ETSI GR ENI 016 V2.1.1 (2021-07)

Experiential Networked Intelligence (ENI); Functional Concepts for Modular System Operation

Reference

DGR/ENI-0026_Funct_Concept_MSO

Keywords

architecture, functional, software

Contents

Intellectual Property Rights

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Experiential Networked Intelligence (ENI).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The purpose of the present document is to provide information on software design principles for constructing modular systems to be applied to the ENI reference system architecture (and any other applicable ETSI reports or standards). This will cover common concepts such as Functional Block design, state machines, cognition, inferencing, along with communication between different domains.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI GS ENI 005 (V2.1.1): "Experiential Networked Intelligence (ENI); System Architecture".

[i.2] Gamma, E., Helm, R. Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Nov, 1994. ISBN 978-0201633610.

[i.3] Martin, R. C.: "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall, 2003 ISBN 978-0135974445.

[i.4] Rowley, J.: "The wisdom hierarchy: representations of the DIKW hierarchy", Journal of Information and Communication Science, 33(2): pp. 163-180.

[i.5] Meyer, B.: "Object-Oriented Software Construction", Prentice Hall PTR, 2nd edition ISBN 0-13-629155-4.

[i.6] Liskov, B.H. and Wing, J.M.: "Behavioral Notion of Subtyping", ACM Transactions on Programming Languages and Systems, November, 1994.

[i.7] Eugster, P. Th., Felber, P.A., Guerraoui, R., and Kermarrec, A.M.: "The Many Faces of Publish/Subscribe", ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114-131.

[i.8] Leymann, F.: "Loose Coupling and Architectural Implications", ESOCC 2016 keynote.

[i.9] Ingeno, J.: "Software Architect's Handbook", Packt Publishing, pg. 175, 2018. ISBN 178862406-8.

[i.10] ETSI GR ENI 018 (V1.1.1): "Artificial Intelligence Mechanisms Introduction to Artificial Intelligence Mechanisms for Modular Systems".

[i.11] The SysML profile is defined at: https://www.omgsysml.org/.

[i.12] Boyd, J. R.: "The Essence of Winning and Losing", June, 1995.

[i.13] Strassner, J., Agoulmine, N., Lehtihet, E.: "FOCALE - A Novel Autonomic Networking Architecture", ITSSA Journal 3(1), pp. 64-79, 2007.

[i.14] MEF 95: "MEF Policy Driven Orchestration:, J. Strassner, editor, April 2021.

[i.15] MEF 78.1: "MEF Technical Specification: MEF Core Model", Strassner, J., editor, July 2020.

NOTE: Available at https://www.mef.net/resources/mef-78-1-mef-core-model-mcm/.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms defined in ETSI GS ENI 005 [i.1] and the following apply:

abstraction: hiding of unnecessary details to focus on data and information that is relevant for defining a particular concept or process

architecture: set of rules and methods that describe the functionality, organization, and implementation of a system

- **software architecture:** high-level structure and organization of a software-based system. this includes the objects, their properties and methods, and relationships between objects

axiom: statement that is assumed to be true, in order to serve as a starting point for further reasoning

capability: type of metadata that represents a set of features that are available to be used from a managed entity

cognition: process of understanding data and information and producing new data, information, and knowledge

context: collection of measured and inferred knowledge that describe the environment in which an entity exists or has existed

design pattern: general, reusable solution in a given context to a commonly occurring software problem

NOTE: This type of design pattern is not an architecture and not even a finished design; rather, it describes how to build the elements of a solution that commonly occurs. It may be thought of as a reusable template.

domain: collection of Entities that share a common purpose

NOTE 1: Each constituent Entity in a Domain is both uniquely addressable and uniquely identifiable within that Domain. This is based on the definition of an MCMDomain in [i.15].

- **administrative domain:** Domain that employs a set of common administrative processes to manage the behaviour of its constituent Entities. This is based on the definition in [i.15].

- **management domain:** Domain that uses a set of common Policies to govern its constituent Entities

NOTE 2: A Management Domain refines the notion of a Domain by adding three important behavioral features:

1) it defines a set of administrators that govern the set of Entities that it contains;

2) it defines a set of applications that are responsible for different governance operations, such as monitoring, configuration, and so forth;

3) it defines a common set of management mechanisms, such as policy rules, that are used to govern the behavior of MCMManagedEntities contained in the MCMManagementDomain. This is based on the definition of an MCMDomain in [i.15].

entity: object in the environment being managed that has a set of unique characteristics and behaviour

NOTE: Objects are represented by classes in an information model.

formal: study of (typically linguistic) meaning of an object by constructing formal mathematical models of that object and its attributes and relationships

functional architecture: model of the architecture that defines the major functions of each module, and how each module interacts with each other

functional block: abstraction that defines a black box structural representation of the capabilities and functionality of a component or module, and its relationships with other functional blocks

graph: collection of nodes, where some subset of the nodes is connected

NOTE: Visually, a node is a "point" and a connection is a "line", called an "edge". For the purposes of ENI, any graph may be directed, weighted, or both.

knowledge: analysis of data and information, resulting in an understanding of what the data and information mean

NOTE: Knowledge represents a set of patterns that are used to explain, as well as predict, what has happened, is happening, or is possible to happen in the future; it is based on acquisition of data, information, and skills through experience and education.

- **inferred knowledge:** knowledge that was created based on reasoning, using evidence provided
- **measured knowledge:** knowledge that has resulted from the analysis of data and information that was measured or reported
- **propositional knowledge:** knowledge of a proposition, along with a set of conditions that are individually necessary and jointly sufficient to prove (or disprove) the proposition

learning: process that acquires new knowledge and/or updates existing knowledge to optimize a function using sample observations

logic: formal or informal language that evaluates a conclusion based on a set of premises

- **first-order logic:** extension of propositional logic to include predicates and quantification
- **modal logic:** representing, using mathematical formalisms, expressions involving necessity and possibility
- **propositional logic:** manipulation of a set of propositions, possibly with logical connectives, to prove or disprove a conclusion

NOTE: Propositional logic does not deal with logical relationships and properties that involve the parts of a statement smaller than the statement itself. It also called sentential logic, zeroth-order logic, and propositional (or sentential) calculus.

model: representation of the entities of a system, including their relationships and dependencies, using an established set of rules and concepts

- **information model:** representation of concepts of interest to an environment in a form that is independent of data repository, data definition language, query language, implementation language, and protocol

NOTE: This definition is taken from [i.15].

ontology (for ENI): language, consisting of a vocabulary and a set of primitives, that enable the semantic characteristics of a domain to be modelled

policy: set of rules that is used to manage and control the changing and/or maintaining of the state of one or more managed objects

NOTE: This is defined in [i.13] and [i.14].

repository: centralized location of a set of storage devices that enable different functional blocks to store and retrieve information

- **active repository:** repository that pre- and/or post-processes information that is stored or retrieved

NOTE: It may contain dedicated (typically internal) Reference Points that provide the loading, activation, deactivation, and unloading of specialized functions that change the pre- and/or post-processing functionality according to the needs of the application.

