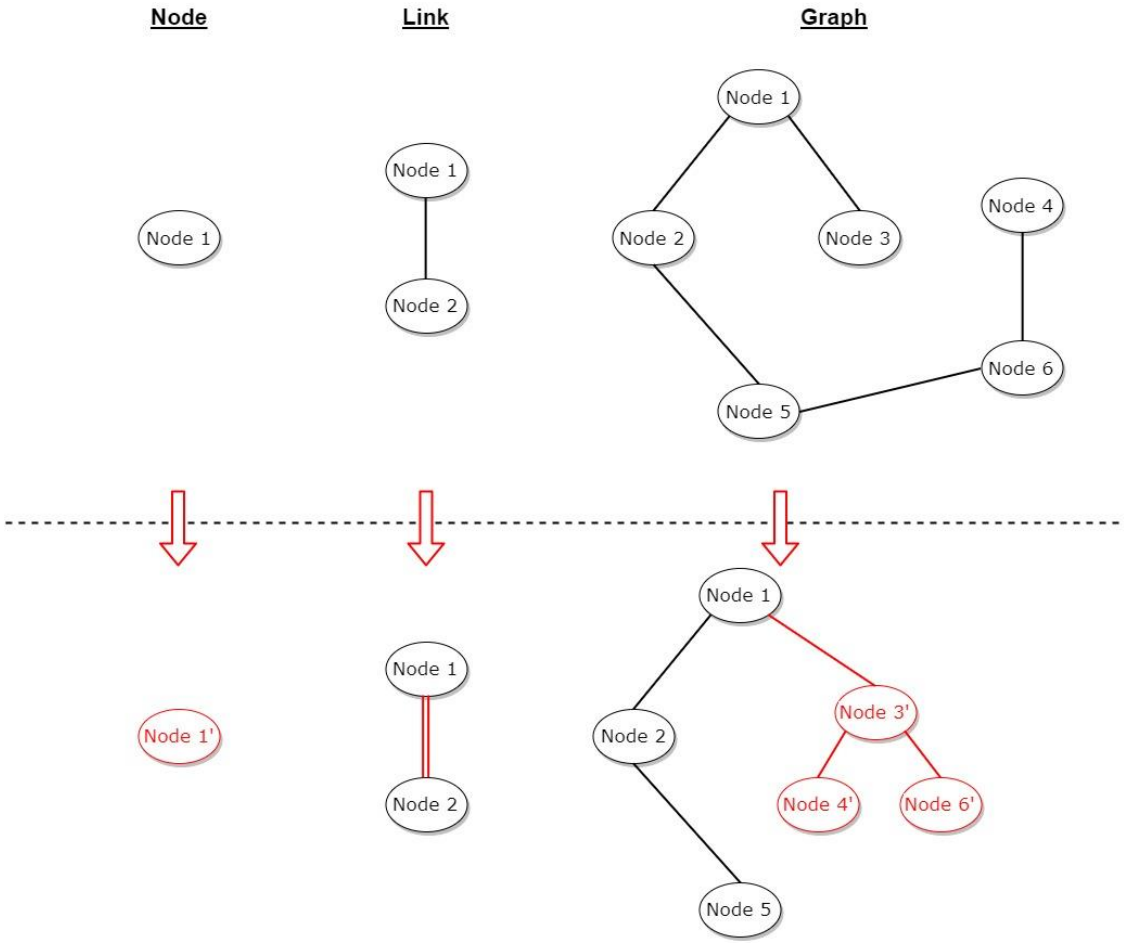


FIGURE 2. EXAMPLES OF PATTERN ADAPTATIONS AT THE NODE, LINK AND GRAPH LEVEL

3.1.1 RELATED WORKS

The popularity of Internet of Things (IoT) brought the realization that the application of semantic technologies (ontologies, semantic annotation, Linked Data and semantic Web services) to IoT offers many advantages with the most important of them being interoperability among IoT resources promoting the interoperability among data providers and consumers, data access and integration, resource discovery, semantic reasoning, and knowledge extraction. Semantic technologies are the principal solutions for the realization of the IoT [21]. However, the ultimate goal of achieving more capable and powerful applications remains and leads to service composition approaches oriented in the area of IoT.

There are some attempts for describing IoT service compositions such as those of [21], [22] and [23] which focus on the energy consumption of the involved IoT devices. The latter pays attention to QoS properties reducing the services search space and the composition time, but none of them take under consideration possible Security properties of the individual IoT services or the whole composition.

Moreover, there is the work of [24] that introduce a contExt Aware web Service dEscription Language (wEASEL) is introduced. wEASEL is an abstract service 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 [25]. 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, [26] present a mechanism that manages IoT choreographies at runtime dynamically. According to their approach IoT service compositions are described by templates called Recipes, which are consisted of Ingredients and their Interactions. In addition, 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 this chapter.

Table 5 in the next page summarizes related works on different approaches/frameworks of service composition. This survey of the SoTA, and considering the expertise available within the consortium, the choice is to combine the expertise on pattern-driven SPDI management with the Recipes approach by [26], tailoring the former to the intricacies of IoT environments covered in SEMIoTICS and extending the latter with SPDI property specification, monitoring and adaptation at design and at runtime (through said patterns).

TABLE 5: RELATED WORKS

Composition Category	Title	Composition Type	Targeted Environment	Service representation	Pros	Cons	Security
Energy-consumption service composition	Zhangbing Zhou, Deng Zhao, Lu Liu, Patrick C.K. Hung, "Energy-aware composition for wireless sensor networks as a service", Future Generation Computer Systems, Volume 80, 2018, Pages 299-310, ISSN 0167-739X, https://doi.org/10.1016/j.future.2017.02.050.	Automated, static	sensor nodes in wireless sensor networks (WSNs)	* **WSN service** is a tuple (nm,dsc, op, eng, spt, tpr), where (i) nm is the name, (ii) dsc is the text description, (iii) op is an operation (functionality), (iv) eng is the remaining energy, (v) spt is the spatial constraint, and (vi) tpr is the temporal constraint. * **Service network** snSC is adirected graph, and is represented as a tuple (SvC (=service classes), Lnk (=direct links), InvP (=invocation possibility))	+ approximately optimal WSN services compositions + no need for the users to represent their requirements in a explisit specification, just input, output and description + energy-aware service composition + high availability	- the linkage between services and physical sensor nodes is not explored - high complexity - low scalability - a certain, but limited number of service classes can be identified in a certain domain (service network)	Not covered / Not mentioned
	Baker, Thar & Asim, Muhammad & Tawfik, Hissam & Aldawsari, Bandar & Buyya, Rajkummar. (2017). An Energy-aware Service Composition Algorithm for Multiple Cloud-based IoT Applications. Journal of Network and Computer Applications. 10.1016/j.jnca.2017.03.008.	Automated, static	IoT	a **service** s is described by its provider in a 3-tuple format (si,so,sec), where sec is the energy required for the service computation at the hosting datacenter, si is the input and so is the output	+ power efficiency of the physical devices incomposition approach + minimum number of IoT services in the composition +transitional relationships between the customer and the datacenters are taken under consideration +superior performance against established composition algorithms	- each cloud provider must have pre-defined/developed composition plans - high time and cost	Not covered / Not mentioned
Data-oriented service composition	A. Urbieta, A. González-Beltrán, S. Ben Mokhtar, M. Anwar Hossain, L. Capra, "Adaptive and context-aware service composition for IoT-based smart cities", Future Generation Computer Systems, Volume 76, 2017, Pages 262-274	Automated, static	IoT	contExt Aware web Service dEscription Language (wEASEL)	+ deals simultaneously with signature and specification matching and supports several concept matching techniques	- no QoS attributes in the evaluation	Not covered / Not mentioned

	Montori, Federico & Bedogni, Luca & Bononi, Luciano. (2017). A Collaborative Internet of Things Architecture for Smart Cities and Environmental Monitoring. IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2017.2720855.	Automated, static	IoT	N/A	+ exploitation of the devices owned by the end users +crowdsensing +homogeneity to data	- end users must want to participate (may want a reward) - some sources may be unreliable - no prediction for sensitive information - single point of failure (server-client system)	Not covered / Not mentioned
IP-based service composition	Kleinfeld, Robert & Steglich, Stephan & Radziwonowicz, Lukasz & Doukas, Charalampos. (2014). glue.things – a Mashup Platform for wiring the Internet of Things with the Internet of Services. 10.13140/2.1.3039.9049.	Designer, static	Web-enabled IoT devices, web services	Json-based data models (triggers and actions)	+ token management of devices + user management + integration of IoT and web services, + Interfaces for registration, configuration and monitoring	- availability of nodes - low scalability	Web service authorization with OAuth
	Chen, Lei & Englund, Cristofer. (2017). Choreographing Services for Smart Cities: Smart Traffic Demonstration. 1-5. 10.1109/VTCSpring.2017.8108625.	Designer	IoT	N/A	+ runtime insurance for services communication, + BPMN usage	- platform under construction, - availability of stakeholders	Filters the interac protocols of the service with resp to different secur requirements
	Doukas, Charalampos & Antonelli, Fabio. (2015). Developing and deploying end-to-end interoperable & discoverable IoT applications. 673-678. 10.1109/ICC.2015.7248399.	Designer (uses Node-RED), static	IoT	iServe (a service warehouse which unifies service publication, analysis, and discovery through the use of lightweight semantics as well as advanced discovery and analytic capabilities.)	+ bridge between REST and MQTT/ WebSockets/STOMP, + reuse of services, + monitoring	- availability of services	Access control, D privacy, Integrity
	Georgios Pierris , Dimosthenis Kothris , Evaggelos Spyrou , Costas Spyropoulos, SYNAISTHISI: an enabling platform for the current internet of things ecosystem, Proceedings of the 19th Panhellenic Conference on Informatics, October 01-03, 2015, Athens, Greece [doi>10.1145/2801948.2802019]	Designer, static	IoT (sensors, processors, actuators)	IoT ontology + SNN ontology + qu-rec20 ontology	+ Reuse of registers services, + secure storage	- no GUI yet, - availability of services, - no runtime monitoring	Authentication, Authorization, Da anonymization in storage

	Mayer, Simon & Verborgh, Ruben & Kovatsch, Matthias & Mattern, Friedemann. (2016). Smart Configuration of Smart Environments. IEEE Transactions on Automation Science and Engineering. 13. 1-9. 10.1109/TASE.2016.2533321.	Automated (goal-driven configuration), dynamic	IoT, Web of things	RESTdesc expressed in Notation3	+ adaptation to dynamic environments, + fault tolerance, + scalability, + correct service composition, + security requirements	- no universal remedy for false compositions, - inefficient reasoning for large number of devices	Confidentiality of data exchanged within a mashup
	J. Seeger, R. A. Deshmukh and A. Broring, "Running Distributed and Dynamic IoT Choreographies," in Global IoT Summit (GIoTS), Bilbao, 2018.	Designer, dynamic	IoT	Recipe, Offerings, OSRs, RRCs	+ Dynamic update of IoT components, + choreography approach, + scalability, + failure detection	- No SDN support yet - Limited availability of offerings	Not covered / Not mentioned
	Huber, Steffen & Seiger, Ronny & Kühnert, André & Schlegel, Thomas. (2016). Using Semantic Queries to Enable Dynamic Service Invocation for Processes in the Internet of Things. 10.1109/ICSC.2016.75.	Designer (table-based editors for Ecore models), dynamic	IoT	arbitrary ontologies can be integrated (DogOnt ontology)	+ concept can be generalized and applied to different models and systems from the BPM and IoT communities + allows for context-sensitive resource allocation	- In large-scale systems containing more than 106 IoT services, service discovery and invocation will most likely take minute - SPARQL queries introduces an additional overhead	Not covered / Not mentioned
Other (No Service Composition	Nambi, S. & Sarkar, Chayan & Prasad, Venkatesha & Biswas, Abdur Rahim. (2014). A unified semantic knowledge base for IoT. 2014 IEEE World Forum on Internet of Things, WF-IoT 2014. 575-580. 10.1109/WF-IoT.2014.6803232.	N/A	IoT	Resource Ontology, Location ontology (extension of GeoNames ontology), Context and Domain Ontologies (Aspect-Scale-Context), Policy ontology (Belief-Desire-Intention-Policy model), Service ontology (extension of OWL-S)	+ The proposed knowledge base integrates several existing ontologies that were mainly related to sensor resources, web services and extends them for IoT	- This is just the knowledge base and they do not mention anything about composition of services	Not covered / Not mentioned
	W. Wang, S. De, R. Toenjes, E. Reetz, and K. Moessner, ''A comprehensive ontology for knowledge representation in the Internet of Things,'' in Proc. IEEE 11th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom), Jun. 2012, pp. 1793–1798	N/A	IoT	IoT service is a subclass of the Service class defined in the OWL-S, therefore, an IoT Service can have one Service Profile and one Process that describe its functional and non-functional properties, as well as links to domain knowledge (e.g., service category and physical location ontologies),	+ functional and non-functional properties for the services, + a model based approach to guide automatic test generation and control	- No QoS and QoL aware methods for service composition and adaptation	Not covered / Not mentioned

	Vögler, Michael & Li, Fei & Claeßens, Markus & Schleicher, Johannes & Sehic, Sanjin & Nastic, Stefan & Dustdar, Schahram. (2015). COLT Collaborative Delivery of Lightweight IoT Applications. 10.1007/978-3-319-19656-5_38.	N/A	IoT	an IoT Application is represented as a self-contained archive with corresponding metadata, containing the following information: (i) a Name that uniquely identifies the application, (ii) a natural language Description, (iii) Provider name and id, (iv) a list of Suitable Devices the application can be deployed and executed on, and (v) a Version number	+ browse the market for applications + buy and deploy applications +monitoring component	- No pricing and revenue sharing models that allow more stakeholders that are involved in the development process, to collaborate	Not covered / Not mentioned

3.2 IoT application architecture and orchestration modelling

The overall objective of SEMIoTICS 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. Thus, the security and other quality properties may be required of such components and their orchestrations. A model with such characteristics will effectively serve as a general "architecture and workflow model" of the IoT application. Once defined, this model will be used in conjunction with patterns to enable the reasoning required for determining the applicability of particular SPDI patterns in specific IoT applications and subsequently reason based on them to enable the different types of adaptation that were introduced in Sect. 4.1.

The development of the language for specifying an IoT application architecture and workflow model (referred to as "IoT application model" in the rest of this deliverable) has also taken into account the requirements identified in Section 2 and the current SoTA presented in subsection 3.1.

For the creation of the IoT application model we used Eclipse that through the EMF modelling framework enables the use of the default tree-based editor. The tree-based editor allows to define properties for classes, attributes and references.

The basic constructs for defining an IoT application model in SEMIoTICS is shown in Figure 3.The figure shows the basic modelling constructs of the language and their relations in the form of a UML diagram[4].

[4] http://www.uml.org/

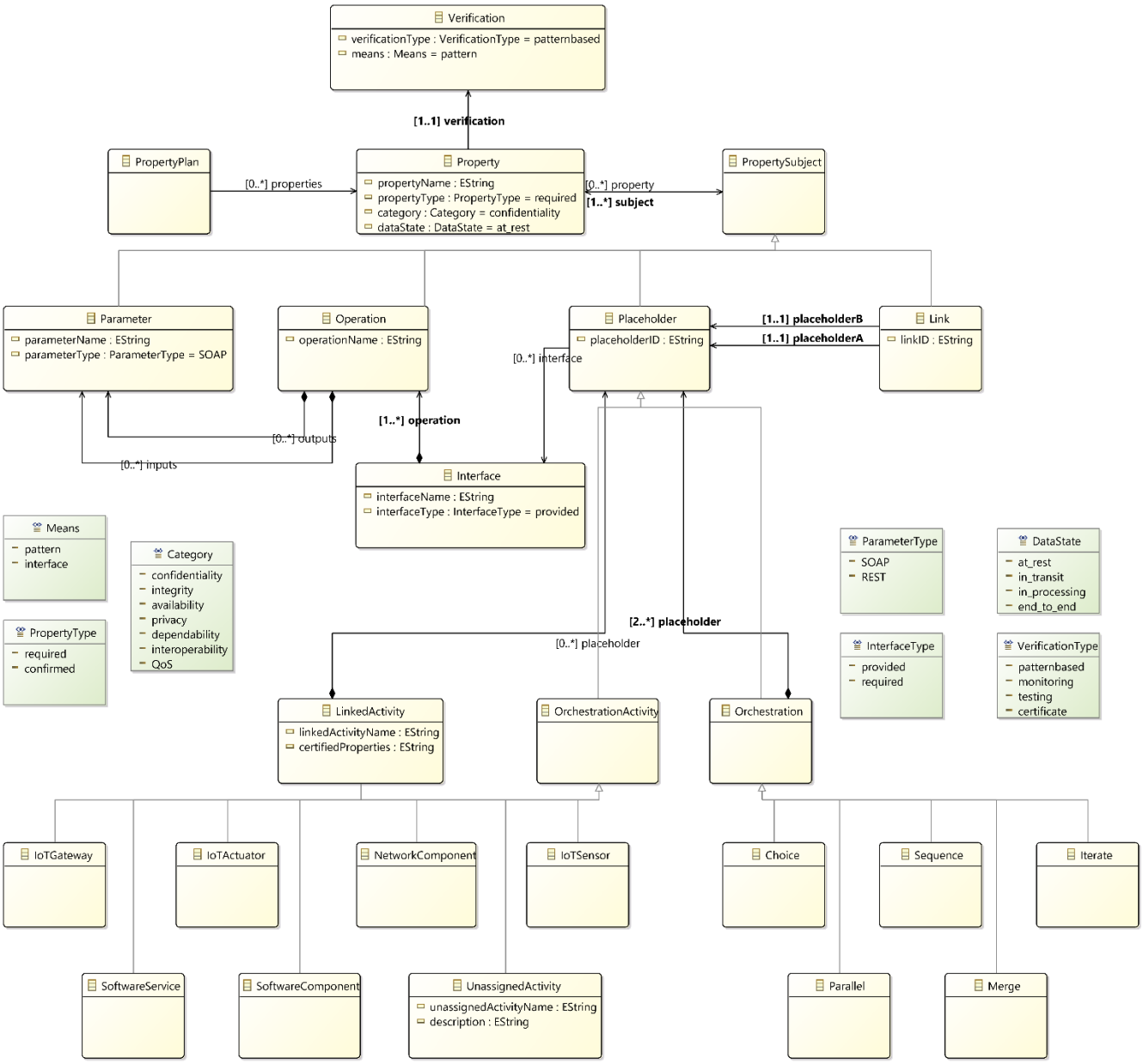


FIGURE 3. SEMIOTICS IOT ORCHESTRATION SYSTEM MODEL

The language for defining IoT application models advocates an orchestration based approach. In this approach, the interactions between the different types of components of such applications (e.g., software components, software services, sensors, actuators) interact with each other as specified as orchestration(s) within the IoT application. Such orchestrations are modelled by the class *Orchestration* in Figure 3. 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 a Sequential, Parallel, Merge, Choice or Iterate orchestration. The meaning of these types of orchestrations is as follows:

(i) Sequence is a segment of a process instance in which several activities are executed in sequence under a single thread of execution.

(ii) Parallel is a segment of a process instance where two or more activity instances are executing in parallel within the workflow, giving rise to multiple threads of control.

(iii) Merge is a point in the workflow where two or more parallel executing/alternative activities converge into a single common thread of control.

(iv) 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.

(v) Iterates is a workflow activity cycle involving the repetitive execution of one (or more) workflow activity(s) until a condition is met.

