ETSI GS ENI 005 V3.1.1 (2023-06)

Experiential Networked Intelligence (ENI); System Architecture

Disclaimer

The present document has been produced and approved by the Experiential Networked Intelligence (ENI) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
RGS/ENI-005v311_Sys_Arch

Keywords
API, architecture, artificial intelligence, closed control loop, cognition, functional architecture, functional block, intent management, management, model-driven engineering, network, policy management

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

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

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**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.

Executive summary

The present document specifies a high-level functional abstraction of the ENI System Architecture in terms of Functional Blocks and External Reference Points. This includes describing how different classes of systems interact with ENI. Processes, models, and detailed information are beyond the scope of the present document.

Introduction

The present document defines a high-level functional abstraction of the ENI System Architecture. The organization of the present document is as follows:

- clause 1 defines the scope of the present document;
- clauses 2 and 3 provide normative and informative references and definition of terms, respectively;
- clause 4 provides an informative overview of the ENI System Architecture, including its motivation, benefits, important concepts, and an overview of its Functional Blocks;
- clause 5 lists requirements of the ENI System Architecture;
- clause 6 defines important design principles of the ENI System Architecture, and then specifies the different Functional Blocks that make up the ENI System Architecture;
- clause 7 specifies the External Reference Points of the ENI System Architecture;
- clause 8 specifies the initial design of ENI Application Programming Interfaces
- clause 9 describes how ENI interacts with other SDO Systems; and
- clause 10 delineates a list of future study items.

1 Scope

The present document specifies the functional architecture of an ENI System, which is a high-level decomposition of an ENI System into its major components, along with a characterization of the externally visible behaviour (e.g. as defined by a set of reference points) of the components. This includes:

- defining the functionality and behaviour of a system that satisfy the ENI Requirements (ETSI GS ENI 002 [i.40]);
- defining a functional architecture, in terms of Functional Blocks, that addresses the goals specified by the ENI Use Cases (ETSI GS ENI 001 [3]);
- defining Reference Points used by the above Functional Blocks for all communication with systems and entities that are external to the ENI System;
- proposing a progression plan towards full support of the proposed ENI System and intermediary level of compliance (e.g. support of some architecture components or a subset of the Reference Points).

The purpose of the present document is to continue the development of ETSI GS ENI 005 [i.53] (V2.1.1) to:

- define and specify APIs, Interfaces, and protocols used by ENI based on information and data models;
- specify the ENI cognition model in detail;
- enhance the description and specification of the ingestion, normalization, and output generation of data, information, and policies (imperative, declarative, and intent) in greater detail;
- enhance the description and specification of the control loops used in ENI;
- enhance the description and specification of policy management used in ENI;
- enhance the description and specification of architectural principles for interacting with other groups within and outside ETSI.

2 References

2.1 Normative 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.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference.

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 necessary for the application of the present document.

[1] ETSI GS NFV-MAN 001: "Network Functions Virtualisation (NFV); Management and Orchestration".

[2] IETF RFC 4949: "Internet Security Glossary, Version 2", Shirey, R., August 2007.

[3] ETSI GS ENI 001 (V3.1.1): "Experiential Networked Intelligence (ENI); ENI use cases".

[4] Strassner, John and Agoulmine, Nazim and Lehtihet, E. (2006): "FOCALE: A Novel Autonomic Networking Architecture". In: Latin American Autonomic Computing Symposium (LAACS), 2006, Campo Grande, MS, Brazil.

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

[6] Strassner. J.: "Knowledge Representation, Processing, and Governance in the FOCALE Autonomic Architecture", chapter 11 of Autonomic Network Management Principles, 2011, Elsevier.

[7] MEF Standard MEF 78.1: " MEF Core Model (MCM)", July 2020.

[8] MEF Standard MEF 95: "MEF Policy Driven Orchestration (PDO)", July 2021.

[9] ETSI GS ENI 019 (V3.1.1): "Experiential Networked Intelligence (ENI); Representing, Inferring, and Proving Knowledge in ENI".

[10] IETF RFC 7301: "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", S. Friedl, A. Popov, A. Langley and E. Stephan, July 2014.

[11] IETF RFC 8447: "IANA Registry Updates for TLS and DTLS", J. Salowey, S. Turner, August 2018.

[12] Void.

[13] Void.

[14] Void.

[15] IETF RFC 5280: "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", D. Cooper, et al., May 2008.

[16] IETF RFC 6818: "Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", P.Yee, January 2013.

[17] IETF RFC 8399: "Internationalization Updates to IETF RFC 5280", R. Housley, May 2018.

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] Strassner J.: "Policy-Based Network Management", Morgan Kaufman, ISBN 978-1558608597, September 2003.

[i.2] Strassner J., de Souza J.N., Raymer D., Samudrala S., Davy S., Barrett K.: "The Design of a Novel Context-Aware Policy Model to Support Machine-Based Learning and Reasoning", Journal of Cluster Computing, Vol 12, Issue 1, pages 17-43, March 2009.

[i.3] Strassner J., van der Meer S., O'Sullivan D. and Dobson S.: "The Use of Context-Aware Policies and Ontologies to Facilitate Business-Aware Network Management", Journal of Network and Systems Management 17(3), pages 255-284, 2009.

[i.4] Strassner J., Betser J., Ewart R., Belz F.: "A Semantic Architecture for Enhanced Cyber Situational Awareness", Secure and Resilient Cyber Architectures Conference, MITRE, 2010.

[i.5] Gamma E., Helm R. Johnson R., Vlissides J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, November 1994. ISBN 978-0201633610.

[i.6] Bäumer D., Riehle D., W. Siberski, M. Wulf: "The Role Object Pattern", Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97), ACM Press, 1997, Pages 218-228.

[i.7] Chin K.O., Ganb K.S., Alfred R., Anthony P. and Lukose, D.: "Agent Architecture: An Overview", Transactions on Science and Technology, vol 1, No 1, pp 18-35, 2014.

[i.8] Shehory O. and Sturm A. editors: "Agent-Oriented Software Engineering", Springer, 2014.

[i.9] Martin R. C.: "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall, 2003 ISBN 978-0135974445.

[i.10] Ritter F.E., Tehranchi F., Oury J.D.: "ACT-R: A Cognitive Architecture for Modeling Cognition", Wiley Interdisciplinary Reviews, Cognitive Science 10(4): e1488.

[i.11] IETF RFC 8328: "Policy-Based Management Framework for the Simplified Use of Policy Abstractions (SUPA)", Liu W., Xie C., Strassner J., Karagiannis G., Klyus M., Bi J., Cheng Y. and D. Zhang.

[i.12] Rothenberg, J.: "The Nature of Modelling", Artificial Intelligence, Simulation, and Modeling, John Wiley and Sons, Inc., 1989, pp. 75-92.

[i.13] Recommendation ITU-T 9594-1: "Information Technology - Open Systems Interconnection - The Directory: Overview of Concepts, Models, and Services".

[i.14] Recommendation ITU-T 9594-7: "Information Technology - Open Systems Interconnection - The Directory: Selected Object Classes".

[i.15] ETSI GR ENI 003 (V1.1.1): "Experiential Networked Intelligence (ENI); Context-Aware Policy Management Gap Analysis".

[i.16] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation).

[i.17] Mitchell Tom M.: "Machine Learning", McGraw-Hill, 978-0070428072.

[i.18] Gruber Thomas R.: "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", International Journal of Human Computer Studies, Vol 43, pp 907-928, 1993.

[i.19] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M.: "Pattern-Oriented Software Architecture - A System of Patterns", John Wiley and Sons, 1996.

[i.20] IEEE P7003™: "Algorithmic Bias Considerations".

[i.21] Void.

[i.22] The Moral Machine project demonstration.

[i.23] Anderson M. and Anderson S.L.: "GenEth: A general ethical dilemma analyzer", AAAI, pages 253-261, 2014.

[i.24] Cointe N., Bonnel G. and Boissier O.: "Ethical judgment of agents' behaviors in multi-agent systems", AAMAS, pages 1106-1114, 2016.

[i.25] The IEEE™ Global Initiative on Ethics of Autonomous and Intelligent Systems.

[i.26] Gartner: "Magic Quadrant for Full Life Cycle API Management", October 2019.

[i.27] Koene A., Smith A.L., Egawa T., Mandalh S. and Hatada Y.: "IEEE P70xx, Establishing Standards for Ethical Technology", KDD, 2018.

[i.28] Rao A.S. and Georgeff M.P.: "BDI Agents: From Theory to Practice", AAAI, 1995.

[i.29] IEEE™: "Ethically Aligned Design: A Vision for Prioritizing Human Well-being with Autonomous and Intelligent Systems".

[i.30] Famaey J., Latré S., Strassner J. and De Turck F.: "An Ontology-Driven Semantic Bus for Autonomic Communication Elements", IEEE International Workshop on Modeling Autonomic Communication Environments, pages 37-50, 2010.

[i.31] OMG: "Semantics of Business Vocabulary and Rules", version 1.5, December 2019.

[i.32] Attempto Controlled English.

[i.33] ETSI GR ENI 008 (V2.1.1): "Experiential Networked Intelligence (ENI); InTent Aware Network Autonomicity (ITANA)".

[i.34] Hohpe G. and Woolf B.: "Enterprise Integration Patterns", Addison-Wesley, 2003, ISBN 9780321200686.

[i.35] ETSI GR ENI 016 (V2.1.1): "Experiential Networked Intelligence (ENI); Functional Concepts for Modular System Operation".

[i.36] ETSI GR ENI 017 (V2.1.1): "Experiential Networked Intelligence (ENI); Overview of Prominent Control Loop Architectures".

[i.37] ETSI GR ENI 018 (V2.1.1): "Experiential Networked Interlligence (ENI); Introduction to Artificial Intelligence Mechanisms for Modular Systems".

[i.38] Barrett K., Davy S., Strassner J., Jennings S., van der Meer S., Donnelly W.: "A Model Based Approach for Policy Tool Generation and Policy Analysis", Global Information Infrastructure Symposium, 2007.

[i.39] BBF TR-384: "Cloud Central Office Reference Architectural Framework", January 2018, G. Karagiannis, D. Hai.

[i.40] ETSI GS ENI 002 (V3.1.1): "Experiential Networked Intelligence (ENI); ENI requirements".

[i.41] ETSI TR 103 240: "Powerline Telecommunications (PLT); Powerline communication recommendations for smart metering and home automation".

[i.42] ETSI GS ENI 005 (V1.1.1): "Experiential Networked Intelligence (ENI); System Architecture", September 2019.

[i.43] Minsky M.: "The Society of Mind", Simon and Schuster, New York, 1988.

[i.44] GRPC Core.

[i.45] Fielding R.T.: "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. thesis, 2000.

[i.46] IEEE 7000-2021™: "IEEE Standard Model Process for Addressing Ethical Concerns during System Design".

[i.47] IEEE 7001-2021™: "IEEE Standard for Transparency of Autonomous Systems".

[i.48] IEEE 7002-2022™: "IEEE Standard for Data Privacy Process".

[i.49] IEEE 7006™: "Personal Data AI Agent" (work in progress).

[i.50] IEEE 7007-2021™: "IEEE Ontological Standard for Ethically Driven Robotics and Automation Systems".

[i.51] gPRC.

[i.52] OAuth 2.0.

[i.53] ETSI GS ENI 005 (V2.1.1): "Experiential Networked Intelligence (ENI); System Architecture".

[i.54] IEEE P7008™: "Standard for Ethically Driven Nudging for Robotic, Intelligent and Autonomous Systems".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

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

NOTE: ETSI GR ENI 003 [i.15], defined abstraction as the "process of focusing on the important characteristics and behaviour of a concept, and realizing this as a set of one or more elements in an information or data model". The above definition is introduced to emphasize the importance of hiding (not deleting) unnecessary details, and more importantly, removing the constraint of use for a model. Abstraction is fundamentally a mental process that may take the form of model elements, but does not have to.

actor: role, played by an external entity (human or machine), which interacts with the subject of a use case

NOTE: An actor is always a type of stakeholder (but not vice versa). See stakeholder.

agent: computational process that implements the autonomous, communicating functionality of an application:

- **software agent:** software that acts on behalf of a user or another program
- **software autonomous agent:** software agent that acts on behalf of the entity that owns it without any communication from the owning entity
- **software intelligent agent:** software agent that reasons about its environment and take the best set of actions to satisfy a set of goals

NOTE: This has the connotation of containing AI mechanisms to provide the reasoning and decision-making capabilities.

- **software multi-agent:** set of software agents that are physically separate that work together to satisfy a set of goals

anomaly: measurable consequences of an unexpected change in state of a datum, or set of data, which is outside of its local or global norm

API: set of communication protocols, code, and tools that enable one set of software components to interact with either a human or a different set of software components

NOTE: This is also known as an Application Programming Interface.

API Broker: software entity that mediates between two systems with different APIs, enabling the two different systems to communicate transparently with each other

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

- **cognitive architecture:** system that learns, reasons, and makes decisions in a manner resembling that of a human mind

NOTE 1: Specifically, the learning, reasoning, and decision-making is performed using software that makes hypotheses and proves or disproves them using non-imperative mechanisms that typically involve constructing new knowledge dynamically during the decision-making process.

- **deliberative architecture:** symbolic world model that enables problem-solving components to be built using a sense-plan-act paradigm
- **hybrid architecture:** system made up of reactive and deliberative components that are combined into a hierarchy of interacting layers, where each layer reasons at a different level of abstraction

- **reactive architecture:** system that is aware of changes that affect its computations and adjusts accordingly

NOTE 2: The adjustment is made by reacting to an event in real-time without centralized control. The availability of new information drives program logic execution.

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

assisted system: system that the ENI system is providing recommendations and/or management commands to is referred to as the "assisted system"

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

bias: systematic difference in treatment of certain objects, ideas, or people in comparison to others:

- **algorithmic bias:** algorithm that possesses systematic and repeatable errors that create unfair outcomes
- **emergent bias:** reliance on an algorithm that has not been adjusted to evaluate new forms of data
- **inductive bias:** set of assumptions that are used in a machine learning algorithm are used to predict outputs for inputs that it has not encountered

Bidirectional Encoder Representations from Transformers (BERT): unsupervised deep learning strategy that utilizes bidirectional models that considers all words of the input sentence simultaneously and then uses an attention mechanism to develop a contextual meaning of the words

blackboard: architecture that uses a shared workspace that a set of independent agents contribute to, which contains input data along with partial, alternative, and completed solutions

BSS-like functionality: used to support customer-facing activities for the operator

NOTE: Examples include customer service, rating, order management, billing, and settlement.

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

NOTE: These features may, but do not have to, be used. These features may represent all or a subset of the functionality provided by a managed entity. Since a Functional Block is 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.

case-based reasoning: use of existing experiences and knowledge to understand and solve new problems

catastrophic forgetting: tendency of an artificial neural network to forget previously learned information when learning new information

class: template for defining a specific type of object that exhibits a common set of characteristics and behaviour:

- **abstract class:** class that cannot be directly instantiated
- **concrete class:** class that can be directly instantiated

classifier: procedure that predicts which elements of a set belong to which (pre-defined) classes:

NOTE 1: The classification is done using training data whose category membership is known, and can be thought of as a function that assigns a new observation a class label.

- **binary classifier:** classifier that decides whether or not an input belongs to one of two groups (i.e. classes) based on a classification function
- **discriminative classifier:** classifies an object based on the class labels

NOTE 2: This directly estimates the conditional probability of P(Y|X). An example is logistic regression.

- **generative classifier:** classifier that learns a model of the joint probability of an input x and the label y, uses Bayes rules to calculate p(Y|X), and then assigns the most likely label

NOTE 3: This estimates P(Y|X) by estimating P(Y) and P(X|Y). An example is Naïve Bayes classifier.

- **hierarchical classifier:** classifier that maps input data into a tree-like set of output categories by first, classifying at a low level, and then iterating each lower-level classification into a higher-level classification
- **linear classifier:** classifier that assigns a label based on a linear combination of its features
- **probabilistic classifier:** classifier that assigns a label to an object based on a (conditional) probability distribution

clustering: grouping of a set of objects such that objects in the same group are more similar to each other, by one or more measures, than to other objects in other groups

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

- **cognition model:** computer model of how cognitive processes, such as comprehension, action, and prediction, are performed and influence decisions

collaborating: two or more managed entities cooperate to accomplish a given task

compiler: computer program that translates the content of a source programming language into a different, or target, programming language

concept drift: not taking changing data and its meanings into account when training an ML model

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

control loop: mechanism that senses the performance of an object or process being controlled to achieve desired behaviour:

- **adaptive closed control loop:** closed control loop whose controlling function adapts to the object or process being controlled using parameters that are either unknown and/or vary over time
- **closed control loop:** control loop whose controlling action is dependent on feedback from the object or process being controlled

NOTE 1: This type of control loop measures the difference between the actual and desired values of a set of variables to adjust a set of parameters to change the behaviour of the system to bring the actual value closer to that of the desired value.

- **cognitive closed control loop:** closed control loop that selects data and behaviours to monitor that can help assess the status of achieving a set of goals, and produce new data, information, and knowledge to facilitate the attainment of those goals
- **distributed closed control loop:** closed control loop whose components are physically distributed among different locations
- **federated closed control loop:** set of semi-autonomous closed control loops that use formal agreements to govern their interaction and behaviour
- **hierarchical closed control loop:** closed control loop that is organized in the form of a tree
- **open control loop:** control loop whose controlling action is independent of the output of the object or process being controlled

NOTE 2: This type of control loop does not link the control action to the object or process being controlled (it simply continues to apply the control action).

- **peer closed control loop:** two or more closed control loops that may interact, but are independent of each other

coupling: amount of interdependence between two components, modules, or systems

data mining: procedure that discovers patterns in, and extracts knowledge from, data sets

NOTE: For the purposes of ENI, these patterns are of two principal types:

1) patterns that cause the generation of data; and

2) patterns that relate data (typically in a semantic manner).

decidable: procedure that determines, by a mathematical formal means in a finite amount of time, whether a formula is valid

decision making: set of processes that result in the selection of a set of actions to take from among several alternative possible actions

denormalization: process of changing information from a canonical form to one specialized for a particular actor and/or domain

designated entity: operator, nms, ems, controller, or orchestrator acting on behalf of the assisted system

NOTE: The designated entity is a trusted system, (a type of trusted entity [2]).

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.

- **design pattern, architecture:** general, reusable solution in a given context to a commonly occurring problem in the design of the software architecture of a system
- **design pattern, software:** general, reusable solution in a given context to a commonly occurring problem in the design of a software system

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 [7].

- **administrative domain:** domain that employs a set of common administrative processes to manage the behaviour of its constituent Entities

NOTE 2: This is based on the definition in [7].

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

NOTE 3: A Management Domain refines the notion of a Domain by adding three important behavioural 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 behaviour of MCMManagedEntities contained in the MCMManagementDomain. This is based on the definition of an MCMDomain in [7].

ENI Interface: point across which two or more components exchange information:

- **ENI API Interface:** ENI Interface set of communication mechanisms through which a developer constructs a computer program
- **ENI Hardware Interface:** ENI Interface across which electrical, mechanical, and/or optical signals are conveyed from a sender to one or more receivers using one or more protocols
- **ENI Software Interface:** ENI Interface point through which communication with a set of resources (e.g. memory or CPU) of a set of objects is performed

ENI Reference Point: logical point of interaction between specific Functional Blocks:

- **ENI External Reference Point:** ENI Reference Point that is used to communicate between an ENI Functional Block and an external Functional Block of an external system
- **ENI Internal Reference Point:** ENI Reference Point that is used to communicate between two or more Functional Blocks that belong to the ENI System

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.

ethics: set of principles that govern the moral behaviour of a person or machine:

- **consequentialist ethics:** agent is ethical if and only if it considers the consequences of each decision and chooses the decision that has the most moral outcome
- **deontological ethics:** agent is ethical if and only if it respects obligations, duties, and rights appropriate for a given situation
- **ethical dilemma:** situation in which any available decision leads to infringing on one or more ethical principles
- **virtue ethics:** agent is ethical if and only if it acts according to a set of moral values

feature: (traditionally), individually measurable property of an object under observation

feature: (for ENI), individually measurable characteristic or behaviour of an object being observed:

NOTE 1: Traditionally, individually measurable characteristics were assigned numerical values. For ENI, these individually measurable characteristics or behaviours may be allowed to be numeric or other types of data.

- **feature construction:** creating higher-level features from lower-level features (see feature hierarchy)
- **feature engineering:** process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data

NOTE 2: Feature engineering is concerned with determining the best representation of the sample data to learn a solution for a given problem.

- **feature hierarchy:** tree-like structure of features, where a higher-level object represents the composition of its lower-level objects

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

- **formal grammar:** set of structural rules that define how to form valid strings from a language's alphabet that obey the syntax of the language

formula: finite sequence of symbols from an alphabet that is part of a formal language:

- **atomic formula:** formula that does not have logical connectives
- **first-order logic formula:** well-formed formula that has a subject and a predicate that can have quantifiers

NOTE 1: First-order logic restricts the predicate to refer to a single subject. Both the universal and existential quantifiers may be used in constructing a first-order logic formula.

- **propositional formula:** well-formed formula that has a unique truth value
- **well-formed formula:** formula used in logic that obeys the grammatical rules of its formal language

NOTE 2: Feature engineering is concerned with determining the best representation of the sample data to learn a solution for a given problem.

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 1: 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.

- **directed graph:** graph where each connection, or edge, has an associated direction
- **graph loop:** edge of a graph that joins a vertex to itself

NOTE 2: For ENI, graph loops are not permitted.

- **hypergraph:** graph in which generalized edges may connect more than two nodes
- **multigraph:** graph in which multiple edges between nodes are permitted
- **weighted graph:** graph where each connection, or edge, has an associated weight

hyperparameter: learning parameter that is set before the learning process is started:

- **algorithm hyperparameter:** hyperparameter that affects only the speed and/or quality of the learning process, and does not affect the mathematical or statistical model used in the learning process (e.g. learning rate)
- **model hyperparameter:** hyperparameter that selects the mathematical or statistical model used in the learning process (e.g. size and topology of the ANN)

hypothesis: set of statements for explaining an observation that is not yet known to be true

interpreter: computer program that directly executes code from a programming language without requiring the code to have been compiled into a machine language program

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

knowledge representation: definition of data and information, applied in a particular context, that enables a machine to understand and use in computations

NOTE: This is available at http://groups.csail.mit.edu/medg/ftp/psz/k-rep.html.

label: identification of an output value for a given input

NOTE: Supervised learning uses labelled data; semi-supervised learning uses labels for a portion of the training data (the remaining training data are not labelled); unsupervised learning is based on training data that are not labelled.

language: structured and well-defined system of communication:

- **controlled language:** restricted version of a single Natural Language that uses a subset of the grammar of the Natural Language

- **Domain Specific Language (DSL):** small human-understandable language that uses a higher level of abstraction to communicate and configure software systems for a particular application domain:

  - **external DSL:** DSL that has its own custom syntax

NOTE 1: An external DSL is not dependent on another language.

  - **internal DSL:** DSL that defines a specific way to use a host language to give it a different feel

NOTE 2: An internal DSL does not require a custom compiler or interpreter, because it is embedded into its base language.

- **general purpose language:** programming language that can address a wide variety of problems and domains

- **natural language:** human-understandable language that is is used to interact with a computer program

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

NOTE 1: The learning process adjusts parameters to minimize observed errors; if the error rate becomes too high, then the ANN needs to be redesigned.

- **active learning:** learning algorithm that can query a user interactively to label data with the desired outputs

NOTE 2: The algorithm proactively selects the subset of examples to be labeled next from the pool of unlabeled data. The idea is that an ML algorithm could potentially reach a higher level of accuracy while using a smaller number of training labels if it were allowed to choose the data it wants to learn from.

- **batch learning:** type of offline learning algorithm that is updated (i.e. retrained) periodically

- **deep learning:** use of hierarchical computational models, which are composed of multiple processing layers, to learn representations of data with multiple levels of abstraction

NOTE 3: This replaces manually-intensive processes, and enables a machine to both learn features and use them to perform a task. Deep learning can be applied to almost any of the other algorithms defined here, as long as there are at least two hidden layers.

- **dictionary learning:** use of sparse matrices to represent input data using a linear combination of elements from a dictionary learned from training data

- **distributed data learning:** sets of data that are used to train multiple instances of the same model on different subsets of the training data set in parallel

NOTE 4: The same model is available to all computational nodes, so that a single coherent output emerges naturally by combining each of the model updates

- **distributed learning:** distribution of machine learning applications to multiple computing nodes

- **distributed model learning:** multiple exact copies of the same data sets are processed by working nodes that operate on different parts of the model; the resulting model is an aggregate of each of these operations

- **ensemble learning:** use of multiple learning algorithms to obtain better performance in predicting results than is possible from using any single learning algorithm

- **explanation-based learning:** learning generalized problem-solving by analysing solutions to specific problems

NOTE 5: Explanation-based learning enables a search procedure, constrained by general domain knowledge related to the context of the actual problem, to be used to provide more accurate and efficient learning in knowledge-intensive systems.

- **feature learning:** learning representations of data that make it easier to discover information from raw data when building different types of predictors (e.g. classifiers)

NOTE 6: This replaces manually-intensive processes, and enables a machine to both learn features and use them to perform a task.

- **federated learning:** approach that trains an algorithm across multiple decentralized entities holding local data samples, without exchanging their data samples

NOTE 7: Each device trains the model on their own local data set, and then each client sends a model or model update to a centralized service, which aggregates each client's contribution into one global model. The centralized service then distributes the global model back to the clients.

- **incremental learning:** learning from a continuously changing source of data (e.g. streaming data) that arrives over time

NOTE 8: This is a form of online learning.

- **machine learning:** use of a series of inputs to build a model, followed by the use of that model to create a representation, a decision, a prediction, or an answer

NOTE 9: Decisions are made without explicit instructions (e.g. through inferencing or other types of logical actions). A more formal definition is "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [i.17]. It is a subset of Artificial Intelligence genre.

- **offline learning:** data set being worked on does not change

NOTE 10: This means that parameters defined during the training depend on the entire training data set (e.g. are global).

- **online learning:** learning when data is not previously available

NOTE 11: In this approach, data arrives over time, and a model is first inferred, and then refined after each subsequent time step. It is similar to incremental learning, except that it is bounded in time (and possibly other factors, such as model complexity and resources).

- **reinforcement learning:** use of software agents to take actions in an environment in order to maximize a cumulative reward

NOTE 12: The learning agent is not told which actions to take, but instead needs to discover which actions yield the highest reward.

- **rule-based learning:** use of rules to represent the knowledge of a system

NOTE 13: This type of system learns rules to make decisions, instead of using a model. These rules are different than other types of rule-based systems because this set of rules are learned, while rules in other types of systems are defined.

- **semi-supervised learning:** hybridisation of supervised and unsupervised learning, where the training data consists of both labelled and unlabelled data

- **supervised learning:** learning a function that maps an input to an output based on example pairs of labelled inputs and outputs

- **unsupervised learning:** learning a function that maps an input to an output without the benefit of the data being classified or labelled

lexeme: linguistic unit of meaning, consisting of a word or group of words

NOTE: A lexeme is an abstract unit that can have many different forms. For example, in inflectional languages, a lexeme will have many forms (e.g. present and past). Idioms, as well as expressions, are also lexemes.

lexicon: collection of all words, phrases and symbols used in a language, along with their definition(s) and meaning(s)

NOTE: More formally, a lexicon is the complete set of morphemes in a language, along with their definitions and grammatical rules, that enables a complete vocabulary to be defined.

location: physical geographic location (e.g. a geocode or a bounding polygon) of an entity (e.g. a server)

NOTE: Contrast this with Placement.

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

- **alethic logic:** representing, using mathematical formalisms, expressions involving necessity, possibility, and contingency
- **defeasible logic:** representing, using mathematical formalisms, weak rules that are not necessarily justified by the fact, and could thus be proven incorrect

NOTE 1: Defeasible logic (and reasoning) uses three types of rules: strict rules that are TRUE, weak rules that may be true if no evidence can be found to contradict them, and "defeaters" that represent contradictory evidence to prove that a weak rule is incorrect.

- **deontic logic:** representing, using mathematical formalisms, expressions involving obligation, permission, and the concept of being forbidden
- **description logic:** family of formal languages that are subsets of first-order logic to ensure decidability and efficiency
- **doxastic logic:** representing, using mathematical formalisms, expressions involving the belief of a particular entity or set of entities
- **first-order logic:** extension of propositional logic to include predicates and quantification
- **fuzzy logic:** type of many-valued logic that allows a truth value to be any real number between 0 and 1 inclusive

NOTE 2: Fuzzy logic is most often used to reason about the degree of truth, or probability, in a system.

- **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 3: 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.

- **temporal logic:** representing, using mathematical formalisms, expressions involving time (e.g. it will be, it will always be, it was, and it has always been)

logic clause: expression made up of a finite set of literals (i.e. literals), including the negation of literals:

NOTE 1: In propositional logic, a literal is a variable. In predicate logic, a literal is an atomic formula.

- **Boolean clause:** expression that, when evaluated, produces a value of either true or false
- **Fact (clause):** Horn clause with no negative literals
- **Goal clause:** Horn clause without a positive literal
- **Horn clause:** disjunction of literals that contains at most one positive literal

NOTE 2: This is also called a definite clause. This is used in some types of automated theorem proving.

message system: system that transfers data between components

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

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

NOTE 1: This definition is taken from [7].

- **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 2: This definition is taken from [7].

- **machine learning model:** representation of a deterministic system using mathematical and/or logical formalisms

NOTE 3: A deterministic system is a system in which successive states of a system are determined by its preceding state; this means that an algorithm, given a particular input, will always produce the same output

- **ML model:** machine learning model

- **statistical model:** representation of a non-deterministic system using a set of statistical assumptions that describe how system data are generated

NOTE 4: In this model, randomness is introduced via defining some variables as stochastic (i.e. their values depend on the outcome of a random phenomenon). This is typically represented using probability distributions.

Model-Driven Behaviour (MDB): approach in which the behaviour of components, modules of systems are managed using MDE

NOTE: For ENI, this applies to Functional Blocks, not components within a Functional Block.

Model-Driven Engineering (MDE): approach in which models are central to all phases of the development and implementation processes

model element: attributes, methods, constraints, relationships, and stereotypes used in constructing a model

morpheme: smallest unit of meaning in a language

negotiation: set of communications that is intended to reach a beneficial outcome for a set of conflicting issues:

- **distributive negotiation:** zero-sum game, in which each participant assumes that there is a fixed amount of value to be divided between the (winning) bidders

- **integrative negotiation:** win-win (or non-zero-sum) game, in which all collaborating participants receive optimal value

neural network: network of nodes that communicate with other nodes via specialized connections:

NOTE 1: The above technically refers to the biological concept, where a node is a neuron (i.e. nerve cell). ENI will use the term artificial neural network, or ANN, to refer to the artificial intelligence concept.

- **artificial neural network:** computing system that learns to perform functions by using artificial neurons that take the form of a directed, weighted graph

NOTE 2: An ANN learns to perform a function by analysing examples (i.e. training data) instead of being programmed to perform a task.

- **artificial neuron:** node in an ANN hat receives weighted input data, adds the data, and produces an output using a non-linear output function

NOTE 3: The output function is also called a transfer function or activation function, and represents what portion of the potential action is transmitted.

- **feedforward neural network:** ANN whose inter-nodal connections do not form a cycle

NOTE 4: Put another way, information always moves forward and never backwards. In general, any directed acyclic graph is a type of feedforward neural network.

- **recurrent neural network:** ANN where connected between nodes form a directed graph across a temporal sequence

NOTE 5: RNNs contain cyclic connections that make them effective in storing history and state. This is done by using feedback loops in the processing of an output.

normalization: process of changing ingested information to a canonical form

np-complete: class of computational problems for which no efficient solution algorithm has been found

object: instance of a concrete class

ontology (traditionally): explicit specification of a conceptualization [i.18]

NOTE: This definition is the basis for definitions in OneM2M and SmartM2M.

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

OSS-like Functionality: used to support back-office activities that configure and operate a network for the operator

NOTE: Examples include inventory, configuration management, service assurance, and service activation.

parser: computer program that creates one or more data structures from an input programming language using an associated formal grammar

perceptron: supervised learning algorithm for binary classifiers, where the classification function is based on a linear function that combines the weighted sum of inputs with the feature vector:

NOTE 1: Since a linear function is used, a perceptron can only distinguish data that is linearly separable (compare to multi-layer perceptron).

- **multi-layer perceptron:** type of feedforward ANN that consists of an input layer, an output layer, and one or more hidden layers

NOTE 2: Since a non-linear function is used, a multi-layer perceptron can distinguish data that is not linearly separable (compare to perceptron).

placement: logical placement of an entity (e.g. a virtual machine) on or in another entity (e.g. a server)

NOTE: Contrast this with Location.

planning: finding a procedural course of action for a declaratively described system to reach its goals while optimizing overall performance measures

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 1: This is defined in [7] and [8], but see also [i.4].

- **declarative policy:** type of policy that uses statements from a formal logic to describe a set of computations that need to be done without defining how to execute those computations
- **ENI Policy Rules:** set of imperative, declarative, and/or intent policy rules
- **imperative policy:** type of policy that uses statements to explicitly change the state of a set of targeted objects

NOTE 2: The canonical form of an imperative policy is a triple, consisting of a set of Event, Condition, and Action Boolean clauses. Conceptually, it is evaluated as: when the Event Clause occurs, IF the Condition Clause is TRUE, THEN a set of actions may be executed.

- **intent policy:** type of policy that uses statements from a restricted natural language (e.g. an external DSL) to express the goals of the policy, but does not specify how to accomplish those goals

NOTE 3: In particular, formal logic syntax is not used. Therefore, each statement in an Intent Policy may require the translation of one or more of its terms to a form that another managed functional entity can understand.

- **policy conflict:** two policies that, when executed, cause contradictory and otherwise incompatible results within a given execution time window

predicate: statement that can be true or false depending on the value of its variables

probability: numerical value defining the likelihood of an occurrent occurring:

- **conditional probability distribution:** probability of an event occurring given that another event already occurred
- **probability distribution:** mathematical function that defines the probability of occurrence of all possible outcomes of the random phenomenom being observed

process: execution of a set of instructions that produce a result

proposition: statement that can be true or false

NOTE: The above definition, though incomplete, will be used for ENI for this release. Formally, this definition of a proposition is used in truth-functional propositional logic.

quantifier: specification of the number of objects in a given domain that satisfy a formula with at least one variable:

- **existential quantifier:** assertion that a property or relation holds for at least one member of a domain
- **universal quantifier:** assertion that a property or relation holds for all members of a domain

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

sample: set of objects selected or collected from a statistical population using a procedure

SDO system: part of an Assisted System that is defined by another SDO

NOTE: Examples include NFV MANO and MEF LSO.

semantic bus: type of message bus used to orchestrate and filter communications between ENI Functional Blocks based on the meaning, attributes, and metadata of a message using a shared set of interfaces

semantic closeness: measure of how similar the semantics of two different sets of concepts are

NOTE: The types of semantic relationships may be changed to achieve a closer amount of semantic closeness. This is one important mechanism in the knowledge discovery and alignment processes (see clause 6.3.4 for more information).

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:

- **situation awareness:** perception of data and behaviour that pertain to the relevant circumstances and/or conditions of a system or process, the comprehension of the meaning and significance of these data and behaviours, and how processes, actions, and new situations inferred from these data and processes are likely to evolve in the near future

NOTE: This definition is based on the material in [i.4].

stakeholder: set of individuals and/or organizations that may affect, be affected by, or perceive themselves to be affected by a decision, activity, or outcome of a process

syntax: set of rules that govern how elements of a statement are structured, including what element goes where in a statement

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

training: process of teaching an entity a set of knowledge, skills, processes, and/or behaviours:

- **offline training:** phase in machine learning that is used to create a model when the algorithm is not currently being executed
- **online training:** phase in machine learning that is used to perform inferences in real-time (i.e. when the model is actively being used)

use case: description of a specific configuration/deployment scenario of a system from the user point of view (ETSI TR 103 240 [i.41])

use case (for ENI): list of actions defining the interactions between a set of actors and the system in order to achieve a goal that has an observable result beneficial to a set of stakeholders

NOTE: The above definition of use case is more applicable for ENI modelling activities. In addition, the list of actions may consist of different sets of actions to accomplish a goal.

3.2 Symbols

Void.

3.3 Abbreviations

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

AAA	Authentication, Authorization, and Accounting
ACT-R	Adaptive Control of Thought-Rational
AES	Advanced Encryption Standard
AI	Artificial Intelligence
ALPN	Application-Layer Protocol Negotiation
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BBF	BroadBand Forum
BERT	Bidirectional Encoder Representations from Transformers
BSS	Business Support System
CA	Certificate Authority
CLI	Command Line Interface
CPU	Central Processing Unit
DIKW	Data-Information-Knowledge-Wisdom

NOTE: See ETSI GR ENI 016 [i.35].

DSL	Domain Specific Language
EMS	Element Management System
ENI	Experiential Networked Intelligence
FB	Functional Block
FOCALE	Foundation Observation Comparison Action Learning Environment

FTP	File Transfer Protocol
GDPR	General Data Protection Regulation
GPL	Graphics Purpose Language
GPS	Global Positioning System
GPU	Graphics Processing Unit
GR	Group Report
gRPC	gRPC Remote Procedure Call

NOTE: Originally, gRPC was a recursive acronym (the 'g' stood for gRPC). Now, the 'g' stands for various words that start with the letter g. See [i.44].

GS ENI	Group Specification of the Experiential Networked Intelligence ISG
GS	Group Specification
GUI	Graphical User Interface
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDL	Interface Definition Language
IoT	Internet of Things
IP	Internet Protocol
ISG	Industry Specification Group
ITU-T	International Telecommunications Union - Telecommunication standardization sector
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LSO	Lifecycle Service Orchestration
LSTM	Long Short-Term Memory
MAC	Media Access Control
MANO	Management and Network Orchestration
MBT	Model-Based Translation
MCM	MEF Core Model
MDB	Model-Driven Behaviour
MDE	Model Driven Engineering
MEF	MEF (a Standards body)
ML	Machine Learning
MPM	MEF Policy Model
mTLS	mutual TLS
NER	Named Entity Recognition
NFV	Network Functions Virtualisation
NFVI	Network Functions Virtualisation Infrastructure
NFVO	Network Functions Virtualisation Orchestrator
NMS	Network Management System
OAuth	Open Authentication
OODA	Observe-Orient-Decide-Act
OPEX	OPerational EXpenditures
OSS	Operations Support System
PMFB	Policy Management Functional Block
RA	Reference Architecture
RAML	RESTful API Modelling Language
RDBMS	Relational DataBase Management System
REST	Representational State Transfer
RMI	Remote Method Invocation
RNN	Recurrent Neural Network
ROA	Resource Orientated Architecture
RPC	Remote Procedure Call
SBVR	Semantics of Business Vocabulary and business Rules
SDN	Software Defined Network
SDO	Standards Defining Organization
SLA	Service Level Agreement
SQL	Structured Query Language
SSL	Secure Socket Layer

TCP	Transmission Control Protocol
TLS	Transport Level Security
TR	Technical Report
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
VIM	Virtualised Infrastructure Management
VNF	Virtualised Network Function
VNFM	Virtualised Network Function Manager
VoLTE	Voice over Long-Term Evolution
VPN	Virtual Private Network
WADL	Web Application Description Language
WAN	Wide Area Network
WG	Working Group
WSDL	Web Service Description Language
XML	eXtensible Markup Language
YANG	Yet Another Next Generation

4 Overview of System Architecture (informative)

4.1 Introduction

This clause provides an informative introduction to the ENI System Architecture. Clauses 4.2 and 4.3 describe the motivation and benefits of ENI. Clause 4.4 provides a high-level description of the ENI System Architecture, including how it interacts with the Assisted System (and/or its Designated Entity), different types of Assisted Systems, the mode of operation that the Assisted System (or its Designated Entity) can choose, direct and indirect communication between the ENI System and the Assisted System (or its Designated Entity), and important concepts used in this System Architecture. Clause 4.5 describes the functional architecture of the ENI System in terms of Functional Blocks. It ends with a discussion of how decision-making is done in the ENI System.

NOTE: See clause 9 for further applicable future work.

4.2 Motivation for ENI

Current network management provisioning and monitoring functions are time-consuming and error-prone. This is due to the proliferation of different technologies, as well as different implementations from different vendors. In addition, users are demanding more complex services (e.g. context-aware, personalized services). Hence, operators are concerned about the increasing complexity of integration of different platforms in their network and operational environment. These human-machine interaction challenges increase the time to market of innovative and advanced services. Moreover, there is no efficient and extensible standards-based mechanism to provide context-aware services (e.g. services that adapt to changes in user needs, business goals, or environmental conditions).

These and other factors contribute to a very high OPerational EXpenditure (OPEX) for network operation and management. Operators need to optimize the use of networked resources (e.g. through the automation of their network configuration and monitoring processes to reduce this OPEX). More importantly, operators need to improve the use and maintenance of their networks.

Indeed, operators are concerned about decreasing the number of the operator's personnel dedicated to managing the infrastructure (e.g. how many people are required to control a given number of network devices, or particular structures, such as SD-WANs). This requires improved automation and real-time closed control loops. Operators are also keenly interested in improving infrastructure efficiencies. For example, it is likely that 5G and IoT are expected to have complex interactions between different sets of network devices. Concurrently, user needs, business goals, and the environment frequently change. Thus, network intelligence is needed to detect these contextual changes, determine which groups of devices and services affect each other, and manage the resulting services while maintaining Service Level Agreements (SLAs).

To accomplish these targets, the associated challenges are stated as:

a) automating complex human-dependent decision-making processes (e.g. managing and optimizing network and system configuration processes) by providing system and network intelligence tools and services;

b) determining which services need to be offered, and which of those services are in danger of not meeting their Service-Level Agreements (SLAs), as a function of changing context;

c) determining how to avoid service performance degradation through analysis that leads to further automated and intelligent operations;

d) defining how best to visualize how network services are provided and managed to improve network maintenance and operation;

e) providing an experiential architecture (i.e. an architecture that uses Artificial Intelligence (AI) and other mechanisms to improve its understanding of the environment, and hence the operator experience, over time);

f) improving operator's personnel efficiency through improved management and automation, while providing increased visibility and a simplified interface between the operator and the networks and networked applications; this reduces errors and makes human-directed commands more efficient and intuitive; and

g) improving infrastructure utilization and agility (response to real-time changes) while maintaining SLAs. The deployed networks and systems likely need to be aware of the needs of Services and Applications, and handle environmental changes in an automated way without human involvement.

4.3 Benefits of ENI

The ENI System defines a Functional Block architecture that helps address the seven challenges of clause 4.2. It specifies Functional Blocks and Reference Points for providing a model-based, policy-driven, context-aware system that provides recommendations and/or commands to Assisted Systems. This communication is to be done directly or indirectly via a Designated Entity acting on the behalf of the Assisted System. A Designated Entity is an NMS, EMS, controller, or current or future management and orchestration systems. The ENI System is expected to enable the Assisted System to perform more accurate and efficient decision making.

The ENI System provides the following important benefits:

1) to measure and quantify the operation and performance of the resources and services of an operator;

2) to enable personalized services to be provided to customers;

3) to learn from the operation of the network, and decisions made by the operator;

4) to automate the operator's complex human-dependent decision-making processes by translating changing user needs, business goals, and environmental conditions into closed-loop configuration and monitoring;

5) to enable the optimization and adjustment of resources and services managed by the operator, as well as associated tools and applications needed by the operator to conduct business.

The ENI System is based on an experiential architecture. This means that it learns through experience (i.e. through the operation of the systems that it assists in governing). This self-learning principle is key to improving operator experience, and enables the system to evolve over time from proposing to implementing decisions.

The ENI architecture enables the operator to adjust the offering of services in response to contextual changes. It is expected that an ENI System enables the deployment, administration, and control of migration toward new technologies, such as SDN and NFV.

4.4 High-Level Description of the ENI System Architecture

4.4.1 Overall Description

Carrier and Enterprise network architectures are becoming increasingly complex and diverse. For example, it is possible to split a single application into separate distributed entities. As another example, managers, controllers, and orchestrators are also being distributed, making their interaction critical. Furthermore, the network and applications use new technologies and are programmed in diverse languages. Therefore, it is becoming more and more difficult to realize potential and significant reductions in CapEx and OpEx. A primary goal of ENI is to provide a robust, distributed platform that uses modelling, policy management, and AI to solve these problems.

The ENI System is an innovative, policy-based, model-driven functional architecture that improves operator experience. In addition to network automation, the ENI System assists decision-making of humans as well as machines, to enable a more maintainable and reliable system that provides context-aware services that more efficiently meet the needs of the business. For example, the ENI System enables the network to change its behaviour (e.g. the set of services offered) in accordance with changes in context, including business goals, environmental conditions, and the varying needs of end-users.

This is achieved by using policy-driven closed control loops that use emerging technologies, such as big data analysis, analytics, and artificial intelligence mechanisms, to adjust the configuration and monitoring of networks and networked applications. It dynamically updates its acquired knowledge to understand the environment, including the needs of end-users and the goals of the operator, by learning from actions taken under its direction as well as those from other machines and humans (i.e. it is an experiential architecture). It also ensures that automated decisions taken by the ENI System are correct and increase the reliability and stability, and lower the maintenance required, of the network and the applications that it supports. It improves and simplifies the management of network services through their visualization, and enables the discovery of otherwise hidden trends and interdependencies.

Finally, it helps determine which services are appropriate to be offered for a specific context, and which services are in danger of not meeting their Service-Level Agreement (SLA), as a function of changing context. In order to achieve the latter and to assist in monitoring and improving infrastructure efficiencies, it draws knowledge from telemetry shared with it.

It is possible to use the ENI System to advise on balancing the resources or availability of the network elements or function used to provide the services.

NOTE: The goal of the present document, at this stage in the industry evolution of AI/ML, is to ensure proper management mechanisms are present in any control loop used within the ENI System. This enables the ENI System or a human operator to manage the degree of autonomicity used within the control loop. This applies to passive and active recommendations and commands (e.g. monitoring information and sending reconfiguration commands), which in turn ensures the prevention of undesired outputs.

4.4.2 The Assisted System

4.4.2.1 Introduction

The system that the ENI System is providing recommendations and/or management commands to is referred to as the "Assisted System". Within the current scope of ENI, there are three classes of Assisted Systems. In each case, an ENI System requires data from the Assisted System. Changes to the Assisted System are not required for any class of Assisted System in order to facilitate the use and rapid adoption of ENI.

The ENI System uses an API Broker to mediate between the ENI System and the Assisted System (regardless of its class). This is because the ENI System and the Assisted System typically have different APIs. The API Broker defines the correct way for one system to request services from the other system without requiring an ENI System to understand the details of every API of every Assisted System that wants to communicate with it. All communication is done using external Reference Points.

The Assisted System and/or its Designated Entity do not have to accept the recommendations offered by an ENI System when in recommendation mode. It is expected that the Assisted System and/or its Designated Entity accept the recommendations and/or commands offered by an ENI System when in management mode. This includes those decisions that apply to recommendations and commands, respectively, when the Assisted System is in a mixed mode of operation. These different operational modes are defined in clause 4.4.3, and specified as requirements [MOP15] and [MOP16] in clause 5.3.

4.4.2.2 Communication Options for All Classes of Assisted Systems

There are two common options for communication between an System ENI and the Assisted System. These two options are shown in Figure 4-1.

1) In option 1, an ENI System communicates, using its API Broker, with the Operator or Designated Entity of the Assisted System. Data ingestion by an ENI System (e.g. "telemetry data") and assistance in response from an ENI System (e.g. "recommendations and commands") are communicated via the API Broker. This is shown in Figure 4-1 by using the solid red line and solid blue line, respectively. All communication is defined by the capabilities of the ENI API; hence, additional communication types not shown in Figure 4-1 are also possible.

2) In option 2, an ENI System preferentially communicates with the APIs of the Assisted System to conduct data ingestion and assistance response. This is again shown in the figure by the combination of red line and blue lines. All communication is defined by the capabilities of the ENI API. Hence, additional communication types not shown in Figure 4-1 are also possible. Both communication alternatives in option 2 are able to be used interchangeably.

   The dashed brown line in option 2 shows that there are two alternative communication options.

Both communication alternatives are able to be used interchangeably. For example, the API of the Assisted System (e.g. the solid blue line) is used to implement some policy changes, while other policy changes are implemented by communicating directly to the operator or Assisted System (e.g. the dashed brown line).

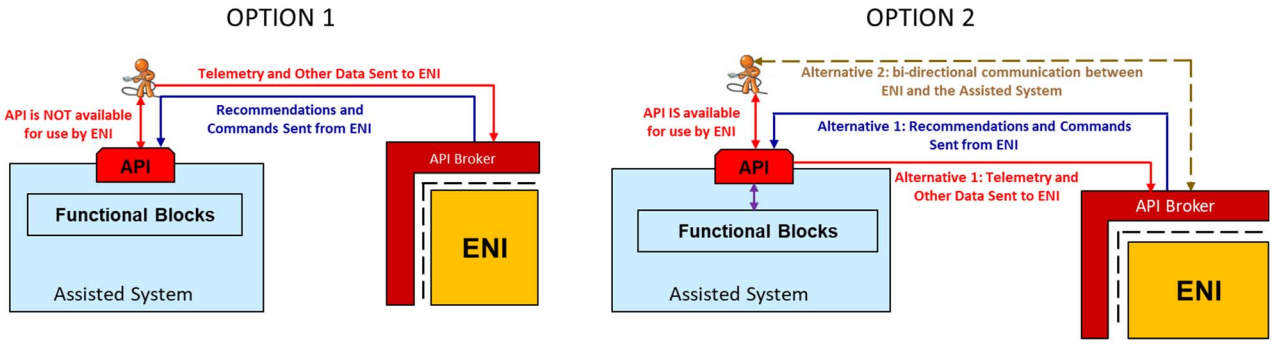

Figure 4-1: Two Common Communication Options of Classes of the Assisted System

For illustration purposes, in option 2, API translation is performed by the API Broker between a standard API provided by ENI and the API of the Assisted System (or its Designated Entity) that is communicating with an ENI System. This translation is external to the ENI System, because each Assisted System (or Designated Entity) typically has its own unique API. It is desirable to insulate an ENI System from the specific details of the Assisted System, so that it does not have to change with every change of APIs of the Assisted System. Thus, an API Broker is used, and functions as a gateway between the constant standardized set of APIs provided by an ENI System and the varying set of APIs that are particular to each individual Assisted System. Alternatively, the ENI System communicates with the Operator or Designated Entity of the Assisted System using the API Broker in option 2.

4.4.2.3 Class 1: An Assisted System that has No AI-based Capabilities

This type of Assisted System has no AI-based decision-making capabilities. Two options are shown in Figure 4-2.

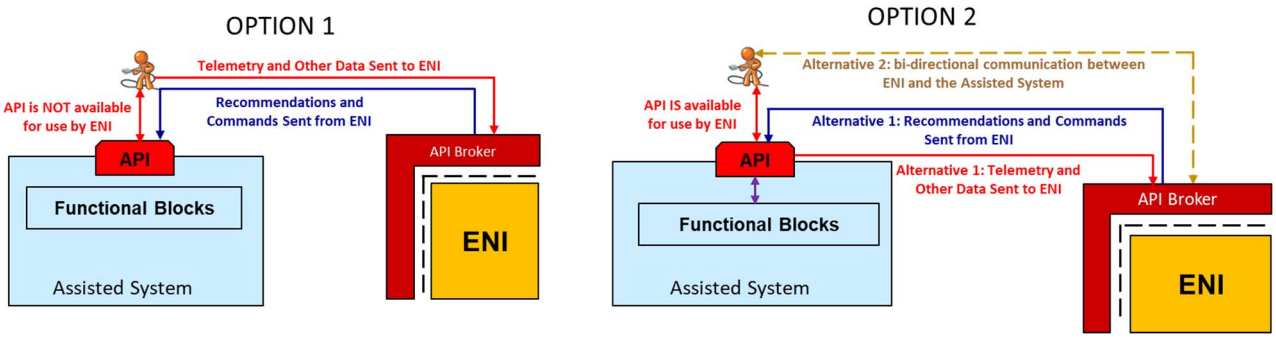

Figure 4-2: Class 1: The Assisted System Has No AI Capabilities

This class of Assisted System has no AI capabilities, so an ENI System operates as an external system that communicates with the Assisted System through standard Reference Points and APIs defined by ENI. In addition, most Assisted Systems of this class were not designed to accommodate new internal modules, so ENI needs to be incorporated as a discrete external system. The Assisted System and its Designated Entity are likely not able to use the outputs of the ENI System to their full potential, due to their lack of having AI capabilities.

In this class of Assisted System, the ENI System is not involved in making decisions in the real-time control loop of the Assisted System. The ENI System sends recommendations and/or management commands to improve decisions that do not involve real-time configuration (e.g. change some statically defined parameters to improve resource utilization). The burden is on ENI (and the API Broker, in the case of APIs) to present its results in a form that the Assisted System is able to understand. Hence, nothing has to change in the Assisted System in order for ENI to be used.

4.4.2.4 Class 2: An Assisted System with AI that is Not in the Control Loop

This type of Assisted System has some AI-based decision-making capabilities. Two options are also possible as shown in Figure 4-3.

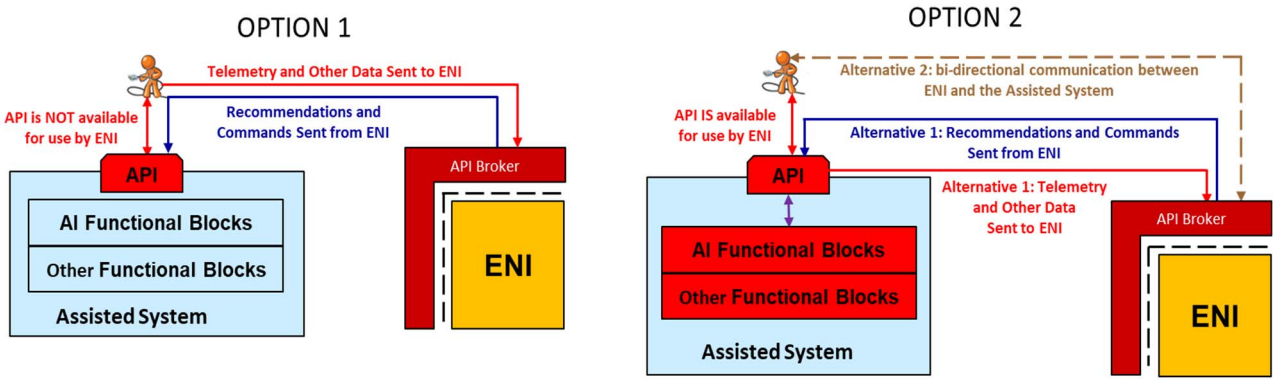

Figure 4-3: Class 2: Assisted System with AI, but Not in the Control Loop

This class of Assisted System uses some AI-based algorithms, but critically, does not use AI in its internal control loop. For example, optimization of the physical server location, virtual machine, and/or Container placement on a server, as well as the geographic location of the server, is likely performed using an AI-algorithm by this class of Assisted System. However, for this class of Assisted System, the ENI System is not involved in making decisions in the real-time control loop of the Assisted System. The ENI System sends recommendations and/or management commands to improve decisions that do not involve real-time configuration (e.g. as indicated above, change some statically defined parameters to improve resource utilization). This class of system works as an extension of class 1.

The architectural details of the Assisted System for class 2 systems are not visible to ENI. The two types of Functional Blocks shown in Figure 4-2 (i.e. "AI Functional Blocks" and "Other Functional Blocks") are meant to illustrate the separation of AI and non-AI Functional Blocks, and emphasize that AI Functional Blocks are not involved in real-time operational decisions.

Again, as for Class 1, a Class 2 Assisted System is not designed to directly accommodate all of the features provided by an ENI System. Hence, ENI operates as an external system that communicates with the Assisted System and/or its Designated Entity, as shown in Figure 4-3. The burden is on ENI (and the API Broker, in the case of APIs) to present its results in a form that the Assisted System and its Designated Entity are able to understand. Hence, nothing has to change in the Assisted System, or its Designated Entity, in order for ENI to be used.

4.4.2.5 Class 3: An Assisted System with AI Capabilities in its Control Loop

4.4.2.5.1 Introduction

This type of Assisted System has AI-based decision-making capabilities as part of its internal control loop. Two options are shown in Figure 4-4. Both of these options are shown using the API of the Assisted System. While this is not mandated, the lack of API communication in a Class 3 system means that it cannot take advantage of real-time operational changes from an ENI System and hence, will behave like a Class 2 system.

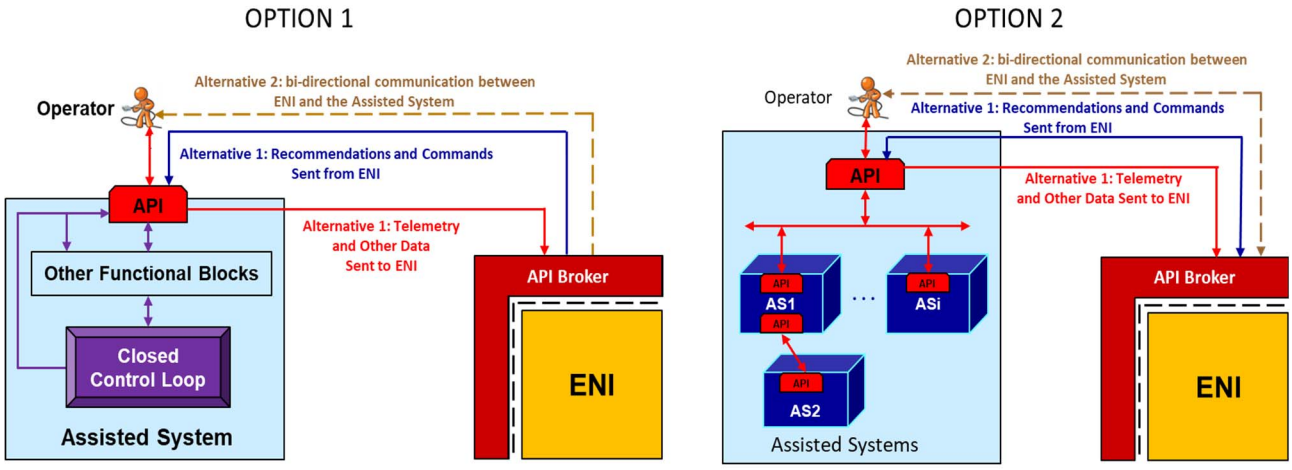

Figure 4-4: Class 3: Assisted System with AI in the Control Loop

This class of Assisted System optionally has other AI capabilities. The ENI System is involved in making decisions for any function performed by the Assisted System, as long as it has permission to do so. Significantly, this includes real-time decisions for this class of Assisted System. This is a fundamental difference between this class of Assisted System and those of Classes 1 and 2.

4.4.2.5.2 Class 3 Options

4.4.2.5.2.1 Introduction

There are two different options for using an ENI System with the control loop(s) of class 3 Assisted Systems. The first option (shown on the left of Figure 4-4) uses an ENI System as a modular addition that is able to directly affect the control loop(s) of the Assisted System through API commands. The second option (shown on the right of Figure 4-4) enables an ENI System to communicate its recommendations and commands to one or more Assisted Systems using a communication mechanism that the set of Assisted Systems use. This alternative enables an ENI System to communicate with distributed arrangements of collaborating Assisted Systems.

4.4.2.5.2.2 Option 1 Feedback

Option 1 emphasizes two key points:

- An ENI System is only communicating with a single Assisted System (compared to option 2).
- The Assisted System has one or more closed control loops.

Since there is only a single Assisted System, ENI is concerned with managing and optimizing the performance of just a single Assisted System.

An ENI System is not aware of the number or type of control loops that the Assisted System has. Two typical types of control loops are shown in Option 1.

For informational purposes, the first type of control loop is to feedback information from the control loop to other Functional Blocks; this enables the control loop to affect the behaviour of multiple Functional Blocks (regardless of what type of task each Functional Block performs). The second type of control loop is to feedback information via the API. This enables Functional Blocks that are in different administrative domains to participate in behavioural decisions that are allowed by their containing administrative domain. This theoretically enables an ENI System to tap into the control loop of the Assisted System. However, this is not appropriate for some real-time decisions due to the communication and security constraints of using an API [2]. For example, a delay in communication caused by using the API (versus, for example, remote method invocation or other types of communication), as well as additional time required to secure the communication between two entities that are not co-located, is likely not going to be acceptable in certain scenarios.

4.4.2.5.2.3 Option 2 Communication

Option 2 emphasizes two key points:

- ENI is communicating with multiple Assisted Systems (compared to option 1).
- For some scenarios, ENI does not have to be aware that it is communicating with a particular Assisted System, whereas in other scenarios, it does.

In one set of scenarios, an ENI System does not differentiate between Assisted Systems. This is common for collaborative scenarios in which each Assisted System has the same capabilities and responsibilities. An ENI System registers or otherwise logs that it is communicating with specific Assisted Systems, as long as it is either communicating with them directly or is told that a particular Assisted System is communicating with it. For example, this may be necessary if a given Assisted System is acting as a Controller, and other Assisted Systems are dependent on the decisions of it. Another example is a peer-to-peer architecture. In a different set of scenarios, an ENI System needs to be aware of a specific Assisted System, since the set of Assisted Systems do not have the same capabilities and responsibilities.

4.4.2.5.2.4 Working as an External Discrete System

In both Class 3 alternatives, an ENI System also works as its own discrete system external to the Assisted System, as shown in Figure 4-4. However, most Assisted Systems of this class were not designed to accommodate new internal modules, so ENI needs to be incorporated as a discrete external system. The burden is on ENI (and the API Broker, in the case of APIs) to present its results in a form that the Assisted System (or its Designated Entity) are able to understand. Hence, nothing has to change in the Assisted System, or in its Designated Entity, in order for ENI to be used.

4.4.2.6 Summary of Interaction between the Assisted System and ENI

In each of the above classes of Assisted System, the ENI System interacts with either the Assisted System directly, or with the Designated Entity of the Assisted System. The Assisted System does not need to know if it is communicating with the ENI System or with the operator, an EMS, an NMS, or a controller. The point is to reduce the barrier of adoption for using an ENI System by not requiring new APIs from the Assisted System to interact with the ENI System. The use of an API Broker insulates the ENI System from having to know what entity it is specifically communicating with.

Common Features are shown in Table 4-1.

Table 4-1: Common Characteristics and Behaviour of Assisted Systems

Feature	Class 1 Assisted System	Class 2 Assisted System	Class 3 Assisted System
Assisted System Need to Change to Use ENI?	NO	NO	NO
AI Capabilities of Assisted System	NONE	Only in non-real-time decision making	Real- or non-real-time decision making
Scope of Assisted System APIs	None in Option 1, limited to non-AI functionality in Option 2	None in Option 1, limited to non-AI functionality in Option 2	Real- or non-real-time decision making
Communication Types	Option 1: through Operator or Designated Entity; Option 2: through APIs	Option 1: through Operator or Designated Entity; Option 2: through APIs	Option 1: communication with a single Assisted System using APIs; Option 2: communication with one or more Assisted Systems using APIs
ENI Operation as an External System to Assisted System	YES; embedding is not an option (see clause 6.2.3.1)	YES; embedding is not an option	YES; embedding is not an option
Directly Affect Control Loop?	NO	NO	YES
ENI is able to interact with modules of Assisted System	NO	NO	YES, as defined by the Assisted System

4.4.3 Communication and Interaction with Other External Systems

All standardized communication between the ENI System and the Assisted System (or its Designated Entity) uses External Reference Points, which are defined in clause 7 of the present document. The ENI System is defined as a set of Functional Blocks (see clause 6.3 and ETSI GR ENI 016 [i.35], clause 4.2). Functional Blocks define a recursive mechanism to represent functionality with well-defined inputs and outputs; if these inputs and/or outputs are meant to be interoperable, then they are defined as Reference Points. An External Reference Point (see clauses 4.4.6.1, 7.2 and 7.3) is a Reference Point between an ENI System Functional Block and an external system, whereas an Internal Reference Point (see clauses 4.4.6.2, 7.6 and 7.7) is a Reference Point between different ENI System Functional Blocks.

4.4.4 Mode of Operation

4.4.4.1 Allowed Modes of Operation

The ENI System operates in two different modes, called "recommendation mode" and "management mode".

The operation of the ENI System in recommendation mode is analogous to that of a Recommender system - it provides recommendations to the Assisted System. In contrast, when the ENI System is operating in management mode, the ENI System provides decisions and commands to be implemented by the Assisted System. These decisions and commands are subject to the approval of the Assisted System (or its Designated Entity).

The ENI System supports a recommendation mode and optionally supports a management mode of operation. A mixed mode of operation (i.e. the combination of recommendation and management modes) for different sets of decisions is also optionally supported. Both modes may be used with the same Assisted System. For example, if the Designated Entity of the Assisted System decides that recommendations belonging to a particular category have been verified, it then changes decisions for that category to operate in management mode; this is done independently of other categories, so that other operations stay in recommendation mode.

4.4.4.2 Setting the Mode of Operation

The ENI System initially sets the mode of operation. In order to do that, the ENI System selects its mode of operation based on the capabilities of the Assisted System (see [7] for a definition of capabilities from the MEF Core Model), along with other applicable information (e.g. regulatory policies, status of the infrastructure, and goals input by the operator) in a given context or situation. All standardized communication between the ENI System and the Assisted System is through Reference Points defined in the present document (see clause 4.4.6).

The ENI System optionally has a default mode of operation that is used when:

1) the ENI System is initialized; and/or

2) a class of decisions is reached that is not specified by the Assisted System.

For example, suppose the user has specified how to handle best effort traffic, but not how to handle prioritized service class traffic. In this example, if the ENI System has defined defaults that are appropriate to each service class, then best effort traffic is handled as specified by the user, while prioritized traffic is handled by the default specified by the ENI System. The default mode of operation is optionally changed as desired by either the Assisted System or its Designated Entity.

4.4.4.3 Interaction with the Assisted System

All communication regarding mode of operation change(s) is done using a request-reply interaction paradigm over an appropriate External Reference Point defined in the present document. This paradigm applies to any entity that requests a change (e.g. the User, the Designated Entity of the Assisted System, or an agent ([i.7] and [i.8]) acting on behalf of these).

In any mode of operation, the ENI System interacts with the data, control, management, and/or orchestration planes of the Assisted System, underlying infrastructure, and network. That interaction requires the Assisted System to expose new data for analysis by the ENI System; this is communicated as requests to change the configuration of the affected components, devices, and/or systems.

4.4.4.4 Selecting a Mode of Operation for a Class of Decisions

The Designated Entity of the Assisted System optionally selects a desired mode of operation for a class of decisions to the ENI System.

Alternatively, since the ENI System is an intelligent system, the ENI System optionally suggests that a particular mode of operation is used for a set of decisions that the Assisted System (or the Designated Entity on its behalf) has not specified or envisaged. In this case, the Assisted System needs to approve that it accepts ENI recommendations and/or management commands, as appropriate. These decision classes may be described by a set of capabilities (e.g. metadata) advertised by ENI.

4.4.4.5 Communication of Mode of Operation

Communication is done by the Designated Entity of the Assisted System, which by definition is a trust anchor [2] (a type of trusted entity). This means that the ENI System and the Designated Entity of the Assisted System collectively operate as expected, according to design and policy, despite environmental disruption, human user and operator errors, and attacks by hostile parties. In addition, in either mode of operation, this behaviour is validated by each entity before application.

4.4.4.6 Normal Operation of the Selected Mode of Operation

4.4.4.6.1 Overview

The ENI System notifies the Assisted System (or its Designated Entity) when it wants to change the mode of operation for a particular decision. It does so by sending a request to the Assisted System (or its Designated Entity) using an appropriate External Reference Point. These two options are described in clauses 4.4.4.6.2 and 4.4.4.6.3, respectively. If this mode of operation is not supported by the Assisted System for this decision, then the Assisted System (or its Designated Entity) informs the ENI System that this is not allowed. If this operation is supported by the Assisted System, then the Assisted System analyses the proposed change to determine if it is acceptable. If it is not, then optionally, negotiation is then initiated by either the Assisted System or the ENI System. Once agreed, the Assisted System (or its Designated Entity) informs the ENI System that the change is allowed. The ENI System operates the same way in either mode of operation; the difference is the set of outputs (information, recommendations, and commands) generated by the ENI System that are sent to the Assisted System.

All standardized communication with an ENI System is done using the External Reference Points specified in the present document. If an Assisted System is not capable of supporting some or all of these ENI External Reference Points, a possible solution is to use an API Broker to mediate between an ENI System and the Assisted System (or its Designated Entity) as described above.

A precondition of the entity requesting a change in the mode of operation is that it is a *trusted* entity (see requirements [MOP11] and [MOP12] in clause 5.3). The Designated Entity of the Assisted System chooses to either trust all decisions to be made, or to only trust decisions for a particular set of functions belonging to a particular category.

Ultimately, the Designated Entity of the Assisted System has the final decision regarding what mode of operation is used for any (and all) decisions. Conceptually, there is no difference between switching from recommendation to management mode or vice-versa. There are two cases to be considered, depending on which entity the ENI System communicates with, as detailed below.

4.4.4.6.2 Case 1: ENI Indirectly Instructs the Assisted System to Switch Modes

This case assumes that the Assisted System is not able to decide on its own whether it is able to switch modes, and always defers this decision to its Designated Entity. Hence, the ENI System communicates directly with the Designated Entity of the Assisted System, which in turn instructs the Assisted System to switch modes.

4.4.4.6.3 Case 2: ENI Directly Instructs the Assisted System to Switch Modes

This case assumes that the Assisted System decides on its own whether it is able to switch modes. Hence, the ENI System communicates directly with its Designated Entity; either the Assisted System or its Designated Entity informs its manager (either network manager or orchestrator) that it has switched modes.

4.4.4.7 Normal Operation of the Selected Mode of Operation

The default configuration of the ENI System is to start in recommendation mode when first booted. This mode of operation is defined for all types of decisions, since the ENI System has not been able to learn about its environment, services and resources that it is managing, as well as the preferences of the operator.

After booting, the ENI System then queries the Designated Entity of the Assisted System to determine the set of capabilities the Assisted System has, which Reference Points are defined, and what additional information and requirements (e.g. regulatory policies) are appropriate to be used. Given these initial inputs, the ENI System then asks the Designated Entity of the Assisted System if there are any types of decisions that it would like assistance with:

- If not, the ENI System is limited to passively learning through observing the actions of the Designated Entity of the Assisted System, and their effect on the infrastructure, as shared with the ENI System through information provided on the external Reference Points of an ENI System.
- If so, then the ENI System asks which mode of operation is desired for each class of decision.

For both cases, the ENI System will also ask if the Designated Entity would like to subscribe to notifications.

The Designated Entity of the Assisted System optionally defines a preferred mode for the ENI System to operate in for a given class of decisions. An ENI System acknowledges each such request, and also confirms when the requested mode of operation is ready to be used for each class of decisions.

Operation then continues with the ENI System learning through observation, except that the ENI System provides recommendations and/or commands based on the selected mode of operation for each decision defined above.

The ENI System notifies the Assisted System (or its Designated Entity) when it wants to change the mode of operation for a particular decision. If this mode of operation is not supported by the Assisted System for this decision, then the Assisted System (or its Designated Entity) informs the ENI System that this is not allowed. If this operation is supported by the Assisted System, then the Assisted System analyses the proposed change to determine if it is acceptable. If it is not, then optionally, negotiation is initiated by either the Assisted System or the ENI System. Once agreed, the Assisted System (or its Designated Entity) informs the ENI System that the change is allowed.

If the ENI System requires data that is not being monitored, it asks the Designated Entity of the Assisted System to change its configuration to report those data. The configuration change is required to enable data that are not currently being monitored to be made available to the ENI System.

4.4.4.8 Exception Handling for the Selected Mode of Operation

An exception is raised any time that:

1) a request or a response from either the Assisted System or an ENI System is not delivered; or

2) an ENI System is not able to change the mode of operation as requested.

4.4.5 Functional Concepts

4.4.5.1 Functional Concepts for Modular System Operation

NOTE: Clauses 4.4.5.1 to 4.4.5.9.2.3 are moved from the present document to a new document, titled "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]).

4.4.5.2 Overview of Prominent Control Loop Architectures

NOTE: This clause (previously clause 4.4.5.10) is moved from the present document to a new document, titled "Overview of Prominent Control Loop Architectures" (ETSI GR ENI 017 [i.36]).

4.4.6 ENI Reference Points

4.4.6.1 Definition of an ENI Reference Point

An ENI Reference Point is the logical point of interaction between specific Functional Blocks. Each Reference Point defines a set of related interfaces that specify how the Functional Blocks communicate and interact with each other.

4.4.6.2 Definition of an ENI External Reference Point

An ENI External Reference Point is a Reference Point that is used to communicate between an ENI Functional Block and an external Functional Block of an external system (e.g. a Functional Block of the OSS, BSS, or Assisted System).

4.4.6.3 Definition of an ENI Internal Reference Point

An ENI Internal Reference Point is a Reference Point that is used to communicate between two or more Functional Blocks that belong to the ENI System. This communication stays within an ENI System, and is not seen by systems that are external to an ENI System.

4.4.7 ENI Interfaces

4.4.7.1 Definition of an ENI Interface

An Interface is a point across which two or more components exchange information. An interface describes the public characteristics and behaviour that specify a contract for performing a service. In an ENI System, there are three types of Interfaces:

1) hardware interfaces;

2) software interfaces; and

3) application programming interfaces.

ENI Interfaces exist at defined External and Internal Reference Points (see clause 7 of the present document). ENI Reference Points differentiate between "consumer" and "producer" by separating data, control, management, and other specific types of communication information from being communicated on the same interface. See clause 7.1 of the present document for further details.

4.4.7.2 Definition of an ENI Hardware Interface

An ENI Hardware Interface is a point across which electrical, mechanical, and/or optical signals are conveyed from a sender to one or more receivers using one or more protocols. A Hardware Interface decouples the hardware implementation from other Functional Blocks in a system.

4.4.7.3 Definition of an ENI Software Interface

An ENI Software Interface defines a point through which communication with a set of resources (e.g. memory or CPU) of a set of objects is performed. This decouples the implementation of a software function from the rest of the system.

4.4.7.4 Definition of an ENI Application Programming Interface

An API is a set of communication mechanisms through which a developer constructs a computer program. It consists of tools, object methods, and other elements of a model and/or code. APIs simplify producing programs, since they abstract the underlying implementation and only expose the objects, and the characteristics and behaviour of those objects that are needed.

4.4.7.5 Comparison of ENI Software Interfaces with ENI APIs

An ENI Software Interface is a managed logical entity that supports communication of data, information, and/or knowledge. In contrast, an ENI API uses one or more ENI Software Interfaces to provide external access for developers to write programs to interact with ENI.

4.4.7.6 Interaction between ENI Hardware and Software Interfaces

Simplistically, software contains instructions that tells hardware modules what functionality is needed. Hardware interfaces include plugs, sockets, cables and set of electrical, optical, and/or logical signals traveling between the entities that it connects, along with the protocol(s) used. In contrast, Software interfaces perform programming functions. They can be used to communicate with hardware and other software modules. ENI does not specify a hardware-software interface except for those interfaces that support ENI APIs.

4.4.7.7 Interaction between ENI Hardware and Software APIs

An API sometimes reflects the language that it is meant to be used with. For example, an API for an object-oriented language would provide a specification for creating, editing, deleting, and manipulating classes. Other types of APIs provide specifications at different abstraction levels (e.g. for create, read, update, or delete operations) that are dependent on the object that they are being applied to. For example, a "service-orientated" API could apply operations such as the previous four to a service, while a "resource-orientated" API would do the same thing to resources. Note that if appropriate abstractions are used all of these examples are independent of hardware and software entities. ENI APIs operate through either an External or an Internal Reference Point, and are specified in clause 7.4.

4.5 Functional Architecture

4.5.1 Functional Block Diagram of the ENI System

A high-level Functional Block diagram that includes the use of an API Broker is shown in Figure 4-5 (for inputs to an ENI System) and Figure 4-6 (for outputs from an ENI System). This is a simplified view of the main processing components of an ENI System. It is important to realize that a linear flow from input to output is not prescribed, and does not have to actually occur. This is explained in the following clauses, and elaborated further in clause 6. The arrows in Figure 4-5 and Figure 4-6 represent the directionality of data and information using any of the eighteen External Reference Points defined in clause 7.

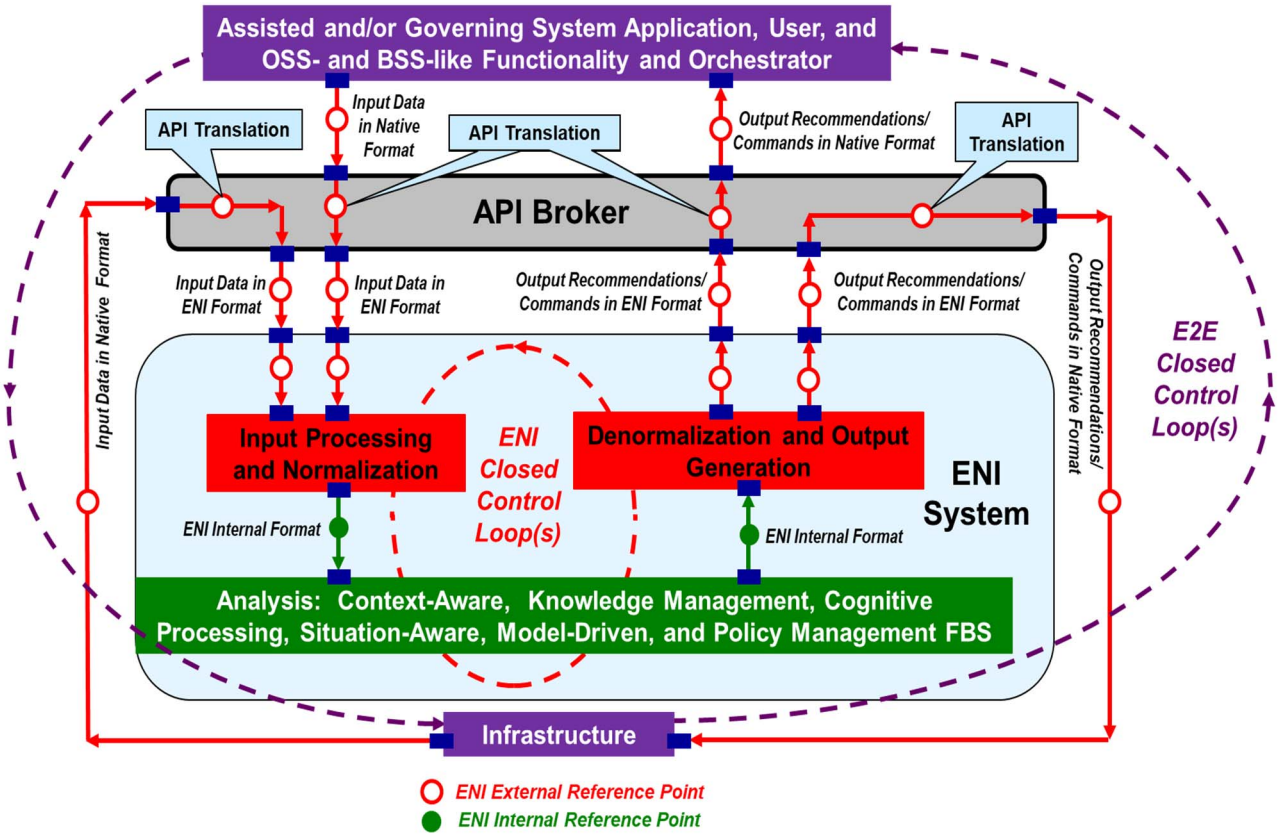

Figure 4-5: High-Level Functional Architecture of ENI When an API Broker Is Used

A high-level Functional Block diagram that includes the use of an API Broker is shown in Figure 4-5 (for inputs to an ENI System) and Figure 4-6 (for outputs from an ENI System). This is a simplified view of the main processing components of an ENI System. It is important to realize that a linear flow from input to output is not prescribed, and does not have to actually occur. This is explained in the following clauses, and elaborated further in clause 6. The arrows in Figure 4-5 and Figure 4-6 represent the directionality of data and information using any of the eighteen External Reference Points defined in clause 7.