- **passive repository:** repository that stores or retrieves information without pre- or post-processing

semantics: study of the meaning of something (e.g. a sentence or a relationship in a model)

situation: set of circumstances and conditions at a given time that may influence decision-making

telemetry: automated process of recording and transmitting data to receiving equipment for monitoring purposes

NOTE: The process is typically automated, and the data transfer may include wireless, cellular, optical, and other mechanisms.

theorem: set of statements that has been mathematically proven to be true, based on a set of axioms and/or (previously proven) theorems

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AD	Access Device
AG	Aggregation Device
API	Application Programming Interface
CR	Core Router
DIKW	Data-Information-Knowledge-Wisdom
IP	Internet Protocol
OODA	Observe-Orient-Decide-Act
OWL	Web Ontology Language

4 Introduction

4.1 Fundamental Software Design Principles

4.1.1 Introduction

The purpose of the clauses below is to describe some key software architecture principles that were used in the design of the ENI System Architecture document. Each of these principles is based on designing modular software components and systems, and can be used for general purposes. In each of the following, the term "unit" means class, component, or module.

4.1.2 Information Hiding and Encapsulation

Information hiding is a design principle that states that if one unit does not need to know how another unit works, then it does not need do so. This ensures that each unit can be developed independently. This principle applies to classes, components, and modules (hereafter referred to as "units").

Put another way, information hiding mandates *loose coupling* (see clause 4.1.8). If there is a possibility that the functionality of the unit will change, then that unit needs to be separated from other units.

Encapsulation is *not* the same as information hiding. Encapsulation is an implementation mechanism that defines the boundaries of a unit. For example, a class is a collection of attributes and methods that are part of a single object. Put another way, encapsulation prevents the direct access to a unit's implementation details by a client.

Continuing the example of a class, it hides information by hiding implementation detail, and it encapsulates the object by combing code and data. Information hiding prevents clients of the class from knowing too much about the details of the class, and reduces coupling (see clause 4.1.8). Encapsulation prevents clients from accessing the implementation of the class, and increases cohesion (see clause 4.1.9).

4.1.3 Single Responsibility Principle

The original definition of this principle comes from [i.3], and is stated as follows:

A class needs to have only one reason to change.

In this definition, "reason to change" is what the unit is designed to do. This does not mean that a unit consists of one attribute or method; rather, it means that the set of all attributes and methods in a unit are related to a single responsibility. In practical terms, this means that different functions that have different purposes (e.g. analysis and printing data), and therefore, need to be split into separate units. This also increases the readability, testability, and maintenance of the unit.

4.1.4 Open-Closed Principle

This was first defined in [i.5] as follows:

"Software entities (classes, modules, functions, etc.) need to be open for extension, but closed for modification."

This principle is best illustrated by an example. Consider the action of writing to an entity, such as a disk file. This principle states that the write action can apply to any device (e.g. a printer or a screen, which makes it *open for extension*) without having to change the implementation to be device-specific (which makes it *closed for modification*).

Note that this can be implemented in two different ways: via inheritance and via interfaces. The problem with using inheritance is that subclasses are tightly coupled to their superclasses if they depend on the implementation details of their superclasses. In contrast, interfaces introduce an additional level of abstraction, which enables loose coupling. Interfaces can be changed without effecting the implementation that uses them. Furthermore, the interfaces of a unit are independent of each other. The most common way to do this effectively is to use composition.

4.1.5 Liskov Substitution Principle

This was first defined in [i.6] as follows:

"If an object X is a subclass of an object Y, then objects of type Y may be replaced with objects of type S without altering the behaviour of the program".

This principle is also called (strong) behavioural subtyping, because it guarantees semantic interoperability between object types in a system. This is often enforced using Design by Contract (see clause 4.1.11). In particular, Liskov Substitution requires the following restrictions to be true when a subclass is substituted for its superclass:

- Pre-conditions cannot be strengthened in the subclass (see clause 4.1.11).
- Post-conditions cannot be weakened in the subclass (see clause 4.1.11).
- Invariants are preserved in the subclass (see clause 4.1.11).
- New exceptions cannot be generated by methods in the subclass unless they are subtypes of exceptions that are generated by methods of the superclass.
- Method return types in the subclass are preserved (e.g. the return type of the subclass is a subtype of the return type of the superclass).

Method parameter types in the subclass is reversed (e.g. the parameter types of the superclass are subtypes of the parameter types of the subclass).

Liskov substitution provides more robust and modular unit designs.

4.1.6 Interface Segregation Principle

This was first defined in [i.3] as follows:

"Clients have a duty to not be forced to depend upon interfaces that they do not use".

In other words, instead of having a small number of interfaces that have multiple responsibilities, a modular unit designed using this principle will have a large number of client-specific interfaces, where each client-specific interface has a single responsibility. Separate clients need separate interfaces. This increases the cohesion between the interfaces of a given unit.

The goal of this design principle is to reduce side effects caused by changing a unit's implementation. It is similar to the Single Responsibility Principle (see clause 4.1.3), in that by splitting software into multiple independent parts, it enables each part to evolve independently. For example, if unit A depends on unit B at *compile* time, but not at *run time*, then changes to unit B will force unit A to change. This is especially important for statically typed languages, like Java, C++, and C#. The Adaptor software pattern [i.2] is an example of a design pattern that can be used to support interface segregation.