Moreover, an orchestration involves orchestration activities (see class *OrchestrationActivity* in Figure 3). At any instance of time, these activities may have a known implementation or a not known implementation. In the former case, the activity will be a linked activity (see class *LinkedActivity* in Figure 3). In the latter, the activity will be an unassigned activity (see class *UnassignedActivity* in Figure 3). 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., is a physical device or software program that serves as the connection point between the field devices and the SEMIoTICS backend, via the software-defined network layer, t.
(vii) A (sub) orchestration of IoT application activity implementers of types (i) to (vi).

Software component may also represent external IoT platform services. By adding this class to our model, the description of two different types of platform connectivity, within SEMIoTICS project and across IoT platforms, becomes feasible. In that way we can create patterns that can be used for the verification of SPDI properties in IoT application orchestrations described just within the SEMIoTICS ecosystem and/or across SEMIoTICS and other IoT platform services, such as FI-WARE.

The above types of IoT application activity implementers are grouped under the general concept of placeholder (see the class *Placeholder* in Figure 3). The language introduces also subclasses of the general class *Placeholder* to represent the above elements. These are the classes Orchestration and OrchestrationActivity. As already described Orchestration class above, the OrchestrationActivity class is extended by LinkedActivity and UnassignedActivity classes. Both of these classes have an attribute *Name* to identify them unambiguously. LinkedActivity, referring to activities whose implementation is known, defines the specification of the SDPI properties of the involved activity. On the other hand, UnassignedActivity, referring to a not known implementation, requires a Thing Description, 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. For the exact information that may be included within these Thing Descriptions, please refer to Deliverable D3.3 – “Bootstrapping and interfacing SEMIoTICS field level devices (first draft)”.

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. Interfaces are

represented by the class *Interface* in Figure 3. 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 Figure 3). 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 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 *requires* association end between the class *Placeholder* and *Interface* in Figure 3.

The individual operations that constitute the interface of a placeholder are represented by the class *Operation* in Figure 3. 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 a set of parameters such as the items to be purchased, the number of the credit card and the address for the items to be delivered.

Placeholders (of all different types) may also be characterised by their SPDI and QoS properties. A property of a placeholder is specified according to the class *Property* in Figure 3. According to it, a Property has a *name*, a *type*, a *verification*, a *category* and a *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 in order to be included (considered for) the orchestration. For example, if the required property of an orchestration defining a secure logging process 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. This means that the existence of a monitoring service or a testing tool allows the verification of the SPDI property of a placeholder activity. Such a monitoring service could, for example, justify that a service or a device is available at specific time windows if the desirable property is a specific target for availability. Another way of verifying SPDI properties could be a repository with certificates that are able to justify that a certain placeholder satisfies a certain property. In case of a pattern the *Mean* of verification is the pattern itself; in all the other cases we need an interface to a corresponding monitoring tool, testing service or certificate repository through which the verification can take place.

Moving on with category attribute, the *Category* enumerator in Figure 3, shows the different categories. A Property can belong to confidentiality, integrity, availability, 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 Figure 3). In SEMIoTICS, all three data states are considered, i.e. data in transit, at rest or in processing. If the Placeholder is an Orchestration, then the state of the data will be “in_transit”. If we have to do with an OrchestrationActivity and the OrchestrationActivity is bound to a storage service for example, then dataSate could also be “at_rest”. If the OrchestrationActivity is bound to a service or device that transforms data, then dataSate could be “in_processing”. This attribute was added in the model to allow description of different pattern regarding the three aforementioned data states. This can be done by creating Orchestrations that are subjects to Properties with variant datastates.

Finally, the set of all the SPDI properties that are inferred for the different placeholders of an orchestrator by a pattern are aggregated into PropertyPlan object.

3.3 Language Constructs

Based on the IoT application model presented above we created a corresponding language the constructs of which are described using an EBNF grammar. This language can be used to define activities, as well as basic control flow operations (namely sequential, parallel, choice and merge) enabling their composition into complex orchestrations, and to define the associated individual and composition properties. Upon instantiation of the

orchestration, the abstract definition of the orchestration structure is replaced with the actual components implementing said orchestration. This grammar is presented in Table 6.

In order to create the said language, we used Eclipse's Xtext textual editor, which enables the production of a textual representation of the IoT application model. This textual (Xtext) representation was used as input for an online language converter[5] to produce the equivalent constructs expressed in EBNF[6].

TABLE 6. PATTERN LANGUAGE CONSTRUCTS

Placeholder	::=	Placeholder_Impl | Orchestration_Impl | Sequence | Parallel | Merge | Choice | Iterate | OrchestrationActivity_Impl | LinkedActivity | UnassignedActivity | SoftwareComponent | SoftwareService | NetworkComponent | IoTSensor | IoTActuator | IoTGateway
Property Subject	::=	PropertySubject_Impl | Link | Parameter | Operation | Placeholder_Impl | Orchestration_Impl | Sequence | Parallel | Merge | Choice | Iterate | OrchestrationActivity_Impl | LinkedActivity | UnassignedActivity | SoftwareComponent | SoftwareService | NetworkComponent | IoTSensor | IoTActuator | IoTGateway
Property	::=	'Property' '{' ('propertyName' EString)? ('propertyType' PropertyType)? ('category' Category)? ('dataState' DataState)? 'verification' EString 'subject' '(' EString (',' EString)* ')' '}'
EString	::=	STRING | ID
Interface	::=	'Interface' '{' ('interfaceName' EString)? ('interfaceType' InterfaceType)? 'operation' '{' Operation (',' Operation)* '}' '}'
Placeholder_Impl	::=	'Placeholder' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}
Verification	::=	'Verification' '{' ('verificationType' VerificationType)? ('means' Means)? '}'
PropertySubject_Impl	::=	'PropertySubject' '{' ('property' '(' EString (',' EString)* ')')? '}'
Link	::=	'Link' '{' ('linkID' EString)? ('property' '(' EString (',' EString)* ')')? 'placeholderA' EString 'placeholderB' EString '}'
Parameter	::=	'Parameter' '{' ('parameterName' EString)? ('parameterType' ParameterType)? ('property' '(' EString (',' EString)* ')')? '}'
Operation	::=	'Operation' '{' ('operation Name' EString)? ('property' '(' EString (',' EString)* ')')? ('inputs' '{' Parameter (',' Parameter)* '}')? ('outputs' '{' Parameter (',' Parameter)* '}')? '}'
Orchestration_Impl	::=	'Orchestration' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'
Sequence	::=	'Sequence' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'
Parallel	::=	'Parallel' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'

[5] https://bottlecaps.de/convert/
[6] https://tomassetti.me/ebnf/

Choice	::=	'Choice' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'
Merge	::=	'Merge' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'
Iterate	::=	'Iterate' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? 'placeholder' '{' Placeholder (',' Placeholder)* '}' '}'
OrchestrationActivity_Impl	::=	'OrchestrationActivity' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
LinkedActivity	::=	'LinkedActivity' '{' ('placeholderID' EString)? ('linkedActivityName' EString)? ('certifiedProperties' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? ('placeholder' '{' Placeholder (',' Placeholder)* '}')? '}'
UnassignedActivity	::=	'UnassignedActivity' '{' ('placeholderID' EString)? ('unassignedActivityName' EString)? ('description' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
SoftwareComponent	::=	'SoftwareComponent' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
SoftwareService	::=	'SoftwareService' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
NetworkComponent	::=	'NetworkComponent' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
IoTSensor	::=	'IoTSensor' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
IoTActuator	::=	'IoTActuator' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
IoTGateway	::=	'IoTGateway' '{' ('placeholderID' EString)? ('property' '(' EString (',' EString)* ')')? ('interface' '(' EString (',' EString)* ')')? '}'
PropertyType	::=	'required' | 'confirmed'
Category	::=	'confidentiality' | 'integrity' | 'availability' | 'privacy' | 'dependability' | 'interoperability' | 'QoS'
DataState	::=	'at_rest' | 'in_transit' | 'in_processing' | 'end_to_end'
VerificationType	::=	'patternbased' | 'monitoring' | 'testing' | 'certificate'
Means	::=	'pattern' | 'interface'
ParameterType	::=	'SOAP' | 'REST'
InterfaceType	::=	'provided' | 'required'

3.4 Specification of SPDI patterns

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, which defines the activity-level SPDI properties which are 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, which defines the abstract form of the orchestration that the pattern applies to. As such, the ORCH is specified as an orchestration of abstract activity placeholders. When the pattern is matched against a specific orchestration, the placeholders in its ORCH may be bound to operations of specific nodes or sub-orchestrations of it.
iii. The **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.
iv. The ***Orchestration Properties (OP)*** part, which defines the orchestration-level SPDI properties that the pattern can guarantee 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:

$$AP \wedge ORCH \wedge CONDITIONS \vDash OP,$$

where $\vDash$ denotes the entailment relation that has been established by the proof of the pattern.

APs are materialized using the Property class in Figure 3. 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. This Orchestration can be of different types (Sequential, Parallel, Merge, Choice or Iterate) depending on the order that the involved activities are executed.

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”.

3.5 Example of Orchestration Definition

To showcase the use of the above, let us consider an example of a simple orchestration involving three activities in a sequential composition, as depicted in Figure 4.

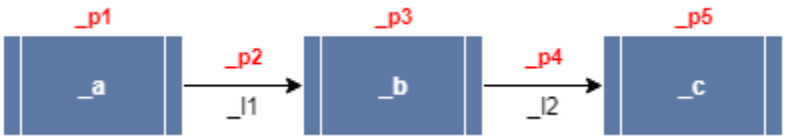


FIGURE 4. A SIMPLE SEQUENTIAL ORCHESTRATION INVOLVING THREE ACTIVITIES

Moreover, the **sequential orchestration pattern** is defined as follows:

0. _ORCH "Seq2"
1. Placeholder (_a, (ActivityName, Description))
2. Placeholder (_b, (ActivityName, Description))
3. Sequence (_a, _b)

4. Link (_l1, _a, _b)
5. Property (_p1, _a, PropertyType, (VerificationType, VerificationMeans), PropertyCategory, DataState)
6. Property (_p2, _l1, PropertyType, (VerificationType, VerificationMeans), PropertyCategory, "in_transit")
7. Property (_p3, _b, PropertyType, (VerificationType, VerificationMeans), PropertyCategory, DataState)
8. Property (_OP, "ConfSeq2", required, (pattern-based, _PR), PropertyCategory, "end_to_end")
9. PatternRule (_PR: _p1, _p2, _p3 → _OP)

In the above, the underscore before the name (e.g. as in "_a") is used to express that these are **placeholders**, which, as mentioned, will be **replaced** with actual activities when the pattern is matched to actual workflows. Line 1 denotes that the **orchestration** type between the three activities is **sequential** (other orchestration patterns are also supported, as mentioned above; e.g. Parallel(_a,_b)). Moreover, the involved activities are further specified within the pattern in lines 2-3, to define specific **parameters** about each, if needed. Furthermore, the **link** between the activities have to be specified, as managing and monitoring the properties of the networking infrastructure is an important aspect of the SEMIoTICS framework and also necessary to guarantee the end-to-end satisfaction of individual properties. Therefore, line 4 defines the links between the involved activities and their type. Lines 5 to 7 define **Activity Properties (AP)**, such as _p1 for placeholder _a and _p2 for _l1, i.e. the link between placeholders _a and _b. Finally, line 8 includes an **Orchestration Property (OP)** that can be guaranteed as long as the Activity Properties _p1 to _p3 hold, as defined in pattern rule _PR (defined in line 9).

The sequential orchestration pattern of Figure 4, which involves three activities, can be defined via the "Seq2" orchestration pattern as follows:

$$\text{\_ORCH "Seq3" : Sequence (\_a, \_b, \_c)} == \text{\_ORCH "Seq2" : Sequence (Sequence (\_a, \_b), \_c)}$$

Therefore, activities _a and _b are composed into a single (complex) activity by applying the sequential orchestration pattern "Seq2", and the resulting activity forms the first part of the next application of "Seq2", with _c as the second term in it.

When instantiating the above template, the placeholders are substituted by specific activities (or orchestrations of activities, which can be considered as sub-orchestrations), and also the properties become specific. Continuing with the same example above, a specific instance of the orchestration template can be seen in Figure 5 below, whereby placeholders _a, _b and _c are instantiated with activities "A1", "A2" and "A3", respectively.

FIGURE 5. INSTANCE OF THE SEQ3 ORCHESTRATION DEPICTED IN FIGURE 4

Thus, the description of the instance of the orchestration of Figure 4, as shown in Figure 5, can be defined by replacing placeholder _a with activity "A1" (e.g. , _b with activity "A2" and _c with activity "A3". Then the individual descriptions become specific, such as:

- Placeholder (A1, (PaymentActivity, PaymentDescription))
- Placeholder (A2, (PlaceOrderActivity, PlaceOrderDescription))
- Placeholder (A3, (WriteReportActivity, WriteReportDescription))
- Link (L1, A1, A2)
- Link (L2, A2, A3)

The same goes for the individual and orchestration-wide properties; examples include:

- Property (P1, A1, required, (monitoring, interface), confidentiality, in_processing)
- Property (P2, L1, required, (monitoring, interface), confidentiality, in_transit)
- Property (P3, A2, required, (pattern-based, PSP), confidentiality, in_processing)
- Property (P4, L2, required, (monitoring, interface), confidentiality, in_transit)
- Property (P5, A3, required, (certificate, interface), confidentiality, at_rest)
- Property (OP, "Seq2", required, (pattern-based, PR1), confidentiality, "end_to_end")

In the above instantiation example, we assume that the end-to-end confidentiality is pursued for the instantiated orchestration, and therefore individual component confidentiality properties need to be verified. Specific details about the individual properties are included in the instantiated form of the orchestration; e.g., property P1 (see line 7) refers to confidentiality of data in processing at activity A1, and this is verified through monitoring of a specific interface. Similarly, P3 refers to the confidentiality in processing at activity A2, but in this case the verification is pattern-based, and more specifically via pattern *PSP* (more details on the *PSP* property can be found in section 4.1.1).

The verification takes place by iteratively applying the Sequential composition pattern for two activities ("A1" and "A2") and then again for the derived complex activity and "A3", as previously defined for the generic example.

3.6 Implementation aspects

3.6.1 MACHINE-PROCESSABLE PATTERN ENCODING

An important requirement for implementing the SPDI pattern-driven management and adaptation of the SEMIoTICS infrastructure is to support the automated processing of developed patterns. To achieve this, the SEMIoTICS SPDI patterns will be expressed as Drools business production rules, and the associated rule engine, by applying and extending the Rete algorithm [27]. 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.

In more detail, the language constructs depicted in Table 3 above will be represented as Java classes in a Drools project for loading and executing Drools rules. It is decided, SPDI patterns to be expressed as Drools production rules to take advantage of the associated rule engine that comes with Drools rules for automated processing of the patterns.

A Drools production rule has the following generic structure:

```
rule name <attributes>*
when <conditional element>* then <action>* end
```

The **when** part of the rule specifies a set of conditions and the **then** part of the rule a list of actions. When a rule is applied, the Drools rule engine checks whether the rule conditions (defined within the `<conditional element>` above) match with the facts in the *Drools Knowledge Base (KB)* and if they do, it executes the actions (i.e. "`<action>`") of the rule. Rule actions are typically used to modify the KB by inserting, retracting or updating the objects (*facts*) in it, through the standard Drools actions *"insert", "retract"* and *"update"*,

respectively. The conditions of a rule are expressed as patterns of objects that encode the facts in the Drools KB. These patterns define object types and constraints for the data encoded in objects which may be atomic or complex. Complex Drool object constraints are defined through logical operators (e.g. `and, or, not, exists, forall, contains`). The full grammar of the current version of the Drools rule language (version 7.16.0 as of writing this deliverable) can be found online[7], while an overview of the main specification constructs is provided in Table 7 to allow the reader to follow the pattern specifications provided within the work presented herein.

TABLE 7. HIGH LEVEL DROOLS RULE SPECIFICATION CONSTRUCTS

Type	Construct	Description
Conditional element	`and-CE | or-CE | not-CE | exists-CE | forall-CE | contains-CE | from-CE | collect-CE | accumulate-CE | eval-CE`	Conditional elements are used to specify conditions in the *when* part of a rule and in constraint expressions (see Pattern construct below). Conditional elements realise basic logical operators (e.g. *and*, *or*, *not*); quantified logic operators (*contains*, *forall* and *exists*); and object collection operators (e.g. *collect*, *accumulate*).
Pattern	`Top level syntax:` `Pattern: <pattern-Binding ":" > PatternType "(" Constraints ")"`	Patterns are matched with elements in the working memory. The pattern binding is typically a variable and the pattern type refers to declared object types that could be matched with the pattern. Constraints are specified by logical expressions. Such expressions can be constructed by logic conditional elements (see above); object collection elements; unification operators; relational; arithmetic; property/list access operators; data accumulation functions; regular expression matching operators, and; temporal operators.
Action	`Modify | Update | Insert | Retract`	Pattern-related actions include *Modify* to modify the contents of a fact, Update a face, *Insert* to insert new fact in the KB and *Retract* to delete a fact.

As mentioned, Drools are used in SEMIoTICS 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 matching between Drool rules and patterns happens as follows:

- The **`when`** part 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 $ORCH \wedge Conditions \wedge OP \Rightarrow AP_i\ (i = 1, \dots, n)$, 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 defined above (namely $AP \wedge ORCH \wedge CONDITIONS \vDash OP$). 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.

In Section 4 a first set of patterns is defined and for each, the corresponding representation in Drools is also given.

[7] http://docs.jboss.org/drools/release/7.16.0.Final/drools-docs/html_single/

3.6.2 SYSTEM ARCHITECTURE AND KEY COMPONENTS

The implementation of the IoT/IIoT service orchestrations as well as the SPDI approach in SEMIoTICS relies on the presence of some key components in the framework's architecture; these are detailed in the subsections below.

3.6.2.1 ORCHESTRATION-ENABLING COMPONENTS

The key SEMIoTICS architectural components that handle data during workflow executions are described below:

- **Backend**
  - **Recipe Cooker:** Module responsible for cooking (creating) recipes providing high level definitions of service workflows
  - **Backend Semantic Validator**: Module responsible for providing semantic validation and translation between different semantic models found in IoT environments.
  - **Security Module:** Module responsible for granting access and necessary security checks at the backend layer
  - **IoT Platforms:** Different IoT platforms that SEMIoTICS is interfaced with, such as FIWARE and MindSphere.
  - **Use case-specific Apps:** The various backend applications pertinent to the specific use cases (e.g. industrial applications for UC1, patient monitoring applications for UC2, and node management applications for UC3)
  - **Web Services:** Private and public cloud monolithic services that are part of the running workflows.
- **Network layer**
  - **Network Service Functions:** The different network service functions (e.g. load balancing, firewall) running on the VIM.
  - **Switches**: The switches that form the underlying network, including both hardware and virtual programmable devices.
  - **SDN Controller**: the controller(s) of the software defined network infrastructure
- **Field Layer**
  - **IoT Gateway:** The IoT gateway, including its various modules, such as the semantics mediator, semantic API etc.
  - **Field devices:** The different field devices present in the SEMIoTICS deployments, such as the various industrial and healthcare sensors and actuators, as well as their counterparts with increased analytics capabilities defined in the context of UC3.