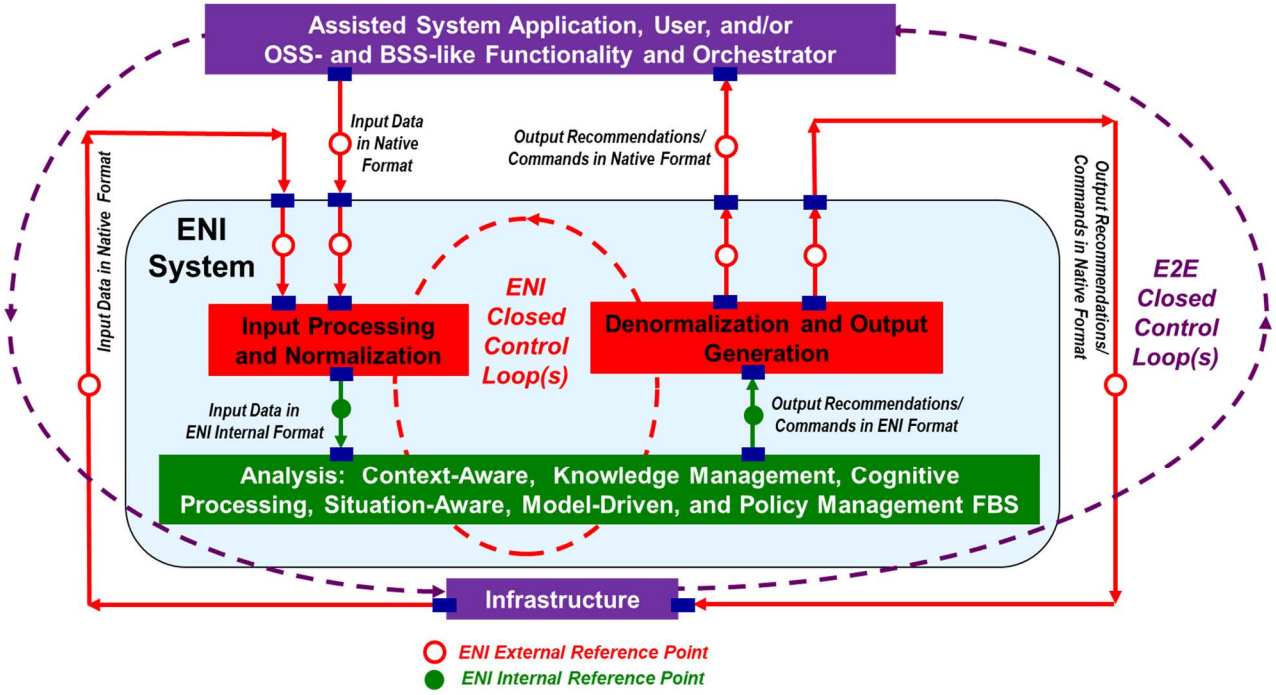

Figure 4-6: High-Level Functional Architecture of ENI When an API Broker is Not Used

The purpose of the API Broker is to serve as a gateway (i.e. translation mechanism) between different systems. There is currently no dedicated Functional Block to translate between the APIs and data formats used external to the ENI System and the APIs and data formats used internally by the ENI System. Specifically, the Input Processing and Normalization group of Functional Blocks assumes that the input from the Assisted System has been translated to a form that it can understand. This is because there are significant differences in the format and language used by vendors for both telemetry information as well as APIs. Similarly, the Denormalization and Output Generation group of Functional Blocks assumes that, due to the above heterogeneity in APIs and commands, another entity will perform the translation from ENI format to the Assisted System's Native format. Therefore, without the API Broker, it is possible that the ENI System is not able to communicate directly to external systems (in this case, the Assisted System and its Designated Entity, Applications, Users, and the Infrastructure).

The ENI System functional architecture consists of three types of Functional Blocks:

1) One or more Functional Blocks for ingesting and normalizing data.

2) One or more Functional Blocks for denormalizing and generating recommendations and/or commands.

3) One or more Functional Blocks for analysing ingested data, determining if the Assisted System is operating as expected, and if not, providing a set of recommendations and/or commands.

A key assumption is that the ENI System functionality evolves over time. It is anticipated that the capabilities of various Functional Blocks also evolve over time. As human operators gain confidence in automated decision-making, it is likely that ENI Functional Blocks operate with different and increased levels of autonomicity. Therefore, no attempt is made to standardize the interaction among internal blocks.

Figures 4-5 and 4-6 define the main Functional Blocks of the ENI System. Different use cases (refer to ETSI GS ENI 001 [3]) are realized by different flows through the above Functional Blocks, where any Functional Block is visited zero or more times.

ENI is a system that functions in two different modes, as described in clause 4.4.3. Both modes are depicted by showing the interaction of an ENI System as a separate system that gets data from, and provides information (including recommendations and/or commands) to the Assisted System (or its Designated Entity); the difference between the modes is whether commands are provided to the Assisted System (or its Designated Entity) or not. The ENI System itself consists of three or more Functional Blocks; these perform input processing, analysis, and output processing, respectively. An overview of these functions is described in the following clauses; see clause 5.3 for normative descriptions and requirements.

An ENI System operates in recommendation or command mode, or a combination of these.

NOTE: Any Functional Block (except the Data Ingestion, Normalization, Denormalization, and Output Generation Functional Blocks) is allowed to simultaneously operate in recommendation and/or command mode. However, this is not externally observable by the Assisted System (or its Designated Entity).

4.5.2 API Broker

4.5.2.1 Introduction

An API is a set of communication protocols, code, and tools that enable one set of software components to interact with either a human or a different set of software components. APIs are critical for platform and ecosystem development. [i.26] discusses this point, and says that API programs enable organisations to build a platform and associated ecosystem for digital transformation.

In general, each product has its own API. An API Broker is software that mediates between two systems with different APIs, enabling the two different systems to communicate transparently with each other. There are many benefits of using API Brokers, including error reduction via software transmitting data instead of humans and business process automation through automated transfer of data between applications. API Brokers also enable custom applications that integrate different application data.

4.5.2.2 Definition of the ENI API Broker

The ENI API Broker is used to translate between the ENI APIs and APIs of the Assisted System and its Designated Entity. More specifically, the ENI API Broker ingests APIs through an appropriate ENI External Reference Point, analyses the API, and then routes the functionality of the ingested API to an appropriate ENI Functional Block (this is typically the Data Ingestion Functional Block, but may be a different Functional Block if the API is known to an ENI System). This is discussed more in clause 6.4.

4.5.2.3 Use of an API Broker in ENI

One purpose of the API Broker is to decouple the ENI System from other external systems that it communicates with. In general, each Assisted System has its own unique functionality and set of APIs. This implies that the ENI System would have to understand each of these different sets of APIs in order to communicate with them. This is not desirable, and makes the building of an ENI System very complex. Instead, the present document defines an API Broker to aid in the translation between external systems and the ENI System. This enables the ENI System to define a single set of APIs. Therefore, the purpose of the API Broker is to:

1) translate data communicated from the Assisted System (or its Designated Entity) into a normalized form that all ENI Functional Blocks are able to understand; and

2) translate recommendations and commands from the normalized form used in an ENI System to a form that the Assisted System (or its Designated Entity) is able to understand.

Another purpose of the API Broker is to manage APIs. This includes the authentication and authorization of the entities that want to communicate using ENI APIs (e.g. between the ENI System and third-party applications, and vice-versa).

4.5.2.4 Alternatives to Using an API Broker

There is no alternative to using an API Broker. This is because an ENI System is designed to be used by a variety of existing and future systems that each can have their own unique APIs. The alternative of embedding API plug-ins for each targeted Assisted System directly in an ENI Functional Block (e.g. the Data Ingestion and Output Generation Functional Blocks) does not scale, and directly impedes the continued development of ENI. Rather, an extensible API Broker that itself has plug-ins for different targeted Assisted Systems, as well as generic API capabilities, is a preferred approach. It also enables API composition, which is discussed further in clause 6.4.

4.5.3 ENI System Functional Blocks

4.5.3.1 Introduction

The ENI System shown in Figures 4-5 and 4-6 has three types of Functional Blocks. They are:

- Input Processing.
- Analysis, which includes Knowledge Management, Context Aware Management, Cognition Management, Situation-Aware Management, Model-driven, Policy Management.
- Output Generation.

Different types of decision-making strategies that apply to all three types of ENI System Functional Blocks are described in clause 4.5.4.

4.5.3.2 Input Processing

4.5.3.2.1 Overview

Input processing consists of two Functional Blocks: the Data Ingestion and the Normalization Functional Blocks. All external data enters the Data Ingestion Functional Block. It is possible to combine the functionality of the Data Ingestion and Normalization Functional Blocks into a single Functional Block if desired.

4.5.3.2.2 Data Ingestion Functional Block

The purpose of the Data Ingestion Functional Block is to collect data from multiple input sources and implement common data processing techniques to enable ingested data to be further processed and analysed by other ENI Functional Blocks.

Data used by the Functional Blocks in an ENI System typically comes from different sources, is created using different applications and programming languages, and typically is ingested using different protocols. Data is produced by different entities, layers, and domains of the Assisted System (or its Designated Entity), and communicated to the API Broker of ENI via the exposed External Reference Points and/or APIs of the Assisted System (or its Designated Entity).

The API Broker may assist in parsing and possibly translating communicated data into a format defined by the Knowledge Management Functional Block.

It is possible to combine this Functional Block with the Normalization Functional Block if desired.

4.5.3.2.3 Normalization Functional Block

The purpose of the Normalization Functional Block is to process and translate data received from the Data Ingestion Functional Block into a form that other ENI Functional Blocks are able to understand and use. This enables the data used by the Functional Blocks in an ENI System to be interpreted and understood in a unified and consistent manner; this facilitates the reuse of these Functional Blocks, enables the modularization and generalization of the design of the other Functional Blocks, and supports vendor neutrality. The normalization optionally includes pre- and/or post-processing of the data from different domains. The normalized data is then passed to other Functional Blocks in the ENI System for further processing.

Different data models are likely to be used by different ENI Functional Blocks. Each such data model typically uses different data structures, objects, and protocols to represent its concepts. It is possible for the same concept to be represented differently in different data models (e.g. customer data in a Lightweight Directory Access Protocol (LDAP) or electronic Directory Service (Recommendation ITU-T 9594-1 [i.13] and Recommendation ITU-T 9594-7 [i.14]) vs. the same customer data in a Relational Database Management System (RDBMS)). This is addressed by ensuring that each data model is derived from a single information model, which facilitates reconciling these different representations of the same concept into a single object.

It is possible to combine this Functional Block with the Data Ingestion Functional Block if desired.

4.5.3.3 Analysis

4.5.3.3.1 Knowledge Management and Processing

4.5.3.3.1.1 Overview

Knowledge Management and Processing consists of three Functional Blocks: Knowledge Management, Context-Aware Management, and Cognition Management Functional Blocks. It is possible to combine the functionality of these three Functional Blocks into a single Functional Block if desired.

4.5.3.3.1.2 Cognition Model

Cognition is defined as "the process of understanding data and information and producing new data, information, and knowledge". A cognition model defines how cognitive processes, such as comprehension, action, and prediction, are performed and influence decisions. This is how, for example, behaviour is recognized.

A cognition model enables a cognitive system to reason about what actions to take in a methodological and structured manner. It can learn from its experience to improve its performance. It can also examine its own capabilities and prioritize the use of its services and resources, and if necessary, explain what it did and accept external commands to perform necessary actions. This includes:

1) revising (i.e. correcting) existing knowledge;

2) acquiring and encoding new knowledge from instruction or experience; and

3) combining existing components to infer and deduce new knowledge.

4.5.3.3.1.3 Knowledge Management Functional Block

The purpose of the Knowledge Management Functional Block is to represent information about both the ENI System as well as the system being managed. This includes differentiating between known facts, axioms, and inferences. There are many examples of knowledge representation formalisms, ranging in complexity from models and ontologies to semantic nets and automated reasoning subsystems [66]. It is likely that the ENI System Architecture relies heavily on logic-based as well as various inferencing mechanisms. A more complete and formal definition of knowledge representation is defined in clause 3.1. This Functional Block is used by all other Functional Blocks of the ENI System.

4.5.3.3.1.4 Context Aware Management Functional Block

The purpose of the Context-Aware Management Functional Block is to describe the state and environment in which a set of entities in the Assisted System (i.e. system being assisted and/or governed) exists or has existed. Context-aware management is used to continuously update the context in which decisions are made.

Context consists of measured and inferred knowledge, and typically changes over time. For example, it is possible that a company has a business rule that prevents any user from accessing the code server unless that user is connected using the company intranet. This business rule is context-dependent, and the system is required to detect the type of connection of a user, and adjust access privileges of that user dynamically.

A more complete and formal definition of context, and how it is used in management, is defined in [i.1], [i.2] and [i.3].

4.5.3.3.1.5 Cognition Management Functional Block

For the purposes of the present document, cognition is defined in clause 3.1 as the "process of understanding data and information and producing new data, information, and knowledge".

The purpose of the Cognition Management Functional Block is to enable the ENI System to understand normalized ingested data and information, as well as the context that defines how those data were produced; once that understanding is achieved, the Cognition Management Functional Block then evaluates the meaning of the data, and determines if any actions need to be taken to ensure that the goals and objectives of the system are met. This includes improving or optimizing performance, reliability, and/or availability. These metrics are exemplary - it is possible to use any other metric used for evaluation of the ENI and/or the Assisted System. The Cognition Management Functional Block mimics some of the processes involved in human decision-making to better comprehend the relevance and meaning of ingested data.

4.5.3.4 Situation-based, Model-driven, Policy Generation

4.5.3.4.1 Overview

Situation-based Policy Generation Functional Block consists of three Functional Blocks:

- Situation Awareness;
- Model-Driven Engineering; and
- Policy Management Functional Blocks.

4.5.3.4.2 Situation Awareness Functional Block

The purpose of the Situation Awareness Functional Block is to enable the ENI System to be aware of events and behaviour that are relevant to a set of entities in the environment of the Assisted System (i.e. system being assisted and/or governed). This includes the ability to understand how information, events, and recommended commands given by the ENI System impact the management and operational goals and behaviour, both immediately and in the near future. Situation awareness is especially important in environments where the information flow is high, and poor decisions have the possibility to lead to serious consequences (e.g. violation of SLAs). A thorough introduction to situation awareness is defined in [i.4].

4.5.3.4.3 Model Driven Engineering Functional Block

The purpose of the Model Driven Engineering Functional Block is to use a set of domain models that collectively abstract all important concepts for managing the behaviour of objects in the system(s) governed by the ENI System.

The use of reusable models defines a set of concepts that are shared by all constituencies that use them. It is permissible for a given constituency to use them directly or indirectly; examples of these are an application developer and a business user, respectively. It maximizes productivity by defining common definitions and usage of concepts used by different entities, including Functional Blocks and systems.

A model is made reusable through the use of common elements that are arranged in similar patterns, or templates. This practice is called design patterns, and occurs at multiple levels of abstraction. ENI is principally concerned with the use of software design patterns and architecture design patterns. A software design pattern is a general, reusable solution in a given context to a commonly occurring problem in the design of a software system. It is not a finished design that is able to be transformed directly into code; rather, it describes how to build the elements of a solution to a type of problem that commonly occurs in designing software. It typically shows concepts and interactions between those concepts; these are translated into classes, attributes, operations, and relationships. An architectural design pattern is similar to a software design pattern, but has a broader scope. It is not an architecture and not even a finished design; rather, it describes how to build the elements of a solution to a type of architecture problem that commonly occurs.

Model Driven Engineering is a software development and implementation methodology that creates reusable domain models, which are conceptual models of concepts, including how concepts relate to each other, which are important to the system being managed. It simplifies the design and implementation processes, and promotes a common understanding of terminology and concepts of the system by different teams working on the system. Put another way, a set of domain models abstract the concepts and activities to be managed. The MDE approach is meant to increase productivity by maximizing compatibility between Functional Blocks and systems through the reuse of standardized models.

A domain is modelled using existing and new design patterns wherever possible.

4.5.3.4.4 Policy Management Functional Block

The purpose of the Policy Management Functional Block is to provide decisions to ensure that the system goals and objectives are met (see clause 6.3.9 for more information on how decisions are made). Policies are used to provide scalable and consistent decision-making. Policies are generated from data and information received by the Knowledge Management and Processing set of Functional Blocks. Formally, according to [i.4], the definition of policy is:

> *"Policy is a set of rules that is used to manage and control the changing and/or maintaining of the state of one or more managed objects",* see also [7], [8] and [i.1].

Policies may be used in several ways in an ENI System:

- Policies are defined by an ENI System for managing, monitoring, controlling, and orchestrating behaviour of Functional Blocks in the Assisted System.
- Policies are defined by an ENI System to request changes in the Assisted System (e.g. for monitoring a new output).
- Policies that are input to an ENI System by an external entity (e.g. end-user or application) are subject to verification by the ENI System (e.g. they need to pass a parsing or compilation stage with no errors or warnings produced).

In each case, policies may represent goals, recommendations, or commands. Typically, any information to be conveyed to the Assisted System or its Designated Entity take the form of a set of policies. Each set of policies may be made up of one or more imperative, declarative, and/or intent policy. The details of policy definition, generation, and processing are defined in clause 6.3.9.

4.5.3.5 Output Generation

4.5.3.5.1 Overview

Output Generation consists of two Functional Blocks: Denormalization and Output Generation Functional Blocks. It is permissible that the functionality of these two Functional Blocks is combined into a single Functional Block if desired.

4.5.3.5.2 Denormalization Functional Block

The purpose of the Denormalization Functional Block is to process and translate data received from other Functional Blocks of the ENI System into a form that facilitates subsequent translation to a form that a set of targeted entities are able to understand. For example, different data models are likely to be used by different ENI Functional Blocks. Each such data model typically uses different data structures, objects, and protocols to represent its concepts. It is possible to represent the same concept differently in different data models (e.g. customer data in an LDAP or X.500 directory vs. the same customer data in an RDBMS). This is addressed by ensuring that each data model is derived from a single information model, which facilitates reconciling these different representations of the same concept into a single object.

The Denormalization Functional Block receives policies, recommendations, and/or commands. The Denormalization also optionally includes pre- and/or post-processing of the data from different domains to facilitate the translation of these data for a targeted set of entities. The denormalized data is then passed to the Output Generation Functional Block for further processing.

It is possible to combine this Functional Block with the Output Generation Functional Block if desired.

4.5.3.5.3 Output Generation Functional Block

The purpose of the Output Generation Functional Block is to convert data received by the Denormalization Functional Block into a form that the Assisted System (or its Designated Entity) is able to understand. This includes defining an appropriate set of protocols, changing the encoding of the data, and other related functions.

It is possible to combine this Functional Block with the Denormalization Functional Block if desired.

4.5.4 Decision-Making

4.5.4.1 Overview

Decision-making is the cognitive process of selecting a course of action from several possible alternative actions based on the information available and the preferences of the decision maker. Every decision-making process produces a final choice. The final choice does not have to be implemented immediately; this depends on the situation that the decision is made in.

The ENI architecture is specifically designed to provide collaborative interaction of each Functional Block. This enables the domain-specific functionality of each Functional Block to be reused by other Functional Blocks. This avoids having to build too much processing functionality in any one particular Functional Block, and more importantly, implementing the same function differently in different Functional Blocks. The ENI architecture combines localized and domain-specific knowledge to develop higher-level, global decisions.

The following clauses describe the progression of decision-making envisioned by the ENI System, from hindsight to reactive to predictive to (full) cognitive processing.

4.5.4.2 Decision-Making using Hindsight

Hindsight is defined as the "realization of the significance and nature of data and events after they have occurred". This realization is typically based on understanding, which in turn requires data, information, and knowledge; wisdom is of course very helpful and needed for all but simple and straightforward (i.e. easy to understand) occurrences. This type of decision-making dominated early management systems until more functionality was available.

4.5.4.3 Decision-Making using Deterministic Processing

Deterministic processing is a step beyond hindsight. Deterministic processing recognizes the occurrence of significant events and data, and invokes predefined rules, processes, and/or policies in response to correct behaviour. A simple example is an expert system, which defines rules to handle different inputs. This type of system responds very quickly and accurately to some events, because the input matches one or more rules that already exist. However, if an input does not match any rule, then this type of system has a problem - it has no choice to give, and no extensible way to make a decision. It has to wait for a new rule.

Put another way, this type of system reacts to past events, and cannot predict future behaviour. However, it represents a significant improvement over hindsight, which is only able to make a decision after the fact.

4.5.4.4 Decision-Making using Predictive Processing

Predictive processing uses a set of processes to predict, or estimate the likelihood of, a future event, state, or behaviour. The prediction or estimation is based on analysing current and historical facts in order to identify risks and opportunities to improve the likelihood of the prediction. Predictive processing also typically allows probability and/or risk assessment. Predictive models often perform calculations during transactions in order to estimate the risk or opportunity involved with making a decision; the actual decision is implemented by the predictive processing entity or some other entity.

4.5.4.5 Decision-Making using Cognitive Processing

A cognitive processing system uses cognition (see clause 6.3.6) to prove or disprove a hypothesis about how to correct an undesirable state in the system. Cognitive processing requires the tight integration of different mechanisms (e.g. knowledge, short- and long-term memory, perception, planning, and reasoning) that together understand what has happened and plan a corrective set of actions. This tight integration is in contrast to modern software architectures that emphasize loose coupling of Functional Blocks.

There is a significant difference between machine learning and cognitive processing. Some examples include:

- Machine Learning is based on concrete mechanisms, such as classification and statistical pattern recognition. In contrast, cognitive processing emphasizes abstract reasoning and problem solving.
- Machine Learning is based on dissimilar, discrete mechanisms. In contrast, cognitive processing emphasizes uniform and integrated representation of data and information.
- Machine Learning is based on pipelines of different functions. In contrast, cognitive processing emphasizes integrated architectures that provide seamless integration between various cognitive functions.

The ENI System architecture employs both traditional AI mechanisms as well as cognitive processes to achieve the best of both worlds.

4.5.5 Introduction to Artificial Intelligence Mechanisms for Modular Systems

NOTE: This clause is moved from the present document to a new document, titled "Introduction to Artificial Intelligence Mechanisms for Modular Systems" (ETSI GR ENI 018 [i.37]).

5 ENI Architectural Requirements

5.1 Introduction

The following clauses define requirements for the ENI system architecture.

NOTE: The numbering of these requirements in the present document has changed from ETSI GS ENI 005 [i.42].

5.2 Functional Architectural Requirements for ENI Operation

The ENI System Architecture shall meet the following requirements.

[FAR1] ENI shall operate as a model-driven system.

NOTE 1: See clause 5.6 for more information.

[FAR2] ENI shall operate using one or more closed control loops.

NOTE 2: See clause 5.7 for more information.

[FAR3] ENI shall use a modular architecture, realized as a set of Functional Blocks.

[FAR3.1] A Functional Block may consist of zero or more nested Functional Blocks, forming a hierarchy.

[FAR3.2] In a set of hierarchical Functional Blocks, a lower-level Functional Blocks shall implement functionality of a higher-level Functional Block in a modular fashion.

[FAR3.3] In a set of hierarchical Functional Blocks, a lower-level Functional Block need not implement functionality that is not needed by a higher-level Functional Block.

NOTE 3: [FAR3.2] and [FAR3.3] mean that a lower-level Functional Block only implements functionality required by a higher-level Functional Block, and cannot implement orthogonal functionality. Hence, the purpose of a lower-level Functional Block is to modularize functionality for better reusability, availability, performance, and other similar metrics.

[FAR4] ENI shall use and recognize administrative domains.

[FAR4.1] An Administrative Domain shall be governed by a common set of administrative policies.

[FAR4.2] An Administrative Domain may consist of zero or more nested Administrative Domains, forming a hierarchy.

[FAR4.3] In a set of hierarchical Administrative Domains, a lower-level Administrative Domain shall implement all administrative policies of all of its higher-level Administrative Domains.

[FAR4.4] In a set of hierarchical Administrative Domains, a lower-level Administrative Domain may implement new administrative policies that do not conflict with any administrative policies defined by any higher-level Administrative Domain.

NOTE 4: [FAR4.3] and [FAR4.4] mean that a lower-level Administrative Domain implements all administrative policies defined by a higher-level Administrative Domain. It cannot implement any administrative policies that conflict with any administrative policies defined by a higher-level Administrative Domain.

[FAR4.5] Operations between entities within an Administrative Domain may interact via Services defined within that Administrative Domain.

[FAR4.6] Operations between entities between different Administrative Domains shall interact via Services defined explicitly for Federated Domains.

[FAR4.7] The behaviour of Federated Domains shall be governed by special Policies, called Federation Policies.

[FAR4.8] Tasks should be assigned to Domains in a distributed Domain by Policies.

[FAR4.9] Task assignment should be based on mechanisms that match the functionality offered by a Domain to the needs of the tasks.

[FAR5] ENI may use agents, organized in various architectures, to implement various functions defined in one or more of its Functional Blocks.

[FAR5.1] A set of autonomous and/or intelligent agents may be used for data ingestion, normalization, denormalization, and output generation.

[FAR5.2] ENI may use a variety of agent architectures, ranging from simple reactive architectures to fully cognitive architectures.

[FAR5.3] ENI may use a distributed or hybrid agent architecture.

[FAR6] The ENI System may use negotiation to arrive at a decision that is satisfactory to both the ENI System and the entity that it is communicating with.

[FAR6.1] If negotiation is not used, or if it is used but fails, the Designated Entity of the Assisted may define a preferred mode for the ENI System to operate in.

[FAR6.2] If negotiation is not used, or if it is used but fails, the ENI System may have a default mode of operation that specifies the mode of operation to use when a class of decision is reached that is not specified by the Assisted System.

[FAR6.3] Negotiation shall include the ability for each entity participating in the negotiation process to discover the capabilities of all other entities that it is negotiating with.

[FAR6.4] Negotiation shall include the ability for each entity participating in the negotiation process to synchronize state with each other.

[FAR6.5] Negotiation shall include the ability for each entity participating in the negotiation process to agree on parameters and resources to use as part of the negotiation process.

5.3 Architectural Requirements for Mode of Operation

The ENI Mode of Operation shall meet the following requirements.

[MOP1] The ENI System shall support a recommendation mode.

[MOP2] The ENI System may support a management mode of operation.

[MOP3] The ENI System may support a mixed mode of operation for different sets of decision classes (e.g. some types of decisions are made using one mode of operation, and other sets of decisions are made using the other mode of operation).

[MOP4] The ENI System shall discover (and/or be told) the capabilities of the Assisted System in supporting the desired mode of operation.

[MOP5] The ENI System shall support and adapt to external inputs for each mode of operation.

[MOP5.1] The ENI System shall support and adapt to changes in the context and/or situation of the Assisted System.

[MOP5.2] The ENI System shall support external input of regulatory policies and operator goals.

[MOP6] The ENI System shall use the above two factors in [MOP5] to select its mode of operation.

[MOP7] The ENI System shall ask permission from the Operator or Designated Entity to change modes of operation using an agreed External Reference Point.

[MOP8] The Assisted System, or its Designated Entity shall ask the ENI System to change modes of operation using an agreed External Reference Point.

[MOP9] The ENI System shall confirm through the agreed External Reference Point to the Operator or Designated Entity of the Assisted System when it has successfully switched modes of operation.

[MOP10] The ENI System may suggest that a particular mode of operation is used when a class of decision is reached that is not specified by the Assisted System.

[MOP11] The Assisted System and/or its Designated Entity need not accept the recommendations offered by the ENI System when in recommendation mode. This includes those decisions that apply to recommendations when the Assisted System is in a mixed mode of operation.

[MOP12] The Assisted System and/or its Designated Entity need not accept the recommendations offered by the ENI System when in management mode. This includes those decisions that apply to commands when the Assisted System is in a mixed mode of operation.

[MOP13] Decisions and commands in management mode are subject to the approval of the Assisted System (or its Designated Entity).

[MOP14] The Assisted System (or its Designated Entity) may tell the ENI System that it approves all commands sent to it when it is in management mode.

[MOP15] The Assisted System may revoke the above setting at any time, in which case [MOP13] then applies (once acknowledged by the ENI System).

[MOP16] If the Assisted System (or its Designated Entity) rejects a command sent to it by the ENI System when it is in recommendation mode, it shall send a notification to the ENI System.

[MOP17] If the Assisted System (or its Designated Entity) rejects a command sent to it by the ENI System when it is in management mode, it shall send a notification to the ENI System.

5.4 Non-Functional Architectural Requirements for ENI Operation

ENI shall meet the following non-functional requirements.

[NFA1] All classes of Assisted System need not change in order to benefit from, and use the outputs of, the ENI System.

[NFA1.1] If an API Broker is used, then the ENI System need not be responsible for translating its recommendations and commands to a form that the Assisted System, of any class, can understand.

[NFA1.2] The ENI System may use an API Broker, or similar function, to perform the translation specified in [NFA1.1].

[NFA1.3] All classes of Assisted System need not change in order for the ENI System to ingest data from the Assisted System.

[NFA1.4] All classes of Assisted System need not change in order for the ENI System to send recommendations and commands to the Assisted System.

5.5 Reference Point Requirements

ENI shall meet the following requirements for Reference Points that it defines.

[RPR1] The ENI System shall use Reference Points defined in the present document (see clause 7) for all communication and interaction with the Designated Entity of the Assisted System.

[RPR1.1] The ENI System shall communicate with either the Assisted System or the Designated Entity of the Assisted System by using established ENI Reference Points.

NOTE 1: In both cases, the operator (or its Designated Entity) is in control.

[RPR1.2] The ENI System need not communicate with the Assisted System by using non-ENI Reference Points.

NOTE 2: Reference Points not defined by ENI are not within the current scope of the ENI Architectural Framework.

[RPR2] All External Reference Points shall provide common characteristics and behaviour.

[RPR2.1] The ENI System shall provide RESTful interfaces for each External Reference Point.

[RPR2.2] All External Reference Points may have the capability to define operations that work on lists of objects.

[RPR2.3] All External Reference Points shall provide standardized exception handling for all operations.

[RPR2.4] All External Reference Points may provide one or more mechanisms to process multiple messages at a given Reference Point in parallel.

[RPR3] All Internal Reference Points shall provide common characteristics and behaviour.

[RPR3.1] All Internal Reference Points may have the capability to define operations that work on lists of objects.

[RPR3.2] All Internal Reference Points shall provide standardized exception handling for all operations.

[RPR3.3] All Internal Reference Points may provide one or more mechanisms to process multiple messages at a given Reference Point in parallel.

[RPR4] All Internal and External Reference Points shall provide common messaging behaviour.

[RPR4.1] ENI shall employ standardized messaging patterns to promote interoperability.

[RPR4.2] ENI may use synchronous and/or asynchronous messaging for internal and external communication using Reference Points.

[RPR4.3] ENI shall use a set of dedicated message channels for communication over its Reference Points.

[RPR4.4] ENI shall use messages with appropriate properties to enable data to be exchanged over a channel.

[RPR4.5] ENI Functional Blocks may use routing to decouple a message sender from a message receiver.

NOTE 3: Communication can be from channel-to-channel, or employ different Functional Blocks to create more complex messaging flows.

[RPR4.6] ENI Functional Blocks may use filtering or similar operations to determine the destination of the message.

NOTE 4: Filtering enables a message to be forwarded or routed based on it matching certain criteria. This includes the presence or absence of fields in a message as well as their content.

[RPR4.7] ENI Functional Blocks may use point-to-point, point-to-multi-point, publish-subscribe, or receiver lists to communicate with recipients.

NOTE 5: Combinations of these (e.g. receive a message, split, process, and recombine) may also be supported.

[RPR4.8] ENI Functional Blocks may use a set of translation mechanisms to change the format and/or structure of the message to suit the needs of message receivers.

[RPR4.9] ENI Functional Blocks may use a set of enrichment mechanisms to change the content of the message to suit the needs of message receivers.

[RPR4.10] ENI Functional Blocks may use a set of patterns to consume messages.

[RPR4.11] ENI Functional Blocks may support durable messaging.

NOTE 6: For the purposes of the present document, durable messaging ensures that if a subscriber is temporarily disconnected when a message is sent, the subscriber is guaranteed to see this message when it reconnects.

[RPR5] All Internal and External Reference Points shall provide testing to verify that the operation occurring on that Reference Point was successful.

5.6 Knowledge Modeling Requirements

[KPR1] A domain shall be modelled using existing and new design patterns wherever possible.

[KPR2] ENI shall produce normalized data and information in a consistent format that all ENI Functional Blocks can understand.

[KPR2.1] ENI shall provide the ability to ingest structured, semi-structured, and unstructured data from different data sources.

[KPR2.2] ENI may provide the ability to ingest telemetry data in streaming and batch modes, as well as on-demand.

[KPR2.3] ENI shall pre-process all ingested data to provide a uniform and consistent input format.

[KPR2.4] The API Broker may help translate input data to be ingested.

[KPR2.5] The functions of data ingestion and normalization may be separated or combined.

[KPR2.6] ENI shall use context information to augment ingested data and information where applicable.

[KPR2.7] ENI shall provide the ability to change the sources of data to be ingested based on changing context information.

[KPR2.8] ENI shall use situation information to augment ingested data and information where applicable.

[KPR2.9] ENI shall provide the ability to change the sources of data to be ingested based on changing situation information.

[KPR2.10] ENI may use a set of models, including data types and data structures, to perform the transformation into a unified data format.

[KPR2.11] ENI may use a set of ontologies, along with either description logic or first-order logic, to perform the transformation into a unified data format.

[KPR3] ENI shall process new data and information using its stored knowledge.

[KPR3.1] If new data, information, or knowledge does not conform to the internal ENI models, then either the models and/or the new data, information, or knowledge shall be changed to realign them.

[KPR3.2] New data, information, and knowledge that is fully processed by one ENI Functional Block shall conform to the internal models held by the ENI System, at that moment in time, before being passed to another ENI Functional Block.

[KPR3.3] Alignment of new knowledge with existing knowledge may be done by any set of Functional Blocks at any time in the ENI architecture.

[KPR3.4] Data, information, or knowledge shall be semantically annotated.

[KPR4] ENI shall represent policies using an information model and one or more data models.

[KPR5] ENI shall represent contextual information using an information model and one or more data models.

[KPR6] ENI shall represent situational information using an information model and one or more data models.

[KPR7] ENI shall resolve perceived changes to existing data, information, and knowledge.

[KPR8] ENI may provide augmentation of information and data models using one or more ontologies.

[KPR8.1] ENI may use hypergraphs and/or multigraphs to perform the semantic augmentation process.

[KPR8.2] ENI shall use semantic closeness to determine which model elements are related to which ontologies when building a knowledge representation.

[KPR8.3] ENI shall use a semantic relationship to construct edges in a hypergraph or multigraph as part of the augmentation process.

[KPR8.4] ENI shall use a form of first order logic (including description logics) when it is required to prove a hypothesis.

[KPR9] ENI shall provide different Repositories for the storage, update, and retrieval of different types of data and information that require different types of access control.

[KPR9.1] ENI shall provide at least one Data Repository for the storage, update, and retrieval of collected data and information from different parts of the Assisted System after processing by the Data Ingestion and Normalization Functional Blocks.

[KPR9.2] One or more Data Repositories may be used for the storage, update, and sending of data and information to the Denormalization and Output Generation Functional Blocks.

[KPR9.3] ENI shall provide at least one Knowledge Repository for the storage, update, and retrieval of knowledge and wisdom used for making decisions.

NOTE: See ETSI GR ENI 016 [i.35] for further information and examples of knowledge and wisdom.

[KPR9.4] ENI shall provide at least one Model Repository storage, update, and retrieval of data models, ontologies, and the information model.

[KPR9.5] ENI shall provide at least one Blackboard Repository to provide a shared working space for computations required by different Functional Blocks.

[KPR9.6] All Repositories may be used to store and retrieve historical data and information of its appropriate type.

[KPR9.7] Enhanced security mechanisms shall be used to govern the access, update, use, and transmission of the information and data associated with any Model.

5.7 Control Loop Processing Requirements

[CLR1] ENI shall use one or more closed control loops to implement decisions to ENI Functional Blocks.

[CLR1.1] An ENI closed control loop shall operate on a clearly defined goal.

[CLR1.2] If ENI uses more than one closed control loop, each closed control loop may operate independently or as part of a group of closed control loops to accomplish a goal.

[CLR2] Real-time ENI closed control loop decisions may only be allowed for class 3 Assisted Systems.

[CLR2.1] Class 1 and Class 2 Assisted Systems may only use ENI recommendations and/or commands in non-real-time decisions.

[CLR2.2] The ENI System may only be involved in real-time control loop decisions for class 3 Assisted Systems.

[CLR3] ENI shall use one or more closed control loops as part of its main processing architecture.

[CLR3.1] ENI shall use at least one OODA-like closed control loop to implement decisions.

NOTE 1: See [i.1] for a detailed explanation of why OODA-based control loops, such as FOCALE, are superior to other types of control loops.

[CLR3.2] ENI may use other, non-OODA-like closed control loops for specialized purposes.

[CLR3.3] ENI may provide the ability to accelerate the processing through a closed control loop (i.e. not have to examine each of the constituent elements of the closed control loop).

[CLR3.4] ENI shall provide the ability to interrupt the processing of a closed control loop to perform other actions.

[CLR3.5] ENI closed control loops shall be able to request data, information, and knowledge from any Functional Block in the ENI System that it has access privileges for.

[CLR3.6] ENI closed control loops shall be able to supply decisions to any Functional Block in the ENI System that it has access privileges for.

[CLR3.7] ENI closed control loops shall be able to provide data, information, and knowledge in support of its decisions to any Functional Block in the ENI System that it has access privileges for.

[CLR3.8] An ENI closed control loop shall provide instructions in the form of an ENI policy.

NOTE 2: An ENI policy may be expressed in an imperative, declarative, or intent form, or as a combination of these forms.

[CLR3.9] An ENI closed control loop may accept instructions in the form of an ENI policy.

[CLR3.10] An ENI closed control loop shall log each goal and its final set of actions for that goal.

NOTE 3: Achieving a goal may require multiple iterations and multiple decisions per iteration. This requirement states that the goal and the final set of actions that either achieved the goal or concluded that the goal could not be achieved will be logged.

[CLR4] An ENI closed control loop shall be associated with one or more administrative or management domains.

[CLR4.1] An ENI closed control loop may span multiple administrative and/or management domains.

[CLR4.2] An ENI closed control loop in one administrative and/or management domain shall be able to request an ENI closed control loop in a different administrative and/or management domain to implement decisions on its behalf.

[CLR5] An ENI System need not be aware of the number of closed control loops that an Assisted System has.

[CLR6] An ENI System need not be aware of how commands that it sent to the closed control loop of an Assisted System are realized by the Assisted System.

NOTE 4: Commands sent by an ENI System may directly or indirectly affect the behaviour of a closed control loop of an Assisted System. In the former case, the commands are delivered directly to the closed control loop, while in the latter case, the commands are first processed by one or more Functional Blocks of the Assisted System before they are delivered to the closed control loop. In either case, the ENI System is unaware of these mechanics.

5.8 Functional Block Processing Requirements

5.8.1 Context Processing Requirements

[CTX1] Context shall be used for assigning different roles to perform context-sensitive behaviour and processing.

[CTX2] Context shall be used for establishing the identity of an object of interest that needs to interact with the ENI System.

[CTX3] Context shall be used for establishing the authentication, authorization, accounting, auditing, and similar privileges of an entity that needs to interact with the ENI System.

[CTX4] Context shall be used for determining what information and data should be retrieved for a given context, as well as how often the information and data need refreshing.

[CTX5] Context shall be used for determining the relevance of ingested information and data.

NOTE 1: ENI uses a model-driven architecture, which means that the current context, as well as changes to that context, can be mapped directly to model elements (e.g. objects, attributes, and relationships). This enables various types of functions, such as fuzzy logic, to be used to define and alter the relevance of ingested information and data.

[CTX6] Context shall be used for determining whether goals are being satisfied.

NOTE 2: Goals are represented as formal (i.e. mathematically well-defined) policies or rules. An ENI System may use various mechanisms, such as simple weighting or decision trees, as well as fuzzy logic, for measuring relevance and to determine if goals are being satisfied.

5.8.2 Cognition Requirements

[COG1] Cognition Processing in ENI shall use formal structures to model and represent each stage of the cognition process [7] and [i.4].

[COG2] A formal cognition model shall be used to guide the cognition processes used in an ENI System.

[COG3] Cognition shall be aware of its own capabilities.

NOTE 1: Cognition is the process of acquiring data and information, understanding the data and/or information, and producing new knowledge. Its previously stored information and knowledge, as well as modelled objects and behaviours, define the capabilities of an ENI cognitive System. This is similar to how Autonomic Systems are designed and implemented. See [4], [i.2] and [i.4].

[COG4] Cognition shall examine its own behaviour to ensure that inappropriate conclusions were not reached.

NOTE 2: Examination of recommendations and commands issued by an ENI System may be done using a variety of methods. For example, modelled information and data may be translated into finite state automata, and the projected action to be taken by an ENI System may then be matched to the actual behaviour observed.

[COG5] Cognition processing may be used for providing explanations as to which actions an ENI System took.

[COG6] Cognition shall use various types of repositories, such as ones dedicated for storing short- and long-term knowledge.

[COG7] Cognition processes shall continuously evaluate and dynamically update the knowledge it has.

[COG8] Cognition shall be used to meet, preserve, or protect the end-to-end goals of the system.

[COG9] An ENI System may use different processes, including first-order logic or computational linguistics, to derive or infer the understanding of a new fact or behaviour.

[COG10] An ENI System may store decisions, along with their stimuli, for future use.

[COG11] An ENI System shall review decisions to assess their accuracy and to determine if that was the best, or optimal, decision at that particular time.

5.8.3 Policy Management Requirements

[PMR1] Policies may be used to implement recommendations or commands.

[PMR1.1] ENI Policies may be defined as imperative, declarative, and/or intent policies.

[PMR1.2] ENI Policies shall be stored in a form that facilitates their reuse.

[PMR1.3] ENI Policies may be defined and implemented using one or more Domain Specific Languages.

NOTE 1: ENI DSLs are defined by developing a formal grammar that is used to formally verify the structure and correctness of the grammar, and hence, are testable by definition. The grammar of ENI DSLs contain modelled knowledge.

[PMR1.4] ENI Policies shall be used to recommend or define (depending on operational mode) desired behavioural changes to the Assisted System.

[PMR1.5] ENI Policies may be used to recommend or define desired behavioural changes to Functional Blocks of the ENI System.

[PMR2] ENI Policies shall incorporate any contextual changes.

NOTE 2: ENI Policies incorporate changes in context before they are sent to the Entities that they affect. This enables the behaviour of those Entities to adjust to changing context.

[PMR3] ENI Policies may incorporate situationally aware information.

NOTE 3: ENI Policies may incorporate situationally aware information before they are sent to the Entities that they affect. This enables the behaviour of those Entities to adjust to changing situations, such as changes in goals or threats.

[PMR4] The ENI System shall use DSLs to specify end-to-end goals to the Cognitive System.

[PMR5] The ENI System and the Assisted System shall agree on what types of Policy Rules are exchanged.

[PMR5.1] The ENI System and the Assisted System may use negotiation to specify the type and format of Policy Rules that are exchanged.

[PMR5.2] The ENI System and the Assisted System may specify the grammar of any Policy Rule DSL that will be exchanged between them.

[PMR5.3] The ENI System shall acknowledge the receipt of Policy Rules sent by the Assisted System.

[PMR5.4] The Assisted System shall acknowledge the receipt of Policy Rules sent by the ENI System.

[PMR6] The ENI System and the Assisted System shall agree on how Policy Rules are executed.

[PMR6.1] The Assisted System may request the ENI System to negotiate parameters to arrive at a mutually acceptable Policy.

[PMR6.2] The Assisted System shall report to the ENI System if any data, information, recommendations, or commands sent to it by the ENI System are not understood.

[PMR6.3] The Assisted System may ask for further clarifying information, including examples, if it has received data, information, or Policies from the ENI System that it does not understand.

[PMR6.4] The ENI System may request the Assisted System to negotiate parameters to arrive a mutually acceptable Policy.

[PMR6.5] The ENI System shall report to the Assisted System if any data, information, recommendations, or commands sent to it by the Assisted System are not understood.

[PMR6.6] The ENI System may ask for further clarifying information, including examples, if it has received data, information, or Policies from the Assisted System that it does not understand.

[PMR6.7] The ENI System shall record any comprehension problems details in [PMR6.2], [PMR6.3], [PMR6.5], and [PMR6.6] in its knowledge base.

[PMR6.8] The ENI System shall record the correct format and content to any comprehension problems found in [PMR6.7] in its knowledge base.

[PMR7] Policy Rules may have descriptive and/or prescriptive metadata attached.

5.9 AI Modelling and Training Model Requirements

[AIMT1] An ENI System shall incorporate methods to discover bias against individuals and groups of objects, ideas, and people.

[AIMT2] An ENI System shall be able to demonstrate that it uses a set of repeatable mechanisms as part of the machine learning process to help mitigate bias.

[AIMT3] An ENI System shall be able to demonstrate adherence to appropriate standards for showing that it can detect and mitigate bias.

[AIMT4] An ENI System shall analyse and provide feedback and/or explanation on ethical decisions.

[AIMT4.1] Agents used by an ENI System should use one or more mechanisms to enable their ethical decisions to explain their actions.

[AIMT4.2] An ENI System should augment its learning and reasoning mechanisms with whitelist and blacklist rules.

[AIMT5] An ENI System should use formal logic(s) to be able to mathematically prove hypotheses.

[AIMT6] An ENI System may use either an individual or a collaborative ethical decision-making framework.

[AIMT6.1] An ENI System shall use a formal cognition model for each intelligent agent that is used for ethical decision-making.

[AIMT6.2] An ENI System may use a logically separate mechanism to reconcile ethics-based requirements with the proposed actions of an agent before those actions are implemented.

[AIMT6.3] Individual ethical decision-making frameworks used by an ENI System shall enable agents to make ethical decisions, either on their own or in consultation with other agents.

[AIMT6.4] Individual ethical decision-making frameworks used by an ENI System shall be able to determine if an ethical decision made by another agent is in fact ethical or not.

[AIMT6.5] Collaborative ethical decision-making frameworks used by an ENI System shall form a network of trust to ensure that ethical decisions are free of uncertain and malicious behaviour.

[AIMT6.6] Collaborative ethical decision-making frameworks used by an ENI System may use a collection of specialized agents (e.g. some being deontological agents, and others being consequentialist agents).

[AIMT6.7] A learning mechanism should be used to develop strategies to compute a collective decision by these agents in a collaborative ethical decision-making framework.

[AIMT6.8] The learning mechanism in a collaborative ethical decision-making framework may include a voting strategy, where different agents are assigned different weights based on the type of the ethical decision to be made.

[AIMT7] An ENI System shall be compliant with any applicable standards that define how ethics should be processed.

[AIMT7.1] An ENI System shall protect privacy data.

[AIMT7.2] An ENI System should make as many of its decisions as possible to be transparent.

[AIMT7.3] An ENI System should be able to represent the needs of an end-user (e.g. act as a proxy for the user in machine-to-machine decisions).

5.10 API Requirements

[API1] An ENI System should use an API Broker to communicate with external entities such as the Assisted System.

[API2] An ENI System may interact with one or more Assisted Systems.

[API2.1] An ENI System may interact with APIs and/or the Designated Entity for class 1 and 2 Assisted Systems.

[API2.2] An ENI System should use APIs to interact with multiple Assisted Systems based on the communications architecture used by the Assisted Systems, as specified in [API3] or [API4].

[API3] An ENI System should interact with one or more collaborating Assisted Systems by sending the same API commands to each of the collaborating Assisted Systems.

NOTE: "Collaborating" means that each of the Assisted Systems has the same capabilities and responsibilities.

[API3.1] An ENI System need not know that it is communicating with a specific Assisted System.

[API3.2] An ENI System need not be aware of the number of collaborating Assisted Systems that it is sending recommendations and commands to.

[API3.3] An ENI System may use any communication mechanism that ensures that the same recommendations and commands are sent to each collaborating Assisted System.

[API3.4] The interactions between the API module in each Functional Block of each collaborating Assisted System need not be visible to an ENI System.

[API4] An ENI System shall know the identity of each non-collaborating Assisted System that the ENI System is directly connected to, if that Assisted System wants to receive recommendations and/or commands.

[API4.1] An ENI System shall be aware of the capabilities and responsibilities of each non-collaborating Assisted System that it is directly connected to.

[API4.2] An ENI System should be aware of the number of non-collaborating Assisted Systems that it is sending recommendations and commands to.

[API4.3] An ENI System may be able to indirectly control non-collaborating Assisted Systems that it is not directly connected to.

[API4.4] An ENI System need not be aware of its control of the behaviour of any non-collaborating Assisted Systems that it is not directly connected to.

[API4.5] The interactions between the API module in each Functional Block of each non-collaborating Assisted System need not be visible to an ENI System.

[API5] An ENI System shall include API Management.

[API5.1] An ENI System shall include authentication of APIs.

[API5.2] An ENI System shall include authorization of APIs.

[API5.3] An ENI System shall include accounting of APIs.

[API5.4] An ENI System shall include auditing of APIs.

6 ENI Reference Architectural Framework

6.1 Introduction

This architectural framework describes the ENI System as a set of Functional Blocks. Each Functional Block is described in terms of its inputs, outputs, state, and optionally, transfer function. This specifically means that this architectural framework does not define a specific implementation.

6.2 Design Principles of the ENI System architecture

6.2.1 Overview

The following are generic design principles that shall be followed in the design of the ENI System Architecture:

1) The internal implementation of a Functional Block shall not be defined.

2) Functional Blocks may be nested, where the nested Functional Blocks provide greater detail for the containing Functional Block. See clause 6.2.2 for more information.

3) Management, control, interaction with the Assisted System and orchestration of the ENI System shall be defined in terms of Reference Points that may use APIs and/or DSLs.

4) The ENI System may provide management, control, and/or orchestration commands and recommendations to the Assisted System.

5) Architectures from other SDOs shall be interfaced to using a subset of dedicated Reference Points for that purpose.

6) It is desirable for the ENI System to communicate with external actors. To maximize the ease with which this is accomplished, the ENI System shall translate external policies, data, services, and other information to a form it can understand, and shall not rely on external actors understanding its internal functionality. Similarly, outputs from the ENI System shall be translated to a form that external actors can consume. An ENI System shall use an API Broker to insulate the ENI System from having to know what entity it is specifically communicating with.

7) Any Functional Block may use negotiation to achieve its goals. See clause 6.2.3.5 for more information.

8) ENI shall use role-based modelling [i.5] and [i.6] to enable different functions and services to be viewed, accessed, and managed. This includes individual and groups of users, objects, Functional Blocks, and applications representing these entities.

9) ENI shall use one or more closed control loops as part of its main processing architecture. The primary control loop shall be at least one OODA-like closed control loop. See clause 5.7 for control loop requirements.

6.2.2 Nesting of Functional Blocks

Figure 6-1 shows a simplified notation (compared to UML) for Functional Blocks. The following terms are defined:

- Nested Functional Block: a Functional Block that is contained by another Functional Block.
- External Reference Point: a Reference Point that is external to this Functional Block.
- Internal Reference Point: a Reference Point that is internal to this Functional Block.
- Port: an interaction point between the Functional Block and its environment.
- Logical Connectives: logical AND, OR, and NOT operators may be used to define flow paths.

Ports may be uni- or bi-directional; this is indicated with one or two arrows, respectively.

Two examples of nested Functional Blocks are shown in Figure 6-1. There are two top-level Functional Blocks, named A and B. A contains two nested Functional Blocks, named A.1 and A.2. B contains two nested Functional Blocks, named B.1 and B.2; B.1 contains two nested Functional Blocks, named B.2.1 and B.2.2.

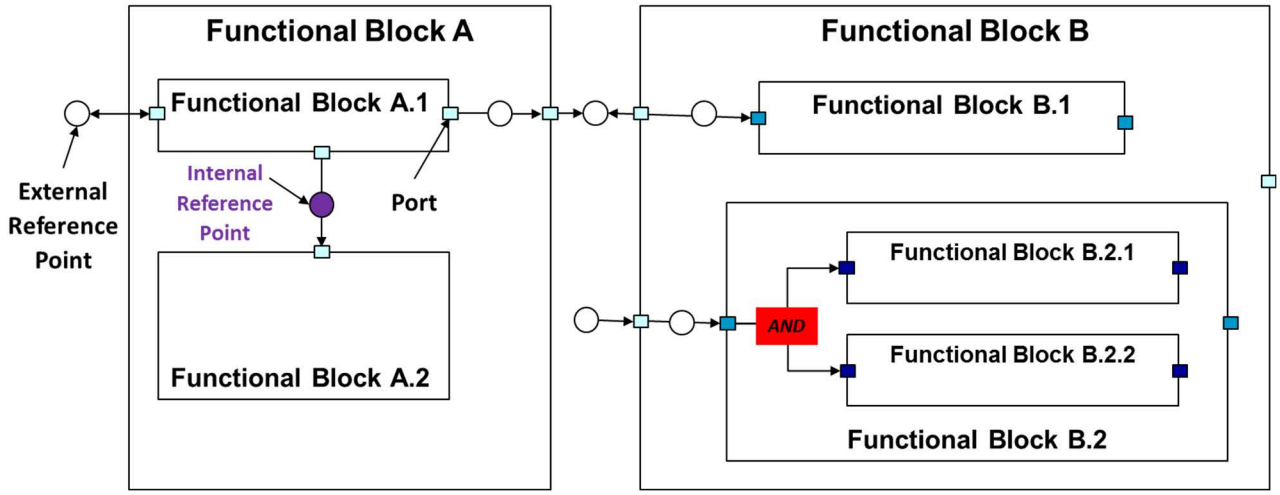

Figure 6-1: Functional Block Notation

In text form, using indentation to indicate nesting, Figure 6-1 may be represented as follows:

- Functional Block A:
  - Functional Block A.1.
  - Functional Block A.2.
- Functional Block B:
  - Functional Block B.1.
  - Functional Block B.2:
    - Functional Block B.2.1.
    - Functional Block B.2.2.

In Functional Block B, a logical AND connector is used to indicate that the input interface (at the level of Functional Block B) shall connect to both Functional Block B.2.1 and Functional Block B.2.2. Logical connectives are not limited to occurring within a Functional Block; they may control flows between any Functional Blocks at any level.

Administrative and Management Domains may be associated with multiple Functional Blocks. For example, in Figure 6-1, one configuration could be that Functional Blocks A and B (and their constituent Functional Blocks) could be part of the same Administrative or Management Domain. As another example, Functional Block A.1 and Functional Block B.2 could be part of the same Administrative or Management Domain.

Closed control loops may also be associated with multiple Functional Blocks, in the same way as described above for Administrative and Management Domains.

6.2.3 Communication and Interaction

6.2.3.1 Introduction

All standardized communication between the ENI System and the Assisted System (or its Designated Entity) shall use External Reference Points, which are defined in clause 7 of the present document. The ENI System shall be defined as a set of Functional Blocks (see clauses 4.5 and 6.3). Functional Blocks define a recursive mechanism to represent functionality with well-defined inputs and outputs; if these inputs and/or outputs are meant to be interoperable, then they shall communicate using an ENI Reference Points. An External Reference Point (see clauses 4.4.6.1, 7.2 and 7.3) is a Reference Point between an ENI System Functional Block and an external system, whereas an Internal Reference Point (see clauses 4.4.6.2, 7.6 and 7.7) is a Reference Point between different ENI System Functional Blocks.

Communication is not limited to a single mechanism: direct (e.g. point-to-point) and/or indirect (e.g. systems connected by a bus) communication may be used for any ENI Reference Point.

Communication shall be performed between two or more trusted entities, and forms a trusted system [2] (see requirement [MOP11] in clause 5.3).

An ENI System shall be able to communicate and interact with Assisted Systems as an external system. The current ENI System need not communicate as an entity embedded in an Assisted System. This is because all of the ENI External Reference Points would then have to have two flavors, or profiles: one for normal operation as an external means of communication, and one that is used only for internal communication.

NOTE: It is currently possible to co-locate (physically and/or logically) an ENI System with other systems (e.g. an OSS). This enables the ENI System to keep using its defined External Reference Points in a consistent manner.

In general, the ENI System shall acknowledge all commands and information given to it by the Assisted System or its Designated Entity. Note that only the Designated Entity behaves as a trusted entity. The ENI System sends notifications of other important events, such as completing the change to a new mode of operation, to the Designated Entity.

6.2.3.2 Discovery

A basic assumption is that when a device, application, or system starts up, it has no information about any peer devices, the network structure, or what specific set of roles [i.1], [i.4] and [i.6] it is assigned to play. Therefore, the discovery process should be repeated as often as necessary in order to find peers that support each function required. The discovery process may be event-driven.

The discovery process also needs to be flexible enough to accommodate different topologies. The ENI System needs to understand the capabilities of the Assisted System so that it may provide recommendations and commands for functions that involve those capabilities.

EXAMPLE: The ENI System requires data to be monitored from the Assisted System in order for the ENI System to supply recommendations and commands. If the Assisted System changes its functionality, then the System Architecture should be notified of such changes so that the monitored data may be adjusted.

The understanding of these capabilities shall be used to support appropriate recommendations and commands for the desired mode of operation.

6.2.3.3 Direct Communication

ENI Functional Blocks may support direct (e.g. point-to-point) communication through sequences of messages.

EXAMPLE: Routing protocols use a model based on distributed devices that communicate repeatedly with each other. Current routing protocols mainly consider simple data, such as link status. This is a form of information synchronization between peers.

6.2.3.4 Indirect Communication

ENI Functional Blocks may support indirect (e.g. systems connected by a bus, proxy, or broker) communication of control and management parameters through sequences of messages.

6.2.3.5 Negotiation

6.2.3.5.1 Introduction

Negotiation is fundamentally a decentralized process that requires at least two entities, one designated as the buyer and one or more designated as sellers. It is fundamental to various architectures, including message-driven and reactive architectures (see https://www.reactivemanifesto.org/).

ENI shall use role-based modelling to identify buyers and sellers. ENI entities, such as services available in a Functional Block, may participate as a buyer and/or a seller when negotiating externally (i.e. with the Assisted System or its Designated Entity) or internally (i.e. with other Functional Blocks that are in different administrative domains of an ENI System).

6.2.3.5.2 Distributive Negotiation

Distributive negotiation may be used by the buyer and/or the seller when either wants to gain as much in the negotiating process. Distributive negotiation is a zero-sum game (i.e. each agent's gain or loss is balanced by the losses or gains of the other agents). The participants may adopt different fixed positions (which may be extreme in nature, such as being overly expensive); then, each seeks to give as little as possible before reaching a deal. This encourages one participant to view the other as an adversary, rather than a partner. Put another way, distributive negotiation assumes that there is a fixed amount of value (which could also be services and/or resources) to be divided between the bidding agents.

6.2.3.5.3 Integrative Negotiation

Integrative negotiation may be used by the buyer and/or the seller when either wants to form partners in the negotiating process to ensure that it, and its partners, can maximize their gain. Integrative negotiation is a win-win (or non-zero-sum) game (i.e. all participating agents can profit). In theory, this improves the quality and likelihood of negotiated agreement by taking advantage of the fact that different parties often value various outcomes differently. Integrative negotiation may have some distributive elements, especially when different agents value different items the same, or when critical details are left to be allocated at the end of the negotiation. Integrative negotiation may involve a higher degree of trust and the formation of relationships, which enables each collaborating agent to "win". Hence, a good agreement is not one with maximum individual gain, but rather, one that provides optimum gain for all collaborating agents. Put another way, integrative negotiation attempts to create value in the course of the negotiation by either compensating the loss of one item with the gain of another item, or by restructuring the contract to specifically enable all collaborating agents to benefit.

6.2.3.5.4 Functional Model: an Informative Example

A simplified version of a market-based distributed negotiation model is as follows:

- An agent declares itself as a buyer.
- The buyer publishes one or more contracts, where each contract includes a specification of the set of tasks to be performed. The specification encodes a description of the task, any constraints, and metadata information.
- Agents that receive the contract request then decide if they want to bid on the contract. If so, then:
  - Each bidding agent announces itself as a seller.
  - Each seller then replies to the contract by either accepting the contract as is, or presenting a counter-proposal. The counter-proposal may include differences on one or more parameters in each task in each contract.
  - Any seller may generate a set of sub-contracts to help it meet the terms of the contract. If this is done, then the seller becomes a contract-buyer, and follows the above steps until it can satisfy the original contract.

- The (original) buyer then may either accept a seller's bid, or provide a counter-proposal, to all agents that submitted a bid on the contract.

6.2.3.5.5 Usage

ENI Functional Blocks may support negotiation of control and management commands. Negotiation is fundamental to agent interaction, as it enables interaction between entities that have different functions, and helps entities arrive at a mutually agreed upon compromise in functionality. It also may be used to negotiate data and commands exchanged between an ENI System and an external entity.

In the context of an ENI System, a negotiation process between digital agents of each system achieves agreement on the use of services and/or resources; this may also include a set of rules governing the use of each.

ENI is an experiential system, which means that it constantly learns while it is operating. This means that if the context in which an ENI System is operating changes the behaviour and functionality required by an ENI System, then the appropriate ENI Functional Blocks also change dynamically.

Therefore, static configuration of the behaviour of ENI Functional Blocks is not desirable, since the provided functionality cannot change in response to changing context and/or situations. Negotiated behaviour enables each negotiating entity to customize the functionality being negotiated dynamically at runtime. More importantly, it ensures that the collaborating entities arrive at a mutually agreed-upon solution. For example, suppose that an ENI System has a network slice management system to deliver sets of services to different customers from the same networked infrastructure. It is possible to use AI-based solutions to dynamically analyse current operational performance, as well as trends in that performance, and recommend or change network resources and services to adapt to changing user needs, business goals, and environmental changes. In this example, the benefit of negotiation is efficiency. For example, if the Functional Blocks that are supplying the changed behaviour have multiple programmable parameters, negotiating a new service is likely to be more effectively done by asking for a desired overall service and using negotiation to fine-tune applicable parameters. This also enables the negotiation to optionally include additional compensation, such as credit issued.

The negotiation process is a request-response message exchange pattern that is guaranteed to terminate with success or failure. More specifically, negotiation is a process by which two or more entities iteratively interact to agree on parameter settings that best satisfy the objectives of all participants.

Robust negotiation processes include loop prevention, time-outs, and generic tie-breaking rules for each parameter that is being negotiated. In addition, if negotiation is used in ENI, then the following additional communication requirements are typical:

- ability for each negotiating entity to discover each other;
- ability for each negotiating entity to synchronize state with each other;
- ability for each negotiating entity to agree (negotiate) on parameters and resources to use directly with each other.

If negotiation is not used, or if it is used but is not successful, architectural requirements [MOP11] and [MOP12] of clause 5.3 apply.

6.2.4 Administrative and Management Domains

6.2.4.1 Introduction

A domain is a collection of Entities that share a common purpose (e.g. has a set of common characteristics and/or behaviours and/or roles). Each constituent Entity in a Domain is both uniquely addressable and uniquely identifiable within that Domain.

ENI uses two types of domains: Administrative Domains and Management Domains.

An Administrative Domain is a Domain that employs a set of common administrative processes to manage the behaviour of its constituent Entities. A Management Domain is an Administrative Domain that uses a set of common Policy Rules as the management mechanism to govern its constituent Entities.

NOTE: A Management Domain refines the notion of a Domain by adding three important behavioural 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 behaviour of MCMManagedEntities contained in the MCMManagementDomain. This is based on the definition of an MCMDomain in [7].

6.2.4.2 Domain Operations

An Administrative Domain shall be governed by a Domain Authority. A Domain Authority is a set of management entities that determines the Policies for a given Domain (and all child Domains of the given Domain). The Domain Authority shall have the ability to delegate and revoke this ability, on a Domain-by-Domain basis, to another set of management entities. ENI shall model a Domain Authority as a set of Roles.

An Administrative Domain shall use a set of management processes to govern the behaviour of its constituent entities. If Policy Rules are used, then an Administrative Domain is considered a Management Domain.

Policies associated with a given Management Domain govern the behaviour of that Management Domain. All child Management Domains of this given Domain shall implement the same behaviour, and shall not implement behaviour that conflicts with the behaviour, defined in a parent Management Domain. In the present document, Policies conflict with each other if, for the same stimulus, they produce different behaviours. Any type of Policy Rule (e.g. imperative, declarative, intent) may be used to specify the behaviour of a Domain and entities within that Domain. The behaviour of a Management Domain includes any operation whose execution is specified using a Policy. Exemplary operations include the creation, reading, updating, managing, and deleting of objects (and their attributes), and invoking of methods in information and data models (and associated implementations), as well as defining access privileges to data, information, and knowledge within a Management Domain.

From this point forward in the present document, the term "Domain" shall refer specifically to a Management Domain unless otherwise indicated.

6.2.4.3 Interaction between Hierarchical Domains

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). Examples are shown in Figure 6-2.

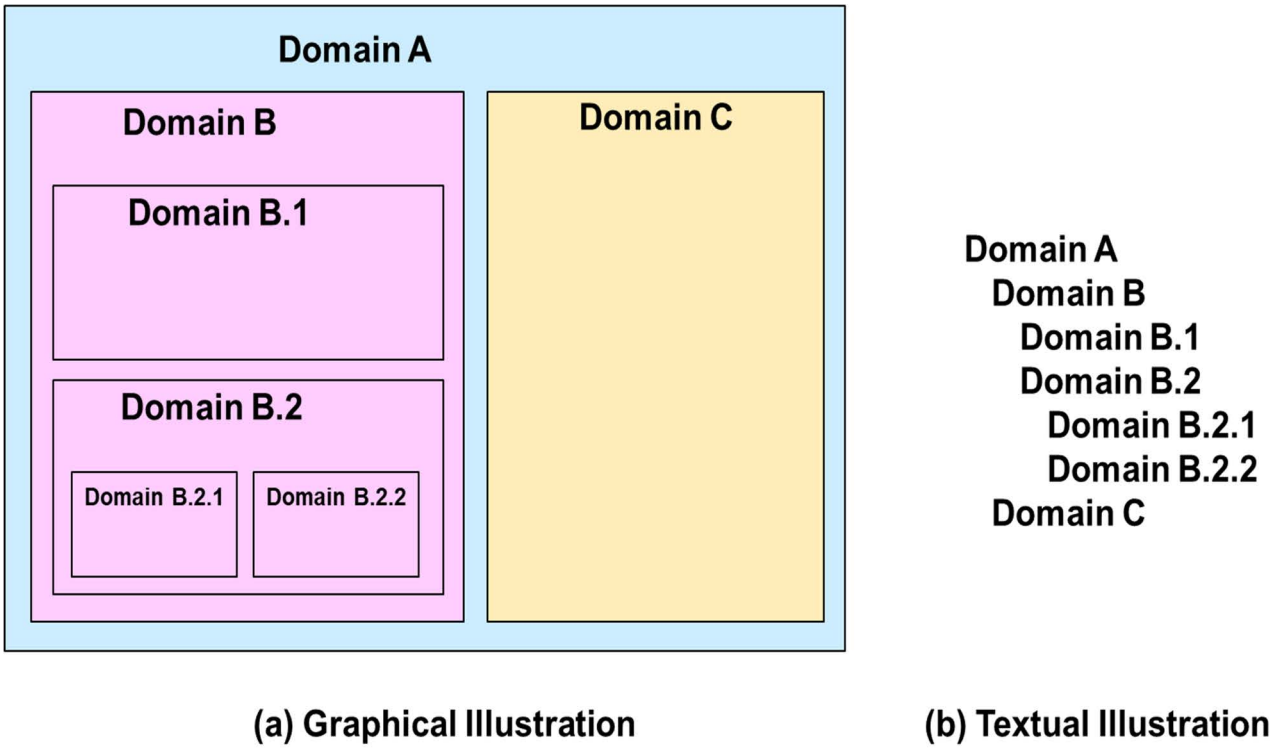

Figure 6-2: Exemplary Domain Relationships

In Figure 6-2, Domain B contains two child Domains, B.1 and B.2. Conceptually, these Domains are at the same level, and are called sibling Domains. Domain B.2 also contains 2 child Domains, B.2.1 and B.2.2. Thus, containment relationships shown in Figure 6-2 are summarized as follows:

- Domains B and C are both child Domains of A and sibling Domains (of each other).
- Domains B.1 and B.2 are also sibling Domains.
- Domain B.2 is a child Domain of Domain B and the parent Domain of B.2.1 and B.2.2.
- Domains B.2.1 and B.2.2 are aso sibling Domains.

For this example, the following rules apply concerning all types of Policies that are applied to these Domains:

- Domain A shall be governed by a common set of administrative Policies (see requirement [FAR4.1] in clause 5.2).
- If Domain A defines a Policy P1, then all child Domains of Domain A shall implement P1 (see requirement [FAR4.3] in clause 5.2).
- In addition, all child Domains of Domain A shall not define a Policy P2 that conflicts with P1 (see requirement [FAR4.4] in clause 5.2).
- Any child Domain of Domain A may define zero or more new administrative Policies; however, each new administrative Policy defined by a child Domain shall not conflict with any administrative Policy in any parent Domain of that child Domain (see requirement [FAR4.4] in clause 5.2).
- Sibling Domains to Domain A (not shown) shall use federation Policies to govern how the Domains interact; examples include controlling access to and communication with Domain entities, and deciding to work together (e.g. in a master-slave, 3-tier, collaborative, distributed, or other manner) (see requirement [FAR4.7] in clause 5.2).

6.2.4.4 Interaction between Distributed Administrative Domains

A distributed set of Domains is a set of Domains whose components work together to achieve a set of common goals. One or more communication mechanisms are used to coordinate actions between the Domains. Since each Domain in a set of distributed Domains typically has only a limited and incomplete view of the system, the components of each Domain in a distributed system have access to their own distinct Domain memory. Data may be local to a given Domain. In addition, some data may be shared across multiple Domains.

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.

Policies should be used to assign tasks to appropriate Domains (see requirement [FAR4.8] in clause 5.2). Task assignment should use one or more mechanisms, such as metadata, to guide the assignment of tasks to Domains (see requirement [FAR4.9] in clause 5.2). For example, the concept of a Capability is defined in [7]. A Capability is a type of metadata, and represents a set of features that are available to be used from a managed entity. These features may include all, or a subset, of the available features of the managed entity. These features may, but do not have to, be used. 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.

Distributed Domains also enable failures in all or part of a Domain to be more easily tolerated.

6.2.4.5 Interaction between Federated Administrative Domains

A federated set of Domains is a set of Domains that use formal agreements to govern their interaction and behaviour. 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.

Data is typically local for each Domain in a Federated Domain. Each domain may have access to common shared data, information, and knowledge, but are limited in their ability to edit or remove these common shared data, information, and knowledge. For example, each Domain may be given the same starting information model and/or ML model, but can only send its own suggested updates to a Governing Authority that is responsible for implementing those changes.

Policies should be used to govern the behaviour of each Domain in a federated Domain (see requirement [FAR4.7] in clause 5.2). This includes determining which Domains in a federated Domain assist which other Domains on a given task or goal.

Entities in each federated Domain may communicate with each other. External users and applications interact with each Domain independently (i.e. there is typically no mechanism, such as a proxy, that enables the same user or application to talk to all federated Domains together).

6.2.5 Modelled Knowledge

NOTE: This entire clause has been moved from the present document to a new document, titled "Representing, Inferring, and Proving Knowledge in ENI", and will be specified in a future release.

6.2.6 Bias

6.2.6.1 Introduction

Bias is defined as "the systematic difference in treatment of certain objects, ideas, or people in comparison to others". In this definition, *systematic* means that the difference in treatment is predictable and typically constant.

The different types of Bias are defined in ETSI GR ENI 018 [i.37], clause 4.4.2.

6.2.6.2 Protection Against Bias

An ENI System shall incorporate methods to discover bias and discrimination against both individuals and groups of objects, ideas, or people. Such methods may include statistical and/or probabilistic methods and, more generally, types of data mining. Data mining methods include anomaly detection (i.e. identifying objects that are outliers or significantly different from other elements in their class), association rule learning (i.e. discovering relationships between variables), clustering (i.e. discovering similar groups and structures in a data set that are more similar to each other than to others in the data set). Data mining should use both protected attributes (e.g. those affecting privacy, gender, religion, etc.) and combinations of attributes that indirectly discriminate in some way. For example, the association rule:

IF city == X and neighborhood == Y THEN perform Z (1)

appears fine. However, if it is combined with another rule that contains protected attributes, such as:

IF city == X and neighborhood == Y THEN protected attribute is TRUE (2)

the conjunction of (1) and (2) is:

IF city == X and neighborhood == Y and protected attribute is TRUE THEN perform Z (3)

which is clearly biased.

Many types of machine learning algorithms, such as neural networks, are too complex to understand how data is computed in each stage of each iteration. This is because the nested non-linear structure of such systems makes it very difficult to determine what information in the input data makes the system actually arrive at its decisions.

An ENI System shall be able to demonstrate that it uses a set of repeatable mechanisms as part of the machine learning process to help mitigate bias. Exemplary techniques include:

- pre-processing, which is used to change the learning procedure of an ML model;
- post-processing, which treats the ML model as a "black box" and does not modify the learning algorithm or training data;
- processing during learning (e.g. adding a discrimination-aware regularization term to the learning objective).

6.2.6.3 Adherence to Applicable Standards to Mitigate Bias

An ENI System shall be able to demonstrate adherence to appropriate standards for demonstrating that it can detect and mitigate bias. The IEEE P7003 [i.20] standard enables creators to communicate to users, and regulatory authorities, that best practices were used in the design, testing and evaluation of the algorithm(s) used in a product or system to attempt to avoid bias, and that its results are fair. The standard describes specific approaches that allow users of the standard to define how addressed and eliminated bias in the creation of their algorithmic system. For example, P7003 defines methods for checking if all customer groups are sufficiently represented in the testing data.

As another example, the GDPR [i.16] describes a "right to explanation". When combined with the "data protection by design" principles, the GDPR implies that data auditing methodologies designed to safeguard against algorithmic bias throughout the entire product life cycle will likely become the new standard for promoting compliance in automated systems.

6.2.7 Ethics

6.2.7.1 Introduction

The definition of ethics, as used in ENI, is "*a normative philosophical discipline of how a person or object should act towards others. It comprises three dimensions:*

1) *Consequentialist Ethics: an agent is ethical if and only if it considers the consequences of each decision and chooses the decision that has the most moral outcome.*
2) *Deontological Ethics: an agent is ethical if and only if it respects obligations, duties, and rights appropriate for a given situation.*

3) *Virtue Ethics: an agent is ethical if and only if it acts according to a set of moral values.*"

An introduction to ethics is provided in ETSI GR ENI 018 [i.37], clause 4.5.

6.2.7.2 Methods to Ensure Ethical Decision-Making

Two approaches to creating ethical decision-making are individual [i.23] and collaborative [i.24] decision frameworks. The issue of ethical dilemmas are present in both approaches. An ENI System shall analyse and provide feedback and/or explanation of ethical decisions.

Agents used by an ENI System should use one or more mechanisms to enable their ethical decisions to explain their actions. An ENI System should augment its learning and reasoning mechanisms with whitelist and blacklist rules. The former are rules that explicitly allow a set of identified entities to be granted a set of particular privileges, while the latter is the opposite (rules that explicitly disallow a set of identified entities to be granted a set of particular privileges).

An ENI System should use formal logic(s) to be able to mathematically prove hypotheses. In particular, it should define rules that:

- shall be implemented (i.e. obligation, permission, and delegation): deontic logic;
- may be proved wrong by contrary evidence: defeasible logic;
- may modify context by expressing modality: alethic logic.

An ENI System may use either an individual or a collaborative ethical decision-making framework.

Individual ethical decision-making frameworks used by an ENI System shall enable agents to make ethical decisions, either on their own or in consultation with other agents [i.22]. Similarly, an individual ethical decision-making framework used by an ENI System shall be able to determine if an ethical decision made by another agent is in fact ethical or not. Both of these properties shall use a formal cognition model for the agent (or set of agents), such as the belief-desire-intention model [i.28]. In addition, an ENI System may use a logically separate mechanism to reconcile ethics-based requirements with the proposed actions of an agent before those actions are implemented.

Collaborative ethical decision-making frameworks used by an ENI System shall form a network of trust to ensure that ethical decisions are free of uncertain and malicious behaviour. Collaborative ethical decision-making frameworks used by an ENI System may use a collection of specialized agents (e.g. some being deontological agents, and others being consequentialist agents). In this approach, a learning mechanism should be used to develop strategies to compute a collective decision by these agents. The learning mechanism may include a voting strategy, where different agents are assigned different weights based on the type of the ethical decision to be made.

6.2.7.3 Adherence to Applicable Standards and Initiatives

There are a number of international standards that may be applicable for an ENI System. These include:

- IEEE P70xx Series (i.e. P7001-P7014) [i.27].
- GDPR [i.16].
- The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems [i.25].

There are several specific works in the IEEE P7000 series that involve Ethics.

IEEE P7000 [i.46] is dedicated to identify, analyse, and reconcile ethical concerns. An ENI System shall perform these or similar efforts at the beginning of system and software life cycles. This affects both technologists and end-users.

IEEE P7001 [i.47] states that the operation of any intelligent system shall be transparent. An ENI System should make as many of its decisions to be transparent. This enables end-users to ask why the intelligent system made the decision it did, and helps trace internal processes that led to erroneous behaviour. More importantly, it helps users of the intelligent system trust the intelligent system, since it is more understandable.

IEEE P7002 [i.48] defines requirements for a privacy-oriented process for ensuring that personal data are protected. An ENI System shall make every effort to ensure private data are protected.

IEEE P7006 [i.49] defines standards for personal data AI agents, ensuring that machine-to-machine decisions will be made with appropriate transparency and explanations. An ENI System should apply this principle to all machine-to-machine decisions that directly affect the end-user. In addition, an AI agent should be able to represent the needs of an end-user (e.g. act as a proxy for the user in machine-to-machine decisions).

IEEE P7007 [i.50] standards formal communication for defining Ethical and Moral Theories. An ENI System should use ontologies to realize ethical and moral features, since ontologies use formal logics, and hence, can be mathematically proved. This is important for auditing and explanation. Similarly, IEEE P7008 [i.54] standardizes the notion of ethically-driven "nudging" (i.e. a recommendation or action that influences the behaviour of a user). An ENI System may use nudging to guide the end-user in ethical decision-making.

The GDPR [i.16] implies that AI systems should enable equitable societies by supporting human agency and fundamental rights, and not decrease, limit or misguide human autonomy. A preferred way of doing this is to augment pure learning and reasoning AI systems with different types of formal logics, including deontic, defeasible, and alethic logic. This enables ethical decisions made by an ENI System to be explained mathematically:

- The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems has defined Ethically Aligned Design [i.29]. This is a large document that defines a number of goals that an ENI System should implement. This includes ensuring that an intelligent system should not infringe on internationally recognized human rights.
- An intelligent system should prioritize metrics of well-being in the design and use of AI-based systems.
- An intelligent system should use appropriate practices, such as anonymization, to protect the discovery and use of privacy attributes.
- An intelligent system shall provide informed consent for the user of personal privacy data.
- The designers, developers, and operators of an intelligent system should be responsible and accountable for decisions made by the intelligent system.
- An intelligent system should operate in a transparent manner.
- An intelligent system should provide safeguards against its misuse.
- An intelligent systems shall be subject to applicable law.
- An intelligent system shall respect the rules of applicable government, industry, and other stakeholders in which it operates. This includes obeying rules governing decisions that shall never be allowed to be performed by an intelligent system.
- An intelligent system should similarly obey rules governing decisions that shall only be allowed to be performed by an intelligent system in coordination with humans.
- All applicable parties shall have access to all data and information generated and used by an intelligent system.
- The logic and rules embedded in an intelligent system should be available for examination and subject to risk assessments and testing.
- An intelligent system should generate audit trails recoding the facts and law supporting its ethical decisions.
- An intelligent system should enable third-party verification of ethical decisions that it has made.