4.1.7 Dependency Inversion Principle

This principle is based on the Open-Closed Principle and the Liskov Substitution Principle (see clauses 4.1.3 and 4.1.4, respectively), and was defined in [i.3]. It enables higher-level units to be loosely coupled to any lower-level units that depend on them. Specifically, the principle states:

High-level modules have a duty to not depend on low-level modules. Both have a duty to depend on abstractions (e.g. interfaces).
Abstractions have a duty to not depend on details. Details (concrete implementations) have a duty to depend on abstractions.

The essence of this principle is that both high-level and low-level modules depend on the abstraction. Put another way, this principle splits the dependency between the high-level and low-level modules by introducing an abstraction between them.

High-level modules, which provide complex logic, have a duty to be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, an abstraction that decouples the high-level and low-level modules from each other is required. For example, if this principle is not followed, then high-level business logic will depend on low-level implementation details.

4.1.8 Loose Coupling

Coupling [i.8] refers to how inextricably linked different aspects of an application are. High coupling needs to be avoided between units, because this forces the evolution of each unit to depend on each other. In contrast, low coupling ought to be used whenever possible, as this enables each unit to evolve independently. This also enables each unit to be more easily reused, since it has fewer dependencies to encumber its usage. Hence, units in a loosely coupled system may be replaced with different implementations that provide the same services.

The concept of loose coupling may be applied to classes, interfaces, services, and data. The Enterprise Service Bus, and more specifically, ENI's Semantic Bus, are designed to promote loose coupling.

Loose coupling is typically achieved in APIs by using standard datatypes in parameters, and ensuring that a standard format is used in communication between units.

Loose coupling is associated with high cohesion, and vice versa.

4.1.9 High Cohesion

Cohesion [i.9] refers to how closely related the contents of a particular unit are. It answers the question "do these elements inside a unit belong together". It can be thought of as a measure of how well the elements of a unit serve the purpose of a unit.

Cohesion is increased if:

- the elements of a unit are abstracted and serve the same purpose
- each method performs as few activities that are strongly related as possible
- each method avoids using unrelated data

High cohesion provides:

- increased reusability, because the responsibility of the unit more closely matches its attributes and operations
- increased robustness, due to reduced complexity of each unit
- increased system maintainability and understandability, because:
    - changes in a system affect fewer units
    - changes in one unit require fewer changes in other units

High cohesion is associated with loose coupling, and vice versa.

4.1.10 Design by Contract

Design by Contract [i.5] states that every software component should have a formal interface specification that can be verified and tested. Its simplest form consists of the following five principles:

1) **Pre-condition** is an assertion that is always true prior to the execution of an operation
2) **Post-condition** is an assertion that is always true after the execution of an operation
3) **Invariant** is an assertion that is always true during the execution of an operation
4) **Acceptable values and types** for inputs and outputs (i.e. return types)
5) **Definition of side effects, errors and exceptions** that can occur as a result of an operation

Regarding subclasses:

- Pre-conditions cannot be strengthened in a subclass
- Post-conditions cannot be weakened in a subclass
- Invariants are preserved in the subclass

Design by Contract has been incorporated into multiple software programming languages, and enables interfaces to have formal semantics defined. This ensures that interfaces between different units can interact and exchange information that preserves their semantics.

4.1.11 Summary of Design Principles

Table 1 summarizes each of the nine design principles explained in the previous clauses.

Table 1: Software Design Principle Summary

Design Principle	Clause	Explanation
Information Hiding	4.1.2	Different classes, components, and modules can be developed independently and do not have to know how other software entities work.
Encapsulation	4.1.2	Prevents the direct access of the implementation details of a software entity.
Single Responsibility Principle	4.1.3	A class only has one responsibility, and hence, one reason to change.
Open-Closed Principle	4.1.4	Software entities can be extended for other similar uses without having to change its implementation.

Design Principle	Clause	Explanation
Liskov Substitution Principle	4.1.5	Subclasses can be substituted for superclasses without affecting the behaviour of the system.
Interface Segregation Principle	4.1.6	Clients do not depend on interfaces that they do not use (i.e. it is better to have more client-specific interfaces that a lesser number of generic interfaces).
Dependency Inversion Principle	4.1.7	High-level modules do not depend on low-level modules. Modules depend on abstractions. Abstractions do not depend on implementation details. Implementation details depend on abstractions.
Loose Coupling	4.1.8	Each element of a loosely coupled system depends on as little knowledge as possible of other elements (preferably none).
High Cohesion	4.1.9	Each element of a highly cohesive system work together to achieve the same purpose, and do not depend on data or information that is not required for their task.
Design By Contract	4.1.10	Software entities should be designed using formal specifications that use pre- and post-conditions, as well as invariants, to specify the behaviour of the entity.

4.2 Functional Blocks

4.2.1 Introduction

A Functional Block is a system and software engineering abstraction that defines a black box structural representation of the capabilities and functionality of a component or module. A Functional Architecture is a model of the architecture that defines the major functions of each module, and how each module interacts with each other. A Functional Architecture therefore describes how the functions performed by the Functional Blocks operate together to achieve the goals of the system.

4.2.2 Functional Design

Functional design uses Functional Blocks to decompose the functionality of a system into a set of sub-functions. Each Functional Block corresponds to a distinct task performed by the system. This is a *recursive* process that continues until a level of abstraction is achieved where it does not make sense to continue the decomposition process.

This approach enables the inputs and outputs to each Functional Block to be specified via a *transfer function* without specifying the implementation details of a Functional Block. Functional design enables hierarchies of Functional Blocks to be specified to represent the decomposition of functions into sub-functions.

4.2.3 Functional Block Diagrams

Functional Block diagrams are optionally used to pictorially represent the flow of information and control between Functional Blocks and their relationships. SysML [i.11] is proposed for specifying Functional Blocks, as it is able to unambiguously represent different types of control flow; SysML is able to precisely specify the semantics and behaviour of a Functional Block.

4.2.4 Usage

A Functional Block is an abstraction of the characteristics and behaviour of a portion of a system. A Functional Block is completely self-contained; this means that it is free of side-effects. This further implies that a Functional Block does not rely on global variables, I/O, or system-wide communication paths.

A Functional Block is a modular unit that describes a portion of a system. It defines a collection of features to describe an element of interest. The specific kinds of Functional Blocks, the kinds of connections between them, and the way these elements combine to define the total system is defined according to the goals of a particular system model. Functional Blocks interact by one or more means, such as software operations, discrete state transitions, flows of inputs and outputs, or continuous interactions.