More specifically, as described above and can be seen in Figure 6, the components that handle data during workflow executions are dispersed over the three different layers of SEMIoTICS architecture. The arrows of the figure define the basic components that are involved in the data flows. We may consider as an example of a data flow the following order of participated functional components:

1. Sensing Data are received by Sensors (or actuation commands to Actuators)
2. Processed transported through the Use Case Specific devices
3. Transformed and evaluated by the GW Semantic Mediator
4. Semantic integration into IoT semantic models by the Semantic API & Protocol Mediator
5. Forwarded traffic through the programmable by the SDN SEMIoTICS controller Switches
6. Forwarded traffic through the predefined service function chains if needed
7. Processed traffic is forwarded through the backend related components (i.e. web/cloud service)
8. Forwarded to the MindSphere Apps in case of Use Case 1 or through the FIWARE in Use Case 2
9. And finally, the use case apps are responsible to process the data

The same procedure can be followed starting from the step 9 and going to step 1 in case of an actuation command.

More details about the individual components can be found in the corresponding architecture deliverable, i.e. Deliverable D2.4 - SEMIoTICS high level architecture (Cycle 1), where the first version of the SEMIoTICS architecture and the included components are detailed. The pattern-specific modules in the architecture and some initial sequence diagrams of their operation can be found in the subsection below.

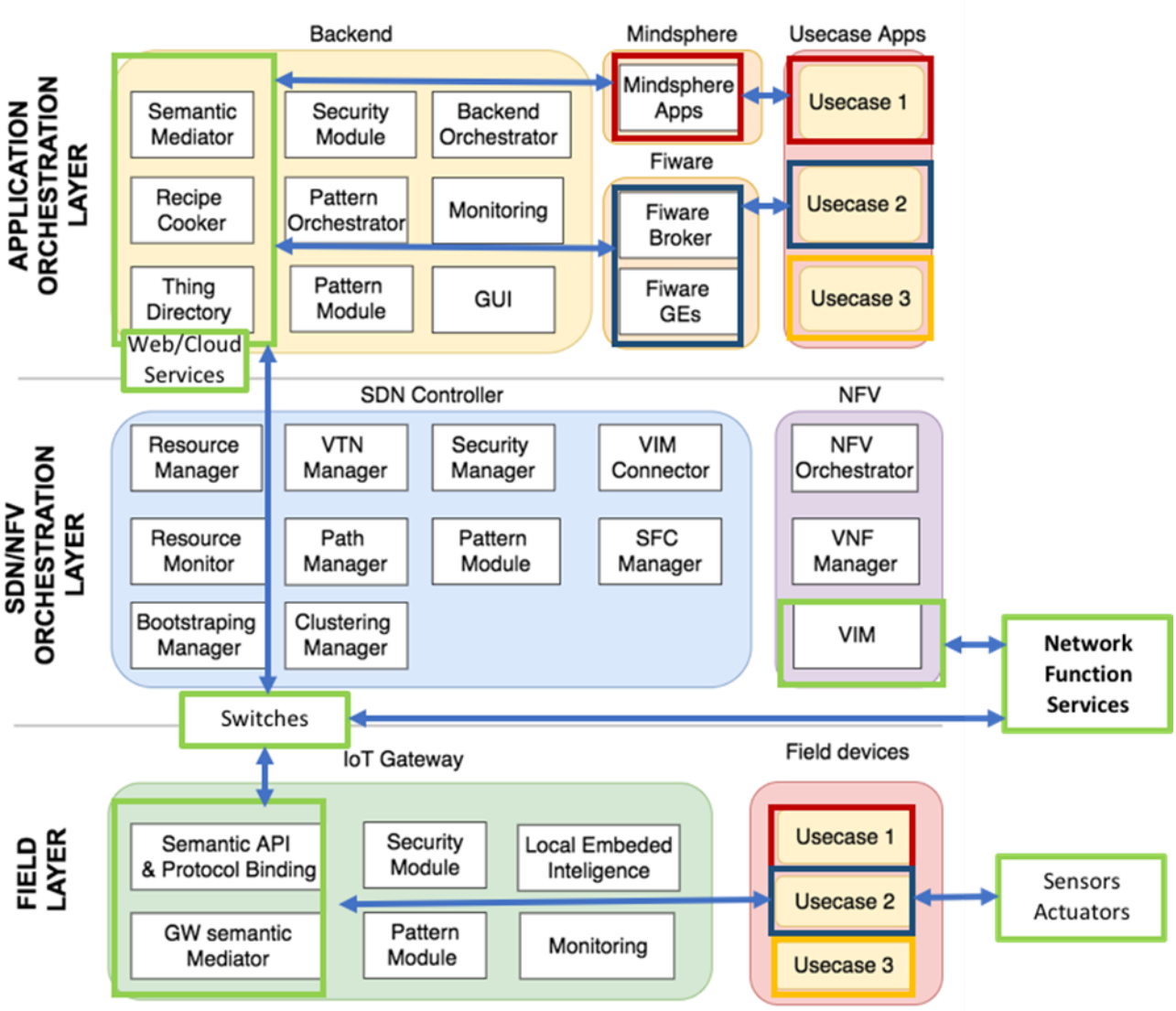


FIGURE 6. SEMIOTICS ARCHITECTURE

3.6.2.2 PATTERN COMPONENTS

In addition to the components of the SEMIoTICS architecture enabling the implementation of IoT service orchestrations (as defined in the previous subsection), pattern-related components are present in all layers of the SEMIoTICS framework (see Figure 7), in line and towards realising the SEMIoTICS vision of **embedded intelligence across all layers** of the IoT deployment.

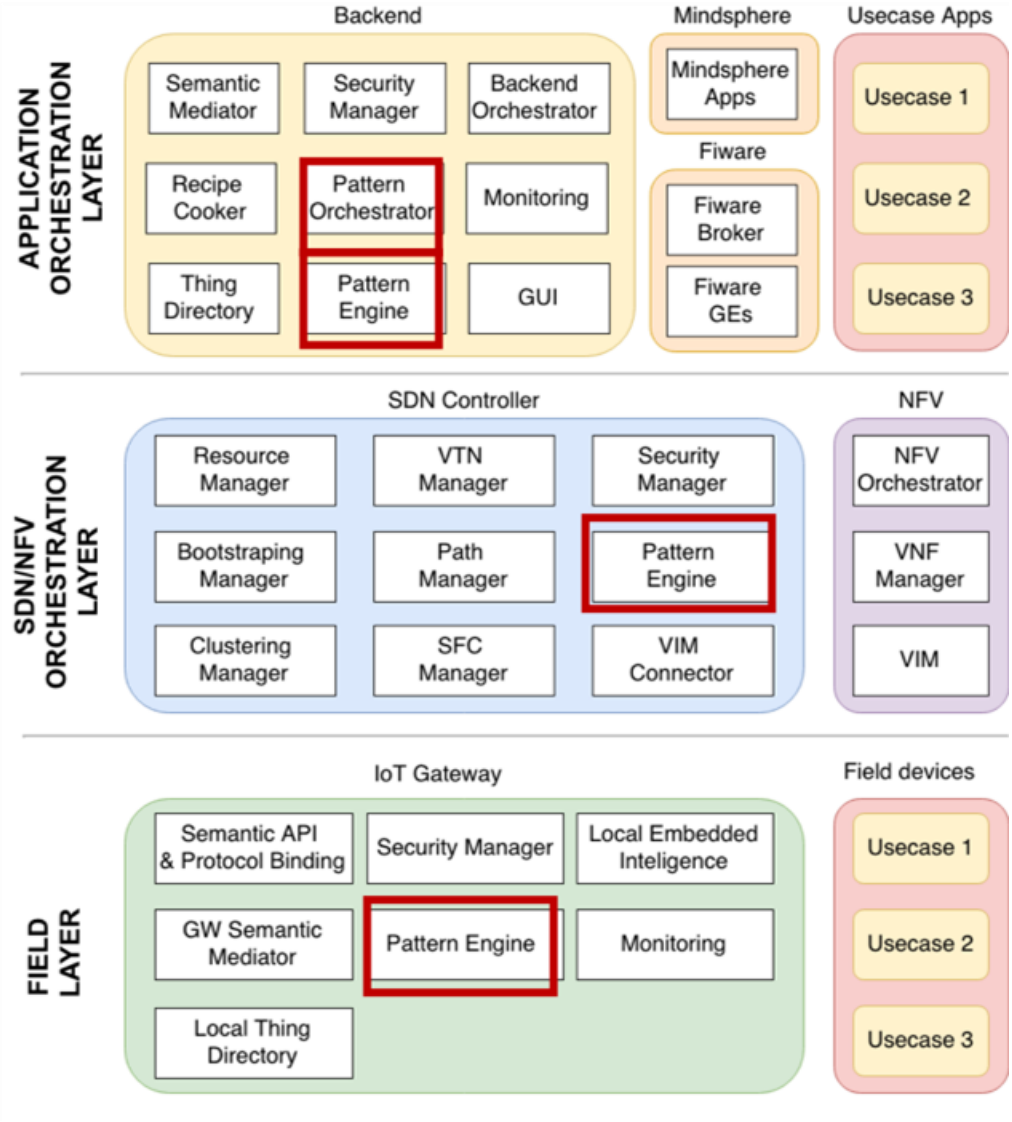


FIGURE 7. PATTERN MODULES WITHIN THE SEMIOTICS ARCHITECTURE

In more detail, these components are:

- **(Backend) Pattern Orchestrator:** Module featuring an underlying semantic reasoner able to understand instantiated Recipes, as received from the Recipe Cooker module and transform them into composition structures (orchestrations) to be used by architectural patterns to guarantee the required properties. The Pattern Orchestrator is then responsible to pass said patterns to the corresponding Pattern Modules (as defined in the Backend, Network and Field layers), selecting for each of them the subset of these that refer to components under their control (e.g. passing Network-specific patterns to the Pattern Module present in the SDN controller).
    - Regarding the current implementation status, the first version of Pattern Orchestrator is created using Java, ANTLR and Maven. A first set of Java classes has been created corresponding to main components of the IoT orchestration system model. Moreover, a first version of ANTLR parser recognises the given orchestration components and gRPC Protocol buffers are planned

to be used for the communication between the Pattern Orchestrator and the other Pattern-related components.

- **Backend Pattern Module:** Features the pattern engine for the SEMIoTICS backend, along with associated subcomponents (knowledge base, reasoning engine). It will enable the capability to insert, modify, execute and retract patterns at design or at runtime in the backend; these interactions will happen through the interfacing with the Pattern Orchestrator (see above), though additional interfaces may be introduced to allow for more flexible deployment and adjustments if needed. Will be able reason on the SPDI properties of aspects pertaining to the operation of the SEMIoTICS backend. Moreover, at runtime the backend Pattern module may receive fact updates from the individual Pattern Modules present at the lower layers (Network & Field), allowing it to have an up-to-date view of the SPDI state of said layers and the corresponding components.
    - Regarding the current implementation status, a first version of the Backend Pattern Module has been created as a Maven project, using Java. In this first version, the installed Drools Rules Engine will be used for the automated processing of SPDI patterns expressed as Drools rules. The gRPC Protocol buffers are planned to be used for the communication between the Backend Pattern Module and the other Pattern-related components.
- **Network Pattern Module:** Integrated in the SDN controller to enable the capability to insert, modify, execute and retract network-level patterns at design or at runtime. It will be supported by the integration of all required dependencies within the network controller, as well as the interfaces allowing entities that interact with the controller to be managed based on SPDI patterns at design and at runtime. It will feature different subcomponents as required by the rule engine, such as the knowledge base, the core engine and the compiler.
    - Regarding the current implementation status, the Drools Rules Engine dependencies have been included in the Maven project for the processing of Drools rules. The communication towards Network Pattern Module can be achieved using exposed NBIs of the SDN controller. These NBIs are REST RPCs that are defined utilizing the YANG model.
- **Field Layer Pattern Module:** Typically deployed on the IoT/IIoT gateway, able to host design patterns as provided by the Pattern Orchestrator. Since the compute capabilities of the gateway can be limited, the module will be able to host patterns in an executable form compared to the pattern rules as provided in the other layers. The executable patterns will be able to guarantee SPDI properties locally based on the data retrieved and processed by the monitoring module, the thing directory in the IoT gateway and based on the interaction as well with other components in the field layer. Will store pattern executables in a local knowledge base that will be updated by the pattern orchestrator as needed and requested.
    - In terms of implementation, the Field Layer Pattern Module is a lightweight version of the Backend Pattern Module. So the first version of the Field Layer Pattern Module will be based on the current version of the Backend Pattern Module. Technologies that will be used include Java, Maven, Drools Engine, gRPC Protocol Buffers.

3.7 Language Interpretation and Instantiation

Regarding the language interpretation, the EBNF grammar is used as input to an ANTLR4 lexer, parser and listener. These programs manage to create for every orchestration activity, control flow operation and property a Drools fact, i.e. an instance of the corresponding Java class. The Drools facts are then inserted in the Knowledge base of Drools, a repository of all the application's knowledge definitions, in the three Pattern Engines of SEMIoTICS. Sessions are created from the KnowledgeBase in which data can be inserted and process instances started. A knowledge session is the way to interact with Drools and the core component to fire Drools rules. Rules themselves are also hold in a knowledge session. The information that is stored in the KnowledgeBase is used for reasoning.

Figure 8 shows a simple orchestration along with its description using the IoT application language. As we can see, the orchestration consists of two Placeholders, Camera and ObjectDetector, and a Link between them, named L1. Moreover, they are in sequence (Sequence1), which means that the output of the former is consumed as input by the latter.

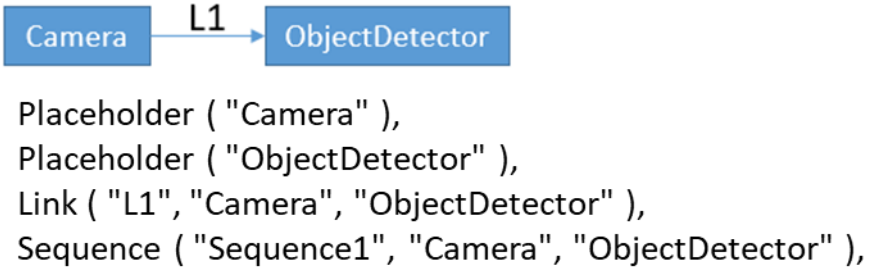


FIGURE 8: SIMPLE ORCHESTRATION EXAMPLE

During the first step of the translation of an IoT application orchestration to Drools facts the ANTLR4 lexer recognizes keywords and transforms them in tokens. The created tokens are used by the ANTLR4 parser for creating the logical structure, i.e. the parse tree.

Next, the ANTLR4 listener allows us to communicate with Drools every time a node in the parse tree is entered. The listener takes information from the tokens and sends it to Drools. Drools then creates instances from the corresponding Java classes and stores the received information at the class attributes.

During the last step, the created java instances are inserted as facts into the knowledge session. These Drools facts are used by Drools rules, which are fired when a condition is met.

3.8 Language Expressiveness and Versioning

A key design choice early on in the project was to implement a language tailored to the intrinsic requirements and characteristics of IoT environments. This was the result of considering the complex set of requirements of the SEMIoTICS pattern language (see Section 2) and the gap analysis carried out on the existing SoTA approaches (see subsection 3.1.1), and which led to definition of the elaborate system model presented in subsection 3.2. The rationale and methodology behind the definition of said model, which is the source of the associated language (see subsection 3.3), as well as the anticipation in the work programme that said language will evolve throughout the runtime of the project (thus the provision of two deliverables with first and final version of the language), eventually covering all use cases and a full set of patterns covering all SPDI properties and data state and connectivity options, provides significant guarantees that the end result (i.e., final version of the language) will provide all the needed expressive means to fully cover for the needs of the project and the covered IoT environments.

Nevertheless, it is foreseen that in order to address additional domains (e.g., smart vehicles, smart agriculture), the model will have to be extended to cover the devices and interactions intrinsic to each of the targeted domains. This is not an obstacle and is supported by the (by design) extensible approach followed: the system model is by design extensible and it is trivial to define additional classes and interactions. Following the same techniques presented in the language definition process above, extensions to the model are transferred to the language, enabling it to support additional expressive constructs, as needed. Thus, the language itself is also volatile and adding more concepts to newer versions of the language can be done easily.

In terms of versioning, and while all typical file and software versioning tools can be used for that purpose, care has to be taken when introducing new classes in the definition of relationships among old ones. Changing any part of the older version and/or the relationships between the old classes can break backward compatibility with previous versions of the language (and, thus, the associated reasoning). Nevertheless, even in cases where this is needed, the only additional measure that the system owners have to take is to re-define the older orchestrations and ensure that the reasoning engines at the different layers are updated with the new rules and facts.

4 PATTERN RULES

This section presents the first set of SEMIoTICS pattern rules, using the language and associated constructs defined in the previous section. The Security properties of Confidentiality, Integrity and Availability are analysed separately in the corresponding subsections below, as different types of property reasoning and monitoring conditions need to be defined for each one of them.

4.1 Security

4.1.1 CONFIDENTIALITY

4.1.1.1 PATTERN DEFINITION

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. Formal definitions of Confidentiality are typically based on the concept of Information Flow (IF) [28], separating users in classes with different access rights to the system's information and distinguishing the information flows within the system according the user classes they should be accessible to. Based on this concept, the *Perfect Security Property (PSP) [29]* 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.

A *sequential orchestration* ***P*** with two activity placeholders, ***A*** and ***B***, whereby *B* is executed after *A*, is depicted in Figure 9. We assume 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.

Further conditions that define P, as depicted in Figure 9, include:

- The inputs of A are the inputs of the workflow P
- The inputs of B are the outputs of A
- The outputs of the orchestration P are the outputs of B

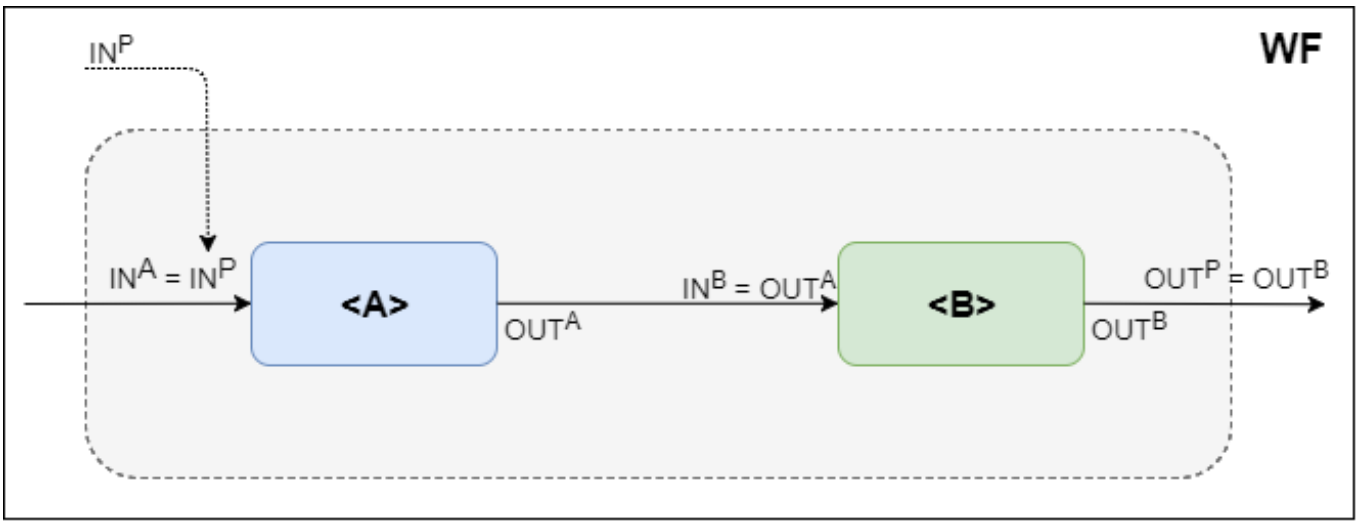


FIGURE 9. PSP ON A SEQUENTIAL SERVICE ORCHESTRATION

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

i. NP:

a. $PSP(A,\ V^A,\ C^A)\ and\ V^A \subseteq V^P\ and\ C^A \cap V^P = \emptyset$
b. $PSP(B,\ V^B,\ C^B)\ and\ V^B \subseteq V^P\ and\ C^B \cap V^P = \emptyset$

ii. OP:

a. $SecReq^P = PSP(P, V^P, C^P)$

Interpreting the pattern above, and as proven in [30], PSP then holds on the orchestration ***P*** if, for all activity placeholders *x* in *{A, B}*, the following are true:

a) $V^X \subseteq V^P$; i.e. the actions of *x* that reveal public information are part of the actions of P that reveal public information are part of the actions of P that reveal public information, and
b) $C^X \cap V^P = \emptyset$; 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 *NP* properties of the pattern and entail the PSP property on P, as expressed in the *OP* part of the pattern.