6.2.8 The Assisted System

6.2.8.1 Overview

An important design principle of an ENI System is that changes need not be required to an Assisted System in order for an Assisted System to make use of recommendations or commands sent to it by an ENI System (see requirement [NFA1] in clause 5.4). This is true for all three classes of Assisted Systems (see clause 4.4.2).

In general, if an Assisted System of any class is able to expose an API, then it may be able to take better advantage of the capabilities of an ENI System. This is because this defines a direct mapping between the functionality of an Assisted System and that of an ENI System. The API translation shall be done by an API Broker (see clause 6.4).

6.2.8.2 Class 1 and 2 Assisted Systems

Class 1 and 2 Assisted Systems are represented as individual systems. Interaction between them and an ENI System may be done using APIs and/or the Designated Entity of an Assisted System.

If it is desired for an ENI System to interact with a set of class 1 or 2 Assisted Systems, then the mechanisms of Class 3 Assisted Systems, Option 1 (see clause 6.2.8.3.1) or Option 2 (see clause 6.2.8.3.2) shall be used, depending on the type of communication that is required. More specifically, if each Assisted System has the same set of responsibilities and capabilities, then Option 1 should be used; otherwise, Option 2 should be used.

6.2.8.3 Class 3 Assisted Systems

6.2.8.3.1 Single Class 3 Assisted Systems

This type of Assisted System, called option 1, was described in clause 4.4.2.5.2.2. In this interaction, an ENI System is aware that it is providing recommendations and commands to a single Assisted System, but is not aware of the number or type of control loops that the Assisted System has. There are two types of control loop arrangements, as shown in Figures 6-3 and 6-4.

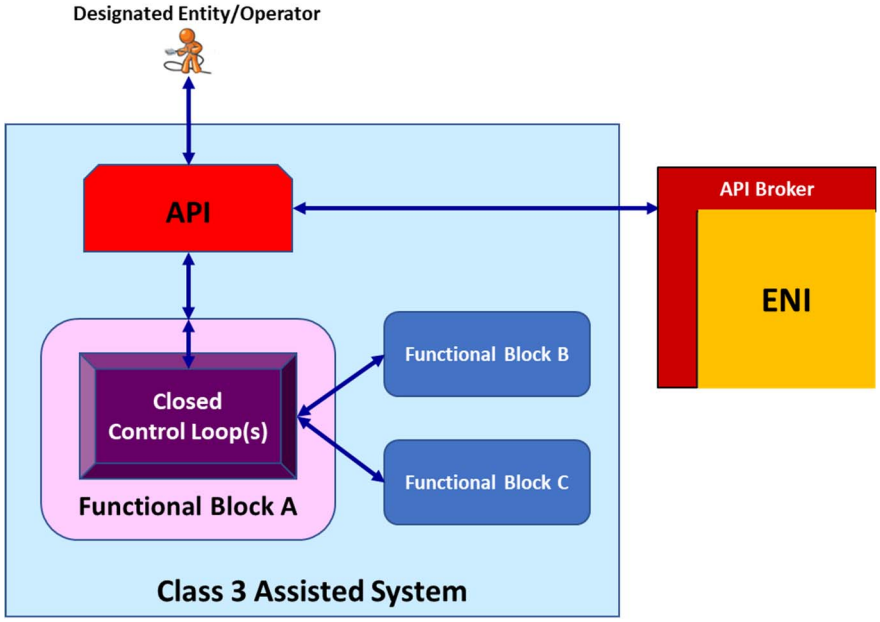

Figure 6-3: API of Single Class 3 Assisted System Directly Affect Closed Control Loop

In Figure 6-3, the API is sent directly to a Functional Block that contains a set of closed control loops. This type of closed control loop enables decisions made by the control loop(s) to be sent to other Functional Blocks, thereby affecting their behaviour (regardless of what type of task each Functional Block performs). The interactions between the API module and Functional Block A, as well as between Functional Block A and the other Functional Blocks in this figure, are not visible to the ENI System.

In contrast, Figure 6-4 shows that the API command(s) that affect the set of closed control loops in Functional Block A are delivered indirectly. This option enables any intervening Functional Blocks (B and C in this example) to pre-process the API command(s) for delivery to Functional Block A. Again, these interactions are not visible to the ENI System.

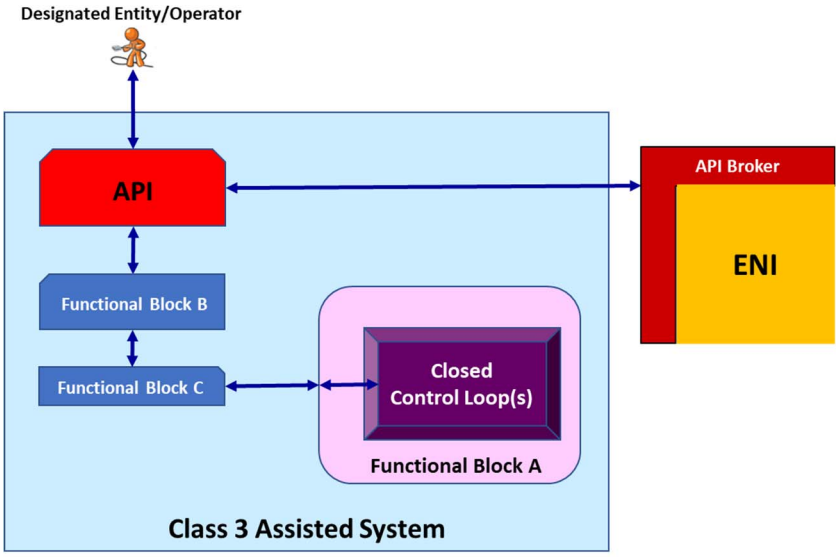

Figure 6-4: API of Single Class 3 Assisted System Indirectly Affect Closed Control Loop

The first option, shown in Figure 6-3, uses the API to direct the operation of the closed control loop(s), which in turn direct the operation of other Functional Blocks. In contrast, the second option (shown in Figure 6-4) uses the API, in combination with other Functional Blocks, to direct the operation of the closed control loop(s).

In both cases, care should be taken to ensure that the effects of using an API do not adversely effect the operation of the Functional Blocks being affected. For example, this may not be appropriate for some real-time decisions due to the communication and security constraints of using an API. This is because a delay in communication caused by using the API (versus, for example, remote method invocation or other types of communication), as well as additional time required to secure the communication between two entities that are not co-located, is likely not going to be acceptable in certain scenarios.

6.2.8.3.2 Multiple Class 3 Assisted Systems

Figure 4-4 (in clause 4.4.2.5.1) describes the possibility of an ENI System managing multiple Assisted System. There are two distinct cases:

- ENI is communicating with multiple Assisted Systems whose capabilities and responsibilities are identical.
- ENI is communicating with multiple Assisted Systems whose capabilities and responsibilities are not identical.

The first case applies to collaborative scenarios, such as peer-to-peer architectures, in which each Assisted System has the same capabilities and responsibilities. This is shown in Figure 6-5.

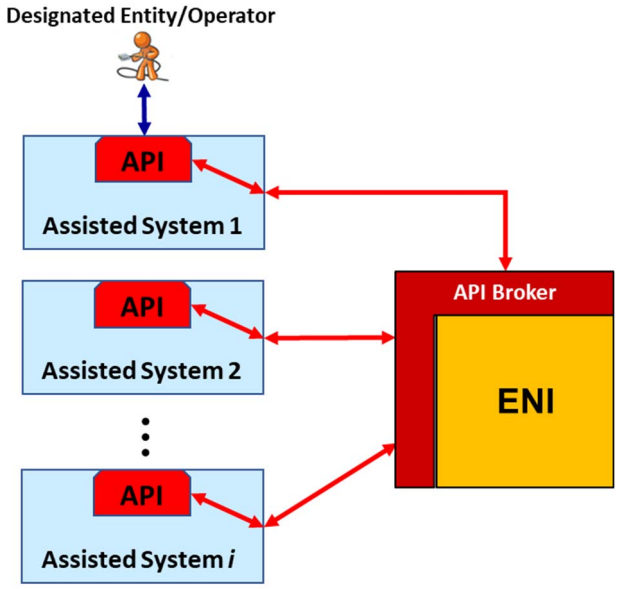

Figure 6-5: Multiple Collaborative Assisted Systems Controlled by an ENI System

In arrangements such as that shown in Figure 6-5, the ENI System need not know that it is communicating with a specific Assisted System, since the capabilities and responsibilities of each Assisted System are identical. Communication mechanisms may include peer-to-peer, individually sent mechanisms (e.g. broadcast), or using a bus.

The interactions between the API module in each Functional Block of each Assisted System in this figure are not visible to the ENI System.

The second case applies to scenarios in which one or more Assisted Systems have different capabilities and responsibilities. This is shown in Figure 6-6.

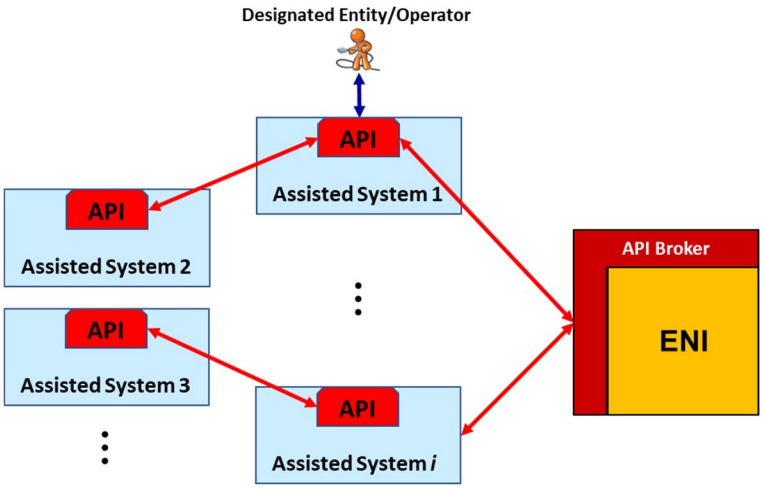

Figure 6-6: Multiple Disparate Assisted Systems Controlled by an ENI System

In Figure 6-6, some Assisted Systems are directly connected to an ENI System, while other Assisted Systems are only connected to other Assisted Systems. However, an ENI System shall know the identity of each non-collaborating Assisted System that the ENI System is directly connected to, if that Assisted System wants to receive recommendations and/or commands.

An ENI System shall be aware of the capabilities and responsibilities of each non-collaborating Assisted System that it is directly connected to. For example, in a floating master architecture, at least one of the Assisted Systems shall be designated as a master, and is responsible for coordinating and synchronizing data and commands to the other Assisted Systems (known as members). If a member Assisted System is promoted to function as a master Assisted System, then the ENI System shall be notified of this change. The ENI System shall interact directly with each master Assisted System in such an arrangement.

An ENI System should be aware of the number of non-collaborating Assisted Systems that it is sending recommendations and commands to. This helps the ENI System to keep track of the state of each non-collaborating Assisted System that it is providing recommendations and commands to.

It may be possible for an ENI System to *indirectly* control Assisted Systems that it is not connected to (e.g. 2 and 3 in Figure 6-6) through the set of Assisted Systems that it is directly connected to. However, the ENI System need not be aware of such behaviour.

The interactions between the API module and each Functional Block of each Assisted System in this figure are not visible to the ENI System.

6.3 Architectural Functional Blocks of the ENI System

6.3.1 ENI Functional Architecture with Reference Points

6.3.1.1 Introduction

The following clauses show the External and Internal Reference Points of the ENI Functional Architecture, along with examples of how Administrative and Management Domains, as well as Control Loops, may be realized.

6.3.1.2 ENI Functional Architecture with External Reference Points

Figure 6-7 shows a more detailed Functional Block Diagram that contains all of its input External Reference Points (see clause 7.3).

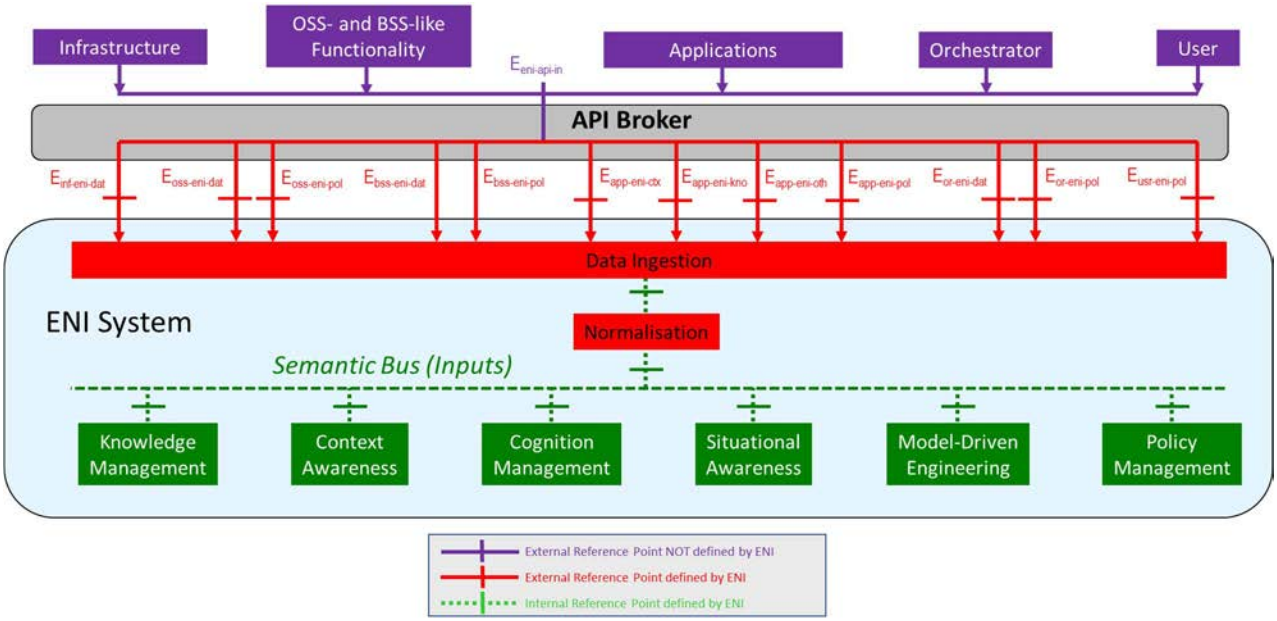

Figure 6-7: Functional Architecture with its Input Reference Points

Similarly, Figure 6-8 shows a more detailed Functional Block Diagram that contains all of its output External Reference Points (see clause 7.3).

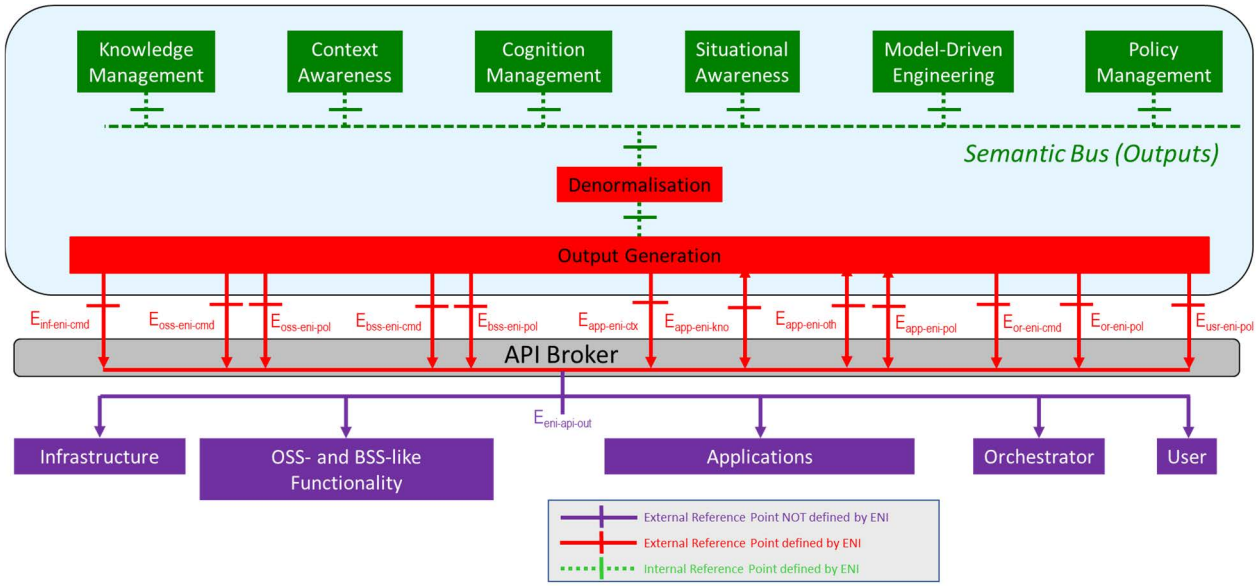

Figure 6-8: Functional Architecture with its Output Reference Points

6.3.1.3 ENI Functional Architecture with Internal Reference Points

Figure 6-9 shows a detailed Functional Block Diagram that contains all of the ENI Internal Reference Points (see clause 7.7).

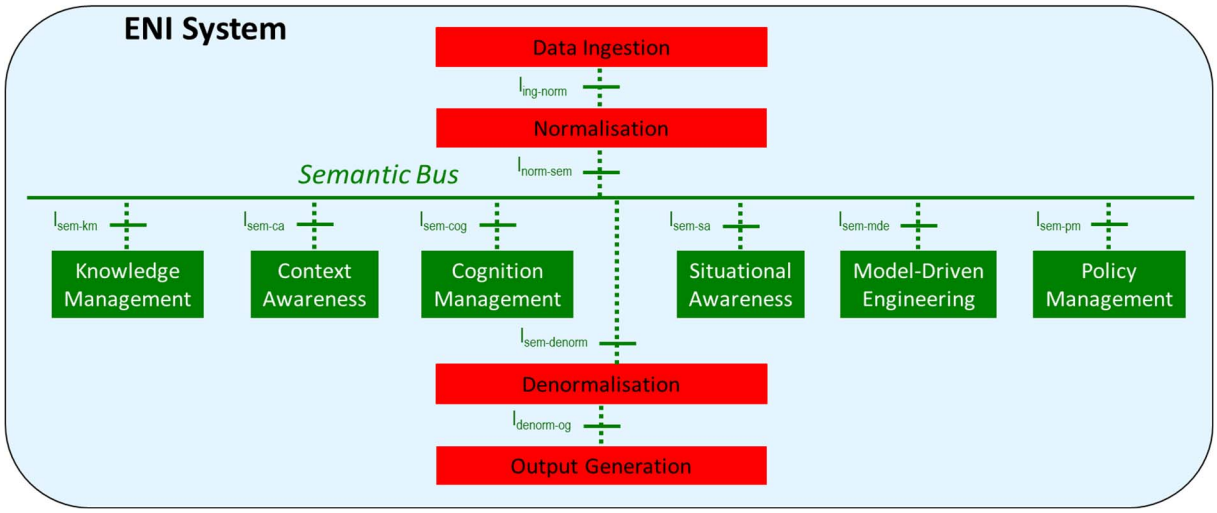

Figure 6-9: Functional Architecture with its Internal Reference

6.3.1.4 ENI Functional Architecture with Administrative and Management Domains

Figures 6-10a and 6-10b show a detailed Functional Block Diagram of the ENI Functional Architecture with Administrative Domains and Policy Management with exemplary external and internal domains, respectively. The External and Internal Reference Points are the same as in the previous figures, but are now shown within the context of an Administrative Domain.

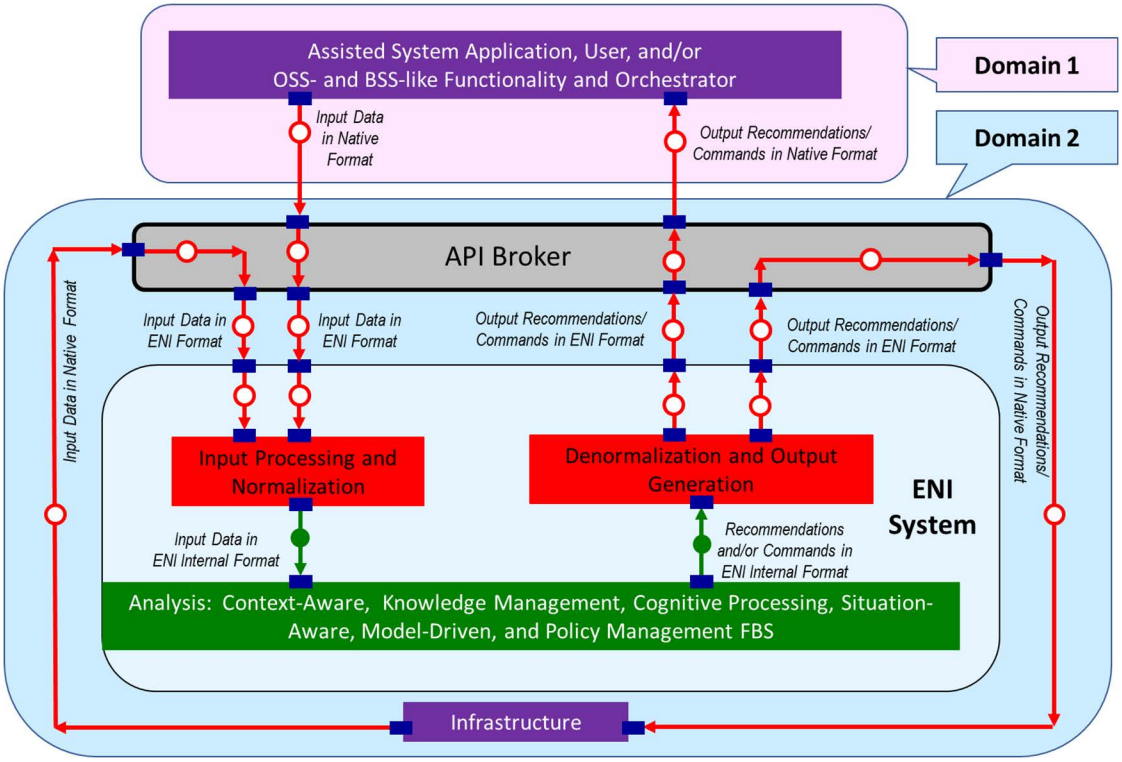

Figure 6-10a: Functional Architecture with An External Exemplary Domain

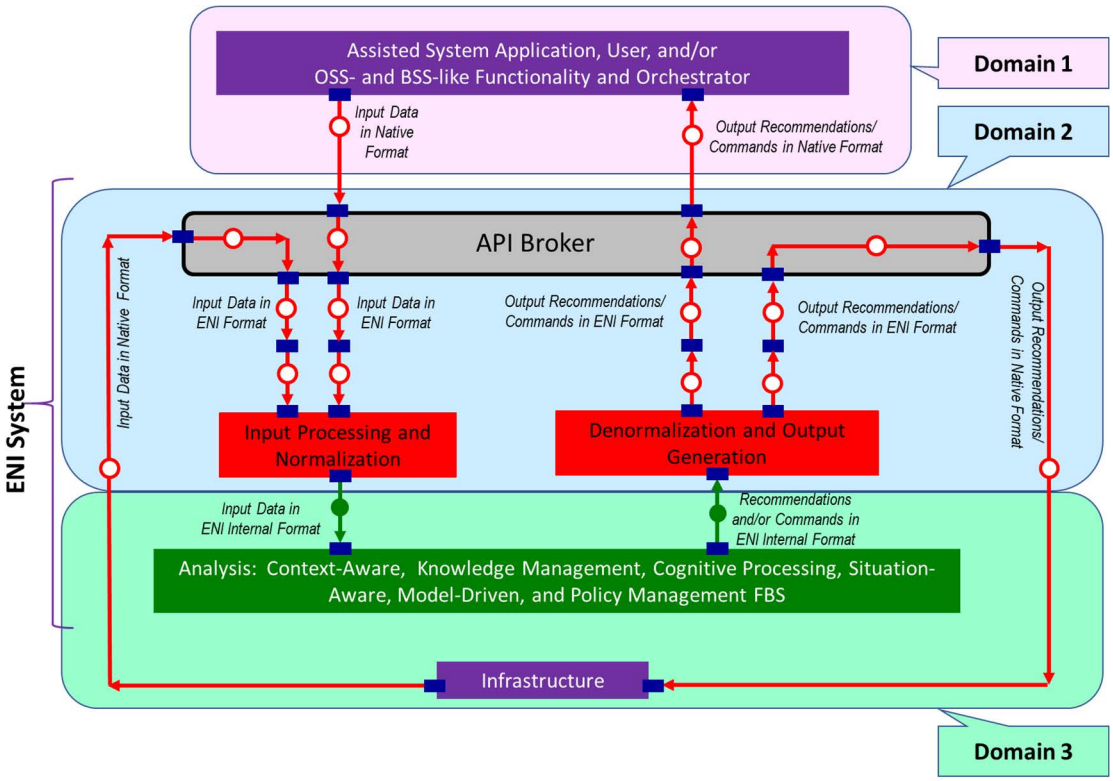

Figure 6-10b: Functional Architecture with Exemplary Internal Domains

6.3.1.5 ENI Functional Architecture with Control Loops

Figure 6-11 shows an exemplary mapping of a simple closed control loop (as described in ETSI GR ENI 017 [i.36].

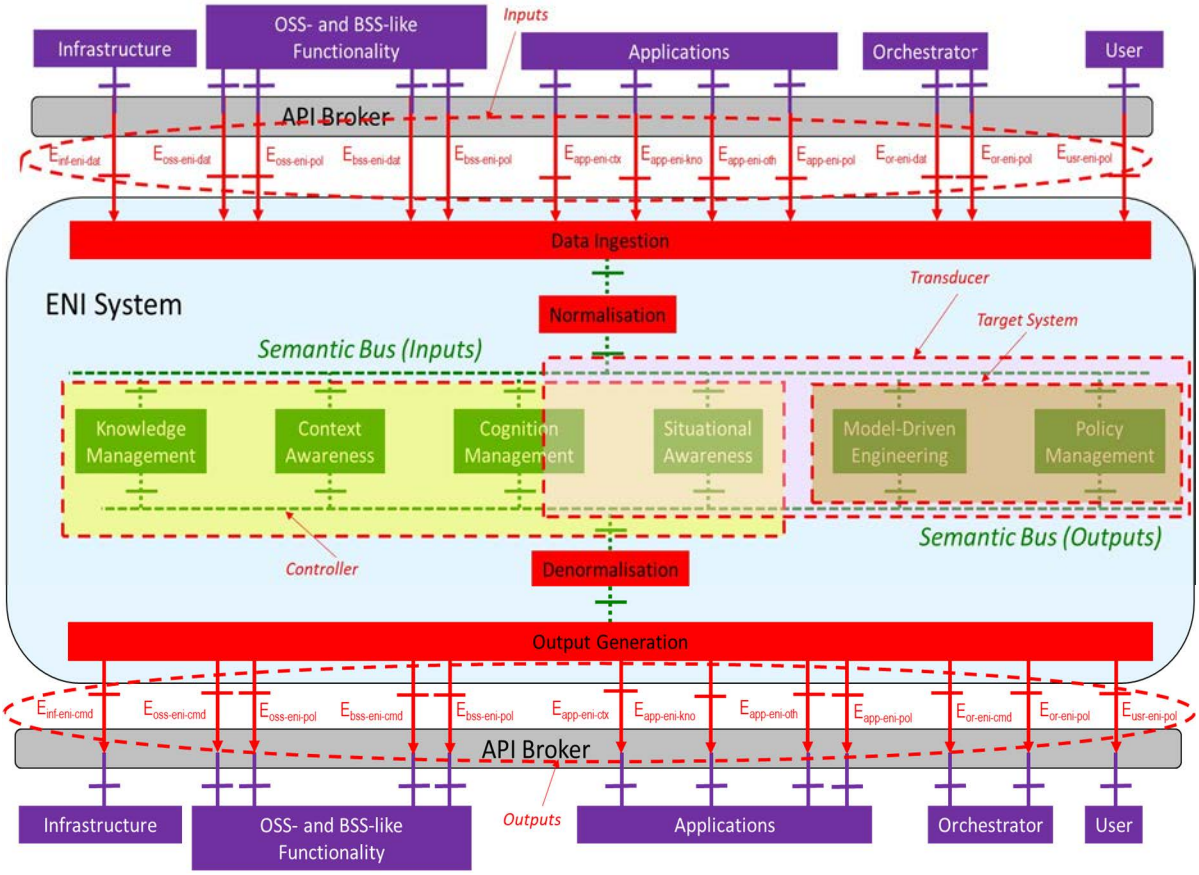

Figure 6-11: Functional Architecture with Exemplary Control Loops

Figure 6-11 specifies that all external entities that have the ability to send Policy Rules to, and receive Policy Rules from, the ENI System use dedicated External Reference Points to do so.

The following exemplary description is done in a somewhat serial flow for ease of understanding. This does not mean that the actual processing done is serial. One reason that a bus is used is to enable complex communication to occur between different internal Functional Blocks.

A closed control loop, as described in ETSI GR ENI 017 [i.36], consists of the following functions:

- A desired input, or goal.
- A function that compares the desired input and the feedback (transduced adjustment in [i.36], Figure 4.3.2-1).
- A controller.
- A target system.
- A transducer.

In Figure 6-11, the above five functions are represented by one or more ENI Functional Blocks (or signals from them). More specifically:

- The goal (which every closed control loop has) is represented in its final form in the Model-Driven Engineering Functional Block.
- The function that compares the desired input and feedback is contained in the Controller in this example.

- The Controller is represented by the four Functional Blocks (Knowledge Management, Context-Aware Management, Cognition Management, and Situational Awareness).
- The Target System is represented by the Model-Driven Engineering and Policy-Based Management Functional Blocks.
- The Transducer is represented by the Situational Awareness, Model-Driven Engineering, and Policy-Based Management Functional Blocks.

The following explains the operation of a generic closed control loop using Figure 6-11.

The closed control loop is first supplied a goal to achieve. This goal shall be supplied by ENI, though it may have originated from the Assisted System. If the former, then that goal is stored in the Knowledge Management Functional Block for use by other Internal Functional Blocks. If the latter, then the goal appears as input data, and hence, is ingested and normalized, followed by being stored in the Knowledge Management Functional Block. In either case, the goal is represented in the Model-Driven Engineering Functional Block.

Input data to the closed control loop may come from one or more of the five main sources (Infrastructure, OSS- and BSS-like functionality, Applications, Orchestrator, and User). All input is transmitted to the API Broker, which then communicates the data using one (or more) of the designated ten input External Reference Points. Each input first goes to the Data Ingestion and then to the Normalization Functional Blocks. At this point, the data is in a format that can be understood by the six Internal ENI Functional Blocks. The six Internal Functional Blocks are notified that new data has been ingested and normalized via a message delivered by the Semantic Bus [i.30].

The function of the Controller is played by the Knowledge Management, Context-Aware Management, Cognition Management, and Situational Awareness Functional Blocks. Briefly, the Knowledge Management Functional Block is responsible for representing data, information, and knowledge about the target system. Once the ingested data is represented, all interested consumers are notified of this via the Semantic Bus.

The Context-Aware Management Functional Block enables an ENI System to adapt its behaviour according to changes in context. For example, the ENI Internal Functional Blocks may collectively decide that a more efficient way to achieve a goal is to use different operations and processes given new contextual information. Current and historical context may be used, since both provide different information when doing statistical analysis and inferencing. Once context data has been integrated with the ingested data, all interested consumers are notified of this via the Semantic Bus.

The Cognition Management Functional Block is responsible for ensuring that an ENI System is achieving its end-to-end goals as efficiently as possible. More specifically, this includes understanding how ingested data and information interact with and are affected by the current (and possibly historical) context(s). This enables the Cognition Management Functional Block to change existing knowledge and/or add new knowledge. Outputs from this Functional Block are then made available to all interested consumers via the Semantic Bus.

The Situation Awareness Functional Block enables an ENI System to understand what has just happened, what is likely to happen, and how both may affect the goals that an ENI System is trying to achieve. Hence, the fundamental part of any closed control loop that makes decisions will use the facilities of this Functional Block. Outputs from this Functional Block are then made available to all interested consumers via the Semantic Bus.

As the Controller completes its processing, the Knowledge Management Functional Block stores updated knowledge representations that all Internal Functional Blocks may use. This includes, but is not limited to, adjustments to information and data models, adjustments to ML and statistical models, and generation of other relevant information (e.g. heuristics).

The Model-Driven Engineering Functional Block is responsible for taking the decision(s) from the Controller and translating it to recommendations and/or commands. This process is made scalable by converting recommendations and commands to a common structural form that is contained in one or more Policy Rules. Any combination of intent, imperative, and declarative Policy Rules may be used to properly represent the recommendations and/or commands. This output is fed directly to the Policy Management Functional Block, but parts of the output may be made available to all interested consumers via the Semantic Bus.

The Policy Management Functional Block is responsible for delivering the output of the closed control loop, which is a set of Policy Rules. Policy Rules may contain recommendations or commands, and multiple Policy Rules may be communicated to the Assisted System.

Recommendations and/or Commands are then sent to the Denormalization and Output Generation Functional Blocks, where they are sent to the Assisted System via the appropriate set of External Reference Points.

6.3.1.6 ENI Functional Architecture with Domains and Control Loops

Figure 6-12 shows an exemplary mapping of ENI with internal domains that are part of a closed control loop.

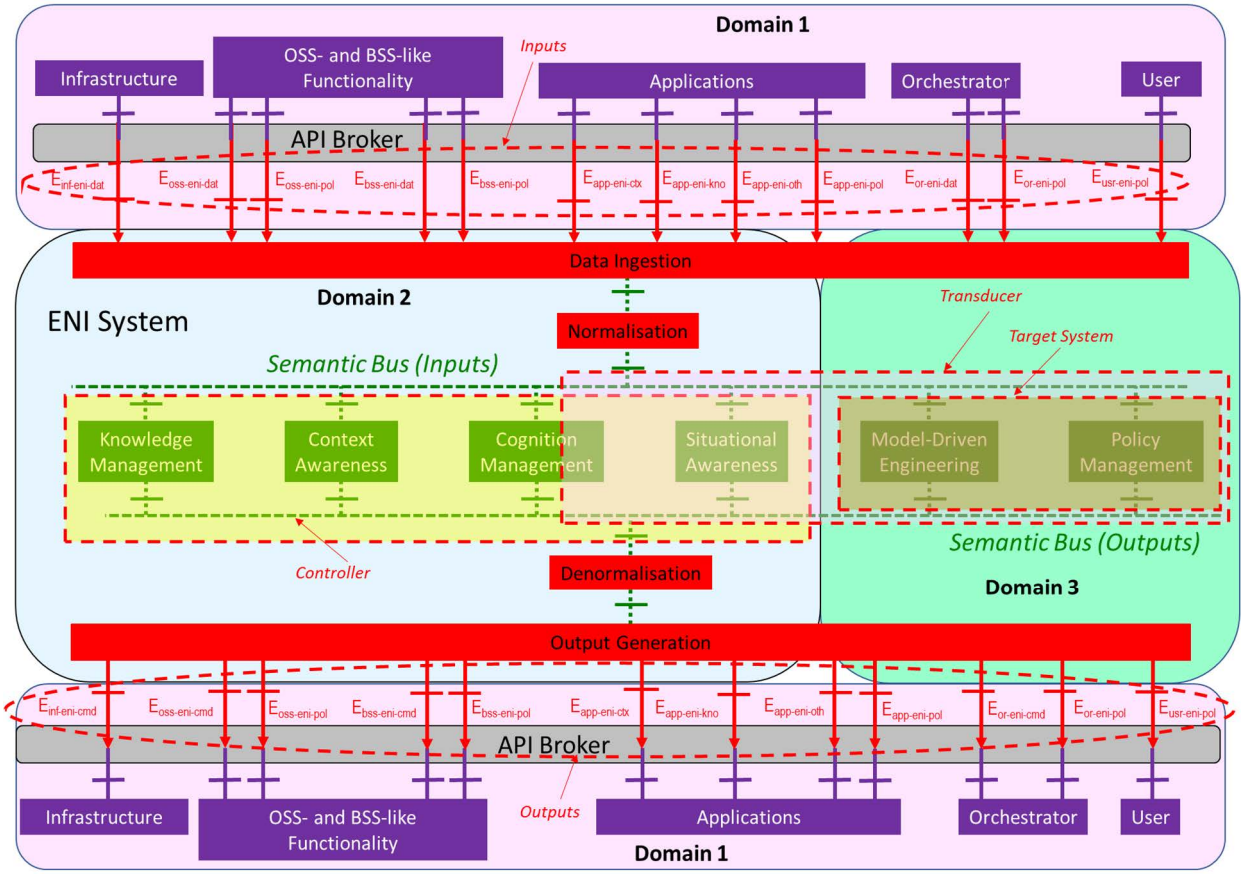

Figure 6-12: Functional Architecture with Exemplary Domains and Control Loops

Figure 6-12 is a combination of Figure 6-10 and Figure 6-11. Inputs and Outputs to ENI are in Domain 1. Inputs are accepted by the API Broker and sent using the appropriate ENI External Reference Point to the Data Ingestion Functional Block. Similarly, outputs from ENI are send from the Output Generation Functional Block to the API Broker, where they are translated into an appropriate form that the consuming external entity can use.

Domains 2 and 3 make up the closed control loop. ENI may implement a closed control loop in one or more management domains. In this example, the parts of the closed control loop that observe, orient, analyse, and plan ENI actions is in Domain 2, while the MDE and policy generation portions are in Domain 3.

Details of control loops are found in ETSI GR ENI 017 [i.36].

6.3.2 Data Ingestion Functional Block

6.3.2.1 Introduction

This clause describes the processes associated with ingesting different data from various data sources. Once this is complete, data is then sent to the Normalization Functional Block, which translates the data into a common format for further processing by the other Functional Blocks of the ENI System.

A network has different domains (e.g. RAN/Fixed Access, Transport, and Core). Each domain has its specific functions and services, as well as specific APIs. In a case where the ENI System helps with a localized network function in a specific domain (e.g. optimizing resource allocation at the RAN/Fixed Access), the ENI System may interact with the interfaces of the Assisted Systems of that domain, and may collect data from that domain only. In a more likely case, where the ENI System helps with a cross-domain function (e.g. end-to-end network service assurance), the ENI System may interact with multiple domains of the network. In either case, this shall be done using External Reference Points, and may use the API Broker to insulate an ENI System from having to change its functionality to accommodate the characteristics and behaviour of different APIs from different Assisted Systems.

These characteristics give rise to the following requirements:

- ENI shall provide the ability to ingest structured, semi-structured, and unstructured data from different data sources. This may be efficiently implemented using a multi-agent architecture.
- ENI shall also provide the ability to ingest data in streaming and batch mode, as well as on-demand. In addition, ENI shall provide the ability to contextually change the sources of data to be ingested. In all cases, the data collected shall be normalized to a uniform data format.
- Data Ingestion may be realized as a Functional Block that is separate from the Data Normalization Functional Block. This adheres to the Single Responsibility Principle [i.9], and enables a more scalable and robust system to be designed and built.

6.3.2.2 Motivation

Each domain has its specific functions, services, APIs, and may run on its specific time cycle. Thus, the data ingestion Functional Block shall be able to operate on different types of domain-specific data. Similarly, the processing operations applied to the ingested data are also a function of what Functional Blocks will use the data created from this Functional Block, as well as the nature of functions performed in that domain. The normalization of the time granularity may also be needed, where up/down sampling and interpolation may be applied.

The ENI System shall collect data based on the tasks it needs to perform (e.g. configuration vs. monitoring changes) as well as the nature of the analysis being done (e.g. as congestion is being controlled, different points in the network and different types of data may need to be monitored). The tasks are defined by either the Assisted System (or its Designated Entity) or an ENI System, typically in response to externally defined goals and policies. In the case when the tasks are defined by the Assisted System (or its Designated Entity), the Assisted System (or its Designated Entity) shall send a request of performing such an action to the ENI System. In the case when the tasks are defined by an ENI System, the ENI System shall send appropriate requests to the Assisted System (or its Designated Entity). Functional Blocks of ENI (e.g. policy management, or cognition management, or situation awareness management) interpret the requirements of each task and are then responsible for defining the types of data, when, and how they are collected.

6.3.2.3 Function of the Data Ingestion Functional Block

6.3.2.3.1 Introduction

If the Data Normalization Functional Block is not present, then the Data Ingestion Functional Block shall perform all required actions that would have been performed by the Data Normalization Functional Block. Otherwise, the following clauses define the actions that may be performed in the Data Normalization Functional Block.

The processing may include learning and inferring from the available raw data of one or more domains; once these data are analysed, the processing shall then decide on what knowledge is forwarded to other Functional Blocks of ENI. The processing may also benefit from Model-Driven Engineering (MDE) mechanisms, since modelled data define how the data should appear and behave in an error-free state. This is discussed more in clause 6.3.8.

In certain cases, the processing may save the raw form of the ingested data for further use. For example, many types of trend processing require access to raw data. In most cases, the processing function may save the processed form of the data; this is both faster and more efficient. The choice of whether to save the raw or processed form of the ingested data is dependent on the current context (see clause 6.3.5) and/or the current and anticipated situations (see clause 6.3.7). The Situation Awareness Functional Block may also define a set of Policy Rules that take action based on, for example, the type of data or the intended Functional Block that will use the ingested data. The Cognition Management Functional Block (see clause 6.3.6) may aid in the understanding of ingested data.

The processing may include aggregation and correlation functions (e.g. to reduce dimensionality) as well as machine learning (e.g. this may yield faster results by dealing with significantly smaller data sets, and enable what-if analysis and other game-theoretic algorithms to be used). In such a case, the resulting normalized data may also contain knowledge of a specific domain, or multiple domains.

The particular set of processing functions required is dependent on the current context (see clause 6.3.5) and/or the current and anticipated situations (see clause 6.3.7). The Situation Awareness Functional Block may also define a set of Policy Rules that take action based on, for example, the type of data or the intended Functional Block that will use the processed data. The Cognition Management Functional Block (see clause 6.3.6) may augment the meaning of ingested data (e.g. by adding metadata).

6.3.2.3.2 Data Filtering

Data filtering is the removal of unnecessary or unwanted information. This is done to simplify and possibly increase the speed of the analysis being performed. This is similar to removing the "noise" in a signal. Filtering is generally (but not always) temporary - the complete data set is kept, but only part of it is used for the calculation.

Filtering requires the specification of rules and/or business logic to identify the data that shall be included in the analysis. Examples include outlier removal, time-series filtering, aggregation (e.g. constructing one data stream from pieces of other data streams, such as merging name, IP address, and application data), validation (i.e. data is rejected because it does not meet value restrictions), and deduplication.

6.3.2.3.3 Data Correlation

Data correlation refers to an association or relationship between data. It expresses one set of data in terms of its relationship with other sets of data. For example, the number of upsells to a higher class of service may increase due to targeted advertising, and may increase even more when offering free time-limited trials. Another example is collecting the complete set of data related to a subscriber. These data are usually collected using different mechanisms, and hence, are fragmented among different collection points. Data correlation can use rules and/or business logic to collect the scattered data and combine it to improve analysis. Data correlation is the first step in gaining increased understanding of relationships between data and their underlying objects.

6.3.2.3.4 Data Cleansing

Data Cleansing is a set of processes that detect and then correct or remove corrupt, incomplete, inaccurate, and/or irrelevant data. Data cleansing typically is performed on batches of data. It differs from data validation in that data validation is performed at the time of ingestion, whereas data cleansing is performed later.

Data cleansing solutions may also enhance the data, either by making it more complete by adding related information or by adding metadata. Finally, data cleansing may also involve harmonization and standardization of data. For example, abbreviations may be replaced by what they stand for, and data such as phone numbers may be reformatted to a standard format.

6.3.2.3.5 Data Anonymization and Pseudonymization

Data Anonymization is the process of either removing or encrypting information that can be used to identify people from a data set. For the purposes of the present document, the Anonymization process is defined as irreversibly severing data that can be used to identify a person from the data set. Any future re-identification is no longer possible.

Data Pseudonymization is the process of replacing information that can be used to identify a person with one or more artificial identifiers (i.e. pseudonyms). For the purposes of the present document, the Pseudonymization process is reversible by certain trusted entities, since the identifying data was not removed, but rather substituted with other data.

6.3.2.3.6 Data Augmentation

For the purposes of the present document, data augmentation is the process of adding other types of data to the existing data set to enrich it in some way. Two examples are the addition of metadata and the addition of ontological data to a data set to increase the understanding of the data. For example, metadata consists of additional information that describe or prescribe the characteristics, behaviour, and operation of the data. Ontological data is logic-based data that has one or more relationships to the original data that help explain those data. For example, ontological data could provide linguistically related information to provide additional information about the data.

6.3.2.3.7 Data Labelling and Annotation

6.3.2.3.7.1 Introduction

Data labelling is the process of adding corresponding class labels to data based on real information provided by the Assisted System (or its Designated Entity). The class labels represent the state or attribute of the data. When used in machine learning, the labels show the answer that the machine learning model should predict. For example, for a fault detection case, labels could identify normal or abnormal traffic types in a traffic identification case. The labelled data is used in model training with supervised ML algorithms.

Labelling is typically done by adding tags and/or metadata to the ingested data. Data labelling is more properly called data *annotation* when metadata is used, as the metadata is more descriptive and/or prescriptive than a simple label.

6.3.2.3.7.2 Types of Data Annotations

The following is a non-exhaustive list of the most common types of data annotations used.

Text Categorization. This type of annotation is used to assign text to predefined categories. For example, different parts of a document could be assigned different *topics* (see clause 6.5.3.5) for transmission using the Semantic Bus (e.g. a parent topic could be called congestion data, with two sub-topics, called domestic and international congestion data).

Entity Annotation. Named Entity Recognition (NER) (see clause 6.3.5.4) is the most common form of entity annotation. NER classifies text according to predefined entities in an information or data model, and/or in an ontology. Examples include finding the names of people, organizations, places, and equipment, and adds facts and meaning to each entity occurrence. This then allows Named Entity *Linking*, where two different named entities that occur in different parts of a document are associated with each other. For example, a Service can be associated with a Provider, or a type of equipment associated with an Internet Gateway.

Image Annotation. This uses one or more processes to target an area of interest in the image. For example, a box can be placed on an image to identify an object. Another example is assigning each pixel a value corresponding to its meaning and/or importance in recognizing the image. This helps a machine learning model recognize the annotated area as a distinct type of object, and can be used for autonomous vehicle guidance, facial recognition, and even to block sensitive or inappropriate content.

Video Annotation. Similar to image annotation, this adds bounding boxes, polygons, or other identifying tags to video content. This can be done using a video annotation tool, or more typically, on a frame-by-frame basis, where frames are subsequently stitched together to help track the movement of the annotated object. This can be used for developing computer vision models for object tracking and similar tasks.

Linguistic Annotation. This type of annotation is used in each of the parsers (or compilers) present in the ENI System (e.g. in the Situational Awareness, MDE, and Policy Management Functional Blocks). It tags the words in a piece of content with their grammatical meaning (e.g. verb, verb phrase, and proper noun).

Semantic Annotation. Semantic annotation adds meaning to annotated data. Machine learning models use semantically annotated data to learn how to categorize new concepts, as well as to infer other relationships to better categorize and annotate other parts of the text, image, or video. ENI shall use this type of annotation to form hypotheses about entities found as well as their behaviour.

6.3.2.3.7.3 Labelling Accuracy and Labelling Quality

Data labelling accuracy measures how close the labelled features match real-world data, and is independent of the type of data and model that is being annotated. Data labelling quality measures the accuracy of the labels across the entire dataset. For example, it examines whether each label works the same and provides the same result everywhere that it occurs.

Annotation shall require context. This enables the annotation system to choose the correct meaning for a label having multiple meanings. For example, the word "bank" could mean a financial institution, to count on something being true, a part of a river, or an aerial maneuver. It also enables *type-of* relationships (e.g. red is a type of colour, where red is a hyponym and colour is a hypernym); this enables substitution of such relationships to be found.

Annotation should require situation awareness. This enables the relevance of the data that is being labelling to be defined based on the goals of the situation (see clause 6.3.7). For example, if telemetry says that 17 packets were dropped, then the action depends on what type of packets were dropped, along with what protocol, when, and where. Continuing the example, if the dropped packets were transmitted using FTP, that protocol will take care of recovering and transmitting them. In contrast, if the dropped packets were control or management packets, that represents a more serious problem, and the system needs to retransmit them (perhaps using a "guaranteed delivery" message channel as described in clause 6.5).

6.3.2.3.7.4 Semantic Annotation

One of the most reliable and efficient ways to implement semantic annotation is to use neural networks. This is because semantic annotation can be seen as a classification problem, where the correct meaning of the input is discovered using various machine learning mechanisms. The basic problem is that there may be a large number of features that contain information, but it is difficult for the classifier to properly combine them. In addition, a particular input may be associated with more than one label. This is why neural networks are preferred to other simpler classification models (e.g. random forest), because neural networks are better suited to handling large number of features and can recognize non-linear combinations.

There are many cases where obtaining labelled data is expensive, even for text classification. Active learning may be used to minimize the amount of labelled data that is required to build an accurate model for machine learning (see ETSI GR ENI 018 [i.37]). The representation used to represent the content of a document is crucial.

NOTE: Release 4 (see clause 9) will compare vectorized representations based on word frequencies (e.g. bag-of-words), word embeddings (e.g. words are mapped into a vector space where the distance between words in the space is related to the syntactic ad semantic features of the words; and example is *word2vec*), and document-level mechanisms (e.g. bi-directional encoder representations from transformers, or BERT) to determine which (if any) of these approaches could be used effectively in active learning.

6.3.2.4 Operation of the Data Ingestion Functional Block

6.3.2.4.1 Introduction

Different data sources use different languages and protocols to communicate their data. One way to accommodate this is to use a set of agents that each understands a particular data source and the data that it sends. This approach is shown in Figure 6-13. The functional block diagram shown in Figure 6-13 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of data ingestion. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

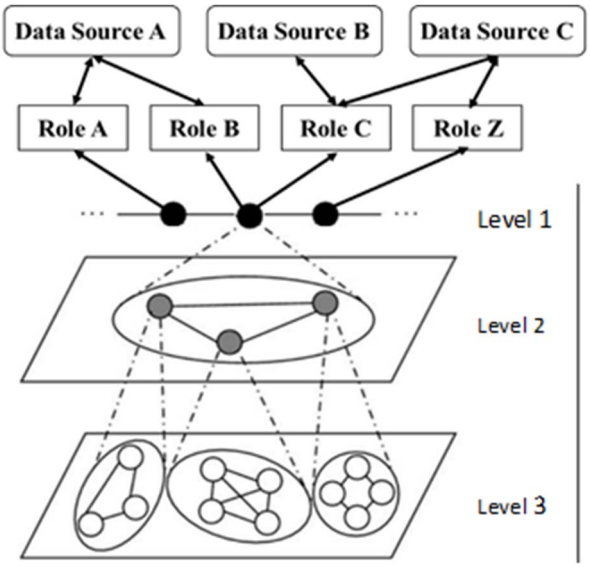

Figure 6-13: Data Ingestion Operation

Figure 6-13 shows a preferred method of operation of the Data Ingestion Functional Block. The bottom three levels represent agents. Level 1 is the agent having the highest level of abstraction, while level 2 is a set of agents that are either controlled by the corresponding level 1 agent or represent functionality required by the corresponding level 1 agent. Similarly, level 3 represent a set of agents that are either controlled by or represent functionality required by the corresponding level 2 agent. This is an example; there is no requirement to implement an agent hierarchy or a hierarchy that has a fixed number of levels for an ENI Agent System. Each agent may take on zero or more roles, and each role may be assigned a set of functions to a given data source. For example, Role A may be assigned the ability to perform any configuration operation required for monitoring, while Role B is assigned the ability to perform traffic engineering operations.

The advantages of this type of architecture include:

- Flexibility: the implementation may change (e.g. lower levels of agents may change) without affecting the API (built from more abstract levels).
- Extensibility: new levels of agents may be added or removed to suit the needs of the application.
- Manageability: instead of having to manage individual agents, the management focus is abstracted to managing roles.

6.3.2.4.2 Telemetry Processing

6.3.2.4.2.1 Cognitive and Situation-Aware Directed Telemetry Gathering

One of the advantages of using an adaptive closed control loop (see ETSI GR ENI 017 [i.36]) is to be able to easily change the goal(s) and/or the data being analysed by the control loop. In this case, the Cognition Management Functional Block of ENI (see clause 6.3.6) shall be responsible for relating the actions taken by the ENI System to the set of end-to-end goals that the ENI System is trying to satisfy. The Cognition Management Functional Block works closely with the Situational Awareness Functional Block to understand the meaning and significance of data and behaviour that is being analysed. It shall build plans that describe various mechanisms that can be performed to move the current state of the entity being monitored to its desired state.

Both may postulate theories as to how the system being monitored, and/or its environment, are behaving. Such theories may necessitate telemetry, as well as one or monitoring points, to change in order to prove these theories.

6.3.2.4.2.2 Detecting Anomalies

An anomaly may be defined as "the measurable consequences of an unexpected change in state of a datum, or set of data, which is outside of its local or global norm".

There are several approaches to detecting anomalies. Uni-variate anomaly detection approaches process the different telemetry parameters independently. In contrast, multi-variate anomaly detection approaches take into account possible correlations between the different parameters, allowing contextual anomalies to be detected:

NOTE: The following will be examined in Release 4 (see clause 9) of the present document.

- **Statistical.** This uses historical data to model the expected behaviour of a system.
- **Probabilistic.** Similar to statistical, but uses probabilities or fuzzy algebra to do the comparison.
- **Distance-based Metrics.** This uses the distance between a newly measured datum and previous data, and defines the datum as an anomaly if the distance is larger than a pre-defined value.
- **Pattern Matching.** This compares each new measurement against a database of known anomalies, and classifies measurements that are more similar to known anomalies than to correct data as anomalies.
- **Structural Matching.** This compares the structure of the data to known data fields from models and/or ontologies. If the data matches correct or anomalous data, then a corresponding decision is made.
- **Clustering.** This projects measured data into a multi-dimensional space. Measurements that do not belong to, or are too far from a cluster, are classified as anomalies.

- **Ensemble Matching.** This approach uses a number of different algorithms to analyse each measurement, and then defines a collective vote from each method.

- **Machine Learning.** Different types of machine learning algorithms can be used. For example, deep neural networks (e.g. LSTM, or Long Short-Term Memory, which are a type of recurrent neural network capable of learning order dependence in sequence prediction problems) could be compared against autoencoders (e.g. a type of neural network that can be used to learn a compressed representation of raw data, which learns a condensed representation of the input data) and transformers ((see ETSI GR ENI 018 [i.37]).

6.3.2.4.2.3 Storing Telemetry Information

Telemetry information shall be gathered as directed by the ENI System. More specifically:

- ENI Policies shall be used to define what telemetry information is desired, where it shall be collected, and the frequency of collection.

- ENI Policies may specify one or more collection methods in prioritized order.

- Raw telemetry information shall be ingested using the Data Ingestion and Normalization Functional Blocks through the appropriate ENI External Reference Point (see clause 7.3).

- Telemetry information that is deemed valuable for making future decisions should be stored in an appropriate repository (see clause 6.3.4.5). Stored telemetry information may be anonymized or pseudonymized as appropriate, and should be compressed.

6.3.2.4.2.4 Changing Telemetry Gathering using Policies

The ENI System should use an adaptive and cognitive control loop (see clause 6.3.4 and ETSI GR ENI 017 [i.36]). This enables its actions to reflect changes in user needs, business goals, and environmental conditions. As these elements change, the ENI System may need to gather new telemetry (e.g. to verify hypotheses made). All requests to gather telemetry should be made using ENI Policies.

6.3.2.4.3 Use of Metadata

The Data Ingestion Functional Block should use metadata to describe and prescribe processing that should be done on different types of ingested data. Metadata may be used in identifying the type of ingested data. Since metadata may contain descriptive as well as prescriptive information, metadata may also contain important information that defines which processing operations should be executed on the ingested data before it is sent to the next Functional Block. This method may be used with any of the methods in clauses 6.3.2.4.4, 6.3.2.4.5 and 6.3.2.4.6.

6.3.2.4.4 Use of Structure, Pattern, and Feature Matching

The same data may be organized and/or formatted in different ways. It may also be ingested where the individual elements of the data are out of order. Therefore, the Data Ingestion Functional Block should use the following set of related model-based approaches to classify and process ingested data that has not yet been identified.

Pattern matching matches groups of objects that occur together. For example, congestion information requires information on at least two endpoints and the link that they use to communicate.

Feature matching concentrates on a set of attributes that together constitute a distinguishing characteristic of an object.

Structural matching uses the structure of a set of elements as a baseline for comparing the structure of the ingested data. There are several variations. The simplest treats the datum as an object, and searches for an *equivalence* match to one or more objects in the ENI information or data models. Here, an equivalence match assigns a probability that the ingested datum is the same as that matched in the ENI model (e.g. 100 % if its attributes are the same, and 0 % if none of its attributes are the same). Object matching extends the above to sets of objects, where the typical definition of an equivalence match is that some minimum percentage of *each object* matches.

The above two variations can each be enhanced by considering the relationships of the object(s) representing the ingested data to the relationships in the ENI model. This provides additional confidence of a match, because it provides an indication that the *behaviour* of the ingested data is likely to be the same as the behaviour of the modelled object(s).

The Data Ingestion Functional Block should use one or more of these methods to identify ingested data before it is passed to subsequent ENI Functional Blocks. This method may be used with any of the methods in clauses 6.3.2.4.3, 6.3.2.4.5 and 6.3.2.4.6.

6.3.2.4.5 Use of AI-based Mechanisms

There are a number of different AI-based mechanisms that can be used to identify ingested data. This approach is typically used in situations where large amounts of real-time data are collected (e.g. optical networks or autonomous driving systems) or when fine-grained analysis is required to identify data. It may be thought of as an extension to structural, pattern, and feature matching, where all three are used to identify learned structures that can be matched. Therefore, the Data Ingestion Functional Block should use appropriate AI-based approaches to classify and process ingested data that has not yet been identified.

The advantage of AI-based mechanisms is that strong correlations and similarities between data are inherently included as part of the algorithm. This can extend to data ingested from different sources and regions. Data-fusion-assisted telemetry enables various data fusion algorithms to be used to analyse ingested data and find matches to known objects or to develop hypotheses as to what known objects are the most similar to ingested data based on their similarity.

For example, optical systems provide optical and digital information. Both of these domains require special systems for transmitting and interpreting data (e.g. fibre amplifiers and digital signal processing algorithms). Correlation between data transmitted optically and received digitally can improve measurements. As another example, signals from different channels share the same fibre and travel through the same equipment in a Dense Wavelength-Division Multiplexing system. Hence, these signals experience similar processing, which once again yields correlations between each other exist. In all of the above cases, fusing information from these different sources provides more accurate monitoring.

More importantly, machine learning models can be made stronger by using an ensemble approach. This combines a set of relatively simple learning models to produce a much stronger learning model.

Exemplary mechanisms include:

- Direct concatenation of knowledge from different sources to produce a feature vector. This method is simple and useful as long as the number of features is small.
- Stage-based learning, where the identification of the data is divided into a set of stages, with each stage using an algorithm (that is possibly different from other stages) that is specific to that type of processing. This type of learning has the advantage of using many open source boosting and bagging algorithms.
- Deep neural networks, which have the advantage of more easily identifying new features that can then be analysed using another system.

The Data Ingestion Functional Block should use one or more of these methods to identify ingested data before it is passed to subsequent ENI Functional Blocks. This method may be used with any of the methods in clauses 6.3.2.4.3, 6.3.2.4.4 and 6.3.2.4.6.

6.3.2.4.6 Use of Formal Logic and Ontologies

If semantic information is known about the ingested data, then the Data Ingestion Functional Block should use formal logic and ontologies to identify and process ingested data. This method can be illustrated using two different approaches.

Many domains provide inherently relational information. In this case, probabilistic methods can be used to process the relational data; this is called Statistical Relational Learning (or sometimes, Probabilistic Logic Models). The advantage of this approach is that it can easily and succinctly represent probabilistic dependencies among the attributes of different related objects, which generates a compact representation of learned models. This is done by combining probability theory with first-order logic to represent the rich structure of the data.

Another method is inductive logic programming. This exploits contextual knowledge to constrain the search space while learning. The contextual knowledge consists of a set of facts and a small set of rules. Learning then consists of first, developing hypotheses by using the contextual knowledge to prove all positive examples; this is then refined so that no negative examples are proven. This method is often used to handle noisy data by maximizing the number of positive examples proved while minimizing the number of proved negative examples. It may be combined with fuzzy logic and/or probability theory.

This method may be used with any of the methods in clauses 6.3.2.4.3, 6.3.2.4.4 and 6.3.2.4.5.

6.3.3 Data Normalization Functional Block

6.3.3.1 Introduction

The Data Normalization Functional Block receives ingested data from the Data Ingestion Functional Block, which translates the data into a common format for further processing by the other Functional Blocks of the ENI System.

These characteristics give rise to the following requirements:

- ENI shall provide the ability to transform structured, semi-structured, and unstructured data from different data sources into a single common, uniform format.
- ENI shall use a set of models, including data types and data structures, to perform the transformation into a unified data format.
- Data Normalization may be realized as a Functional Block that is separate from the Data Ingestion Functional Block. This adheres to the Single Responsibility Principle [i.9], and enables a more scalable and robust system to be designed and built.

6.3.3.2 Motivation

Each domain produces data of interest to ENI using its own processes. Each such process may use different programming languages and protocols, which means that the same data may be received in completely different formats. For example, the same data about a customer may be processed using relational databases and directories; if these customer data are both ingested by the Data Ingestion Functional Block, then the data comprising the customer will be organized differently and may contain different information (e.g. a relational database is easier to query and can store richer data than a directory). If these differences can be identified early in the data acquisition process, then the knowledge processing process is greatly simplified.

In this example, recognizing that the object is a Customer, even though the data is different, may be done by comparing the ingested data with expected data from the model. For example, the Customer object in an information model has a set of mandatory (and optional) attributes, as well as relationships to other objects. It may also have metadata that disambiguates this instance of a Customer from all other instances of a Customer. The normalization process arranges and formats the ingested data that facilitates the comparison and recognition of these and other common characteristics and behaviours.

6.3.3.3 Function of the Data Normalization Functional Block

This clause explains the importance of data normalization.

Data normalization translates the data to a standardized form. This includes the use of pre-defined data structures, where data is converted to a standard format that uses a standard encoding. In the case of an ENI System, this is facilitated using its set of models. In ENI, models represent objects as templates; this includes defining a set of mandatory and optional attributes, operations, relationships, and other standard features. This enables ingested data to be compared to the same data that is error-free. This is discussed more in clause 6.3.3.4.

A normalized form is critical for the operation of other ENI Functional Blocks. A normalized form means that every ENI Functional Block may be designed using knowledge about the characteristics and behaviour of pertinent data and objects. Otherwise, each ENI Functional Block would have to accommodate data that could be represented in different formats using different data structures.

6.3.3.4 Operation of the Data Normalization Functional Block

6.3.3.4.1 Introduction

The principles of normalization, as used in relational calculus, are a good analogy for the operation of the Data Normalization Functional Block. This analogy is explained in clause 6.3.3.4.2, followed by an example of how data normalization may be used in machine learning in clause 6.3.3.4.3. Finally, clause 6.3.3.4.4 relates these concepts more specifically to the needs of ENI.

If the Data Ingestion Functional Block is not present, then the Data Normalization Functional Block shall perform all required actions that would have been performed by the Data Ingestion Functional Block. Otherwise, the following clauses define the actions that may be performed in the Data Normalization Functional Block.

6.3.3.4.2 Database Design Analogy (informative)

Normalization is a relational calculus design technique that organizes tables in a manner that reduces redundancy and dependency of data. This is done by dividing larger tables to smaller tables and linking the tables using relationships. This ensures that a table is about a specific topic, and that only supporting topics are included. For example, a table that contains information about sales people, support technicians, and customers serves several purposes:

- Identify sales people in an organization.
- Identify support technicians in an organization.
- Identify all customers of an organization.
- Identify which sales people call on which customers.
- Identify which support technicians help which customers.
- Identify which sales people rely on which support technicians.

There are several advantages of having a highly normalized data schema:

- Increased consistency. Information is stored in one place only, reducing the possibility of inconsistent data.
- Single purpose. By limiting a table to one purpose, the number of duplicate data contained within the database is reduced; this eliminates many issues encountered when the database is modified.
- Simplify queries. Normalizing a database identifies different dependencies between data, and makes querying more efficient.
- Easier object-to-data mapping. Highly-normalized data schemata are closer conceptually to object-oriented schemata because the object-oriented goals of promoting high cohesion and loose coupling between classes result in similar solutions.

There are five standardized forms of data normalization. Brief descriptions of the first three are:

- An entity type is in First Normal Form when all entities have a unique identifier, and when each column does not contain any repeating groups of data (i.e. each table cell has a single value, and each record is unique) or composite fields.
- An entity type is in Second Normal Form when it is in First Normal Form and when all columns that do not contain unique identifiers depend on the entire unique identifier(s), and not just a part of the identifier.
- An entity type is in Third Normal Form when it is in Second Normal Form and when each column that is not part of the unique identifier does not depend on another column that is not part of the unique identifier (i.e. changing a non-key column does not cause other non-key columns to change - this is called a transitive functional dependency).

The primary disadvantage of normalization is slower reporting performance. This is accomplished through using a separate, denormalized data model.

6.3.3.4.3 Normalization for Machine Learning

A neural network may be trained by supplying data and then comparing the expected output to the true output of the network. The model parameters may then be updated using a number of algorithms. For example, gradient descent updates the parameters of the model in the direction that minimizes the difference between the expected (or ideal) outcome and the true outcome. There are different types of gradient descent; however, all of them scale the magnitude of the parameter update by a learning rate. This rate ensures that the parameter is not being changed too drastically, which can cause the update to overshoot and fail to find the optimal value. The normalization of the input data to a standard scale enables the network to *more quickly* learn the optimal parameters for each input node. It also reduces complex computational problems associated with floating point number precision.

In a neural network, changing one weight affects subsequent layers, which then affect subsequent layers, and so on. This means changing one weight can affect activations in subsequent layers in complex ways. By ensuring the activations of *each* layer are normalized, the overall loss function topology is simplified. This is especially helpful for the hidden layers of the network, since the distribution of unnormalized activations from previous layers will change as the network evolves and learns more optimal parameters.

One method for doing this is called batch normalization. This controls the magnitude and mean of the activations independent from all other layers. Other methods include weight normalization (i.e. instead of normalizing activations in the intermediate layers of neural networks, the weight vectors of a neural network are reparameterized to enable optimization to converge more quickly); layer normalization (i.e. instead of normalizing the input features across the batch dimension as in batch normalization, it normalizes the inputs across the features); group normalization (i.e. the mean and standard deviation over groups of channels for each training example).

6.3.3.4.4 Applying Normalization to ENI

In ENI, the Normalization Functional Block uses modelled information (e.g. objects, attributes, operations, and relationships) to *standardize* the information being sent to other ENI Functional Blocks. ENI may use a number of different mechanisms, including those described in the previous two clauses, to identify and convert data into model elements (e.g. classes, attributes, operations, and relationships). Metadata may also be used to describe and prescribe how normalization is performed; this may help reversing the process when ENI needs to output recommendations and commands to the Assisted System (and/or its Designated Entity). In order to do this, accumulated knowledge from other ENI Functional Blocks (included predefined model information) shall be made available to the Normalization Functional Block, as shown in Figure 6-14. The functional block diagram shown in Figure 6-14 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of normalization. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

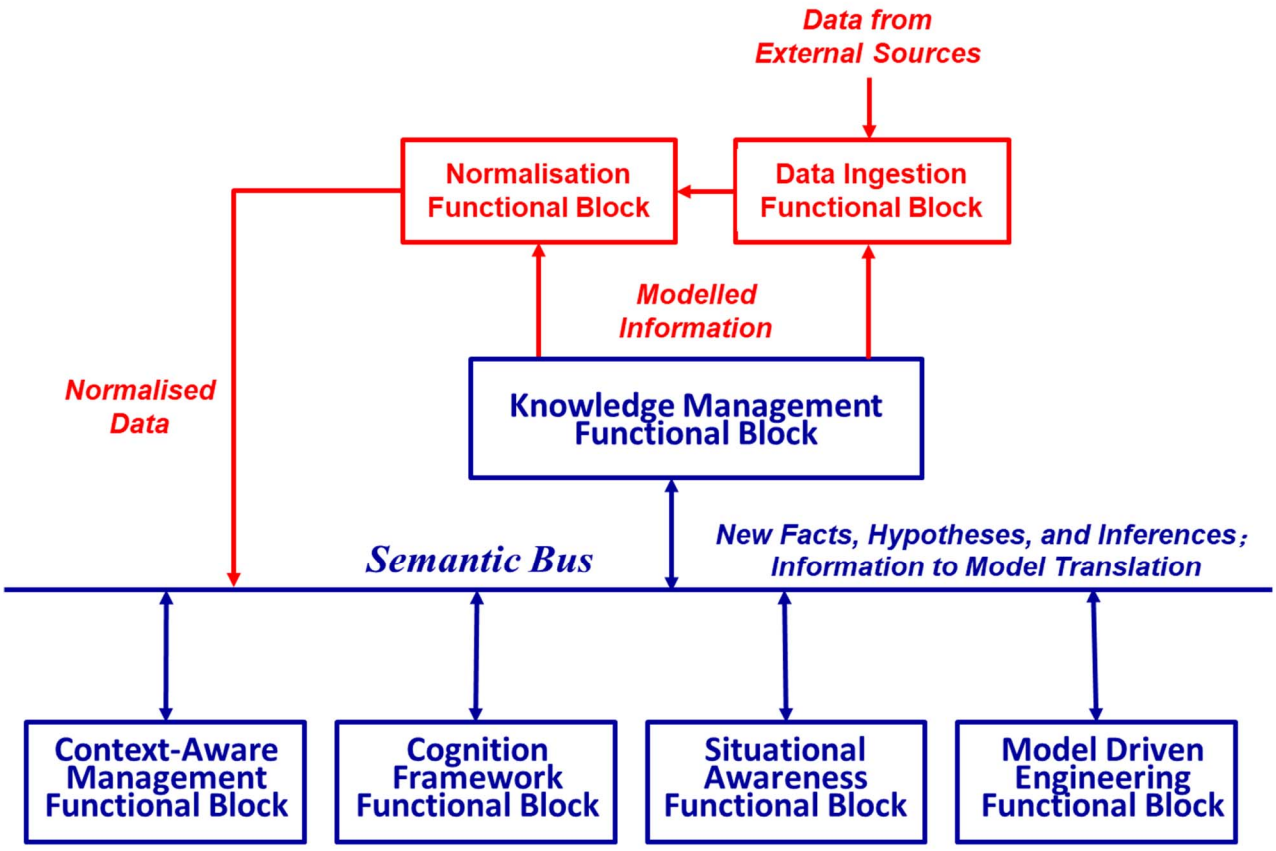

Figure 6-14: Data Normalization Operation

Data from the Normalization Functional Block is sent to the Semantic Bus [i.30]. This is a uni-directional communication; information from the Semantic Bus is not sent to the Normalization Functional Block; it is first filtered by the Knowledge Management Functional Block.

Knowledge from the Context-Aware, Cognition Management, Situational Awareness, and Model Driven Engineering Functional Blocks result in changes to the models used by ENI. Changes can be in the form of new model elements as well as corrections and elaborations to existing model elements. These changes are important, as it enables the modelled information to evolve as the ENI System learns; it also enables the knowledge itself to adapt (e.g. be used in different ways) as the context and situation changes. Hence, the ENI models are a form of active repository. The results are sent to the Knowledge Management Functional Block. This is discussed more in clause 6.3.4.

6.3.3.4.5 Storing Normalized Telemetry Information

Telemetry information shall be gathered and normalized as directed by the ENI System. Specifically, this includes:

- ENI Policies shall be used to define what telemetry information is desired, where it shall be collected, and the frequency of collection.
- ENI Policies may specify one or more collection methods in prioritized order.
- Raw telemetry information shall be ingested using the appropriate ENI External Reference Point (see clause 7.3).
- Telemetry information that is deemed valuable for making future decisions should be stored in an appropriate repository (see clause 6.3.4.5). Stored telemetry information may be anonymized or pseudonymized as appropriate, and should be compressed.

6.3.3.4.6 Changing Telemetry Gathering using Policies

The ENI System should use an adaptive control loop (see clause 6.3.4.7 and ETSI GR ENI 017 [i.36]). This enables its actions to reflect changes in user needs, business goals, and environmental conditions. As these elements change, the ENI System may need to gather new telemetry (e.g. to verify hypotheses made). All requests to gather telemetry should be made using ENI Policies.

The ENI System shall use an appropriate mechanism to normalize ingested data. The normalization may consist of multiple processing stages. However, the result shall conform to the ENI models being used.

6.3.3.4.7 Cognitive and Situation-Aware Directed Normalized Telemetry Gathering

This process is the same as that described for clause 6.3.3.4.6, except that metadata and/or ENI Policies should be used to specify any special processing requirements to normalize newly specified telemetry to be gathered. This method may be used with any of the methods in clauses 6.3.3.4.8, 6.3.3.4.9, 6.3.3.4.10 and 6.3.3.4.11.

6.3.3.4.8 Use of Metadata

The Data Normalization Functional Block should use metadata to describe and prescribe processing that should be done on different types of ingested data to ensure its proper normalization. Since metadata may contain descriptive as well as prescriptive information, metadata may also contain important information that defines which processing operations should be executed to normalize the ingested data before it is sent to the next Functional Block. This method may be used with any of the methods in clauses 6.3.3.4.9, 6.3.3.4.10 and 6.3.3.4.11.

6.3.3.4.9 Use of Structure, Pattern, and Feature Matching

The Data Normalization Functional Block should use the same set of related model-based approaches to help normalize ingested data as those described in clause 6.3.2.4.4. This method may be used with any of the methods in clauses 6.3.3.4.8, 6.3.3.4.10 and 6.3.3.4.11.

6.3.3.4.10 Use of AI-based Mechanisms

The Data Normalization Functional Block should use the same set of AI-based mechanisms to help normalize ingested data as those described in clause 6.3.2.4.5. This method may be used with any of the methods in clauses 6.3.3.4.8, 6.3.3.4.9 and 6.3.3.4.11.

6.3.3.4.11 Use of Formal Logic and Ontologies

The Data Normalization Functional Block should use the same set of Formal Logic mechanisms to help normalize ingested data as those described in clause 6.3.2.4.6. This method may be used with any of the methods in clauses 6.3.3.4.8, 6.3.3.4.9 and 6.3.3.4.10.

6.3.4 Knowledge Management Functional Block

6.3.4.1 Introduction

Knowledge management is fundamental to all disciplines of modelling and AI. Briefly, knowledge representation defines a set of formalisms that define data, information, and knowledge in a form that a computer system can utilize. Knowledge management includes processes, strategies, and systems that create, revise, sustain, and enhance the storage, assessment, use, sharing, and refinement of knowledge assets using a consensual knowledge representation.

Knowledge management also enables machine learning and reasoning - without a formal and consensual representation of knowledge, algorithms cannot be defined that reason (e.g. perform inferencing, correct errors, and derive new knowledge) about data, information, and knowledge. Knowledge representation defines mechanisms for the characteristics and behaviour of the set of entities being modelled; this enables the computer system to plan actions and determine consequences by reasoning using the knowledge representation, as opposed to taking direct action on the set of entities.

This Functional Block also manages the Data and Knowledge Repositories of ENI. In addition, this Functional Block is responsible for managing the lifecycle of the knowledge in the Data and Knowledge Repositories. Knowledge obtained from one ENI System may be reused by another ENI System or the Assisted System, through dedicated External Interfaces that share part or all of the knowledge repositories of the two systems. This shall be subject to access control permissions. The Knowledge management Functional Block may check, or instruct the Knowledge Repository to check, if any of the required knowledge is already available, before carrying out any tasks. If that knowledge exists, then conflict resolution (see clause 6.3.4.6.3) will be performed to either edit the existing knowledge or add the new knowledge. It may also retrieve knowledge from the Knowledge Repositories, if such knowledge is available; or instructing the intelligent system to collect appropriate data if the requested knowledge is not available. The Knowledge Management Functional Block shall proactively publish new knowledge to the knowledge repository, where any consumer (including ENI Functional Blocks and external systems) interested in that knowledge may retrieve it.

Knowledge management works with data, information, knowledge, and wisdom; see "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]). In particular, this Functional Block directs which context and situation information are applied to the raw data, transforming it to information and then knowledge. If appropriate, wisdom is then derived from those two translations. As such, it is recommended that data is always first transformed into information; this is because there is no context or situation knowledge to understand the data.

The Knowledge Management Functional Block provides foundational functionality that shall be used to analyse, validate, and infer new and changed data, information, knowledge, and wisdom. Thus, it is used in a variety of tasks, including analysis and decision-making.

6.3.4.2 Inferencing

An inference system automatically extends the knowledge base of the system. If inductive or abductive inferencing is used, then the conclusions are not guaranteed, and some knowledge may be in error. ENI shall use inferencing to extend and correct its knowledge base. For example, an inference system can gather evidence and develop a hypothesis, which can later be proved to be true or false. Such hypotheses can also be generalized. For instance, suppose that ENI has observed multiple occurrences of a set of faults that cause performance degradation that eventually lead to a Service Level Agreement (SLA) violation. A possible conclusion could be that whenever this set of faults occurs, an SLA will always occur. That may not be true. In this situation, ENI may add this conclusion to its knowledge base while using metadata to describe its source and/or add its approximate probability. Since ENI is an experiential system, ENI shall update the metadata (and most importantly, its probability) as it gathers more evidence to prove or disprove this hypothesis. ENI may also use multiple types of inferencing to validate hypotheses. For example, in addition to abduction, ENI may use reinforcement learning to more quickly converge to a conclusion as to the validity of the hypothesis.

Inferencing is critical for realizing cognition. ENI uses a closed control loop powered by a model-driven architecture. The closed control loop is based on FOCALE's extensions to the Observe-Orient-Decide-Act (OODA) control loop [5], [i.1], [i.2] and [i.4]. See clause 6.3.4.7 for more details.

6.3.4.3 Motivation

A knowledge representation framework defines a set of primitives that define and describe the meaning, and optionally additional semantic characteristics (e.g. synonyms and antonyms), of concepts important to the managed environment that can be processed by a computer system. This includes how these concepts relate to each other.

Key concepts that determine the function and nature of a knowledge representation framework include:

- How can knowledge be represented
- How can knowledge be discovered, extracted, translated, and searched
- How can knowledge be used for further inference
- How can knowledge from different sources be normalized and composed into different knowledge

A knowledge framework can be as simple as a collection of data structures and tools to manipulate those data structures, to an architecture for representing and processing knowledge (e.g. semantic networks), to dedicated languages for representing knowledge (these also include languages known as theorem provers that are typically based on First Order Logic). Ontologies, and ontological tools, are typically used to provide and/or reinforce knowledge that is linguistic in nature and/or is amenable to logic-based processing. Finally, dedicated tools for knowledge extraction, filtering, and fusion may be required in environments where knowledge from different data sources needs to be combined, filtered, correlated, or otherwise processed in order to produce new knowledge.