If the inputs and outputs of a Functional Block are externally visible, they are defined as External Reference Points.

A Functional Block optionally contains other Functional Blocks; in addition, it communicates with and asks services of different Functional Blocks.

A Functional Block is formally a a type of managed entity. Capabilities can be defined for Functional Blocks as well. A Capability provides information about the functionality of a managed entity that enables management entities to decide whether that managed entity is useful for a given task.

4.3 State and State Machines

The efficacy of the ENI System depends on the amount of state information available to it. In order for the ENI System to make a decision based on end-to-end goals, it is proposed that the ENI System possesses knowledge of the current state and the current goals of the systems that it is providing recommendations and/or management decisions to. Hence, state information is focused on the state of the Assisted System and its environment, but optionally also includes the state of affected Functional Blocks in the ENI System. For a large, complex system, it is unlikely that the ENI System knows the state of all elements of the Assisted System. This is because of the number of dynamically changing system elements, as well as the relatively high cost of communication. The level of compliance to ENI, as determined also by the support level for the ENI Reference Points and their proposed grouping, defined herewith, determines the functionality and optimizations provided by the ENI System. This is why the ENI System is based on a cognition framework; this framework allows the ENI System to dynamically adjust its knowledge, and infer new knowledge, as required. Such knowledge typically affects, or is reflected in, state. For example, new knowledge can direct the change of state of the Assisted System. Similarly, system operation can be associated with a current (or past) state of the Assisted System; this enables such knowledge to be optionally added to the knowledge base of the Assisted System. The ENI System is therefore a *distributed* system, in which its Functional Blocks share information and work together, collectively, to manage the state of the Assisted System.

4.4 Data, Information, Knowledge, and Wisdom

4.4.1 Introduction

There are four types of knowledge that need to be managed by ENI. In general, these types of knowledge represent a progression of increased understanding and semantics (from Data to Wisdom) [i.1] and [i.6]. Formal semantics, which uses mathematical models, is used to help understanding of data and information that is textual in nature.

Figure 1 presents a variation of the DIKW (Data-Information-Knowledge-Wisdom) model [i.1] and [i.6] that is used in a variety of disciplines. It has been adapted to more closely fit the operation of ENI. In Figure 1:

- Data corresponds to Sense.
- Information corresponds to Perceive.
- Knowledge corresponds to Learn.
- Wisdom corresponds to Adapt.

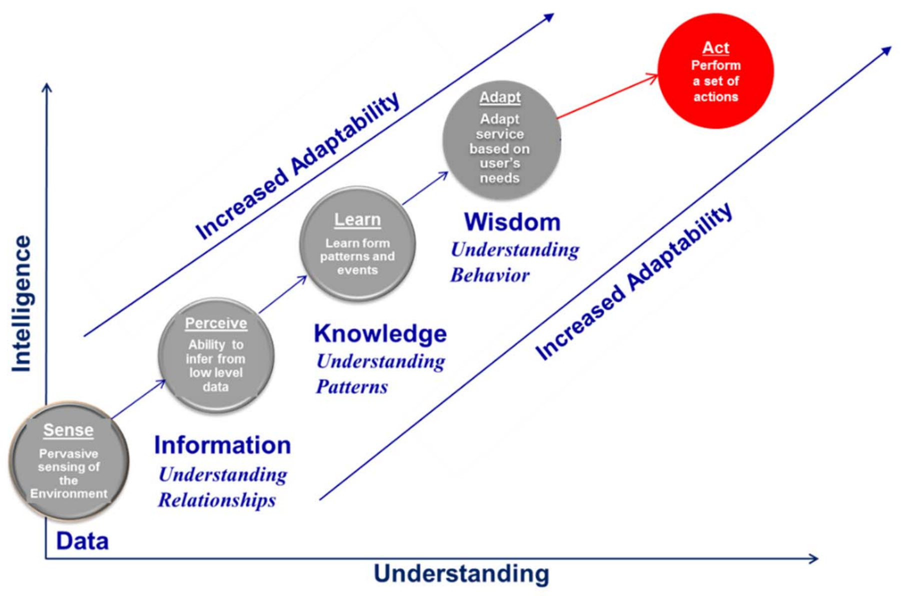

Figure 1: Understanding and Decision-Making

The Sense level gathers data and information from the system and environment. It is often directed to gather specific data by the management system (e.g. ENI). This level is made up of only measured knowledge (see clause 4.4.6). Data is thus defined as discrete unprocessed facts or observations, and therefore, have no meaning because of the lack of context and interpretation. This level is used to produce classes and class attributes.

The Perceive level uses a variety of techniques, including statistics and inferencing, to better understand the data (e.g. was the data complete, how likely was it that the data contain errors). This level also enables relationships between different concepts to be defined and discovered. This level, as well as subsequent levels, consists of measured and inferred knowledge (see clause 4.4.6). Information is data that is processed so that the resulting information now is relevant for a specific set of contexts or purposes. This level produces classes, class attributes, and relationships between classes.

The Learn level uses a variety of techniques to discover and learn patterns that cause data to be generated. Patterns are identified by comparing new information with existing knowledge. These patterns identify concepts, parts of concept (e.g. attributes of a class), and behaviour (e.g. methods of a class, as well as relationships between classes). Knowledge is characterized by justifiable belief, often in the form of formal mathematical proofs. This level enables a set of actions to be formulated and enacted, typically in response to a set of stimuli.

The Wisdom level applies current and new knowledge to understand why and how behaviours occurred, and supports predicting new behaviour. It is viewed as using knowledge to produce the right set of actions in a given context. This level enables plans of actions to be defined to achieve long-term goals. It also enables information and knowledge to be adapted to suit changing context, and enables existing knowledge to be corrected and augmented as necessary.

The first four levels (Sense, Perceive, Learn, and Adapt) correspond to the Observe, Orient, and Decide levels of the OODA control Loop. They culminate in a set of Actions (the Act part of OODA) that are executed [i.12]. This is discussed further in ETSI GS ENI 005 [i.1], clauses 5.7 and 6.3.4.7.

4.4.2 Data

Data consists of a sequence of symbols that are collected for a purpose (e.g. monitoring the bandwidth of a connection). Data represents a fact or statement of event without relation to other things. Data, by itself, has no meaning; it requires interpretation in a specific context to become information.. Examples of the interpretation context include the creator of the data, the time of the creation and reception of the data, and metadata.