Regarding Data State Coverage, this pattern covers all three states, in_transit, at_rest and in_processing. Moreover, it refers to components that are within the SEMIoTICS platform.

4.1.1.2 PATTERN SPECIFICATION RULE

Based on the above, the confidentiality (PSP) pattern defined in subsection 0 can be represented in Drools as shown in Table 7.

The `when` part of the rule specifies: the two activity placeholders A and B of the PSP pattern (variables `$A` and `$B` on lines 3-4 and 5-6); the order in which `$A` and `$B` are executed (variable `$ORCH` on line 7) and the conditions between the outputs of `$A`, and the inputs of `$B` as required by the PSP pattern (lines 7-9), and; the OP property that can be guaranteed by applying the pattern, i.e. the PSP property in this case (variable `$WP` in lines 10-11). Lines 3-9 are the specification of the ORCH part of the pattern.

The `then` part of the rule generates a security plan that includes the NP security properties that (if satisfied by the activity placeholders that will be selected for the pattern's ORCH) would lead to a ORCH satisfying the OP (i.e. the PSP property). Based on the proof of the PSP property detailed earlier in this document, both of the placeholders A and B should satisfy the PSP property; thus, PSP is defined as the NP property that both placeholders should satisfy in lines 17 and 22, respectively. Moreover, the additional conditions defined earlier (i.e. $V^A \subseteq V^P \; and \; C^A \cap V^P = \emptyset$ for placeholder A and $V^B \subseteq V^P \; and \; C^B \cap V^P = \emptyset$ for B are also added to the corresponding NPs, as can be seen in lines 18-19 and 23-24, respectively.

TABLE 8. 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 = $OP.getAttributesMap().get("V");
  Property NP_A = new Property($OP, "PSP", $A);
  NP_A.getAttributesMap().put("V", new Operation("subset", V_P));
  NP_A.getAttributesMap().put("C", new Operation("subset", new
Operation("complement",V_P)));
  newPropertyPlan.getProperty().add(NP_A);
  insert(NP_A);
  Property NP_B = new Property($OP, "PSP", $B);
  NP_B.getAttributesMap().put("V", new Operation("subset", V_P));
  NP_B.getAttributesMap().put("C", new Operation("subset", new
Operation("complement",V_P)));
  newPropertyPlan.getProperties().add(NP_B);
  insert(NP_B);
  insert(newPropertyPlan);
end
```

4.1.2 INTEGRITY

4.1.2.1 PATTERN DEFINITION

Data Integrity refers to the maintenance and assurance of the accuracy and consistency of data. Following the definition in 4.1.1 A *sequential orchestration* ***P*** with two activity placeholders, ***A*** and ***B***, whereby *B* is executed after *A*, is depicted in Figure 9. We assume 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*
- $D^x(i)$ the data of x at the given time t
- Hash(i) are the cryptographic hash function result applied to data i

Further conditions that define P, as depicted in Figure 9, include:

- 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, a pattern for preserving integrity for data that are at in processing and in transit on the service orchestration P can be defined as follows:

- $\mathrm{Hash}(\mathrm{IN}^{\mathrm{P}})=\mathrm{Hash}(\mathrm{IN}^{\mathrm{A}})$

- $Hash(OUT^{P})=Hash(OUT^{B})$
- $Hash(IN^{B})=Hash(OUT^{A})$

Interpreting the pattern above, we have for every data that is transmitted not only through datalinks but also through inter process communication to evaluate that the data that an activity A sends to activity B are not by any chance changed.

Moreover, based on the above specification we can define a generic pattern for integrity at data at rest as the following:

$$Hash(D^{x}(i))=Hash(D^{x}(i-1))$$

Which means that whenever we check data at rest those data must not be changed.

Regarding Data State Coverage, this pattern covers all three states, in_transit, at_rest and in_processing. Moreover, it can be used for verifying integrity property of components that are within the SEMIoTICS platform.

4.1.2.2 PATTERN SPECIFICATION RULE

The specification rule of the above patterns in Drools is shown in Table 8 and Table 9, respectively. Specifically, for the Integrity At Rest rule (Table 9), it specifies the data of the activity that are we check at line 3. Then in line 4 we define a special activity that becomes true every n seconds and forces the drools engine to run the then part of the rule. At line 12 we calculate the hash checksum of the data and we retrieve the checksum that it is already stored and those must be true since the data are at rest.

TABLE 9. SPECIFICATION OF INTEGRITY PROPERTY VIA DROOLS

```
1.  rule "Integrity"
2.  when
3.    $A: Placeholder($input : operation.inputs,
4.      $intData : parameters.outputs)
5.    $B: Placeholder(parameters.inputs == $intData,
6.      $output : parameters.outputs)
7.    $ORCH: Link(firstActivity == $A, secondActivity == $B)
8.    $OP: Req( propertyName == "Integrity",
9.      subject == $ORCH, satisfied == false)
10.   $SP: PropertyPlan (properties contains $OP)
11. then
12.   PropertyPlan newPropertyPlan = new PropertyPlan($SP);
13.   newPropertyPlan.removeRequirement($OP);
14.   Req Hash1 = new Req($OP,
    "equality",sha512($A.input),sha512(operation.input));
15.   newPropertyPlan.getProperties().add(Hash1);
16.   insert(Hash1);
17.   Req Hash2 = new Req($OP,
    "equality",sha512($A.output),sha512($B.inputs));
18.   newPropertyPlan.getProperties().add(Hash2);
19.   insert(Hash2);
20.   Req Hash3 = new Req($OP,
    "equality",sha512($B.output),sha512(operation.inputs));
21.   newPropertyPlan.getProperties().add(Hash3);
22.   insert(Hash3);
23.   insert(newPropertyPlan);
24. end
```

TABLE 10. SPECIFICATION OF INTEGRITY AT REST PROPERTY VIA DROOLS

```
rule "IntegrityAtRest"
when
  $A: Placeholder($intData : datastore.Data)
  $T: Timer(time.Interval(“Default time interval”))
  $ORCH: Check(firstActivity == $A, secondActivity == $T)
  $OP: Req( propertyName == "Integrity",
    subject == $ORCH, satisfied == false)
  $SP: PropertyPlan (properties contains $OP)
then
  PropertyPlan newPropertyPlan = new PropertyPlan($SP);
  newPropertyPlan.removeRequirement($OP);
  Req Hash1 = new Req($OP,
 "equality",sha512(&intData),datastore.StoredHash($A));
  newPropertyPlan.getProperties().add(Hash1);
  insert(Hash1);
  insert(newPropertyPlan);
end
```

4.1.3 AVAILABILITY

4.1.3.1 PATTERN DEFINITION

According to [31], availability is defined as “readiness for correct system service”; a service is deemed to be correct if it implements the specified system function. Readiness of a system in this definition means that if some agent invokes an operation to access some information or use a resource, it will eventually receive a correct response to the request.

Regarding Data State Coverage, this pattern covers two states, at_rest and in_processing. Moreover, it refers to both platform connectivity cases, among components that are within the SEMIoTICS platform or across IoT platforms.

4.1.3.2 PATTERN SPECIFICATION RULE

The specification of the Availability pattern in Drools is provided in Table 10 below. The rule checks at default time intervals that the response of the Activity is within the predefined time constrains.

TABLE 11. SPECIFICATION OF AVAILABILITY PROPERTY VIA DROOLS

```
1.  rule "Availability"
2.  when
3.    $A: Placeholder($input : operation.inputs,
4.     output : parameters.outputs)
5.    $T: Timer(time.Interval(“Default time interval”))
6.    $ORCH: Check($A,$T)
7.    $OP: Req( propertyName == "Availability", subject == $ORCH,
    satisfied == false)
8.    $SP: PropertyPlan (properties contains $OP)
9.  then
10.   PropertyPlan newPropertyPlan = new PropertyPlan($SP);
11.   newPropertyPlan.removeRequirement($OP);
12.   Req Hash1 = new Req($OP,"ResponseTime",$A, ”Default response
    time”);
13.   newPropertyPlan.getProperties().add(Hash1);
14.   insert(Hash1);
15.   insert(newPropertyPlan);
16. end
```

4.2 Privacy

4.2.1 PATTERN DEFINITION

4.2.1.1 CONSENT

Due to GDPR constrains, patterns should be developed in order for SEMIoTICS to be GDPR compliant. One of the constrains that need to be considered is for the user to give her consent on their data to be used.

On a simple service composition as the below depicted on Figure 10 we make the following assumptions

- IN^A and OUT^A are the sets of inputs and outputs of *A*
- D^x Are the data which belong to owner X
- C is a set of users who have agreed their data can be processed and stored

FIGURE 10 PRIVACY ON A SIMPLE SERVICE COMPOSITION

Then in order to be able to able to create every service composition the following pattern should be applied

$$\mathrm{IN}^{\mathrm{P}} = \mathrm{D}^{\mathrm{A}} \text{ where } \mathrm{A} \subseteq \mathrm{C}$$

This means that for every service composition we must first check that the beholder of the data has agreed to this composition can process them.

Regarding data state coverage, this pattern covers two states, at_rest and in_processing. Moreover, it refers to components that are within the SEMIoTICS platform.

4.2.1.2 IDENTIFIABILITY

In order to guarantee privacy not only components that form the service should be checked for privacy but also their composition. At each layer of composition, the data union that the layer produces should be evaluated. As an example, consider the composition of a service of two components.

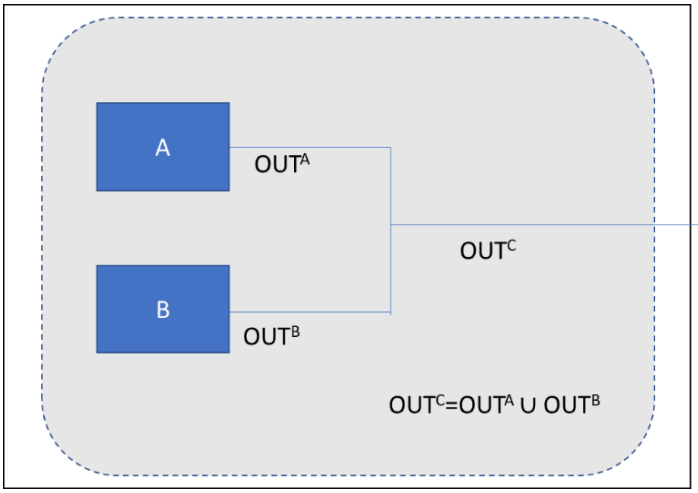


FIGURE 11. PRIVACY PATTERN EXAMPLE

Let us assume that for each *x* in *{A, B, C}*

- $\mathrm{OUT}^{\mathrm{X}}$ are the sets of outputs of x
- IN^{X} are the sets of inputs of x
- $\mathrm{E}^{\mathrm{X}}=\mathrm{IN}^{\mathrm{X}} \cup \mathrm{OUT}^{\mathrm{X}}$
- V^{X} and C^{X} are two disjoint subsets of E^{X} which partition it into public parts V^{X} and confidential parts C^{X}
- L is a corpus of sets that are pre-defined that expose privacy

Then in order the composition to satisfy the privacy requirements, the following properties must hold:

a. $\mathrm{V}^{\mathrm{A}} \cap \mathrm{L} = \varnothing$
b. $\mathrm{V}^{\mathrm{B}} \cap \mathrm{L} = \varnothing$
c. $\mathrm{V}^{\mathrm{C}} \cap \mathrm{L} = \varnothing$

Moreover, when data are at rest (i.e. in storage) we should take precautions that:

d. $(\mathrm{V}^{\mathrm{A}} \cup \mathrm{V}^{\mathrm{B}} \cup \mathrm{V}^{\mathrm{C}}) \cap \mathrm{L} = \varnothing$

Still, the following properties should also hold:

e. $(\mathrm{V}^{\mathrm{A}} \cup \mathrm{V}^{\mathrm{B}}) \subseteq \mathrm{V}^{\mathrm{C}}$
f. $(\mathrm{V}^{\mathrm{A}} \cup \mathrm{V}^{\mathrm{B}}) \cap \mathrm{C}^{\mathrm{C}} = \varnothing$

As an example, let us assume that there are two components *A* and *B* that we want to use to create a service *C*. Moreover, a set *L* that exposes users privacy is *L={(name, location), (name, medical_condition)};* i.e., we do not want a service that exposes a person's name along with her location and/or her medical conditions.

Component *A* publicly sends the user's ID, environmental temperature and location, while component *B* publicly sends the user's name, user's ID and the humidity of the environment

In this case, *Req(A, Privacy)* is validated as True (since $OUT^A \cap L = \varnothing$), and also *Req(B, Privacy)* is validated as True (since $OUT^B \cap L = \varnothing$).

Nevertheless, the composition of *A* and *B* to form *C,* as in **FIGURE 11**, creates:

$$OUT^C = OUT^A \cup OUT^B = \{userID, temperature, location, UserName, humidity\}$$

This means that $OUT^C \cap L = \{name, location\}$, which is not empty; thus, the composition of those 2 services is not viable, as it violates the privacy pattern rule and creates a privacy leak.

Regarding data state coverage, this pattern covers two states, at_rest and in_processing. Moreover, it refers to components that are within the SEMIoTICS platform.

4.2.2 PATTERN SPECIFICATION RULE

4.2.2.1 CONSENT

Based on the approach presented in subsection 3.3, a representation of our pattern in our pattern language can be defined as:

0. **ORCH "Consent"**
1. Activity(_a)
2. AP_1("UserConsensus",_a, pattern)
3. OP(GDPR_Consensus, subject == "Consent", satisfied == false)
4. **Pattern rule: AP_1 ➔ OP**

1. **ORCH "Consent"**
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Property (UserConsensus, A1, required, (pattern-based, pattern), ?, at_rest)
4. Property (GDPRConsensus, "Consent", required, (pattern-based, PR1), ?, end_to_end)
5. **Pattern rule (PR1: UserConsensus -> GDPRConsensus**

This pattern can be then translated to a drools engine compatible pattern as the following:

TABLE 12. SPECIFICATION OF GDPR CONSENT VIA DROOLS

```
rule "Consent"
when
  $A: Placeholder($input : operation.inputs,
$output:operation.output)
  $ORCH: Single(parameters.inputs == $input,
    parameters.outputs == $output)
  $OP: Property( propertyName == "UserConsenus",
    subject == $ORCH, satisfied == false)
  $SP: PropertyPlan(properties contains $OP)
then
  PropertyPlan newPropertyPlan = new PropertyPlan ($SP);
  newPropertyPlan.removeProperty($OP);
  insert(newPropertyPlan);
end
```

4.2.2.2 IDENTIFIABILITY

Following a similar approach, the pattern definition on our language could be:

0. **ORCH "Identifiability"**
1. Activity(_a)
2. Activity(_b)
3. Merge(_a,_b)
4. AP_1("Identifiability",_a, certificate)
5. AP_2("Identifiability",_b, certificate)
6. AP_3("Identifiability",dataMerge(_a,_b), patern)
7. OP(Identifiability, subject == "Identifiability", satisfied == false)
8. **Pattern rule: AP_1,AP_2,AP_3 ➔ OP**

1. **ORCH "Identifiability"**
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Merge (A1, A2)
5. Link (L1, A1, A2)
6. Property (Identifiability1, A1, required, (certificate, interface), ?, in_processing)
7. Property (Identifiability2, L1, required, (pattern-based, pattern), ?, in_processing)
8. Property (Identifiability3, A2, required, (certificate, interface), ?, in_processing)
9. Property (Identifiability4, "Identifiability", required, (pattern-based, PR1), ?, end_to_end)
10. **Pattern rule: (PR1: Identifiability1, Identifiability4, Identifiability3 ➔ Identifiability4)**

This pattern can be then translated to a drools engine compatible pattern as the following:

TABLE 13. SPECIFICATION OF GDPR IDENTIFIABILITY VIA DROOLS