Knowledge frameworks are specific to the applications that use them. The knowledge framework of ENI is designed to support situation awareness and cognitive decision-making.

6.3.4.4 Knowledge Processing

6.3.4.4.1 Knowledge Representation and Enhancement

The present document does not define a single approach to represent knowledge, since knowledge is dependent on context, situation, and usage. Rather, it describes how models and ontologies may be used to build a lexicon that represents knowledge in an extensible manner (see clause 5.6 for knowledge management requirements).

ENI shall provide at least one separate repository for processing changes to its knowledge base.

Information and data models are far more common than ontologies for networking, especially for representing low-level concepts such as network telemetry. Hence, the ENI System shall use an information model as the authoritative source for defining concepts in a technology neutral fashion.

An information model represents concepts in their pure form. In the real world, concepts are reified as data models, which binds the concept to a particular implementation. The ENI System shall use its information model to construct a set of data models according to the needs of the system.

In this approach, the information model serves as the authoritative source for defining the concepts, and the set of data models represent commonly encountered data from the real world. This basic combination is more useful than it appears. For example, given the knowledge of an output format of a command, this combination may be used to determine if data is missing when that command is executed. However, it lacks both general semantics as well as usage for a given context and/or situation. For example, telemetry may indicate that 42 bytes were dropped. This data is not usable, since it is not known what type of message the bytes were from, and what protocol was used to send the message. Knowledge of both of these enable the system to conclude what, if any, action to take (e.g. if this was a control or management message, then it needs to be retransmitted; if this was data, then determine whether the protocol will do the retransmission automatically or not). As another example, suppose that a business policy states that no one can access the code server unless they are connected on the company intranet. In this context, the type of connection determines whether actions are allowed or disallowed, as well as whether other actions should be taken (e.g. determine if this is a malicious attack or not).

Thus, most data and information should require the addition of context and situation information to enhance it for decision-making processes. However, neither of these will provide all of the semantics that are possibly required for robust decision-making. This is because both context and situation are, fundamentally, facts. While additional inferences may be derived from both, neither provides additional semantics for the data and information.

Therefore, the ENI System should add ontological data and processes to the models. One way to do this is to build a multigraph (or a set of multigraphs), as shown in Figure 6-15. In this approach, the models and the ontologies are both represented as graphs; semantic edges (i.e. relationships of a semantic nature, such as synonymy and meronymy) are then created between the graphs to define how one set of concepts is related to the other set of concepts. The resulting multigraph consists of semantic relationships that join the model on the left to the set of ontologies on the right; these are represented by the double-edged arrow connecting them for simplicity. This semantic representation is built iteratively, and is summarized in Figure 6-15.

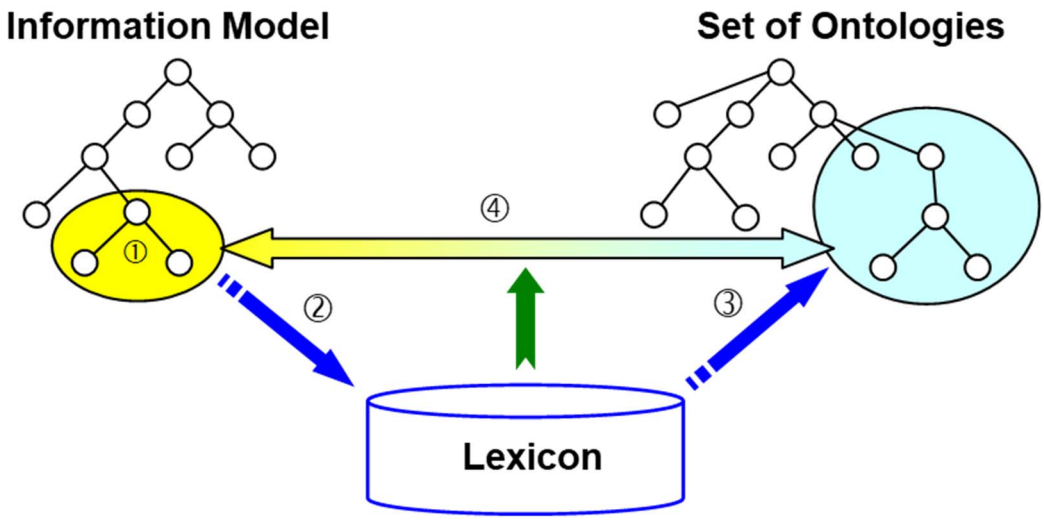

Figure 6-15: Knowledge Representation and Enhancement

Step 1 starts with identifying one or more model elements in the information model. Step 2 may use a number of different tools, including computational linguistics, semantic equivalence, and pattern and structural matching to relate the set of model elements to a set of terms in the lexicon. This broadened view is then used to search for semantically equivalent concepts in a set of ontologies. In general, this method will identify a part of a taxonomy without identifying all concepts in the taxonomy. Each identified concept may be weighted and, for the moment, is represented as a uni-directional edge, forming a directed graph. Each ontology concept that was identified in the semantic matching process in step 3 may then be examined to see if it is related to other concepts in this or other ontologies. As each new concept is found, it shall be marked (e.g. annotated and/or added to a list) for possible addition to the existing concepts that were already matched from the lexicon. Step 4 then takes the semantic concepts and relationships and determines if there are any new model elements that are related to them. In this step, the weight of the edge representing the relationship may be adjusted, as well as its directionality; however, if it is found to be bi-directional, then this should be represented using two edges, since in general, the nature of the relationship will differ. Step 4 is bidirectional, which signifies that the above process is iterative. This process shall iterate until either no more relationships can be found, or until a sufficient number of relationships are found. When a specific task is being executed, often all that is required is to find enough matching elements to either prove or disprove a hypothesis.

Figure 6-15 has been simplified to only show the process for finding concepts that are directly related to the concept being searched. For example, if a concept is found that is not directly related to a term in the, then additional checks are formed to see if:

1) it is indirectly related; or

2) it should be added as a new relation (i.e. this represents new knowledge).

The underlying idea for these additional checks is to verify that each new concept *reinforces* or *adds additional support* for the concept that was already selected. Hence, this process can be thought of as *strengthening the semantics* of the match. As larger groups of concepts are matched to larger groups of model elements, a stronger correlation between the meaning of the grouped concept and the group of facts is established. This is, in effect, a self-check of the correctness of the mapping, and is used to eliminate concepts and model elements that match each other, but are not related to the managed entity that is being modelled.

In the above process, the types of semantic relationships may be changed to achieve a closer amount of semantic closeness. This may help speed the alignment process.

6.3.4.4.2 Knowledge Normalization

The discussion in the previous clause shows the complexity that may be involved in generating and validating knowledge. Therefore, knowledge needs to be normalized.

Knowledge normalization requires the correct meaning of a word or phrase to be discovered so that incorrect and ambiguous information can be corrected. Many of the same techniques discussed in clauses 6.3.3.4.8, 6.3.3.4.9, 6.3.3.4.10 and 6.3.3.4.11 of the Data Normalization Functional Block may also be used to normalize knowledge. AI-based techniques for natural language processing are of particular interest. For example, Bidirectional Encoder Representations from Transformers (BERT) is a context-dependent algorithm that considers all of the words of an input sentence simultaneously and then uses an attention mechanism to develop a contextual meaning of the words. This means if the same word appears twice in the same sentence, but has different meanings, (e.g. "He is banking on the top 5 banking stocks increasing in value"), each meaning will have a different embedding. This enables common recognition and classification tasks to be performed. Examples include:

1) determine if a question-answer pair is relevant or not;
2) classify a single sentence (e.g. detect if the input sentence is spam or not);
3) generate an answer to a given question; and
4) single sentence tagging tasks such as named entity recognition.

See "Introduction to Artificial Intelligence Mechanisms for Modular Systems" (ETSI GR ENI 018 [i.37]) for a description of transformers and related algorithms for natural language processing.

Natural language processing models excel at identifying the meaning of a word or phrase in a sentence. This can then be coupled with ontologies and formal logic to ensure that when a given word or phrase is changed, both the meaning of that word or phrase as well as the meaning of the sentence are not altered. Natural language processing models, such as transformers, should be used in an ENI System to recognize and normalize knowledge.

Natural language processing is particularly important for processing ENI Intent Policies. This is because ENI Intent Policies are written using a restricted natural language. However, they are also useful for other applications, such as business rules, contextual, and situational information, as well as any other input that can be expressed using natural language.

6.3.4.4.3 Transforming Data, Information, and Knowledge into Wisdom

The Data-Information-Knowledge-Wisdom (DIKW) hierarchy is often used to contextualize data, information, knowledge, and wisdom, and to identify and describe the processes involved in the transformation of an entity at a lower level in the hierarchy (e.g. data) to an entity at a higher level in the hierarchy (e.g. wisdom). Each of the higher types in the hierarchy includes the categories that fall below it. See "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for more background information for a more thorough discussion of the DIKW hierarchy.

The ability to transform an entity at a lower level in the hierarchy to an entity at a higher level in the hierarchy shall be based on the *understanding* required to move to that higher level. Hence:

- transforming data into information involves understanding how contextual and situational information are used to add meaning to ingested data;
- transforming information to knowledge involves understanding complex relationships to detect patterns of and relationships between information;
- transforming knowledge to wisdom involves understanding the relevance and significance of information and knowledge and, more importantly, why the information and knowledge exists.

Data shall be transformed into information when relations between data, and the context that generated the data, are understood. This also includes understanding the relevance of data given the current context or situation. Information shall be transformed into knowledge when what the information means, as well as the patterns that generated the information, are understood. Knowledge consists of a mix of contextual information, values, experience, and rules that represent situational choices. Knowledge involves the synthesis of multiple sources of information over time, and hence, should be refined experientially. Knowledge shall be required to perform decision-making and predict future outcomes.

Knowledge shall be transformed into wisdom when the specific occurrence of specific data, information, and knowledge is understood. Wisdom is accumulated knowledge that enables situational decision-making to be performed accurately and efficiently. Wisdom shall enable more effective planning and prediction.

6.3.4.4.4 Semantic Bus

A Semantic Bus is a type of message bus used to orchestrate and filter communications between ENI Functional Blocks based on the meaning, attributes, and metadata of a message using a shared set of interfaces. See "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for more background information.

A Semantic Bus shall use meaning, attributes, and metadata for the following tasks: publication of and subscription to messages, routing messages, and transformation of message content. For example, message content can be normalized (to use consensual terminology and concepts) by the Semantic Bus in order to facilitate different endpoints understanding a message. As another example, content can be orchestrated by routing on the meaning of the message, as opposed to just attributes. This shall be performed by semantically interpreting a message (including its payload) using an ontology and/or natural language processing (see clause 6.3.4.4.2).

6.3.4.5 Repositories

6.3.4.5.1 Overview

The Knowledge Management FB shall contain at least four different types of repositories:

- a **Data Repository**, to store the collected data from different parts of the Assisted System after appropriate translation by the API broker;
- a **Knowledge Repository**, to store knowledge and wisdom (see clause 6.3.4.5);
- a **Model Repository**, to store the information model, data models, and ontologies;
- a **Blackboard Repository**, to provide a shared working space for computations.

Each Repository shall have a different Internal Reference Point, with different APIs, for communication with other ENI functional blocks. This is necessary to accommodate different access permissions to the types of data and information that are stored in each Repository.

Each Repository may be used to store and retrieve historical data and information of its appropriate type.

An ENI system shall have the capability of proactively publishing new data and knowledge to the respective repositories, and proactively retrieving updated data and knowledge from both repositories as needed.

Figure 6-16 shows a functional overview of the Repository Management Functional Block, which is a part of the Knowledge Management Functional Block. The following clauses are organized as follows. Clauses 6.3.4.5.1 through 6.3.4.5.5 explain the function of the different Repositories. Then, clause 6.3.4.5.6 provides a functional overview of how the rest of the Functional Blocks operate to manage the different Repositories, as well as communicate to the other ENI Functional Blocks. Finally, clause 6.3.4.5.7 describes semantic querying.

Figure 6-16 only shows the Semantic Bus that is used by different Functional Blocks for communication; it does not show any Internal Reference Point of the Knowledge Management Functional Block for simplicity.

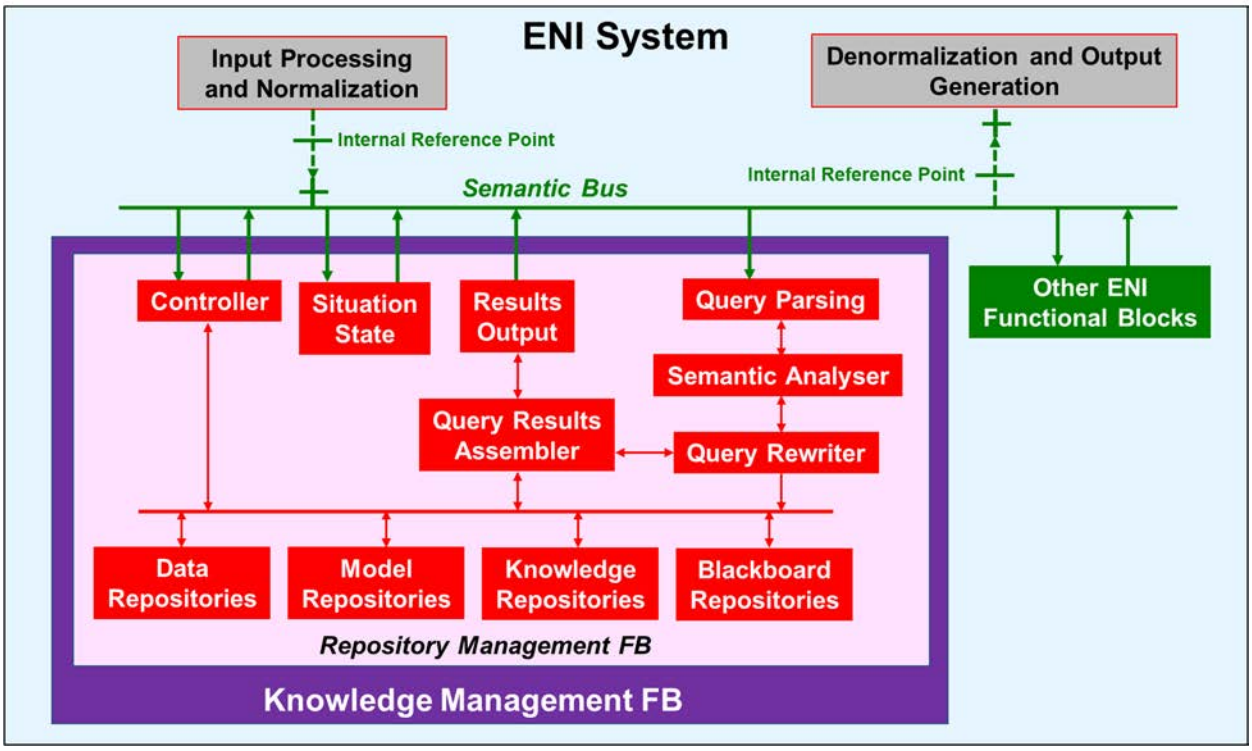

Figure 6-16: Repository Management Functional Block

6.3.4.5.2 Data Repository

The ENI System shall provide one or more Data Repositories for collecting ingested data and information from external sources. The Data Repository may contain raw network data, obtained using the appropriate External Reference Points, between an ENI System and the Assisted System, as well as contextual, business, and other types of data and information. The Data Repository should interact with the Context Aware, Cognition Management, Situation Awareness, Model-Driven Engineering, and Policy Management Functional Blocks using the Semantic Bus. In addition, a set of Internal Reference Points may be used for direct communication between ENI Functional Blocks.

The Data Repository interacts with the Data Ingestion and Normalization Functional Blocks to ingest and store input data and information to ENI functions, and interacts with the Denormalization and Output Generation Functional Blocks to store and provide output data from ENI.

6.3.4.5.3 Model Repositories

The ENI System shall use a single Information Model, as well as a set of Data Models that are each derived from the Information Model, as the foundation of its knowledge representation. The ENI System may use a set of ontologies to augment the knowledge representation with deeper semantics.

The ENI Information Model, along with its set of derived Data Models and, possibly, Ontologies, represent the foundation of how ENI represents knowledge. Therefore, additional security mechanisms shall govern operations on accessing, updating, using, and sending the information and data associated with any Model Repository.

The Model Repository should interact with the Context Aware, Cognition Management, Situation Awareness, Model-Driven Engineering, and Policy Management Functional Blocks using the Semantic Bus. In addition, a set of Internal Reference Points may be used for direct communication between ENI Functional Blocks.

6.3.4.5.4 Knowledge Repositories

The ENI System shall use at least one dedicated repository, called the Knowledge Repository, to store, manipulate, edit, and retrieve knowledge and wisdom. This may include axioms, theories, and hypotheses.

The ENI System should annotate hypotheses with their current state (i.e. proven, disproven, or work in progress). This annotation may include the context and/or situation in which the hypothesis was made.

The Knowledge Repository should interact with the Context Aware, Cognition Management, Situation Awareness, Model-Driven Engineering, and Policy Management Functional Blocks using the Semantic Bus. In addition, a set of Internal Reference Points may be used for direct communication between ENI Functional Blocks.

6.3.4.5.5 Blackboard Repositories

The ENI System should use at least one repository that works as a blackboard system. A blackboard system uses a shared workspace that a set of independent agents contribute to, which contains input data along with partial, alternative, and completed solutions. Both the blackboard and the contributing agents are under the control of a dedicated management entity. Each agent is specialized in its function and operation, and typically is completely independent of other agents that are using the blackboard. A controller monitors the state of the contents of the blackboard and synchronizes the agents that are working with the blackboard. This is a basic software architectural pattern ([i.19]).

ENI may use a variant of a blackboard architecture, in which the blackboard is divided into a set of layers. This type of system is shown in Figure 6-17.

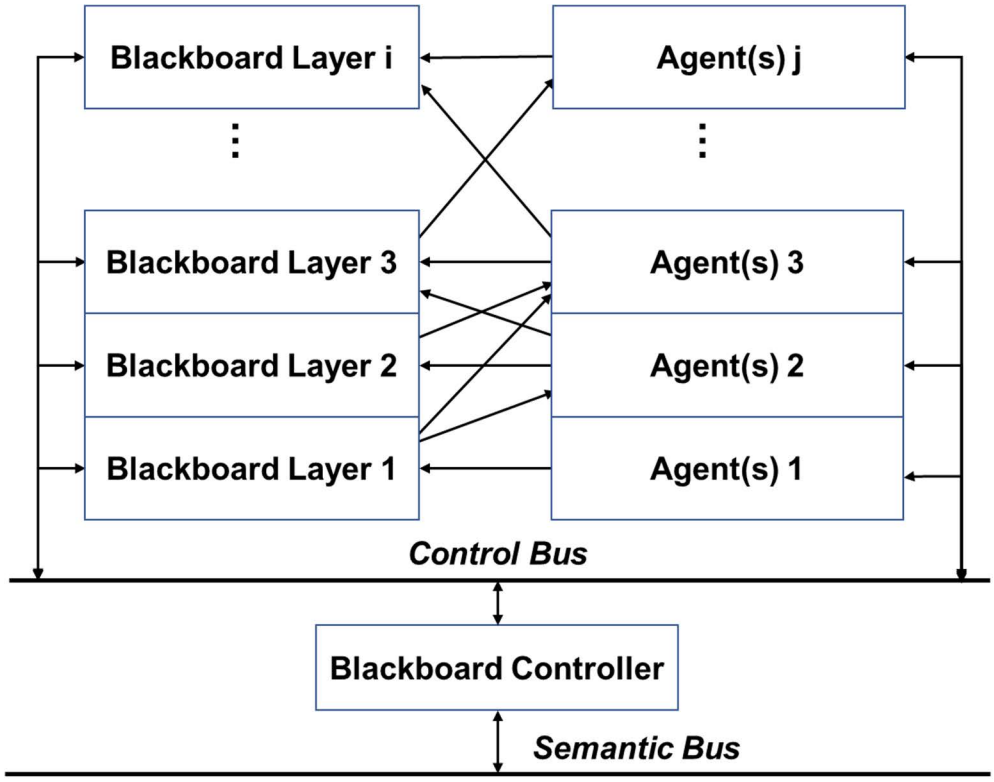

Figure 6-17: Blackboard Architecture

In Figure 6-17, each blackboard layer is a shared working space, and ingests data and information from a set of agents that are at that level. A level could be one hypothesis from a set of hypotheses, or a particular stage in the computation of a program, or other similar stages. For example, if the blackboard was being used as part of a larger compiler program, then level 1 could be the raw input, level 2 could be part-of-speech tags, level 3 could be grouping those tags into larger phrases, and so forth.

The lowest level agent typically supplies the raw data and information to the blackboard system. Higher-level agents may have additional data and information received from outside the blackboard system, as well as inputs from one or more layers of the blackboard, and infer new entities to be stored in one or more higher layers.

The blackboard controller is responsible for coordinating the blackboard and the agents. This is typically done using an inference cycle, which is driven by a set of events received. The blackboard controller responds to an event by publishing a message to all agents that are interested in this event. The ENI System may prioritize the publishing of events to a set of agents by using a set of criteria, such as relevance, context, and other factors. The blackboard ingests the input data and information from the affected agents and produces new entities on the blackboard, which triggers a new event. The blackboard controller examines the entities produced by the last inference cycle, and then a new inference cycle starts; this process iterates until no other agent can contribute anything else.

NOTE: This process may not succeed (e.g. a hypothesis cannot be proven in finite time); however, this information is valuable in and of itself.

The Blackboard Repository should interact with the Context Aware, Cognition Management, Situation Awareness, Model-Driven Engineering, and Policy Management Functional Blocks using the Semantic Bus. In addition, a set of Internal Reference Points may be used for direct communication between ENI Functional Blocks.

6.3.4.5.6 Repository Operation

Each of the four types of repositories described in the preceding clauses operates in a common manner, which is described below. Each repository may be realized as an active repository (i.e. one that pre- and/or post-processes information that is stored or retrieved). In addition, the Blackboard Repository may be used by any of the other repositories to aid in processing queries.

The Repository Management Functional Block should provide a repository service that clients may subscribe to without having to know about the internal structure of any repository. Therefore, a query for information or data shall be sent either using the Semantic Bus or, alternatively, using an appropriate Internal Reference Point, to the Knowledge Management Functional Block. This query is read by the Query Parsing function of the Repository Management Functional Block, which parses the query for completeness and syntactic correctness. The results are then sent to the Semantic Analyser, which analyses the meaning of the query with respect to the current context and/or situation. This enables the Repository Management Functional Block to understand which set of Repositories contain data and/or information that can satisfy the query (see clause 6.3.4.5.3). Accordingly, the Query Rewriter takes this information and rewrites the original query into a set of queries, where one or more of the rewritten queries are formulated specifically for a particular repository. The rewritten set of queries may target one or more repositories.

Each rewritten query is sent to the appropriate repository. The results of each rewritten query are collected by the Query Results Assembler, which organizes the results into one or more result statements; these result statements are then sent by the Results Output to the Semantic Bus (or alternatively, to an appropriate Internal Reference Point).

The Controller monitors the external state of the ENI System, and correlates that with the internal state of the Repository Management Functional Block. This is used to synchronize the blackboard (if it is used), along with the querying and collecting of results from each repository.

Note that while the Data Repositories are typically used to store and retrieve data, other Repositories may actively pre- and post-process information that they store and retrieve. For example, an ontology in the Model Repository will make inferences, which produces new data and information. Once validated, those changes shall be added to affected Repositories by the Knowledge Management Functional Block, which shall publish a notification of those changes to the Semantic Bus. Such Repositories are referred to as Active Repositories.

6.3.4.5.7 Semantically Augmented Query and Learning

The purpose of the Semantic Analyser shown in Figure 6-16 is to analyse the meaning of the query with respect to the current context and/or situation. More specifically, the context and/or situation may alter the meaning of the query. Hence, the Semantic Analyser shall use an ontology and/or natural language processing (see clause 6.3.4.4.2) to ensure that the query as written takes contextual and situational data into account. For example, a query may ask for all authenticated connections to a domain. However, a business rule could be in force that restricts the type of connections to that domain to a particular type (e.g. only the private Intranet is allowed to connect to the domain). Therefore, when the query is examined by the Semantic Analyser, it shall understand the constraints imposed by the situation and return only those connections from the private Intranet that have been authenticated.

6.3.4.6 Function of the Knowledge Management Functional Block

6.3.4.6.1 Introduction

The purpose of the Knowledge Management Functional Block is to discover, analyse, validate, and infer new and changed data, information, knowledge, and wisdom. See "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for more background information.

The Knowledge Management Functional Block shall attach multiple contextual- and/or situational-specific meanings to a word, phrase, or concept. This enables the ENI System to account for the correct meaning of that word, phrase, or concept when user needs, business goals, and/or environmental conditions change. The attachment of multiple contextual- and/or situational-specific meanings to a word, phrase, or concept shall be done during the Knowledge Processing phase (see clause 6.3.4.4) and stored in one or more appropriate Repositories (see clause 6.3.4.5). The attachment should include metadata that defines the context and/or situation for each meaning.

6.3.4.6.2 Grounding Knowledge Using Semantics

Context defines the knowledge about an Entity that exists or has existed. It is modelled as a set of objects and relationships. The context model shall include mechanisms that represent changes in the environment, as well as changes in behaviour of an ENI Functional Block as a set of closed loop systems.

There are a number of existing technologies and open source that can be used to represent context and knowledge.

6.3.4.6.3 Resolving Knowledge Conflicts

A knowledge conflict is defined when the definition of a word, phrase, object, or behaviour does not match its stored definition. The Knowledge Management Functional Block shall be able to detect such conflicts. The Knowledge Management Functional Block need not declare that a knowledge conflict exists until suitable processing (e.g. as described in clause 6.3.3.4.3) has been performed.

There are three potential reasons for a knowledge conflict. First, an error was made in storing the previous definition. The Knowledge Management Functional Block shall compare the previous definition with the current definition using semantic reasoning as described in clauses 6.3.3.4.8 and 6.3.3.4.9), AI-based natural language processing (e.g. as described in clause 6.3.3.4.10), and/or formal logic (e.g. as described in clause 6.3.3.4.11).

Second, contextual and/or situational information was not correctly taken into account. The Knowledge Management Functional Block shall compare the metadata of the current definition to the metadata of all stored definitions to determine if this was the case and, if so, correct it.

Third, a new meaning of that word, phrase, object, or behaviour was discovered. The Knowledge Management Functional Block shall first perform the above two checks. Then, the Knowledge Management Functional Block may define a hypothesis to test the new definition, and attempt to prove the hypothesis using either AI-based natural language processing and/or formal logic. If the hypothesis is validated, then the new definition shall be added; metadata defining the current context and/or situation shall also be added to the definition.

6.3.4.6.4 Knowledge Distribution

Knowledge should be distributed using the Semantic Bus (see clause 6.3.4.4.4). Knowledge may be distributed using a shared repository or via file transfer if using the Semantic Bus does not meet application-specific requirements. See "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for more background information on the Semantic Bus.

6.3.4.7 Operation of the Knowledge Management Functional Block

6.3.4.7.1 Introduction

Figure 6-18 shows a highly simplified view of basic knowledge processing based on the OODA control loop [5]. The functional block diagram shown in Figure 6-18 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of knowledge management. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

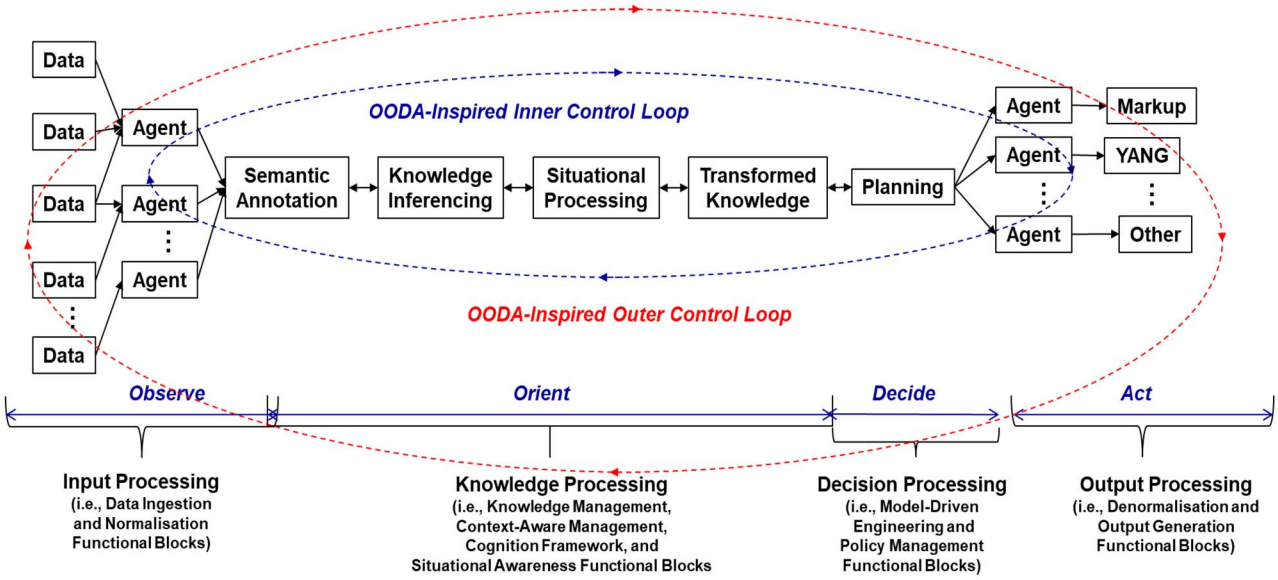

Figure 6-18: Simplified Knowledge Processing Flow

Figure 6-18 uses a multi-level set of OODA-inspired control loops [5], similar to FOCALE [4]. Both of these are discussed in more detail in clauses 6.3.4.7.2 through 6.3.4.7.5.

The assignment of Functional Blocks to each stage of OODA processing illustrates the main activities of those Functional Blocks. Some Functional Blocks, such as Knowledge Management, may perform tasks in other parts of the OODA loop. For example, the Knowledge Management Functional Block provides the knowledge representation used by all ENI Functional Blocks. In FOCALE, this principle was used to facilitate input processing of ingested data by matching the data, along with observed patterns of data, to known knowledge to more quickly and efficiently validate and verify ingested data. As another example, the Situational Awareness Functional Block is responsible for understanding the relevant changes that pertain to the data, information, and behaviour of the Assisted System, how those changes affect the goals of the ENI System, and to project what is likely to happen in the future (see clause 6.3.6). This spans the Orient and Decide functions of OODA.

Figure 6-18 is depicted as a sequential flow for simplicity. However, OODA is a form of closed control loop, and hence, the operations in the Knowledge and Decision Processing groups of functional blocks typically interact in a non-sequential manner. This is made more apparent in FOCALE, which is an autonomic architecture using an enhanced version of the OODA control loop.

The following clauses describe the functionality of the Knowledge Processing and Decision Processing functions in the context of an OODA control loop. In actuality, this is a hierarchical set of OODA-inspired control loops; the "outer" control loop consists of the four sets of processing operations, and the "inner" control loop is the Knowledge and Decision Processing operations. This inner loop is necessary to properly process and derive knowledge and wisdom, and perform accurate inferencing (see clause 6.3.5), on contextually-changing information.

Reference [i.1] describes why the OODA loop was chosen over other methodologies, such as MAPE-K.

6.3.4.7.2 Observe Functionality

Input processing uses a set of agents to ingest input data. It consists of the Data Ingestion and Normalization Functional Blocks, as described in clauses 6.3.2 and 6.3.3. This stage corresponds to the "Observe" part of the OODA control loop. The purpose of this part of the OODA control loop is to ensure that the correct data and information is gathered. The output of the input processing agents is sent to the Knowledge Processing functional block.

The OODA loop assumed that different data and information was normalized and combined. This process was first explicitly realized using the FOCALE architecture. This is why it is recommended that the input processing consist of two Functional Blocks; this adheres to the Single Responsibility Principle [i.9], a crucial software design tenet for building scalable software-intensive systems. See also "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for more background information on such principles.

6.3.4.7.3 Orient Functionality

Knowledge Processing consists of four main Functional Blocks: Knowledge Management, Context-Aware, Cognition Management, and Situational Awareness. The last three are described in clauses 6.3.5, 6.3.6 and 6.3.7. This set of Functional Blocks correspond to the "Orient" part of the OODA control loop. The purpose of this part of the OODA control loop is to ensure that input data and information is adapted to the current context and situation.

The "Orient" part of the OODA control loop consists of three types of processing: semantic annotation, knowledge inferencing, and situational processing. In each case, one or more of these four Functional Blocks may perform each of these three functions.

Semantic Annotation is a set of processes that examine the input data, and annotates it where possible to enable it to be better understood by subsequent Functional Blocks. The annotation is based on the current context and situation. This enables behaviour to be adapted based on changes to the current context and/or situation to ensure that the goals of the system and protected and maintained. Knowledge Inferencing, which is done primarily by the Cognition Management and the Situation Awareness Functional Blocks, then follows.

The contextually-aware data is then ready for inferencing operations to be performed to discover more about the nature and meaning of the data. Different types of inferencing (e.g. structural and logical) may then be performed. Inferencing produces knowledge from knowledge, and the newly produced knowledge shall also be integrated into the collective ENI knowledge base (this is not shown in Figure 6-18). The inferencing may take different forms, but is predominantly semantic in nature. For example, if a Customer object is recognized, then the annotations may describe relevant SLAs and contracted Services that are currently active. As another example, it may define applicable business and regulatory policies that shall be considered that are based on the current context. At this point, the newly ingested information has completed "pre-processing", and is ready for situational processing (the last stage of the "Orient" operation of the OODA control loop).

Situational processing analyses the "pre-processed" knowledge to determine what (if anything) has just occurred that does not align with its system goals. This could take many forms, such as determining that possible system degradation may occur, or that the system could be better optimized, or that one or more of its goals are in jeopardy. If no problems have occurred, then actions need not be taken. Otherwise, the situational processing then decides what is likely to happen, and how that affects the goals that the system is trying to achieve. This produces a set of possible alternative actions. An ENI System shall examine each set of actions, and choose the set that best meet its current set of goals.

6.3.4.7.4 Decide Functionality

The "Decide" part of the OODA control loop consists of two Functional Blocks: Model-Driven Engineering and Policy Management. These are described in clauses 6.3.8 and 6.3.9, respectively. The Model-Driven Engineering Functional Block acts as the "brains", and the Policy Management Functional Block acts as the "brawn". The purpose of this part of the OODA control loop is to decide whether any action should be taken to preserve the goals of the system.

The Model-Driven Engineering Functional Block takes its input from the "Orient" function, that compares the current state and situation of the system being managed to its desired state for that situation, and has recommended a set of corrective actions to take. Up to this point, all analysis has been done on the system model, which is a logical model that defines the manageable objects, relationships, and constraints that are being managed. The Model-Driven Engineering processes take the changes to the system model and then translates the appropriate set of actions into a form that can be implemented by the Assisted System or its Designated Entity. The result of this translation is then given to the Policy Management Functional Block.

NOTE: The present document does not define how the above translation is done, as that is implementation-specific.

The Policy Management Functional Block translates the output of the Model Driven Engineering Functional Block to a reusable set of policies. ENI Policies are made reusable in two ways:

1) storing the policy as an object in one or more models and repositories; and

2) storing the language definition of the policy in one or more repositories.

Each ENI policy may be written in imperative, declarative, and/or intent forms, and may represent recommendations or commands. The use of policies provides two important benefits. First, it defines a reusable output that can be stored and retrieved; this is critical for explanation-based auditing of what decisions the ENI System made. Second, it simplifies construction and integration issues, since any policy is an instance of an underlying language. ENI may use Domain Specific Languages (DSLs) to implement one or more forms of policies.

6.3.4.7.5 Model-Driven-Enhanced Decide Functionality

The "Decide" portion of the OODA control loop may be enhanced by using Model-Driven Engineering principles (MDE). MDE is a model-centric software engineering approach for building software systems that can be dynamically modified at runtime. It treats models as first-class artifacts, and does design and analysis of models in place of code. MDE is particularly effective when the use of software design patterns is maximized [i.5] and [i.10].

The addition of MDE mechanisms should be used to enhance the Decide portion of the OODA loop. This is especially useful when the context and situation changes. The use of MDE in these circumstances enables the functionality of this portion of the control loop to be dynamically changed at runtime, thereby facilitating its adaptation to the changes in the context and/or situation. More specifically, this enhancement may enable the Decide portion to generate better and more complete plans.

6.3.4.7.6 Act Functionality

The "Act" part of the OODA control loop consists of two Functional Blocks: Denormalization and Output Generation. These are described in clauses 6.3.10 and 6.3.11, respectively. The functionality of the Denormalization and Output Generation Functional Blocks may be combined into a single Functional Block if desired. The purpose of this part of the OODA control loop is to take one or more actions to preserve the goals of the system.

The input to the Denormalization Functional Block is the set of Policies defined by the Policy Management Functional Block. These policies are in a normalized form for ENI; the Denormalization Functional Block translates them to a denormalized form to enable external Functional Blocks to understand each policy. This may be done using a set of output processing agents. Each agent is responsible for de-normalizing the data (e.g. translating the knowledge into a specific language and format for a given set of devices). For example, these agents could translate an ENI policy to an alternative form; one example is translating an ENI declarative policy to Open Stack Congress. This is then sent to the Output Generation Functional Block.

The Output Generation Functional Block packages the translated ENI Policy to a form appropriate for the external entity that is using the ENI Policy. Output processing may use a set of agents to translate the ENI Policy into specific forms (e.g. per vendor and per device, or into a payload for a specific protocol that uses a particular encoding format).

6.3.4.7.7 Model-Driven-Enhanced Act Functionality

The addition of MDE mechanisms should be used to enhance the Act portion of the OODA loop. Similarly to enhancing the Decide portion of the OODA loop with MDE (see clause 6.3.4.7.5), this enables the functionality of this portion of the control loop to be dynamically changed at runtime, thereby facilitating its adaptation to the changes in the context and/or situation. More specifically, this enhancement may enable the Act portion to generate more accurate recommendations and commands.

6.3.4.7.8 Learning-Enhanced OODA

FOCALE [4] added both learning and reasoning to the OODA control loop. Learning included both machine learning in the traditional AI sense as well as learning using reasoning mechanisms, which is covered in clause 6.3.4.7.9.

Supervised learning (see ETSI GR ENI 018 [i.37]) should be used when one or more datasets that have labelled input and output values. Supervised learning algorithms are ideal for classification and regression tasks. Classification algorithms are used to predict the category that a new datum belongs to based on one or more independent variables. In contrast, regression algorithms predict an associated numerical value for the input datum based on previously observed data.

Unsupervised learning (see ETSI GR ENI 018 [i.37]) should be used when there is a large amount of data that do not have labels, and the task is to determine the structure of the data. Clustering is a multivariate statistical procedure that collects data containing information about a sample of objects and then arranges the objects into groups, where objects in the same group are more similar to each other than to objects in other groups. Clustering identifies commonalities in the objects in each group, which can also be used to detect anomalous data that do not fit into any group.

Reinforcement learning (see ETSI GR ENI 018 [i.37]) should be used when there is no data, or the dataset is inadequate, and the task is to learn what action to take in a particular situation when interacting with a new entity. This type of learning should also be used when the only way to collect information about the entity is to interact with it. More specifically, reinforcement learning shall interact with the entity to first, negotiate its capabilities, and then discover how to exchange data and commands through learning how to react with the entity.

These and other types of learning should be used to enable the OODA-based control loops used in the ENI System to operate more efficiently. In essence, learning provides memory to all phases of the OODA control loop.

6.3.4.7.9 Reasoning-Enhanced OODA

Reasoning is the process of making a conclusion or decision by analysing the facts available in a situation. There are two types of reasoning, inductive and deductive.

Inductive reasoning involves drawing conclusions from facts, using logic. Inductive reasoning involves Bayesian updating. A conclusion can seem to be true at one point until further evidence emerges and a hypothesis needs to be adjusted. Bayesian updating modifies the probability of a hypothesis being true as new evidence is supplied. This is why formal logic plays an important role in the decision making processes used in the ENI System.

Thus, induction can thus be strong or weak. If an inductive argument is strong, the truth of the premise means that the probability that the conclusion is correct is stronger. Similarly, if an inductive argument is weak, then the probability that the conclusion is correct is weak. Note that in particular, a conclusion is either strong or weak, and shall not be considered right or wrong. In contrast, deductive reasoning involves deciding whether an inference is correct or incorrect from a given set of premises. In deduction, the idea is to determine whether link between the set of premises and the conclusion is always true or not. More formally, a hypothesis is formed, and then evidence is collected to support it. If observations support its truth, the hypothesis is confirmed.

Deductive reasoning shall be used for decision-making in an ENI System when enough facts, axioms, and/or knowledge is present to support inferencing. Inductive reasoning should be used for decision-making in an ENI System when it is required to prove a general principle, not a specific fact or inference.

6.3.5 Context-Aware Management Functional Block

6.3.5.1 Introduction

This clause describes the motivation for using context-aware behaviour, the effect it has on the System Architecture, and the benefits that it provides. See clause 6.3.7 or a description of situation awareness, and specifically clause 6.3.7.5 for a comparison between this Functional Block and the situation awareness Functional Block.

6.3.5.2 Motivation

Most management systems today are collections of functionality that operate as silos. They are characterized by:

- Static, end-to-end process-based management.
- Little or no understanding of context, and hence, no ability to provide behaviour and services personalized to a given context.
- No ability to change behaviour due to changes in context.

Context may produce a higher intrinsic value for data versus raw data, and hence, make the generation of information easier. Context-awareness assumes that either the data and/or the associated metadata deliver additional information about the characteristics and behaviour of the environment that an ENI System interacts with. Context ensures that an ENI System is focused on tasks that the operator defines as important. The additional knowledge provided by context may offer a greater level of reliability and usefulness, both for the information and knowledge gathered by an ENI System as well as for the actions taken by the ENI System.

6.3.5.3 Function of the Context-Aware Management Functional Block

6.3.5.3.1 Introduction

Context-awareness enables a system to gather information about itself and its environment [i.2], [i.3] and [i.15]. This enables the system to provide personalized and customized services and resources corresponding to that context. More importantly, it enables the system to *adapt its behaviour according to changes in context*.

Context-awareness enables diverse data and information to be more easily correlated, and hence, integrated, since context acts as a unifying filter. As such, identifying contextual information is critical for understanding both ingested data and information as well as how data and information, as well as existing knowledge and wisdom, can be affected.

The contextual history of a user, a user application, or a device, as well as its prior interactions with an ENI System (including, for example, session state), may be useful for driving policy decisions regarding the current and future interaction between an ENI System and that entity, including decisions made by the ENI System that affect that entity. For example, past behaviour can be used to more quickly arrive at a decision. Alternatively, historical information can be used to flag anomalies that need further action to resolve.

Contextual data may be categorized in terms of data and information that an ENI System interacts with. This includes a user, the applications of a user, the time and location of objects that the user interacts with, and different types of relationships that exist among entities that an ENI System interacts with as well as between an ENI System and those entities. Examples include:

- Personal and group information for a user (e.g. contact information, roles played by the user, and profile and preference information).
- Location of the user and any devices that the user is using or interacting with (e.g. geo-code, the centroid of a surrounding polygon).
- Characteristics and behaviour of a device (e.g. type of access provided (if any), MAC and IP addresses, and features such as forward, routing, encryption).
- Characteristics and behaviour of any applications used by a user.
- Types of relationships that exist between an ENI System and entities that use or interact with an ENI System, as well as the nature of those relationships (e.g. is it temporally sensitive or static).
- Types of relationships that exist between entities that an ENI System interacts with.

The Context of an Entity is a collection of measured and inferred knowledge that describe the state and environment in which an Entity exists or has existed [i.2]. In particular, this definition emphasizes two types of knowledge - facts (which can be measured) and *inferred* data, which results from machine learning and reasoning processes applied to past and current context. It also includes context history, so that current decisions based on context may benefit from past decisions, as well as observation of how the environment has changed.

6.3.5.3.2 Modelling and Representation of Context Awareness

Different aspects of the DEN-ng [i.1] context model are described [i.2] and [i.3]. This is a robust information model, and shall serve as the foundation of the context information model used for the ENI System Architecture. It is shown in Figure 6-19.

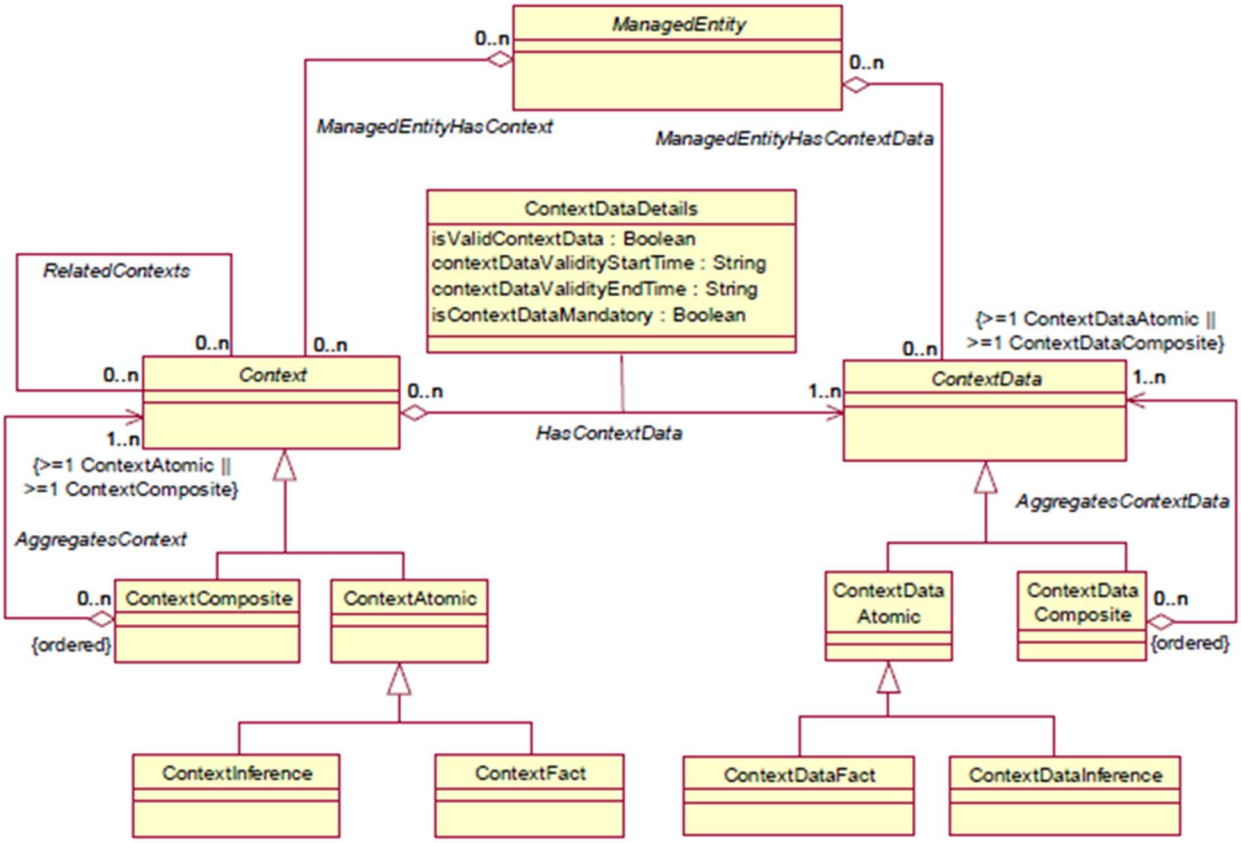

Figure 6-19: Foundation of the DEN-ng Context Model

The DEN-ng model represents the Context of a ManagedEntity as a collection of data, information, and knowledge that result from collecting measurements of and reasoning about that ManagedEntity. This is represented by the ManagedEntityHasContext aggregation. Not all ManagedEntities have associated Context, so the multiplicity of this relationship is 0..n - 0..n. This enables both Context and the ManagedEntity to be defined independent of whether one is associated with the other.

Context is defined as an extensible set of building blocks that can be combined as required. Hence, two classes are defined, called Context and ContextData, that represent the concepts of whole (or assembled) context and partial (or component pieces of) context, respectively. They each a set of subclasses, and each may be associated with different concepts (e.g. location, time, and connectivity). This is a unique model in this field, as it enables the component pieces of context (i.e. instances of the ContextData class) without affecting the overall context. This is how the context model is adjusted to reflect changing user needs, business goals, and environmental conditions. This approach avoids a common weakness of traditional context models, which try to represent context using an unorganized smattering of objects that each contain some interesting attributes, but are not cohesive (see ETSI GR ENI 016 [i.35]) in nature. It also avoids the temptation to define types of context as rigid classes that always have to be used. Instead, this model enables ContextData to model the same concept (e.g. Location or Connectivity) in different ways to match the specific needs of the application using it.

The ContextData class represents the context of one or more ManagedEntity classes that are part of a larger context. The Context class represents the context of the ManagedEntity that the management system is interested in, and consists of its own knowledge plus a set of aggregated ContextData objects. Note that the HasContextData aggregation has a multiplicity of 0..n - 1..n. This means that a Context has one or more ContextData objects, but each ContextData object can be instantiated and updated before it is aggregated by a particular Context object. The advantages of this approach are:

1) it encourages the modelling of smaller "context building blocks" (i.e. instances of the ContextData object) that can be reused and combined to form larger contexts (i.e. instances of the Context object);

2) the information model can be used to define different aspects of context as either ContextData or as Context; and

3) different aspects of context can be modelled in the varying detail that they require whilst still being able to assemble them into a coherent whole.

The composite pattern [7] and [i.5] is applied to both the Context as well as the ContextData objects, enabling the construction of extensible hierarchies of different aspects of aggregate contexts. The ContextDataAtomic class models ContextData as a single, stand-alone object, while the ContextDataComposite class models objects that are composite in nature (e.g. made up of multiple distinct ContextData objects that can each be separately managed). ContextDataComposite objects function as containers that can contain ContextDataComposite and/or ContextDataAtomic objects, similar to folders and files in a directory. Both Context and ContextData objects can represent facts and inferences.

The ContextDataDetails association class defines the semantics of how each particular type of ContextData object is aggregated by the Context object. The four attributes of ContextDataDetails are common to all types of aggregations of ContextData; this enables different applications to have a common "control point" for processing context information. This is explained more in clause 6.3.5.3.3.

6.3.5.3.3 Processing Contextual Updates

The Policy Pattern [i.1] is used in DEN-ng as well as in the MEF Core Model [7] and MEF Policy Model [8]. This pattern is represented in Figure 6-20, and shall be used in the ENI System Architecture due to its flexibility and extensibility.

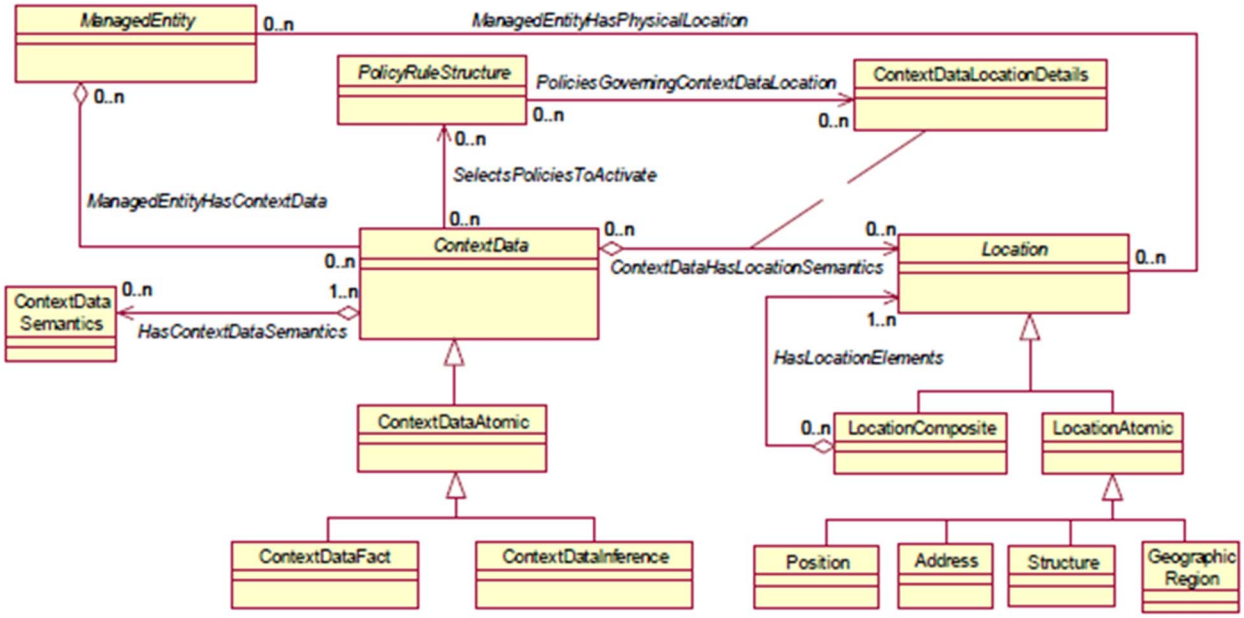

Figure 6-20: Controlling the Semantics of Location ContextData using Policies

The Policy Pattern enables different applications to represent their specific needs by constructing an association between the superclass of all Policies [8] and the association class of a particular ManagedEntity that the set of Policies should control. This is shown in Figure 6-20 as the PoliciesGoverningContextDataLocation association. Policy management of context is thus consistently done by defining application-specific policies and associating them with the appropriate context association classes.

Figure 6-20 also shows how another model (in this case, Location) may be used in the context model. In Figure 6-20, the ContextDataHasLocationSemantics aggregation is used to associate the top part of the DEN-ng Location model with the ContextData class. Since all relationships are model-independent, this enables any other Location model to be used in place of the DEN-ng Location model. The above model is far superior to the common practice of defining a set of attributes in a context class to represent location. First, there are many different types of locations that are of interest to a managed networking environment, such as the location of a card in a networking device, the GPS coordinates of a Customer Premise Equipment, the physical or logical address to deliver an Order to, and so forth. More generally, location is more than just latitude and longitude, but also includes height, proximity, co-location with other entities, orientation, and other factors. How can a single set of *static* attributes represent all of these different concepts?

Second, according to MDB principles, this maximizes reusability in an extensible and technologically-neutral manner. The same concepts for different types of locations can be reused, partially or in their entirety.

Finally, this enables different parts of the Location model to be used for different purposes; it also enables the same parts of the Location model to have different semantics attached to them. The physical characteristics of Location are defined in these Location classes, while the semantics of the Location are defined in the ContextData classes.Put another way, this approach allows the semantics of location to *change as a function of context*.

6.3.5.4 Operation of the Context-Aware Management Functional Block

A good overview of designing for context-awareness is provided in [i.2], [i.3] and [i.15].

Figures 6-7, 6-8 and 6-9 shows the Context-Aware Management Functional Block connected to a Semantic Bus (see clause 6.3.4.4.4 for a description of the functionality of a Semantic Bus). The functional block diagram shown in these figures should not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of context-awareness. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

Figure 6-21 shows some of the roles that can be used in a context-aware system. The overall (aggregate) context is used to derive one or more of the five exemplary roles shown, which in turn are used to define behaviour appropriate for that context. The five roles are described in Figure 6-21.

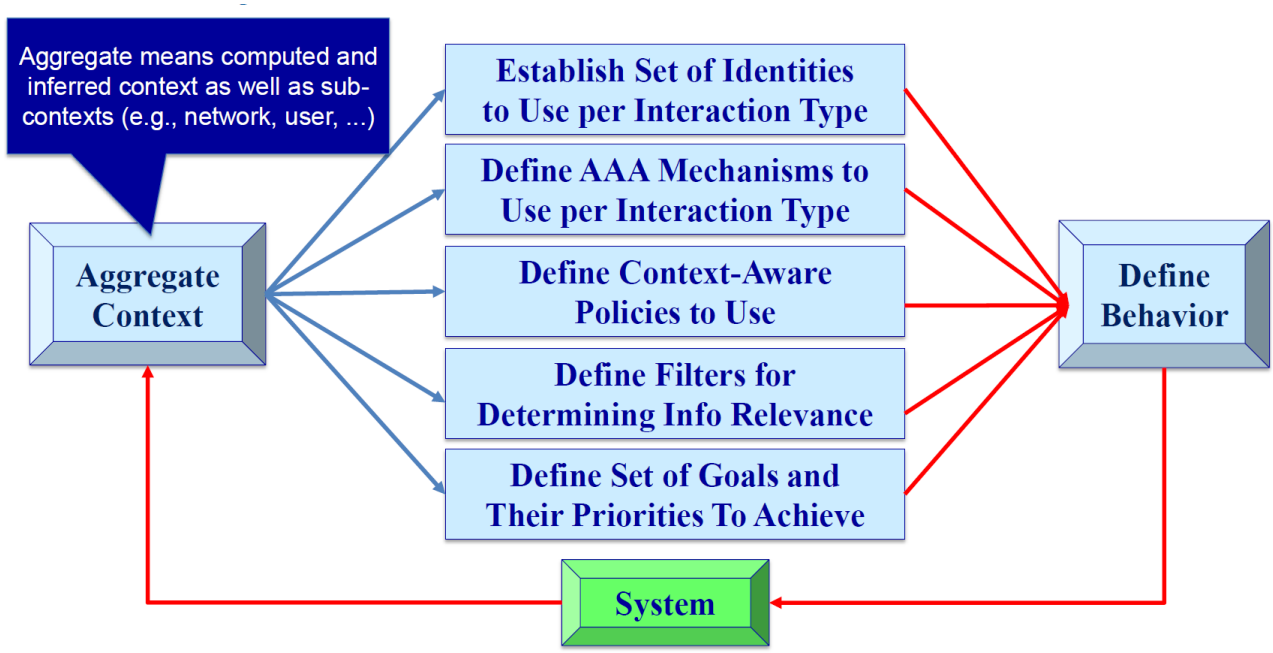

Figure 6-21: Contextual Roles

- **Identities per Interaction Type.** This role defines a set of processes that define the characteristics and behaviour of a particular set of identities for objects in this context. Interaction type includes the set of entities in the context that this object interacts with, along with how the interaction is done (e.g. which set of protocols are used). For example, a passport is generally required to authorize travel to a foreign country, but is likely not usable when a user logs onto their personal computer.

- **AAA Mechanisms per Interaction Type.** This role defines the applicable set of authentication, authorization, and accounting interactions for this particular context. It may include other related operations, such as auditing.

- **Context-Aware Policies to Use.** This role defines a set of policies that shall be produced by the Policy Management Functional Block in response to changes in the current state deviating from the desired state for this context.

- **Filters for Determining Relevance.** This role defines a set of filters that determine the relative relevance of ingested data and information for this particular context. This enables the system to:

  1) adapt which data are gathered as context changes; and

  2) modify its processing according to the contextual changes.

- **Set of Goals to Achieve.** This role defines a set of goals to achieve for this particular context. For example, if the ratio of the users of the Platinum, Gold, Silver, and Bronze service classes change, then the ENI System may need to generate a new set of policies that govern the behaviour of each service class in order to maximize revenue.

Figure 6-22 shows an exemplary set of operations that occur when a context is analysed.

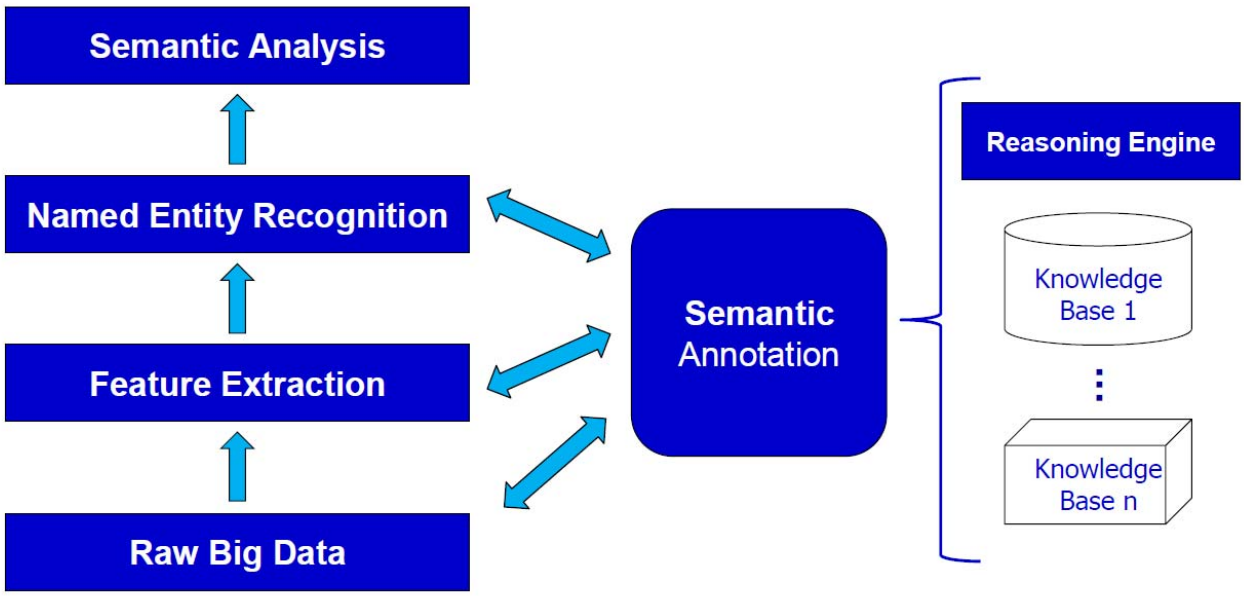

Figure 6-22: Context-Based Reasoning

Context may be modelled as Big Data, since there is an abundance of Big Data, and the critical factor is extracting value from Big Data. The three operations above Big Data in Figure 6-22 describe a set of increasingly specific operations that can be used to semantically annotate information.

A "feature" may be defined as an important element that helps describe and aid in the understanding of an object. For example, edges and corners are points of interest in an image. Feature extraction reduces the number of resources required to describe data. Reducing the number of features is important in analysis and reasoning, since if there are too many features, it could cause overfitting in training.

Named Entity Recognition identities and maps entities that have specific names into models. For example, the sentence "John worked at ETSI in the 1990s" could be analysed, producing three named entities: "John", "ETSI", and "1990s". This could be annotated as "John$_{[person]}$ worked at ETSI$_{[organization]}$ in the 1990s$_{[time_period]}$".

Semantic Analysis analyses the information for specific semantic concepts, searching on those concepts, and then adding additional semantic relationships to enrich the information and provide more specific meaning. From a linguistic perspective, this analyses text and finds sets of syntactic structures that are related to each other. This may be represented as a graph, or network, of related words, phrases, and other elements of a sentence. From a machine learning perspective, this computes metrics such as semantic similarity (i.e. the meaning of an object compared to the meaning of other objects, where the comparison is done using synonymy, antonymy, hyponymy, hypernymy, and other types of relationships). This is a practical and more computationally tractable approach than "absolute understanding", since the latter requires a rigorous world model, which is np-complete.

Semantic Annotation is the process of providing annotations (either as metadata and/or in a specialized markup) to add meaning to a given object. An example of semantic annotation was given in the named entity recognition description.

6.3.6 Cognition Management Functional Block

6.3.6.1 Introduction

The Cognition Management Functional Block is a collection of Functional Blocks that are responsible for operating the ENI Cognition Model (see clause 6.3.6.3.5) to understand ingested data and information in order to produce new data, information, and knowledge. This means that the scope of the cognitive framework is conceptually operating above the scope of both the infrastructure and the other Functional Blocks of the ENI System.

6.3.6.2 Motivation

The purpose of the Cognition Management Functional Block is to enable the ENI System to understand ingested data and information, as well as the context and situation that defines how those data and information were produced. Once that understanding is achieved, the Cognition Management Functional Block then provides the following functions:

- change existing knowledge and/or add new knowledge corresponding to those data and information;
- perform inferences about the ingested information and data to generate new knowledge;
- use raw data, inferences, and/or historical data to understand what is happening in a particular context and/or situation, why the data were generated, and which entities could be affected; and
- determine if any new actions should be taken to ensure that the goals and objectives of the system will be met.

In each of the four functions above, the Cognition Management Functional Block uses existing knowledge to validate and generate new knowledge. This means that new knowledge may be added, and in some cases, existing knowledge may be changed. Hence, the ENI System uses a *dynamically changing set of repositories* (as opposed to other management systems, which typically use repositories that use *fixed content*). A cognition framework uses multiple diverse processes and technologies, including linguistics, computer science, AI, formal logic, neuroscience, psychology, and philosophy, along with others, to analyse existing knowledge and synthesize new knowledge.

In addition, the Cognition Management Functional Block will not *take* actions; it will just determine what does not agree with its Cognition Model and annotate accordingly. The actual actions to be taken will be computed by the Situation Awareness Functional Block (see clause 6.3.7) and then implemented by the MDE and Policy Management Functional Blocks (see clauses 6.3.8 and 6.3.9). The actions are embedded in a set of ENI Policies, and then sent through the Denormalization and Output Generation Functional Block. The transformed set of ENI Policies is then sent to the API Broker, which puts the set of ENI Policies in an appropriate API readable by the receiving entity (i.e. the Policy Target, see ETSI GS ENI 019 [9]).

6.3.6.3 Function of the Cognition Management Functional Block

6.3.6.3.1 Introduction (informative)

Cognition seeks to understand concepts in a way similar to how the human brain understands concepts. This is done by using a set of specialized data structures and computational procedures that mimic how the human brain operates. Connectionist theories use this principle to define artificial neural networks. Other approaches, such as those involving formal logic, Bayesian models, and deep learning, provide different algorithms, but are still based on the above premise.

Cognition can be used to process new data and information, along with new inferences, and compare those to previously stored knowledge. The function of this Functional Block is to process and understand goals so that it can institute behaviour that protects and meets those goals. This is done using knowledge and inferencing to explain why input data occurred and how to adapt to it. Intelligent agents [i.7] and [i.8] are examples of entities that exhibit goal-directed behaviour. Other examples are cognitive architectures that solve problems by creating their own sub-goals to solve a problem; such cognitive architectures also learn from their experience.

Any entity that exhibits cognition will exhibit at least the following three functions:

1) interfaces that interact with the environment providing stimuli;
2) processing that can analyse and manipulate data, information, and knowledge; and
3) memories that hold data, information, and knowledge.

These three functions provide cognitive control and cognitive capabilities, which differentiate a cognitive architecture from other architectures.

Cognitive control includes reflexive and habitual behaviour that respond to long-term intentions. Cognitive capabilities include functions that reflect processing as done in the human brain, such as perception, reasoning, learning and planning. Critically, a system that uses cognition is able to explain why it acted a certain way in response to stimuli, and more importantly, can learn whether that action was incorrect and, if correct, whether it was optimal.

There are two main approaches to building cognitive systems: symbolic and connectionist. Hybrid architectures, which combine these approaches, are also starting to be researched.

6.3.6.3.2 The Symbolic Approach (informative)

This approach (also called computationalism) views cognition as a set of computational processes that act on a set of structures that represent abstract (mental) representations of the world that can be manipulated by symbols. There is a clear separation of cognition into low-level (e.g. sensorimotor) and high-level (e.g. planning and reasoning) processes. This approach builds models of high-level cognition that resemble the structure of the brain (as opposed to connectionist models, which build models that resemble neurological structures). The symbolic models obey rules for processing data and information, as well as for interacting with other symbolic models. This leads to domain-specific symbolic sub-systems for processing different types of data and information (e.g. language processing, reasoning, and planning).

Each of the different domain-specific sub-systems are interconnected, forming a modular architecture. Actions in this architecture can be modelled as operations acting on these sub-systems that collectively form a modular program. The representations and actions taken in the cognitive architecture correspond to the real-world objects and their behaviour. In other words, the symbolic abstractions used in the program are kept in sync with the external world. For example, in robotic systems, the sensorimotor processing is responsible for this synchronization.

An AI programming language can be used to construct a cognitive model. This is ideal for knowledge-based problem solving and learning. For example, the AI programming language could be used to construct a search through a problem space, where a problem space defines a set of states and operators that manipulate the states. The solving of a problem is achieved by traversing from an initial state to a final state using a set of operators. An example of such an architecture is provided in [i.10]. In this architecture, a set of IF-THEN (imperative) rules are used to operate on the symbolic representations. When the IF clause is satisfied, the set of actions in the THEN clause are executed. A high-level functional architecture of ACT-R, one example of a symbolic cognitive architecture, is shown in Figure 6-23.

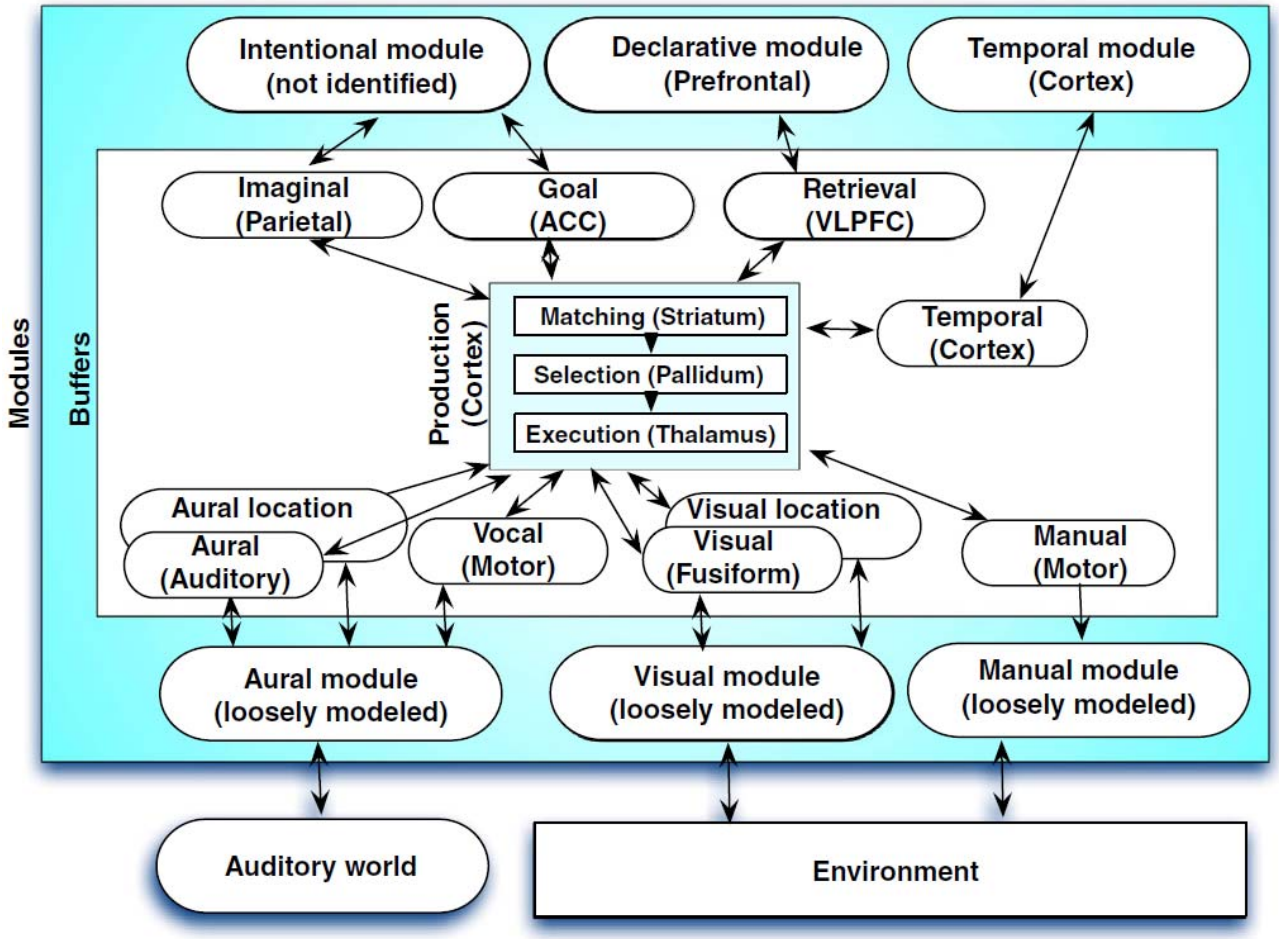

Figure 6-23: The ACT-R Cognitive Architecture

The main modules of ACT-R are:

1) a visual module for identifying objects in the visual field;

2) a goal module for keeping track of current goals and intentions;

3) a declarative module for retrieving information from memory;

4) a manual module for controlling the hands; and

5) a production system for taking actions and coordinating the communication and performance of these modules.

Actions in each of these modules take time, although they work concurrently. The time used within models is based on human performance. This time can be provided as a real-time trace. The default cycle for taking an action in the production system is 50 msec.

Information is passed between the modules and the central production system through buffers, which hold a limited amount of information (this is motivated from psychological and neuroscientific studies). The central production system matches the content of the buffers against the IF portions of its rules, and selects a single rule for execution. The selection mechanism is based on the computation of expected utility of a rule for the current goal and the input. The expected utility is learned from experience using Bayesian processing. A rule can have multiple actions, including updating declarative or goal memory, modifying the value of a buffer, and issuing a command.

6.3.6.3.3 The Connectionist Approach (informative)

This approach assumes that all cognitive processes are the same, and are derived from neural activation dynamics. In contrast to the symbolic approach, connectionism argues that mental representations are not structured as explicit models, but rather, are implicitly encoded in the activation values of neurons. This leads to the parallel processing of many simple modules that are connected as a network. Information is stored in the form of weights between connections.

Knowledge is not provided explicitly to the system, but rather, is learned by the system through the processing of training samples. Learning algorithms extract statistical information from sample pairs of input and output values, and the network adapts its connection weights to approximate the training data. Thus, task execution by connectionist approaches requires suitable training data, and depends on both the network structure methodology and the learning algorithm(s) used. A simplistic Recurrent Neural Network (RNN) is shown in Figure 6-24.

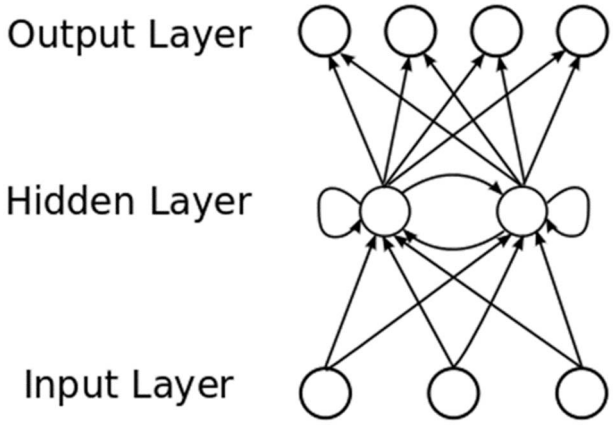

Figure 6-24: A Simplified Cognitive Processing Architecture using Neural Networks

RNNs use looping of the hidden layers back to themselves, which enables them to accept variable length sequences of inputs. RNNs provide a way to process data where time and order are important. For example, with textual data, the ordering of words is important. Changing the order or words can alter the meaning of a sentence. In contrast, in simple feed forward networks, the hidden layer only has access to the current input. It has no "memory" of any other input that was already processed. An RNN, by contrast, is able to "loop" over its inputs and see what has come before. This provides context for processing words that come later in a sentence.

6.3.6.3.4 Cognitive System

A cognitive system is a system capable of independently developing strategies for and solving human tasks. A cognitive system is both context- and situation-aware. As such, a cognitive system should understand, identify, and extract contextual elements such as meaning, syntax, time, location, appropriate domain, regulations, user's profile, process, task and goal. A cognitive system may draw on multiple sources of information, including both structured and unstructured digital information, as well as sensory inputs (visual, gestural, auditory, or sensor-provided).

Typically, a cognitive system is built to perform comprehension, learning, and reasoning by mimicking those tasks as done by a human. More formally, cognition is defined as the process of acquiring and understanding data and information and producing new data, information, and knowledge. Cognitive systems may utilize artificial intelligence (AI) methods such as machine learning, neural networks and deep learning as well as other methods such as multimodal perception and declarative memory (i.e. the encoding, storage, and retrieval of facts and events).

The individual functional blocks of a cognitive system, as well as multiple cognitive systems, shall be able to collaborate on a set of tasks. Once the set of tasks has been completed, the collective may disband.

An individual system shall be capable of taking over mission-critical functions on its own in case coordination or communication with other collaborating entities is not functioning. The collective should furthermore optimally and efficiently complete its tasks in such a situation.

A cognitive system should be able to adapt as information and context changes. A cognitive system should also be able to adapt as goals and requirements evolve.

A cognitive system need not be explicitly programmed to do its tasks. Rather, a cognitive system shall learn and reason from their interactions with humans and from their experiences with their environment. Cognitive systems may use deterministic mechanisms; however, cognitive systems should be mostly probabilistic in nature. Cognitive systems should generate hypotheses, reasoned arguments and recommendations. Cognitive systems should be able to generate explanations of their reasoning processes.

A cognitive system shall be able to *reason* about what actions to take, even if a situation that it encounters has not been anticipated. It shall learn from its experience to improve its performance. It should be able to examine its own capabilities and prioritize the use of its services and resources, and if necessary, explain what it did and accept external commands to perform necessary actions.

Cognitive systems should be used to augment human decision-making and action processes. Cognitive systems are not meant to replace humans, but rather, enhance them. An analogy is how Garry Kasparov, the one-time world chess champion who lost to IBM Deep Blue in 1997, changed from being "against" computers to using computers. He competed in in "freestyle" chess leagues, where players were able to compete in chess tournaments with the assistance of computers. Kasparaov wrote: "*Teams of human plus machine dominated even the strongest computers. Human strategic guidance combined with the tactical acuity of a computer was overwhelming. We [people] could concentrate on strategic planning instead of spending so much time on calculations. Human creativity was even more paramount under these conditions*".

6.3.6.3.5 Cognition Model

6.3.6.3.5.1 Introduction

A cognition model is a computer model of how cognitive processes, such as comprehension, action, and prediction, are performed and influence decisions.

For the purposes of ENI, a cognition model mimics human cognition. More specifically, the perception portion provides the notion of classifying data into pre-defined representations that are understood and relevant to the current situation; memory is used to increase comprehension of the situation; actions are judged by how effectively they perform to support the situation. These concepts are supported in cognitive psychology, where Minsky modeled this using three interacting layers, called reactive (or subconscious), deliberative, and reflective [i.43].

Reactive processes shall take immediate responses based upon the reception of an appropriate external stimulus. In humans, these processes correspond to instinctual and learned behaviours. In ENI, such processes may have no understanding of the semantics of the external stimulus; rather, they shall respond with some combination of pre-defined and learned reactions.

Deliberative processes shall receive data from and can send recommendations and/or commands to the reactive processes. Deliberative processes need not interact directly with the external world. In humans, this part of the brain is responsible for our ability to achieve more complex goals by applying short- and long-term memory in order to create and carry out more elaborate plans. In ENI, these processes shall use short- and long-term memory. Furthermore, these processes shall accumulate and generalize knowledge from experience, and combine that with what is learned from other people and systems.

Reflective processes shall supervise the interaction between the deliberative and reactive processes. In humans, these processes enable the brain to reformulate and reframe its interpretation of the situation in a way that may lead to more creative and effective strategies. In ENI, these processes should consider what predictions turned out wrong, along with what obstacles and constraints were encountered, in order to prevent sub-optimal performance from occurring again. In ENI, these processes should also include self-reflection, which analyses how well the actions that were taken solved the problem at hand.

6.3.6.3.5.2 ENI Cognition Model

A cognition model is used to both embed intelligence as well as to enable a system to learn and reason while taking long-term intentions into account. The conceptual relationship between perception, comprehension, and action and the three types of cognitive processing are shown in Table 6-1.