4.4.3 Information

Information is data that have been processed in a specific context to give it meaning. Information embodies the understanding of a relationship of some sort, possibly cause and effect. Information is used to increase knowledge. More formally, information is defined as a set of data, where each element in each data sequence is well-formed (i.e. syntactically correct). Furthermore, each data sequence is meaningful (i.e. its semantics complies with the translation of data in a given context). Information provides answers to the fundamental questions "who", "what", "where", and "when".

4.4.4 Knowledge

Knowledge is the analysis of data and information, resulting in an understanding of what the data and information mean. Knowledge represents a set of patterns that are used to explain, as well as predict, what has happened, is happening, or is possible to happen in the future; it is based on acquisition of data, information, and skills through experience and education.

Knowledge is more than just collecting data and information, since merely collecting information does not allow its integration and making decisions. Knowledge is specific to the context(s) and situation(s) in which it was created.

There are many types of knowledge; however, in the context of ENI, ENI is typically concerned with what is called "propositional knowledge". More specifically, propositional knowledge is knowledge of a proposition, along with a set of conditions that are individually necessary and jointly sufficient to prove (or disprove) the propositional knowledge. This type of knowledge is implemented in formal logic systems. Knowledge answers the question "how".

Decision-making requires understanding, which includes basic cognition (i.e. the ability to use probability, as well as to synthesise new knowledge from the previously held knowledge). Knowledge enables the system to learn new information.

4.4.5 Wisdom

Wisdom is the ability to understand the current context, any regulations and restrictions (e.g. legal, ethical, temporal) that apply, and to reason and act using knowledge and experience. It uses the fundamental principles of information and knowledge to better understand the environment, its goals, and how best to achieve those goals. For example, it provides an explanation of why data, information, and knowledge occur.

Wisdom, in the specific context of ENI, is the collection of ideas and understandings that an entity possesses that are used to take effective action to achieve the goals of an entity.

Wisdom answers the question "why".

4.4.6 Measured vs. Inferred Knowledge

Two important concepts for ENI are the concepts of measured and inferred knowledge.

Measured knowledge is defined as knowledge that has resulted from the analysis of data and information that was measured or reported. For example, data sources could include metrics and statistics that were ingested.

In contrast, inferred knowledge is defined as knowledge that was created based on reasoning, using evidence provided. For example, observations could show that performance is decreasing; a system could then infer that an associated service level agreement is now at risk.

4.5 Logic and Inferencing

4.5.1 Introduction

This clause defines and contrasts the concepts of logical reasoning and inferencing. Reasoning uses an appropriate knowledge representation to make decisions. Logic is defined as the analysis of a set of statements that are derived from a formal language that evaluates a conclusion based on a set of premises. The present document assumes that reasoning is done using logic and/or inference, which are described in the following clauses.

4.5.2 Logical Reasoning

Logical reasoning is defined as the use of a formal logic, such as first-order logic, to derive a result (e.g. proving a hypothesis). More formally, logical reasoning is the study of the criteria used in evaluating logical arguments as well as inferences.

A logical argument is a collection of statements or propositions, some of which are intended to provide support or evidence in favour of one of the others. This does not require any artificial intelligence. Rather, it uses logic to mathematically prove a conclusion based on a set of premises provided. A premise is a statement that is designed to support a conclusion (though in reality it may or may not do so), and is either true or false.

Logical arguments are typically divided into three categories: propositional logic, first-order logic, and higher-order logics.

Propositional logic, also called propositional calculus, sentential calculus, or zeroth-order logic, uses Boolean predicates (i.e. statements that can be true or false) and relations between those predicates. Propositional logic allows predicates to be combined using logical connectives (e.g. AND (conjunction), OR (disjunction), and NOT (negation)). Propositional logic does not deal with non-logical objects, predicates about them, or quantifiers. Hence, it can only represent facts that are either true or false. However, both first-order logic and higher-order logics include all of the features of propositional logic.

First-order logic, also known as predicate logic, is a formal logic that uses quantified variables over non-logical objects, and allows the use of sentences that contain variables. Each sentence has a subject and a predicate, the latter of which is a relation that binds two elements in the predicate together. In first-order logic, a predicate can only refer to a single subject. However, first-order logic can represent objects, relationships between objects, and functions. For example, the premise "Socrates is a man" can be written as "there exists x such that x is Socrates and x is a man", where "x" is a variable and "there exists" is a quantifier. If the sentence "Hippocrates is a man" is added as a premise, propositional logic considers this as a separate, unrelated premise. However, first-order logic enables a common predicate, "is a man", to be defined; this enables the common structure of the two premises to be related to each other. In addition, first-order logic enables quantifiers to be applied to variables in sentences. For example, if the sentences are changed to "If Socrates is a man, then Socrates is a philosopher" and "If Plato is a man, then Plato is a philosopher", then each sentence has the common form "if x is a man, then x is a philosopher". Then, the quantifier "for all" (or "for every") can be used to express the concept that "if x is a man, then x is a philosopher is true *for all* values of x.

First-order logic is undecidable. This means that a decision algorithm for determines whether an arbitrary statement is logically valid or not does not exist. This has led to the definition of many subsets of first-order logic that are decidable; most notable among these are Description Logics. Two such examples are OWL DL and OWL QL.

Higher-order logics also exist. For example, second-order logic extends first-order logic by enabling variables to be quantified over relations, sets, functions, and other variables. The higher the order of a logic, the more expressive it is, but the harder it is to compute decidability.

In contrast, an inference is a process of reasoning in which a new belief is formed on the basis of or in virtue of evidence or proof supposedly provided by other beliefs (see clause 4.5.3).

4.5.3 Inferencing

Inferencing is a process, or set of processes, that produces knowledge by reasoning about evidence produced. There are three different types of inferencing:

- Deduction: deriving conclusions, through reasoning, from a set of premises (that are known or assumed to be true) to reach a (logically certain and consistent) conclusion.

- Induction: deriving conclusions, through reasoning, from a set of premises that supply some evidence for the conclusion. Critically, there is a possibility that an induced conclusion turns out to be false, even if all of its premises are true; this is not possible if the conclusion is deduced.