```
rule "Identifiability"
when
  $A: Placeholder($output_A: Activity.output)
  $B: Placeholder($output_B: Activity.output)
  $ORCH: Merge($A, $B)
  $OP: Property( propertyName == "Identifiability",
    subject == $ORCH, satisfied == false)
  $SP: PropertyPlan(propeties contains $OP)
then
  PropertyPlan newPropertyPlan = new PropertyPlan($SP);
  newPropertyPlan.removeProperty($OP);
  Property NP_A = new Property($OP, "Identifiability", $A);
  Property NP_B = new Property($OP, "Identifiability", $B);
  insert(NP_A)
  insert(NP_B)
  insert(newPropertyPlan);
end
```

4.3 Dependability

4.3.1 PATTERN DEFINITION

Dependability typically refers to the provision of expected service, towards task accomplishment in a reliable and trustworthy manner, and it entails reliability, safety, availability and security [32]. Nevertheless, the concept of security is covered separately above (see subsection 4.1), and in modern computer engineering, security is considered to encompass availability (along with confidentiality and integrity). Therefore, in the context of this work, Dependability properties will mainly focus on reliability, fault tolerance and safety aspects.

In order to guarantee end-to-end dependability properties, suitable component orchestrations on the different SEMIoTICS layers should be found in order to guarantee the required dependability property. If this does not exist, the substitution or the addition of current components with other atomic ones or orchestrations is required in order to guarantee dependability. This is also related to the components and the topology of the composition.

However, it should be noted that the composition of two components which preserve a dependability property does not necessarily guarantee that the composition will also preserve the same property. In addition, if a composition guarantees the conditions of a property, the atomic components may not preserve the property. As an example, let's consider a sequential (AND) composition of two components:

$$C \rightarrow C_1 \wedge C_2$$

If a required property should be guaranteed by the C, the subcomponents C_1 and C_2 should satisfy the condition:

$$\text{Property}(C, \text{Category}) \rightarrow \text{Property}(C_1, \text{Category}) \wedge \text{Property}(C_2, \text{Category})$$

If there are no atomic components to guarantee the required property a recursive procedure is used in which successive (sub-) orchestrations are generated until the atomic components bound to them satisfy the required properties. The decomposition can be analysed as follows:

$$\text{Property } (C, \text{Category}) \rightarrow \text{Property } (C_1, \text{Category}) \wedge \text{Property } (C_2, \text{Category}) \rightarrow$$

$$(\text{Property } (C_{11}, \text{Category}) \wedge \text{Property } (C_{12}, \text{Category}))$$

$$\wedge (\text{Property } (C_{21}, \text{Category}) \wedge \text{Property } (C_{22}, \text{Category}))$$

…until components C_{11}, C_{12}, C_{21}, C_{22} that satisfy the required property Pro are found

On the other hand, the multi-choice (OR) composition of two components can be expressed as follows:

$$C \rightarrow C_1 \vee C_2 \rightarrow \text{Property } (C, \text{Category}) \rightarrow \text{Property } (C_1, \text{Category}) \vee \text{Property } (C_2, \text{Category})$$

The above procedures can be used to satisfy not only for dependability, but also for all the other SEMIoTICS property requirements.

Moreover, certain dependability properties will need to hold at the **component level** to enable the E2E properties to be achieved. One of the most important issues for a system designer is to validate system dependability of components as a critical condition for the design of complex network infrastructures and identify the weakest components in order to replace, redesign and find alternative solutions. System dependability properties such as reliability and availability depend on component's arrangements. Stepwise decomposition can be used to recursively build network topologies using forward or de-orchestrations using backward chaining respectively, as depicted in Figure 12.

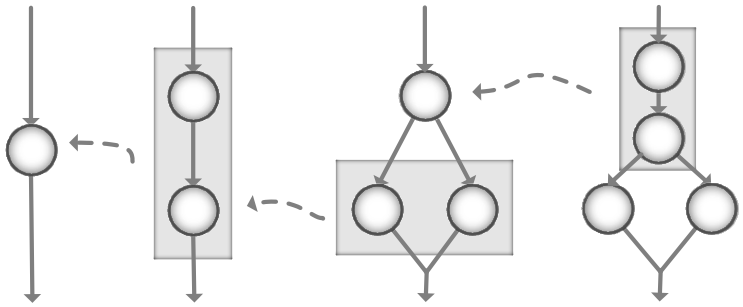


FIGURE 12 STEPWISE DECOMPOSITION

The two basic arrangements which we are focused on are components in series and in parallel. Other arrangements can include parallel-series, k-out-of-n or non-series-parallel systems. More specifically, for components in series, the reliability (or availability) for probabilistic models, quickly decreases as the number of components increases. In a serial system a single failure results in entire assembly or system failure. The addition of new components in series decreases the reliability (or availability) of system. Components in series may have arrangements either following the sequence or parallel-split workflow patterns. This occurs because a failure of a single component will result the failure of the system. Reliability (or availability) of systems in series can be defined as follows:

Definition 1. Let $C=\{C_1,C_2,...C_n\}$ be a number of components in series and R_1, R_2,...,R_n be the reliability of each component, then the component composition C will have reliability r equal to:

$$R=\prod_{k=1}^{n}(R_k)$$

In components in parallel, the reliability (or availability) of the system exists only when at least one component is functional. The reliability of the system is the 1 minus the probability that all fail. In parallel components, all redundant units' failure causes system failure. Thus, the addition of components in parallel increases the reliability of the subsystem. We may associate the multi-choice pattern as a parallel arrangement because the failure of a single component does not cause system failure. Reliability of components in parallel can be defined as follows:

Definition 2. Let C = {C_1,C_2,...C_n} be a number of components in parallel and R = {R_1,R_2,⋯,R_n} be the reliability of each component, then the parallel component composition C will have reliability R:

$$R=1-\prod_{k=1}^{n}(1-R_k)$$

In case of arithmetic models such as latency for availability, the following approaches can be used:

1) For components in series (sequential): $A=\sum_{k=1}^{n}A_k$
2) For components in parallel (multi-choice): $A=min\{A_1,A_2,\dots,A_n\}$
3) For components in parallel (parallel split): $A=max\{A_1,A_2,\dots,A_n\}$

Where A is the total system availability and A_k for k=1:n.

For the SEMIoTICS IoT applications that require end-to-end dependability, a vertical cross layer component composition in series can be defined.

As far as data state coverage is considered, the Dependability pattern covers the in_transit data state. Moreover, it refers to components that are within the SEMIoTICS platform.

4.3.2 PATTERN SPECIFICATION RULE

Reliability pattern can be expressed as rules in Drools production rules. They encode orchestrations in Drools corresponding to the structure of the logical reliability arrangements. It also specifies rules that dictate the properties that the constituent components must have.

$$R(t)=\text{Prob}(Comp \text{ is fully functioning in } [0,t])$$

metric to measure the Reliability of the composition.

We may consider two activities *A* and *B* having specific source operation inputs and outputs and reliability r1 and r2 respectively. The composition of the two activities will be described as a new activity with reliability R based on the components' arrangement. For the component composition in series, the control flow describes the serial arrangement of the components based on the sequence workflow pattern. The data flow defines that the outputs of the activity *A* will be the inputs of component *B*. In addition, the reliability property guaranteed by a serial component composition is equal to R = R1 · R2. Therefore, the guaranteed reliability property R should satisfy the required reliability property Rreq ≤ R. The encoded pattern in Drools is depicted in Table 13.

TABLE 14. VERIFICATION OF SEQUENTIAL RELIABILITY VIA DROOLS

```
rule "Serial Reliable Composition"
when
  $A: Placeholder($input : operation.inputs, $intData: parameters.outputs,
                  $r1:= reliabilityValue)
  $B: Placeholder(parameters.inputs == $intData, $output: parameters.outputs,
                  $r2:= reliabilityValue)
  $ORCH: Sequence(parameters.inputs:= $input, parameters.outputs == $output,
                firstActivity == $A, secondActivity == $B)
  $OP: Property(subject:= $ORCH, propertyName== “Reliability",
        $rel:= propertyValue, $rel<= $r1*$r2, satisfied == false)
  $SP: PropertyPlan(property contains $OP)
then
  PropertyPlan newPropertyPlan = new PropertyPlan($SP);
  newPropertyPlan.removeProperty($OP);
  Property NP_A = new Property($OP, "Reliability", $A);
  newPropertyPlan.getProperty().add(NP_A);
  insert(NP_A);
  Property NP_B = new Property($OP, "Reliability", $B);
  newPropertyPlan.getProperties().add(NP_B);
  insert(NP_B);
  insert(newPropertyPlan);
  modify($OP){satisfied=true};
end
```

4.4 Interoperability

As discussed in Section 2.4, four levels of interoperability are considered in SEMIoTICS: technological, syntactic, semantic and organizational interoperability. An approach towards a pattern rule definition for these cases is detailed below.

As far as data state coverage is considered, the set of interoperability patterns covers all three data states, in_transit, at_rest and in_processing. Moreover, all of them may refer to components that are within the SEMIoTICS platform or across different IoT platforms.

4.4.1 PATTERN DEFINITION

4.4.1.1 TECHNOLOGICAL INTEROPERABILITY

Definitions:

C := the set of all instantiated components

TA:= A set of technological attributes

$C_1, C_2 \subseteq C$, where $C_1 \neq C_2$

$C_{iTA} \subseteq TA :=$ technological attributes of Ci

TMD:= Technological mediator, Mediator which connects to components with various technological attributes

Lemma 1: If C_1, C_2 are at the same domain and $C_{1TA} \cap C_{2TA} \neq \varnothing$ then C_1 and C_2 are directly technological interoperable

Lemma 2: If C_1,C_2 are on different domain but are both direct technological interoperable with TMD **FIGURE 13** then C_1,C_2 are indirect technological interoperable

FIGURE 13 COMPONENTS TECHNOLOGICAL DOMAINS

Lemma 3: If C_1,C_2 are direct or indirect technological interoperable then C_1,C_2 are technological interoperable

4.4.1.2 SYNTACTIC INTEROPERABILITY

Definitions :

C := the set of all instantiated ingredients/activities

PR:= A set of protocols

C_1,$C_2 \subseteq C$, where $C_1 \neq C_2$

$C_{iPR} \subseteq PR :=$ protocols supported by Ci

SMD:= Syntactic mediator, Mediator which connects to components with various protocols

Lemma 1: If C_1,C_2 are technologically interoperable and $C_{1PR} \cap C_{2PR} \neq \varnothing$ then C_1 and C_2 are directly syntactical interoperable

Lemma 2: If C_1,C_2 are technologically interoperable and are both direct syntactical interoperable with SMD **FIGURE 14** then C_1,C_2 are indirect syntactical interoperable

FIGURE 14 SYNTACTICAL INTEROPERABILITY DIAGRAM

Lemma 3: If C_1,C_2 are direct or indirect syntactical interoperable then C_1,C_2 are syntactical interoperable.

4.4.1.3 SEMANTIC INTEROPERABILITY

Definitions :

C := the set of all instantiated components

MDL:= A set of models

C_1,$C_2 \subseteq C$, where $C_1 \neq C_2$

$C_{iMDL} \subseteq MDL :=$ semantic models used by Ci

SB:= Semantic broker, Broker defined in Section 2.4

Lemma 1: If C_1,C_2 are syntactic interoperable and $C_{1MDL} \cap C_{2MDL} \neq \varnothing$ then C_1 and C_2 are directly semantic interoperable

Lemma 2: If C_1,C_2 are syntactic interoperable and are both direct semantic interoperable with SMD **FIGURE 14** then C_1,C_2 are indirect semantic interoperable

FIGURE 15 SEMANTIC INTEROPERABILITY DIAGRAM

Lemma 3: If C_1,C_2 are direct or indirect semantic interoperable then C_1,C_2 are semantic interoperable

4.4.2 PATTERN SPECIFICATION RULE

Using the above detailed pattern we can define our interoperability patterns as follows.

1. WF "technical-interoperability"
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (TMD, (PlaceholderActivity,"technological mediator"))
5. Link (L1, A1, A2)
6. Link (L2, A1, TMD)
7. Link (L3, A2, TMD
8. Property (conn1, L1, required, (pattern-based, pattern)," technical-interoperability" , in_processing)
9. Property (conn2, L2, required, (pattern-based, pattern)," technical-interoperability" , in_processing)
10. Property (conn2, L3, required, (pattern-based, pattern)," technical-interoperability" , in_processing)
11. Property (conn4, "technical-interoperability", required, (pattern-based, PR1),"technical-interoperability", end_to_end)
12. **Pattern rule: (PR1: conn1 || (conn2, conn3) ➔ conn4)**

1. WF "syntactic-interoperability"
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (SMD, (PlaceholderActivity,"syntactic mediator"))
5. Link (L1, A1, A2)
6. Link (L2, A1, SMD)
7. Link (L3, A2, SMD
8. Property (conn0, L1, required, (pattern-based, pattern)," technical-interoperability" , in_processing)
9. Property (conn1, L1, required, (pattern-based, pattern)," syntactic-interoperability" , in_processing)
10. Property (conn2, L2, required, (pattern-based, pattern)," syntactic-interoperability" , in_processing)
11. Property (conn2, L3, required, (pattern-based, pattern)," syntactic-interoperability" , in_processing)
12. Property (conn4, "syntactic-interoperability", required, (pattern-based, PR1),"syntactic-interoperability", end_to_end)
13. **Pattern rule: (PR1: (conn0,conn1) || (conn0,conn2, conn3) ➔ conn4)**

1. WF "semantic-interoperability"
2. Placeholder (A1, (PlaceholderActivity, PlaceholderDescription))
3. Placeholder (A2, (PlaceholderActivity, PlaceholderDescription))
4. Placeholder (SB, (PlaceholderActivity,"Semantic Broker"))
5. Link (L1, A1, A2)
6. Link (L2, A1, SB)
7. Link (L3, A2, SB
8. Property (conn0, L1, required, (pattern-based, pattern)," syntactic-interoperability" , in_processing)
9. Property (conn1, L1, required, (pattern-based, pattern)," semantic-interoperability" , in_processing)
10. Property (conn2, L2, required, (pattern-based, pattern)," semantic -interoperability" , in_processing)

11. Property (conn2, L3, required, (pattern-based, pattern),"_semantic -interoperability" , in_processing)
12. Property (conn4, "semantic-interoperability", required, (pattern-based, PR1)," semantic-interoperability", end_to_end)
13. **Pattern rule: (PR1: (conn0,conn1) || (conn0,conn2, conn3) ➔ conn4)**

Moreover, the specification in Drools is shown in Table 14.

TABLE 15. SPECIFICATION OF INTEROPERABILITY VIA DROOLS

```
rule "Interoperability"
when
  $A: Placeholder($input : operation.inputs,
    $intData : parameters.outputs)
  $B: Placeholder(parameters.inputs == $intData,
    $output : parameters.outputs)
  $ORCH: Link(firstActivity == $A, secondActivity == $B)
  $OP: Req( propertyName == "Interoperability",
    subject == $ORCH, satisfied == false)
  $SP: PropertyPlan (properties contains $OP)
then
  PropertyPlan newPropertyPlan = new PropertyPlan($SP);
  newPropertyPlan.removeRequirement($OP);