Table 6-1: Types of Cognition and their Responsibilities

Type of Cognition	Perception	Comprehension	Action
Reactive	Prioritise event handling	Not needed	Fixed action taken
Deliberative	Analyse stimuli and related short-term events	Consider different possibilities and rank **according to current needs**	Take appropriate action based on planning outcome to **satisfy needs**
Reflective	Analyze stimuli and related short- and long-term events	Consider different possibilities and rank **according to goals**	Take appropriate action based on planning outcome to **satisfy goals**

Reactive cognition assumes that a pre-defined set of actions are associated with each stimulus. As will be seen, even though an action is taken, learning and reasoning functions of the ENI System will analyse the action(s) taken and evaluate their effectiveness, both to satisfy current needs as well as the long-term goals of the system. Conceptually, this is a "shortcut" through the processing steps that are required for deliberative and reflective processing.

Deliberative processing compares the input stimuli to short-term events, and their responses, which are related to it. Planning is then initiated to consider different types of responses and their effect on the current needs of the system being managed. These possibilities are then ranked according to how well each satisfies the current needs of the system. The action that ranks the best is then executed. However, the set of possible responses, along with their stimuli, are recorded for further analysis. Learning and reasoning functions will then compare this and other actions to see if the collected set of actions were the optimal responses that could be taken.

Reflective processing compares the input stimuli to short- and long-term events, and their responses, which are related to it. Planning is then initiated to consider different types of responses and their effect on the goals (both short- and long-term) of the system being managed. These possibilities are then ranked according to how well each satisfies the goals of the system being managed. The action that ranks the best is then executed. However, the set of possible responses, along with their stimuli, are recorded for further analysis. Learning and reasoning functions will then compare this and other actions to see if the collected set of actions were the optimal responses that could be taken. Conceptually, reflective processing is a type of meta-management, where the current needs of the system being managed are weighed against the goals that are to be achieved.

A simplified functional architecture that uses a Cognitive Model is shown in Figure 6-25.

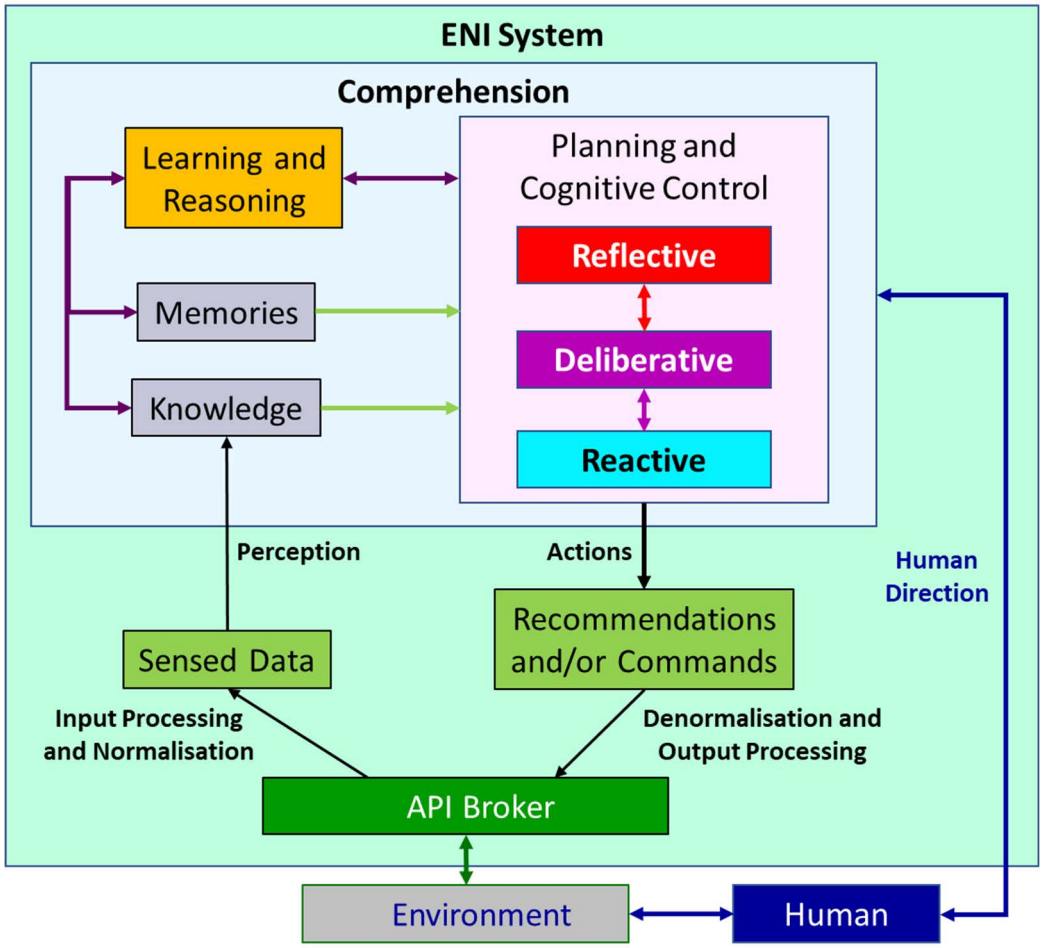

Figure 6-25: A Simplified Functional Architecture using a Cognitive Model

Figure 6-25 starts with the "outer" ENI closed control loop (environment → API Broker → Sensed Data → Processing → Actions → API Broker → Environment). The Processing portion of the ENI System (i.e. the six Inner Functional Blocks) are redrawn to emphasize the cognitive processing that they collectively provide. This provides comprehension of the perceived information. The actions are then sent to the API Broker, which delivers them back to the system being managed by ENI.

Planning is defined as the task of finding a procedural course of action for a declaratively described system to reach its goals while optimizing overall performance measures. In an ENI System, additional constraints, such as business rules and most importantly, the set of goals to be achieved, determine the current set of goals and what specific measures to optimize (e.g. time to solve, resources used, or a combination of these and other metrics). Finally, planning in an ENI System also includes reviewing the courses of action that are available and predicting their expected (and unexpected) results (e.g. "think before act").

The Cognition Management Functional Block shall use the ENI Cognition Model for planning and scheduling functions to be performed in order to achieve a goal. Planning determines what steps to take, while scheduling decides when to carry out a certain step. The planning process may use a finite state machine for planning. In this approach, the finite state machine consists of two main states (an initial state and a goal state) that are connected through a set of one or more actions defined as intermediate states.

The planning performed in the Cognition Management Model is optimized according to current context and situation. For example, this may mean that the Finite State Machine(s) being used are updated with state transitions leading to a new state. The Cognition Management Functional Block then determines if this new state has achieved its goal state, or if not, is the new state closer or farther away from achieving its goal state. This information, along with an optionally defined schedule, is then sent to the Situation Awareness Functional Block.

The mechanisms used in planning should know nothing about the specific problem that they are solving. This makes the planning portion of the Cognition Management Functional Block independent of the goals that it is trying to achieve.

Both AI- and non-AI based planners treat planning as a search problem. The program will traverse a potentially large search space and find a plan that starts at the initial state and generates a final end state that contains the desired goals.

The Cognition Management Functional Block's planner uses short-term and long-term memories (part of the Knowledge Management Functional Block), knowledge from the Context-Aware and Situation-Aware Functional Blocks (represented as "Knowledge" in Figure 6-25), and Learning and Reasoning (part of the Cognition Management Functional Block). The Planning and Cognitive Control Functional Block also contains part of the Cognition Management Functional Block. The actions (i.e. recommendations and/or commands) are produced by the MDE and Policy Management Functional Blocks.

NOTE: Add a new section in clause 6 that describes the operation of ENI as a whole, and the role played by the Cognition Management system.

6.3.6.4 Operation of the Cognition Management Functional Block

NOTE: This is for further study in Release 3 (see clause 9), as this is dependent on an agreed upon Cognition Model.

6.3.7 Situational Awareness Functional Block

6.3.7.1 Introduction

This clause describes the motivation for using situational awareness, the effect it has on the System Architecture, and the benefits that it provides. See clause 6.3.5 for a description of the context-awareness Functional Block, and clause 6.3.7.5 for a comparison of this Functional Block to the context-awareness Functional Block.

Situation awareness enables the system to understand what has just happened, what is likely to happen, and how both may affect the goals that the system is trying to achieve. This implies the ability to understand how and why the current situation evolves. ENI shall observe the evolving of different situations, examining them for patterns within each situation and between different situations. Such knowledge shall be stored in the knowledge base of ENI. As such, identifying changes in both the current situation as well as possible future situations are critical for understanding how the environment is changing, and how those changes affect the goals that ENI is trying to achieve or maintain.

For example, security situation awareness could include being aware of the scope and impact of the attack, correlating that with the behaviour of the adversary, so that effective counter-measures can be implemented.

6.3.7.2 Motivation

Networks are fundamentally heterogeneous in nature. Current as well as legacy devices have different software, hardware, and use different protocols. Data may be represented in multiple ways. Hence, there is a need for a common and scalable mechanism to abstract this heterogeneity so that their functionality can be represented in a normalized manner. This would enable common approaches to be developed for enhancing interoperability between heterogeneous systems, letting developers create new applications that use these common characteristics and behaviour.

6.3.7.3 Function of Situational Awareness

The working definition of situation awareness for ENI is:

> *"The perception of data and behaviour that pertain to the relevant circumstances and/or conditions of a system or process ("the situation"), the comprehension of the meaning and significance of these data and behaviours, and how processes, actions, and new situations inferred from these data and processes are likely to evolve in the near future to enable more accurate and fruitful decision-making".*

It consists of five actions: gathering data (perception), understanding the significance of the gathered data (through both facts and inferences), determining what to do (if anything) in response to a given event, making a decision (or set of decisions), and performing those actions. It enables the application of context and policies to a particular situation, and can use inference as well as historical data to understand what is happening at a particular context, why, and what (if anything) should be done in response.

6.3.7.4 Operation of the Situational Awareness Functional Block

6.3.7.4.1 Introduction

A situation shall be determined by the analysis of data and behaviour. The evolution of future situations is a function of understanding the particular context, the factors determining the evolution of that context, and inferring what the future situation will be based on the past and current data and behaviour. Semantics play an important role in understanding the significance and cause of data and behaviour, and shall be used to understand the underlying meaning of data and information that have been ingested.

A situation reflects an entity's contextual view of a collection of data and processes at a particular instance in time. Shared situational awareness is therefore a consensus view of a number of individual views that each describes the same situation. The ENI System may use any distributed mechanisms, such as agents, to realize shared situational awareness when needed.

6.3.7.4.2 Use of Memory and the Cognition Model

The cognition model used in the ENI System may be quite involved. For example, the cognition model of FOCALE was complex (since it modelled how the human brain thought) and very memory-intensive. The actual cognition model of an ENI System is for further study (see clause 9).

However, an ENI System will, in general, have the following three types of memory modules that are used in the Situational Awareness Functional Block:

- **Working Memory** is a memory of limited capacity for temporary storage of information. Information may be manipulated and transformed in a working memory. It may consist of a one or more modules, where each module is optimized for a particular category of information. It is used to analyse information before committing it to short- or long-term memory.

- **Short-Term Memory** is a memory of limited capacity for temporarily holding, but not manipulating, information. It is a cognitive memory that holds events and concepts that have significance to the managed environment, such as names, numbers, and other managed objects.

- **Long-Term Memory** is a memory of moderate to large capacity where information may be indefinitely held. Long-term memory examines information held in short-term memory and semantically augments it. Repeated occurrences of the same information in short-term memory cause that information to be strengthened (e.g. made more important and certain) in long-term memory. Different types of long-term memory exist; this is beyond the scope of the present document.

6.3.7.4.3 Definition and Management of Goals to be Achieved

There are two types of goals used in this Functional Block. External goals are provided by the Operator or Designated Entity of the Assisted System, and specify objectives that the Assisted System wants to achieve. Internal goals are defined by ENI, and represent goals and/or sub-goals that the ENI System has defined in order to make solving the goal easier. Both internal and external goals may be represented by a set of policies.

6.3.7.4.4 Architecture of a Cognitive Functional Block

A high-level functional block diagram of a situational awareness Functional Block is shown in Figure 6-26. The green rounded rectangle represents the Situational Awareness Functional Block; all other rounded rectangles define a Functional Block that is nested within the Situational Awareness Functional Block. The red arrows represent a closed control loop within the Situational Awareness Functional Block.

The functional block diagram shown in Figure 6-26 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement Situational Awareness. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

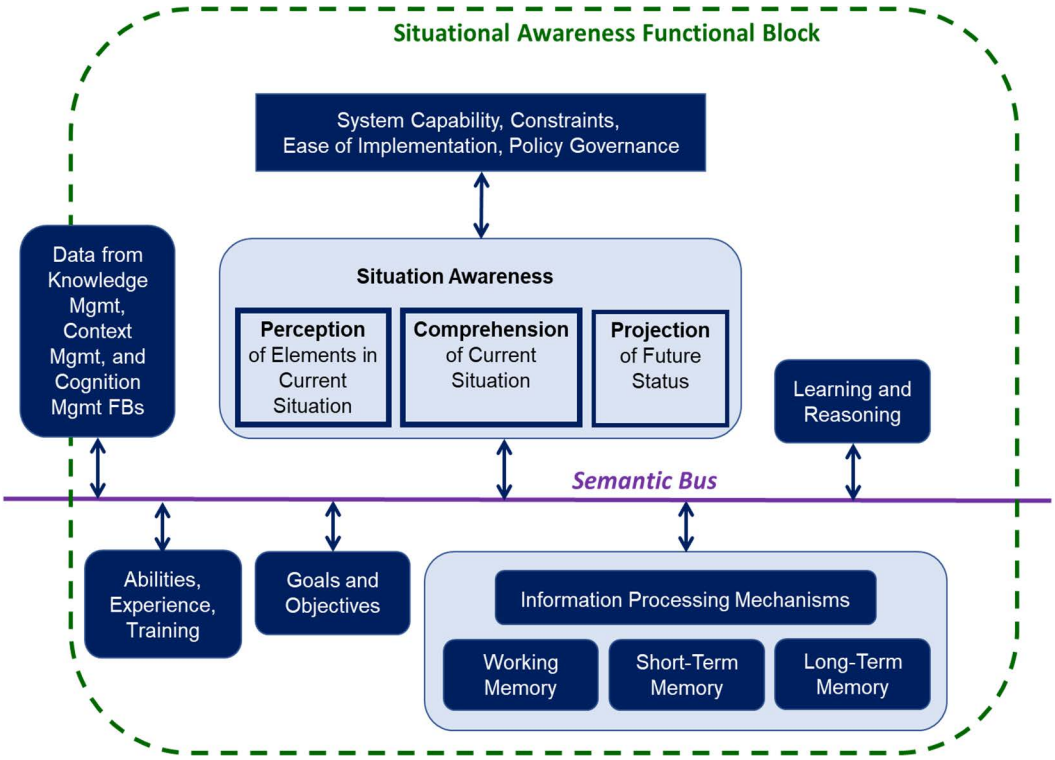

Figure 6-26: A Simplified Cognitive Processing Architecture using Neural Networks

Each of the boxes and rounded rectangles in Figure 6-26 may be modelled as a Functional Block. They are defined as follows:

- **Data from Knowledge Management, Context-Aware Management, and Cognition Functional Blocks:**
  - The combination of data and information from this set of Functional Blocks shall serve as input that enables the Situation Awareness nested Functional Block determine the current state of the Assisted System and its Operational Environment.
  - The other main input comes from external goals received from the Assisted System (or its Designated Entity), as well as internal goals defined by the ENI System that are necessary for its correct operation.
- **System Capabilities, Constraints, Ease of Implementation, Policy Governance:**
  - **System Capabilities** are optional information that define the various functions that the ENI System can perform. They may be specified by metadata. This may be used by this Functional Block to determine the set of viable next actions to take, and choose among them.
  - **Constraints** take two forms. External constraints are provided by the Operator or Designated Entity of the Assisted System, and shall specify limitations that shall be obeyed when an ENI System defines its recommendations and commands. Internal constraints shall be defined by ENI, and represent restrictions that ENI imposes on its decision-making processes in order to achieve a set of goals.
  - **Ease of Implementation** is typically represented as either metadata or using a probabilistic or fuzzy logic mechanism. It enables the Situation Awareness Functional Block to take near-optimal courses of action if taking an optimal course of action is too costly (e.g. in time, resources, or other factors).
  - **Policy Governance** is a set of ENI-generated policies (see clause 6.3.9) that shall be used to manage the operation of the Situation Awareness Functional Block.

- **Situation Awareness:**

  - This defines the ENI System's internal model of the state of the environment. Its input comes from the Knowledge Management, Context-Aware Management, and Cognition Functional Blocks and from external and internal goals. The Situation Awareness function shall be separate from decision-making and action specification functions. This is because even if there is a perfect understanding of the situation, incorrect decisions can still be made. The three following functions describe data analysis and planning performed by this Functional Block.

  - **Perception** produces an awareness of situational elements (e.g. objects, events, people, systems, and environmental factors) and their current states (e.g. modes and locations). Perceived objects may be stored in any of the three types of memories shown in Figure 6-26.

  - **Comprehension** examines the situational elements that have been perceived in order to better understand how they fit together; this helps characterize the situation as a whole, and how this situation affects the goals that the ENI System is trying to achieve.

  - **Projection** is focused on predicting the most likely evolution of the current situation, since it is typically too costly and time-consuming to enumerate all possible situations. However, it may be possible in some situations to specify more evolution predictions. This may be done in a constrained fashion on-line, and/or a more complete fashion off-line. This model also reflects the differences between working, short-, and long-term memory from cognitive psychology.

- **Abilities, Experience, Training:**

  - These are represented by metadata, and provide inputs to enable an assessment of how positive the Situational Awareness Functional Block is in performing a particular operation.

- **Information Processing Mechanisms:**

  - Information processing analyses and optionally changes ingested information. It is made up of four components: input, storage, processor, and output. See clause 5.3 of ETSI GS ENI 019 [9] for a definition of those portions of the MPM that are used in the ENI Information Model, as well as clause 5.3.2.7 and clause 5.3.28 of ETSI GS ENI 019 [9] for a description of ENI Extensions to the MPM and the overall ENI Extended Policy Model, respectively.

NOTE: Information processing is for further study in Release 4 of the present document (see clause 9).

- **Learning and Reasoning:**

  - Learning is the process of acquiring new, or modifying existing, data, information, or knowledge. Reasoning is the process of understanding stimuli and the environment, verifying facts, making inferences, applying a decision-making mechanism (e.g. logic), and then implementing a set of actions to induce change. Both learning and reasoning are defined by different types of AI algorithms. Some important examples of learning include:

    - Non-associative learning is the strengthening of a response to a given stimulus due to repeated exposure to that stimulus.

    - Associative learning is the process of learning an association between different stimuli.

    - Episodic learning is the production of a change in behaviour as a result of one or more events.

6.3.7.4.5 Leveraging Historical Situation Information

Historical information may be used if the algorithm being employed requires such data. It also may be used for trend analysis. Historical information should be stored in an appropriate repository in either this Functional Block or in the Repository Functional Block (see clause 6.3.4.5); the choice depends on whether this information is deemed important for this Functional Block only, or for this Functional Block in particular.

6.3.7.5 Difference between Context Awareness and Situational Awareness

The Context of an Entity describe the state and environment in which an Entity exists or has existed; it uses a combination of historical data, as well as facts and inferences to do this. In contrast, situational awareness includes contextual information and other inputs in order to understand the meaning and significance of data and behaviour of the entire Assisted System and its operational environment; more importantly, situational awareness includes a prediction of the evolution of the situation, and how that evolution affects the goals that the ENI System is trying to achieve. Hence, context is one aspect of situational awareness.

6.3.7.6 Difference between Cognition Management and Situational Awareness

Cognition is the process of acquiring and understanding data and information and producing new data, information, and knowledge. In contrast, situation awareness is the perception of data and behaviour that pertain to the relevant circumstances and/or conditions of a system or process, the comprehension of the meaning and significance of these data and behaviours, and how processes, actions, and new situations inferred from these data and processes are likely to evolve in the near future.

The planner in the Cognition Management Functional Block is responsible for achieving a set of goals given a set of constraints. This may include competing goals (e.g. cost vs. performance) as well as goals that change (e.g. considering a service class less important if resources are scare and allowing resources to be taken from it to satisfy higher priority services). The Cognition Management Functional Block uses the Cognition Model (see clause 6.3.6.3.5) as a guide to determining when state transitions should be made. The output of the Cognition Management Functional Block is input to the Situational Awareness Functional Block.

A situation is defined as a set of circumstances and conditions at a given time that may influence decision-making. This may be constantly changing. Hence, a snapshot of the current situation, and its goals, are sent as input to the Cognition Management Functional Block, which responds with progress on achieving the set of goals defined in that situation.

Hence, the difference between these two Functional Blocks is that the Cognition Management Functional Block is responsible for ingesting, understanding, and producing new data, information and knowledge (by using the ENI Cognition Model), whereas the Situational Awareness Functional Block is responsible for comparing progress on achieving goals from the Cognition Management Functional Block to the achievement of its current goals.

6.3.8 Model Driven Engineering Functional Block

6.3.8.1 Introduction

The Model Driven Engineering (MDE) Functional Block is responsible for enabling software development to be accomplished using models instead of code. The advantage of MDE is that models are, by definition, machine-readable. Hence, they can be used to specify Functional Blocks, programs, and applications. An example of MDE is to generate code directly from a model.

MDE represents an approach to software development where models are used in the understanding, design, implementation, deployment, operation, maintenance and modification of software systems. A set of models may be defined based on different viewpoints. Formally, a viewpoint is an abstraction of the function and behaviour of a system using a selected set of architectural concepts; this facilitates focusing on a particular aspect or set of responsibilities of the system. Model transformation tools and services are used to align the different models (e.g. deriving a set of data models from an information model), and for generating code.

6.3.8.2 Motivation

Software systems continue to grow in complexity. While modelling is commonly used, models are often only used for idea generation and design, and are not linked to implementation. Worse, models are not updated as frequently as code, which increases the separation between the models and the implementation.

One of the original reasons for using models was that concepts that were more familiar to domain experts could be more easily represented in a way that those experts could understand (as opposed to having to know how to write and debug code). Furthermore, this was deemed an easier and more straightforward way of specifying business log that was independent of the platform and technology used. Modelling abstracts technical concepts to make them more accessible to domain experts as well as more casual users. It focusses on aspects of what occurs in the domain of interest. See [i.12] for more detailed information.

Thus, the motivation for using MDE is to focus on the business logic, not code, by using a methodology that uses abstractions (instead of, for example, software libraries and function calls).

6.3.8.3 Function of the Model Driven Engineering Functional Block

The function of the MDE Functional Block is to decide how to implement the selected actions from the Situational Awareness Functional Block. It uses model-driven engineering mechanisms to convert the actions into a form that enables imperative, declarative, and/or intent policies to be constructed (by the Policy Management Functional Block). More specifically, information and data models shall represent key grammatical concepts (e.g. nouns and verbs) of policies, as well as other concepts that a policy grammar can refer to. Therefore, use of the model serves three important purposes:

1) It ensures that all different data models used in the ENI System maintain a consistent definition and understanding of concepts, even if a concept is represented using different data structures in different parts of the ENI System.

2) It enables different policies at different levels of abstraction to communicate with each other using a common vocabulary and data dictionary.

3) It decouples the need for policy (defined by the Situation Awareness Functional Block) from the specification of policy (defined in the MDE Functional Block) from the implementation of policy (defined in the Policy Management Functional Block).

6.3.8.4 Operation of the Model Driven Engineering Functional Block

6.3.8.4.1 Introduction

A high-level functional block diagram of an MDE Functional Block is shown in Figure 6-27. The functional block diagram shown in Figure 6-27 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of model-driven engineering. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

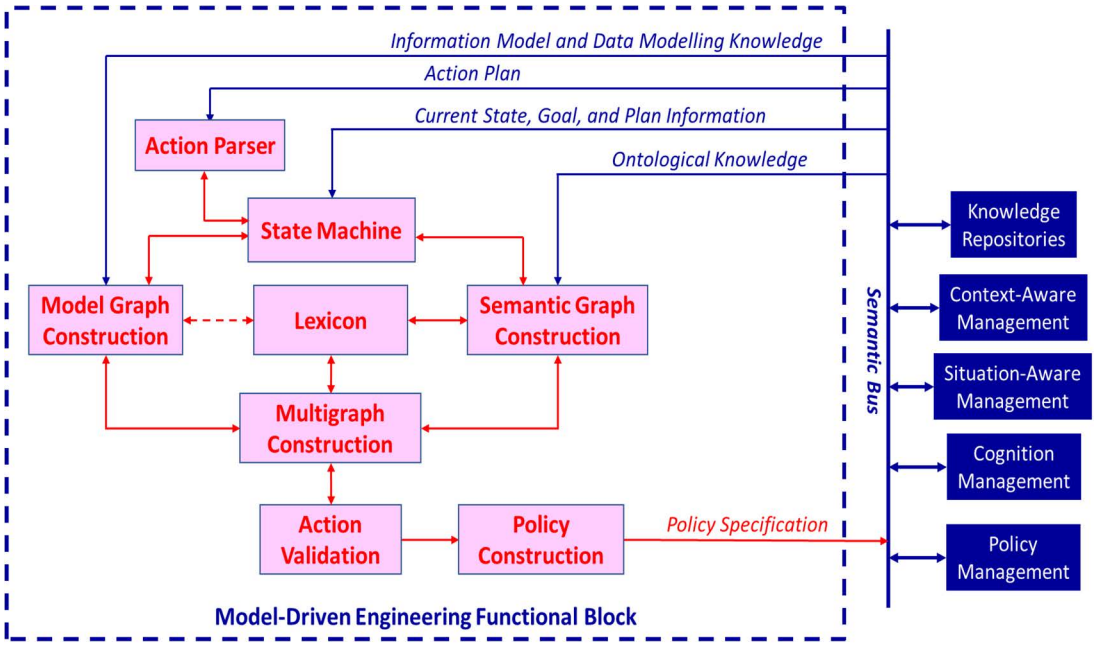

Figure 6-27: A Simplified Functional Architecture of the MDE Functional Block

A high-level description of the flow of operations represented by Figure 6-27 is as follows.

The Situation Awareness Functional Block sends its current state and goal(s), along with any actions to take, to the State Machine of the Model-Driven Engineering Functional Block. These action descriptions shall consist of text, metadata, and/or modelled objects. This is used to synchronize the state of the Model-Driven Engineering Functional Block with the state of the Situation Aware Management Functional Block. The correlation between these actions and the internal and external goals of the ENI System is attached as metadata, and will be embedded into the resulting ENI Policy Rules.

The Model-Driven Engineering Functional Block shall contain multiple active repositories. The two most important are the Information Model Repository and the Data Model Repository. The Information Model Repository shall contain the authoritative information model of record, and may contain one or more copies of that information model that are used to process and evaluate changes to be made to the information model of record. Changes are made to the information model of record to reflect new information that has been learned, including changes to existing information. Any changed information shall be verified by at least one other independent ENI Functional Block before a change is made to the information model of record.

The information model of record shall be used as the authoritative version of the model by all other ENI Functional Blocks. All data models used by ENI Functional Blocks shall be derived from the information model of record.

The Data Model Repository shall contain one or more data models. Each ENI Functional Block may use one or more data models to suit its needs. For example, an LDAP directory and an RDBMS may be used to store information about the same or different managed objects. Each data model shall be optimized to reflect the needs of the Functional Block that is using it. For example, a data model may flatten a set of objects defined in the information model of record to produce a single class that contains all of the attributes, operations, constraints, and behaviours of the information model classes; this may be done to increase and simplify access to data and behaviour. As another example, a data model may restructure and translate data into different data structures (e.g. in order to facilitate protocol operations), as long as in so doing, it does not alter the meaning of the original data.

The Model-Driven Engineering Functional Block should maintain one or more active repositories for applicable ontologies. This is in order to verify the semantics of applying the recommended set of actions from the Situational Awareness Functional Block.

The set of actions defined by the Situational Awareness Functional Block are parsed by the Action Parser, which verifies the structure and meaning of the action descriptions, and ensures that the Model-Driven Engineering Functional Block understands the actions that the Situational Awareness Functional Block wants to take. The first part of this process is to verify the syntax used in the action description. For example, a Customer is a type of Named Entity (i.e. it shall exist in a data model as an instance of a Customer object); if that is not the case, then an error is produced. Any errors and warnings found are sent back to the Situational Awareness Functional Block, along with as much context as the Action Parser can provide; work in the Model-Driven Engineering Functional Block is then stopped. Continuing the Customer example, the Action Parser could annotate that the usage of a term indicates that it could possibly be a Customer, but that no Customer with such a name was found. Two mechanisms of doing this parsing are:

1) using modelled objects to help understand words and phrases in the action description; and

2) using the models to build an (internal) language, whose syntax and semantics are formally defined, and then using the syntax and semantics of that formal language to compile the action descriptions.

The Model-Driven Engineering Functional Block uses its state machine, along with any applicable models and ontological information, to determine how best to implement the recommended set of actions of the Situational Awareness Functional Block. A principal concern is to check the semantics of the recommended set of actions, and any effects that those semantics may have on the running system. Clause 6.3.4.4.1 defined how knowledge is represented in the ENI System. The Model-Driven Engineering Functional Block uses this approach to check the semantics of the recommended set of actions to be applied by iteratively constructing a multigraph, where the nodes are derived from the data models and the edges are semantic relationships that relate one or more ontologies to that node. In addition, some nodes may be derived from concepts in an ontology, and its edges are semantic relationships that relate one or more models to that node.

The Action Validation logic then compares the set of recommended actions from the Situational Awareness Functional Block to the multigraph to ensure that the recommended set of actions are safe. In particular, it examines the set of semantic relationships, looking for semantic closeness between managed entities that may be affected by the set of actions. If it is deemed unsafe, then errors and warnings are sent back to the other Functional Blocks of the ENI System, and the actions are not taken.

If the actions are safe, then the Model-Driven Engineering Functional Block takes those actions and applies them to all applicable data models. This constructs an updated data model, reflecting the effects that the set of actions will have in the runtime system when the set of actions is applied. This defines important data and information that will indicate the success or failure of each action when deployed.

The set of actions are constructed into an internal format, called the Policy Intermediate Form. This is the start of defining a set of policies that represent the formal translation of the set of actions sent by the Situational Awareness Functional Block into a set of ENI Policy Rules. Formally, this is a compilation stage that connects the initial actions of the compiler (in the Model-Driven Engineering Functional Block) to their final output form (in the Policy Management Functional Block). The Policy Intermediate Form may be structured in a tree or graph format. The relationship between the set of actions and the current goal(s) of the ENI System should be embedded in each ENI Policy Rule using metadata or other similar mechanisms. These Policy Definitions take the form of text, and are output to the Policy Management Functional Block, which completes the compilation process by producing a set of imperative, declarative, and/or intent policies for the set of actions produced by the Situational Awareness Functional Block.

6.3.8.4.2 Knowledge Data Fusion, Transformation, and Processing

Data fusion is necessitated when heterogeneous systems are required to interoperate in an open world, where the syntax and semantics of data provided by a sensor or a human is domain-specific and does not conform to any one specific vocabulary. This requires the translation of each vocabulary used into a common set of concepts and terms, so that the data can be integrated. The ENI System shall perform data fusion in order to associate different definitions of the same concept with each other; this is used to provide a more comprehensive understanding of situations.

6.3.8.4.3 Knowledge Transformation into Policy Information

Some systems may need to know additional data, information, and/or knowledge concerning how to deploy and execute a set of ENI Policy Rules. For example, pre- and/or post-conditions, or best current practices, may be useful to execute, monitor, and validate that the actions of an ENI Policy Rule had the desired effect. There are two options for conveying this knowledge:

1) If there is a distributed set of ENI Systems, then this knowledge sharing should be in the form of models, ontologies, and ENI Policy Rules to ensure that each ENI System instance is working on the same knowledge.

2) If the knowledge is being sent to an external, non-ENI System component (e.g. the Assisted System, or an application), then this knowledge sharing shall be in the form of metadata. This lowers the attack surface of the ENI System by ensuring that external entities are not allowed to see or edit models or the knowledge derived from them directly.

6.3.9 Policy Management Functional Block

6.3.9.1 Introduction

This clause describes the motivation for using policy management, the effect it has on the System Architecture, and the benefits that it provides. This Functional Block provides a set of uniform and intuitive mechanisms for providing consistent recommendations and commands. These characteristics give rise to the following requirements:

- ENI shall provide the ability to transform data and information from its own internal format to format that facilitates generating outputs that are understandable by the Assisted System and/or its Designated Entity.
- ENI shall use a set of models, including data types and data structures for producing outputs that are understandable by the Assisted System and/or its Designated Entity.
- Data Denormalization may be realized as a Functional Block that is separate from the Output Generation Functional Block. This adheres to the Single Responsibility Principle [i.9] and enables a more scalable and robust system to be designed and built.

6.3.9.2 Motivation

Management involves monitoring the activity of a system, making decisions about how the system is acting, and performing control actions to modify the behaviour of the system. The purpose of Policy Management is to ensure that consistent and scalable decisions are made governing the behaviour of a system.

Organizations are policy-driven entities. Policy is a natural way to express rules and restrictions on behaviour, and then automate the enforcement of those rules and restrictions. However, the number of policies can be very large (e.g. 100 000+), and the relationships between policies can be complex. In addition, policy can change *contextually.* For example, different actions can be taken based on type of connection, time of day, and network state.

The present document uses the following definition of Policy, see [7], [8] and [i.1]:

> *"Policy is a set of rules that is used to manage and control the changing and/or maintaining of the state of one or more managed objects".*

Policy is a mechanism for controlling the behaviour of an Entity, not the actual end result. For example, an access control list is created and managed using policy, but is not a policy instance or type of policy.

Policy is not absolute. The actions of a policy shall be verified. In addition, a goal of ENI is to continually evaluate and optimize policy, so that it becomes more effective with experience.

The size and complexity of modern systems has resulted in the need for automating management operations. If Policies are coded into a management component, their lifecycle becomes interlocked with that component, and their behaviour can only be altered by recoding the component. Hence, Policies should be defined and management independent from management components to enable policies to be changed and reused without affecting the lifecycle of the management system. It also enables the Policies to adapt to evolutionary changes in the system being managed, as well as to accommodate new application requirements. **Ultimately, the business and operational policies that govern the construction and deployment of configuration changes are more important than the configuration changes themselves!**

6.3.9.3 Modelling and Representing Types of Policies

6.3.9.3.1 Introduction

Management involves monitoring the activity of a system, making decisions about how the system is acting, and performing control actions to modify the behaviour of the system. The purpose of policy is to ensure that consistent decisions are made governing the behaviour of a system.

ENI is a model-driven system. Hence, it uses a single information model that can be used to represent different types of policies (e.g. imperative, declarative and intent). This is described in ETSI GS ENI 019 [9].

6.3.9.3.2 Reuse of the MEF Policy Model

Currently, the only industry information model that defines a unified model for representing different types of policies is the MPM [8]. A unified policy model enables different types of policies to be used to accomplish tasks independent of the type or structure of policy. It also enables one type of policy to call any other types of policy. Therefore, ENI shall reuse the MPM as a starting point to develop its policy information model.

In the present document, reuse of a model means to incorporate text and graphics from an external modelling standard into the present document. Specifically, the ENI model for the present document shall reuse the MPM as it, without any changes.

See clause 5.3 of ETSI GS ENI 019 [9] for a definition of those portions of the MPM that are used in the ENI Information Model, as well as clause 5.3.2.7 and clause 5.3.28 of ETSI GS ENI 019 [9] for a description of ENI Extensions to the MPM and the overall ENI Extended Policy Model, respectively.

6.3.9.3.3 Reuse of the MEF Core Model

The MPM is derived from the MCM [7]. Specifically, the MPM model elements are all derived from an MCM superclass. The MCM is a generic model that defines managed entities (e.g. products, services, and resources), unmanaged entities (e.g. locations and cell towers), entities that are controlled by other managed entities (e.g. IP and MAC Addresses), and metadata. Since the MPM is derived from the MCM, any object (managed or otherwise) defined in the MCM may be defined as a target of an ENI Policy.

In the present document, the ENI model shall reuse MCM model elements referenced by the MPM as it, without changes.

See clause 5.2 of ETSI GS ENI 019 [9] for a definition of those portions of the MCM that are used in the ENI Information Model, as well as clause 5.2.3 and clause 5.2.4 of ETSI GS ENI 019 [9] for a description of ENI Extensions to the MCM and the overall ENI Extended Core Model, respectively.

6.3.9.3.4 Types of Policies Used in ENI

As described in [7], [8] and [i.1], there are three different types of policies that are defined for an ENI System:

Imperative policy: a type of policy that uses statements to explicitly change the state of a set of targeted objects. Hence, the order of statements that make up the policy is explicitly defined. An example of an imperative policy, using informal English, is:

WHEN an Alarm is received
IF the severity of the Alarm is Critical
THEN execute the CriticalAlarm Policy

In the present document, Imperative Policy will refer to policies that are made up of Event, Condition, and Action clauses.

Declarative policy: a type of policy that uses statements from a formal logic to describe a set of computations that need to be done without defining how to execute those computations. Hence, state is not explicitly manipulated, and the order of statements that make up the policy is irrelevant. An example of a declarative policy, using First Order Logic, is:

$$\exists x \exists y\ (Customer(x) \wedge SLA(y) \wedge have(x, y))$$

The English equivalent is:

Some Customers have an SLA

In the present document, Declarative Policy will refer to policies that execute as theories of a formal logic. The syntax of a declarative policy typically uses some type of first order logic, though predicate and description logics may also be used.

Intent policy: a type of policy that uses statements from a restricted natural language (e.g. an external DSL) to express the goals of the policy, but does not specify how to accomplish those goals. In particular, formal logic syntax is not used. Therefore, each statement in an Intent Policy may require the translation of one or more of its terms to a form that another managed functional entity can understand.

In the present document, Intent Policy will refer to policies that do not execute as theories of a formal logic. They typically are expressed in a restricted natural language, and require a mapping to a form understandable by other managed functional entities. An example of an intent policy is:

No processor shall run at more than 75 % utilization

The above example indicates different types of ambiguity that may exist in an intent statement. For example, does the term "processor" include both CPUs and GPUs? What about ASICs that have processing capabilities? As another example, the term "utilization" could refer to memory, I/O operations, or processor utilization.

A further example is:

VoLTE drop rate is less than 0,5 % in cities greater than 1 000 000 people

In this example, the term "drop rate" could refer to average, minimum, or maximum drop rate. In addition, the clause "in cities greater than 1 000 000 people" shall be translated to a specific city or area by the ENI System.

An ENI System may use any combination of imperative, declarative, and intent policies to express recommendations and commands to be issued to the system that it is assisting and/or managing. Each of these types of policies are defined in MEF 95 [8].

6.3.9.3.5 Overview of a Unified Policy Information Model

6.3.9.3.5.1 Introduction

A unified policy information model serves as a common language that enables concepts used by different policy authors to be mapped to equivalent concepts in other levels. It also enables one type of policy to invoke other types of policies. The specification is based on the MEF Policy Model (MPM) [8]. In this model, any policy, regardless of its structure and semantics, shall be abstracted into a set of statements. Each statement may in turn be abstracted into a set of clauses. Each clause is made up of a set of policy elements. Thus, the type of MPMPolicyStructure shall determine the type of statements that it can contain; this in turn shall determine the types of clauses and policy elements that are allowed by this type of statement. The MPM is described in ETSI GS ENI 019 [9].

ENI Policies contain commands and/or recommendations. ENI Policies, as defined in the MPM, may affect the behaviour of managed entities under the management of an ENI System. The MCM defines managed entities of interest to an ENI System. Hence, ENI Policies are able to affect managed entities defined in the MCM since the MPM is also based on the MCM.

6.3.9.3.5.2 Representing Different Types of Policies with a Single Information Model

This clause has been moved to ETSI GS ENI 019 [9], where it has also been enhanced.

6.3.9.4 Processing Policies

6.3.9.4.1 Introduction

Clause 6.3.9.3.2 described three types of policies that may be used by an ENI System (i.e. imperative, declarative, and intent). Clause 6.3.9.3.3 described the important classes of the information model that will be used to represent these policies. The following clauses first describe options for representing and processing policies, then describe the types of different languages that may be used to formally define policies, and conclude with differences between policies that are used within an ENI System versus policies that are used between an ENI System and the Assisted System.

6.3.9.4.2 Constructing Policies: Parsers vs. Compilers vs. Interpreters

Policies used in an ENI System shall be derived from a formal grammar. This simplifies the parsing, compiling, or interpreting of the policy, and increases the understandability of the policy. It also simplifies debugging.

From a linguistics point-of-view:

- the syntax of a grammar is the set of rules used in a grammar to create sentences;
- the semantics of a grammar is the meaning of a sentence;
- the pragmatics of a grammar is the meaning of a sentence in a particular context.

In general, each of these linguistics aspects should be verified for each type of policy used in an ENI System.

Each type of policy used in an ENI System shall be verified using either a parser and/or a compiler. Multiple parsers and/or compilers may be used in the verification process.

Once verified, each type of policy used in an ENI System may be either compiled into executable code or interpreted without having to perform the compilation process.

6.3.9.4.3 Policy Languages

6.3.9.4.3.1 Introduction

Each type of policy in an ENI System shall be written using either a Controlled Language, a Domain-Specific Language (DSL), or a General Purpose Language (GPL).

6.3.9.4.3.2 Controlled Languages

A Controlled Language is a restricted version of a language that has been engineered to meet a particular purpose. The most common form of Controlled Language is a Controlled *Natural* Language, which is a restricted version of a Natural Language, such as English.

More formally, a Controlled Natural Language is a restricted version of a single Natural Language (i.e. the base language) that uses a subset of the grammar of the base language. A Controlled Natural Language preserves most of the properties of the base language, so that speakers of the base language can correctly understand the majority of texts of the Controlled Natural Language. The vocabulary of a Controlled Natural Language is also restricted (typically to 1 000 words or less). Two examples of a Controlled Natural Language is SBVR (Semantics of Business Vocabulary and business Rules) [i.31] and Attempto Controlled English [i.32].

6.3.9.4.3.3 DSLs

A Domain Specific Language (DSL) is a small human-understandable language that uses a higher level of abstraction to communicate and configure software systems for a particular application domain. The term "higher level of abstraction" means that some programming constructs are simplified (possibly at the expense of the associated details being more clearly understood). Examples include constructs that determine the flow of execution of a program, the use and specification of functions, and the types of data structures allowed. It emphases simplicity and the comprehension by application domain experts at the expense of expressiveness and precision.

An important difference between DSLs and General Purpose Languages (GPLs) is that DSLs are typically designed to be used by non-programmers that are experts in the application domain that the DSL is addressing. This is not always true, as DSLs exist for different specialized tasks (e.g. network configuration vs. network monitoring).

There are two main types of DSLs, referred to as internal and external DSLs. An internal DSL does not require a custom compiler or interpreter, because it is embedded into its base language (GPL); hence, its grammar is restricted to a subset of the grammar of the base language. In contrast, an external DSL requires the creation of its own grammar that exists outside of a base language, and hence, requires a compiler or interpreter to execute or interpret it.

6.3.9.4.3.4 GPLs

A General Purpose Language (GPL) is a programming language that can address a wide variety of problems and domains. It emphases expressiveness and precision at the expense of simplicity. It is typically used by professional programmers and developers.

6.3.9.4.3.5 Recommendation

The recommendations for languages used by an ENI system are as follows:

- Intent Policies should use an external DSL, in order to maximize the ability of non-programming constituencies to define and use intent policies.
- Declarative Policies should use either a dedicated logic programming language or a DSL (internal or external) built specifically to handle the logic formalisms used.
- Imperative Policies may use either an appropriate DSL (internal or external) or a GPL.

The ENI system should include the intent grammar specification for Intent Policies. This facilitates the intent translation process and increases interoperability.

While Controlled Natural Languages are attractive for Intent Policies, they would require significant work to develop and maintain, and will not be further discussed in this release of the present document.

6.3.9.4.4 Policy Scope

6.3.9.4.4.1 Introduction

There are two different uses of Policies processed by an ENI System. The first use is when an External Entity (e.g. an Operator) sends a Policy (of any type) to the ENI System that *affects the behaviour of the Assisted System* (or its Designated Entity). This means that the ENI System will translate the Intent Policy, process it, and send recommendations and/or commands back to the Assisted System (or its Designated Entity).

The second use is when an External Entity sends a Policy (of any type) to the ENI System that *affect the behaviour of the ENI System*. This means that the ENI System will translate the Intent Policy, process it, and act on it to affect its own behaviour (e.g. add or remove knowledge from the Knowledge Management Functional Block, or define new goals that it should try and achieve).

For either use of Policies, the Functional Blocks of an ENI System are conceptualized into two categories: External and Internal.

The External Functional Blocks are the Data Ingestion, Data Normalization, Data Denormalization, and Output Generation Functional Blocks. The Internal Functional Blocks are the other six Functional Blocks.

External Functional Blocks communicate to the API Broker using External Reference Points (see clauses 7.2 and 7.3). Internal Functional Blocks communicate using Internal Reference Points via the Semantic Bus (see clauses 7.6 and 7.7).

6.3.9.4.4.2 Policy Communication Requirements

The following requirements apply to both uses of Policies (i.e. external to affect the behaviour of an External Entity vs. internal to affect the behaviour of an ENI System).

All Policy communication between an ENI System and the Assisted System shall use an API Broker.

All Policies sent to and received from the API Broker shall use an appropriate External Reference Point (e.g. $E_{oss\text{-}eni\text{-}pol}$, see clause 6.3.9.5). The External Reference Point is determined by the entity that the API Broker is communicating with. For example, if the communicating entity is the OSS, then it shall use the $E_{oss\text{-}eni\text{-}pol}$ External Reference Point. As another example, if the communicating entity is the end user, then it shall use the $E_{usr\text{-}eni\text{-}pol}$ External Reference Point.

6.3.9.4.4.3 Policies that Affect the Behaviour of an External Entity

The following steps shall be used to process Policies that affect the behaviour of an External Entity:

1) The Data Ingestion Functional Block shall accept input Policies only through designated External Reference Points (see clause 6.3.9.5).

2) The ENI System shall send an acknowledgement of receiving the Policy through the appropriate External Reference Point (see clause 6.3.9.5).

3) The Normalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{ing\text{-}norm}$, see clause 6.3.9.5) to accept Policies from the Data Ingestion Functional Block.

4) The Normalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{norm\text{-}sem}$, see clause 6.3.9.5) to send Policies from the Normalization Functional Block to the Semantic Bus (see clause 6.3.4.4.4).

5) The input Policy is processed using the relevant ENI Internal Functional Blocks, and transformed to a set of recommendations and/or commands, which is published using the Semantic Bus.

6) The ENI System shall send one or more status messages to the External Entity using the appropriate External Reference Point (see clause 6.3.9.5). The number and type of status messages are defined by the External Entity.

7) The Data Denormalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{sem\text{-}denorm}$, see clause 6.3.9.5) to accept Policies from the Semantic Bus.

8) The Data Denormalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{denorm\text{-}out}$, see clause 6.3.9.5) to send Policies to the Output Generation Functional Block.

9) The Output Generation Functional Block shall use an appropriate External Reference Point (e.g. $E_{oss\text{-}eni\text{-}pol}$, see clause 6.3.9.5) to send Policies to the API Broker, which shall subsequently send the recommendations and/or commands to the appropriate External Entity (the OSS in this example) via an API.

10) The External Entity shall acknowledge receipt of the set of recommendations and/or commands through the appropriate External Reference Point (see clause 6.3.9.5).

DSLs and/or GPLs may be used to build Policies that are used within an ENI System. The same type of Policy should be used consistently throughout an ENI System for the same purpose by the same constituency. For example, if Intent Policies are used for business tasks in one Functional Block, they should be used for business tasks in the other Functional Blocks.

6.3.9.4.4.4 Policies that Affect the Behaviour of an ENI System

The following steps shall be used to process Policies that affect the behaviour of an External Entity:

1) The Data Ingestion Functional Block shall accept input Policies only through designated External Reference Points (see clause 6.3.9.5).

2) The ENI System shall send an acknowledgement of receiving the Policy through the appropriate External Reference Point (see clause 6.3.9.5).

3) The Normalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{ing-norm,}$ see clause 6.3.9.5) to accept Policies from the Data Ingestion Functional Block.

4) The Normalization Functional Block shall use a dedicated Internal Reference Point (i.e. $I_{norm-sem}$, see clause 6.3.9.5) to send Policies from the Normalization Functional Block to the Semantic Bus.

5) The input Policy is processed using the relevant ENI Internal Functional Blocks, and transformed to a set of recommendations and/or commands, which is published using the Semantic Bus (see clause 6.3.4.4.4).

6) The transformed set of recommendations and commands are then processed by the ENI System.

7) The ENI System shall send one or more status messages to the External Entity using the appropriate External Reference Point (see clause 6.3.9.5). The number and type of status messages are defined by the External Entity.

8) The External Entity shall acknowledge receipt of the set of recommendations and/or commands through the appropriate External Reference Point (see clause 6.3.9.5).

The Policy Management Functional Block shall use the following Internal Reference Point to communicate with other ENI Functional Blocks:

- $I_{ing-norm}$ is used to transfer data from the Data Ingestion Functional Block to the Data Normalization Functional Block. Ingested data may include all types of data, information, knowledge, Policies, and metadata sent from the API Broker through any of the External Reference Points that supply inputs to the Data Ingestion Functional Block.

- $I_{norm-sem}$ is used to transfer normalized data from the Data Normalization Functional Block to the Semantic Bus. This is a uni-directional Internal Reference Point, meaning that data for processing shall only flow from the Data Ingestion Functional Block to the Data Normalization Functional Block.

- I_{sem-pm} is used to transfer all types of data, information, knowledge, Policies, and metadata from the Semantic Bus to the Policy Management Functional Block that the Policy Management Functional Block has subscribed to. The Policy Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

- $I_{sem-denorm}$ is used to transfer data from the Semantic Bus to the Data Denormalization Functional Block. These data may be data, information, knowledge, policies, and metadata from any internal ENI Functional Block that is necessary to communicate to the Assisted System or its Designated Entity.

- $I_{denorm-out}$ is used to transfer data from the Data Denormalization Functional Block to the Output Generation Functional Block, where it will be sent by an appropriate External Reference Point (see Figure 7-2 and clauses 7.2 and 7.3 for their definitions). Data output may include all types of data, information, knowledge, policies, and metadata.

6.3.9.5 Function of the Policy Management Functional Block

The function of the Policy Management Functional Block is to turn the input from the Model Driven Engineering Functional Block into a set of ENI Policies that contain recommendations and/or commands. This enables a common mechanism for an ENI System to communicate actions to take to an entity. The ENI Policies may be any combination of imperative, declarative, and/or intent policies.

The Policy Management Functional Block shall use the following External Reference Points to send ENI Policies to the API Broker for subsequent transmission to any external entity:

- $E_{oss\text{-}eni\text{-}pol}$ defines Policies and associated information and/or metadata exchanged between the OSS-like Functionality and the ENI System that control behaviour (including services and resources) for the Assisted System.
- $E_{app\text{-}eni\text{-}pol}$ defines Policies and associated information and/or metadata exchanged between the BSS-like functionality and the ENI System that control behaviour (including services and resources) for the Assisted System.
- $E_{bss\text{-}eni\text{-}pol}$ defines Policies and associated information and/or metadata exchanged between the BSS-like functionality and the ENI System that control behaviour (including services and resources) for the Assisted System.
- $E_{usr\text{-}eni\text{-}pol}$ defines Policies and associated information and/or metadata exchanged between Applications and the ENI System that control behaviour (including services and resources) for a user (or an agent acting on behalf of the user).
- $E_{or\text{-}eni\text{-}pol}$ defines Policies and associated information and/or metadata exchanged between the Orchestrator and the ENI System that control behaviour (including services and resources) for the Assisted System.

The Policy Management Functional Block shall use the following Internal Reference Point to communicate with other ENI Functional Blocks:

- $I_{sem\text{-}pm}$ is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Policy Management Functional Block that the Policy Management Functional Block has subscribed to. The Policy Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

6.3.9.6 Operation of the Policy Management Functional Block

6.3.9.6.1 Introduction

The Policy Management Functional Block is used to send recommendations, commands, and associated data and metadata to external entities. Imperative, declarative, and/or intent policies may be used. More specifically, all classes in the MPM inherit from MCMPolicyObject, which in turn inherits from MCMEntity. The MCMEntityHasMCMMetaData aggregation defines which MCMMetaData objects may be aggregated by which MCMEntity objects. This aggregation is owned by the MCMEntity class, which means that all of its subclasses inherit this aggregation. Hence, both ENI Policies (represented as subclasses of the MPMPolicyStructure class) and their components (represented as subclasses of the MPMPolicyComponentStructure class) may all have zero or more subclasses of the MCMMetaData class.

MCMMetaData is used to describe as well as prescribe information that is associated with a given MCMEntity. For example, best current practices and version information are examples of descriptive This is shown in Figure 6-28.

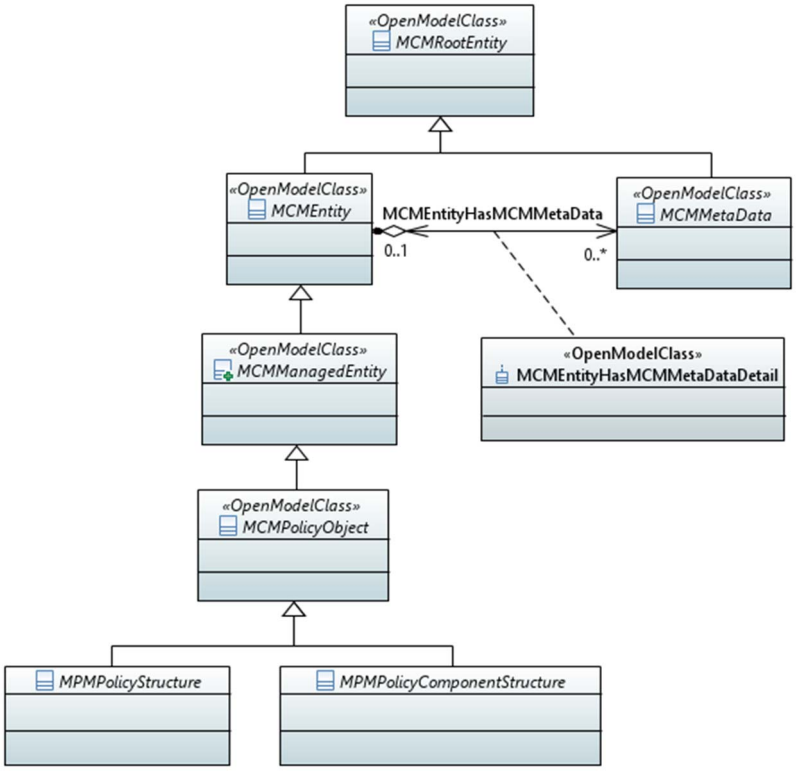

Figure 6-28: Policies and Metadata

6.3.9.6.2 The Policy Continuum

Figure 6-29 illustrates a key concept of Policy, called the Policy Continuum [7], [8], [i.1], [i.2] and [i.3].

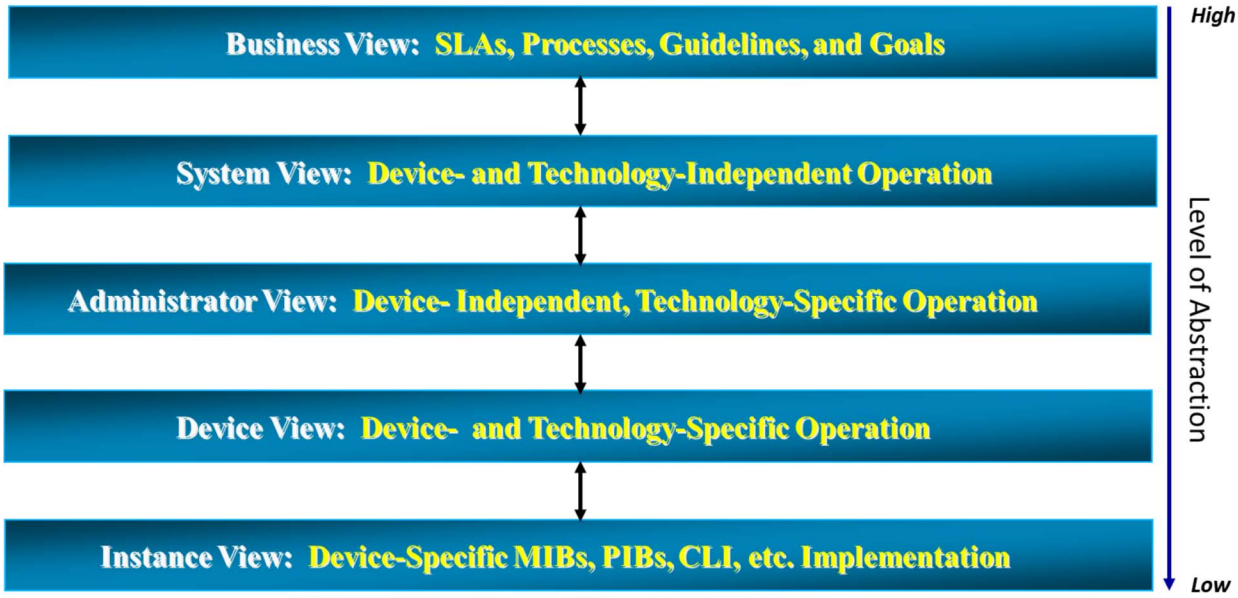

Figure 6-29: The Policy Continuum

The purpose of the Policy Continuum is to formally differentiate between the needs of different constituencies in defining and expressing policy. Each constituency is made up of a set of users that have similar business needs, and more importantly, use similar concepts and terminology. For example, business users and product managers use significantly different terminology than application developers or network administrators. The number of continua in the Policy Continuum shall be determined by the applications using it. There is no fixed number of continua. Figure 6-29 shows five, because this enables a set of much smaller translations of terms (e.g. from a representation without technology, to one with technology while being device, vendor, and technology independent, to successively lower levels that fix each of these three dimensions). However, Figure 6-29 is used to illustrate the principles of the Policy Continuum, not to define the type or number of continua used in ENI.

6.3.9.6.3 Policy Management Architecture

6.3.9.6.3.1 Policy Input

Figure 6-30 shows the set of External Reference Points that may be used to send policies to the ENI System. Five types of Policy Users (the End-User, an Application, the OSS, the BSS, and the Orchestrator) shall be able to send Policies to the ENI System. In addition, certain types of Applications may be able to submit knowledge specific to a set of Policies that they want to execute.

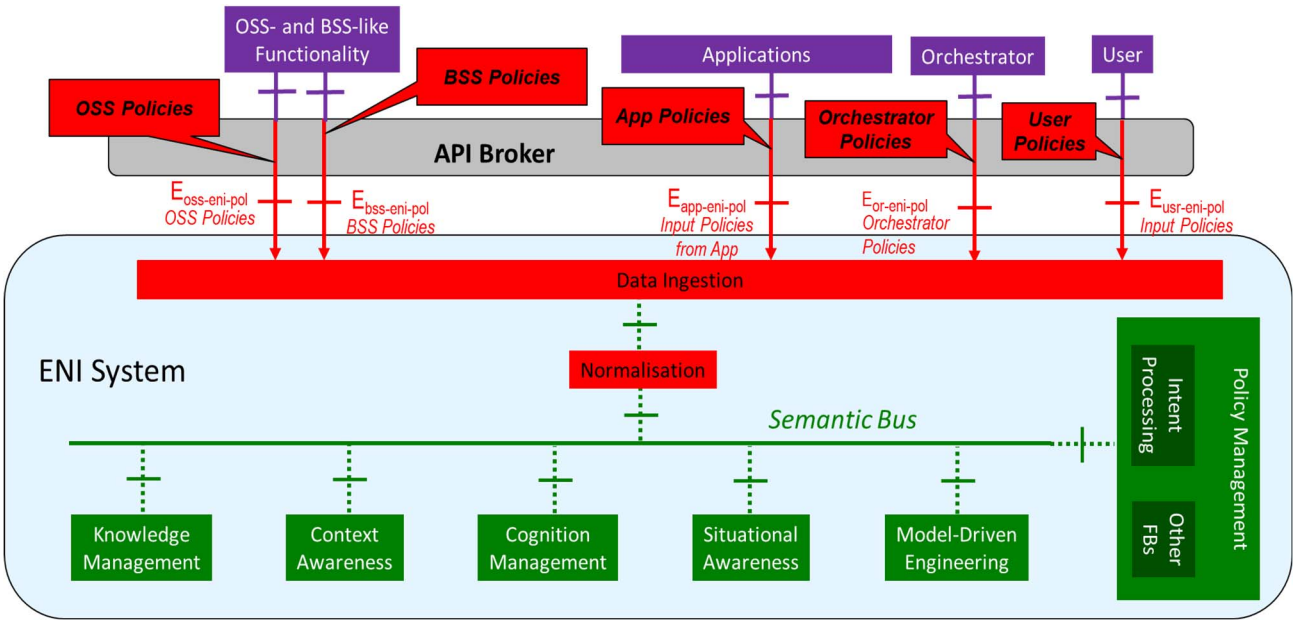

Figure 6-30: Policy Input External Reference Points

There are five types of input Policies, plus the ability to ingest information and knowledge that applies to policies from a particular source, that are considered in the present document. They are:

1) **End-User Policies.** End-users, such as Subscribers, shall use the $E_{user\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the ENI System.

2) **Application Policies.** Different types of Applications shall use the $E_{app\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the ENI System.

3) **OSS Policies.** Most OSSs and BSSs currently do not have a standardized Policy Language. This is mitigated by the use of a dedicated External Reference Point. Hence, the OSS will shall use the $E_{oss\text{-}eni\text{-}pol}$ External Reference Point to send OSS Policies to the ENI System.

4) **BSS Policies.** Most OSSs and BSSs currently do not have a standardized Policy Language. This is mitigated by the use of a dedicated External Reference Point. Hence, the BSS shall use the $E_{bss\text{-}eni\text{-}pol}$ External Reference Point to send BSS Policies to the ENI System.

5) **Orchestrator Policies.** Orchestrators shall use the $E_{or\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the ENI system.

Related external knowledge and information ingestion for input policies are:

- **Application Knowledge.** The External Reference Point $E_{app\text{-}eni\text{-}kno}$ may be used to add application-specific knowledge to the Knowledge Repositories of ENI. For example, such knowledge could describe particular semantics that an application wants to associate with a set of Policies.

Related internal knowledge and information for input policies are:

- **Application Knowledge.** The Internal Reference Point $I_{sem\text{-}pm}$ may be used to add internal knowledge from other Functional Blocks of the ENI System that is relevant to the processing of any ingested policy. For example, such knowledge could include the current context and situation, as well as historical events.

6.3.9.6.3.2 Policy Acknowledgements and Output

Figure 6-31 shows the set of External Reference Points that may be used to send policies to ENI. Four types of Policy Users (the End-User, an Application, the OSS, and the BSS) shall be able to receive Policies from ENI. In addition, certain types of Applications may be able to receive knowledge specific to a set of Policies that they want to execute.

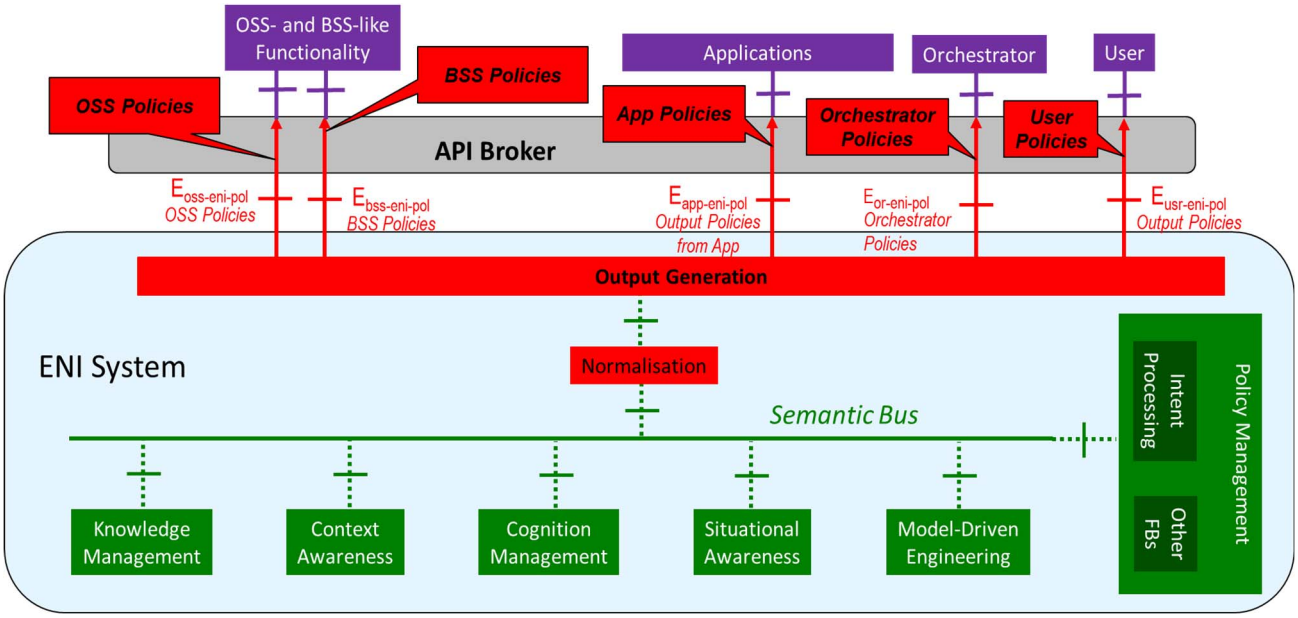

Figure 6-31: Policy Output External Reference Points

There are five types of output Policies, plus the ability to export information and knowledge that applies to policies from ENI, that are considered in the present document. They are:

1) **End-User Policies.** The ENI System shall use the $E_{user\text{-}eni\text{-}pol}$ External Reference Point to send Policies to different End-Users.

2) **Application Policies.** The ENI System shall use the $E_{app\text{-}eni\text{-}pol}$ External Reference Point to send Policies to applications.

3) **OSS Policies.** The ENI System shall use the $E_{oss\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the OSS.

4) **BSS Policies.** The ENI System shall use the $E_{bss\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the BSS.

5) **Orchestrator Policies.** The ENI System shall use the $E_{or\text{-}eni\text{-}pol}$ External Reference Point to send Policies to the Orchestrator.

Related external knowledge and information ingestion for output policies are:

- **Application Knowledge.** The ENI System shall use the $E_{app\text{-}eni\text{-}kno}$ External Reference Point to send knowledge curated by an ENI System to specific applications. For example, such knowledge could describe particular semantics that an application needs to enforce when it uses a particular set of Policies.

Related internal knowledge and information for output policies are:

- **Application Knowledge.** The Internal Reference Point I_{sem-pm} may be used to send policy information and knowledge to other Functional Blocks of the ENI System that is relevant. For example, such knowledge could include the current policy used to remedy a problem for future processing in similar situations.

6.3.9.6.3.3 Policy Management Options

There are several alternatives for building a Policy Management Architecture. This is because ENI treats different types of policies, such as intent, imperative, declarative, and utility functions, as instances of a common Policy. That is, their content is different, but their object type is the same. This is a powerful abstraction, and enables ENI to have different policy types interact. This clause will consider two such options. Option 1 assumes that the operation of different types of policies is supported using the same set of (nested) Functional Blocks. Option 2 assumes that the operation of different types of policies is supported using a fundamentally different set of (nested) Functional Blocks.

6.3.9.6.3.4 Option 1: A Single Unified Policy Management Architecture

This option assumes that the same set of (nested) Functional Blocks are used to manage and process different types of Policies, whether those Policies are internal or external. This may be thought of as a "templatized" architecture; the same set of high-level Functional Blocks always exist, and may operate differently in response to processing different policy types.

Figure 6-32 shows a simplified functional architecture of the Policy Management Functional Block. The functional block diagram shown in Figure 6-32 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of policy-based management in a given administrative domain. Different implementations may need to add other Functional Blocks to meet their particular operational requirements. An exemplary implementation is described in [i.1] and [i.11].

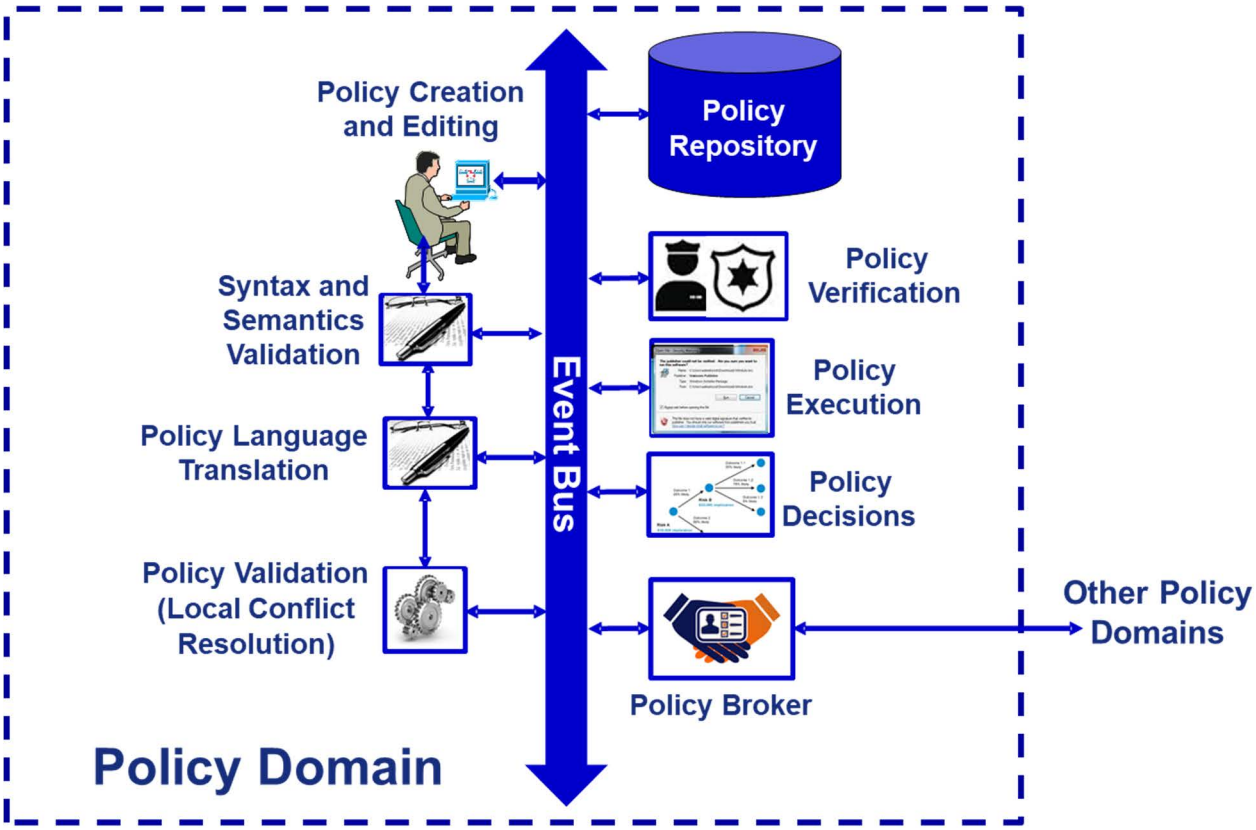

Figure 6-32: Option 1: a Templatized Policy Management Functional Architecture

Any combination of textual, graphical, and/or command-line tools may be used to author any type of Policy. The Policy Management Functional Block (PMFB) may assist in helping the user to author and edit policies. For example, the PMFB may identify errors, such as incorrect syntax, to the user, before and after the user submits the policy to the ENI System. An event may be created when the user is finished authoring or editing the policy and submits it to the ENI System.

Once a completed policy is received by the ENI System, that policy may be saved in a Policy Repository. A completed policy shall then be either parsed or compiled. In either case, the parsing or compilation shall first check for correct syntax and semantics. Any warnings or errors shall be sent back to the user to correct. Operation on a policy with errors and/or warnings shall not proceed until those errors and/or warnings are fixed.

Once a policy has been successfully parsed or compiled, additional operations may be necessary before it can be deployed, depending on the type of policy and the type of actions to be performed:

- **Policy Language Translation:**
  - Sometimes a policy may need to be translated from one form to another form. The most common case is when a policy is input at a high-level of abstraction and needs to be translated to a more concrete form in order to be validated and processed. This shall be done for all intent policies that are input to the ENI System, since by definition, an intent policy is written in a restricted natural language, such as an external DSL.
- **Policy Validation - Local Conflict Resolution:**
  - Local Conflict Resolution ensures that a new policy shall not conflict with any deployed policies. This ensures that policy management is always deterministic.
  - A policy is said to conflict with another policy if the actions of a policy cause the behaviour of other policies to be changed during the same execution scope and time. For example, if a Policy Foo sets an attribute bar to 1, and a subsequent policy sets the same attribute bar to a different value, that is a conflict. This subject is treated more thoroughly in [8].
  - The essential part of policy conflict resolution is determining the execution scope and time. For the purposes of the present document, the execution scope is defined as the set of variables, object, functions, and data structures that a Policy has access to at a particular time during its lifetime. Execution time is defined as the time during which a Policy is deployed, enabled, and active.
    For Imperative Policies, execution time and scope are conceptually straightforward. Given a set of Policies P_1 - P_n:

    Execution scope: the time when all Policies are active and enabled, their event clauses all evaluate to TRUE, their condition clauses all evaluation to TRUE, but two or more of their action clauses cause conflicting actions.

    Execution time: the time between the start of the first Policy to execute (e.g. P_1) and the end of the last Policy to finish execution (e.g. P_n).
  - Local conflict resolution ensures that no actions in a set of policies conflict when they are executed.
- **Policy Language Validation - Global Conflict Resolution:**
  - Global Conflict Resolution ensures that policies that execute in different Policy Domains shall not conflict with each other.
  - Policy Domains communicate with other Policy Domains through a special mechanism called a Policy Broker. This entity facilitates communication between Policy Domains that are collaborators, consumers, and producers:
    - Two Policy Domains collaborate when the actions taken by both Policy Domains affect the same set of entities in a non-conflicting manner.
    - A Policy Domain C is a consumer of a Policy Domain P when one or more policies of Policy Domain P are used by Policy Domain C as part of its execution. Similarly, in this example, Policy Domain P is said to be a producer for Policy Domain C.

- **Policy Decision Entity:**
  - A Policy Decision Entity is a managed entity in a Policy Management Domain that is responsible for deciding which, if any, policies shall be executed in response to a request by another managed entity for a set of governance actions.
- **Policy Execution Entity:**
  - A Policy Execution Entity is a managed entity in a Policy Management Domain that is responsible for managing the execution of a set of Policies.
- **Policy Verification Entity:**
  - A Policy Verification Entity is a managed entity in a Policy Management Domain that is responsible for verifying that a Policy that was executed by a Policy Execution Entity operated as expected.
- **Policy Broker:**
  - The Policy Broker is responsible for facilitating communication between different Policy Domains.

NOTE: This subject is for further study in Release 4 of the present document (see clause 9).

Figure 6-32 shows that each Policy Domain may provide its own closed control loop. This may start at any one of the nested Functional Blocks (e.g. the Syntax and Semantics Validation) and return to that nest Functional Block via the Event Bus. In addition, the combination of Policy Decision Entity, Policy Execution Entity, and Policy Verification Entity provide feedback for any decisions made; this feedback may be logged by the Policy Domain. Once logged, this feedback may be published to other Policy Domains to aid their decision-making.

6.3.9.6.3.5 Option 2: Differentiating between Intent and Other Types of Policies

This option assumes that a different set of (nested) Functional Blocks are used to manage and process each type of Policy, whether those Policies are internal or external.

Figure 6-33 illustrates the important Functional Blocks that are particular to processing Intent Policies.

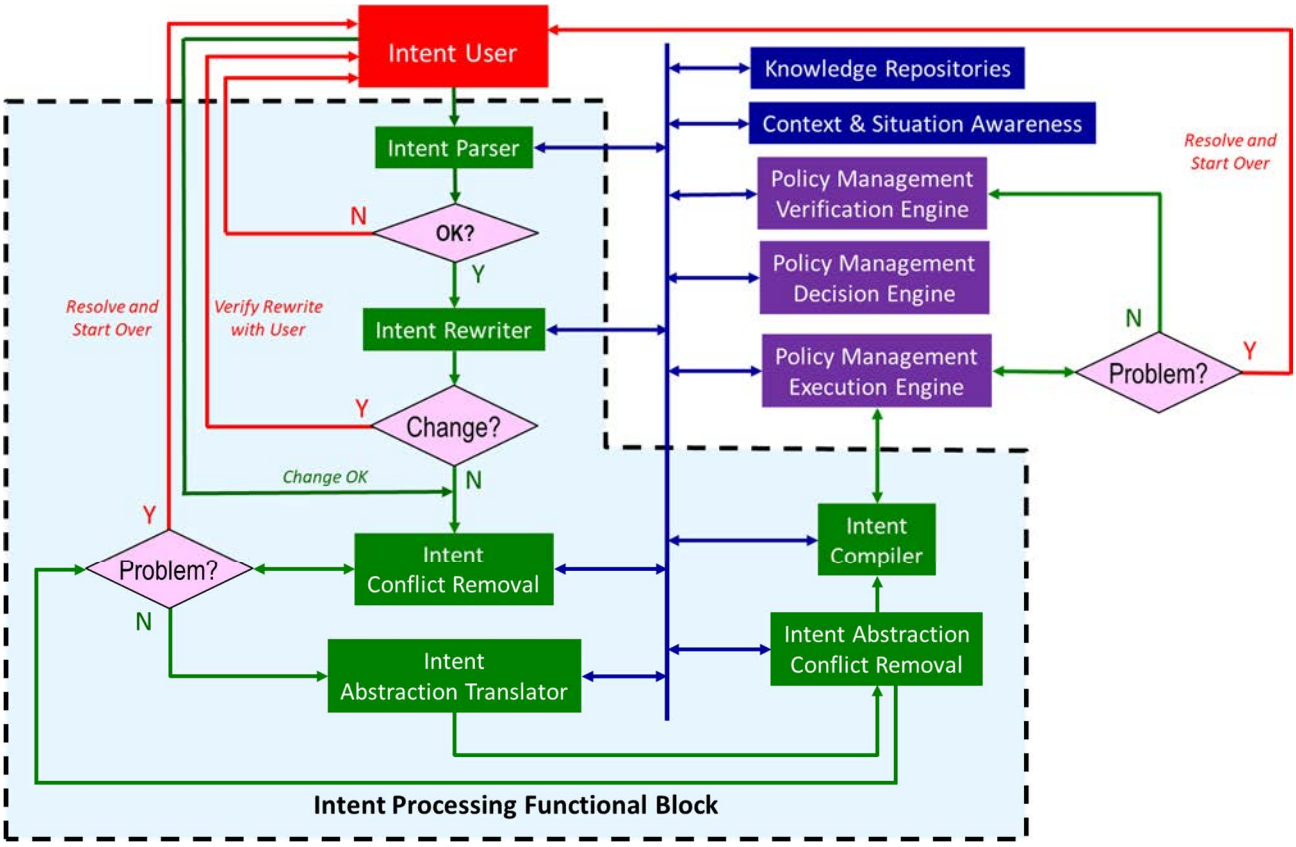

Figure 6-33: Option 2: a Dedicated Processing of Intent Policies

Since Intent Policies are submitted by External Entities, then they are subject to validation by an ENI System. This is particularly important in the case of Intent Policies, since the input policy text may have syntactic or semantic errors, and also be ambiguous in nature. If the Intent Parser detects such an error, then it will stop processing and return the error to the user.