- Abduction: deriving the simplest and most plausible conclusion from analysing a set of observations. The conclusion is not certain.

Given a set of premises P and a conclusion C, then:

- Deduction derives C from P using formal logic (i.e. C is a logical consequence of P).

- Induction allows surmising that C follows from P, but does not ensure its validity.

- Abduction allows surmising P as an explanation of C: there is a possibility that P provides good reason to accept C, but does not guarantee its validity.

All three types of inferencing are used by ENI. Deduction and induction are typically used with logic programming systems. Inductive logic programming is often used for machine learning and probabilistic calculations. Abduction is particularly effective in fault detection and identification.

4.6 Data Acquisition

4.6.1 Introduction

Data can be acquired using different approaches. The following clauses define telemetry and non-telemetry based approaches for ingesting data and information.

4.6.2 Telemetry Approaches

- There are a number of telemetry approaches that can be used for monitoring network operation and management. They can be divided into pull techniques, which polls data from network devices at specified time intervals vs. push techniques, which subscribes to different types of data and collects it when available. Network telemetry can be performed on the data, control, or management planes:Data plane provides flow and packet quality of service, quality of experience, traffic and queue statistics. It takes three primary forms:

    - In-band telemetry is carried directly in packets.

    - Out-of-band telemetry is exported directly without modifying packets.

    - Hybrid telemetry combines the two (e.g. instructions for what telemetry is to be exported is carried in the packets).

    - Customized telemetry assumes the support of a programmable data plane to enable the data collected to be customized to fit the specific needs of an application.

- Control plane provides statistical information on any control and signalling protocols used, so as to ensure their proper operation.

- Management plane provides configuration and operational state information (e.g. network state, errors, statistics, and performance data).

Network telemetry can be requested using either direct commands or indirectly, based on events that occur outside of the network.

4.6.3 Non-Telemetry Approaches

An ENI System gathers data using both telemetry and other mechanisms. There are several different mechanisms for collecting information that do not use telemetry. These include:

- Network monitoring protocols - a set of protocols that communicate monitoring data between a network node and an external server.

- Network programming - a mechanism that enables a network operator or application to encode a sequence of instructions to instruct the node to provide data.
- Data-driven collection - a mechanism where the type of data is determined by the data being analysed.
- Decision-based collection - a mechanism used where an ENI System decides to monitor data based on changes in context and/or situation.

The decision to use a specific protocol and collection mechanism is performed in the Analysis portion of the ENI Functional Architecture (see ETSI GS ENI 005 [i.1], clause 4.5.3.3). The decision is defined as a set of policies and transmitted to the API Broker (see ETSI GS ENI 005 [i.1], clauses 4.4.2 and 4.5.2). In all cases, the ENI System collects data using an API Broker.

4.6.4 Use of Policies to Change Data Acquisition

It is recommended that telemetry is acquired using a standard and consistent mechanism. For example, the ENI System Architecture uses policies to direct the gathering of telemetry information. The advantage of this approach is that policies can define what data and information to collect when and where in the network, and includes metadata that describes and/or prescribes additional details about the collection.

For example, the ENI System Architecture uses a set of adaptive and cognitive closed control loops to adapt data and information acquisition according to changes in user needs, business goals, and environmental conditions. The use of policies standardizes these requests, making it simpler for the API Broker to map ENI Policies to an appropriate API supported by the external system.

4.7 Communication

4.7.1 Introduction

This clause defines and compares different types of communication mechanisms.

4.7.2 Centralized

Centralized communication uses a common mechanism, such as a bus, to enable all nodes to connect to and communicate with a single server (or set of servers). This centralized server typically stores all communications in one place, including all the necessary documentation and information.

4.7.3 Decentralised

Decentralised communication views each node as independent. Therefore, each node can decide which nodes it wants to communicate with. The degree of decentralisation refers to how many nodes are independent of each other, and can range from 2 to all. This also enables groups of nodes to be formed for processing particular types of data (in this case, one node is typically designated to communicate with other external nodes).

4.7.4 Comparison of Different Communication Styles

Centralized and Decentralised communication each have advantages and disadvantages.

Centralized communication is easier to manage and implement, and provides better consistency, efficiency, and affordability. Since all data goes through a central server, it is easier to collect and track data. However, it is also a single point of failure (e.g. if the central server crashes, all nodes become unavailable). Care needs to be taken to avoid performance and I/O bottlenecks.

Decentralised communication providers greater redundancy, fault tolerance, security and privacy. It provides greater scalability, and does not have a single point of failure. However, Decentralised systems are harder to implement, and may require complex routing to avoid problems such as latency. It also requires more complex management.

4.8 Domains

4.8.1 Introduction

A Domain is a mechanism used to define a collection of different Entities that share a common purpose. In addition, each constituent Entity in a Domain is both uniquely addressable and uniquely identifiable within that Domain. This is based on the definition of an MCMDomain in MEF 78.1 [i.15]. ENI uses two principal types of Domains: administrative Domains and Management Domains.

4.8.2 Administrative Domains

An Administrative Domain is a Domain that employs a set of common administrative processes to manage the behaviour of its constituent Entities.

4.8.3 Management Domains

A Management Domain is an Administrative Domain that uses a set of common Policies to govern its constituent Entities. The latter definition is based on the definition in MEF 78.1 [i.15].

In particular, a Management Domain adds three important behavioral features:

1) it defines a set of administrators that govern the Entities that it contains;

2) it defines a set of applications that are responsible for different governance operations, such as monitoring, configuration, and so forth;

3) it defines a common set of management mechanisms, such as policy rules, that are used to govern the behavior of Entities contained in the Management Domain.

4.8.4 Domain Organization

4.8.4.1 Centralized

A centralized arrangement of Domains means that all client users and applications are attached directly to a central computer. This is shown in Figure 2.

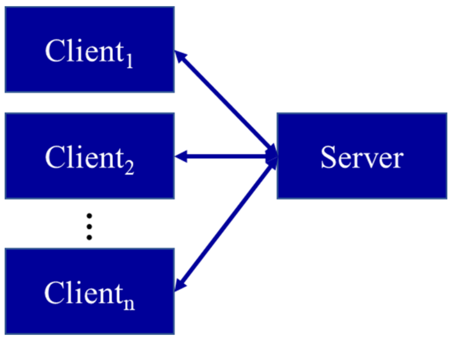

Figure 2: A Centralized Domain System

All operations are done by the constituent components contained in the centralized Domain. The advantages of using a centralized domain system include management simplicity and greater security in some situations (because all operations are controlled in a centralized location). Disadvantages include the centralized domain becoming a single point of failure and the inability to perform specialized processing for particular clients.