  Req Technological = new Req($OP,"Technological",ORCH);
  newPropertyPlan.getProperties().add(Technological);
  insert(Technological);
  Req Syntactic = new Req($OP,"Syntactic",ORCH);
  newPropertyPlan.getProperties().add(Syntactic);
  insert(Syntactic);
  Req Semantic = new Req($OP,"Semantic",ORCH);
  newPropertyPlan.getProperties().add(Semantic);
  insert(Semantic);
  insert(newPropertyPlan);
end
```

4.5 Pattern Summary

Aggregating the above first set of patterns presented in this deliverable, Table 16 presents the coverage that this first set of patterns offers in terms of the considered properties, data states and platform connectivity cases covered.

TABLE 16. SUMMARY OF 1ST SET OF SPDI PATTERNS AND THEIR COVERAGE

Pattern		Property							Data State Coverage			Platform Connectivity	
		Security			Priv.	Depend.	Interop.	QoS	In Transit	At Rest	In Processing	Within	Across
#	Name	Conf.	Int.	Avail.									

1	Perfect Security Property (PSP)	✓							✓	✓	✓	✓	
2	Sequential Orchestration Integrity		✓						✓	✓	✓	✓	
3	Readiness			✓						✓	✓	✓	✓
4	Consent				✓					✓	✓	✓	
5	Identifiability				✓					✓	✓	✓	
6	Serial Reliability					✓			✓			✓	
7	Parallel Reliability					✓			✓			✓	
8	Technical Interoperability						✓		✓	✓		✓	✓
9	Syntactic Interoperability						✓			✓	✓	✓	✓
10	Semantic Interoperability						✓				✓	✓	✓

5 IOT SERVICE ORCHESTRATION

The materialization of the Internet of Things is done nowadays by various IoT platforms, offering devices and services (IoT offerings). This is a world where applications are distributed and realized by the interoperations among services, in order to become more capable and powerful. The next logical step towards the facilitation of the usage of IoT offerings is their composition. As a result, we need a language to express how composing parts of IoT applications are put together in a workflow. Our choice is the IoT Recipe model (see D3.4, Section 3.3.2), which allows the aggregation of IoT offerings based on semantic composition rules. The tool-kit that accompanies the Recipe model includes a lightweight graphical tool that eases the creation of Recipes as composition of offerings. What follows is the instantiation of the Recipe and the automatic production of executable application code.

Regarding the description of a workflow in details, the IoT Recipe model is quite expressive. Its core structure shares many characteristics with BPMN 2.0, the global standard for business processes. First of all, the *Task* of BPMN corresponds to the Ingredient of the Recipe model. A *Task* is an atomic *Activity,* a work that cannot be broken down to a finer level of detail. An Ingredient corresponds to data or a function offered by a provider. Then, in BPMN we have the concept of Sub-process that can be opened up to show its internal details. This is also offered by Recipe model since a Recipe is an offering. That means that Sub-recipes are supported. Moreover, Recipe Patterns correspond to BPMN *Gateways*. *Gateways* are responsible for the control of the Process flow and are separated in different types (Exclusive, Inclusive, Parallel, Event-Based). Recipe Patterns are also Ingredients and are used for the expression of the conditions under which Ingredients connect. The list of the Recipe Patterns types includes If-Then, If-Then-Else, Sequence, Conjunction, Disjunction, Negation, Iterate, Repeat-until, Repeat-while, Split, Unordered list, Choice and Split-Join. Finally, in BPMN there is the concept of *Event*, something that happens during the Process and affects the flow. *Events* make event-driven processes possible. The concept of Event was not present in the first version of Recipe model. However, an extension allowed the support of for asynchronous, event-based offerings composition. A detailed presentation of the aforementioned model is available in Section 5.1 below.

Although Recipe model is quite capable to describe IoT workflows, some extensions are necessary in order to for us to achieve application orchestrations that guarantee SPDI properties. First of all, we need to be able to describe Links of the network level. The attributes of a Link are described in Section 3.3. Furthermore, Ingredients must be extended to include information about their SPDI properties. Only QoS constraints are offered at the moment. Finally, monitoring capabilities should be added to Ingredients that will provide the evidence for the presence of the SPDI and QoS properties.

5.1 Recipe-driven IoT Application Workflow definition

In this section[8], we present an extension of the IoT Recipe model (D3.4, Section 3.3.2), which allows the description of IoT application workflows. We add the ability to specify QoS requirements within the model. This can be utilized by user interfaces for IoT application developers to enable the expression of QoS requirements in a simplified manner at the application layer. We aim to translate these user-defined QoS requirements then into concrete SPDI patterns that define a specific SDN/NFV configuration. Using the recipe model here has the advantage of facilitating the IoT application development and will make IoT applications more reliable as networking QoS constraints can be integrated more easily.

[8] This section is based on a paper currently under review for publication at the EUCNC conference (https://www.eucnc.eu).

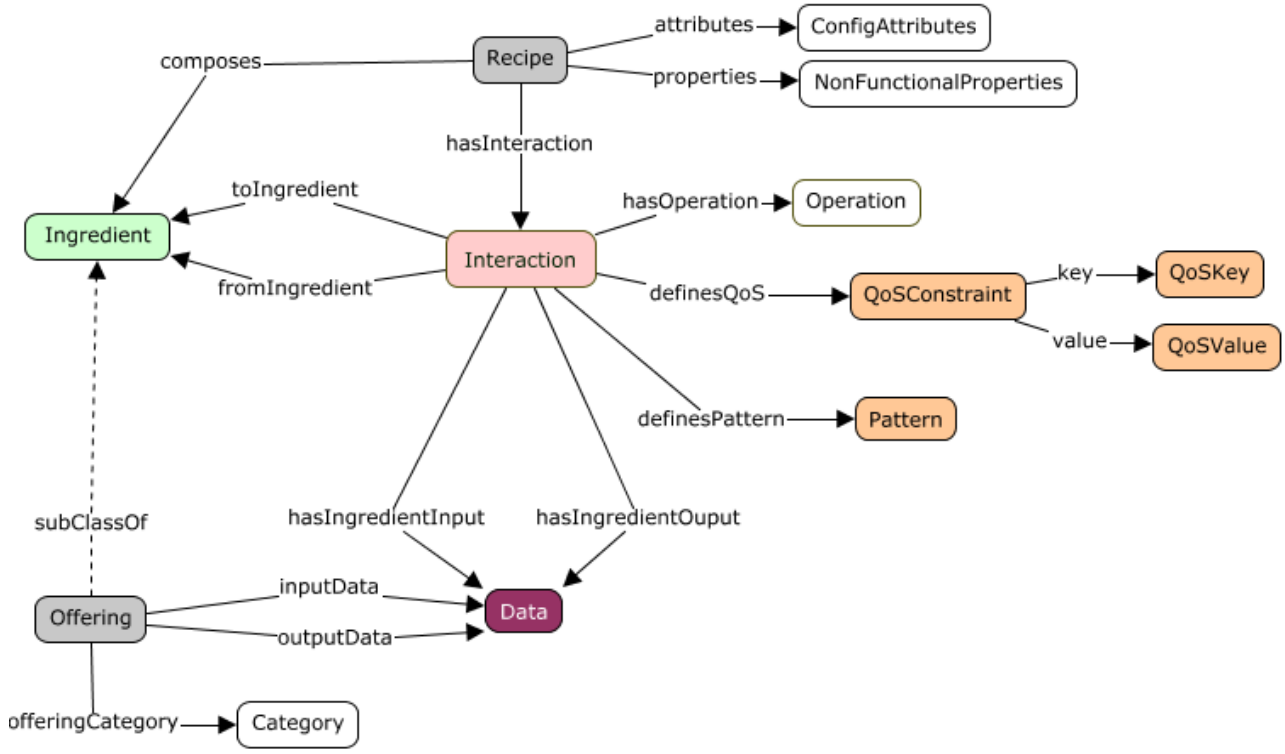


FIGURE 16. THE RECIPE MODEL

In previous work [26], [1] we have presented a model to define abstract IoT orchestrations as *recipes*. In this model, shown in Figure 16, a recipe is a template for a workflow of interactions between multiple *ingredients*, i.e., devices or services. When a recipe is instantiated, ingredients are replaced with concrete *things*, described with their own respective Thing Description. A draft for a user interface (UI) for the specification of recipes can be seen in Figure 17. Besides the workflow of the recipe, QoS constraints and SPDI patterns can be defined on the interactions.

The user of this tool would be typically an IoT application developer. This user wants to focus on the logic of the application flow. Specifically, the user does *not* have to have expertise in configuring the network and physical connections between the involved IoT devices. The **benefit of the recipe approach** is that these configurations are automatically done by the tool and the underlying technologies.

In the example, the UI has been used to define a recipe that combines multiple services of devices within a wind turbine (microphone, accelerometer and anemometer) with the purpose of sending out alarms in case of severe conditions (i.e., detected noise, motion and winds) are above a threshold. The abstract service composition and associated constraints are first defined in the UI. The composition and graph are then translated to concrete configurations.

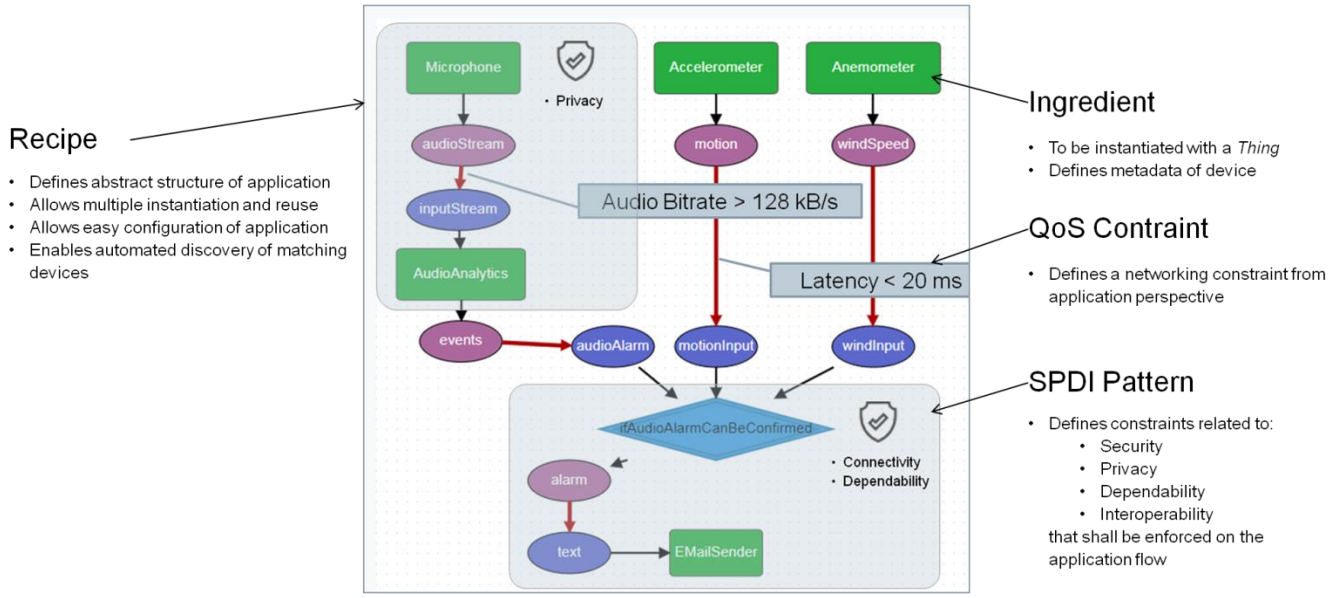


FIGURE 17. EXAMPLE USER INTERFACE TO DEFINE RECIPES

In Figure 18, the sequence of creating and instantiating a recipe is depicted. First, the *recipe creation* phase takes place. This is independent of the later instantiation phase and can also be done by a different user.

1. At startup, the Recipe Cooker tool requests the capabilities of all available things on the network from the Thing Directory on the backend. These capabilities are reported back to the Recipe Cooker in the Thing Description format. Besides the syntax and semantics of inputs and outputs, the Thing Description defines general capabilities (e.g., resolution of a camera).
2. Next, the user defines the recipe (i.e., the application flow including if/else and for-loops) and specifies the expected capabilities (selected from the downloaded thing capabilities) of ingredients, such as input and output data types. The user utilizes the Recipe Cooker tool for this specification.

Second, the *recipe instantiation* phase takes place. This phase can be conducted by a different user and could potentially happen at a much later point in time.

1. The user starts by selecting a recipe that reflects as a template the workflow he wants to implement in his site. Therefore, the Recipe Cooker requests all semantically matching things (for the ingredients of the selected recipe) from the Thing Directory. The computational complexity of this matching process (simple subsumption reasoning) was tested in our previous work [26]. It scales well enough, on a machine with 8GB RAM and 2.4 GHz i5 intel processor, it results in a computation time of one second being broken at about 650 recipes. The system scales quadratically in the number of recipes, but with low constant factors. Thereby, the closed world assumption is held here: the knowledge base is known to be complete, since the Thing Directory is held up-to-date by design of our multi-layer architecture.
2. Next, the user can select a concrete thing for each ingredient. This manual step, conducted by the user (application developer), ensures the proper selection of suitable activities in the application composition. The activities in this composition are the matching and selected things retrieved from the Thing Directory. They are described using the Thing Description (TD) model and format defined by the W3C Thing Description specification.
3. Then, the user triggers the deployment of the recipe instance. Therefore, the recipe instance is transmitted to the Pattern Orchestrator in the format of a so-called Recipe Runtime Configuration (RRC). The RRC is then translated into:
    a. network pattern (i.e., configurations of the SDN controller as Drools rules)

b. interaction descriptors for each involved thing; which are then uploaded to the Semantic API of the Gateway to which the thing belongs.

4. If the network configuration and the interaction configuration were successful, the RRC is stored as “active” in the Recipe Cooker and the successful deployment is displayed to the user (or an error is displayed otherwise).

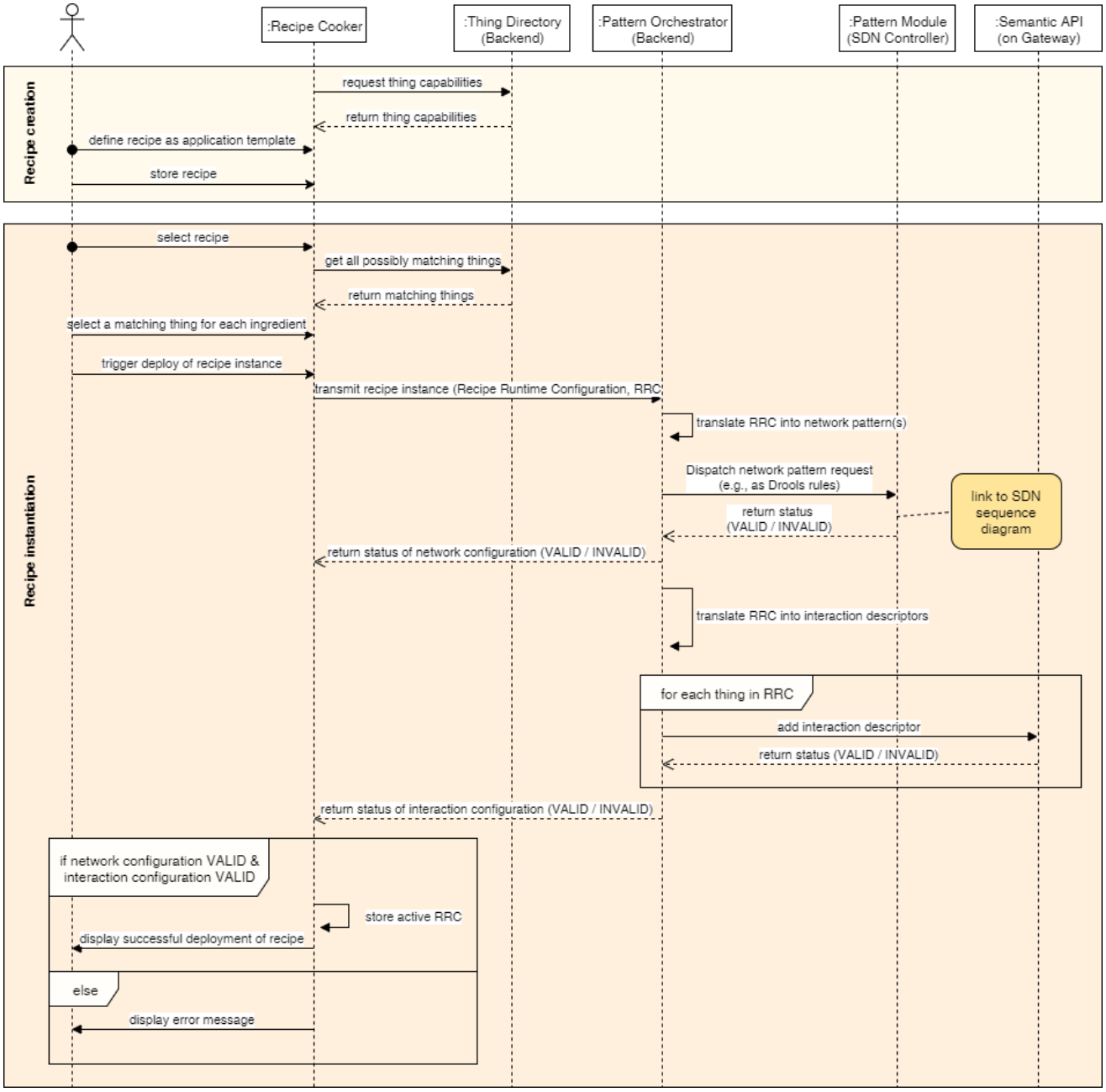


FIGURE 18: SEQUENCE OF DEFINING AND DEPLOYING A RECIPE

6 RECIPES & PATTERNS INTEGRATION

The integration of the Recipes approach detailed in section 5 above, with the SEMIoTICS SPDI patterns will enable the user-friendly, abstract definition of IoT service orchestrations, with the SPDI guarantees provided by the patterns. In more detail, the encoding of the dependencies will allow the verification that a specific SEMIoTICS service workflow, as defined in the associated Recipe, satisfies certain SPDI properties, but also the generation (and adaptation) of a workflow in a manner that guarantees the satisfaction of the needed SPDI properties

For integrating the user-friendly abstraction for IoT service orchestration provided by Recipes, the activity-based IoT orchestration model followed for Pattern definition, as detailed in Section 3, is mapped to the ingredient-based view of Recipes.

In more detail, Recipes are mapped to Workflows (i.e. orchestrations of activities), and similarly, the individual atomic building blocks are also mapped, i.e. ingredients are mapped to activities. A simple example of this is depicted in Figure 19, showing a simple Recipe involving two ingredients and a Workflow involving two activities that matches said Recipe.