The first important difference between processing Intent Policies and other types of Policies is that Intent Policies are written in a restricted natural language, such as an external DSL, and then submitted to the ENI System. Hence, Intent Policies shall need a translation mechanism to change the submitted Intent Policy into a form that the ENI System is able to understand. This intent translation function is performed by the Intent Parser. It uses knowledge from various Knowledge Repositories (e.g. the ENI Information and Data Models, along with knowledge that is specific to working with Intent Policies, referred to as the Intent Knowledge) to translate the input Intent Policy to a form that can be understood by the ENI System. The Intent Parser contains both syntactical (i.e. lexical analysis, token generation, and syntactic analysis) and semantic (e.g. type checking, binding of objects with references, and semantic annotation to guide further analysis) analysis.

The second important difference between processing Intent Policies and other types of Policies is that Intent Policies, once translated, may need to be rewritten. For example, a word or phrase may be substituted to better match objects in the information or data model. If the rewriting changes the Intent Policy in any significant way, then the Intent Processing Functional Block should first verify with the Intent User (i.e. an End-User, Application, OSS, or BSS) that the rewrite of the intent policy is acceptable to the Intent User. If it is not, then the processing stops for that Intent Policy until a new Intent Policy is submitted by that Intent User. Otherwise, the Intent Policy is then checked for conflicts with other existing Intent Policies at that particular level of abstraction. For example, multiple Intent Policies that involve End-Users may be checked for conflicts, while Intent Policies that involve specific devices may not be checked for conflicts at this time. If a conflict is detected, then the Intent User is notified of the problem, and processing of this Intent Policy is stopped until the Intent User submits a new policy. Otherwise, processing continues.

The third important difference between processing Intent Policies and other types of Policies is that Intent Policies may need to be translated to a different level of abstraction. For example, an Intent Policy that identifies a specific End-User may need to be translated to a different form, such as a range of IP or MAC addresses, for further processing. One or more such abstraction translations are performed, as required. Each such abstraction translation may require a repeat of one of more of the previous steps.

A necessary consequence of the above processing is that, for each abstraction translation, conflicts shall be checked for that level of abstraction. Once all abstraction conflicts are resolved, then the final Intent Policy is compiled to a form that can be sent to the Policy Management Execution Functional Block. This is where additional checks (e.g. conflicts between the Intent Policy and other Policies that are not Intent-based) are performed.

6.3.9.6.4 Policy Management Federation

NOTE: This is for further study in Release 4 (see clause 9).

6.3.9.6.5 Constructing, Deploying, and Activating Policies

ENI Policies should be constructed using model-driven techniques. A model of a system is a description or specification of that system and its environment for some certain purpose. Model-driven is a methodology for using models to direct the understanding, design, construction, deployment, operation, maintenance and modification of software systems.

Figure 6-34 shows a functional block diagram of model-driven engineering for constructing and deploying policies [i.38]. In this figure, the numbered steps are explained as follows:

- Step 1 annotates the information model with tag-value pairs, a UML extension to the meta-attributes of a UML model; this is used to add information for tools to manipulate the model.
- Step 2 generates one or more DSLs that are used to describe the object model(s) created from the information model. This enables these models to be shared across different tools and programming languages to enable a consistent and coherent view of the information to be accessible. Multiple DSLs, along with associated parsers and editors, may be generated from identified and tagged subsets of an information model. This enables different object models to be defined that correspond to the disparate aspects of the managed.
- Step 3 generates a "policy DSL" to model the behaviour of the system being managed. This ensures that behaviours defined by policies are supported in and consistent with the information model.
- Step 4 generates one or more ontologies that can be used to provide automated reasoning capabilities for the set of produced policies. Alternatively, existing ontologies may ne matched to the needs of the above DSLs. However, ontology matching is a complex subject, and is for further study in a future Release.
- Step 5 analyses the produced policies. This includes syntactic and semantic checking and conflict resolution. Reasoning is used to help resolve any conflicts found. In addition, reasoning is used to transform a policy to a different form in the Policy Continuum (see clause 6.3.9.6.2). This can also be used to transform a policy of one type (e.g. intent) to a policy of another type.

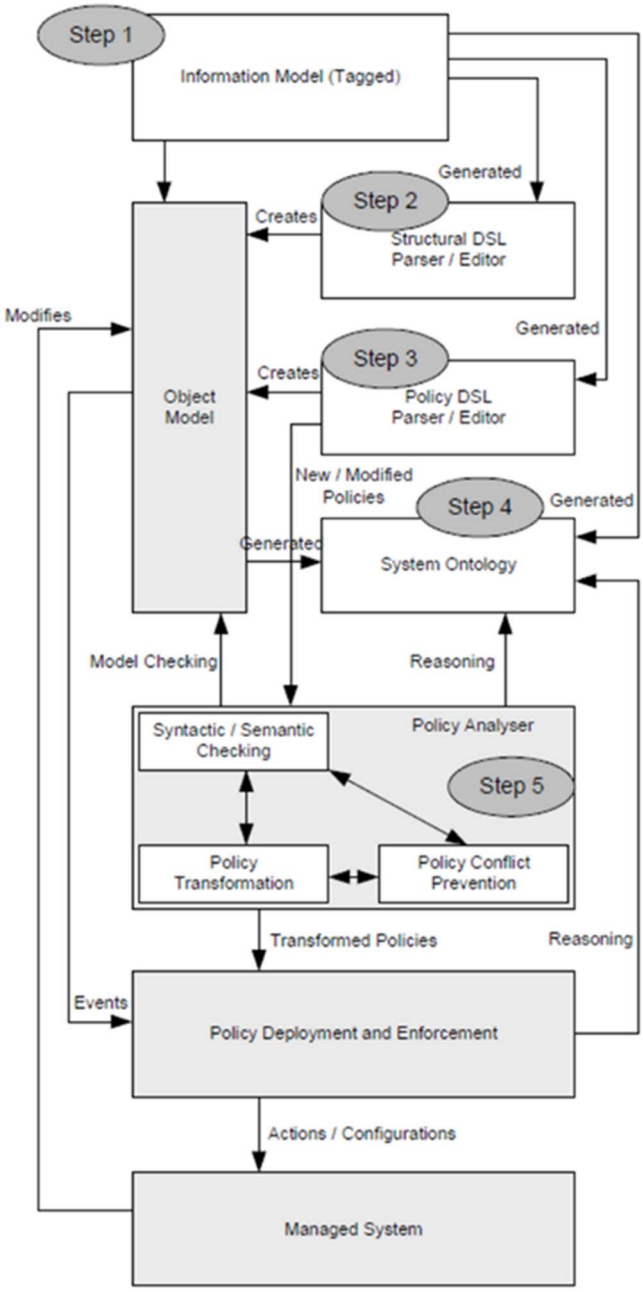

Figure 6-34: Model-Driven Engineering using Policies

Once an ENI Policy has been analyses, it may be deployed, activated, and enforced. Activation enables the policy to be used in decision-making. Metadata contained in the policy may constrain the use of a policy (e.g. between a particular set of start and end times). Enforcement may be done by a range of different mechanisms, from simple assertions embedded in the policy to dynamic reasoning. Policy execution is used to control state transitions, which represent the behaviour of the system being managed.

6.3.9.6.6 Managing Policies

Figure 6-35, [4], [i.1] and [i.3] show a simplified functional block diagram of the FOCALE architecture. This is a cognitive and adaptive set of closed control loops (see ETSI GR ENI 017 [i.36] that uses policies to direct the processing of adaptive control loops that are model-driven.

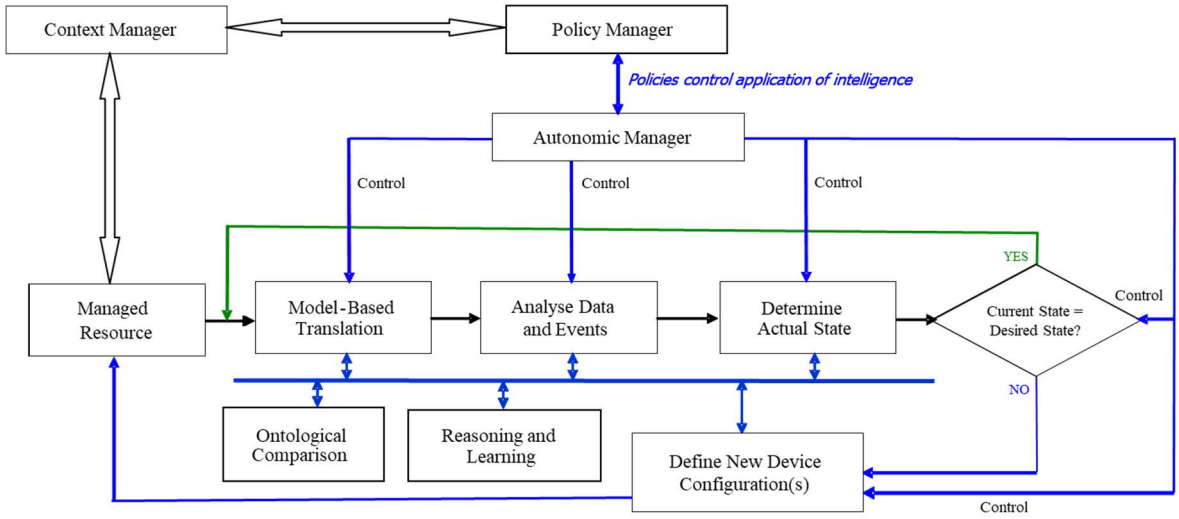

Figure 6-35: Simplified Functional Block Diagram of the FOCALE Architecture

The Model-Based Translation (MBT) layer is similar in function to the Data Ingestion and Normalization Functional Blocks of the ENI Architecture, except that the MBT is purely model-driven. Normalized data and events are then analysed, and the current state of the system being managed is determined. This is then matched to the desired state. If the two match, the upper monitoring loop is taken, since no action is needed, and the loop shall begin again where it started. If they do not match, then the lower reconfiguration loop is taken. The reconfiguration loop may direct the control loop to start at a new (logical) part of the system; it may also instantiate a completely new control loop.

The Policy Manager, in conjunction with the Autonomic Manager, controls the behaviour of FOCALE. Context-aware Policies are used to capture business goals and constraints, as well as convert *intent policies* from various constituents to a common normalized form. The Autonomic Manager then takes these policies, along with additional information, then translates these directives into a form that controls the composition and function of each of the Functional Blocks shown in Figure 6-35.

The models and ontologies are *continuously updated* to reflect the changing state of the system being managed and its environment. This motivates the need for *active repositories* (see clause 6.3.4.5). Knowledge processing (see clauses 6.3.4.4, 6.3.4.6 and 6.3.4.7) is thus continually performed.

Policies are thus managed throughout the execution of the control loops.

ENI Policies shall be continuously monitored and updated throughout the lifecycle of the ENI System. This should include, but is not limited to:

- continuously monitoring the working set of policies for correct execution;
- continuously monitoring the working set of policies for possible conflicts;
- continuously look to improve the accuracy of its active knowledge repositories;
- support *goal-orientated* tasks (i.e. enable the system to focus on dynamic procedures, rather than static, pre-defined tasks) to reflect the changing prioritization of user needs, business goals, and environmental conditions.

6.3.9.6.7 Deactivating and Removing Policies

ENI Policies should be deactivated when they are no longer applicable to the current situation. One approach is to scan the current set of states in the state machine(s) that are being used by the ENI System. As previously mentioned, a policy is related to a state transition. Therefore, if no states exist that could be reached by the policy, then it is not currently applicable.

ENI Policies that are deactivated may still be kept in local or remote storage for later use. In general, an ENI Policy should only be removed when it is known that it cannot be applicable to the current tasks at home, or that it is not operating correctly. Care should be taken to ensure that the plan(s) generated by the control loops (see clause 6.3.4.7 and ETSI GR ENI 017 [i.36]) will not require the use of a policy. If that is true, then the policy in question may safely be removed. However, it is still much simpler to deactivate a policy, as this keeps the policy, and all of the associated work required for its instantiation, available.

6.3.10 Denormalization Functional Block

6.3.10.1 Introduction

The Data Denormalization Functional Block receives processed data from the ENI System. These data and information are in one or more ENI internal formats. The Data Denormalization Functional Block then translates these data and information into a form that the Assisted System (and/or its Designated Entity) may understand.

These characteristics give rise to the following requirements:

- ENI shall provide the ability to transform recommendations and commands into a form that the Assisted System (and/or its Designated Entity) may understand.
- ENI shall use a set of models, including data types and data structures, to perform the transformation into native format(s) used by the Assisted System (and/or its Designated Entity).
- Data Denormalization may be realized as a Functional Block that is separate from the Output Generation Functional Block. This adheres to the Single Responsibility Principle [i.9], and enables a more scalable and robust system to be designed and built.

6.3.10.2 Motivation

Each ENI Functional Block may use different data structures and representation of information that is specific to its own processing. Such data structures and representation may be different in nature to the data structures and representation used by other ENI Functional Blocks. In all cases, these data structures and representations are internal to ENI for efficiency of internal understanding and operation. Therefore, each ENI Functional Block may use different programming languages and protocols, which means that the same data may be represented in completely different formats. For example, the same data about a customer may be processed using relational databases and directories. It is important to consolidate all changes to information and data needed by the Assisted System (and/or its Designated Entity); this ensures that conflicting commands and recommendations are not sent to the Assisted System (and/or its Designated Entity).

In this example, recognizing that the object is a Customer, even though the data is different, may be done by comparing the ingested data with expected data from the model. For example, the Customer object in an information model has a set of mandatory (and optional) attributes, as well as relationships to other objects. It may also have metadata that disambiguates this instance of a Customer from all other instances of a Customer. The Denormalization process arranges and formats the data and information to be output so that it may be more easily and efficiently translated into a form that is understandable by the Assisted System (and/or its Designated Entity).

6.3.10.3 Function of the Denormalization Functional Block

As stated previously, the ENI System may use one or more internal formats to represent, analyse, and process data and information. The ENI System shall not expect the Assisted System (and/or its Designated Entity) to be able to understand the internal format(s) of ENI. Data Denormalization translates information and data in one or more internal formats used by the ENI System to a standardized form; this facilitates the generation of recommendations and commands in a form that the Assisted System (and/or its Designated Entity) is able to understand.

A normalized form is critical for the operation of other ENI Functional Blocks. A normalized form means that every ENI Functional Block may be designed using knowledge about the characteristics and behaviour of pertinent data and objects. Otherwise, each ENI Functional Block would have to accommodate data that could be represented in different formats using different data structures. Similarly, a denormalized form is critical for the Assisted System (and/or its Designated Entity) to be able to understand recommendations and commands given to it by ENI without having to be modified. This lack of modification is important to facilitate early adoption of ENI. The Assisted System (and/or its Designated Entity) need not be able to understand recommendations and commands in the internal format(s) of the ENI System. Therefore, a denormalized form is critical for enabling the ENI System to produce outputs in a form that the Assisted System (and/or its Designated Entity are able to understand without any additional assistance. The purpose of the Denormalization Functional Block is to decouple the internal format(s) used by the ENI System from the native format(s) used by the Assisted System (and/or its Designated Entity); this enables both Systems to change independently without affecting the operation of the other.

6.3.10.4 Operation of the Denormalization Functional Block

Data and information Denormalization translate data and information from one or more internal forms to a single standardized form. The denormalization process includes the use of pre-defined data structures, where data is converted to a standard format that uses a standard encoding. In the case of ENI, this is facilitated using its set of models. In ENI, models represent objects as templates; this includes defining a set of mandatory and optional attributes, operations, relationships, and other standard features. Metadata may be used to help describe and/or prescribe this process.

In order to do this, accumulated knowledge from other ENI Functional Blocks (included predefined model information) shall be made available to the Denormalization Functional Block, as shown in Figure 6-36. This is essentially the inverse of the example provided in clause 6.3.3.4. The functional block diagram shown in Figure 6-36 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of Denormalization. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

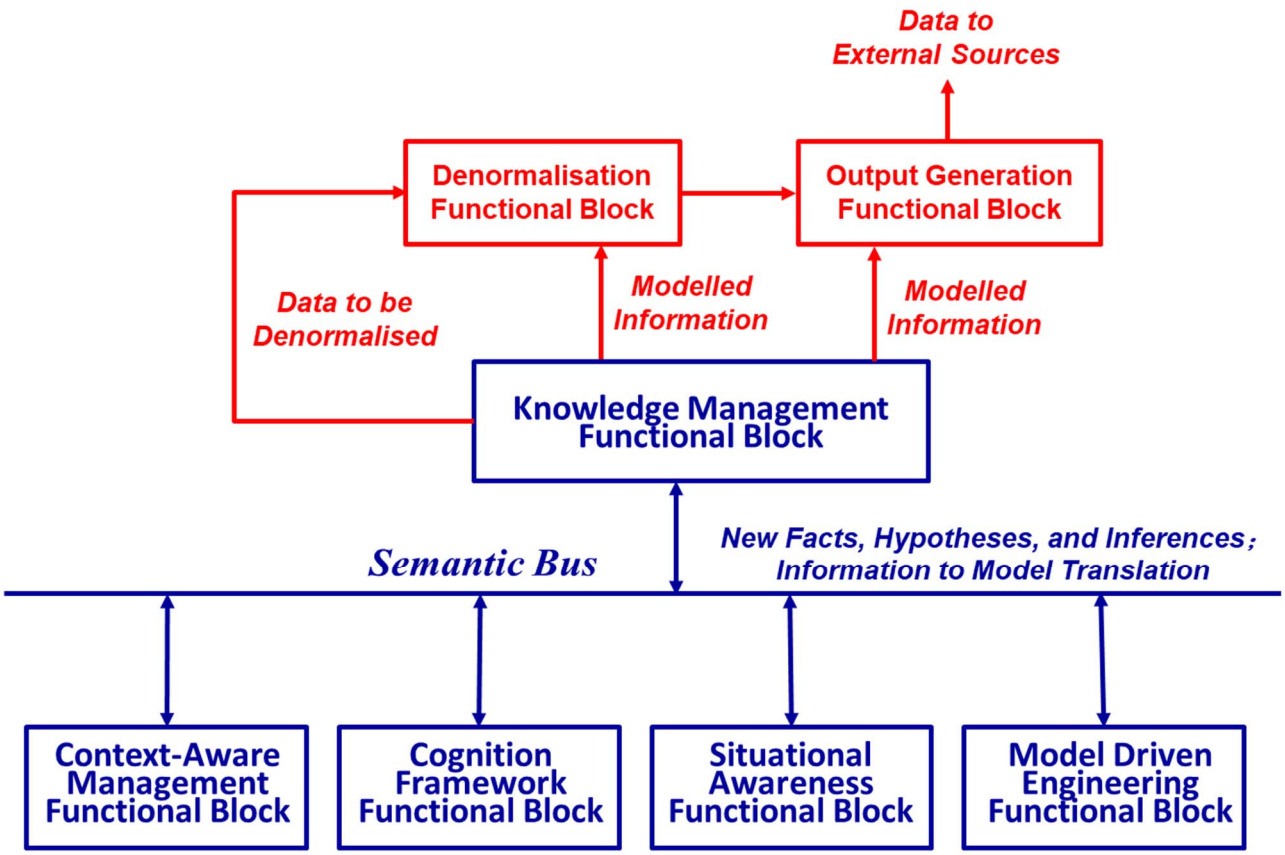

Figure 6-36: Data Denormalization Operation

Applicable data and information are not sent directly to the Denormalization Functional Block. Rather, they are first filtered by the Knowledge Management Functional Block, which also supplies modelled data and information to aid the Denormalization Functional Block in denormalizing the data and information that it receives. After the denormalization process is complete, it sends its result to the Output Generation Functional Block.

6.3.11 Output Generation Functional Block

6.3.11.1 Introduction

This clause describes the processes associated with generating recommendations and commands that are specific to the format(s) of a particular Assisted System (and/or its Designated Entity). Recommendations and commands are received from the Data Denormalization Functional Block. The Output Generation Functional Block then translates the data into a specific format (or set of formats) required by the Assisted System (and/or its Designated Entity). This is essentially the inverse of the example provided in clause 6.3.2.4.

These characteristics give rise to the following requirements:

- ENI shall provide the ability to output recommendations and commands in one or more formats specified by the Assisted System (and/or its Designated Entity). This may be efficiently implemented using a multi-agent architecture.
- ENI shall provide the ability to output recommendations and commands on-demand as well as in batch, push, or other agreed upon communication modes.
- Output Generation may be realized as a Functional Block that is separate from the Data Denormalization Functional Block. This adheres to the Single Responsibility Principle [i.9], and enables a more scalable and robust system to be designed and built.

6.3.11.2 Motivation

The motivation for this Functional Block is similar to that of the Denormalization Functional Block (see clause 6.3.10.2). It is important that the Assisted System (and/or its Designated Entity) are decoupled from the ENI System; this enables each to change independent of the other. In addition, it is important that the Assisted System (and/or its Designated Entity) need not have to change in order to benefit from interacting with the ENI System. Hence, the function of the Output Generation Functional Block is to translate recommendations and commands from the ENI System to the form(s) specified by the Assisted System (and/or its Designated Entity).

6.3.11.3 Function of the Output Generation Functional Block

As stated previously, the ENI System may use one or more internal formats to represent, analyse, and process data and information. The ENI System shall not expect the Assisted System (and/or its Designated Entity) to be able to understand the internal format(s) of ENI. Data Denormalization translates information and data in one or more internal formats used by the ENI System to a standardized form; this facilitates the generation of recommendations and commands in a form that the Assisted System (and/or its Designated Entity) is able to understand.

6.3.11.4 Operation of the Output Generation Functional Block

6.3.11.4.1 Introduction

Figure 6-37 shows one possible functional block diagram of an agent-based system for generating outputs. The functional block diagram shown in Figure 6-37 does not prescribe an implementation. Rather, it describes the high-level Functional Blocks that are needed to implement the needs of output generation. Different implementations may need to add other Functional Blocks to meet their particular operational requirements.

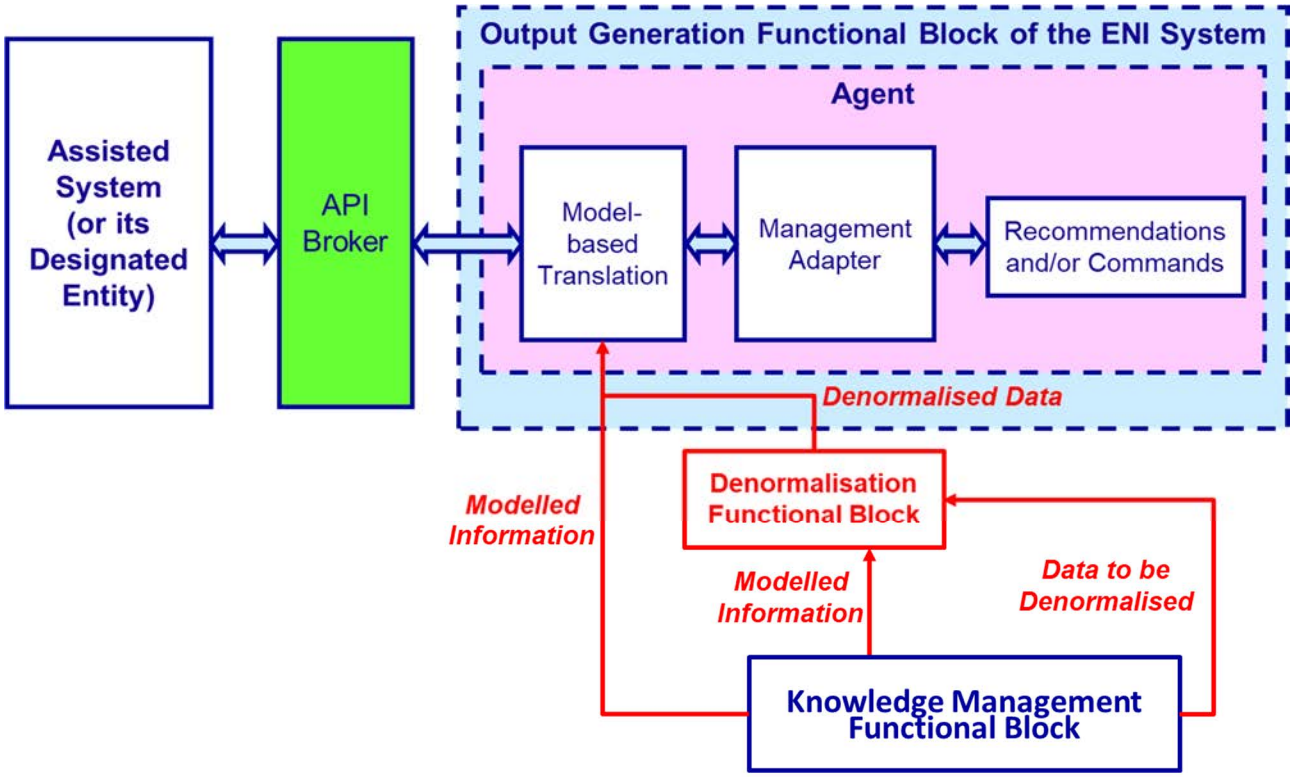

Figure 6-37: Output Generation Operation

The Output Generation Functional Block of the ENI System communicates with the Assisted System (and/or its Designated Entity). This communication may use an API Broker to translate between the APIs of the ENI System and the APIs of the Assisted System (and/or its Designated Entity). The Output Generation Functional Block is shown as an agent-based architecture. This architecture hides the native interfaces of both Assisted System (and/or its Designated Entity) as well as each managed entity of the Assisted System that is being managed by the ENI System.

In recommendation mode, communication from the Output Generation Functional Block shall go to the Assisted System (and/or its Designated Entity); it shall not be sent to managed entities of the Assisted System.

In management mode, communication from the Output Generation Functional Block shall go to the Assisted System (and/or its Designated Entity); it may also be sent to managed entities of the Assisted System, with the approval of the Assisted System (and/or its Designated Entity).

The model-based translation shown in the Agent is separate from the MDE Functional Block. The purpose of the agent's model-based translation is to use modelled information to ensure that the recommendations and commands are in a format that can be understood by the Assisted System (and/or its Designated Entity).

The advantages of this type of architecture include:

- Flexibility: the implementation may change (e.g. agent functionality may change) without affecting the API (built from more abstract levels) of either the ENI System or the Assisted System.
- Extensibility: new agents may be added or removed to suit the needs of the application.
- Manageability: instead of having to manage individual agents, the management focus is abstracted to managing roles.

6.3.11.4.2 Treating Output Generation as the Inverse of Normalization

NOTE: This is for further study in Release 4 (see clause 9).

6.4 API Broker

6.4.1 Introduction

The API Broker shall be implemented as a trusted entity. Only trusted entities can see, let alone interact with, the API Broker. The API Broker shall be a mandatory component, which implies that the API Broker shall be compliant with all ENI External Reference Points.

The API Broker has four main functions:

1) serve as an API gateway (i.e. an entity that can translate between different APIs);

2) provide programmable tools that enable the developer to design and realize APIs for the ENI API Broker;

3) provide the developer with the ability to change the mode of operation of the ENI API Broker (i.e. development, runtime, and test modes);

4) provide API management.

Management of APIs shall include, but not be limited to, the following functions: authentication, authorization, accounting, auditing, and related functionality.

The design of the API Broker should facilitate external analysis, including but not limited to sending API statistics to an analytics module (e.g. to understand API performance) as well as to an ANN (e.g. to use different algorithms to assess if the API Broker is being attacked; examples include malicious URL detection and other types of intrusion detection).

6.4.2 Motivation

The motivation for using an API Broker is threefold:

- The use of an API Broker enables the continuing development of the ENI System architecture to proceed independently of any specific requirements of interacting with external entities.
- The use of an API Broker provides a more scalable and extensible solution, as it facilitates the use of generic (e.g. RESTful) technologies as well as custom plug-ins to meet the needs of communication with different external entities.
- The use of an API Broker enables advanced solutions, such as API composition, to be used.

In addition, it is important that the Assisted System (and/or its Designated Entity) need not have to change in order to benefit from interacting with the ENI System. An API Broker provides the best solution, based on accepted industry practice, for enabling communication with external entities independent of their structure and function.

6.4.3 Function of the API Broker

The function of the API Broker is to interact with external entities using APIs. More specifically, the API Broker shall:

- accept incoming APIs transmitted through an ENI External Reference Point and route them to the appropriate ENI Functional Block(s);
- accept outgoing ENI APIs transmitted through an ENI External Reference Point and route them to the appropriate external entity;
- convert protocols used by external entities to protocols used by the ENI System, and vice-versa;
- manage different versions of the same API.

NOTE: An API Broker can also be used for API composition. This item is for further study in Release 4 (see clause 9).

APIs that are received from external entities by the API Broker may include metadata that aids the API Broker in properly routing the API.

ENI APIs that are sent to external entities by the API Broker should include metadata that aids the API Broker in properly routing the API.

The API Broker shall use the following External Reference Points to send ENI APIs to, and receive APIs from, an external entity:

- $E_{eni\text{-}api\text{-}in}$ defines APIs and associated information and/or metadata sent from an external entity to the ENI System.
- $E_{eni\text{-}api\text{-}out}$ defines ENI APIs and associated information and/or metadata sent from the ENI System to an external entity.
- $E_{eni\text{-}api\text{-}dev}$ defines ENI APIs and associated information and/or metadata for a developer to create and manage the Functional Blocks that make up the ENI API Broker.
- $E_{eni\text{-}api\text{-}run}$ defines ENI APIs and associated information and/or metadata sent from the ENI System to an external entity.
- $E_{eni\text{-}api\text{-}dmg}$ contains generic operational, administrative, management, and performance data and associated information and/or metadata used by developers to manage the ENI Broker.
- $E_{eni\text{-}api\text{-}emg}$ contains generic operational, administrative, management, and performance data and associated information and/or metadata used by an external entity to manage the ENI Broker. Such information and data may be a subset of the information and data contained in the $E_{eni\text{-}api\text{-}dmg}$ Reference Point.

The API Broker shall interact with all 22 ENI External Reference Points to transmit received APIs to, and send ENI APIs from, ENI Functional Blocks. See clause 7.3 for a description of each of the External Reference Points.

See clause 7.5 for a description of the six APIs used in the API Broker.

6.4.4 Operation of the API Broker

The purpose of the ENI API Broker is to function as a common intermediary between external entities and an ENI System that communicate using APIs. This is shown in Figure 6-38.

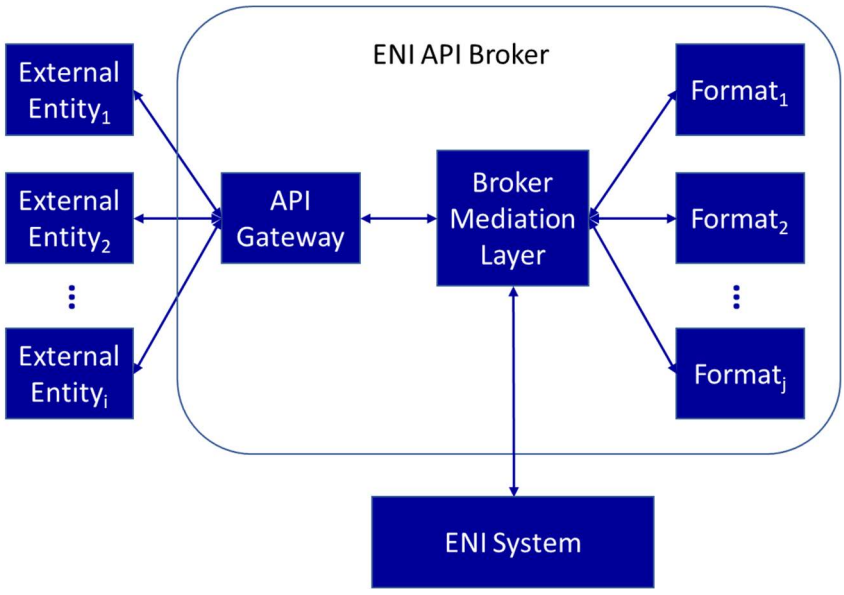

Figure 6-38: Function of the ENI API Broker

The API Broker consists of two distinct parts, the API Gateway and the Broker Mediation Layer. This is an application of the Façade pattern [i.5]. The Façade pattern creates an object that serves as a front-facing interface masking more complex interfaces and protocols that are used by other parts of the system. This pattern thus creates a consistent view from the perspective of the API consumer. An API Facade provides future compatibility, since it insulates clients from any changes to the functionality of the underlying system that the client is accessing. From the API consumer perspective, nothing will have changed.

The API Gateway is the entry point for external entities to request API-based services of an ENI System (and similarly, the exit point for an ENI System to request ENI API-based services of an external system). As such, the API Gateway is responsible for enabling each client (including an ENI System) to see and use only those services that they need. The API Gateway may provide a generic or a tailored API to a client to route requests, transform protocols, and implement shared logic like validation, authentication and rate-limiting. Tailored responses are supplied by plug-ins; this enables different responses for the same task to suit client-specific needs.

The Broker Mediation Layer is responsible for integrating client API requests with the underlying system (e.g. integrating API requests from the Assisted System to an ENI System, or vice-versa). It transforms API requests into formats acceptable for different systems (e.g. XML, JSON, YANG), delivers the request, and then processes and transforms the response of the underlying system into response and data formats the API has promised to return to the API consumers. This layer can perform tasks ranging from simple data manipulations, such as converting a response from XML to JSON, to much more complex operations where an application-specific client is required to run in order to connect to existing systems.

6.5 Communication Between Functional Blocks

6.5.1 Introduction

ENI uses a set of patterns to communicate between its own internal Functional Blocks as well as with external systems. These patterns are documented in the following clauses.

Enterprise integration is the task of making separate applications work together to produce a unified set of functionality. Some applications may be custom developed in-house while others are bought from third-party vendors. The applications probably run on multiple computers, which may represent multiple platforms, and may be geographically dispersed. Some of the applications may be run outside of the enterprise by business partners or customers. Some applications may need to be integrated even though they were not designed for integration and cannot be changed. These issues and others like them are what make application integration difficult. This clause will explore the options available for application integration.

6.5.2 Common Communication Requirements

Communication patterns used within ENI shall meet the following criteria:

- An ENI System shall use a messaging system for communication and interaction with other ENI Functional Blocks.
- An ENI System should use a Semantic Bus to implement its messaging system.
- Each ENI Functional Block shall be *loosely coupled*, so that each ENI Functional Block can evolve independently of other ENI Functional Blocks.
- Each ENI Functional Block should use the Semantic Bus for communication and interaction with other ENI Functional Blocks.
- ENI External and Internal Functional Blocks should exchange data in a timely manner.
- ENI External and Internal Functional Blocks should enable asynchronous processing.
- Each ENI Internal Functional Block shall use a common data format.
- Each ENI Internal Functional Block shall avoid duplicating the functionality and behaviour of another ENI Internal Functional Block.

The components of a loosely coupled systems are independent, enabling changes in one module to be performed without affecting other modules. For example, the code in one component should not be reused by other components.

Timeliness of exchanging data is important and should be minimized. Otherwise, the data and information used in making a decision may become irrelevant. All data exchange between Internal Functional Blocks should use either a messaging bus or another mechanism to populate information and data as quickly as possible, such as a shared database. This enables the appropriate ENI Internal Functional Blocks to decide if the data and information is useful in their own contexts. Similarly, different ENI Functional Blocks should be able to complete their tasks asynchronously, unless there are constraints that demand synchronous task completion.

The ENI Internal Functional Blocks shall use normalized input data and information to avoid complex transformation processing. For input data and information coming from external sources, the normalized data is created by the Data Normalization Functional Block and sent using the $I_{norm-sem}$ Internal Reference Point (see clause 7.7.2); otherwise, such data and information is exchanged using an appropriate Internal Reference Point that connects the Internal Functional Block to the Semantic Bus (see clauses 7.7.3 through 7.7.8). For output data and information, if the output data is to be sent to an external source, then the Internal Reference Point $I_{sem-denorm}$ (see clause 7.7.9) is used to send the normalized data to the Denormalization Functional Block, where it will be converted to an appropriate form for external consumption); otherwise, the data is exchanged using an appropriate Internal Reference Point that connects the Internal Functional Block to the Semantic Bus (see clauses 7.7.3 through 7.7.8).

6.5.3 Recommended Communication Patterns to be Used Within ENI

6.5.3.1 Introduction

Communication between ENI Functional Blocks should use appropriate ENI Reference Points and the Semantic Bus whenever possible. Communication patterns follow standard enterprise messaging exchange patterns (e.g. [i.34]).
A messaging exchange pattern is a software design pattern that describes how two different parts of a messaging system connect and communicate with each other. For example, the Request-Reply design pattern enables a sender to request information (via a message) and ensures that the receiver replies to that message. As another example, the Pipes and Filters design pattern connects a set of functions (the filters) to a stream of data (using pipes).

6.5.3.2 Remote Procedure Calls and Remote Method Invocations

Remote Procedure Call (RPC) and Remote Method Invocation (RMI) are two mechanisms that allow the user to invoke or call processes that will run on a different component from the one that is currently being used. The main difference is that RPC is an imperative approach that calls specific functions, while RMI uses a reference to a method of an object that is being invoked.

Both approaches isolate the sender from the receiver. However, both approaches have two disadvantages. First, both impose a potentially brittle point-to-point communication mechanism between the sender and the receiver. Second, such mechanisms tend to be vendor-specific and not interoperable. As such, this type of mechanism need not be used as a general mechanism.

Therefore, these and similar mechanisms should only be used for exclusive (i.e. point-to-point) communication between two components with well-defined interfaces whose semantics are known to and understood by each component.

6.5.3.3 Batch File Exchange

Files are an efficient means of transferring data. It provides physical decoupling between different system components, and may be stored for later use. However, files themselves are rarely efficient, especially if only changes in previous data are of interest. More importantly, if data formats change, then data integrity problems are caused.

Therefore, batch file exchange should be restricted to content whose data format do not change, and the content should be highly cohesive (i.e. all content should be related to each other). For example, a data model update could use this approach. However, if there are only small updates, then other approaches, such as federated learning, are both more secure and more efficient.

6.5.3.4 Shared Database

The motivation for using a shared database is to eliminate unnecessary data duplication and replication. It takes advantage of inherent tools of a database to notify clients of content changes. Databases can be distributed and scaled both horizontally and vertically. The key deficiency is that this system *exchanges data*, as opposed to *responding to changes in goals to be obtained.*

6.5.3.5 Messaging

6.5.3.5.1 Introduction

In general, the preferred communication mechanism for communication between ENI Functional Blocks should be to use a messaging system. This is because such systems provide asynchronous and reliable data transfer without requiring knowledge of specific publishers and subscribers. A messaging system allows loosely coupled communication to serve as an intermediate layer between producers and consumers. The ENI Semantic Bus (see clause 6.3.4.4.4) is similar in functionality to the Semantic Service Bus described in [i.30], but has additional functionality. This enables the ENI Semantic Bus to use an event-driven architecture for registering, discovering, persisting, processing, and routing messages. A message may contain data, information, metadata, and/or policies.

The ENI Semantic Bus shall provide an automatic semantic transformation of services, via event-driven messages, which guarantees interoperability between heterogeneous Functional Blocks. This provides *loose coupling* of the Functional Blocks of ENI, ensuring that the evolution of each ENI Functional Block is independent of all other ENI Functional Blocks.

6.5.3.5.2 Common Requirements of Messaging Systems

All types of messaging systems used by ENI shall support the following semantics.

Publishers: Publishers need not know the identity of a subscriber. Publishers need not know which subscribers are subscribed to which messages.

Subscribers: Subscribers need not know the identity of a publisher. Subscribers need not know which publishers created which messages.

Time Decoupling: Publishers and subscribers need not be active at the same time.

Asynchronicity: Subscribers shall be able to retrieve a message any time after it has been published (subject to persistence characteristics).

Guaranteed Delivery: Each message sent by a publisher shall be guaranteed to be delivered to all subscribers.

No Duplicates: Each message sent by a publisher shall be guaranteed to be delivered to all subscribers.

Content Subscription: An ENI messaging system shall support content-based subscription.

Topic Subscription: An ENI messaging system shall support topic-based subscription.

Semantic Subscription: An ENI messaging system should support semantic-based subscription.

Storage Persistence: Each message should be stored persistently.

NOTE 1: An ENI Messaging System may provide a configurable choice for which types of messages are stored persistently.

Logging: An ENI messaging system should support the logging of messages.

NOTE 2: This feature facilitates auditing, guaranteed delivery, and message replay.

Request-Reply: An ENI Messaging System may support the request-reply pattern [i.34].

Message Ordering: An ENI Messaging System may support message ordering (i.e. messages arrive at a subscriber in the same order as sent by the producer).

The above requirements describe three types of subscriptions. A content-based subscription instructs the messaging system to deliver messages if the attributes and/or content of those messages matches constraints defined by the subscriber. Similarly, a topic-based subscription assigns a message to one or more topics by the publisher; the subscriber then will receive all messages for the topics that they subscribe to.

A semantic-based subscription [i.30] matches messages based on the *meaning* of the message (in addition to content and topic). The meaning could be simplistically approximated using metadata, but should be defined using either ontologies or linguistics. In contrast to classical subscription mechanisms, a semantic-based subscription does not require publishers to explicitly declare the topic a message belongs to. Instead, the subscriber defines 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. This is another reason why it is important to have a single policy information model that enables different policies to be represented, so that interoperability between different types of policies (e.g. imperative, declarative, and intent) is achievable. See clause 6.3.9 for more information.

6.5.3.5.3 Messaging Functionality

Figure 6-39 shows a functional block diagram of a messaging system suitable for ENI, such as the ENI Semantic Bus.

NOTE: Each of the blocks shown in Figure 6-39 is part of a larger set of patterns. This is for further study in the next release of the present document.

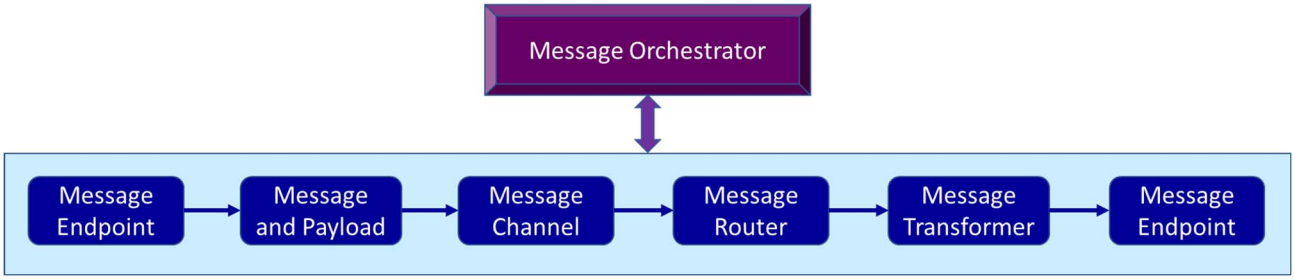

Figure 6-39: Functional Block Diagram of an ENI Messaging System

A Message Endpoint is the connection from an ENI Functional Block to the ENI Semantic Bus. It may be a simple component, a gateway, or have more complex functionality, as explained in [i.30].

A message is a special type of datatype that is sent through a message channel. The message has attributes that can be read by the messaging system to put it in the correct message channel, as well as an optional payload. The attributes and metadata of the message and payload are read, normalized, and then used to help select the appropriate message channel. For example, simple semantic differences, such as the name of an attribute (e.g. "employeeID" vs. "fteID" vs. "ID_employee", are detected and normalized to the consensual name expected in the data model. When such changes are made, metadata is appended to the message specifying the nature of the change and a timestamp of when the change was made.

A message channel is a construct that messages are written into and subsequently removed from; the removed message is then sent to a set of destinations. There are different channel types, such as a publish-subscribe channel and a point-to-point channel.

A message transformer changes the data format and/or content to conform with a desired output data model. This is similar to the Adaptor pattern of [i.5]; it enables two incompatible interfaces to interoperate by converting the interface of an object to the interface expected by the client using that object.

6.5.4 Recommended Communication Patterns to be Used Between ENI and External Systems

NOTE: This is for further study in Release 4 (see clause 9).

6.6 Security Considerations

NOTE: This is for further study in Release 4 (see clause 9).

7 Reference Points

7.1 Introduction

This clause uses the definitions of Reference Points from clause 4.4.6 to define the set of Reference Points used by ENI for communication between its different components as well as between it and the Assisted System.

ENI only standardizes communications that occur over a Reference Point. This enables ENI External and Internal Reference Points to differentiate between data, control, management, and other specific types of communication information. This differentiation is denoted by suffixes for each Reference Point, as follows:

-dat: denotes a data (input-only) interface

-cmd: denotes a management (output-only) interface that sends recommendations and/or commands

-pol: denotes a management bi-directional interface that sends or receives ENI policies

NOTE 1: Policy APIs and interfaces use a separate policy reference point that is separate from other management concepts (e.g. modelled and contextual data). This enables external Systems to more easily communicate policies with ENI.

-ctx: denotes a data only bi-directional interface that sends context- and situation-aware data

NOTE 2: In a distributed ENI system, context and situational data may need to be exchanged between different ENI systems and Functional Blocks.

-oth: denotes a data only bi-directional interface that is used to send and receive other types of application-specific data that is not part of any of these other categories

-kno: denotes a data only bi-directional interface that sends model and other semantic data

EXAMPLE: In a distributed ENI system, this interface is used to send model updates between ENI system Functional Blocks; another example is creating knowledge from data and information as described in clause 6.3.4.4.3 of the present document.

-api: denotes separate APIs that send or receive information and data between external systems and the ENI API Broker

7.2 External Reference Point Overview

Figure 7-1 shows an overview of the External Reference Points that provide internally facing External Reference Points (i.e. External Reference Points that communicate data and information from external entities to ENI). Figure 7-1 shows ENI Reference Points for a single domain only.

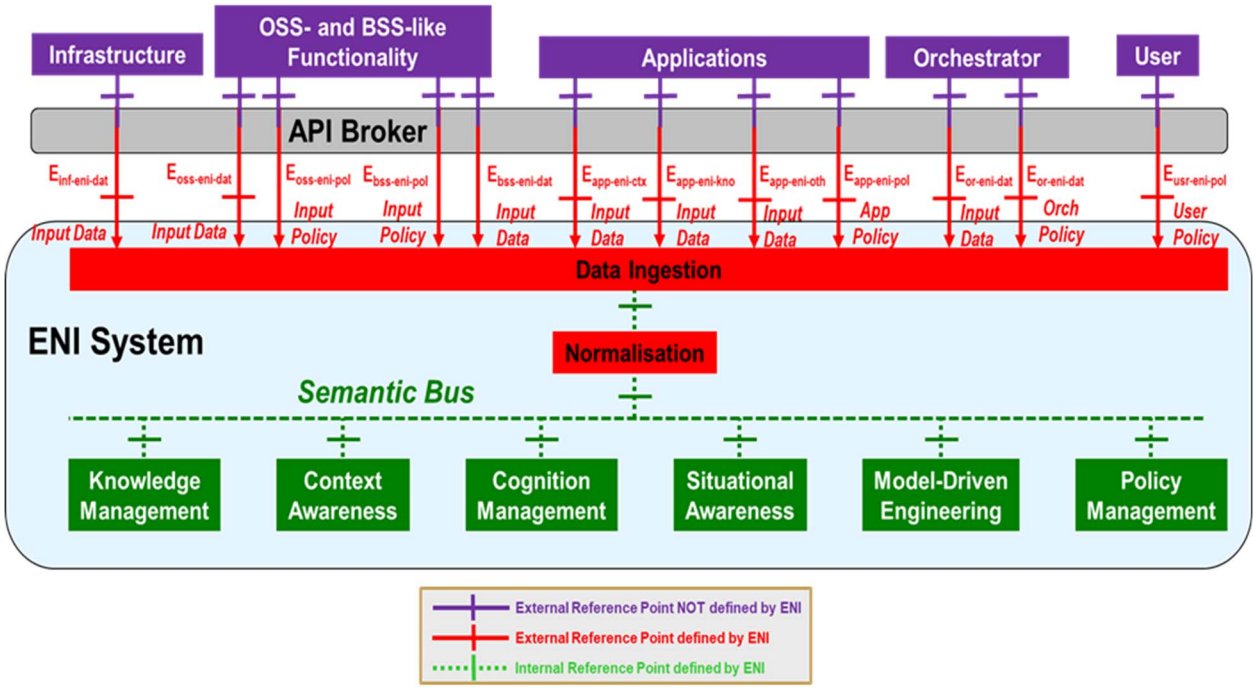

Figure 7-1: Overview of the ENI Internal Facing External Reference Points

Figure 7-2 shows an overview of the External Reference Points that provide externally facing External Reference Points (i.e. External Reference Points that communicate data and information from ENI to external entities.

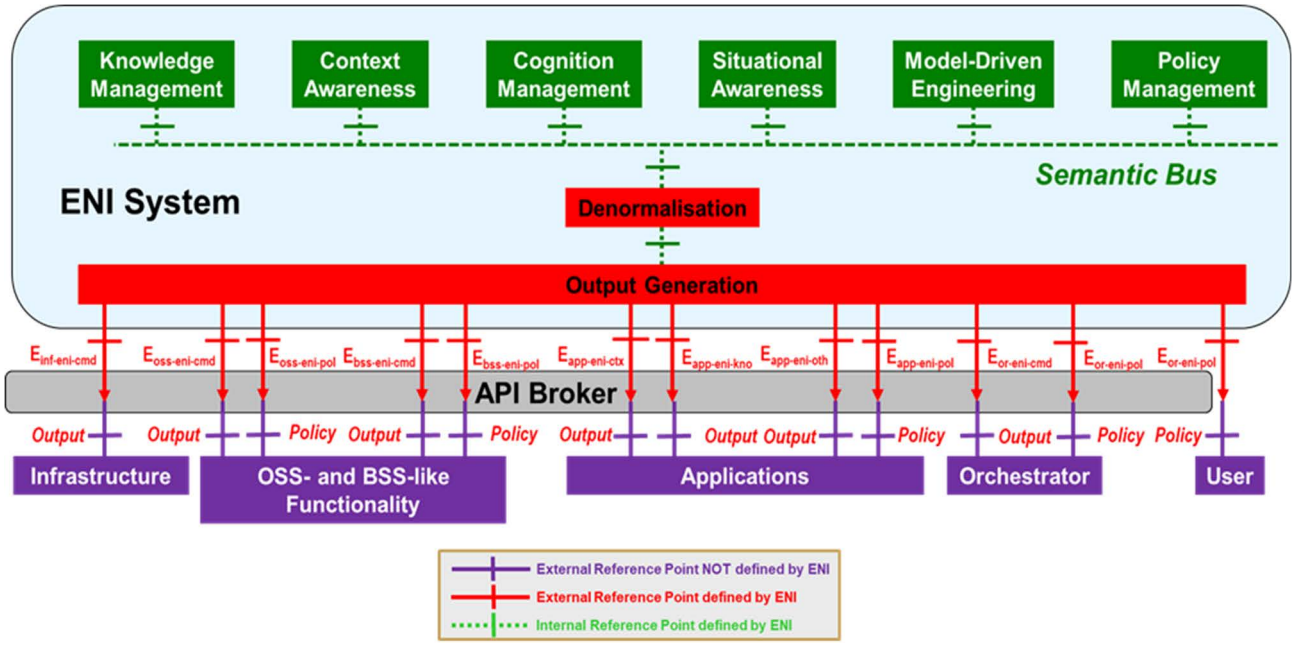

Figure 7-2: Overview of the ENI External Facing External Reference Points

Figure 7-2 shows ENI Reference Points for a single domain only.

NOTE: Currently, it is assumed that all applications are trusted. Work on non-trusted applications will start in a future Release.

Table 7-1a provides brief descriptions of the External Reference Points of an ENI System.

Table 7-1a: ENI External Reference Point Overview (not Including ENI API Broker)

Name	Brief Definition	Interface Functions
$E_{oss\text{-}eni\text{-}dat}$	Defines data and information sent from the OSS-like functionality to the ENI System; these data and information may be acknowledged by the ENI System.	Any data from the OSS-like functionality that the ENI System can use.
$E_{oss\text{-}eni\text{-}cmd}$	Defines recommendations and/or commands sent from ENI to the OSS-like functionality; these recommendations and commands shall be acknowledged by the Assisted System.	The type of data sent by the ENI System depends on the mode that it is operating in.
$E_{oss\text{-}eni\text{-}pol}$	Defines Policies and associated information and/or metadata exchanged between the OSS-like Functionality and the ENI System that control behaviour (including services and resources) for the Assisted System. Policies and information received by the ENI System shall be acknowledged by the ENI System. Similarly, Policies and information received by the Application shall be acknowledged by the Application.	These may be defined by the operator or the OSS-like Functionality. Inputs may be in a controlled language or DSL for Intent.
$E_{app\text{-}eni\text{-}ctx}$	Defines situation- and/or context-aware data and information exchanged between applications and the ENI System. Data and information received by the ENI System may be acknowledged by the ENI System. Similarly, data and information received by the Application may be acknowledged by the Application.	Contains input information for the ENI System to determine context and output context result data from the ENI System for Applications.
$E_{app\text{-}eni\text{-}oth}$	Defines generic application data exchanged between Applications and the ENI System, which is neither situation- or context-aware data and also is not model or knowledge information. Data and information received by the ENI System may be acknowledged by the ENI System. Similarly, data and information received by the Application may be acknowledged by the Application.	Contains generic application data that may be useful to the ENI System, and generic data from the ENI System that may be useful to applications.
$E_{app\text{-}eni\text{-}kmo}$	Defines model and/or knowledge information and acknowledgements exchanged between Applications and the ENI System. Data and information received by the ENI System may be acknowledged by the ENI System. Similarly, data and information received by the Application may be acknowledged by the Application.	Limited to just DIKW and model information. Requires additional security.
$E_{app\text{-}eni\text{-}pol}$	Defines Policies and associated information and/or metadata exchanged between the BSS-like functionality and the ENI System that control behaviour (including services and resources) for the Assisted System. Policies and information received by the ENI System shall be acknowledged by the ENI System. Similarly, Policies and information received by the Application shall be acknowledged by the Application.	These are defined by the application. Inputs may be in a controlled language or DSL for Intent.
$E_{bss\text{-}eni\text{-}dat}$	Defines data and information sent by the BSS-like functionality to the ENI System; these data and information may be acknowledged by the ENI System.	Any data from the BSS-like functionality that the ENI System can use.
$E_{bss\text{-}eni\text{-}cmd}$	Defines data and acknowledgements sent from the ENI System to the BSS-like functionality; recommendations and commands sent from the ENI System shall be acknowledged by the BSS-like functionality.	The type of data sent by the ENI System depends on the mode that it is operating in.
$E_{bss\text{-}eni\text{-}pol}$	Defines Policies and associated information and/or metadata exchanged between the BSS-like functionality and the ENI System that control behaviour (including services and resources) for the Assisted System. Policies and information received by the ENI System shall be acknowledged by the ENI System. Similarly, Policies and information received by the Application shall be acknowledged by the Application.	These may be defined by the operator or the BSS-like functionality. Inputs may be in a controlled language or DSL for Intent.
$E_{usr\text{-}eni\text{-}pol}$	Defines Policies and associated information and/or metadata exchanged between Applications and the ENI System that control behaviour (including services and resources) for a user (or an agent acting on behalf of the user). Policies and information received by the ENI System shall be acknowledged by the ENI System. Similarly, Policies and information received by the Application shall be acknowledged by the Application.	These may be defined by the user or the operator. Inputs may be in a controlled language for Intent.
$E_{or\text{-}eni\text{-}dat}$	Defines data and acknowledgements sent from the Orchestrator to the ENI System; data sent from the Orchestrator may be acknowledged by the ENI System.	Any data from the Orchestrator that the ENI System can use
$E_{or\text{-}eni\text{-}cmd}$	Defines commands and acknowledgements sent from the ENI System to the Orchestrator; recommendations and commands sent from the ENI System shall be acknowledged by the Orchestrator.	The type of data sent by the ENI System depends on the mode that it is operating in.

Name	Brief Definition	Interface Functions
$E_{or\text{-}eni\text{-}pol}$	Defines Policies and associated information and/or metadata exchanged between the Orchestrator and the ENI System that control behaviour (including services and resources) for the Assisted System. Policies and information received by the ENI System shall be acknowledged by the ENI System. Similarly, Policies and information received by the Application shall be acknowledged by the Application.	These may be defined by the operator or the OSSOrchestrator. Inputs may be in a controlled language or DSL for Intent.
$E_{inf\text{-}eni\text{-}dat}$	Defines data and acknowledgements sent from the infrastructure to the ENI System; data sent to the ENI System may be acknowledged by the ENI System.	Any data from the Orchestrator that the ENI System can use (e.g. log files, telemetry, alarms).
$E_{inf\text{-}eni\text{-}cmd}$	Defines recommendations and/or commands sent from the ENI System to the infrastructure; recommendations and commands sent from the ENI System shall be acknowledged by the Assisted System.	The type of data sent by the ENI System depends on the mode that it is operating in.

Figure 7-3 shows the six External Reference Points dedicated to ENI API Broker interaction.

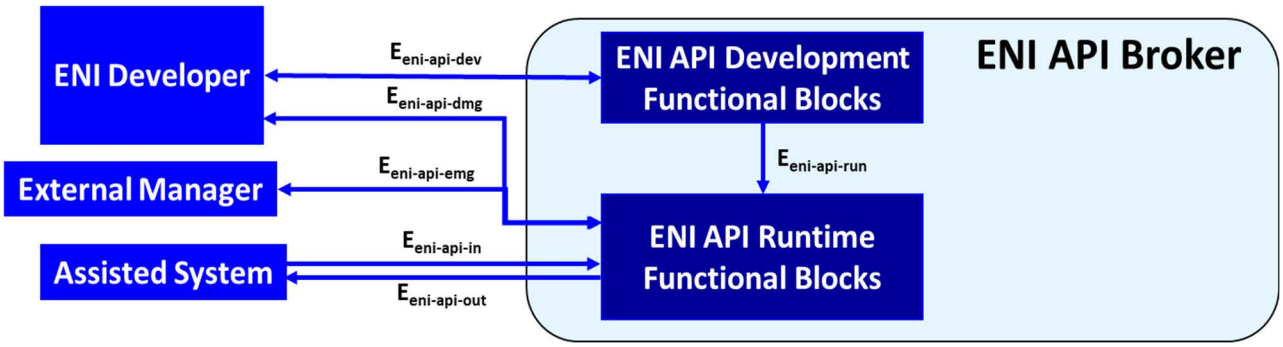

Figure 7-3: Overview of the ENI External Reference Points for the ENI API Broker

Table 7-1b provides brief descriptions of the External Reference Points of an ENI System. Refer to clause 7.5.1, Figure 7-4, for a picture of the ENI API Broker External Reference Points.

Table 7-1b: ENI External Reference Point Overview (just the ENI API Broker)

Name	Brief Definition	Interface Functions
$E_{eni\text{-}api\text{-}in}$	Define recommendations and/or commands and associated information and/or metadata sent by an external entity to the ENI System. Data and information received by the ENI Broker may be acknowledged by the ENI Broker. Similarly, data and information received by an external entity may be acknowledged by that external entity.	Sends input recommendations and commands from an external entity to the ENI API Broker.
$E_{eni\text{-}api\text{-}out}$	Defines ENI APIs and associated information and/or metadata sent by the ENI System to an external entity. Data and information received by the ENI Broker may be acknowledged by the ENI Broker. Similarly, data and information received by an external entity may be acknowledged by that external entity.	Sends output recommendations and commands from the ENI API Broker to an external entity.
$E_{eni\text{-}api\text{-}dev}$	Defines ENI APIs and associated information and/or metadata for a developer to create and manage the Functional Blocks that make up the ENI API Broker. Data and information received by the ENI Broker may be acknowledged by the ENI Broker.	Management of an ENI API Broker by a developer.
$E_{eni\text{-}api\text{-}run}$	Control the operation of the ENI API Broker. This includes putting the ENI API Broker into different operational states, selectively enable and disable different functionality, and for catching exceptions and other runtime errors produced by the ENI API Broker during its operation. All operational and control messages generated by the ENI API Broker should be sent to the ENI System. The ENI System should acknowledge all such operational and control messages.	Control the operation of the ENI API Broker.
$E_{eni\text{-}api\text{-}dmg}$	Defines generic operational, administrative, management, and performance data and associated information and/or metadata exchanged between developers and the ENI API Broker. Data and information received by the ENI API Broker may be acknowledged by the ENI API Broker.	Contains generic operational, administrative, management, and performance data and associated information and/or metadata used by developers to manage the ENI Broker.

Name	Brief Definition	Interface Functions
$E_{eni\text{-}api\text{-}emg}$	Defines generic operational, administrative, management, and performance data and associated information and/or metadata exchanged between developers and the ENI API Broker. Data and information received by the ENI API Broker may be acknowledged by the ENI API Broker. Such information and data may be a subset of the information and data contained in the $E_{eni\text{-}api\text{-}dmg}$ Reference Point.	Contains generic operational, administrative, management, and performance data and associated information and/or metadata used by an external entity to manage the ENI Broker.

7.3 External Reference Point Definitions

7.3.1 Reference Point $E_{oss\text{-}eni\text{-}dat}$

This External Reference Point is used by the OSS-like Functionality of the Assisted System to provide data and information for the ENI System to ingest and process. This can include data, information, and metadata. Policies, and policy-related information, shall not be sent over this External Reference Point. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a uni-directional External Reference Point, meaning that data for processing shall only flow from the OSS-like Functionality to the ENI System.

The ENI System shall ingest the data and normalize it. If it is not possible to normalize the data, the ENI System shall report this to the OSS-like functionality of the Assisted System and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.2 Reference Point $E_{oss\text{-}eni\text{-}cmd}$

This External Reference Point is used by the ENI System to send recommendations and/or commands, as well as acknowledgements, to the OSS-like functionality. Policies, and policy-related information, shall not be sent over this External Reference Point. Recommendations and commands sent from the ENI System shall be acknowledged by the OSS-like functionality. The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the OSS-like Functionality. This is a uni-directional External Reference Point, meaning that commands and recommendations for processing shall only flow from the ENI System to the OSS-like Functionality.

If any recommendations or commands are ambiguous or are not understood by the OSS-like Functionality, then the OSS-like Functionality shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.3 Reference Point $E_{oss\text{-}eni\text{-}pol}$

This External Reference Point is used to define Policies and associated information and/or metadata sent by the OSS-like Functionality to the ENI System and/or Policies sent by the ENI System to the OSS-like Functionality. Both the OSS-like Functionality and the ENI System may send any type of Policy that is mutually agreeable. This includes acknowledgements by the receiving entity in both cases. Both the OSS-like Functionality as well as the ENI System may include metadata to describe how the recommendations and/or commands are to be used by the OSS-like Functionality or the ENI System. This is a bi-directional External Reference Point, meaning that Policies may be sent to the ENI System by the OSS-like Functionality and/or Policies may be sent from the ENI System to the OSS-like Functionality.

The associated information includes policyRefPoint, which is used to indicate the specific External Reference Point (i.e. $E_{oss\text{-}eni\text{-}pol}$), so that the acknowledgement could be sent back using the appropriate External Reference Point. This is described in [i.33].

The metadata includes policyType, which is used to indicate the specific type of the policy (i.e. imperative policy, declarative policy and intent policy as defined in clause 6.3.9.3.2), so that the ENI system could perform appropriate actions as specified in clause 6.3.9.6.3. This is also described in [i.33].

If any Policy Rule (including any attached metadata) is ambiguous or is not understood by the entity receiving the Policy Rule, then the receiving entity shall report this to the entity sending the Policy Rule and ask for further information that defines the meaning and usage of the Policy Rule. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.4 Reference Point $E_{app\text{-}eni\text{-}ctx}$

This External Reference Point is used to define situation- and/or context-aware data and information and acknowledgements exchanged between applications and the ENI System. Policies, and policy-related information, shall not be sent over this External Reference Point. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. Similarly, the Application shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a bi-directional External Reference Point, meaning that context and situation data and information may flow from the ENI System to the Application or vice-versa.

Input context and situation aware data received by the ENI System shall be ingested and normalized. If it is not possible to normalize the data, the ENI System shall report this to the Application and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity.

The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the Application. If any recommendations or commands are ambiguous or are not understood by the Application, then the Application shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.5 Reference Point $E_{app\text{-}eni\text{-}oth}$

This External Reference Point is used to define generic application data and acknowledgements exchanged between applications and the ENI System that is neither situation- or context-aware data and also is not model or knowledge information. Policies, and policy-related information, shall not be sent over this External Reference Point. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. Similarly, the Application shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a bi-directional External Reference Point, meaning that other types of data and information may flow from the ENI System to the Application or vice-versa.

Input application data received by the ENI System shall be ingested and normalized. If it is not possible to normalize the data, the ENI System shall report this to the Application and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity.

The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the Application. If any recommendations or commands are ambiguous or are not understood by the Application, then the Application shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.6 Reference Point $E_{app\text{-}eni\text{-}kno}$

This External Reference Point defines model and/or knowledge information and acknowledgements exchanged between applications and the ENI System. Policies, and policy-related information, shall not be sent over this External Reference Point. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. Similarly, the Application shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a bi-directional External Reference Point, meaning that knowledge and model information may flow from the ENI System to the Application or vice-versa. This Reference Point shall contain extra security measures to protect the sensitive nature of these types of data and information.

NOTE: The data and information transmitted over this Reference Point require a special protocol to exchange their content. That protocol will be defined in Release 3 of the present document (see clause 9).

Input application data received by the ENI System shall be ingested and normalized. If it is not possible to normalize the data, the ENI System shall report this to the Application and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity.

The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the Application. If any recommendations or commands are ambiguous or are not understood by the Application, then the Application shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.7 Reference Point $E_{app\text{-}eni\text{-}pol}$

This External Reference Point is used to define Policies and associated information and/or metadata sent by Applications to the ENI System and/or Policies sent by the ENI System to Applications. Both Applications and the ENI System may send any type of Policy that is mutually agreeable. This includes acknowledgements by the receiving entity in both cases. Both Applications as well as the ENI System may include metadata to describe how the recommendations and/or commands are to be used by Applications or the ENI System. This is a bi-directional External Reference Point, meaning that Policies may be sent to the ENI System by the Application and/or Policies may be sent from the ENI System to the Application.

The associated information includes policyRefPoint, which is used to indicate the specific External Reference Point (i.e. $E_{app\text{-}eni\text{-}pol}$), so that the acknowledgement could be sent back using the appropriate External Reference Point. This is described in [i.33].

The metadata includes policyType, which is used to indicate the specific type of the policy (i.e. imperative policy, declarative policy and intent policy as defined in clause 6.3.9.3.2), so that the ENI system could perform appropriate actions as specified in clause 6.3.9.6.3. This is described in [i.33].

If any Policy Rule (including any attached metadata) is ambiguous or is not understood by the entity receiving the Policy Rule, then the receiving entity shall report this to the entity sending the Policy Rule and ask for further information that defines the meaning and usage of the Policy Rule. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.8 Reference Point $E_{bss\text{-}eni\text{-}dat}$

This External Reference Point is used by the BSS-like functionality to send data to the ENI System. Policies, and policy-related information, shall not be sent over this External Reference Point. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a uni-directional External Reference Point, meaning that data for processing shall only flow from the BSS-like functionality to the ENI System.

Input application data received by the ENI System shall be ingested and normalized. If it is not possible to normalize the data, then the ENI System shall report this to the BSS-like functionality and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.9 Reference Point $E_{bss\text{-}eni\text{-}cmd}$

This External Reference Point is used by Defines data and acknowledgements exchanged between the BSS-like functionality and the ENI System. Policies, and policy-related information, shall not be sent over this External Reference Point. Recommendations and commands sent from the ENI System shall be acknowledged by the BSS-like functionality. The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the BSS-like functionality. This is a uni-directional External Reference Point, meaning that commands and recommendations for processing shall only flow from the ENI System to the BSS-like functionality.

If any recommendations or commands are ambiguous or are not understood by the BSS-like functionality, then the BSS-like functionality shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.10 Reference Point $E_{bss\text{-}eni\text{-}pol}$

This External Reference Point is used to define Policies and associated information and/or metadata sent by the BSS-like functionality to the ENI System and/or Policies sent by the ENI System to the BSS-like functionality. Both the BSS-like functionality and the ENI System may send any type of Policy that is mutually agreeable. This includes acknowledgements by the receiving entity in both cases. Both the BSS-like functionality as well as the ENI System may include metadata to describe how the recommendations and/or commands are to be used by the BSS-like functionality or the ENI System. This is a bi-directional External Reference Point, meaning that Policies may be sent to the ENI System by the BSS-like functionality and/or Policies may be sent from the ENI System to the BSS-like functionality.

The associated information includes policyRefPoint, which is used to indicate the specific External Reference Point (i.e. $E_{bss\text{-}eni\text{-}pol}$), so that the acknowledgement could be sent back using the appropriate External Reference Point. This is described in [i.33].

The metadata includes policyType, which is used to indicate the specific type of the policy (i.e. imperative policy, declarative policy and intent policy as defined in clause 6.3.9.3.2), so that the ENI system could perform appropriate actions as specified in clause 6.3.9.6.3. This is described in [i.33].

If any Policy Rule (including any attached metadata) is ambiguous or is not understood by the entity receiving the Policy Rule, then the receiving entity shall report this to the entity sending the Policy Rule and ask for further information that defines the meaning and usage of the Policy Rule. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.11 Reference Point $E_{usr\text{-}eni\text{-}pol}$

This External Reference Point is used to define Policies and associated information and/or metadata exchanged between external entities and the ENI System that control behaviour (including services and resources) for a user (or an agent acting on behalf of the user). The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. Similarly, the external entity shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a bi-directional External Reference Point, meaning that data and Policies may flow from the ENI System to the external entity or vice-versa.

The associated information includes policyRefPoint, which is used to indicate the specific External Reference Point (i.e. $E_{usr\text{-}eni\text{-}pol}$), so that the acknowledgement could be sent back using the appropriate External Reference Point. This is described in [i.33].

The metadata includes policyType, which is used to indicate the specific type of the policy (i.e. imperative policy, declarative policy and intent policy as defined in clause 6.3.9.3.2), so that the ENI system could perform appropriate actions as specified in clause 6.3.9.6.3. This is described in [i.33].

Input Policies received by the ENI System shall be ingested and parsed. If the parsing produces any errors or warnings, the ENI System shall report this to the entity that sent the Policy. The ENI System may record the types of errors and/or warnings detected and later analyse them to determine if the grammar can be made clearer. The ENI System shall not continue work on the input Policy that did not parse completely until it has been fixed by the external authoring entity.

The ENI System may include metadata to describe how Policies are to be used by the external entity. If any Policies generated by the ENI System are not understood unambiguously by the external entity, then the external entity shall report this to the ENI System and ask for further information that defines the meaning and usage of the Policy. A negotiation of data and information contained in the Policy may need to occur to arrive at a final viable alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future Policies sent to this entity to facilitate future communications.

7.3.12 Reference Point $E_{or\text{-}eni\text{-}dat}$

This External Reference Point is used to define data and information sent from the Orchestrator to ENI. Policies, and policy-related information, shall not be sent over this External Reference Point. ENI shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. This is a uni-directional External Reference Point, meaning that data for processing shall only flow from the Orchestrator to ENI.

Input application data received by the ENI System shall be ingested and normalized. If it is not possible to normalize the data, the ENI System shall report this to the Orchestrator and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.3.13 Reference Point $E_{or\text{-}eni\text{-}cmd}$

This External Reference Point is used to define recommendations and commands sent from the ENI System to the Orchestrator. Policies, and policy-related information, shall not be sent over this External Reference Point. Recommendations and commands sent from the ENI System shall be acknowledged by the Orchestrator. The ENI System may include metadata to describe how the recommendations and/or commands are to be used by the Orchestrator. This is a uni-directional External Reference Point, meaning that commands and recommendations shall only flow from the ENI System to the Orchestrator.

The ENI System may include metadata to describe how policies are to be used by the external entity. If any recommendations or commands are ambiguous or are not understood by the Orchestrator, then the Orchestrator shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity.

7.3.14 Reference Point $E_{or\text{-}eni\text{-}pol}$

This External Reference Point is used to define Policies and associated information and/or metadata sent by the Orchestrator to the ENI System and/or Policies sent by the ENI System to the Orchestrator. Both the Orchestrator and the ENI System may send any type of Policy that is mutually agreeable. This includes acknowledgements by the receiving entity in both cases. Both the Orchestrator as well as the ENI System may include metadata to describe how the recommendations and/or commands are to be used by the Orchestrator or the ENI System. This is a bi-directional External Reference Point, meaning that data and Policies may flow from the ENI System to the Orchestrator or vice-versa.

The associated information includes policyRefPoint, which is used to indicate the specific External Reference Point (i.e. $E_{or\text{-}eni\text{-}pol}$), so that the acknowledgement could be sent back using the appropriate External Reference Point. This is described in [i.33].

The metadata includes policyType, which is used to indicate the specific type of the policy (i.e. imperative policy, declarative policy and intent policy as defined in clause 6.3.9.3.2), so that the ENI system could perform appropriate actions as specified in clause 6.3.9.6.3. This is described in [i.33].

If any Policy Rule (including any attached metadata) is ambiguous or is not understood by the entity receiving the Policy Rule, then the receiving entity shall report this to entity sending the Policy Rule and ask for further information that defines the meaning and usage of the Policy Rule. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity.

7.3.15 Reference Point $E_{inf\text{-}eni\text{-}dat}$

This External Reference Point is used to define data and acknowledgements exchanged between the infrastructure and the ENI System. This is a uni-directional External Reference Point, meaning that data shall only flow from the Infrastructure to the ENI System.

Input application data received by the ENI System shall be ingested and normalized. The ENI System shall acknowledge the receipt of data and information that it has requested, and may acknowledge the receipt of unsolicited data. If it is not possible to normalize the data, the ENI System shall report this to the infrastructure or the Assisted System and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base to facilitate future communication.

7.3.16 Reference Point $E_{inf\text{-}eni\text{-}cmd}$

This External Reference Point is used to define recommendations and/or commands sent by the ENI System to the infrastructure. Recommendations and commands sent from the ENI System shall be acknowledged by the infrastructure or an agent acting on behalf of the infrastructure. The ENI System may include metadata to describe how the recommendations and/or commands, which may be in the form of policies, are to be used by the external entity. This is a uni-directional External Reference Point, meaning that recommendations and/or commands shall only flow from the ENI System to the Infrastructure.

If any recommendations or commands are ambiguous or are not understood unambiguously by the infrastructure, then the infrastructure shall report this to the ENI System and ask for further information that defines the meaning and usage of the recommendation and/or command. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communication.

7.3.17 Reference Point $E_{eni\text{-}api\text{-}in}$

This External Reference Point is used to define recommendations and/or commands sent by an external entity to the ENI System. The external entity may include metadata to describe or prescribe additional information as necessary. This is a uni-directional External Reference Point, meaning that APIs shall only flow from an external entity to the API Broker over this interface.

If any APIs are ambiguous or are not understood unambiguously by the API Broker, then the API Broker shall report this to the external entity and ask for further information that defines the meaning and usage of the API from the sending external entity. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the API may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.3.18 Reference Point $E_{eni\text{-}api\text{-}out}$

This External Reference Point is used to define ENI APIs sent by the ENI System to an external entity. The ENI System should include metadata to describe how the ENI API is to be used by the external entity. This is a uni-directional External Reference Point, meaning that ENI APIs shall only flow from the ENI System to the external entity.

If any ENI APIs are ambiguous or are not understood unambiguously by the external entity, then the external entity shall report this to the API Broker and ask for further information that defines the meaning and usage of the recommendation and/or command. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the recommendation or command may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.3.19 Reference Point $E_{eni\text{-}api\text{-}dev}$

This External Reference Point is used to define ENI APIs for a developer to create and manage the Functional Blocks that make up the ENI API Broker. Core API Management functions shall include API creating, updating, deleting, documenting, and versioning APIs. It should also include supporting the importing and exporting of API description languages (e.g. RAML, WADL, WSDL, and Swagger) to enable API definitions to be exchanged between the ENI System and external systems and developers that use different languages, platforms, and programming environments. System API Management functions shall include API discovery, API consumer onboarding, configuration management, version management, monitoring, and auditing.

If any APIs are ambiguous or are not understood unambiguously by the API Broker, then the API Broker shall report this to the developer and ask for further information that defines the meaning and usage of the API from the developer. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the API may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.3.20 Reference Point $E_{eni\text{-}api\text{-}run}$

This External Reference Point is used to control the operation of the ENI API Broker. This includes putting the ENI API Broker into development, testing, and operational states. It also includes the ability to selectively enable and disable different functionality via External Reference Points. Finally, it is also responsible for catching exceptions and other runtime errors produced by the ENI API Broker during its operation for mitigation by an appropriate role-based entity (e.g. a developer, an administrator, or an operator).

If any APIs are ambiguous or are not understood unambiguously by the API Broker, then the API Broker shall report this to the developer and ask for further information that defines the meaning and usage of the API from the developer. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the API may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.3.21 Reference Point $E_{eni\text{-}api\text{-}dmg}$

This External Reference Point is used by the developer to monitor the performance and functionality of the ENI API Broker. This includes functionality to analyse the performance of the ENI API Broker in terms of how many APIs have been correctly and incorrectly executed as well as problems in translating from external information to an ENI internal format and vice-versa.

If any APIs are ambiguous or are not understood unambiguously by the API Broker, then the API Broker shall report this to the developer and ask for further information that defines the meaning and usage of the API from the developer. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the API may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.3.22 Reference Point $E_{eni\text{-}api\text{-}emg}$

This External Reference Point enables an external entity to perform all or some of the functionality provided by the $E_{eni\text{-}api\text{-}dmg}$. Some of the functionality that is internal to an ENI System and critical for its operation may not be accessible by some external entities.

If any APIs are ambiguous or are not understood unambiguously by the API Broker, then the API Broker shall report this to the external entity and ask for further information that defines the meaning and usage of the API from the external entity. The API Broker shall notify the ENI System that this API request was unclear. A negotiation of data and information contained in the API may need to occur to arrive at a final viable communication alternative. In this case, the API Broker shall conduct the negotiation. Once the negotiation is completed, the API Broker shall send the results of the negotiation to the ENI System, so that this negotiation may be recorded in its knowledge base to facilitate future communication.

7.4 External Reference Points Protocol Specification

7.4.1 Introduction

Table 7-3 in clause 7.5.2.7 summarizes the advantages and disadvantages of four popular protocols: REST, HATEOAS, GraphQL, and gRPC. This results in the following requirements:

- An ENI System **should** use gRPC and HTTP/2.An ENI System **may** use GraphQL and HTTP/1.1 for specific use cases.
- An ENI System **may** use REST and HTTP/1.1 for specific use cases.
- An ENI System **may** use HATEOAS and HTTP/1.1 for specific use cases.

7.4.2 Generic Protocols for use with External Reference Points

gRPC enables a remote client or server to communicate with another server by simply calling the receiving server's function as if it were local. This makes communicating and transferring large data sets between client and server much easier in distributed systems. Like other RPC systems, gRPC defines a service. It specifies its methods and return types using protobuf to enable the easy definition of services and auto-generation of client libraries. gRPC uses this protocol, currently on version 3, as its interface definition language and serialization toolset.

The gRPC protocol supports various authentication mechanisms, making it easy to adapt to new and existing systems. For example, authentication in gRPC client-server communications can be implemented using recommended mechanisms like TLS (one-way) or mTLS [10], [11] (mutual TLS, for authentication between a client and a server) with or without Google™ token-based authentication. Custom authentication is also supported by extending the built-in authentication function in gRPC [i.51].

By default, gRPC comes bundled with the following authentication mechanisms:

- SSL and TLS to authenticate the server and encrypt the data exchanged between the client and the server.
- Generic token-based authentication mechanism to attach metadata-based credentials to requests and responses.
- Google-specific mechanisms (not covered in the present document).