4.8.4.2 Hierarchical

A hierarchy of items arranges each item in terms of a set of factors. Examples of these factors include position (e.g. above, below, or at the same level as another item), containment, cost, response time, and semantics. The hierarchy forms a tree structure. This is shown in Figure 3.

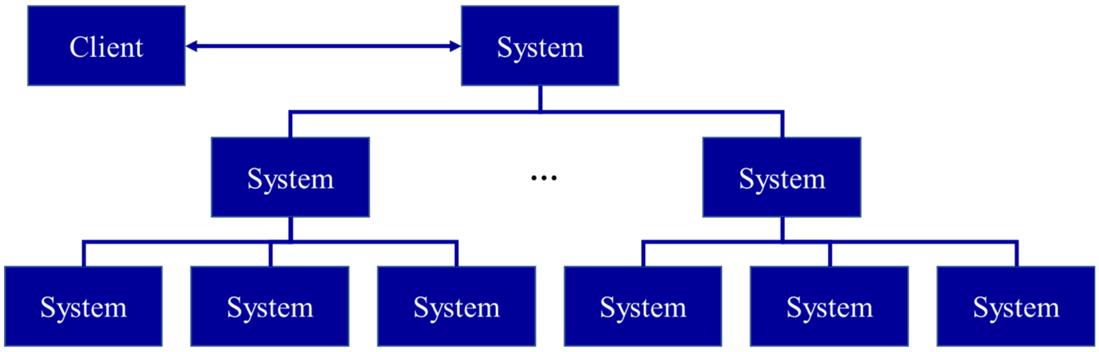

Figure 3: A Hierarchical Domain System

The position of a Domain in a hierarchy typically defines the containment relationships of that Domain (i.e. is that Domain contained in another Domain, and does that Domain contain other Domains). Following the work done in MEF 78.1 [i.15], a Domain in a hierarchy inherits the behaviour defined by its parent (i.e. containing) Domain. See requirements for hierarchical Domains in ETSI GS ENI 005 [i.1], clause 5.2 (i.e. [FAR4]).

4.8.4.3 Distributed

A distributed set of Domains is a set of Domains whose components work together to achieve a set of common goals. Each goal is typically divided into one or more tasks, which are able to be processed by other components in other domains. One or more communication mechanisms are used to coordinate actions between the Domains.

A common example of a distributed system is in networking. Three different domains, commonly called core, aggregation, and access, are used to route and forward traffic. The Access layer is the level where host computers are connected to the network. The Aggregation layer controls the sessions entering and leaving the system. This enables multiple uplinks from access devices to be aggregated into higher speed links. The Core layer connects all Aggregation layer devices and reliably and quickly switches and routes the traffic. This is shown in Figure 4. In Figure 4, AD, AG, and CR stand for Access Device, Aggregation Device and Core Router, respectively.

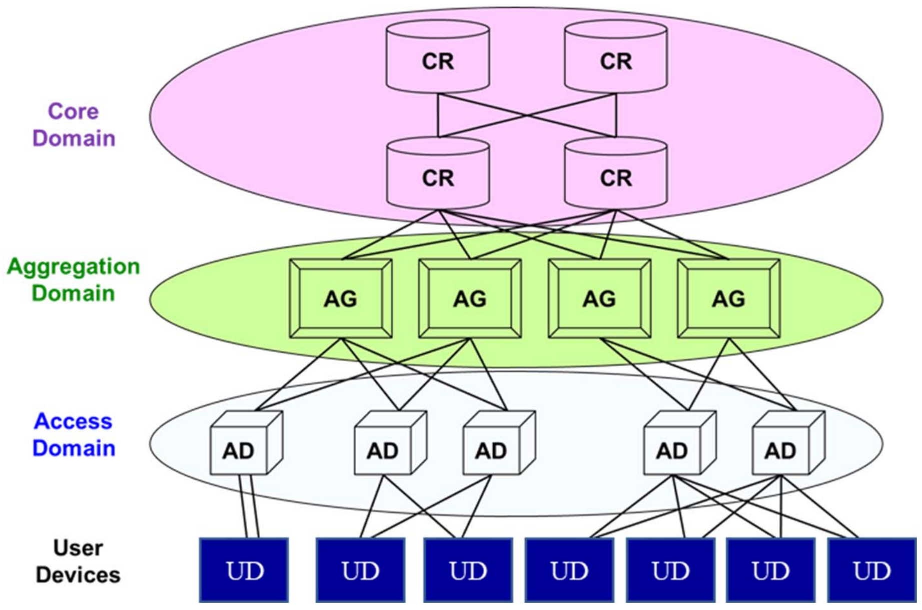

Figure 4: A Distributed Domain System

Distributed Domains are used to distribute tasks to different Domains that have functionality most suited to accomplishing the goals required. If a goal can be subdivided into multiple tasks, then each such task can run concurrently.

4.8.4.4 Federated

A federated set of Domains is a set of Domains that use formal agreements to govern their interaction and behaviour. This includes rules to admit new members of the federation, as well as rules governing the visibility and types of information that can be shared with other members of the federation. Each Domain in the set of Domains in a federated architecture acts as a group of semi-autonomous Domains that exchange information with each other. The agreement(s) are used to standardize interoperability between each federated Domain. The set of federated Domains act collectively, but each Domain is distinct and has its own identity. This is shown in Figure 5.

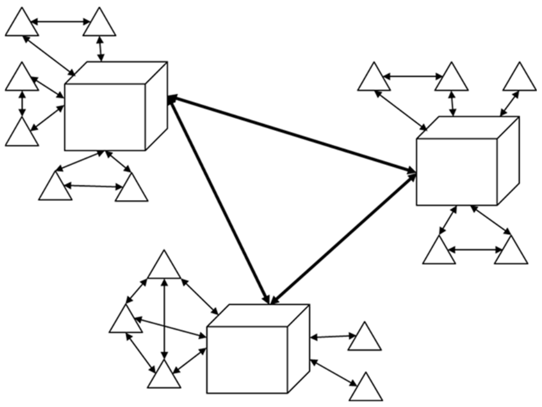

Figure 5: A Federated Domain System

Entities in each federated Domain may communicate with each other. External users and applications interact with each Domain independently.