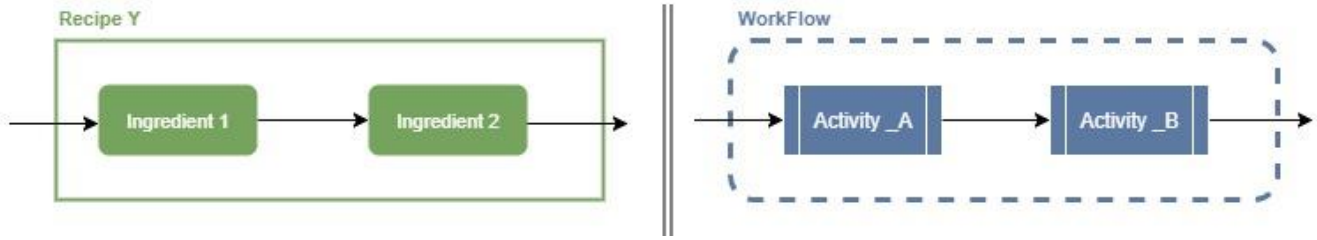


FIGURE 19. A SIMPLE RECIPE (LEFT) AND A MATCHING WORKFLOW PATTERN (RIGHT)

The same match can happen in cases of more complex Recipes, whereby existing Recipes are used as ingredients to new, more complex recipes, since these can be mapped to sub-Workflows. An example of this mapping for a complex Recipe consisting of two ingredients, the second of which is another recipe, is depicted in Figure 20. The various Recipe parameters such as requirements, constraints, properties and orchestration details are also mapped to the corresponding elements in the workflow view.

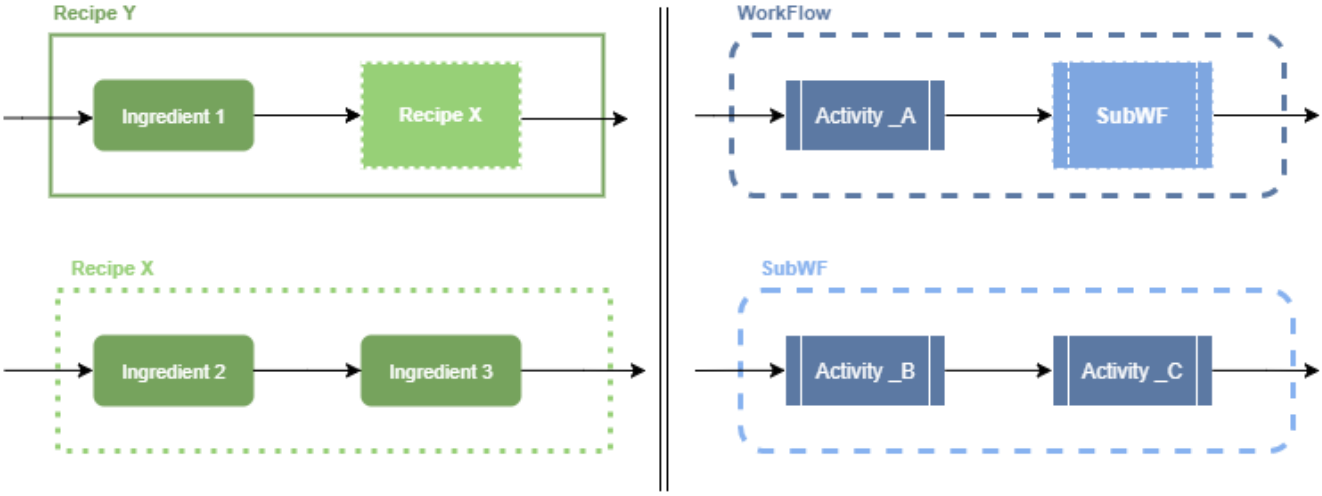


FIGURE 20. A COMPLEX RECIPE (LEFT) AND A WORKFLOW PATTERN MATCHING SAID RECIPE (RIGHT)

Upon Recipe instantiation, the descriptions and characteristics of the specific offerings selected provide the necessary information needed to model the actual workflow, and are passed over from Thing Descriptions' of

the offerings to the various variables and placeholders present in the equivalent Workflow representation. For the Recipe shown in Figure 20, this transformation is visualised in Figure 21.

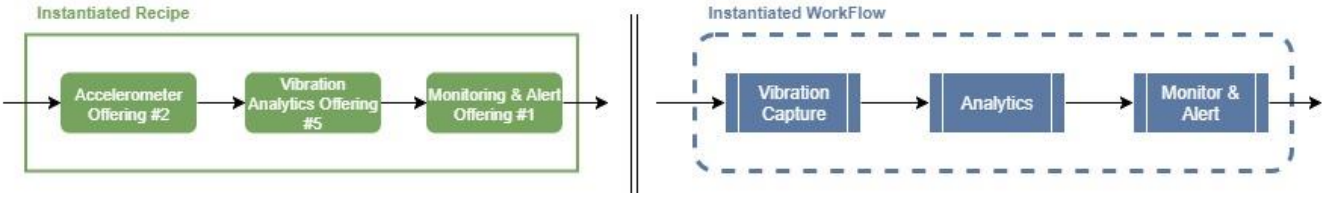


FIGURE 21. INSTATIATION OF RECIPE, REPLACING INGREDIENTS WITH SELECTED OFFERINGS (LEFT) AND INSTANTIATION OF WORKFLOW, REPLACING ACTIVITY PLACEHOLDERS WITH ACTUAL ACTIVITIES (RIGHT)

The sequence diagrams for the interactions between the corresponding components of the SEMIoTICS architecture at design time and at runtime are depicted in Figure 22 and Figure 23 respectively. More specifically, Figure 22 shows the instantiation phase, where, following the Recipe definition and instantiation, via the Recipe Cooker module, the SPDI and QoS properties of the Recipe that are defined for the specific workflow are translated into the equivalent Pattern rules. These are in turn stored at the Pattern Global Repository and, via the Pattern Orchestrator, are sent to the Pattern Repositories residing in the lower layers (i.e. network and field), with each of those receiving the set of rules that pertains to the operation of the specific layer. At runtime (Figure 23), the Pattern Engines at each layer are responsible for retrieving, reasoning upon and updating the rules and facts stored on their local repositories, based on inputs they constantly receive from the Monitoring elements available at the various components at their layers that participate in the workflow. In the case of the Network and Field layers, any updates must be also relayed back at the Global repository, in order to allow it to have an up-to-date view of the state of the system at the various layers.

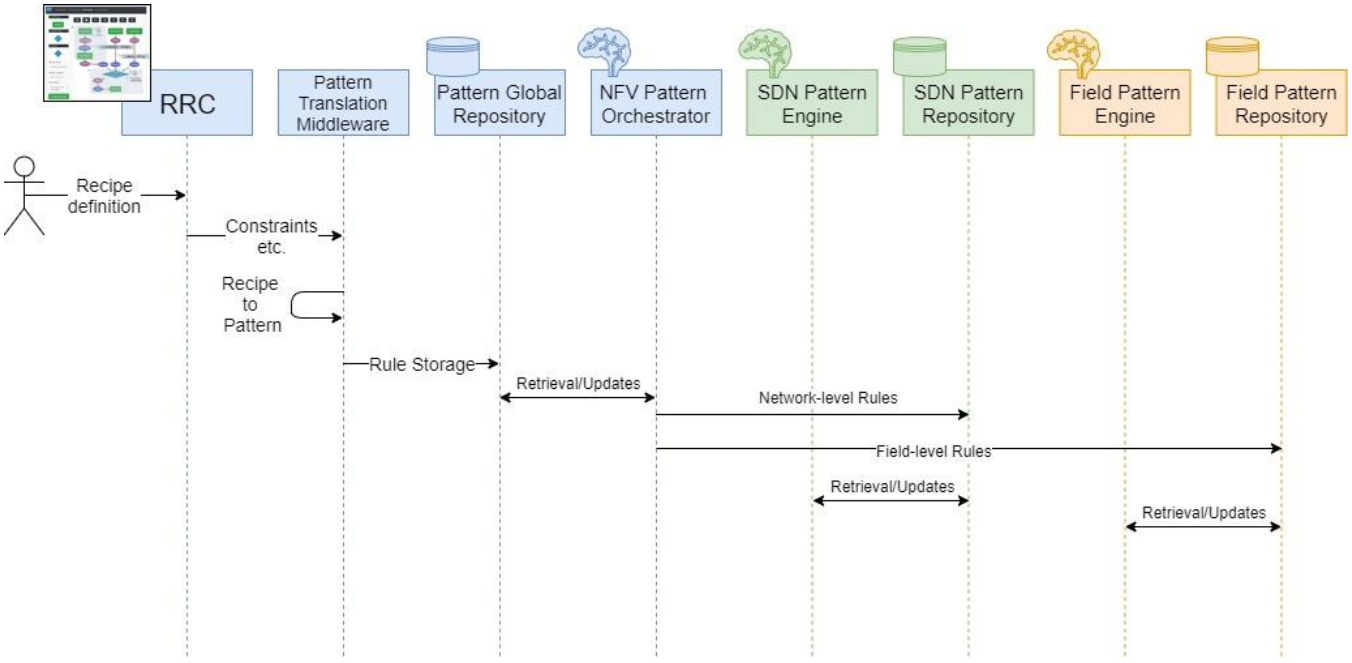


FIGURE 22. INSTANTIATION OF PATTERN COMPONENTS ACROSS LAYERS

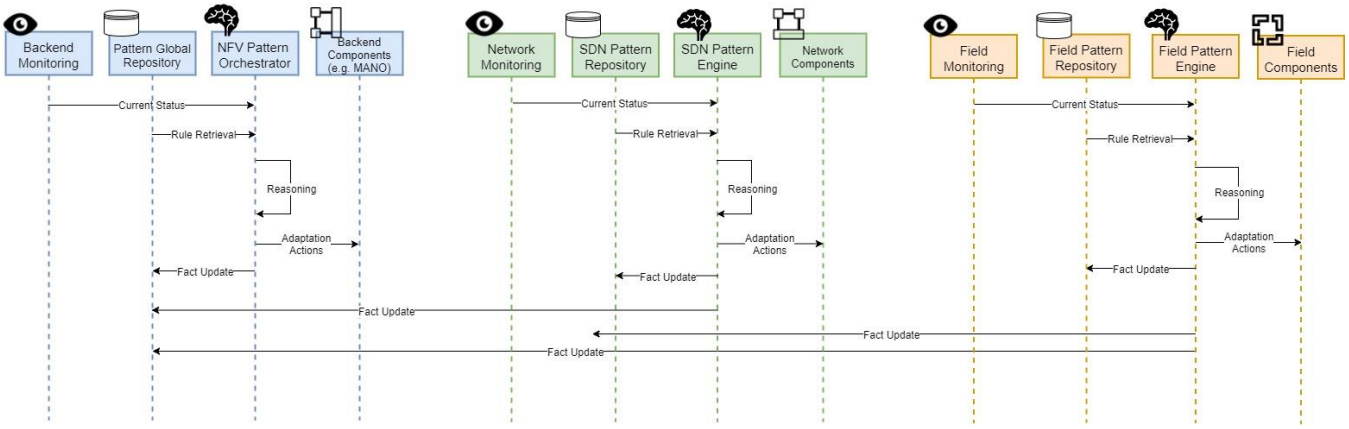


FIGURE 23. RUNTIME DIAGRAM OF PATTERN COMPONENTS

6.1 Application Example

To demonstrate the use of the concepts and constructs defined in the above sections, as well as the Recipes & Patterns integration, a simple application example will be sketched in this subsection. In more detail, the scenario considered is depicted in Figure 24, with key aspects detailed below:

- **Scenario:** SEMIoTICS-enhanced Wind Park IIoT deployment
- **Interaction:** Data captured by IIoT accelerometer sensor on Wind Turbine is relayed to IIoT gateway for vibration analytics, and the output of the analytics is relayed to the backend for monitoring and alarm purposes.
- **SPDI Property required**: End-to-end confidentiality

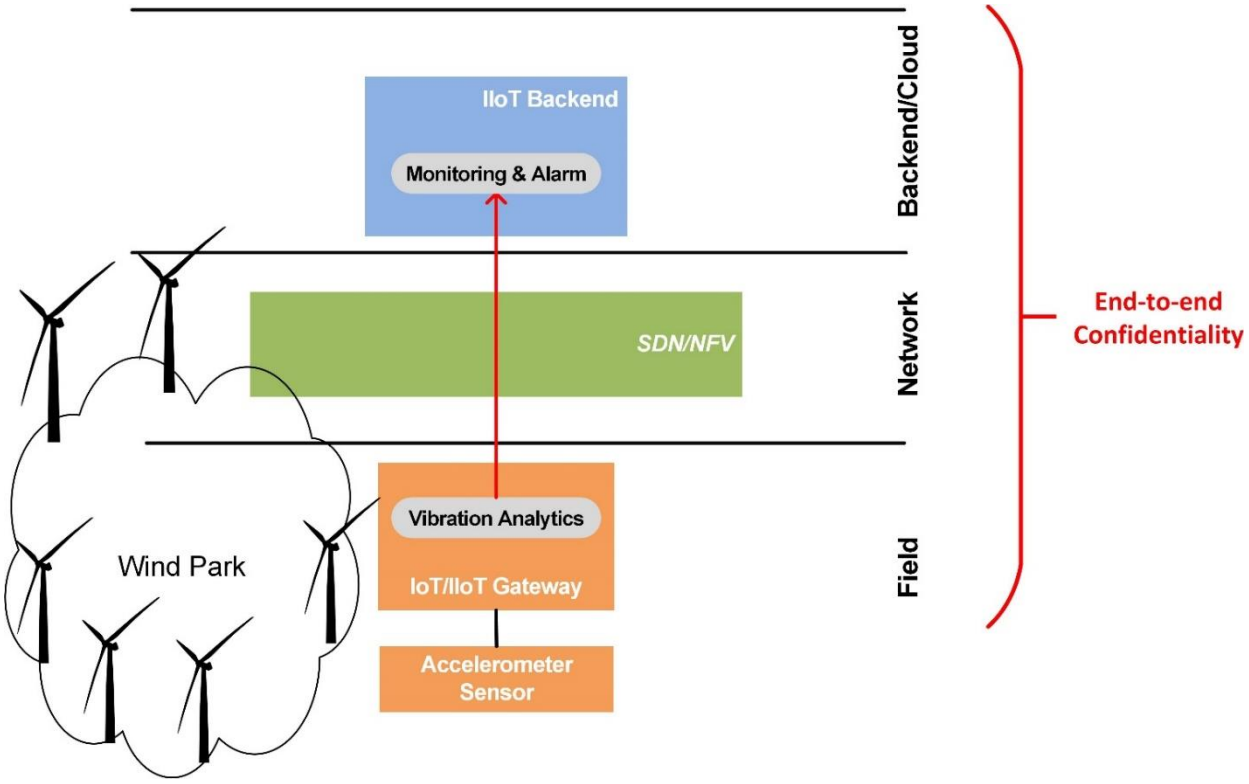


FIGURE 24. EXAMPLE OF IIOT APPLICATION

6.1.1 DESIGN

The design of the "Windturbine Vibration Monitoring" Recipe implementing the above scenario is depicted in Figure 25, with ingredients "Vibration Analytics" and "Monitoring & Alarm", as well as the Confidentiality property covering the whole Recipe. The matching Workflow would be a simple sequential workflow with two activities, much like the one shown on the right side of Figure 19.

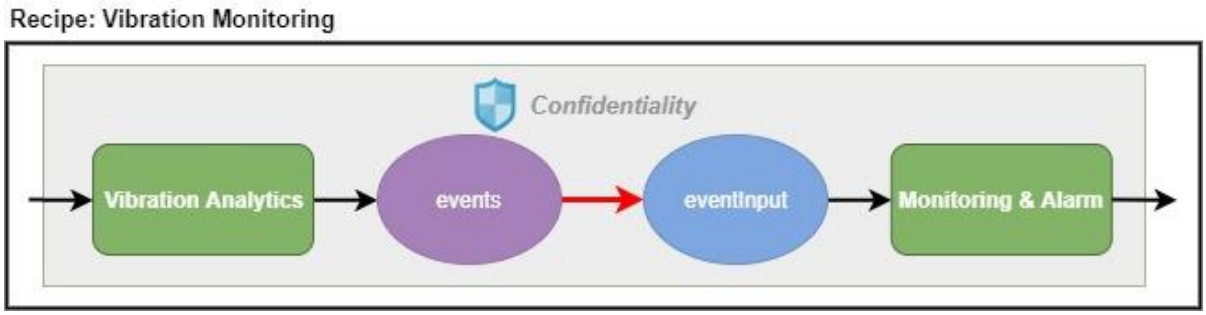


FIGURE 25. THE WINDTURBINE VIBRATION MONITORING RECIPE

6.1.2 INSTANTIATION

When instantiating the above-defined Recipe, the appropriate offerings are selected to implement the desired process; e.g., the "Vibration Analytics Offering #1" offering is selected to implement the "Vibration Analytics" ingredient. Thus, the instantiated version of the recipe of Figure 25 is shown in Figure 26 (top) with the workflow view equivalent also appearing on the same figure (bottom). In the latter, the activity placeholders are replaced with specific activities ("Vibration Analysis" and "Monitoring Alarm", respectively), with specifics on their

characteristics (e.g., inputs/outputs), as well as the end-to-end confidentiality property defined in the Recipe, which is now also broken down to individual properties for the two activities and the link between them.

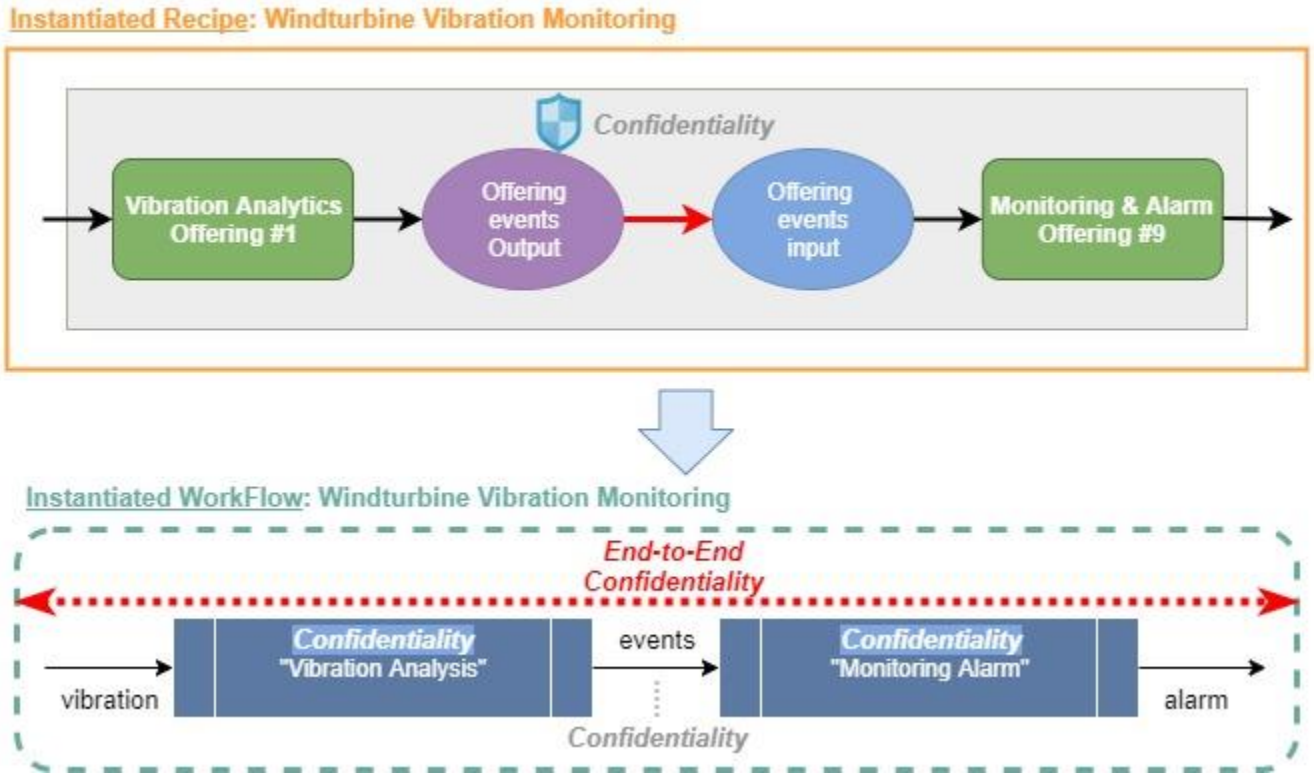


FIGURE 26. INSTANTIATED RECIPE (TOP) AND WORKFLOW (BOTTOM)

Using the language defined in subsection 3.3, the above workflow can be formally described as follows:

0. **ORCH "Seq2"**
1. Placeholder (Placeholder1, (Vibration Analysis Activity, Vibration Analysis Description))
2. Placeholder (Placeholder2, (Monitoring Alarm Activity, Monitoring Alarm Description))
3. Sequence (Placeholder1, Placeholder2)
4. Link (Link1, Vibration Analysis, Monitoring Alarm)
5. Property (AP_1, Placeholder1, required, (certificate, interface), confidentiality, in_processing)
6. Property (AP_2, Link1, required, (pattern, "PSPpattern"), confidentiality, in_transit)
7. Property (AP_3, Placeholder2, required, (monitoring, interface), confidentiality, at_rest)
8. Property (OP, "Seq2", required, (pattern-based, "PR1"), confidentiality, end_to_end)
9. **Pattern rule: (PR1: AP_1, AP_2, AP_3 → OP)**

A visualisation of the above rule for the specific scenario discussed, whereby the end-to-end confidentiality property of the workflow has to be evaluated by checking the individual AP (and if these hold, then the OP holds), is visualised in Figure 27 below.

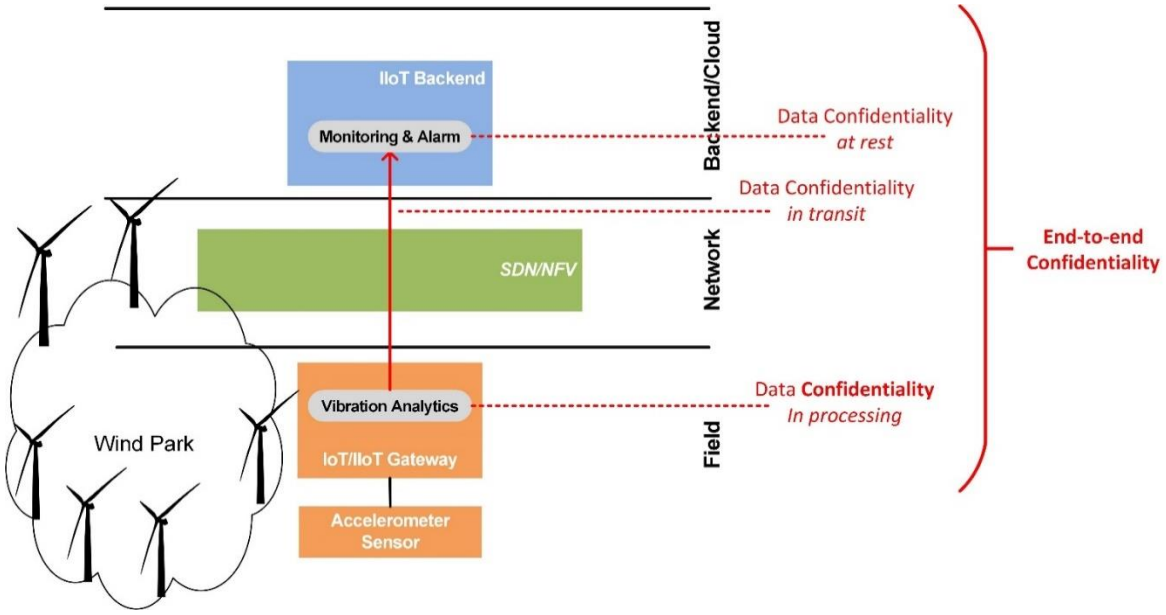


FIGURE 27. VISUALISATION OF SAMPLE APPLICATION, DEPICTING INDIVIDUAL AP

6.1.3 DEPLOYMENT

In Figure 28 the steps of the next phase, i.e. the system deployment, are shown, following the generic process detailed in subsection 3.6.2.2.

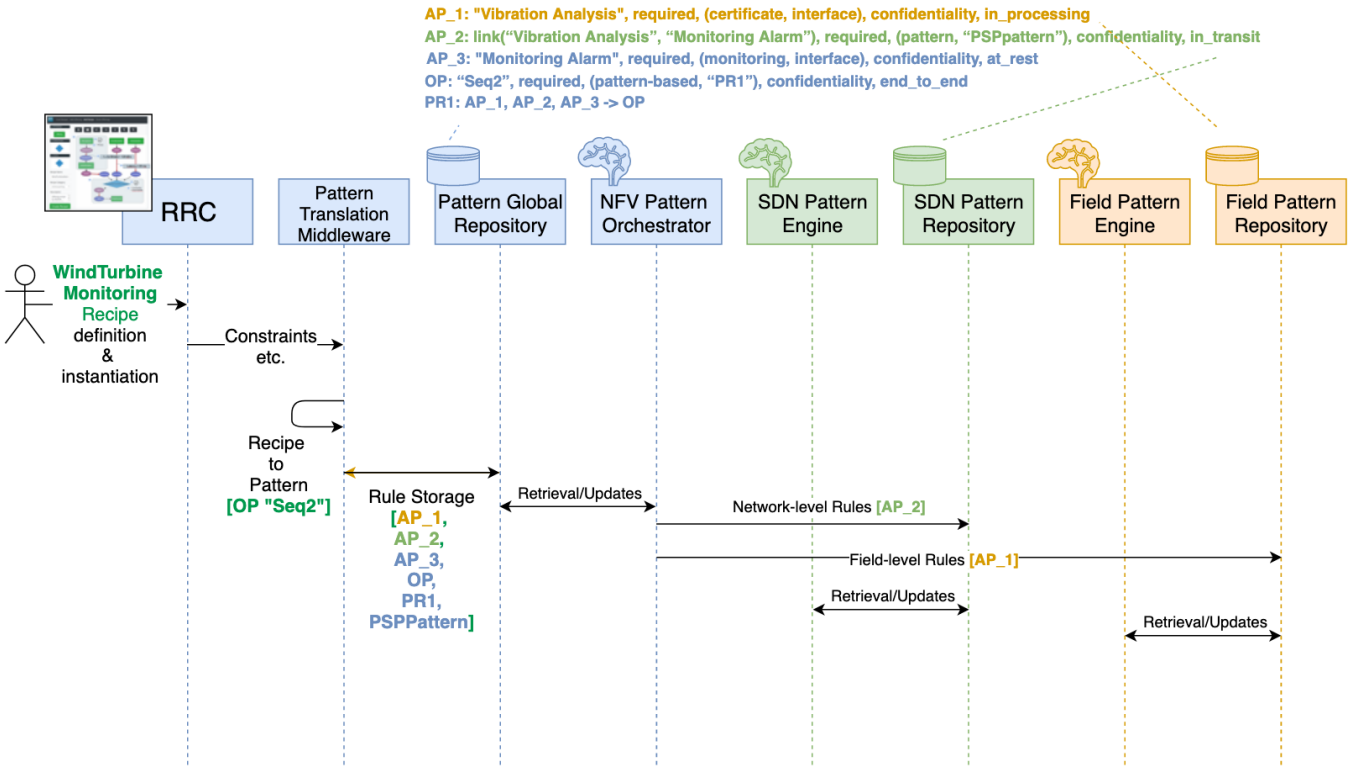


FIGURE 28. SYSTEM DEPLOYMENT PHASE

In more detail, following the transfer of the instantiation specifics from the Recipes plane to the Workflow pattern plane, the rules for the individual properties are stored on the Pattern Global Repository and then relayed by the Pattern Orchestrator to the pertinent layers for monitoring and verification; i.e. AP_1, at the IIoT gateway, AP_2 to the SDN Controller, while AP_3 only stays at the backend.

6.1.4 RUNTIME

At runtime, the individual SDN pattern engines collect monitoring data from the corresponding interfaces defined for each property at the specific layer's components, reason on collected data and trigger adaptation actions if needed. Changes in the system state related to the monitored properties are stored as new facts or trigger updates in the stored facts in the corresponding Pattern repositories; for the network and field pattern engines, these are also transferred to the backend repository, to enable it to have an up-to-date global view of the SPDI state of the whole deployment. This process, again based on the generic scheme defined in subsection 3.6.2.2, is shown in Figure 29.

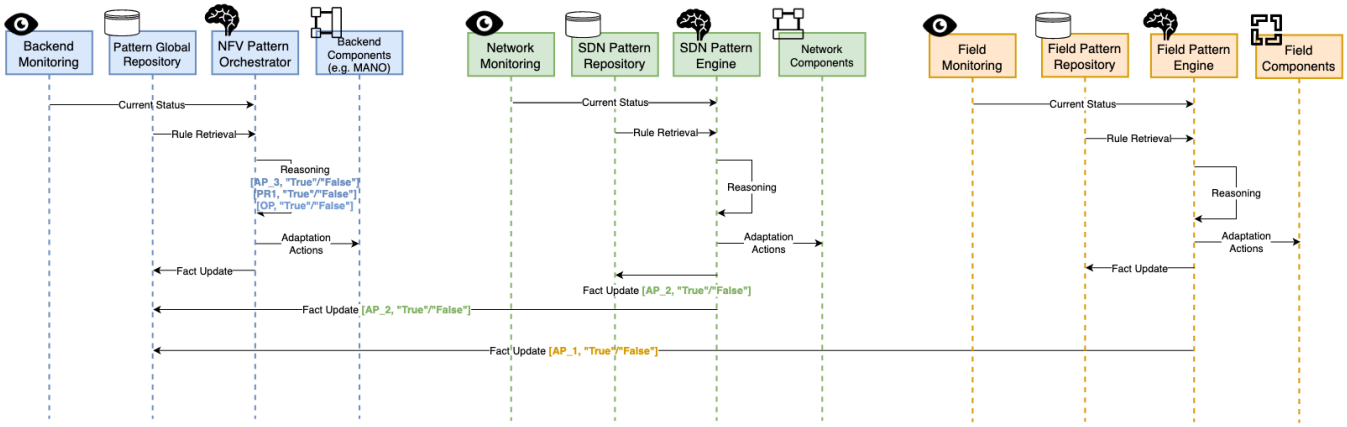


FIGURE 29. SYSTEM RUNTIME MONITORING AND ADAPTAT

6.2 Pattern-driven Orchestration Adaptations

Two different types of pattern-driven orchestration adaptations are envisioned: i) at design-time, and ii) at runtime.

When changes are imposed to the IoT service orchestration in order for an SPDI or QoS orchestration property to be valid at design-time, these changes must be communicated back where the description of the orchestration has been created (in this case, the Recipe Cooker component). The final destination of the changes is the end user that needs to confirm them. As soon as the said changes have been accepted by the user (or automatically accepted based on a set of predefined user preferences), the new, updated IoT service orchestration is deployed. Such a change could be the replacement of a component with another component (or a combination of components; e.g. when a device fails to comply to certain properties, such as because of an expired certification) or even the addition of an extra component into the orchestration to make sure that two services in sequence are interoperable (e.g., a semantic mediator; alterations at the output of the first service are undertaken by the extra component, thus constituting it compatible with the input of the second service).

On the other hand, when the imposed changes have to be done at runtime, there is no need to be communicated back to the end user. In this case, the best fitted change is chosen and the needed actions are taken, however the end user is not informed. For example, let's assume that the IoT service orchestration in question has a Camera component. If, for a reason, the Camera becomes unavailable, another component from the IoT repository with the same functionality and the same SPDI/QoS properties is selected to replace the one that has become unavailable. In that way the initial property of the whole orchestration, before the unavailability event, is not affected and continues to hold. Nevertheless, informing the backend orchestration

component (i.e. Recipe Cooker) may be needed in this case for visualization purposes, to ensure that the GUI depicts an up-to-date orchestration state.

6.3 Use-case driven Scenarios

In addition to the above generic example demonstrating the use of the integrated pattern-driven IoT orchestration approach, the subsections below present three scenarios focusing on the use cases considered in SEMIoTICS. These scenarios will form the basis for demonstrators of these building blocks, to be presented as the implementation matures.

6.3.1 USE CASE 1

In Figure 30 a topology is depicted that corresponds to use case 1, and more s. In this topology a Camera that is connected on a Raspberry Pi captures a video that is sent through a switch to a second Raspberry Pi. On the second Raspberry a video player is deployed, which depicts the captured video. On this orchestration patterns can be leveraged to monitor and ensure at the network level that certain QoS properties are maintained in terms of bandwidth to ensure the uninterrupted and smooth video playback. In that context, the application designer will be able to specify through the Recipe Cooker GUI the desired QoS properties, these will be translated to patterns and relayed to the corresponding Pattern Engine (in this case the Pattern Engine embedder into the SDN Controller) for monitoring and enforcement, per the process described in subsection 6.1.

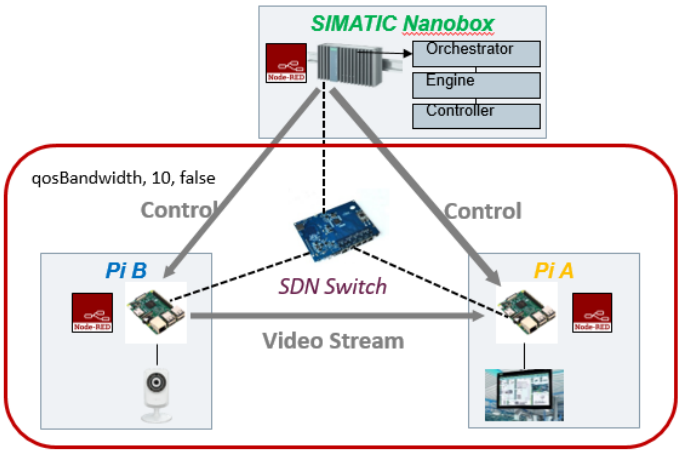


FIGURE 30: USE CASE 1 ORCHESTRATION

Such an orchestration could be described using the IoT pattern language as:

0. **ORCH "QoSBandwidth"**
1. Softwarecomponent("Camera"),
2. Property("Prop0", required, qosbandwidth, "11400000.0", in_processing, "Camera", true),
3. Softwarecomponent("VideoPlayer"),
4. Property("Prop1", required, qosbandwidth, "11400000.0", in_processing, "VideoPlayer", true),
5. Link("Link1", "Camera", "VideoPlayer"),
6. Property("Prop2", required, qosbandwidth, "11400000.0", in_transit, "Link1", true),
7. Sequence("Seq1", "Camera", "VideoPlayer", "Link1"),
8. Property("Prop3", required, qosbandwidth, "50000", end_to_end, "Seq1", false)
9. **Pattern rule: (PR1: Prop0, Prop1, Prop2 → Prop3)**

6.3.2 USE CASE 2

In Figure 31 an orchestration is depicted that corresponds to the use of a chain of network service functions used in the context use case 2 (healthcare scenario). Grouping them based on the different traffic types involved in the use case, five different service chains are designed:

- Chain 1 – Mobile Phone: Firewall -> DPI -> IDS -> Output
- Chain 2 – Robotic Rolator: Firewall -> IDS -> Load Balancer -> Output
- Chain 3 – Smart Home: Firewall -> IDS -> Output
- Chain 4 – Robot: Firewall -> Load Balancer -> Output
- Chain 5 - Malicious: Firewall -> Honeypot

In this case, patterns can be leveraged to reason about the different SPDI and QoS properties of the different chains, since each of the different security service functions (e.g., IDS) provide different guarantees and other functions (e.g., load balancer) provide QoS-related guarantess. For the sake of brevity, and without loss of generality, as these scenarios will be defined and demonstrated further in future deliverables when the implementation of all involved components is mature, we focus on the first chain: the mobile phone of a patient sends an output to the doctor, through three software components to guarantee the security property of their communication. The three software components that compose the said chain are: i) Firewall; ii) Data Packet Inspection and iii) Intrusion Detection System.

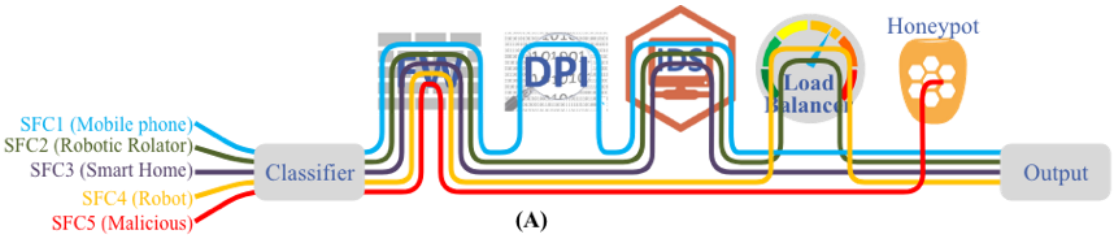


FIGURE 31: USE CASE 2 ORCHESTRATION

Such an orchestration could be described using the IoT pattern language as:

0. ORCH “Security”
1. Placeholder(“MobilePhone”, "macaddress", "activityaddress"),
2. Softwarecomponent ("Firewall"),
3. Link("Link1", “MobilePhone”, "Firewall"),
4. Sequence(“Seq1”, “MobilePhone”, "Firewall", “Link1”),
5. Softwarecomponent ("DPI"),
6. Link("Link2", "Seq1", "DPI"),
7. Sequence(“Seq2”, "Seq1", "DPI", “Link2”),
8. Softwarecomponent ("IDS"),
9. Link("Link3", "Seq2", "IDS"),
10. Sequence(“Seq3”, "Seq2", "IDS", “Link3”),
11. Placeholder ("Doctor"),
12. Link("Link4", "Seq3", "Doctor"),
13. Sequence(“Seq4”, "Seq3", "Doctor", “Link4”),
14. Property("Prop0", required, security, "1", end_to_end, "Seq4", false)

6.3.3 USE CASE 3

In Figure 32 a topology is depicted that corresponds to use case 2, and more specifically distributed vibration monitoring for earthquake detection. In this scenario, we consider that a Gateway is connected with two vibration sensors, which are identical. At any time, only one of them is up and running and the other one is idle, for redundancy in the monitoring. This redundant topology can be modelled and monitored as the Dependability property, through the appropriately defined pattern rule. Therefore, the infrastructure owner will be able to monitor in real-time the dependability status of her deployment, potentially triggering adaptations (e.g., the replacement of one sensor with its redundant mirror).

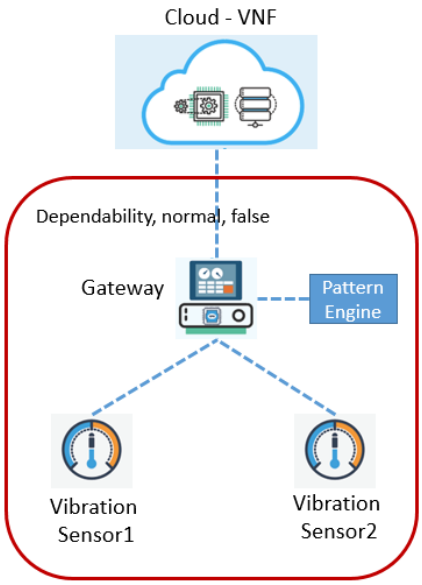


FIGURE 32: USE CASE 2 TOPOLOGY

Such an orchestration could be described using the IoT pattern language as:

0. **ORCH "Dependability"**
1. Iotsensor ("VibrationSensor1", "activityaddress", "activityport"),
2. Iotsensor ("VibrationSensor2", "activityaddress", "activityport"),
3. Iotgateway ("Gateway", "activityaddress", "activityport"),
4. Link("Link1", "VibrationSensor1", "Gateway"),
5. Link("Link2", "VibrationSensor2", "Gateway"),
6. Merge("Merge1", "VibrationSensor1", "VibrationSensor2", "Gateway", "Link1", "Link2"),
7. Property("Prop0", required, dependability, "1", end_to_end, "Merge1", false)

7 CONCLUSION

This deliverable, being the main output of Task 4.1 (“Architectural SPDI Patterns”) presented the requirements, design process and the first take on the specification of the SEMIoTICS SPDI pattern language.

Furthermore, discussion on the design process covered the service-orchestration level integration to be followed with the Recipes approach was discussed, guiding the implementation that will take place in the next stages of the project.

Finally, leveraging said language, a first set of SPDI patterns is also provided, defining them from both a formal, workflow-based perspective, as well as their machine-processable representation in Drools.

Through these efforts, the deliverable directly addresses the first key objective of WP4, which is to: *“Define a language for specifying machine interpretable SPDI patterns and develop patterns encoding horizontal and vertical ways of composing parts of IoT applications that can evidently guarantee SPDI properties across heterogeneous smart objects and components from all layers of the IoT application implementation stack.”*

While the work presented herein fully addresses the definition of the language, it only provides part of the patterns needed in SEMIoTICS. Thus, and in line with the work programme of the project, the next iteration of the deliverable, D4.8 – “SEMIoTICS SPDI Patterns (final)”, will provide the final version of the SEMIoTICS pattern language, with refinements that may be introduced as the implementation progresses. Moreover, the D4.8 will include the full set of SPDI patterns developed in the project (at least 36 in total), as well as the testing and validation results in applied usage scenarios. The latter will include the results of static verification of SPDI properties, as well as confirmation of the automated processing of said patterns.

REFERENCES

[1] A. Lukacs, "What Is Privacy ? the History and Definition of Privacy," p. 256–265, 2017.
[2] M. Dennedy et al., "The Privacy Engineer's Manifesto: Getting from Policy to Code to QA to Value," Apress, p. 400, 2014.
[3] E. Union, "Regulation 2016/679 of the European parliament and the Council of the European Union," Off. J. Eur. Communities, vol. 2014, p. 1–88, 1995.
[4] I. C. Office, "Anonymisation: managing data protection code of practice," Inf. Comm. Off., p. 106, 2012.
[5] P. Ohm, "Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization," UCLA Law Rev., vol. 57, p. 1701–1777, 2010.
[6] ISO/IEC 27018, 2014. [Online]. Available: http://www.iso27001security.com/html/27018.html.
[7] ISO/IEC 29100:2011, 1996. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:29100:ed-1:v1:en.
[8] CDC, "HIPAA Privacy Rule and Public Health Guidance from CDC and the U . S . Department depar," vol. 52, p. 24, 2003.
[9] W. Al-mawee, "Privacy and Security Issues in IoT Healthcare Applications for the Disabled Users a Survey," 2012.
[10] M. F. Mushtaq, S. Jamel, A. H. Disina, Z. A. Pindar, N. S. A. Shakir, and M. M. Deris, "A Survey on the Cryptographic Encryption Algorithms," Int. J. Adv. Comput. Sci. Appl., vol. 8, p. 333–344, 2017.
[11] W. Arthur, D. Challener, and K. Goldman, "A Practical Guide to TPM 2.0," Statew. Agric. L. Use Baseline, p. 375, 2015.
[12] J. Laprie, "Dependable computing and fault-tolerance," in Digest of Papers FTCS-15, 1985.
[13] A. Geraci, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, and F. Springsteel, "IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries," 1991.
[14] P. Park, P. Di Marco, C. Fischione, and K. Johansson, "Modeling and optimization of the ieee 802.15. 4 protocol for reliable and timely communications," Parallel and Distributed Systems, vol. 24, 2013.
[15] Jue Chen, Jinbang Chen, Fei Xu, Min Yin, and Wei Zhang, "When Software Defined Networks Meet Fault Tolerance: A Survey," Springer International Publishing, p. 351–368, 2015.
[16] AFUL G., "Definition: Interoperability.," 2018. [Online]. Available: http://interoperability-definition.info/en/. [Accessed 20 August 2018].
[17] Kiljander J. e. a., "Semantic interoperability architecture for pervasive computing and Internet of Things," IEEE Access, vol. 2, pp. 856-873, 2014.
[18] Haslhofer, B. and Klas, W., "A survey of techniques for achieving metadata interoperability," ACM Computing Surveys, vol. 42, pp. 1-37, 2010.
[19] Bröring, Arne & Schmid, Stefan & Schindhelm, Corina-Kim & Khelil, Abdelmajid & Kabisch, Sebastian & Kramer, Denis & Phuoc, Danh & Mitic, Jelena & Anicic, Darko & Teniente, Ernest. (2017). Enabling IoT Ecosystems through Platform Interoperability. IEEE Software. 34. 54-61. 10.1109/MS.2017.2. P. Barnaghi, W. Wang, C.A. Henson and K. Taylor, "Semantics for the Internet of Things: Early Progress and Back to the Future. International Journal on Semantic Web & Information Systems," International journal on Semantic Web and information systems, 2012.
[20] Barnaghi, Payam & Wang, Wei & Henson, Cory & TAYLOR, KERRY. (2012). Semantics for the Internet of Things: Early Progress and Back to the Future. International Journal on Semantic Web & Information Systems. 8. 10.4018/jswis.2012010101.
[21] T. Baker, M. Asim, H. Tawfik, B. Aldawsari and R. Buyya, "An energy-aware service composition algorithm for multiple cloud-based IoT applications," Journal of Network and Computer Applications, pp. 96-108, 2017.
[22] Z. Zhou, D. Zhao, L. Lui and P. C. K. Hung, "Energy-aware composition for wireless sensor networks as a service," Future Generation Computer Systems, pp. 299-310, 2018.
[23] O. Alsaryrah, I. Mashal and T. Chung, "Energy-Aware Services Composition for Internet of Things," in IEEE 4th World Forum on Internet of Things, Singapore, 2018.
[24] A.Urbieta, A. Gonzalez-Beltran, S. B. Mokhtar, M. A. Hossain and L. Capra, "Adaptive and context-aware service composition for IoT-based smart cities," Future Generation Computer Systems, pp. 262-274, 2017.

[25] L. Chen and C. Englund, "Choreographing Services for Smart Cities: Smart Traffic Demonstration," in IEEE 85th Vehicular Technology Conference (VTC Spring), Sydney, 2017.
[26] J. Seeger, R. A. Deshmukh and A. Broring, "Running Distributed and Dynamic IoT Choreographies," in Global IoT Summit (GIoTS), Bilbao, 2018.
[27] Forgy C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, vol. 19, p. 17–37, 1982.
[28] DENNING D. E., "A lattice model of secure information flow," Communications of the ACM, vol. 19, pp. 236-243, 1976.
[29] Zakinthinos, A., and Lee, E. S, "A general theory of security properties," in IEEE Symposium on Security and Privacy, 1997.
[30] Maidi M., "The common fragment of CTL and LTL.," in Foundations of Computer Science, 2000.
[31] Avizienis A., Laprie J-C., Randell B., "Fundamental Concepts of Dependability," in LAAS-CNRS, 2001.
[32] Laprie J. C. et al., "Dependability: Basic Concepts and Terminology," in Springer-Verlag, ISBN, 1992.
[33] Thuluva, A.S., A. Bröring, G.P. Medagoda Hettige Don, D. Anicic & J. Seeger, "Recipes for IoT Applications," in Proceedings of the 7th International Conference on the Internet of Things (IoT 2017), 2017.