An ENI System **should** use TLS or mTLS for authentication and encryption. An ENI System **may** use token-based authentication mechanism.

OAuth2 [i.52] **should** be used if token-based authentication is used.

7.4.3 Specific Protocols for use with External Reference Points

7.4.3.1 gRPC and HTTP/2

gRPC **shall** use HTTP/2. gRPC requires HTTP/2, which multiplexes calls over a single TCP connection.

Securing gRPC **shall use** HTTP/2 over TLS with the Application-Layer Protocol Negotiation (ALPN) extension to TLS [10], [11], to negotiate whether both the client and the server support HTTP/2. The TLS Handshake Protocol enables the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. During this protocol, the server will send a signed TLS certificate, which the client validates with its CA. (A CA is a trusted party that issues digital certificates.) This X.509v3 certificate format is used, as described in [15], [16] and [17].

gRPC [i.51] **shall use** the Galois/counter mode of AES. AES is a symmetric-key block cipher algorithm that provides data authenticity (integrity) and confidentiality. A block is a fixed length of bits, and the Galois/counter mode of operation defines how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.

7.4.3.2 GraphQL and HTTP/1.1

GraphQL **should** use HTTP/1.1. However, GraphQL is transport-layer agnostic, so it could use other protocols.

GraphQL **may** use other transport-layer protocols, such as WebSockets.

GraphQL's conceptual model is an entity graph. Hence, entities in GraphQL are not identified by URLs. Instead, a GraphQL server operates on a single URL/endpoint, usually /graphql, and all GraphQL requests for a given service **should** be directed at this endpoint.

7.4.3.3 HATEOAS and HTTP/1.1

GraphQL **should** use HTTP/1.1.

HTTP/1.1 **should** use a reliable transport layer protocol. The most common example is TCP.

7.4.3.4 REST and HTTP/1.1

REST **should** use HTTP/1.1.

HTTP/1.1 **should** use a reliable transport layer protocol. The most common example is TCP.

7.5 ENI API Overview

7.5.1 Introduction

A Software Architecture represents the design decisions related to the structure and behaviour of a software system. From the perspective of ENI, a software architecture is the decomposition of a software system into a set of software modules, each of which obeys the principles outlined in "Functional Concepts for Modular System Operation" [i.35].

The ENI API Architecture is a type of software architecture that focusses on developing software interfaces that expose ENI data and information for use by external applications. The ENI API Architecture is a set of communication protocols, code, and tools that enable an ENI System to interact with either a human or an external software system. A high-level functional diagram of the ENI API Architecture is shown in Figure 7-4.

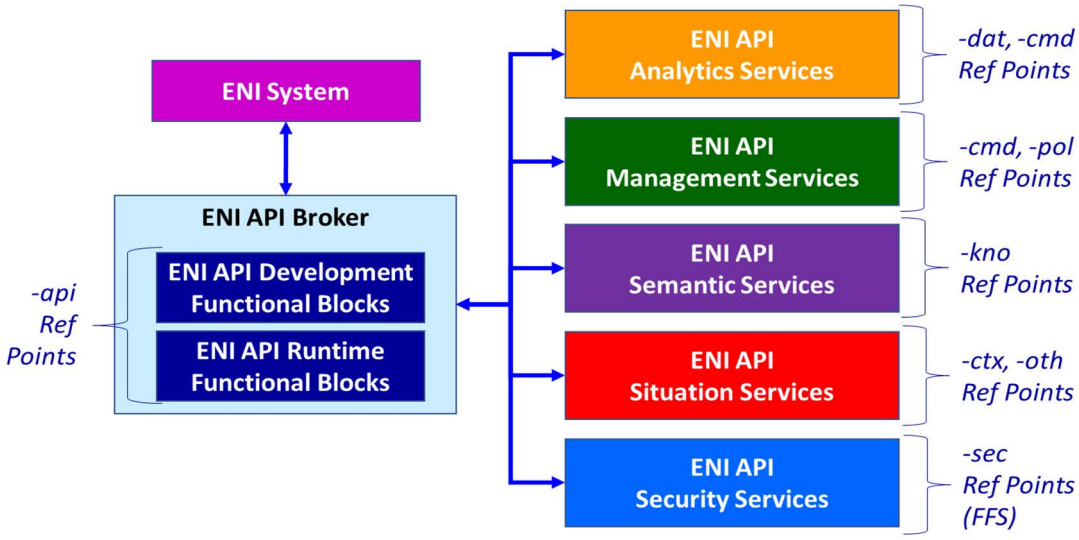

Figure 7-4: High-Level ENI API Architecture Functional Diagram

Figure 7-4 defines 7 types of APIs:

- **ENI Developer APIs:** These APIs are exclusively for the use of the development of the ENI API Broker.
- **ENI Runtime APIs:** These APIs are exclusively for the use of the ENI API Broker, and enable information to be communicated from the Assisted System to the ENI API Broker and hence to the ENI System and vice-versa.
- **ENI API Analytics Services:** These APIs are used to transfer data between external entities and the ENI System for use in analytics.
- **ENI API Management Services:** These APIs are used to send recommendations or commands, including policies, between external entities and the ENI System. Recommendation and commands are uni-directional, APIs, while policies are bi-directional APIs.
- **ENI API Semantic Services:** These APIs are used to transfer semantic information between external entities and the ENI System. These are bi-directional APIs.

NOTE 1: Semantic services, such as models and inferencing explanations, are separated from management services in order to differentiate between a management decision and other information that can be used to make a management decision.

- **ENI API Situation Services:** These APIs are used to transfer context- and situation-aware information, as well as application-specific data, between external entities and the ENI System. These are bi-directional APIs.
- **ENI API Security Services:** These APIs are used to perform security functions between external entities and the ENI System. These APIs are for further study.

Figure 7-5 provides a more detailed view of how these seven sets of APIs are organized among the ENI API Broker and the ENI System. In this figure, the ENI API Analytics Services are input output only (i.e. from the ENI System to the API Broker and hence to the Assisted System). All other ENI API Services are bi-directional.

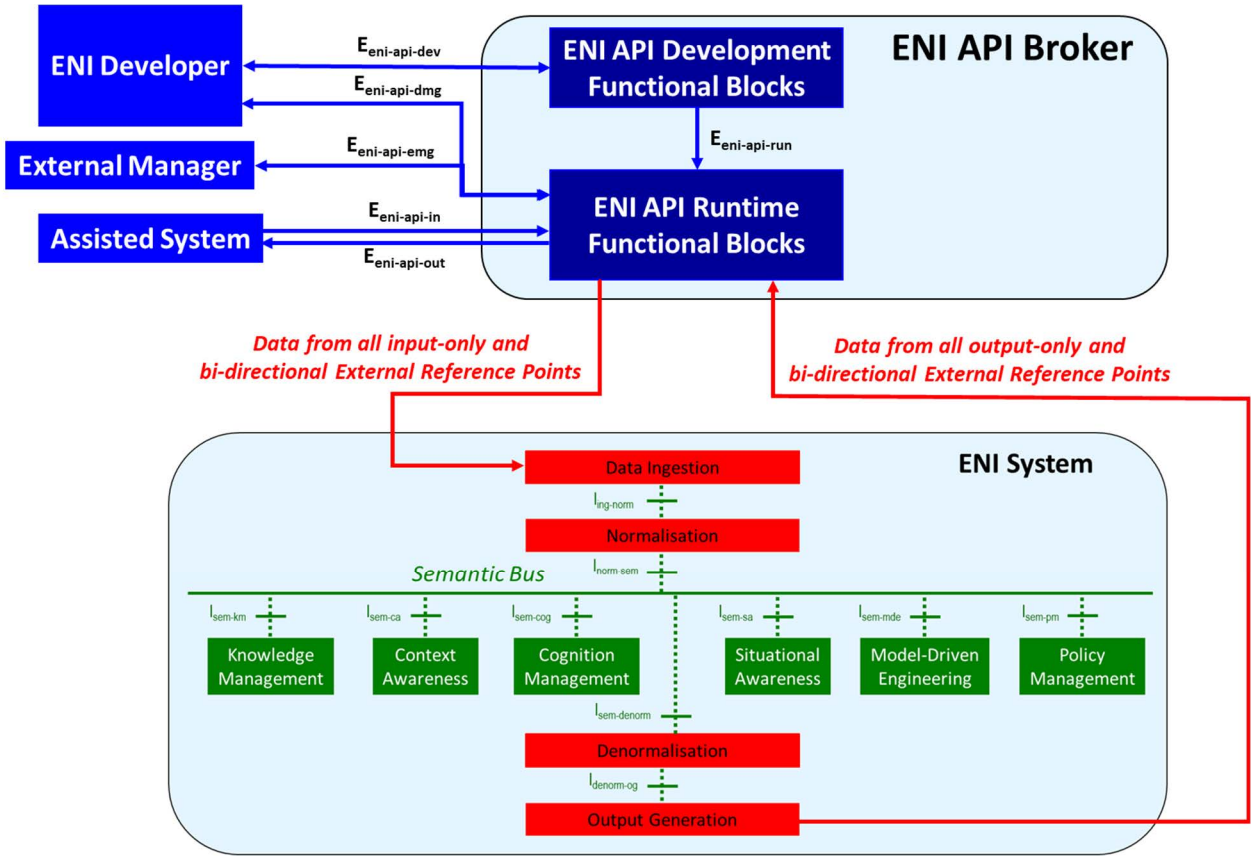

Figure 7-5: ENI API Broker Interaction

The purpose of the Functional Blocks shown in Figure 7-5 are summarized as follows.

- The **ENI System** is an innovative, policy-based, model-driven functional architecture that improves operator experience. The ENI System assists decision-making of humans as well as machines, to enable a more maintainable and reliable system that provides context-aware services that more efficiently meet the needs of the business. It is defined in clause 6.3 of the present document.

- The **ENI API Broker** decouples the ENI System from other external systems that it communicates with. This enables the development of the ENI System and external systems to proceed independently. The ENI API Broker shall be used to translate between the ENI APIs and APIs of the Assisted System and its Designated Entity. More specifically, the ENI API Broker shall ingest APIs through an appropriate ENI External Reference Point, analyse the ingested API, and then route the functionality of the ingested API to an appropriate ENI Functional Block (this is typically the Data Ingestion Functional Block, but may be a different Functional Block if the API is known to an ENI System). Two new API-specific Functional Blocks are introduced into the ENI API Broker:

  - The **ENI API Development Functional Block**. This Functional Block shall be responsible for the ENI API design and development. It can be thought of as a toolbox that contains reusable and modular building blocks to compose ENI APIs. Two External Reference Points are defined. The first, $E_{eni\text{-}api\text{-}dev}$, is used by a developer to create and manage the Functional Blocks that make up the ENI API Broker. Core API Management functions shall include API creating, updating, deleting, documenting, and versioning APIs. It should also include supporting the importing and exporting of API description languages (e.g. RAML, WADL, WSDL, and Swagger) to enable API definitions to be exchanged between the ENI System and external systems and developers that use different languages, platforms, and programming environments. System API Management functions shall include API discovery, API consumer onboarding, configuration management, version management, monitoring, and auditing. The second, $E_{eni\text{-}api\text{-}run}$, is used to control the operation of the ENI API Broker. This includes putting the ENI API Broker into development, testing, and operational states. It is also responsible for catching exceptions and other runtime errors produced by the ENI API Broker during its operation for mitigation by an appropriate role-based entity.

- The **ENI API Runtime Functional Block**. This Functional Block shall be responsible for executing APIs. It enables the APIs to accept incoming requests from external consumers of the ENI APIs and provide appropriate responses. Formally, API Brokering translates external information and commands to a format suitable for ENI ingestion, and similarly, translates ENI internal information and commands to a format suitable for external ingestion. Two External Reference Points are defined. $E_{eni\text{-}api\text{-}in}$ is used to define recommendations and/or commands sent by an external entity to the ENI System. $E_{eni\text{-}api\text{-}out}$ is used to define ENI APIs sent by the ENI System to an external entity. API Service Level Agreements (i.e. contractual definitions defining the performance requirements of the ENI API Broker) may optionally be included in this Functional Block.

- The performance of the ENI API Broker has two External Reference Points. $E_{eni\text{-}api\text{-}dmg}$ is used by the developer to monitor the performance and functionality of the ENI API Broker. Similarly, some or all of these functions may be done by an external entity using $E_{eni\text{-}api\text{-}emg}$.

There are five different types of data services provided by ENI:

- **ENI API Analytics Services** shall consist of a set of data and information sent from the ENI System to an external entity for analysing the performance of an ENI System as well as to further enhance the conclusions of an ENI System. This includes information that was ingested by an ENI System as well as recommendations and commands that were produced by an ENI System (e.g. the data input via the -dat External Reference Points as well as the recommendations and commands sent via the -cmd External Reference Points).

- **ENI API Management Services** shall consist of the recommendations and commands produced by an ENI System (e.g. the recommendations and commands sent via the -cmd External Reference Points and/or the policies sent via the -pol External Reference Points). It may also include insights as to the ENI API Broker performance (e,.g., from $E_{eni\text{-}api\text{-}emg}$).

- **ENI Semantic Services** shall consist of data, model, and related information produced by an ENI system (e.g. information sent via the -kno External Reference Point).

- **ENI Situation Services** shall consist of data and related information produced by an ENI system (e.g. information sent via the -ctx and -oth External Reference Point).

- **ENI API Security Services** is for further study.

NOTE 2: Additional functionality, such as API lifecycle management and an ENI API Developer portal, depend on the type of API architectural style chosen and other factors, and will be addressed in a future version of the present document.

7.5.2 API Architectural Styles

7.5.2.1 Introduction

One of the primary purposes of an API is to exchange information and data. For the purposes of ENI, "information" is represented by Information, Knowledge, and Wisdom; coupled with Data, this represents the DIKW hierarchy referred to in clause 6.3.4.4.3 and defined in "Functional Concepts for Modular System Operation" (ETSI GR ENI 016 [i.35]) for background information and a more thorough discussion of the DIKW hierarchy.

The most generic form of data exchange is to transform data from a form structured according to a source schema to a form structured in a target schema, so that the target data is an accurate representation of the source data. Since this involves a transformation, there are three cases to consider:

1) There is a single way to transform a datum from its original (source schema) to its intended (target schema) representation.

2) There may be no way to transform a datum from its original (source schema) to its intended (target schema) representation.

3) There may be many different ways to transform a datum from its original (source schema) to its intended (target schema) representation.

The current state-of-the-art is to define an architecture that uses APIs for data exchange. More specifically, it has clients interacting with an API layer representing the application on the server-side. The benefit of this API-based approach to application architecture design is that it enables a wide variety of physical client devices and application types to interact with the given application. For example, the same API can be used for PC-based, cellphone, and IoT computing. Communication may occur between humans and/or applications.

7.5.2.2 Challenges in API Architectures

Applications are designed using a variety of languages and protocols. This causes a proliferation of file types and encodings. Two exemplary file types are text (e.g. HTML, XML, TEXT, SQL) and binary (e.g. FOC, XLS, PDF, MP3). File formats are protocols which programs agree to use to represent information. A file format is responsible for ensuring the data is structured properly and correctly represented. For example, a JPG file is an image that has a predefined internal format that allows different programs (Browsers, Spreadsheets, Word Processors) to use the file as an image. This is also true of text files (e.g. the format of HTML enables a Browser to render a web page in a specific way). Another example is a json file, which defines its structure as a set of nested key-value pairs.

An encoding transforms data into a format required for a number of information processing needs Generically, this transforms data into a set of symbols. The reverse process is called decoding. A simple example is adding metadata, such as a checksum, into a transmission that can be used to verify data's correctness. Another example is to increase interoperability, json files should be encoded in UTF-8.

Consequently, there are variations in APIs that depend on file format and encoding.

Different API architectures may be used to implement data exchange between entities. Most API architectures are intended to exchange *data*. ENI shall be able to exchange both data as well as *information*, *knowledge*, and *wisdom*. These latter three categories may include inferences, models, and other information.

ENI has defined specific APIs to differentiate between exchanging data and these other types of information. This is shown in Figure 7-4 and Figure 7-5, and described in clause 7.5.1.

The following four clauses will describe four popular API architectures: REST, HATEOAS, GraphQL, and gRPC. Clause 7.5.2.6 will compare them and provide recommendations for use in ENI.

7.5.2.3 REST API Style

REST (Representational State Transfer) is an architectural style for services, and it defines a set of architectural constraints and agreements for building web-based APIs. The constraints are [i.45]:

1) Client-server. Clients and servers are distinct, and may be implemented and deployed independently, using any language or technology, so long as they conform to the Web's uniform interface.

2) Uniform interface. This is defined as four separate rules:

   i) Identification of resources. Each distinct Web-based concept is known as a resource and may be addressed by a unique identifier.

   ii) Clients manipulate representations of resources. The same exact resource can be represented to different clients in different ways.

   iii) Self-descriptive messages. A resource's desired state can be represented within a client's request message. A resource's current state may be represented within the response message that comes back from a server.

   iv) Links. A resource's state representation includes links to related resources.

3) Layered system. This enables network-based intermediaries such as proxies and gateways to be transparently deployed between a client and server using the Web's uniform interface.

4) Cache. A web server can declare the cacheability of each response's data.

5) Stateless. A web server is not required to keep track of the state of its client applications. As a result, each client shall include all of the contextual information that it considers relevant in each interaction with the web server.

6) Code-on-demand. This enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients.

NOTE: Code-on-demand defines a technology coupling between web servers and their clients, since the client needs to be able to understand and execute the code that it downloads on-demand from the server. Hence, this constraint should be considered optional.

REST is a stateless, cacheable, and simple architecture that is not a protocol but a pattern. REST is designed to make optimal use of an HTTP-based infrastructure. The fundamental concept behind the REST style is using resources. A resource is a data structure, which can be serialized to various concrete representations (e.g. a JSON or an XML representation). REST APIs expose and manipulate these resources.

REST Resources are similar to objects, except that in REST, methods are restricted to the set of HTTP methods (sometimes they are also called HTTP verbs). This set of allowed methods provides a standard set of methods that can be used to manipulate the resource. No other methods can be stated in API requests, neither in the HTTP body nor in the base path nor in the parameters. The five major methods are:

Table 7-2: REST Methods

REST VERB	Description	Success Code(s)	Failure Code(s)
GET	Fetches a set of resources	200	404
POST	Creates a new set of resources	200	n/a
PUT	Updates or replaces the given set of resources	200, 204	404
PATCH	Modifies the given resource	200, 204	404
DELETE	Deletes the given resource	200	404

The status codes are the same status codes used by HTTP.

RESTful APIs may also use HTTP methods. For example, the OPTIONS method enables a client to discover how it can interact with a resource (i.e. which specific REST methods can be used, which version of HTTP the server supports, and what type of content the API is returning. As another example, the HEAD method returns information about the resource itself (e.g. has it been changed, and metadata about the resource).

The REST architectural style ensures that APIs use HTTP correctly by imposing the following constraints:

- APIs are designed to use resources, not methods. Hence, APIs use nouns, not verbs.
- A uniform interface, defined by a subset of HTTP methods and capabilities, is used. This ensures that the semantics of using HTTP are adhered to.
- Stateless communication occurs between client and server.
- Requests should be independent and loosely coupled.

The REST architectural style provides a number of important benefits. First, the implementation can scale easily by adding additional servers. This also provides improved fault tolerance, availability, and reliability. Caching is provided by the HTTP infrastructure. The REST API style is designed to be simple, and does not require much overhead. Finally, REST can support multiple content types (e.g. a REST API may be able to deliver the resource in multiple, alternative formats, enabling different clients that have different capabilities to use the same API). HTTP provides content type negotiation mechanisms, which define how clients can exchange information about their capabilities and negotiate the appropriate content type.

7.5.2.4 HATEOAS API Style

The HATEOAS (Hypermedia as the Engine of Application State) API architecture style constrains a REST API architecture style to enable APIs to be self-descriptive. In HATEOAS, the term "hypermedia" refers to any content that contains links to other forms of media, such as images, movies, and text. The semantics of the resources is provided by media types. That is, all actions that can be performed on a resource are described in the representation of the resource in the form of annotated links. The annotated links can be navigated by a generic client, which can interpret and follow links. Since all resources are linked, the client only needs to have access to the root resource; the client can then follow the links to reach any other resource. This enables a dynamic architecture to be defined, since it effectively decouples the client from the server. All meta-information is obtained before the API request. Hence, if the API is changed, all resources that link to the changed resource are updated with new links and new associated meta-information.

A REST request only provides the data and not any actions around it. A HATEOAS request enables the server to send the data and any related actions. For example, with HATEOAS, a request for a calendar entry may also return URIs to any actions associated with that calendar entry. This simplifies understanding and is a step towards making APIs self-documenting.

The single most important reason for HATEOAS is loose coupling (see ETSI GR ENI 016 [i.35], clause 4.1.8). If a consumer of a REST service needs to hard-code all the resource URLs or URIs, then it is tightly coupled with its service implementation. Instead, if URLs or URIs are returned, then there is no tight dependency on the URI structure, as it is specified and used from the response, and it is loosely coupled.

The HATEOAS API architecture style can be modeled as a state machine. Resources correspond to states, and the links between the resources correspond to the transitions of the state machine. This provides a number of important advantages. First, the HATEOAS API architecture style is inherently flexible, since new versions, or changed media types, can be handled by the server without breaking any clients. Second, since the client and server are decoupled, the API and its clients do not need to evolve together. Most importantly, the client does not need any a-priori knowledge of the API.

7.5.2.5 GraphQL API Style

GraphQL is an open-source data query language for APIs and a runtime for fulfilling those queries. GraphQL is intended to develop APIs used on the web. GraphQL was originally developed by Facebook for internal use before being publicly released in 2015. The GraphQL project was subsequently moved from Facebook to the GraphQL Foundation, hosted by the Linux Foundation.

Despite its name, GraphQl is *not* a graph database. GraphQL defines a a type system, query language and execution semantics, static validation, and type introspection. It supports reading, writing, and subscribing to changes to data. GraphQL is not tied to any specific language. Instead, it enables the developer to define the structure of the data required. This means that query results have the same composition as the query. GraphQL formalized this by defining its own Schema Definition Language.

Simplistically, a GraphQL query reports on fields in an object. In contrast to REST APIs, GraphQL APIs are structured in terms of types and fields, not endpoints. This means that GraphQL may use a single endpoint to describe a system. Furthermore, every GraphQL field and object may have a set of arguments; this means that GraphQL may enable multiple API calls to be made. This is in contrast to a REST-based approach, which is limited to only providing a single set of arguments (i.e. the query parameters and URL segments in a request).

GraphQL's type system defines a set of rules that assign variables, functions, and other parts of a program to a datatype, which governs the behaviour of that program component. A type system enables interfaces between different parts of a computer program to have predictable and consistent semantics (e.g. an integer always behaves as an integer).

GraphQL does have some notable limitations. First, GraphQL is just a simple query language. It does not have the power of a robust query language, such as SQL. For example, it does not support transitive closure (e.g. given a query that returns the parents of an object, it cannot return the set of *all* parents of that object in a single query). A side effect of this (and other limitations) is that queries that return too many results, or are recursive in nature, need to be avoided.

Second, since GraphQL allows each query to be different (e.g. with different parameters), caching results for GraphQL is more complicated than other approaches. GraphQL does not offer built-in, standardized caching. Similarly, limiting the number of queries in a GraphQL system is also very difficult. This is mitigated to some extent by some GraphQL implementations.

Third, GraphQL queries always return a HTTP status code of 200, regardless of whether or not that query was successful. If your query was unsuccessful, the response will have a top-level errors key with associated error messages and stacktrace. This can make it much more difficult to do error handling and can lead to additional complexity for things like monitoring.

7.5.2.6 gRPC API Style

The gRPC (Remote Procedure Call) is an open-source data exchange technology developed by Google using the HTTP2 protocol. HTTP is *not* presented to the API developer or server, so there is no need to explicitly map RPC concepts to HTTP. It uses the Protocol Buffers binary format (Protobuf) for data exchange. This means that Protobuf may be used as both its Interface Definition Language (IDL) and as its underlying message interchange format. Also, this architectural style *enforces rules* that a developer shall follow to develop or consume web APIs.

While REST is a set of *guidelines* for designing web APIs, it does not *enforce* anything. In contrast, gRPC *enforces* rules by defining a .proto file that shall be adhered to by both client and server for data exchange.

gRPC is a Resource-Orientated Architectural (ROA) style. Distributed architectures are made up of components that are consumed across a network through well-defined interfaces. In ROA, these components are referred to as resources. A resource is a self-contained, identifiable entity having a state that can be assigned a uniform resource locator (URI). While a service represents the execution of a requested action, a resource represents a distributed component that is manageable through a consistent, standardized interface.

ROA extends the REST architectural style and offers a broader, extensible, flexible, and transport-independent architecture. ROAs have four main features:

1) **Uniform Interface.** A uniform and consistent set of well-defined methods that manipulate the resources in an application. In particular, standard methods are preferred over custom methods. (see clause 8.3).

2) **Addressability.** An application is addressable if it publishes the salient aspects of its data set as service endpoints. ROA resources are exposed using URIs.

3) **Statelessness.** Statelessness requires the client to provide all information that the server needs for the request to be successful. This is because the server does not store information from previous requests.

4) **Connectedness.** Similar to HATEOAS (see clause 7.5.2.4), applications built on ROA should have all of its resources linked to each other. A hyperlink can be used to connect any resource to another resource.

In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object. gRPC is based on the concept of defining a service. The service consists of a set of methods, with their parameters and return types, that can be called remotely. On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls. The gRPC infrastructure decodes incoming requests, executes service methods, and encodes service responses. On the client side, the client has a local object known as stub (for some languages, the preferred term is client) that implements the same methods as the service. The client can then just call those methods on the local object, wrapping the parameters for the call in the appropriate ProtoBuf message type - gRPC looks after sending the request(s) to the server and returning the server's ProtoBuf response(s). gRPC clients and servers can run and talk to each other in a variety of environments. For example, a gRPC server may be created in one programming language, and one or more clients may be created in the same or different programming languages.

By default, gRPC uses ProtoBuf, although it can be used with other data formats such as JSON.

ProtoBuf is an IDL, and as such, the first step involves defining he structure for the data to be serialized in a specific file, called a proto file. This is an ordinary text file with a .proto extension. ProtoBuf data is structured as messages, where each message is a small logical record of information containing a series of name-value pairs called fields. When all of the data structures have been specified, the ProtoBuf compiler (protoc) is used to generate data access classes in the language(s) defined in the proto definition. The compiler generates gRPC client and server code, as well as the regular ProtoBuf code for populating, serializing, and retrieving messages. Messages provide simple accessors for each field, like name() and set_name(), as well as methods to serialize/parse the whole structure to/from raw bytes. These classes can then be used in applications to populate, serialize, and retrieve those entities from ProtoBuf messages. gRPC provides strongly typed messages automatically converted using the Protobuf exchange format to the chosen programming language.

gRPC follows a client-response model of communication for designing web APIs that rely on HTTP2. Hence, gRPC allows streaming communication and serves multiple requests simultaneously. In addition to that, gRPC also supports unary communication similar to REST.

HTTP2 supports multiple requests over a single TCP connection (HTTP1.1 does not). This is important, since some browsers have limits on the number of TCP connections. In HTTP2, browser performance increases significantly because a large number of requests can be made under a single connection.

Synchronous RPC calls that block until a response arrives from the server are the closest approximation to the abstraction of a procedure call that RPC aspires to. On the other hand, networks are inherently asynchronous and, in many scenarios, it is important to start RPCs without blocking the current thread. The gRPC programming API comes in both synchronous and asynchronous flavors in most languages. gRPC enables four kinds of service methods to be defined:

1) **Unary RPCs**. The client sends a single request to the server and gets a single response back.

2) **Server streaming RPCs**. The client sends a request to the server and gets a sequence of messages back to read. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.

3) **Client streaming RPCs**. The client writes a sequence of messages and sends them to the server. The client waits for the server to read them and return its response. gRPC guarantees message ordering within an individual RPC call.

4) **Bidirectional streaming RPCs**. The client and server both send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like. The order of messages in each stream is preserved.

gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated with a DEADLINE_EXCEEDED error. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC. A deadline (also called a timeout) is language-specific: some language APIs work in terms of timeouts (durations of time), and some language APIs work in terms of a deadline (a fixed point in time) and may or may not have a default deadline. In either case, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. For example, an RPC that finishes successfully on the server side may fail on the client side (e.g. the response arrived after its deadline). Finally, either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done. This means that any changes made before a cancellation are not rolled back.

gRPC relies on HTTP2 protocol, which uses multiplexed streams. Therefore, several clients can send multiple requests simultaneously without establishing a new TCP connection for each one. Also, the server can send push notifications to clients via the established connection.

However, gRPC has limited browser support because numerous browsers (usually the older versions) have no mature support for HTTP2. So, it may require gRPC-web and a proxy layer to perform conversions between HTTP 1.1 and HTTP2. Therefore, gRPC is currently primarily used for internal services.

7.5.2.7 ENI API Architectural Style Recommendations

Table 7-3 compares the key features of REST, HATEOAS, GraphQL, and gRPC.

Table 7-3: Comparison of API Styles

Concept	REST	HATEOAS	GraphQL	gRPC
Abstraction	Communication-Orientated: request-response model of HTTP 1.1	Same as REST	Data-Orientated: Client determines data, how it is served, and in what format	Programming Orientated: Services with Interfaces and structured messages
Comm Model	Unary Client Request-Response (e.g. synchronous)	Same as REST	Supports sync or async calls and responses	Unary or Streaming (Server, Client, or both) Client-Response
Protocol	Typically built on HTTP1.1	Same as REST	Typically built on HTTP1.1	Requires HTTP 2
Format	Many	Same as REST	Typically JSON	ProtoBuf (by default)
Response	Sends everything found	Sends everything found with actions	Sends just what is needed	Defined by data structures
Payload Structure	No structure defined	Same as REST	Defined according to the GraphQL specification	Strong typing defined by the ProtoBuf specification
Browser Support	Fully supported by all common browsers	Same as REST	Most popular browsers have GraphQL addons	Fully supported by HTTP 2 Browsers, else requires gRPC-web and a proxy layer
State	Stateless	Same as REST	Stateless with support of async messages	Stateless
Sync vs. Async	Synchronous only	Same as REST	Supports sync or async calls and responses	Supports unary sync & async and streaming
Data Transmission	Heavyweight (formats need to be serialized)	Same as REST	Heavyweight (formats need to be serialized)	Lightweight (ProtoBuf is binary)
Human Readability of Payload	Full if json is used	Same as REST	Full if json is used	None if ProtoBuf is used

Concept	REST	HATEOAS	GraphQL	gRPC
Different Languages for Client and Server	NONE, needs API Broker	Same as REST	Specification enables support for different clients and servers to be built	YES
Language Support	NONE, needs third party tools	Same as REST	Specification enables support for different clients and servers	ProtoBuf auto converts messages to client and server programming languages
Code on Demand	Supported by the architecture	Same as REST	Not supported	Not supported, but also, not applicable
Code Generation	None; a third party tool is required	Same as REST	Native code generation for support languages	Native code generation for support languages

7.5.3 ENI API Functional Blocks

7.5.3.1 ENI API Development Functional Block

7.5.3.1.1 Introduction

The purpose of the ENI API Development Functional Block is to enable the developer to program the functionality of the ENI API Broker. The External Reference Point $E_{eni\text{-}api\text{-}dev}$ defines ENI APIs and associated information and metadata for a developer to create and manage the Functional Blocks that make up the ENI API Broker (see clause 7.2, Table 7-1b). The ENI API Development Functional Block may be accessible through a number of means, including APIs, a GUI, and/or CLI.

The ENI API Broker APIs define the following functionality that should be made available at runtime:

1) **Proxy** Definition.
2) API Management.
3) Request Routing.
4) API Composition.
5) Protocol T**ranslation**.

Figure 7-6 shows a simplified overview of the internal architecture of the ENI API Broker.

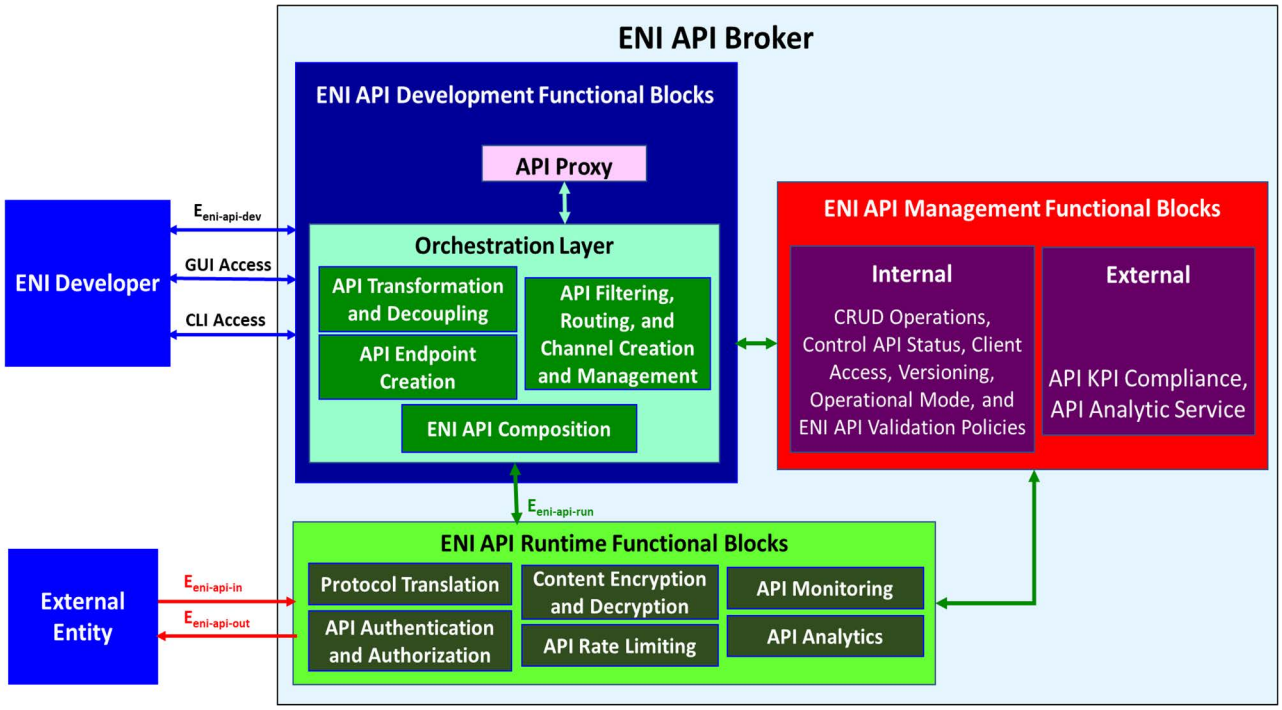

Figure 7-6: Overview of the ENI API Broker Internal Architecture

7.5.3.1.2 ENI Broker API Orchestration Layer

The ENI Broker API Orchestration Layer separates development and runtime concerns according to the Single Responsibility Principle [i.9]. Orchestration is how data, information, and inferences are moved through the ENI API Broker based on APIs. Orchestration can be seen as combining service calls to create higher-level, more useful composite services, and implies implementing business-level processes combining business-specific services across applications and information systems.

Orchestration involves decoupling point-to-point integration in favour of chains of operations that can be reused or changed as a business or its systems need change. For example, if an event triggers a set of complex operations on a set of data that have interdependencies (e.g. data need to be encrypted, validated, ordered, and deduplicated), then the ENI API Broker Orchestration Layer will transform the set of possibly encrypted, duplicated, and unordered data into a stream of unique, ordered, and encrypted data using a set of APIs that are composed in a particular pattern to solve this problem. This is an example of realizing a service-based framework, where multiple components are wrapped with endpoints to decouple them from protocols and routing requirements. This enables multiple services to work as one when implementation requirements demand it.

7.5.3.1.3 ENI Broker API Management Functional Blocks

The ENI Broker API Management Layer controls the internal and external management of the ENI Broker APIs.

Internal functionality includes:

1) The Creation, Reading, Updating, and Deletion (CRUD) of an ENI API.
2) Control the initial deployment status of an ENI API.
3) Control which ENI APIs clients can use.
4) The versioning of an ENI API.
5) The management of metadata for each ENI API.
6) The definition and use of policies for validating each ENI API.
7) Controlling the mode of operation of the ENI API Broker.

CRUD operations of an ENI API shall include defining the capabilities of an ENI API. Examples include defining metadata for a given ENI API, different persistent schemes for API requests and responses, applicable ENI API Validation Policies, and transaction functionality (if any).

Controlling the status of an ENI API shall determine whether it is active or inactive (and can or cannot be used in an operational system), or whether is in a special development or test mode (see below).

Controlling which ENI APIs a client can use should be done using a combination of Roles and metadata from the ENI Extended Core Model [9], as this prescribes an extensible mechanism to group functionality desired by a client into a Role that other clients may share, and defines how to attach one or more metadata objects to any other object.

ENI API versioning should be done using the ENI Extended Core Model [9], as this prescribes an extensible mechanism to attach one or more metadata objects to any other object.

Each ENI API should have attached metadata that describes and/or prescribes how to use this particular API. This should be done using the ENI Extended Core Model [9].

All ENI APIs shall have a set of one or more policies to validate ENI API requests. ENI API responses may also be validated. This type of policy is called an ENI API Validation Policy. An ENI API Validation Policy is a type of ENI Policy (see clause 6.3.9 and [9]) that executes appropriate business logic to determine if an ingress API request to (and optionally, egress response from) the ENI API Broker is well-formed, unambiguous, and has no syntax or semantic errors. In particular, an ENI API Validation Policy:

1) shall be constructed using the ENI Extended Policy Model [9];
2) should be context-dependent (this may require multiple policies, one or more for each context);
3) shall be validated by the ENI Broker API Proxy;
4) shall provide a traceable account of the operation of the ENI API Validation Policy;
5) may provide a rate limit to a particular ENI API in terms of role, context, system load, and other factors.

The mode of operation of the ENI API Broker shall include at least the following modes:

1) **Developer Mode.** This mode places the ENI API Broker into a special state for enabling a developer to change the functionality of the ENI API Broker. In this state:
    a) generic external access to the ENI API Broker shall not be allowed;
    b) only personnel having a developer role shall have full access to the ENI API Broker in this mode;
    c) special build processes may need to be used.
2) **Testing Mode.** This mode places the ENI API Broker into a special state for enabling a developer to test the functionality of the ENI API Broker. In this state:
    a) generic external access to the ENI API Broker shall not be allowed;
    b) ENI System personnel, along with certified external clients and testing personnel that have a testing role shall have full access to the ENI API Broker in this mode;
    c) special build processes may need to be used;
    d) the ability to selectively enable and disable different functionality shall be allowed by personnel having a developer role.
3) **Operational Mode.** This mode places the ENI API Broker into a normal operational mode. In this state:
    a) generic external access to the ENI API Broker shall be allowed;
    b) personnel having a developer role shall have full access to the ENI API Broker (e.g. in case of having to fix a catastrophic failure);
    c) personnel having a developer or an operator role shall have the ability to selectively enable and disable different functionality;

d) catching exceptions and other runtime errors shall be allowed by an appropriate role-based entity (e.g. a developer, an administrator, or an operator).

External functionality includes performance and analytics of ENI APIs. Performance of the ENI API Broker is defined in terms of appropriate KPIs. These include:

1) How many ENI APIs have been correctly and incorrectly executed?

2) How many ENI APIs had translation problems (e.g. protocol, encoding, format) in translating from external information to an ENI internal format and vice-versa?

3) How many ENI APIs contained ambiguous content (this is primarily for policy APIs)?

4) How many ENI APIs had to negotiate to resolve error problems?

5) How many requests had violations?

The ENI API Broker should provide analytics of its APIs. Analytics uses the above performance information, as well as API Event statistics, to provide information suitable for display via reports or preferably, a dashboard. In particular, analytics offer the capability to filter, sort, aggregate, and infer statistical and trending information about the ENI APIs. Exemplary analytics information includes:

- Latency and Delay metrics.
- Request and Response codes.
- Requests by Date or Location.
- Requests resulting in an error or failure.
- Requests per ENI External Reference Point.
- Requests by Function.
- Requests per ENI Internal Functional Block.

7.5.3.2 ENI API Runtime Functional Block

The API Runtime Functional Blocks provides a *single entry point* for all API calls that are sent to the ENI System, regardless of how the ENI System is deployed (e.g. hosted in an on-premises data center or on a cloud). It accepts requests that come in remotely over the External Reference Point $E_{eni\text{-}api\text{-}in.}$ Similarly, it provides a single entry point for all responses to API requests, and returns the requested data over the External Reference Point $E_{eni\text{-}api\text{-}out}$ (see clause 7.2, Table 7-1b for the definitions of both of the External Reference Points). The ENI API Broker makes data available in a way appropriate for the requestor's technology. This goes beyond ensuring that the protocol and encoding are correct. For examples, a requester on a computer is able to see much more information than a requester viewing the same data on a mobile phone, since the computer has more powerful functionality. The ENI API Broker can also enable real-time communication between an external client and an ENI System, as well as between multiple ENI Systems.

The ENI API Broker knows the protocol(s) of incoming API requests, and translates those protocols between the requester and the ENI System on ingress and between the ENI System and the requester on egress. This translation can be device-specific (e.g. a web browser or mobile device) or network-specific (e.g. a WAN, LAN, or RAN).

The ENI API Runtime Functional Blocks provide workflow orchestration as it aggregates the requested information from multiple APIs, bundles the data. and returns it to the requestor in composite form.

7.5.3.3 ENI Management Services Functional Block

API management is an overall solution that manages the entire API lifecycle as well as the ENI API Broker and how ENI APIs are developed and managed. The goal of API management is to ensure that the needs of developers and applications that may use the API are being met, concerning organizations that publish or use APIs to monitor an interface's lifecycle.

The management of the API lifecycle shall include the processes for distributing, controlling, and analysing the APIs that connect applications and data across the enterprise and across clouds. API lifecycle management enables APIs to be manageable from design, through implementation, until retirement. The three main phases of the API lifecycle are creating (building and documenting the API), controlling (applying security) and consuming (publishing and monetizing your APIs). API gateways fall under the control phase of the API lifecycle - they secure APIs and keep data safe.

The goal of API management is to allow organizations that use APIs to monitor activity and ensure the needs of the developers and applications using the API are being met.

The management of the ENI API Broker shall include providing access control, rate limits, and usage policies to govern the use of APIs. Most API management solutions generally also include the following capabilities:

The management of the development of APIs should include a developer portal and analytics. A developer portal should provide API documentation along with developer onboarding processes like signup and account administration. Analytics provide detailed metrics for determining how the ENI APIs are performing, including whether they are meeting their KPIs as well as how many APIs have failed and why.

7.5.3.4 ENI Security Services Functional Block

API security is a vital element of API management. It protects the ENI APIs against unauthorized access and threats. API security requires more than authenticating and authorizing user access to the API. Standards and policies shall be established to protect sensitive data and ensure those data are not leaked or compromised. This is for further study in Release 4.

7.5.3.5 ENI Analytic Services Functional Block

API analytics focusses on the centralized collection and analysis of API metrics provided by real-time monitoring and optionally, tools such as dashboards. API analytics allows developers and organizations to see and understand how their APIs are being used as well as rank their performance. This is for further study in Release 4.

7.5.4 ENI API System Deployment Models

7.5.4.1 On Premise vs. Cloud-Based Deployment

The ENI API Architecture may be deployed on premise or in the cloud. For security reasons, the ENI System backend systems shall not be available via the Internet.

The advantage of deploying the ENI API Architecture on premise is that the required backend systems that the ENI API Architecture communicates and interacts with are also on premise. The API Architecture thus serves as an additional protection and security layer for the backend systems. The advantage of cloud-based deployment models is that they are typically elastic and scale well. API platforms in the cloud make sense as long as they do not need to connect to secured backend systems in a private network. If they do, then a virtual private network, or some other means of securing the network, needs to be used.

7.5.4.2 ENI API Architecture Environment

Large organizations may have a number of legacy systems. In this case, the API Broker shall serve as an API gateway, enabling legacy APIs and protocols to connect to and interact with ENI APIs.

7.5.4.3 Securing the ENI API Broker from the Internet

Security devices (e.g. firewalls) may be placed between the Internet and the ENI API Broker. MAC and IP level filtering are typically performed by dedicated bespoke security devices, while application level filtering is performed by the API runtime Functional Block.

7.5.4.4 Scaling the ENI API Broker

Load balancers may be placed between the ENI API Broker and the Internet. For example, load balancers may be used to route the traffic from the Internet to one of several nodes of the ENI API Broker (e.g. if it is running as a distributed system or cluster) to distribute API requests equally among the nodes. Load balancers may also detect the source of the API request (e.g. the type of client) and type of API request (e.g. its purpose) and preferentially route such APIs to dedicated processing nodes. Load balancers may also be placed between the ENI API Broker and the ENI System. For example, the API Broker may send all requests for a specific ENI System function, such as Intent Policy parsing, to a specific set of load balancers.

7.6 Internal Reference Point Overview

Figure 7-7 shows an overview of the Internal Reference Points that provide internally facing Internal Reference Points (i.e. Internal Reference Points that communicate data and information between internal ENI Functional Blocks). Such communication should use the Semantic Bus to ensure that any ENI Functional Block that needs the information can receive it through an appropriate subscription. Internal Reference Points that are between specific Functional Blocks may be used, but are discouraged; this is because the six internal ENI Functional Blocks collaborate on the large majority of operations. Figure 7-6 shows ENI Reference Points for a single domain only.

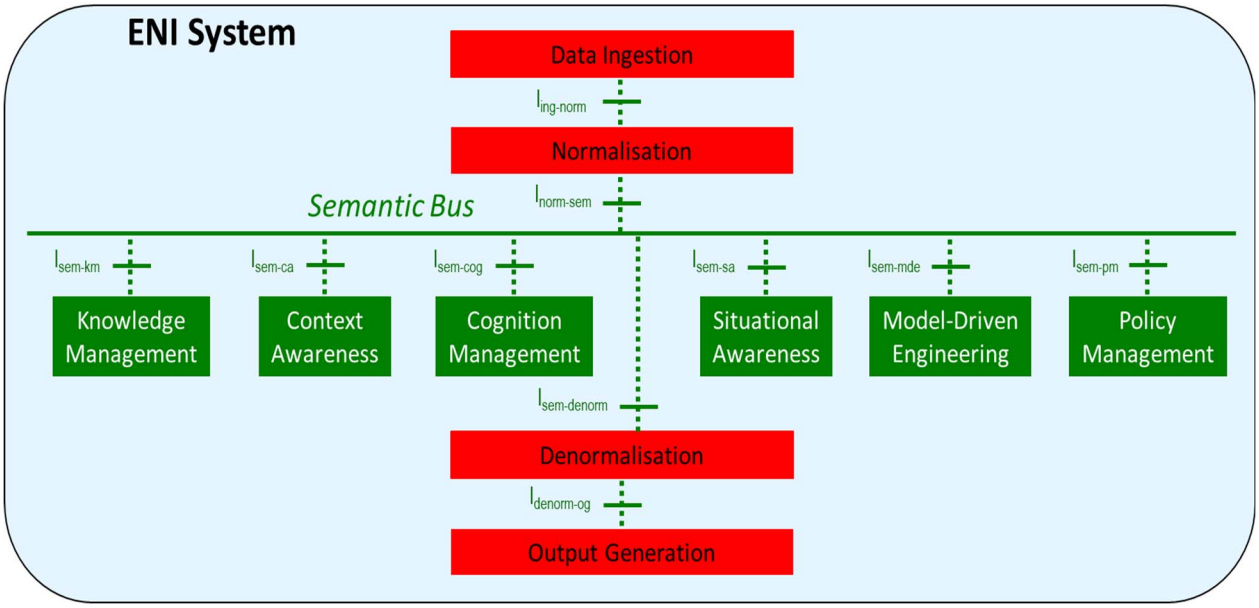

Figure 7-7: Overview of the ENI Internal Facing External Reference Points

Table 7-4 provides brief descriptions of the Internal Reference Points of ENI.

Table 7-4: ENI Internal Reference Point Overview

Name	Brief Definition	Interface Functions
$I_{ing-norm}$	Defines data and information sent by the Data Ingestion Functional Block to the Normalization Functional Block. This is a uni-directional interface.	Data from the API Broker is ingested in preparation for normalization.
$I_{norm-sem}$	Defines normalized data and information that are sent to the Semantic Bus, where subscribed Functional Blocks may consume the normalized data and information. This is a uni-directional interface.	The Semantic Bus enables any of the Internal Functional Blocks to publish data and/or subscribe to data.
I_{sem-km}	Defines the data and information received by the Knowledge Management Functional Block from the Semantic Bus, as well as data and information that the Knowledge Management Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	The Knowledge Management Functional Block shall first define subscriptions that specify the type of data and information that it wants to receive from the Semantic Bus. Then, it automatically receives information from the Semantic Bus that matches its subscriptions.

Name	Brief Definition	Interface Functions
I_{sem-ca}	Defines the data and information received by the Context-Aware Management Functional Block from the Semantic Bus, as well as data and information that the Context Awareness Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	Same procedure as above.
$I_{sem-cog}$	Defines the data and information received by the Cognition Management Functional Block from the Semantic Bus, as well as data and information that the Cognition Management Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	Same procedure as above.
I_{sem-sa}	Defines the data and information received by the Situational Awareness Functional Block from the Semantic Bus, as well as data and information that the Situational Awareness Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	Same procedure as above.
$I_{sem-mde}$	Defines the data and information received by the Model-Driven Engineering Functional Block from the Semantic Bus, as well as data and information that the Model-Driven Engineering Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	Same procedure as above.
I_{sem-pm}	Defines the data and information received by the Policy Management Functional Block from the Semantic Bus, as well as data and information that the Policy Management Functional Block publishes to the Semantic Bus. This is a bi-directional interface.	Same procedure as above.
$I_{sem-denorm}$	Defines the data and information received by the Denormalization Functional Block from the Semantic Bus. This is a uni-directional interface.	Data, information, and policies that are to be sent to external entities (e.g. the Assisted System) are sent from the Semantic Bus to the Denormalization Functional Block to begin processing to a form the external entity can understand.
$I_{denorm-og}$	Defines the data and information received by the Output Generation Functional Block from the Denormalization Functional Block. This is a uni-directional interface.	Data, information, and policies that are to be sent to external entities (e.g. the Assisted System) are sent from the Denormalization Functional Block to the Output Generation Functional Block, where they are sent to the API Broker once the generation is complete.

7.7 Internal Reference Point Definitions

7.7.1 Reference Point $I_{ing-norm}$

This Internal Reference Point is used to transfer data from the Data Ingestion Functional Block to the Data Normalization Functional Block. Ingested data may include all types of data, information, knowledge, policies, and metadata sent from the API Broker through any of the External Reference Points that supply inputs to the Data Ingestion Functional Block (see Table 7-4 and clause 7.6 for their definitions). This is a uni-directional Internal Reference Point, meaning that data for processing shall only flow from the Data Ingestion Functional Block to the Data Normalization Functional Block.

The Data Ingestion Functional Block may generate an event to the Semantic Bus that informs other ENI internal Functional Blocks of the receipt of new data.

If the Data Ingestion Functional Block is combined with the Data Normalization Functional Block, then this Internal Reference Point need not be defined.

7.7.2 Reference Point $I_{norm-sem}$

This Internal Reference Point is used to transfer normalized data from the Data Normalization Functional Block to the Semantic Bus. This is a uni-directional Internal Reference Point, meaning that data for processing shall only flow from the Data Ingestion Functional Block to the Data Normalization Functional Block.

The ENI System shall ingest the data and normalize it. If it is not possible to normalize the data, the ENI System shall report this to the Assisted System and ask for further information that defines the format, as well as expected characteristics and behaviour, of these data. Upon receipt of this information, the ENI System shall record this in its knowledge base and correct the format and content of future recommendations and commands sent to this entity to facilitate future communications.

7.7.3 Reference Point I_{sem-km}

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Knowledge Management Functional Block that the Knowledge Management Functional Block has subscribed to. The Knowledge Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.4 Reference Point I_{sem-ca}

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Context-Aware Management Functional Block that the Context-Aware Management Functional Block has subscribed to. The Context-Aware Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.5 Reference Point $I_{sem-cog}$

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Cognition Management Functional Block that the Cognition Management Functional Block has subscribed to. The Cognition Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.6 Reference Point I_{sem-sa}

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Situational Awareness Functional Block that the Situational Awareness Functional Block has subscribed to. The Situational Awareness Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.7 Reference Point $I_{sem-mde}$

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Model-Driven Engineering Functional Block that the Model-Driven Engineering Functional Block has subscribed to. The Model-Driven Engineering Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.8 Reference Point I_{sem-pm}

This Internal Reference Point is used to transfer all types of data, information, knowledge, policies, and metadata from the Semantic Bus to the Policy Management Functional Block that the Policy Management Functional Block has subscribed to. The Policy Management Functional Block may send any type of data, information, knowledge, policies, and metadata to the Semantic Bus that it deems necessary.

This is a bi-directional Internal Reference Point.

7.7.9 Reference Point $I_{sem\text{-}denorm}$

This Internal Reference Point is used to transfer data from the Semantic Bus to the Data Denormalization Functional Block. These data may be data, information, knowledge, policies, and metadata from any internal ENI Functional Block that is necessary to communicate to the Assisted System or its Designated Entity. This is a uni-directional Internal Reference Point, meaning that data for processing shall only flow from the Semantic Bus to the Data Denormalization Functional Block.

The Data Denormalization Functional Block may generate an event to the Semantic Bus that informs other ENI internal Functional Blocks of the receipt of new data.

If the Output Generation Functional Block is combined with the Data Denormalization Functional Block, then this Internal Reference Point need not be defined.

7.7.10 Reference Point $I_{denorm\text{-}out}$

This Internal Reference Point is used to transfer data from the Data Denormalization Functional Block to the Output Generation Functional Block, where it will be sent by an appropriate External Reference Point (see Figure 7-2 and clause 7.3 for their definitions). Data output may include all types of data, information, knowledge, policies, and metadata. This is a uni-directional Internal Reference Point, meaning that data for processing shall only flow from the Data Denormalization Functional Block to the Output Generation Functional Block.

The Output Generation Functional Block may generate an event to the Semantic Bus that informs other ENI internal Functional Blocks of the transmission of these data.

7.8 Internal Reference Point Protocol Specification

7.8.1 Introduction

Internal protocols need to run as fast as possible while still providing strong security. The protocols defined in clause 7.4 of the present document are also applicable for ENI internal protocols (i.e. protocols used over ENI Internal Reference Points):

- An ENI System **should** use gRPC and HTTP/2 between distributed Functional Blocks.
- An ENI System **may** use GraphQL and HTTP/1.1 for specific use cases between distributed Functional Blocks.
- An ENI System **may** use REST and HTTP/1.1 for specific use cases between distributed Functional Blocks.
- An ENI System **may** use HATEOAS and HTTP/1.1 for specific use cases between distributed Functional Blocks.

7.8.2 Generic Protocols for use with Internal Reference Points

Protocols are only required when Internal Reference Points communicate between distributed Functional Blocks. In that case, the protocols defined in clause 7.4.2 of the present document **should** be used.

7.8.3 Specific Protocols for use with Internal Reference Points

Protocols are only required when Internal Reference Points communicate between distributed Functional Blocks. In that case, the protocols defined in clause 7.4.3 of the present document **should** be used.

8 ENI API Design

8.1 Introduction

Table 7-3 in clause 7.5.2.7 summarizes the advantages and disadvantages of four popular protocols: REST, HATEOAS, GraphQL, and gRPC.

As explained in clause 7.5.2, gRPC offers many advantages compared to standard REST for creating APIs. An API architecture created using gRPC enables multiple functions created in different programming languages to work together. This is very important in ENI, as it enables different functionality in ENI to be free to use the best language implementation available. This is particularly important given the wide diversity of functionality used in ENI (e.g. AI, machine learning, telemetry, and context and situation awareness). gRPC uses the protocol buffers (Protobuf) messaging format, which is a highly efficient messaging format for serializing structured data while being platform- and language-agnostic. Transmission speed is much faster using Protobuf, since it reduces message size and is a simpler and lighter weight protocol. gRPC also uses HTTP2, which uses binary format encapsulation (instead of plain text as in HTTP1.1). In addition, HTTP2 uses three different types of streaming (see clause 7.5.2.6), which enables code optimization. gRPC has built-in library features enabling it to intelligently and quickly select which back-end server to send traffic to. It also supports connection pooling for state-related communications. Finally, gRPC enables native code generation instead of having to rely on third-party tools to generate code by using its own protoc compiler. This works in multiple languages and can be used in polyglot environments.

The main disadvantage of gRPC is that it is more difficult to implement than REST due to the fact that there is not much support yet for this API structure when it comes to third-party tools and code frameworks.

This results in the following requirements:

- An ENI System **should** use gRPC APIs for communication between it and an Assisted System.
- An ENI System **should** use gRPC APIs for communication between multiple ENI Systems.
- An ENI System **may** use GraphQL APIs for communication between it and an Assisted System.
- An ENI System **may** use GraphQL APIs for communication between multiple ENI Systems.
- An ENI System **may** use REST APIs for communication between it and an Assisted System.
- An ENI System **may** use REST APIs for communication between multiple ENI Systems.
- An ENI System **may** use HATEOAS APIs for communication between it and an Assisted System.
- An ENI System **may** use HATEOAS APIs for communication between multiple ENI Systems.

8.2 Design Goals

The following are the design goals for ENI APIs:

1) Each ENI API **shall** be created from the ENI Extended Core Model [9]. This provides inherent extensibility, and enables each ENI API to use class operations to fulfil all or part of the functionality of the API.

2) API design is an interface for *developers*. It **should** therefore be structured for use by developers. For example, instead of focusing on what a service needs to provide, ENI APIs should be easy for the developer to use. In particular:

    a) An ENI Functional Block **shall** have groups of similar APIs.

    b) Each function in an ENI Functional Block **shall** have its own set of similar APIs. This supports several principles in [i.35], such as the Single Responsibility Principle and building modular code.

    c) ENI APIs across all ENI Functional Blocks **shall** have similar naming, structure, error handling, and other functionality.

d) All ENI APIs **shall** be able to call class methods to perform all or part of their functions. Software contracts [i.35] facilitates understanding an API and specifies its behaviour clearly and concisely.

3) Each ENI API **should** be easy to learn and use. Its functionality should be easy to explain, and obvious from how it is named.

4) Each API in the ENI API **should** perform a single function (see the Single Responsibility Principle in [i.35]).

5) Each ENI API **should** be hard to use incorrectly. Initial versions may not contain all possible functionality, parameters, and other variations. Such functionality may be added later in a new version of the API.

6) Each ENI API **should** maximize information hiding [i.35].

7) Each ENI API **should** be easily discoverable.

8) Each ENI API **should** be able to be used, understood, created, tested, and debugged independently.

9) It is impossible to define a perfect API. APIs specify not just the interfaces for programmers to understand and write code against but also for computers to execute, making them brittle and difficult to change. Hence:

   a) Each ENI API **shall** support versioning.

   b) Each version of each ENI API **should** be able to be used. This makes the ENI APIs user-friendly.

   c) When an old version of an ENI API is used, the ENI API **should** inform the user that a newer version is available.

10) The ENI API users **should** be able to abort or reset operations and easily get the API back to a normal state.

11) The ENI **should** help users recognize, diagnose, and recover from errors.

12) The ENI **shall** provide documentation.

13) The ENI **should** provide context-specific help.

8.3 Methodology for Constructing APIs

8.3.1 Introduction

The construction of the ENI set of APIs consists of the following phases:

1) A developer **shall** define a set of paradigms for the ENI Reference Point (Internal or External) that a set of ENI APIs use.

2) A developer **shall** define a set of paradigms that are common for all ENI APIs. This provides a common look-and-feel for the ENI APIs, as well as standard client- and server-side applications and libraries and common authentication and authorization mechanisms.

3) Once a particular architectural style is chosen, a developer **shall** conform to applicable standards for that particular architectural style.

8.3.2 Common API Paradigms

Usability, as well as a common look-and-feel for the ENI APIs is important. Clause 8.2 **shall** be used to guide usability and a common look-and-feel.

This extends to consistent usage of standards. In particular:

1) The ENI APIs **shall** use the same authentication and authorization mechanisms.

2) For specific ENI APIs that require additional security, those ENI APIs **should** use the same additional security protection mechanisms.

3) All ENI APIs **shall** use a secure protocol, as described in clauses 7.4 and 7.8.

4) All ENI APIs **shall** use the same logging, monitoring, and tracing mechanisms to enable detailed API analytics to be gathered.

5) Each set of ENI APIs that operate in a particular ENI Functional Block **should** define limits, quotas and traffic management policies:

   a) Limits and Quotas:

      i) Limits and quotas are placed on ENI API requests to protect the ENI System from receiving more data than it can handle, and to optimize the distribution of the ENI System resources used.

      ii) Quotas are used in production systems to ensure that a customer states within their contractual terms.

      iii) Define which limits and quotas are changeable by the developer, and the range of parameters that a developer is able to use.

      iv) Define which limits and quotas are changeable by the end-user (if any), and the range of parameters that a developer is able to use.

      v) Limits may be static or dynamic. A static limit operates as a threshold that, once exceeded, results in an aborted request and the return of an appropriate error. A dynamic limit will allow a request to complete but will also return a warning. Repeated exceeding of a dynamic limit will result in aborting subsequent API requests and the return of an appropriate error.

      vi) Policies that are used in a consistent fashion if a limit or quota is exceeded.

   b) Traffic Management Policies:

      i) An ENI System **shall** implement load balancing and exponential backoff traffic management policies.

      ii) Define which traffic management policies are changeable by the developer, and the range of parameters that a developer is able to use.

      iii) Load balancing defines fine-grained configuration of how incoming traffic is distributed. A load balancer has two main parts: a frontend and a backend configuration. The frontend configuration describes the exposed public or private IP address of the load balancer. The backend configuration defines how the traffic is distributed.

      iv) The ENI API Broker **shall** provide a default API gateway load balancing mechanism.

      v) The ENI API Broker default API gateway load balancing mechanism **should** support different configurations optimized for different use cases (e.g. high availability and service chaining).

      vi) The ENI API Broker **may** support the ability for a developer to customize load balancing functionality.

      vii) Exponential backoff is the process of a client periodically retrying a failed request over an increasing amount of time. Exponential backoff increases the efficiency of bandwidth usage, reduces the number of requests required to get a successful response, and maximizes the throughput of requests in concurrent environments.

8.3.3 gRPC API Construction

This clause describes how clauses 8.3.1 and 8.3.2 of the present document are used for the specific case of gRPC APIs in an ENI System.

gRPC **should** use a resource-orientated architectural design style (see clause 7.5.2.6). Resources are named entities (e.g. an end-user service), and resource names are their identifiers. Each resource **shall** have its own unique resource name. The resource name **shall** consist of the resource ID, the ID(s) of any parent resources, and its API service name.

gRPC APIs **should** use scheme-less URIs (i.e. a URI that does not specify a protocol to use) for resource names. This enables gRPC APIs to be familiar to users of REST URL conventions. In addition, this enables their behaviour to be similar to a network file path. Finally, this enables gRPC APIs to be mapped to REST URLs.

A collection is a type of resource that contains a list of sub-resources of identical type. Collections can contain collections. For example, a directory is a collection of file resources, and a service collection could contain a collection of end-user services and a collection of ENI internal services. The resource name is organized hierarchically using collection IDs and resource IDs, separated by forward slashes. If a resource contains a sub-resource, the sub-resource's name **shall** be formed by specifying the parent resource name followed by the sub-resource's ID separated by forward slashes. For example, a resource of type *service* has a collection of services, where each service has a collection of objects.

Protocol Buffers (ProtoBuf) is a data serialization protocol that is used to exchange data. The serialization process requires that the structure of data to be exchanged is defined in a .proto file using the Protocol Buffers language (.proto file syntax); this step is also known as defining a Protocol Buffers message type (see clause 7.5.2.6). The ProtoBuf compiler then reads the .proto file, and compiles the data structure into a class in the target language, which can then be used to manipulate the data programmatically. In addition, gRPC offers a special plugin for the ProtoBuf compiler that compiles .proto files to server-side and client-side artifacts for gRPC API services, in addition to the data classes. This requires the addition of service definitions to the .proto files, which describes the gRPC APIs. A service definition contains the methods that allow consumers to call remotely, the method parameters and message formats to use when invoking those methods, and so on. Each gRPC API takes one ProtoBuf message type as input (request), and produces another ProtoBuf message type as output (response).

gRPC defines a set of standard message field definitions that **should** be used when similar concepts are needed. This ensures that the same concept has the same name and semantics across different APIs. gRPC also support a set of common request parameters available across all API methods. These parameters are known as system parameters. gRPC APIs support this feature in HTTP request headers with keys in lowercase. An example of this feature is authorization. Standard field and system parameters, as well as naming conventions, are explained in https://github.com/googleapis/googleapis.

gRPC uses .proto files to define the API and .yaml files to configure the API service. Each API service **shall** have an API directory inside an API repository. The API directory **should** follow the standard gRPC directory layout. The API directory **should** contain all API definition files and build scripts (see https://github.com/googleapis/googleapis).

gRPC uses a simple protocol-agnostic error model. This provides a consistent experience across different APIs and different error contexts (such as asynchronous, batch, or workflow errors). Errors are included in the headers, and errors **should not** exceed 2 kilobytes in size.

The error model for gRPC APIs is logically defined by google.rpc. Status, an instance of which is returned to the client when an API error occurs. Error handling for resource-orientated API design **should** use a small set of standard errors. For example, instead of defining different kinds of "not found" errors, the server uses one standard google.rpc.Code. NOT_FOUND error code and tells the client which specific resource was not found. The smaller error space increases consistency, reduces the complexity of documentation, affords better idiomatic mappings in client libraries, and reduces client logic complexity while not restricting the inclusion of actionable information.

Hence, gRPC API shall use the canonical error codes defined by google.rpc.Code. Individual APIs **shall not** define additional error codes. The error message **should** help users understand and resolve the API error easily and quickly. The following guidelines should be used when writing error messages:

1) ENI API error messages **shall** be clear and easy to understand.

2) ENI API error messages **should** be independent about the underlying service implementation and error context.

3) ENI API error messages **should** be constructed such that a technical user can respond to the error and correct it.

4) ENI API error messages **should** be brief.

5) ENI API error messages **may** include a link to additional information.

gRPC APIs **shall** define a set of standard error payloads for error details, which are defined in google/rpc/error_details.proto. These cover the most common needs for API errors, such as quota failure and invalid parameters. Like error codes, developers **should** use these standard payloads whenever possible, and should not define custom payloads unless absolutely needed.

Additional error detail types **should** only be introduced if they can assist application code to handle the errors. If the error information can only be handled by humans, the developer **should** rely on the error message content and let developers handle it manually rather than introducing additional error detail types.

Common performance paradigms include:

1) Keepalive pings **should** be used to keep HTTP/2 connections alive during periods of inactivity to allow initial RPCs to be made quickly without a delay.

2) Streaming RPCs **should** be used when handling a long-lived logical flow of data from the client-to-server, server-to-client, or in both directions. Streams can avoid continuous RPC initiation, which includes connection load balancing at the client-side, starting a new HTTP/2 request at the transport layer, and invoking a user-defined method handler on the server side. However, streams cannot be load balanced once they have started and can be hard to debug for stream failures. Streams should be used to optimize the application, not gRPC.

3) Each gRPC channel **should** use zero or more HTTP/2 connections. Each connection **should** place a limit on the number of concurrent streams. When the number of active RPCs on the connection reaches this limit, additional RPCs **shall** be queued in the client and will wait for active RPCs to finish before they are sent. Hence:

    a) a separate channel **should** be created for each area of high load in the application; or

    b) a pool of gRPC channels **should** be used to distribute RPCs over multiple connections.

If compatibility with REST APIs is desired, then the following additional semantics **should** be used:

1) **Get**. The Get method **shall** take a resource name, zero or more parameters, and return the specified resource. This method **shall** map to a GET operation. The request message field(s) receiving the resource name **shall** map to the URL path. All remaining request message fields **shall** map to the URL query parameters. A request body **shall not** be defined. The returned resource **shall** map to the entire response body.

2) **Create**. The Create method **shall** take a parent resource name, a resource, and zero or more parameters. It **shall** create a new resource under the specified parent, and **shall** return the newly created resource. This method **shall** map to a POST operation. An API supporting resource creation **shall** have a Create method for each type of resource that can be created. The request message **shall** have a field parent that specifies the parent resource name where the resource is to be created. The request message field containing the resource **shall** map to the HTTP request body. The request **may** contain a field named <resource>_id to allow callers to select a client assigned id. This **may** be inside the resource. All remaining request message fields **shall** map to the URL query parameters. The returned resource **shall** map to the entire HTTP response body. If the Create method supports client-assigned resource names and the resource already exists, the request **shall** fail with an appropriate error code signifying that the resource already exists. This parallels the design of class operations in ETSI GS ENI 019 [9].

3) **Update**. The Update method **shall** take a request message containing a resource and zero or more parameters. It **shall** update the specified resource and its properties, and **shall** return the updated resource. This method maps to either a PUT or a PATCH operation, as follows. If an Update method supports full resource update (as shown in the class operation design in ETSI GS ENI 019 [9]), this method **shall** map to a PUT operation. If an Update method supports partial resource update (as shown in the class operation design in ETSI GS ENI 019 [9]), this method **shall** map to a PATCH operation. Any Update method tht needs more advanced patching semantics **should** use a custom method. The ability to move and/or rename a resource **shall not** be included in the Update method; it **should** be defined using a custom method. The message field receiving the resource name **shall** map to the URL path. The field **may** be in the resource message itself. The request message field containing the resource **shall** map to the request body. All remaining request message fields **shall** map to the URL query parameters. The response message **shall** be the updated resource itself. If the API accepts client-assigned resource names, and a resource name is not found, then the Update method **should** fail with an appropriate error code indicating that the resource name was not found. If an Update method also supports resource creation, then a separate Create method **shall** be provided.

4) **Delete**. The Delete method **shall** take a resource name and zero or more parameters, and deletes the specified resource. The Delete method **should** return google.protobuf.Empty. This method **shall** map to a DELETE operation. An API **shall not** rely on any information returned by a Delete method, as it cannot be invoked repeatedly. The request message field(s) receiving the resource name **should** map to the URL path. All remaining request message fields **shall** map to the URL query parameters. A parameter **may** be specified to define if the resource is to be deleted immediately or at a scheduled time. A Delete operation **shall not** have a request body. If the Delete method immediately removes the resource, it **should** return an empty response. If the Delete method only marks the resource as being deleted, it **should** return the updated resource.

5) **List**. The List method **shall** take a collection name and zero or more parameters as input, and **shall** return a list of resources that match the input. This method **shall** map to a GET operation. This method **may** be used to search for resources. List is suited to data from a single collection that is bounded in size and not cached. For broader cases, the custom method Search **should** be used. The request message field(s) receiving the name of the collection whose resources are being listed **should** map to the URL path. If the collection name maps to the URL path, the last segment of the URL template (the collection ID) **shall** be a literal. All remaining request message fields **shall** map to the URL query parameters. This method **shall not** contain a request body. The response body should contain a list of resources along with optional metadata.

Custom methods **should** only be used for functionality that cannot be easily expressed via the above five methods. In general, standard methods **should** be used instead of custom methods whenever possible. A custom method **may** be associated with a resource, a collection, or a service. It **may** take an arbitrary request and return an arbitrary response, and also supports streaming requests and responses. To map to HTTP, a custom method **shall** use the following generic HTTP mapping:

"*https://service.name/v?/some/resource/name:customVerb*"

where "v?" denotes the current version of the method. The colon (":") is used instead of a forward slash ("/") in order to separate the custom verb from the resource name. This is necessary in order to support arbitrary paths. Custom methods **should** use HTTP POST since it has the most flexible semantics. An important exception is a custom method that defines Get or List semantics. Custom methods **should not** use HTTP PATCH. The request message field(s) receiving the resource name of the resource or collection with which the custom method is associated **should** map to the URL path. If the HTTP verb used for the custom method allows an HTTP request body (e.g. POST, PUT, PATCH, or a custom HTTP verb), the HTTP configuration of that custom method **shall** use the body: "*" clause and all remaining request message fields **shall** map to the HTTP request body. If the HTTP verb used for the custom method does not accept an HTTP request body (GET, DELETE), the HTTP configuration of such method **shall not** use the body clause at all, and all remaining request message fields **shall** map to the URL query parameters.

8.3.4 gRPC Integration

There are several open source community efforts that are creating a gRPC gateway for integrating gRPC APIs with other APIs, such as REST.

The ENI System may support creating a gRPC gateway. This is for further study in Release 4 of the present document (see clause 9).

8.4 Overview of API Functionality

8.4.1 Introduction

Each ENI API runs over a particular ENI External or Internal Reference Point. The applicable protocols **shall** be those described in clauses 7.4 and 7.8 of the present document.

gRPC uses a client-server architecture, where the client application can call functions on a server. gRPC defines a service on the server side that has methods that can be called remotely with their parameters and return types. The client side has a stub that provides the same methods as the one in the server side. This enables a client to call a function on the server application hosted in a different machine as if it were in a local object.

Hence, each object **may** have a set of one or more functions for each structured data object defined in the .proto file.

The following clauses describe the different functions available on each ENI External Reference Point. The functions for each ENI Internal Reference Point are for further study in Release 4.

8.4.2 External Reference Point $E_{oss\text{-}eni\text{-}dat}$

The functionality of this External Reference Point is defined in clause 7.3.1. Ten resources are defined for this External Reference Point:

1) a container for ingested data;
2) ingested data and information;
3) a container for normalized data;
4) normalized data and information;
5) a container for data with problems preventing normalization;
6) data with problems preventing normalization;
7) a container for data with solutions enabling normalization;
8) solved normalized data;
9) a container for negotiation information;
10) negotiated information.

The functions are shown in Table 8-1, which provides brief descriptions of each function.

Table 8-1: ENI External Reference Point $E_{oss\text{-}eni\text{-}dat}$ Functions from the OSS to the ENI System

Resource Name	Resource URI	RPC Method	Description
OSS Input Data	/eoss_data_in	CREATE	Creates a new IngestedOSSData resource and stores ingested data in it
		GET	Retrieves all ingested data from the IngestedOSSData resource
OSS Input Data Operations	/eoss_data_in/{ingestDataID}	GET	Retrieves a single IngestedOSSData resource
		UPDATE	Modifies a single IngestedOSSData resource
		DELETE	Deletes a single IngestedOSSData resource
OSS Normalized Data	/eoss_normalized_data	CREATE	Creates a new NormalizedOSSData resource
		GET	Retrieves all normalized data from the NormalizedOSSData resource
OSS Normalized Data Operations	/eoss_normalized_data/ {normalizedDataID}	GET	Retrieves an existing NormalizedOSSData resource
		UPDATE	Modifies an existing NormalizedOSSData resource
		DELETE	Deletes an existing NormalizedOSSData resource
OSS Normalized Data having Problems	/eoss_normalized_data/ {normalizedDataID}/problem	CREATE	Creates a new NormalizedOSSDataProblem resource
		GET	Retrieves all NormalizedOSSDataProblem resources
OSS Normalized Data having Problems Operations	/eoss_normalized_data/ {normalizedDataID}/problem /{problemID}	GET	Retrieves an individual NormalizedOSSDataProblem resource
		UPDATE	Modifies an individual NormalizedOSSDataProblem resource
		DELETE	Deletes an individual NormalizedOSSDataProblem resource
OSS Normalized Data with Solutions	/eoss_normalized_data/ {normalizedDataID}/solution	CREATE	Creates a new NormalizedOSSDataSolution resource
		GET	Retrieves all NormalizedOSSDataSolution resources
OSS Normalized Data with Solutions Operations	/eoss_normalized_data/ {normalizedDataID}/solution/ {solutionID}	GET	Retrieves an individual NormalizedOSSDataSolution resource
		UPDATE	Modifies an individual NormalizedOSSDataSolution resource
		DELETE	Deletes an individual NormalizedOSSDataSolution resource

Resource Name	Resource URI	RPC Method	Description
OSS Normalized Data with Negotiation	/eoss_normalized_data/{normalizedDataID}/negotiation	CREATE	Creates a new NormalizedOSSDataNegotiate resource
		GET	Retrieves all NormalizedOSSDataNegotiate resources
OSS Normalized Data with Negotiation Operations	/eoss_normalized_data/{normalizedDataID}/solution/{negotiationID}	GET	Retrieves an individual NormalizedOSSDataNegotiate resource
		UPDATE	Modifies an individual NormalizedOSSDataNegotiate resource
		DELETE	Deletes an individual NormalizedOSSDataNegotiate resource

8.4.3 External Reference Point $E_{oss\text{-}eni\text{-}cmd}$

The functionality of this External Reference Point is defined in clause 7.3.2. Eight resources are defined for this External Reference Point:

1) a container for recommendations and commands to be sent;
2) recommendations and commands;
3) a container for recommendations and commands that the OSS could not understand;
4) recommendations and command problems;
5) a container for recommendations and commands with solutions that the OSS accepted;
6) solved recommendations and commands;
7) a container for negotiation information;
8) negotiated information.

The functions are shown in Table 8-2, which provides brief descriptions of each function.

Table 8-2: ENI External Reference Point $E_{oss\text{-}eni\text{-}cmd}$ Functions from the ENI System to the OSS

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the OSS	/eoss_rec_cmd_out	CREATE	Creates a new RecommendCommandOSS resource and stores in it
		GET	Retrieves all recommendations and commands from the RecommendCommandOSS resource
Recommendations and Commands to be Sent to the OSS Operations	/eoss_rec_cmd_out/{recCmdID}	GET	Retrieves a single RecommendCommandOSS resource
		UPDATE	Modifies a single RecommendCommandOSS resource
		DELETE	Deletes a single RecommendCommandOSS resource
Recommendations and Commands to be Sent to the OSS having Problems	/eoss_rec_cmd_out/{recCmdID}/problem	CREATE	Creates a new RecommendCommandOSSProblem resource
		GET	Retrieves all RecommendCommandOSSProblem resources
Recommendations and Commands to be Sent to the OSS having Problems Operations	/eoss_rec_cmd_out/{recCmdID}/problem/{problemID}	GET	Retrieves an individual RecommendCommandOSSProblem
		UPDATE	Modifies an individual RecommendCommandOSSProblem
		DELETE	Deletes an individual RecommendCommandOSSProblem
Recommendations and Commands to be Sent to the OSS with Solutions	/eoss_rec_cmd_out/{recCmdID}/solution	CREATE	Creates a new RecommendCommandOSSSolution resource
		GET	Retrieves all RecommendCommandOSSSolution resources

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the OSS with Solutions Operations	/eoss_rec_cmd_out/ {recCmdID}/solution/ {solutionID}	GET	Retrieves an individual RecommendCommandOSSSolution resource
		UPDATE	Modifies an individual RecommendCommandOSSSolution resource
		DELETE	Deletes an individual RecommendCommandOSSSolution resource
Recommendations and Commands to be Sent to the OSS with Negotiation	/eoss_rec_cmd_out/ {recCmdID}/negotiation	CREATE	Creates a new RecommendCommandOSSNegotiate resource
		GET	Retrieves all RecommendCommandOSSNegotiate resources
Recommendations and Commands to be Sent to the OSS with Negotiation Operations	/eoss_rec_cmd_out/ {recCmdID}/negotiation {negotiationID}	GET	Retrieves an individual RecommendCommandOSSNegotiate resource
		UPDATE	Modifies an individual RecommendCommandOSSNegotiate resource
		DELETE	Deletes an individual RecommendCommandOSSNegotiate resource

8.4.4 External Reference Point $E_{oss\text{-}eni\text{-}pol}$

The functionality of this External Reference Point is defined in clause 7.3.3. Sixteen resources are defined for this External Reference Point:

1) a container for policies sent by the OSS;
2) OSS input policies;
3) a container for policies not understood by the ENI System;
4) ENI System policies having Problems;
5) a container for policies with solutions that the ENI System accepted;
6) ENI System solved policies;
7) a container for negotiation information for problem policies for the ENI System;
8) negotiated ENI System policy information;
9) a container for policies for sent by the ENI System to the OSS;
10) policies for the OSS;
11) a container for policies not understood by the OSS;
12) OSS policies having Problems;
13) a container for policies with solutions that the OSS accepted;
14) solved OSS policies;
15) a container for negotiation information for problem OSS policies;
16) negotiated OSS policy information.