An example of a federated architecture is the introduction of Consortium Blockchain products and technologies. These are also called Federated Blockchains. Leadership in this type of blockchain is defined as a group of nodes. These nodes control the determination of consensus, as well as information permissions (e.g. read, write).

4.8.5 Management Domains and Policy Management

In ENI, a Management Domain is one in which Policy Rules are used to define and manage the behaviour of the Domain. This can range from defining access control and visibility of Entity attributes, to determining which Entities in which Domain can communicate with other Entities in other Domains, to controlling which operations are used to perform which functions on what Entities.

Management Domains provide a consistent and scalable mechanism for applying Policy Rules to selected Entities in the Assisted System and/or to the ENI System. Policy Rules can be used to change or maintain the state of the Entity being managed. In ENI, Policy Rules are invoked at the "Act" stage of an OODA (or OODA-like) control loop, as described in [i.12]. This is discussed further in ETSI GS ENI 005 [i.1], clauses 5.7 and 6.3.4.7.

4.9 Publish-Subscribe Messaging Systems

4.9.1 Introduction

ENI uses a Semantic Bus, which is a type of publish-subscribe messaging system. Publish-subscribe is a loosely coupled communication paradigm for distributed computing environments. Originally, publishers publish information to event brokers in the form of events, and subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers. This evolved to using three decoupling dimensions - time, space, and synchronization - to support large-scale and highly dynamic distributed systems [i.7]. More specifically:

time decoupling:	Publishers and subscribers do not need to be actively interacting for publishers to publish information and subscribers to receive information.
space decoupling:	Publishers and subscribers do not need to know each other, or even how many other publishers and subscribers exist.

synchronization decoupling: Publishers are able to publish information at any time, and subscribers are asynchronously notified.

Decoupling the production and consumption of information increases scalability by removing all explicit dependencies between the interacting entities.

4.9.2 Subscription Models

There are four important subscription models:

Topic Subscription: A topic is defined by a set of keywords that map individual topics to message channels. Topic hierarchies can be used to orchestrate topics, so that subscribing to a topic automatically includes all of its sub-topics.

Type-based Subscription: This approach replaces the name-based topic model by a mechanism that filters events according to their object type. The advantage of this approach is the integration of language used and the messaging system.

Content Subscription: This type of subscription provides more expressiveness, since it can use expressions to designate actual content in the event (i.e. its attributes), as opposed to a (limited and pre-defined) set of keywords. This can be organized as a graph for efficient retrieval of information.

Semantic Subscription: This type of subscription matches messages based on the meaning of the message (in addition to content and topic). The subscriber defines the messages that are of interest by defining the structure and/or meaning of the message types in which it is interested. This may be done by using a declarative policy, by using a formal logical expression, or by other similar means.

4.9.3 The ENI Semantic Bus

This approach uses semantic subscription and a set of active repositories (i.e. a repository that pre- and/or post-processes information that is stored or retrieved). This enables a dynamic knowledge base to be implemented, where the information contained in the knowledge base is experientially learned. For example, as new information is discovered, it can be semantically validated and can then be added to the knowledge base, where subscribers can retrieve it via subscriptions.

Messages are semantically interpreted using a linguistic mechanism, such as an ontology. This approach uses formal logic to describe the concepts of a domain, together with their attributes and relationships. This has the benefit of enabling new message types to be dynamically defined, since a new message is simply a sub-type of an appropriate existing message concept. For example, given an existing concept of a Message that has a payload, suppose a new type of network device is introduced. This can be dynamically defined as follows:

$$\begin{aligned} &\mathit{Message} \quad \cap\ \mathit{hasPayload} = 1\ (\mathit{MessagePayload} \\ &\qquad\qquad \cap\ \exists\ \mathit{hasContent}(\mathit{Entity} \cap \mathit{hasEntityValue} = 1\ \mathit{NewNetworkEntity})) \end{aligned}$$

where:

- the new concept (called NewNetworkEntity) is a sub-concept of Entity
- the new message type is a sub-concept of Message, and hence, inherits the hasPayload relationship
- the payload of the new message type has exactly one NewNetworkEntity

The benefit of using formal logic is that new message types are automatically validated, since two conditions need to be satisfied (by using a formal logic). First, the new message type is obligated to be a sub-type of the appropriate Message concept. Second, when the concepts and relationships are added to the ontology, no inconsistencies are allowed. Both of these constraints are checked using an ontology reasoner.

An ontology reasoner is a software component that performs inferences, and sometimes logical reasoning (see clause 4.5.2), from a set of statements. The statements can include theories and axioms. The inference rules are typically defined using a Description Logic language (see clause 4.5.2).

Similarly, subscriptions can be dynamically defined. A subscription for the newly added NewNetworkEntity with a particular IP address is written as:

$$\begin{aligned} \mathit{Message} \quad & \cap\ \mathit{hasPayload} = 1\ (\mathit{MessagePayload} \\ & \cap\ \exists\ \mathit{hasContent}(\mathit{Entity} \\ & \cap\ \mathit{hasEntityValue} = 1\ (\mathit{NewNetworkEntity} \cap \mathit{hasIPAddress} = \text{"10.10.1.1"}))) \end{aligned}$$

5 Summary and Recommendations

The present document has described generic principles for constructing modular systems. These include fundamental software design principles, functional block based design, state machines, and how knowledge is represented, measured, and inferred. Knowledge representation is based on extensions to the DIKW (Data-Information-Knowledge-Wisdom) hierarchy [i.4]. This serves as an introduction to logic and inferencing, which is described in more detail in ETSI GR ENI 018 [i.10]. The document finishes with describing and comparing centralized and Decentralised communication, the difference between administrative and management domains and their organization, and publish-subscribe messaging system.

These principles are all used in the design of the ETSI GS ENI 005 [i.1], and are applicable to other ETSI reports and standards.

Thus, the present document recommends that the contents of the present document are applicable as primary concepts that sustain the ETSI GS ENI 005 [i.1] and other related documents and specifications.

History

Document history		
V2.1.1	July 2021	Publication