The functions are shown in Table 8-3 and Table 8-4 for policies sent by the OSS(and received by the ENI System) and policies sent by the ENI System (and received by the OSS), respectively, along with brief descriptions of each function.

Table 8-3: ENI External Reference Point $E_{oss\text{-}eni\text{-}pol}$ for Functions Sent by the OSS to the ENI System

Resource Name	Resource URI	RPC Method	Description
Policies Sent by OSS	/eoss_policy_in	CREATE	Creates a new OSSPolicyData resource and stores policies sent by the OSS in it
		GET	Retrieves all policies from the OSSPolicyData resource
Policies Sent by OSS Operations	/eoss_policy_in/{policyID}	GET	Retrieves a single policy received from the OSS
		UPDATE	Modifies a single policy received from the OSS
		DELETE	Deletes a single policy received from the OSS
Policies Sent by OSS having Problems	/eoss_policy_in/{policyID}/ problem	CREATE	Creates a new OSSPolicyDataProblem resource for policies containing problems
		GET	Retrieves all policies sent by the OSS that the ENI System has problems understanding
Policies Sent by OSS having Problems Operations	/eoss_policy_in/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the OSS that the ENI System cannot understand
		UPDATE	Modifies an existing policy sent by the OSS that the ENI System cannot understand
		DELETE	Deletes an existing policy sent by the OSS that the ENI System cannot understand
Policies Sent by OSS with Solutions	/eoss_policy_in/{policyID}/ solution	CREATE	Creates a new OSSPolicyDataSolution resource
		GET	Retrieves all policies sent by the OSS that the ENI System can now understand
Policies Sent by OSS with Solutions Operations	/eoss_policy_in/{policyID}/ solution/{solutionID}	GET	Retrieves an existing policy sent by the OSS that the ENI System can now understand
		UPDATE	Modifies an existing policy sent by the OSS that the ENI System can now understand
		DELETE	Deletes an existing policy sent by the OSS that the ENI System can now understand
Policies Sent by OSS with Negotiation	/eoss_policy_in/{policyID}/ negotiation	CREATE	Creates a new OSSPolicyDataNegotiate resource
		GET	Retrieves all negotiation information for all policies sent by the OSS to the ENI System to resolve problems
Policies Sent by OSS with Negotiation Operations	/eoss_policy_in/{policyID}/ negotiation/{negotiationID}	GET	Retrieves negotiation information for a selected policy sent from the OSS that the ENI System can now understand
		UPDATE	Modifies negotiation information for a selected policy sent from the OSS that the ENI System can now understand
		DELETE	Deletes negotiation information for a selected policy sent from the OSS that the ENI System can now understand

Table 8-4: ENI External Reference Point $E_{oss\text{-}eni\text{-}pol}$ for Functions Sent by the ENI System to the OSS

Resource Name	Resource URI	RPC Method	Description
Policies Sent by the ENI System	/eoss_policy_out	CREATE	Creates a new ENIOSSPolicyData resource and stores policies created by the ENI System in it
		GET	Retrieves all policies from the ENIOSSPolicyData resource
Policies Sent by the ENI System Operations	/eoss_policy_out/{policyID}	GET	Retrieves a single policy created by the ENI System
		UPDATE	Modifies a single policy created by the ENI System
		DELETE	Deletes a single policy created by the ENI System
Policies Sent by the ENI System having Problems	/eoss_policy_out/{policyID}/ problem	CREATE	Creates a new ENIOSSPolicyDataProblem resource
		GET	Retrieves all policies created by the ENI System that the OSS has problems understanding
Policies Sent by the ENI System having Problems Operations	/eoss_policy_out/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the ENI System that the OSS cannot understand
		UPDATE	Modifies an existing policy sent by the ENI System that the OSS cannot understand
		DELETE	Deletes an existing policy sent by the ENI System that the OSS cannot understand
Policies Sent by the ENI System with Solutions	/eoss_policy_out/{policyID}/ solution	CREATE	Creates a new ENIOSSPolicyDataSolution resource
		GET	Retrieves all policies sent by the ENI System that the OSS can now understand

Resource Name	Resource URI	RPC Method	Description
Policies Sent by the ENI System with Solutions Operations	/eoss_policy_out/{policyID}/solution/{solutionID}	GET	Retrieves an existing policy sent by the ENI System that the OSS can now understand
		UPDATE	Modifies an existing policy sent by the ENI System that the OSS can now understand
		DELETE	Deletes an existing policy sent by the ENI System that the OSS can now understand
Policies Sent by the ENI System with Negotiation	/eoss_policy_out/{policyID}/negotiation	CREATE	Creates a new ENIOSSPolicyNegotiate resource
		GET	Retrieves all negotiation information for all policies created by the ENI System for the OSS to resolve problems
Policies Sent by the ENI System with Negotiation	/eoss_policy_out/{policyID}/negotiation/{negotiationID}	GET	Retrieves an existing policy sent by the ENI System to the OSS to resolve problems that the OSS can now understand
		UPDATE	Modifies negotiation information for a selected policy sent by the ENI System that the OSS can now understand
		DELETE	Deletes negotiation information for a selected policy sent by the ENI System that the OSS can now understand

8.4.5 External Reference Point $E_{app\text{-}eni\text{-}ctx}$

The functionality of this External Reference Point is defined in clause 7.3.4. Twenty resources are defined for this External Reference Point, ten resources from the application to the ENI System, and ten resources from the ENI System to the application. The ten resources from the Application to the ENI System are:

1) container for ingested information;
2) ingested context- and situation-aware information;
3) container for processing and normalizing context- and situation-aware information;
4) processed and normalized context- and situation-aware information;
5) container for processing and normalizing context- and situation-aware information from the application that the ENI System cannot understand;
6) processed and normalized context- and situation-aware information;
7) container for processing and normalizing context- and situation-aware information from the application that the ENI System can now understand;
8) processed and normalized context- and situation-aware information;
9) container for negotiation information for resolving problems;
10) negotiated application policy information.

The ten resources from the ENI System to the Application are:

1) container for ingested information;
2) ingested context- and situation-aware information;
3) container for processing and normalizing context- and situation-aware information;
4) processed and normalized context- and situation-aware information;
5) container for processing and normalizing context- and situation-aware information from the ENI System that the application cannot understand;
6) processed and normalized context- and situation-aware information;

7) container for processing and normalizing context- and situation-aware information from the ENI System that the application can now understand;

8) processed and normalized context- and situation-aware information;

9) container for negotiation information for resolving problems;

10) negotiated application policy information.

The functions are shown in Table 8-5 and Table 8-6, which provides brief descriptions of each function in each table.

Table 8-5: ENI External Reference Point $E_{app\text{-}eni\text{-}ctx}$ Functions Sent by the Application to ENI System

Resource Name	Resource URI	RPC Method	Description
App Input Data for the ENI System	/eapp_context_situation_in	CREATE	Creates a new CtxtSitApp resource and stores ingested data in it
		GET	Retrieves all ingested data from the CtxtSitApp resource
App Input Data for the ENI System Operations	/eapp_context_situation_in/ {contexSituationtID}	GET	Retrieves a single ingested CtxtSitApp resource
		UPDATE	Modifies a single ingested CtxtSitApp resource
		DELETE	Deletes a single ingested CtxtSitApp resource
App Input Data for the ENI System that is Normalized	/eapp_context_situation_in/ {contexSituationtID}normalized	CREATE	Creates a new CtxtSitAppNormalized resource
		GET	Retrieves all normalized data from the CtxtSitAppNormalized resource
App Input Data for the ENI System that is Normalized Operations	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}	GET	Retrieves an existing CtxtSitAppNormalized resource
		UPDATE	Modifies an existing CtxtSitAppNormalized resource
		DELETE	Deletes an existing CtxtSitAppNormalized resource
App Input Data Normalized having Problems	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/problem	CREATE	Creates a new CtxtSitAppNormalizedProblem resource
		GET	Retrieves all CtxtSitAppNormalizedProblem resources that the ENI System could not understand
App Input Data Normalized having Problems Operations	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/problem/ {problemID}	GET	Retrieves an individual CtxtSitAppNormalizedProblem resource
		UPDATE	Modifies an individual CtxtSitAppNormalizedProblem resource
		DELETE	Deletes an individual CtxtSitAppNormalizedProblem resource
App Input Data Normalized with Solutions	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/solution	CREATE	Creates a new CtxtSitAppNormalizedSolution resource
		GET	Retrieves all CtxtSitAppNormalizedSolution resources that the ENI System can now understand
App Input Data Normalized with Solutions Operations	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/solution/ {solutionID}	GET	Retrieves an individual normalized data with solutions
		UPDATE	Modifies an individual normalized data with solutions
		DELETE	Deletes an individual normalized data with solutions
App Input Data Normalized with Negotiation	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/negotiation	CREATE	Creates a new CtxtSitAppNormalizedNegotiate resource
		GET	Retrieves all CtxtSitAppNormalizedNegotiate data
App Input Data Normalized with Negotiation Operations	/eapp_context_situation_in/ {contexSituationtID}normalized/ {normalizedID}/negotiation/ {negotiationID}	GET	Retrieves an individual CtxtSitAppNormalizedNegotiate resource
		UPDATE	Modifies an individual CtxtSitAppNormalizedNegotiate resource
		DELETE	Deletes an individual CtxtSitAppNormalizedNegotiate resource

Table 8-6: ENI External Reference Point $E_{app\text{-}eni\text{-}ctx}$ Functions Sent from the ENI System to Application

Resource Name	Resource URI	RPC Method	Description
ENI System Data for an Application	/eapp_context_situation_out	CREATE	Creates a new CtxtSitENI resource and stores ingested data in it
		GET	Retrieves all ingested data from the CtxtSitENI resource
ENI System Data for an Application Operations	/eapp_context_situation_out/ {contexSituationtID}	GET	Retrieves a single ingested CtxtSitENI resource
		UPDATE	Modifies a single ingested CtxtSitENI resource
		DELETE	Deletes a single ingested CtxtSitENI resource
ENI System Data for the App that is Normalized	/eapp_context_situation_out/ {contexSituationtID}normalized	CREATE	Creates a new CtxtSitENINormalized resource
		GET	Retrieves all normalized data from the CtxtSitENINormalized resource
ENI System Data for the App that is Normalized Operations	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}	GET	Retrieves an existing CtxtSitENINormalized resource
		UPDATE	Modifies an existing CtxtSitENINormalized resource
		DELETE	Deletes an existing CtxtSitENINormalized resource
ENI System Data for the App Normalized having Problems	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/problem	CREATE	Creates a new CtxtSitENINormalizedProblem resource
		GET	Retrieves all CtxtSitENINormalizedProblem resources that the Application could not understand
ENI System Data for the App Normalized having Problems Operations	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/problem/ {problemID}	GET	Retrieves an individual CtxtSitENINormalizedProblem resource
		UPDATE	Modifies an individual CtxtSitENINormalizedProblem resource
		DELETE	Deletes an CtxtSitENINormalizedProblem resource
ENI System Data for the App Normalized with Solutions	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/solution	CREATE	Creates a new CtxtSitENINormalizedSolution resource
		GET	Retrieves all CtxtSitENINormalizedSolution resources that the Application can now understand
ENI System Data for the App Normalized with Solutions Operations	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/solution/ {solutionID}	GET	Retrieves an individual CtxtSitENINormalizedSolution resource
		UPDATE	Modifies an individual CtxtSitENINormalizedSolution resource
		DELETE	Deletes an individual CtxtSitENINormalizedSolution resource
ENI System Data for the App Normalized with Negotiation	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/negotiation	CREATE	Creates a new CtxtSitENINormalizedNegotiate resource
		GET	Retrieves all CtxtSitENINormalizedNegotiate data
ENI System Data for the App Normalized with Negotiation Operations	/eapp_context_situation_out/ {contexSituationtID}normalized/ {normalizedID}/negotiation/ {negotiationID}	GET	Retrieves an individual CtxtSitENINormalizedNegotiate resource
		UPDATE	Modifies an individual CtxtSitENINormalizedNegotiate resource
		DELETE	Deletes an individual CtxtSitENINormalizedNegotiate resource

8.4.6 External Reference Point $E_{app\text{-}eni\text{-}oth}$

The functionality of this External Reference Point is defined in clause 7.3.5. Twenty resources are defined for this External Reference Point, ten resources from the application to the ENI System, and ten resources from the ENI System to the application. The ten resources from the Application to the ENI System are:

1) container for ingested information;
2) ingested information;
3) container for processing and normalizing information;
4) processed and normalized information;
5) container for processing and normalizing information from the application that the ENI System cannot understand;
6) processed and normalized information;
7) container for processing and normalizing information from the application that the ENI System can now understand;
8) processed and normalized information;
9) container for negotiation information for resolving problems;
10) negotiated application information.

The ten resources from the ENI System to the Application are:

1) container for ingested information;
2) ingested information;
3) container for processing and normalizing information;
4) processed and normalized information;
5) container for processing and normalizing information from the ENI System that the application cannot understand;
6) processed and normalized information;
7) container for processing and normalizing information from the ENI System that the application can now understand;
8) processed and normalized information;
9) container for negotiation information for resolving problems that the application cannot understand;
10) negotiated application information.

The functions are shown in Table 8-7 and Table 8-8, which provides brief descriptions of each function.

Table 8-7: ENI External Reference Point $E_{app\text{-}eni\text{-}oth}$ Functions from Application to ENI System

Resource Name	Resource URI	RPC Method	Description
App Other Input Data for the ENI System	/eapp_data_in_other	CREATE	Creates a new OtherAppData resource and stores ingested data in it
		GET	Retrieves all ingested data from the OtherAppData resource
App Other Input Data for the ENI System Operations	/eapp_data_in_other/{otherDataID}	GET	Retrieves a single OtherAppData item
		UPDATE	Modifies a single OtherAppData resource
		DELETE	Deletes a single OtherAppData resource
App Other Input Data for the ENI System Normalized Data	/eapp_data_in_other/{otherDataID}/ normalized	CREATE	Creates a new OtherAppDataNormal resource
		GET	Retrieves all normalized data from the OtherAppDataNormal resource
App Other Input Data for the ENI System Normalized Operations	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}	GET	Retrieves an existing OtherAppDataNormal resource
		UPDATE	Modifies an existing OtherAppDataNormal resource
		DELETE	Deletes an existing OtherAppDataNormal resource
App Other Input Data for the ENI System Normalized having Problems	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ problem	CREATE	Creates a new OtherAppDataNormalProblem resource
		GET	Retrieves all OtherAppDataNormalProblem resources
App Other Input Data for the ENI System Normalized having Problems Operations	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ problem/{problemID}	GET	Retrieves an individual OtherAppDataNormalProblem resource
		UPDATE	Modifies an individual OtherAppDataNormalProblem resource
		DELETE	Deletes an individual OtherAppDataNormalProblem resource
App Other Input Data for the ENI System Normalized with Solutions	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ solution	CREATE	Creates a new OtherAppDataNormalSolution resource
		GET	Retrieves all OtherAppDataNormalSolution resources
App Other Input Data for the ENI System Normalized with Solutions Operations	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ solution/{solutionID}	GET	Retrieves an individual OtherAppDataNormalSolution resource
		UPDATE	Modifies an individual OtherAppDataNormalSolution resource
		DELETE	Deletes an individual OtherAppDataNormalSolution resource
App Other Input Data for the ENI System Normalized with Negotiation	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ negotiation	CREATE	Creates a new OtherAppDataNormalNegotiate resource
		GET	Retrieves all OtherAppDataNormalNegotiate resources
App Other Input Data for the ENI System Normalized with Negotiation Operations	/eapp_data_in_other/{otherDataID}/ normalized/{normalizedDataID}/ negotiation/{negotiationID}	GET	Retrieves an individual OtherAppDataNormalNegotiate resource
		UPDATE	Modifies an individual OtherAppDataNormalNegotiate resource
		DELETE	Deletes an individual OtherAppDataNormalNegotiate resource

Table 8-8: ENI External Reference Point $E_{app\text{-}eni\text{-}oth}$ Functions from ENI System to Application

Resource Name	Resource URI	RPC Method	Description
ENI System Other Input Data for the App	/eapp_data_out_other	CREATE	Creates a new OtherENIData resource and stores ingested data in it
		GET	Retrieves all ingested data from the OtherENIData resource
ENI System Other Input Data for the App Operations	/eapp_data_out_other/{otherDataID}	GET	Retrieves a single OtherENIData item
		UPDATE	Modifies a single OtherENIData resource
		DELETE	Deletes a single OtherENIData resource
ENI System Other Input Data for the App Normalized Data	/eapp_data_out_other/{otherDataID}/ normalized	CREATE	Creates a new OtherENIDataNormal resource
		GET	Retrieves all normalized data from the OtherENIDataNormal resource
ENI System Other Input Data for the App Normalized Operations	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}	GET	Retrieves an existing OtherENIDataNormal resource
		UPDATE	Modifies an existing OtherENIDataNormal resource
		DELETE	Deletes an existing OtherENIDataNormal resource
ENI System Other Input Data for the App Normalized having Problems	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ problem	CREATE	Creates a new OtherENIDataNormalProblem resource
		GET	Retrieves all OtherENIDataNormalProblem resources
ENI System Other Input Data for the App Normalized having Problems Operations	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ problem/{problemID}	GET	Retrieves an individual OtherENIDataNormalProblem resource
		UPDATE	Modifies an individual OtherENIDataNormalProblem resource
		DELETE	Deletes an individual OtherENIDataNormalProblem resource
ENI System Other Input Data for the App Normalized with Solutions	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ solution	CREATE	Creates a new OtherENIDataNormalSolution resource
		GET	Retrieves all OtherENIDataNormalSolution resources
ENI System Other Input Data for the App Normalized with Solutions Operations	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ solution/{solutionID}	GET	Retrieves an individual OtherENIDataNormalSolution resource
		UPDATE	Modifies an individual OtherENIDataNormalSolution resource
		DELETE	Deletes an individual OtherENIDataNormalSolution resource
ENI System Other Input Data for the App Normalized with Negotiation	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ negotiation	CREATE	Creates a new OtherENIDataNormalNegotiate resource
		GET	Retrieves all OtherENIDataNormalNegotiate resources

Resource Name	Resource URI	RPC Method	Description
ENI System Other Input Data for the App Normalized with Negotiation Operations	/eapp_data_out_other/{otherDataID}/ normalized/{normalizedDataID}/ negotiation/{negotiationID}	GET	Retrieves an individual OtherENIDataNormalNegotiate resource
		UPDATE	Modifies an individual OtherENIDataNormalNegotiate resource
		DELETE	Deletes an individual OtherENIDataNormalNegotiate resource

8.4.7 External Reference Point $E_{app\text{-}eni\text{-}kno}$

The functionality of this External Reference Point is defined in clause 7.3.6. Twenty resources are defined for this External Reference Point, ten resources from the application to the ENI System, and ten resources from the ENI System to the application. The ten resources from the Application to the ENI System are:

1) container for ingested knowledge information;
2) ingested knowledge information;
3) container for processing and normalizing knowledge information;
4) processed and normalized knowledge information;
5) container for processing and normalizing knowledge information from the application that the ENI System cannot understand;
6) processed and normalized knowledge information;
7) container for processing and normalizing knowledge information from the application that the ENI System can now understand;
8) processed and normalized knowledge information;
9) container for negotiation information for resolving problems understanding knowledge from the application;
10) negotiated application knowledge information.

The ten resources from the ENI System to the Application are:

1) container for ingested knowledge information;
2) ingested knowledge information;
3) container for processing and normalizing knowledge information;
4) processed and normalized knowledge information;
5) container for processing and normalizing knowledge information from the ENI System that the application cannot understand;
6) processed and normalized knowledge information;
7) container for processing and normalizing knowledge information from the ENI System that the application can now understand;
8) processed and normalized knowledge information;
9) container for negotiation knowledge information for resolving problems;
10) negotiated knowledge information.

The functions are shown in Table 8-9 and Table 8-10, which provides brief descriptions of each function.

Table 8-9: ENI External Reference Point $E_{app\text{-}eni\text{-}kno}$ Functions from the Application to the ENI System

Resource Name	Resource URI	RPC Method	Description
App Knowledge Data for the ENI System	/eapp_knowledge_in	CREATE	Creates a new KnowledgeApp resource and stores ingested data in it
		GET	Retrieves all ingested data from the KnowledgeApp resource
App Knowledge Data for the ENI System Operations	/eapp_ knowledge_in/ {knowledgeID}	GET	Retrieves a single KnowledgeApp resource
		UPDATE	Modifies a single KnowledgeApp resource
		DELETE	Deletes a single KnowledgeApp resource
App Knowledge Data for the ENI System Normalized Data	/eapp_ knowledge_in/ {knowledgeID}/normalized	CREATE	Creates a new KnowledgeAppNormalized resource
		GET	Retrieves all normalized data and stores in an existing KnowledgeAppNormalized resource
App Knowledge Data for the ENI System Normalized Data Operations	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}	GET	Retrieves an existing KnowledgeAppNormalized resource
		UPDATE	Modifies an existing KnowledgeAppNormalized resource
		DELETE	Deletes an existing KnowledgeAppNormalized resource
App Knowledge Data for the ENI System Normalized Data having Problems	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/problem	CREATE	Creates a new NormalizedAppProblem resource
		GET	Retrieves all NormalizedAppProblem resources
App Knowledge Data for the ENI System Normalized Data having Problems Operations	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/problem/ {problemID}	GET	Retrieves an individual NormalizedAppProblem resource
		UPDATE	Modifies an individual NormalizedAppProblem resource
		DELETE	Deletes an individual NormalizedAppProblem resource
App Knowledge Data for the ENI System Normalized Data with Solutions	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/solution	CREATE	Creates a new NormalizedAppSolution resource
		GET	Retrieves all NormalizedAppSolution resources
App Knowledge Data for the ENI System Normalized Data with Solutions Operations	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/solution/ {solutionID}	GET	Retrieves an individual NormalizedAppSolution resource
		UPDATE	Modifies an individual NormalizedAppSolution resource
		DELETE	Deletes an individual NormalizedAppSolution resource
App Knowledge Data for the ENI System Normalized Data with Negotiation	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/ negotiation	CREATE	Creates a new NormalizedAppNegotiate resource
		GET	Retrieves all NormalizedAppNegotiate resources

Resource Name	Resource URI	RPC Method	Description
App Knowledge Data for the ENI System Normalized Data with Negotiation Operations	/eapp_ knowledge_in/ {knowledgeID}/normalized {normalizedDataID}/ negotiation/{negotiation ID}	GET	Retrieves an individual NormalizedAppNegotiate resource
		UPDATE	Modifies an individual NormalizedAppNegotiate resource
		DELETE	Deletes an individual NormalizedAppNegotiate resource

Table 8-10: ENI External Reference Point $E_{app\text{-}eni\text{-}kno}$ Functions from the ENI System to the Application

Resource Name	Resource URI	RPC Method	Description
ENI System Knowledge Data for the App	/eapp_knowledge_out	CREATE	Creates a new KnowledgeENI resource and stores ingested data in it
		GET	Retrieves all ingested data from the KnowledgeENI resource
ENI System Knowledge Data for the App Operations	/eapp_ knowledge_ out/ {knowledgeID}	GET	Retrieves a single KnowledgeENI resource
		UPDATE	Modifies a single KnowledgeENI resource
		DELETE	Deletes a single KnowledgeENI resource
ENI System Knowledge Data for the App Normalized Data	/eapp_ knowledge_out/ {knowledgeID}/normalized	CREATE	Creates a new KnowledgeENI resource
		GET	Retrieves all normalized data and stores in an existing KnowledgeENINormalized resource
ENI System Knowledge Data for the App Normalized Data Operations	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}	GET	Retrieves an existing KnowledgeENINormalized resource
		UPDATE	Modifies an existing KnowledgeENINormalized resource
		DELETE	Deletes an existing KnowledgeENINormalized resource
ENI System Knowledge Data for the App Normalized Data having Problems	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/problem	CREATE	Creates a new NormalizedENIProblem resource
		GET	Retrieves all NormalizedENIProblem resources
ENI System Knowledge Data for the App Normalized Data having Problems Operations	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/problem/ {problemID}	GET	Retrieves an individual NormalizedENIProblem resource
		UPDATE	Modifies an individual NormalizedENIProblem resource
		DELETE	Deletes an individual NormalizedENIProblem resource
ENI System Knowledge Data for the App Normalized Data with Solutions	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/solution	CREATE	Creates a new NormalizedENISolution resource
		GET	Retrieves all NormalizedENISolution resources

Resource Name	Resource URI	RPC Method	Description
ENI System Knowledge Data for the App Normalized Data with Solutions Operations	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/solution/ {solutionID}	GET	Retrieves an individual NormalizedENISolution resource
		UPDATE	Modifies an individual NormalizedENISolution resource
		DELETE	Deletes an individual NormalizedENISolution resource
ENI System Knowledge Data for the App Normalized Data with Negotiation	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/ negotiation	CREATE	Creates a new NormalizedENINegotiate resource
		GET	Retrieves all NormalizedENINegotiate resource
ENI System Knowledge Data for the App Normalized Data with Negotiation Operations	/eapp_ knowledge_out/ {knowledgeID}/normalized {normalizedDataID}/ negotiation/{negotiationID}	GET	Retrieves an individual NormalizedENINegotiate resource
		UPDATE	Modifies an individual NormalizedENINegotiate resource
		DELETE	Deletes an individual NormalizedENINegotiate resource

8.4.8 External Reference Point $E_{app\text{-}eni\text{-}pol}$

The functionality of this External Reference Point is defined in clause 7.3.7. Twenty resources are defined for this External Reference Point, ten resources from the application to the ENI System, and ten resources from the ENI System to the application. The ten resources from the Application to the ENI System are:

1) container for ingested policy information;
2) ingested policy information;
3) container for processing and normalizing policy information;
4) processed and normalized policy information;
5) container for processing and normalizing policy information from the application that the ENI System cannot understand;
6) processed and normalized policy information;
7) container for processing and normalizing policy information from the application that the ENI System can now understand;
8) processed and normalized policy information;
9) container for negotiation information for resolving problems understanding policies from the application;
10) negotiated application policy information.

The ten resources from the ENI System to the Application are:

1) container for ingested policy information;
2) ingested policy information;
3) container for processing and normalizing policy information;
4) processed and normalized policy information;

5) container for processing and normalizing policy information from the ENI System that the application cannot understand;

6) processed and normalized policy information;

7) container for processing and normalizing policy information from the ENI System that the application can now understand;

8) processed and normalized policy information;

9) container for negotiation policy information for resolving problems;

10) negotiated policy information.

The functions are shown in Table 8-11 and Table 8-12, which provides brief descriptions of each function.

Table 8-11: ENI External Reference Point $E_{app\text{-}eni\text{-}pol}$ Functions from the Application to the ENI System

Resource Name	Resource URI	RPC Method	Description
App Policy Data for the ENI System	/eapp_policy_in	CREATE	Creates a new AppPolicyData resource and stores ingested data in it
		GET	Retrieves all ingested data from the AppPolicyData resource
App Policy Data for the ENI System Operations	/eapp_ policy_in/{policyID}	GET	Retrieves a single AppPolicyData resource
		UPDATE	Modifies a single AppPolicyData resource
		DELETE	Deletes a single AppPolicyData resource
App Policy Data for the ENI System Normalized Data	/eapp_policy_in/{policyID}/ normalized	CREATE	Creates a new AppPolicyDataNormalized resource
		GET	Retrieves all normalized data and stores in an existing AppPolicyDataNormalized resource
App Policy Data for the ENI System Normalized Data Operations	/eapp_policy_in/{policyID}/ normalized/{normalizedID}	GET	Retrieves an existing AppPolicyDataNormalized resource
		UPDATE	Modifies an existing AppPolicyDataNormalized resource
		DELETE	Deletes an existing AppPolicyDataNormalized resource
App Policy Data for the ENI System Normalized Data having Problems	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ problem	CREATE	Creates a new AppPolicyDataNormalizedProblem resource
		GET	Retrieves all AppPolicyDataNormalizedProblem resources
App Policy Data for the ENI System Normalized Data having Problems Operations	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ problem/{problemID}	GET	Retrieves an individual AppPolicyDataNormalizedProblem resource
		UPDATE	Modifies an individual AppPolicyDataNormalizedProblem resource
		DELETE	Deletes an individual AppPolicyDataNormalizedProblem resource
App Policy Data for the ENI System Normalized Data with Solutions	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ solution	CREATE	Creates a new AppPolicyDataNormalizeSolution resource
		GET	Retrieves all AppPolicyDataNormalizeSolution resources
App Policy Data for the ENI System Normalized Data with Solutions Operations	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ solution/{solutionID}	GET	Retrieves an individual AppPolicyDataNormalizeSolution resource
		UPDATE	Modifies an individual AppPolicyDataNormalizeSolution resource
		DELETE	Deletes an individual AppPolicyDataNormalizeSolution resource

Resource Name	Resource URI	RPC Method	Description
App Policy Data for the ENI System Normalized Data with Negotiation	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ negotiation	CREATE	Creates a new AppPolicyDataNormalizeNegotiate resource
		GET	Retrieves all AppPolicyDataNormalizeNegotiate resources
App Policy Data for the ENI System Normalized Data with Negotiation Operations	/eapp_policy_in/{policyID}/ normalized/{normalizedID}/ negotiation/{negotiationID}	GET	Retrieves an individual AppPolicyDataNormalizeNegotiate resource
		UPDATE	Modifies an individual AppPolicyDataNormalizeNegotiate resource
		DELETE	Deletes an individual AppPolicyDataNormalizeNegotiate resource

Table 8-12: ENI External Reference Point $E_{app\text{-}eni\text{-}pol}$ Functions from the ENI System to the Application

Resource Name	Resource URI	RPC Method	Description
ENI System Policy for the App	/eapp_ policy_out	CREATE	Creates a new ENIAppPolicyData resource and stores ingested data in it
		GET	Retrieves all ENIAppPolicyData resources
ENI System Policy for the App Operations	/eapp_policy_out/{policyID}	GET	Retrieves a single ENIAppPolicyData resource
		UPDATE	Modifies a single ENIAppPolicyData resource
		DELETE	Deletes a single ENIAppPolicyData resource
ENI System Policy for the App Normalized Data	/eapp_policy_out/{policyID}/ normalized	CREATE	Creates a new ENIAppPolicyDataNormalized resource
		GET	Retrieves all ENIAppPolicyDataNormalized resources
ENI System Policy for the App Normalized Data Operations	/eapp_policy_out/{policyID}/ normalized/{normalizedID}	GET	Retrieves an existing ENIAppPolicyDataNormalized resource
		UPDATE	Modifies an existing ENIAppPolicyDataNormalized resource
		DELETE	Deletes an existing ENIAppPolicyDataNormalized resource
ENI System Policy for the App Normalized Data having Problems	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ problem	CREATE	Creates a new ENIAppPolicyDataNormalizedProblem resource
		GET	Retrieves all ENIAppPolicyDataNormalizedProblem resources
ENI System Policy for the App Normalized Data having Problems Operations	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ problem/{problemID}	GET	Retrieves an individual ENIAppPolicyDataNormalizedProblem resource
		UPDATE	Modifies an individual ENIAppPolicyDataNormalizedProblem resource
		DELETE	Deletes an individual ENIAppPolicyDataNormalizedProblem resource
ENI System Policy for the App Normalized Data with Solutions	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ solution	CREATE	Creates a new ENIAppPolicyDataNormalizedSolution resource
		GET	Retrieves all ENIAppPolicyDataNormalizedSolution resources
ENI System Policy for the App Normalized Data with Solutions Operations	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ solution/{solutionID}	GET	Retrieves an individual ENIAppPolicyDataNormalizedSolution resource
		UPDATE	Modifies an individual ENIAppPolicyDataNormalizedSolution resource
		DELETE	Deletes an individual ENIAppPolicyDataNormalizedSolution resource

Resource Name	Resource URI	RPC Method	Description
ENI System Policy for the App Normalized Data with Negotiation	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ negotiation	CREATE	Creates a new ENIAppPolicyDataNormalizedNegotiate resources
		GET	Retrieves all ENIAppPolicyDataNormalizedNegotiate resources
ENI System Policy for the App Normalized Data with Negotiation Operations	/eapp_policy_out/{policyID}/ normalized/{normalizedID}/ negotiation/{negotiationID}	GET	Retrieves an individual ENIAppPolicyDataNormalizedNegotiate resource
		UPDATE	Modifies an individual ENIAppPolicyDataNormalizedNegotiate resource
		DELETE	Deletes an individual ENIAppPolicyDataNormalizedNegotiate resource

8.4.9 External Reference Point $E_{bss\text{-}eni\text{-}dat}$

The functionality of this External Reference Point is defined in clause 7.3.8. Ten resources are defined for this External Reference Point:

1) container for ingested BSS information;
2) ingested BSS information;
3) container for processing and normalizing BSS information;
4) processed and normalized BSS information;
5) container for processing and normalizing BSS information that the ENI System cannot understand;
6) processed and normalized BSS information;
7) container for processing and normalizing BSS information that the ENI System can now understand;
8) processed and normalized BSS information;
9) container for negotiation information for resolving problems understanding BSS information;
10) negotiated BSS information.

The functions are shown in Table 8-13, which provides brief descriptions of each function.

Table 8-13: ENI External Reference Point $E_{bss\text{-}eni\text{-}dat}$ Functions from the BSS to the ENI System

Resource Name	Resource URI	RPC Method	Description
BSS Input Data	/ebss_data_in	CREATE	Creates a new IngestedBSSData resource and stores ingested data in it
		GET	Retrieves all ingested data from the IngestedBSSData resource
BSS Input Data Operations	/ebss_data_in/{ingestDataID}	GET	Retrieves a single IngestedBSSData resource
		UPDATE	Modifies a single IngestedBSSData resource
		DELETE	Deletes a single IngestedBSSData resource
BSS Normalized Data	/ebss_data_in/{ingestDataID}/ normalized	CREATE	Creates a new NormalizedBSSData resource
		GET	Retrieves all NormalizedBSSData resources
BSS Normalized Data Operations	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID}	GET	Retrieves an existing NormalizedBSSData resource
		UPDATE	Modifies an existing NormalizedBSSData resource
		DELETE	Deletes an existing NormalizedBSSData resource

Resource Name	Resource URI	RPC Method	Description
BSS Normalized Data having Problems	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ problem	CREATE	Creates a new NormalizedBSSDataProblem resource
		GET	Retrieves all NormalizedBSSDataProblem resources
BSS Normalized Data having Problems Operations	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ problem/{problemID}	GET	Retrieves an individual NormalizedBSSDataProblem resource
		UPDATE	Modifies an individual NormalizedBSSDataProblem resource
		DELETE	Deletes an individual NormalizedBSSDataProblem resource
BSS Normalized Data with Solutions	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ solution	CREATE	Creates a new NormalizedBSSDataSolution resource
		GET	Retrieves all NormalizedBSSDataSolution resources
BSS Normalized Data with Solutions Operations	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ solution/{solutionID}	GET	Retrieves an individual NormalizedBSSDataSolution resource
		UPDATE	Modifies an individual NormalizedBSSDataSolution resource
		DELETE	Deletes an individual NormalizedBSSDataSolution resource
BSS Normalized Data with Negotiation	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ negotiation	CREATE	Creates a new NormalizedBSSDataNegotiation resource
		GET	Retrieves all NormalizedBSSDataNegotiation resources
BSS Normalized Data with Negotiation Operations	/ebss_data_in/{ingestDataID}/ normalized/{normalizedDataID }/ negotiation/{negotiationID}	GET	Retrieves an individual NormalizedBSSDataNegotiation resource
		UPDATE	Modifies an individual NormalizedBSSDataNegotiation resource
		DELETE	Deletes an individual NormalizedBSSDataNegotiation resource

8.4.10 External Reference Point $E_{bss\text{-}eni\text{-}cmd}$

The functionality of this External Reference Point is defined in clause 7.3.9. Eight resources are defined for this External Reference Point:

1) a container for recommendations and commands to be sent;
2) recommendations and commands;
3) a container for recommendations and commands that the OSS could not understand;
4) recommendations and command problems;
5) a container for recommendations and commands with solutions that the OSS accepted;
6) solved recommendations and commands;
7) a container for negotiation information;
8) negotiated information.

The functions are shown in Table 8-14, which provides brief descriptions of each function.

Table 8-14: ENI External Reference Point $E_{bss\text{-}eni\text{-}cmd}$ Functions from the ENI System to the BSS

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the BSS	/ebss_rec_cmd_out	CREATE	Creates a new RecommendCommandBSS resource and stores in it
		GET	Retrieves all recommendations and commands from the RecommendCommandBSS resource
Recommendations and Commands to be Sent to the BSS Operations	/ebss_rec_cmd_out/{recCmdID}	GET	Retrieves a single RecommendCommandBSS resource
		UPDATE	Modifies a single RecommendCommandBSS resource
		DELETE	Deletes a single RecommendCommandBSS resource
Recommendations and Commands to be Sent to the BSS having Problems	/ebss_rec_cmd_out/ {recCmdID}/problem	CREATE	Creates a new RecommendCommandOSSProblem resource
		GET	Retrieves all RecommendCommandOSSProblem resources
Recommendations and Commands to be Sent to the BSS having Problems Operations	/ebss_rec_cmd_out /{recCmdID}/problem /{problemID}	GET	Retrieves an individual RecommendCommandOSSProblem
		UPDATE	Modifies an individual RecommendCommandOSSProblem
		DELETE	Deletes an individual RecommendCommandOSSProblem
Recommendations and Commands to be Sent to the BSS with Solutions	/ebss_rec_cmd_out/ {recCmdID}/solution	CREATE	Creates a new RecommendCommandOSSSolution resource
		GET	Retrieves all RecommendCommandOSSSolution resources
Recommendations and Commands to be Sent to the BSS with Solutions Operations	/ebss_rec_cmd_out/ {recCmdID}/solution/ {solutionID}	GET	Retrieves an individual RecommendCommandOSSSolution resource
		UPDATE	Modifies an individual RecommendCommandOSSSolution resource
		DELETE	Deletes an individual RecommendCommandOSSSolution resource
Recommendations and Commands to be Sent to the BSS with Negotiation	/ebss_rec_cmd_out/ {recCmdID}/negotiation	CREATE	Creates a new RecommendCommandOSSNegotiate resource
		GET	Retrieves all RecommendCommandOSSNegotiate resources
Recommendations and Commands to be Sent to the BSS with Negotiation Operations	/ebss_rec_cmd_out/ {recCmdID}/negotiation {negotiationID}	GET	Retrieves an individual RecommendCommandOSSNegotiate resource
		UPDATE	Modifies an individual RecommendCommandOSSNegotiate resource
		DELETE	Deletes an individual RecommendCommandOSSNegotiate resource

8.4.11 External Reference Point $E_{bss\text{-}eni\text{-}pol}$

The functionality of this External Reference Point is defined in clause 7.3.10. Sixteen resources are defined for this External Reference Point:

1) a container for policies sent by the BSS;
2) BSS input policies;
3) a container for policies not understood by the ENI System;
4) ENI System policies having Problems;
5) a container for policies with solutions that the ENI System accepted;
6) ENI System solved policies;
7) a container for negotiation information for problem policies for the ENI System;

8) negotiated ENI System policy information;

9) a container for policies for sending to the BSS;

10) BSS output policies;

11) a container for policies not understood by the BSS;

12) BSS policies having Problems;

13) a container for policies with solutions that the BSS accepted;

14) solved BSS policies;

15) a container for negotiation information for problem BSS policies;

16) negotiated BSS policy information.

The functions are shown in Table 8-15 and Table 8-16 for policies sent by the BSS (and received by the ENI System) and policies sent by the ENI System (and received by the BSS), respectively, along with brief descriptions of each function.

Table 8-15: ENI External Reference Point $E_{bss\text{-}eni\text{-}pol}$ Functions from the BSS to the ENI System

Resource Name	Resource URI	RPC Method	Description
Policies Sent by BSS	/ebss_policy_in	CREATE	Creates a new BSSPolicyData resource and stores policies sent by the BSS in it
		GET	Retrieves all policies from the BSSPolicyData resource
Policies Sent by BSS Operations	/ebss_policy_in/{policyID}	GET	Retrieves a single policy received from the BSS
		UPDATE	Modifies a single policy received from the BSS
		DELETE	Deletes a single policy received from the BSS
Policies Sent by BSS having Problems	/ebss_policy_in/{policyID}/ problem	CREATE	Creates a new BSSPolicyDataProblem resource for policies containing problems
		GET	Retrieves all policies sent by the BSS that the ENI System has problems understanding
Policies Sent by BSS having Problems Operations	/ebss_policy_in/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the BSS that the ENI System cannot understand
		UPDATE	Modifies an existing policy sent by the BSS that the ENI System cannot understand
		DELETE	Deletes an existing policy sent by the BSS that the ENI System cannot understand
Policies Sent by BSS with Solutions	/ebss_policy_in/{policyID}/ solution	CREATE	Creates a new BSSPolicyDataSolution resource
		GET	Retrieves all policies sent by the BSS that the ENI System can now understand
Policies Sent by BSS with Solutions Operations	/ebss_policy_in/{policyID}/ solution/{solutionID}	GET	Retrieves an existing policy sent by the BSS that the ENI System can now understand
		UPDATE	Modifies an existing policy sent by the BSS that the ENI System can now understand
		DELETE	Deletes an existing policy sent by the BSS that the ENI System can now understand
Policies Sent by BSS with Negotiation	/ebss_policy_in/{policyID}/ negotiation	CREATE	Creates a new BSSPolicyNegotiate resource
		GET	Retrieves all negotiation information for all policies sent by the BSS to the ENI System to resolve problems
Policies Sent by BSS with Negotiation Operations	/ebss_policy_in/{policyID}/ negotiation/{negotiationID}	GET	Retrieves negotiation information for a selected policy sent by the BSS that the ENI System can now understand
		UPDATE	Modifies negotiation information for a selected policy sent by the BSS that the ENI System can now understand
		DELETE	Deletes negotiation information for a selected policy sent by the BSS that the ENI System can now understand

Table 8-16: ENI External Reference Point $E_{oss\text{-}eni\text{-}pol}$ for Functions Sent by the ENI System to the OSS

Resource Name	Resource URI	RPC Method	Description
Policies Sent by the ENI System	/ebss_policy_out	CREATE	Creates a new ENIBSSPolicyData resource and stores policies created by the ENI System in it
		GET	Retrieves all policies from the ENIBSSPolicyData resource
Policies Sent by the ENI System Operations	/ebss_policy_out/{policyID}	GET	Retrieves a single policy created by the ENI System
		UPDATE	Modifies a single policy created by the ENI System
		DELETE	Deletes a single policy created by the ENI System
Policies Sent by the ENI System having Problems	/ebss_policy_out/{policyID}/ problem	CREATE	Creates a new ENIBSSPolicyDataProblem resource
		GET	Retrieves all policies created by the ENI System that the BSS has problems understanding
Policies Sent by the ENI System having Problems Operations	/ebss_policy_out/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the ENI System that the BSS cannot understand
		UPDATE	Modifies an existing policy sent by the ENI System that the BSS cannot understand
		DELETE	Deletes an existing policy sent by the ENI System that the BSS cannot understand
Policies Sent by the ENI System with Solutions	/ebss_policy_out/{policyID}/ solution	CREATE	Creates a new ENIBSSPolicyDataSolution resource
		GET	Retrieves all policies sent by the ENI System that the BSS can now understand
Policies Sent by the ENI System with Solutions Operations	/ebss_policy_out/{policyID}/ solution/{solutionID}	GET	Retrieves an existing policy sent by the ENI System that the OSS can now understand
		UPDATE	Modifies an existing policy sent by the ENI System that the BSS can now understand
		DELETE	Deletes an existing policy sent by the ENI System that the BSS can now understand
Policies Sent by the ENI System with Negotiation	/ebss_policy_out/{policyID}/ negotiation	CREATE	Creates a new ENIBSSPolicyDataNegotiate resource
		GET	Retrieves all negotiation information for all policies created by the ENI System for the BSS to resolve problems
Policies Sent by the ENI System with Negotiation	/ebss_policy_out/{policyID}/ negotiation/{negotiationID}	GET	Retrieves an existing policy sent by the ENI System to the OSS to resolve problems that the BSS can now understand
		UPDATE	Modifies negotiation information for a selected policy sent by the ENI System that the BSS can now understand
		DELETE	Deletes negotiation information for a selected policy sent by the ENI System that the BSS can now understand

8.4.12 External Reference Point $E_{usr\text{-}eni\text{-}pol}$

The functionality of this External Reference Point is defined in clause 7.3.11. Sixteen resources are defined for this External Reference Point:

1) a container for policies sent by the user;

2) user input policies;

3) a container for policies not understood by the ENI System;

4) ENI System policies having Problems;

5) a container for policies with solutions that the ENI System accepted;

6) ENI System solved policies;

7) a container for negotiation information for problem policies for the ENI System;

8) negotiated ENI System policy information;

9) a container for policies to be sent to the user;

10) policies for the user;

11) a container for policies not understood by the user;

12) user policies having Problems;

13) a container for policies with solutions that the user accepted;

14) solved user policies;

15) a container for negotiation information for problem user policies;

16) negotiated user policy information.

The functions are shown in Table 8-17 and Table 8-18 for policies sent by the user (and received by the ENI System) and policies sent by the ENI System (and received by the user), respectively, along with brief descriptions of each function.

Table 8-17: ENI External Reference Point $E_{bss\text{-}eni\text{-}pol}$ Functions from the User to the ENI System

Resource Name	Resource URI	RPC Method	Description
Policies Sent by User	/eusr_policy_in	CREATE	Creates a new UserPolicyData resource and stores policies sent by the user in it
		GET	Retrieves all UserPolicyData resources
Policies Sent by User Operations	/eusr_policy_in/{policyID}	GET	Retrieves a single policy received from the user
		UPDATE	Modifies a single policy received from the user
		DELETE	Deletes a single policy received from the user
Policies Sent by User having Problems	/eusr_policy_in/{policyID}/ problem	CREATE	Creates a new UserPolicyDataProblem resource for policies containing problems
		GET	Retrieves all policies sent by the user that the ENI System has problems understanding
Policies Sent by User having Problems Operations	/eusr_policy_in/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the user that the ENI System cannot understand
		UPDATE	Modifies an existing policy sent by the user that the ENI System cannot understand
		DELETE	Deletes an existing policy sent by the user that the ENI System cannot understand
Policies Sent by User with Solutions	/eusr_policy_in/{policyID}/ solution	CREATE	Creates a new UserPolicyDataSolution resource
		GET	Retrieves all policies sent by the user that the ENI System can now understand
Policies Sent by User with Solutions Operations	/eusr_policy_in/{policyID}/ solution/{solutionID}	GET	Retrieves an existing policy sent by the BSS that the ENI System can now understand
		UPDATE	Modifies an existing policy sent by the BSS that the ENI System can now understand
		DELETE	Deletes an existing policy sent by the BSS that the ENI System can now understand
Policies Sent by User with Negotiation	/eusr_policy_in/{policyID}/ negotiation	CREATE	Creates a new UserPolicyDataNegotaiate resource
		GET	Retrieves all negotiation information for all policies sent by the user to the ENI System to resolve problems
Policies Sent by User with Negotiation Operations	/eusr_policy_in/{policyID}/ negotiation/{negotiationID}	GET	Retrieves negotiation information for a selected policy sent by the user that the ENI System can now understand
		UPDATE	Modifies negotiation information for a selected policy sent by the user that the ENI System can now understand
		DELETE	Deletes negotiation information for a selected policy sent by the user that the ENI System can now understand

Table 8-18: ENI External Reference Point $E_{oss\text{-}eni\text{-}pol}$ for Functions Sent by the ENI System to the User

Resource Name	Resource URI	RPC Method	Description
Policies Sent by the ENI System	/eusr_policy_out	CREATE	Creates a new ENIBSSPolicyData resource and stores policies created by the ENI System in it
		GET	Retrieves all policies from the ENIBSSPolicyData resource
Policies Sent by the ENI System Operations	/eusr_policy_out/{policyID}	GET	Retrieves a single policy created by the ENI System
		UPDATE	Modifies a single policy created by the ENI System
		DELETE	Deletes a single policy created by the ENI System
Policies Sent by the ENI System having Problems	/eusr_policy_out/{policyID}/ problem	CREATE	Creates a new ENIBSSPolicyDataProblem resource
		GET	Retrieves all policies created by the ENI System that the BSS has problems understanding
Policies Sent by the ENI System having Problems Operations	/eusr_policy_out/{policyID}/ problem/{problemID}	GET	Retrieves an existing policy sent by the ENI System that the BSS cannot understand
		UPDATE	Modifies an existing policy sent by the ENI System that the BSS cannot understand
		DELETE	Deletes an existing policy sent by the ENI System that the BSS cannot understand
Policies Sent by the ENI System with Solutions	/eusr_policy_out/{policyID}/ solution	CREATE	Creates a new ENIBSSPolicyDataSolution resource
		GET	Retrieves all policies sent by the ENI System that the BSS can now understand
Policies Sent by the ENI System with Solutions Operations	/eusr_policy_out/{policyID}/ solution/{solutionID}	GET	Retrieves an existing policy sent by the ENI System that the OSS can now understand
		UPDATE	Modifies an existing policy sent by the ENI System that the BSS can now understand
		DELETE	Deletes an existing policy sent by the ENI System that the BSS can now understand
Policies Sent by the ENI System with Negotiation	/eusr_policy_out/{policyID}/ negotiation	CREATE	Creates a new ENIBSSPolicyDataNegotiate resource
		GET	Retrieves all negotiation information for all policies created by the ENI System for the BSS to resolve problems
Policies Sent by the ENI System with Negotiation	/eusr_policy_out/{policyID}/ negotiation/{negotiationID}	GET	Retrieves an existing policy sent by the ENI System to the OSS to resolve problems that the BSS can now understand
		UPDATE	Modifies negotiation information for a selected policy sent by the ENI System that the BSS can now understand
		DELETE	Deletes negotiation information for a selected policy sent by the ENI System that the BSS can now understand

8.4.13 External Reference Point $E_{or\text{-}eni\text{-}dat}$

The functionality of this External Reference Point is defined in clause 7.3.12. Ten resources are defined for this External Reference Point:

1) container for ingested Orchestrator information;

2) ingested Orchestrator information;

3) container for processing and normalizing Orchestrator information;

4) processed and normalized Orchestrator information;

5) container for processing and normalizing Orchestrator information that the ENI System cannot understand;

6) processed and normalized Orchestrator information;

7) container for processing and normalizing Orchestrator information that the ENI System can now understand;

8) processed and normalized Orchestrator information;

9) container for negotiation information for resolving problems understanding Orchestrator information;

10) negotiated Orchestrator information.

The functions are shown in Table 8-19, which provides brief descriptions of each function.

Table 8-19: ENI External Reference Point $E_{or\text{-}eni\text{-}dat}$ Functions from the Orchestrator to the ENI System

Resource Name	Resource URI	RPC Method	Description
Orchestrator Input Data	/eor_data_in	CREATE	Creates a new IngestedOrchData resource and stores ingested data in it
		GET	Retrieves all IngestedOrchData resources
Orchestrator Input Data Operations	/eor_data_in/{ingestDataID}	GET	Retrieves a single IngestedOrchData resource
		UPDATE	Modifies a single IngestedOrchData resource
		DELETE	Deletes a single IngestedOrchData resource
Orchestrator Input Data Normalized	/eor_normalized_data	CREATE	Creates a new IngestedOrchDataNormalized resource
		GET	Retrieves all IngestedOrchDataNormalized resources
Orchestrator Input Data Normalized Operations	/eor_normalized_data/ {normalizedDataID}	GET	Retrieves an existing IngestedOrchDataNormalized resource
		UPDATE	Modifies an existing IngestedOrchDataNormalized resource
		DELETE	Deletes an existing IngestedOrchDataNormalized resource
Orchestrator Input Data Normalized having Problems	/eor_normalized_data/ {normalizedDataID}/problem	CREATE	Creates a new IngestedOrchDataNormalizedProblem resource
		GET	Retrieves all IngestedOrchDataNormalizedProblem resources
Orchestrator Input Data Normalized having Problems Operations	/eor_normalized_data/ {normalizedDataID}/problem /{problemID}	GET	Retrieves an individual IngestedOrchDataNormalizedProblem resource
		UPDATE	Modifies an individual IngestedOrchDataNormalizedProblem resource
		DELETE	Deletes an individual IngestedOrchDataNormalizedProblem resource
Orchestrator Input Data Normalized with Solutions	/eor_normalized_data/ {normalizedDataID}/solution	CREATE	Creates a new IngestedOrchDataNormalizedSolution resource
		GET	Retrieves all IngestedOrchDataNormalizedSolution resources
Orchestrator Input Data Normalized with Solutions Operations	/eor_normalized_data/ {normalizedDataID}/solution / {solutionID}	GET	Retrieves an individual IngestedOrchDataNormalizedSolution resource
		UPDATE	Modifies an individual IngestedOrchDataNormalizedSolution resource
		DELETE	Deletes an individual IngestedOrchDataNormalizedSolution resource
Orchestrator Input Data Normalized with Negotiation	/eor_normalized_data/ {normalizedDataID}/negotiation	CREATE	Creates a new IngestedOrchDataNormalizedNegotiate resource
		GET	Retrieves all IngestedOrchDataNormalizedNegotiate resources
Orchestrator Input Data Normalized with Negotiation Operations	/eor_normalized_data/ {normalizedDataID}/solution / {solutionID}	GET	Retrieves an individual IngestedOrchDataNormalizedNegotiate resource
		UPDATE	Modifies an individual IngestedOrchDataNormalizedNegotiate resource
		DELETE	Deletes an individual IngestedOrchDataNormalizedNegotiate resource

8.4.14 External Reference Point $E_{or\text{-}eni\text{-}cmd}$

The functionality of this External Reference Point is defined in clause 7.3.13. Eight resources are defined for this External Reference Point

1) a container for recommendations and commands to be sent to the Orchestrator;
2) recommendations and commands;
3) a container for recommendations and commands that the Orchestrator could not understand;
4) recommendations and command problems;
5) a container for recommendations and commands with solutions that the Orchestrator accepted;
6) solved recommendations and commands;
7) a container for negotiation information;
8) negotiated information.

The functions are shown in Table 8-20, which provides brief descriptions of each function.

Table 8-20: ENI External Reference Point $E_{or\text{-}eni\text{-}cmd}$ Functions from the ENI System to the Orchestrator

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the Orchestrator	/eor_rec_cmd_out	CREATE	Creates a new RecommendCommandOrch resource and stores in it
		GET	Retrieves all RecommendCommandOrch resources
Recommendations and Commands to be Sent to the Orchestrator Operations	/eor_rec_cmd_out/{recCmdID}	GET	Retrieves a single RecommendCommandOrch resource
		UPDATE	Modifies a single RecommendCommandOrch resource
		DELETE	Deletes a single RecommendCommandOrch resource
Recommendations and Commands to be Sent to the Orchestrator having Problems	/eor_rec_cmd_out/ {recCmdID}/problem	CREATE	Creates a new RecommendCommandOrchProblem resource
		GET	Retrieves all RecommendCommandOrchProblem resources
Recommendations and Commands to be Sent to the Orchestrator having Problems Operations	/eor_rec_cmd_out /{recCmdID}/problem /{problemID}	GET	Retrieves an individual RecommendCommandOrchProblem resource
		UPDATE	Modifies an individual RecommendCommandOrchProblem resource
		DELETE	Deletes an individual RecommendCommandOrchProblem resource
Recommendations and Commands to be Sent to the Orchestrator with Solutions	/eor_rec_cmd_out/ {recCmdID}/solution	CREATE	Creates a new RecommendCommandOrchSolution resource
		GET	Retrieves all RecommendCommandOrchSolution resources
Recommendations and Commands to be Sent to the Orchestrator with Solutions Operations	/eor_rec_cmd_out/ {recCmdID}/solution/ {solutionID}	GET	Retrieves an individual RecommendCommandOrchSolution resource
		UPDATE	Modifies an individual RecommendCommandOrchSolution resource
		DELETE	Deletes an individual RecommendCommandOrchSolution resource
Recommendations and Commands to be Sent to the Orchestrator with Negotiation	/eor_rec_cmd_out/ {recCmdID}/negotiation	CREATE	Creates a new RecommendCommandOrchNegotiate resource
		GET	Retrieves all RecommendCommandOSSNegotiate resources

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the Orchestrator with Negotiation Operations	/eor_rec_cmd_out/ {recCmdID}/negotiation {negotiationID}	GET	Retrieves an individual RecommendCommandOSSNegotiate resource
		UPDATE	Modifies an individual RecommendCommandOSSNegotiate resource
		DELETE	Deletes an individual RecommendCommandOSSNegotiate resource

8.4.15 External Reference Point $E_{or\text{-}eni\text{-}pol}$

The functionality of this External Reference Point is defined in clause 7.3.14. Sixteen resources are defined for this External Reference Point:

1) a container for policies sent by the Orchestrator;
2) Orchestrator input policies;
3) a container for policies not understood by the ENI System;
4) ENI System policies having Problems;
5) a container for policies with solutions that the ENI System accepted;
6) ENI System solved policies;
7) a container for negotiation information for problem policies for the ENI System;
8) negotiated ENI System policy information;
9) a container for policies to be sent to the Orchestrator;
10) policies for the Orchestrator;
11) a container for policies not understood by the Orchestrator;
12) Orchestrator policies having Problems;
13) a container for policies with solutions that the Orchestrator accepted;
14) solved Orchestrator policies;
15) a container for negotiation information for problem Orchestrator policies;
16) negotiated Orchestrator policy information.

The functions are shown in Table 8-21 and Table 8-22 for policies sent by the user (and received by the ENI System) and policies sent by the ENI System (and received by the user), respectively, along with brief descriptions of each function.

Table 8-21: ENI External Reference Point $E_{usr\text{-}eni\text{-}pol}$ Functions Sent by the Orchestrator to ENI System

Resource Name	Resource URI	RPC Method	Description
Orchestrator Policy Input Data	/eor_policy_in	CREATE	Creates a new OrchPolicyData resource and stores ingested data in it
		GET	Retrieves all OrchPolicyData resources
Orchestrator Policy Input Data Operations	/eor_policy_in/{policyID}	GET	Retrieves a single OrchPolicyData resource
		UPDATE	Modifies a single OrchPolicyData resource
		DELETE	Deletes a single OrchPolicyData resource
Orchestrator Policy Input Data Normalized	/eor_policy_in/{policyID}/ normalize	CREATE	Creates a new OrchPolicyDataNormalized resource
		GET	Retrieves all OrchPolicyDataNormalized resources
Orchestrator Policy Input Data Normalized Operations	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}	GET	Retrieves an existing OrchPolicyDataNormalized resource
		UPDATE	Modifies an existing OrchPolicyDataNormalized resource
		DELETE	Deletes an existing OrchPolicyDataNormalized resource
Orchestrator Policy Input Data Normalized having Problems	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ problem	CREATE	Creates a new OrchPolicyDataNormalizedProblem resource
		GET	Retrieves all OrchPolicyDataNormalizedProblem resources
Orchestrator Policy Input Data Normalized having Problems Operations	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ problem/{problemID}	GET	Retrieves an individual OrchPolicyDataNormalizedProblem resource
		UPDATE	Modifies an individual OrchPolicyDataNormalizedProblem resource
		DELETE	Deletes an individual OrchPolicyDataNormalizedProblem resource
Orchestrator Policy Input Data Normalized with Solutions	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ solution	CREATE	Creates a new OrchPolicyDataNormalizedSolution resource
		GET	Retrieves all OrchPolicyDataNormalizedSolution resources
Orchestrator Policy Input Data Normalized with Solutions Operations	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ solution/{solutionID}	GET	Retrieves an individual OrchPolicyDataNormalizedSolution resource
		UPDATE	Modifies an individual OrchPolicyDataNormalizedSolution resource
		DELETE	Deletes an individual OrchPolicyDataNormalizedSolution resource
Orchestrator Policy Input Data Normalized with Negotiation	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ negotiation	CREATE	Creates a new OrchPolicyDataNormalizedNegotiate resource
		GET	Retrieves all OrchPolicyDataNormalizedNegotiate resources
Orchestrator Policy Input Data Normalized with Negotiation Operations	/eor_policy_in/{policyID}/ normalize/{normalizedDataID}/ negotiation/{negotiationID}	GET	Retrieves an individual OrchPolicyDataNormalizedNegotiate resource
		UPDATE	Modifies an individual OrchPolicyDataNormalizedNegotiate resource
		DELETE	Deletes an individual OrchPolicyDataNormalizedNegotiate resource

Table 8-22: ENI External Reference Point $E_{usr\text{-}eni\text{-}pol}$ Functions Sent by the ENI System to the User

Resource Name	Resource URI	RPC Method	Description
Policies Sent by the ENI System	/eor_policy_out	CREATE	Creates a new ENIOrchPolicyData resource and stores ingested data in it
		GET	Retrieves all ENIOrchPolicyData resources
Policies Sent by the ENI System Operations	/eor_policy_out/{policyID}	GET	Retrieves a single ENIOrchPolicyData resource
		UPDATE	Modifies a single ENIOrchPolicyData resource
		DELETE	Deletes a single ENIOrchPolicyData resource
Policies Sent by the ENI System having Problems	/eor_policy_out/{policyID}/ normalize	CREATE	Creates a new ENIOrchPolicyDataNormalized resource
		GET	Retrieves all ENIOrchPolicyDataNormalized resources
Policies Sent by the ENI System having Problems Operations	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}	GET	Retrieves an existing ENIOrchPolicyDataNormalized resource
		UPDATE	Modifies an existing ENIOrchPolicyDataNormalized resource
		DELETE	Deletes an existing ENIOrchPolicyDataNormalized resource
Policies Sent by the ENI System with Solutions	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ problem	CREATE	Creates a new ENIOrchPolicyDataNormalizedProblem resource
		GET	Retrieves all ENIOrchPolicyDataNormalizedProblem resources
Policies Sent by the ENI System with Solutions Operations	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ problem/{problemID}	GET	Retrieves an individual ENIOrchPolicyDataNormalizedProblem resource
		UPDATE	Modifies an individual ENIOrchPolicyDataNormalizedProblem resource
		DELETE	Deletes an individual ENIOrchPolicyDataNormalizedProblem resource
Policies Sent by the ENI System with Negotiation	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ solution	CREATE	Creates a new ENIOrchPolicyDataNormalizedSolution resource
		GET	Retrieves all ENIOrchPolicyDataNormalizedSolution resources
Policies Sent by the ENI System with Negotiation	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ solution/{solutionID}	GET	Retrieves an individual ENIOrchPolicyDataNormalizedSolution resource
		UPDATE	Modifies an individual ENIOrchPolicyDataNormalizedSolution resource
		DELETE	Deletes an individual ENIOrchPolicyDataNormalizedSolution resource
Policies Sent by the ENI System	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ negotiation	CREATE	Creates a new ENIOrchPolicyDataNormalizedNegotiate resource
		GET	Retrieves all ENIOrchPolicyDataNormalizedNegotiate resources
Policies Sent by the ENI System Operations	/eor_policy_out/{policyID}/ normalize/{normalizedDataID}/ negotiation/{negotiationID}	GET	Retrieves an individual ENIOrchPolicyDataNormalizedNegotiate resource
		UPDATE	Modifies an individual ENIOrchPolicyDataNormalizedNegotiate resource
		DELETE	Deletes an individual ENIOrchPolicyDataNormalizedNegotiate resource

8.4.16 External Reference Point $E_{inf\text{-}eni\text{-}dat}$

The functionality of this External Reference Point is defined in clause 7.3.15. Ten resources are defined for this External Reference Point:

1) container for ingested Infrastructure information;

2) ingested Infrastructure information;

3) container for processing and normalizing Infrastructure information;

4) processed and normalized Infrastructure information;

5) container for processing and normalizing Infrastructure information that the ENI System cannot understand;

6) processed and normalized Infrastructure information;

7) container for processing and normalizing Infrastructure information that the ENI System can now understand;

8) processed and normalized Infrastructure information;

9) container for negotiation information for resolving problems understanding Infrastructure information;

10) negotiated Infrastructure information.

The functions are shown in Table 8-23, which provides brief descriptions of each function.

Table 8-23: ENI External Reference Point $E_{inf\text{-}eni\text{-}dat}$ Functions from the Infrastructure to ENI System

Resource Name	Resource URI	RPC Method	Description
Infrastructure Input Data	/einf_data_in	CREATE	Creates a new IngestedInfraData resource and stores ingested data in it
		GET	Retrieves all IngestedInfraData resources
Infrastructure Input Data Operations	/einf_data_in/{ingestDataID}	GET	Retrieves a single IngestedInfraData resource
		UPDATE	Modifies a single IngestedInfraData resource
		DELETE	Deletes a single IngestedInfraData resource
Infrastructure Normalized Data	/einf_normalized_data	CREATE	Creates a new IngestedInfraDataNormalized resource
		GET	Retrieves all IngestedInfraDataNormalized resources
Infrastructure Normalized Data Operations	/einf_normalized_data/ {normalizedDataID}	GET	Retrieves an existing IngestedInfraDataNormalized resource
		UPDATE	Modifies an existing IngestedInfraDataNormalized resource
		DELETE	Deletes an existing IngestedInfraDataNormalized resource
Infrastructure Normalized Data having Problems	/einf_normalized_data/ {normalizedDataID}/problem	CREATE	Creates a new IngestedInfraDataNormalizedProblem resource
		GET	Retrieves all IngestedInfraDataNormalizedProblem resources
Infrastructure Normalized Data having Problems Operations	/einf_normalized_data/ {normalizedDataID}/problem /{problemID}	GET	Retrieves an individual IngestedInfraDataNormalizedProblem resource
		UPDATE	Modifies an individual IngestedInfraDataNormalizedProblem resource
		DELETE	Deletes an individual IngestedInfraDataNormalizedProblem resource
Infrastructure Normalized Data with Solutions	/einf_normalized_data/ {normalizedDataID}/solution	CREATE	Creates a new IngestedInfraDataNormalizedSolution resource
		GET	Retrieves all IngestedInfraDataNormalizedSolution resources
Infrastructure Normalized Data with Solutions Operations	/einf_normalized_data/ {normalizedDataID}/solution / {solutionID}	GET	Retrieves an individual IngestedInfraDataNormalizedSolution resource
		UPDATE	Modifies an individual IngestedInfraDataNormalizedSolution resource
		DELETE	Deletes an individual IngestedInfraDataNormalizedSolution resource
Infrastructure Normalized Data with Negotiation	/einf_normalized_data/ {normalizedDataID}/negotiation	CREATE	Creates a new IngestedInfraDataNormalizedNegotiate resource
		GET	Retrieves all IngestedInfraDataNormalizedNegotiate resources
Infrastructure Normalized Data with Negotiation Operations	/einf_normalized_data/ {normalizedDataID}/solution / {solutionID}	GET	Retrieves an individual IngestedInfraDataNormalizedNegotiate resource
		UPDATE	Modifies an individual IngestedInfraDataNormalizedNegotiate resource
		DELETE	Deletes an individual IngestedInfraDataNormalizedNegotiate resource

8.4.17 External Reference Point $E_{inf\text{-}eni\text{-}cmd}$

The functionality of this External Reference Point is defined in clause 7.3.16. Eight resources are defined for this External Reference Point:

1) a container for recommendations and commands to be sent to the Infrastructure;
2) recommendations and commands;
3) a container for recommendations and commands that the Infrastructure could not understand;
4) recommendations and command problems;
5) a container for recommendations and commands with solutions that the Infrastructure accepted;
6) solved recommendations and commands;
7) a container for negotiation information;
8) negotiated information.

The functions are shown in Table 8-24, which provides brief descriptions of each function.

Table 8-24: ENI External Reference Point $E_{inf\text{-}eni\text{-}cmd}$ Functions from the ENI System to Infrastructure

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the Infrastructure	/einf_rec_cmd_out	CREATE	Creates a new RecommendCommandInfra resource and stores in it
		GET	Retrieves all RecommendCommandInfra resources
Recommendations and Commands to be Sent to the Infrastructure Operations	/einf_rec_cmd_out/{recCmdID}	GET	Retrieves a single RecommendCommandInfra resource
		UPDATE	Modifies a single RecommendCommandInfra resource
		DELETE	Deletes a single RecommendCommandInfra resource
Recommendations and Commands to be Sent to the Infrastructure having Problems	/einf_rec_cmd_out/ {recCmdID}/problem	CREATE	Creates a new RecommendCommadInfraProblem resource
		GET	Retrieves all RecommendCommadInfraProblem resources
Recommendations and Commands to be Sent to the Infrastructure having Problems Operations	/einf_rec_cmd_out /{recCmdID}/problem /{problemID}	GET	Retrieves an individual RecommendCommadInfraProblem resource
		UPDATE	Modifies an individual RecommendCommadInfraProblem resource
		DELETE	Deletes an individual RecommendCommadInfraProblem resource
Recommendations and Commands to be Sent to the Infrastructure with Solutions	/einf_rec_cmd_out/ {recCmdID}/solution	CREATE	Creates a new RecommendCommadInfraSolution resource
		GET	Retrieves all RecommendCommadInfraSolution resources
Recommendations and Commands to be Sent to the Infrastructure with Solutions Operations	/einf_rec_cmd_out/ {recCmdID}/solution/ {solutionID}	GET	Retrieves an individual RecommendCommadInfraSolution resource
		UPDATE	Modifies an individual RecommendCommadInfraSolution resource
		DELETE	Deletes an individual RecommendCommadInfraSolution resource
Recommendations and Commands to be Sent to the Infrastructure with Negotiation	/einf_rec_cmd_out/ {recCmdID}/negotiation	CREATE	Creates a new RecommendCommadInfraNegotiate resource
		GET	Retrieves all RecommendCommadInfraNegotiate resources

Resource Name	Resource URI	RPC Method	Description
Recommendations and Commands to be Sent to the Infrastructure with Negotiation Operations	/einf_rec_cmd_out/ {recCmdID}/negotiation {negotiationID}	GET	Retrieves an individual RecommendCommadInfraNegotiate resource
		UPDATE	Modifies an individual RecommendCommadInfraNegotiate resource
		DELETE	Deletes an individual RecommendCommadInfraNegotiate resource

9 Interacting with Other Standardized Architectures

9.1 Introduction

The ENI System uses policy management, knowledge engineering, and cognition to enhance the operation and performance of the monitoring, configuration, management, and orchestration processes of the Assisted System. The ENI System will also offer the automation of these processes. The ENI System does not require any changes to Reference Points or their interfaces with existing Systems, including NFV MANO, SDN Controllers, or the MEF-LSO architecture. Rather, ENI uses those definitions and provides adaptation and mediation with external standardized architectures. However, changes made to existing Assisted Systems to comply with ENI External Reference Points and ENI APIs will benefit ENI as well as increase the semantics and understanding of interaction between the ENI System and these existing Assisted Systems.

ENI interacts with architectures from other SDOs using a subset of External Reference Points defined in clause 7.

An Assisted System may contain zero or more SDO Systems. All previous information defined in clauses 3 to 7 still apply to the Assisted System regardless of how many SDO Systems it contains.

Every clause that follows represents initial thinking of the ENI ISG. Each interaction with each external SDO is pending future collaboration and liaison work with that SDO.

Clause 8.2 describes a generic architecture that enables ENI to interact with one or more SDO Systems. Clause 8.4 provides preliminary thoughts about how the generic architecture may be applied to NFV MANO.

NOTE: Further work on interaction with NFV MANO, as well as with other SDO Systems, are for further study in Release 3 of the present document (see clause 9).

9.2 Generic Architecture

Figure 9-1 is a high-level diagram that illustrates how the ENI System may interact with an Assisted System that includes one or more SDO Systems (e.g. NFV MANO, MEF-LSO, BBF-CloudCO, as well as others that may become available in the market that are of interest to the ENI ISG) or their Designated Entities. In particular, it shows how ENI may interact at different levels (e.g. physical, resource, service, customer) of the combination of the SDO System(s) and the Assisted System or its Designated Entity. There may be zero or more instances of an SDO System in an Assisted System.

The SDO System layer(s) are arbitrarily shown between the Service and Resource layers. This need not be construed as a requirement that any particular SDO Component logically exists at that layer. In addition, an SDO System may affect one or more of the Physical, Resource, Service, or Customer Layers. An SDO System may introduce a new logical layer as well.

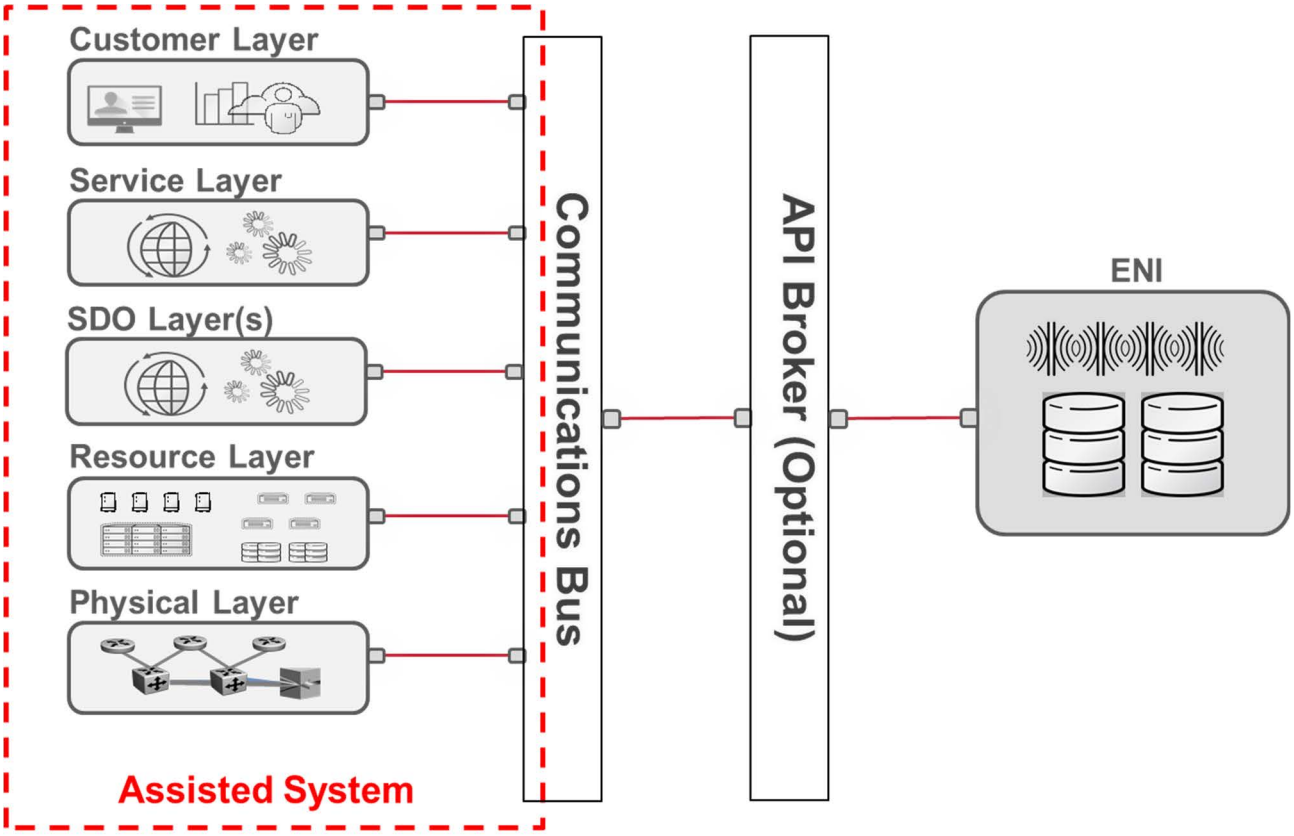

Figure 9-1: Overall view of System Interaction of ENI

Definitions of the above layers are summarized as follows:

- **Customer Layer:** Customer portal, and status notifications of their services and system health. This interface is needed as ENI is potentially changing this layer's behaviour.
- **Service Layer:** Logical constructs that combine multiple elements to deliver a single service.

EXAMPLE: A VPN service is a combination of components, including a firewall service, authentication service and routing services, blended into a single purpose construct.

- **SDO Layer:** Functionality provided by a System defined by an SDO. This appears as a set of Functional Blocks to ENI.
- **Resource Layer:** Data from all physical and virtualisation telemetry (e.g. CPU, storage, and network).
- **Physical Layer:** All hardware resources supporting the infrastructure. Information on system health, usage and resource utilization will contribute to understanding the current state of hardware, and aid in predicting its future state. For example, this will aid in the optimization of the physical layer when using machine learning mechanisms.

This release of the ENI System architecture is limited to proposing how telemetry data from an Assisted System that contains one or more SDO Systems may interact.

NOTE: This topic will be further specified in Release 4 of the present document (see clause 9).

9.3 Generic SDO Interaction Architecture

9.3.1 Introduction

An API Broker may be used to facilitate the interaction between Assisted Systems and the ENI System. This interaction may include one of two options:

1) direct communication between the ENI System and the Assisted System (or its Designated Entity); and/or

2) direct communication between the ENI System and the SDO System.

The difference is that in the first option, the Assisted System (or its Designated Entity) is responsible for communicating between ENI and the SDO System, whereas in the second option, the ENI System communicates directly with the SDO System. In the second option, the SDO System may inform the Assisted System (of which it is a part) of its interaction with the ENI System. In either of these options, the interaction shall not extend, or require the extension of, the functionality of the SDO system, or any of its components, beyond what the SDO System has currently defined in its specifications.

The interaction between the ENI and SDO Systems will use a set of common scenarios to investigate what types of interactions are possible. These scenarios are not unique to any one particular SDO, and some scenarios may not be applicable to all SDOs. They are provided to define a common framework to encourage conversation on interaction between ENI and each SDO System that wants to interoperate with ENI.

A common theme in each scenario is the functionality provided by the SDO System compared to the Assisted System that contains the SDO System. More specifically, the following five exemplary scenarios examine:

- Passive notification to the SDO System, no interaction with the rest of the Assisted System.
- Data analysis between the ENI and SDO Systems, no interaction with the rest of the Assisted System.
- Active assistance to the SDO System, no interaction with the rest of the Assisted System.
- Active assistance to the entire Assisted System for only those functions that are contained in both the SDO System and the Assisted System.
- Active assistance to the entire Assisted System, regardless of whether a function is only in the Assisted System or the SDO System.

The above five scenarios are defined in order of increasing complexity.

NOTE: These are for further study in Release 3 of the present document (see clause 9).

9.4 Interaction with NFV MANO

9.4.1 High Level description of the NFV MANO - ENI Interaction

NOTE 1: The contents of this clause requires a liaison and agreement with the NFV MANO ISG, and is for further study in Release 3 of the present document (see clause 9).

The interaction of ENI with an NFV MANO-based architecture is shown in the following diagram. Note that the functionality provided by NFV MANO is only a part of what service providers work with in orchestration. For example, NFV MANO does not directly monitor or manage some of the components in the Physical Layer, as well as the most functionality of the Service and Customer Layers. All of these components need to be coordinated. This coordination may be done by the ENI System.

The High Level NFV MANO - ENI Interaction Architecture Framework is shown in Figure 9-2, where the ENI System is external to the NFV MANO framework.

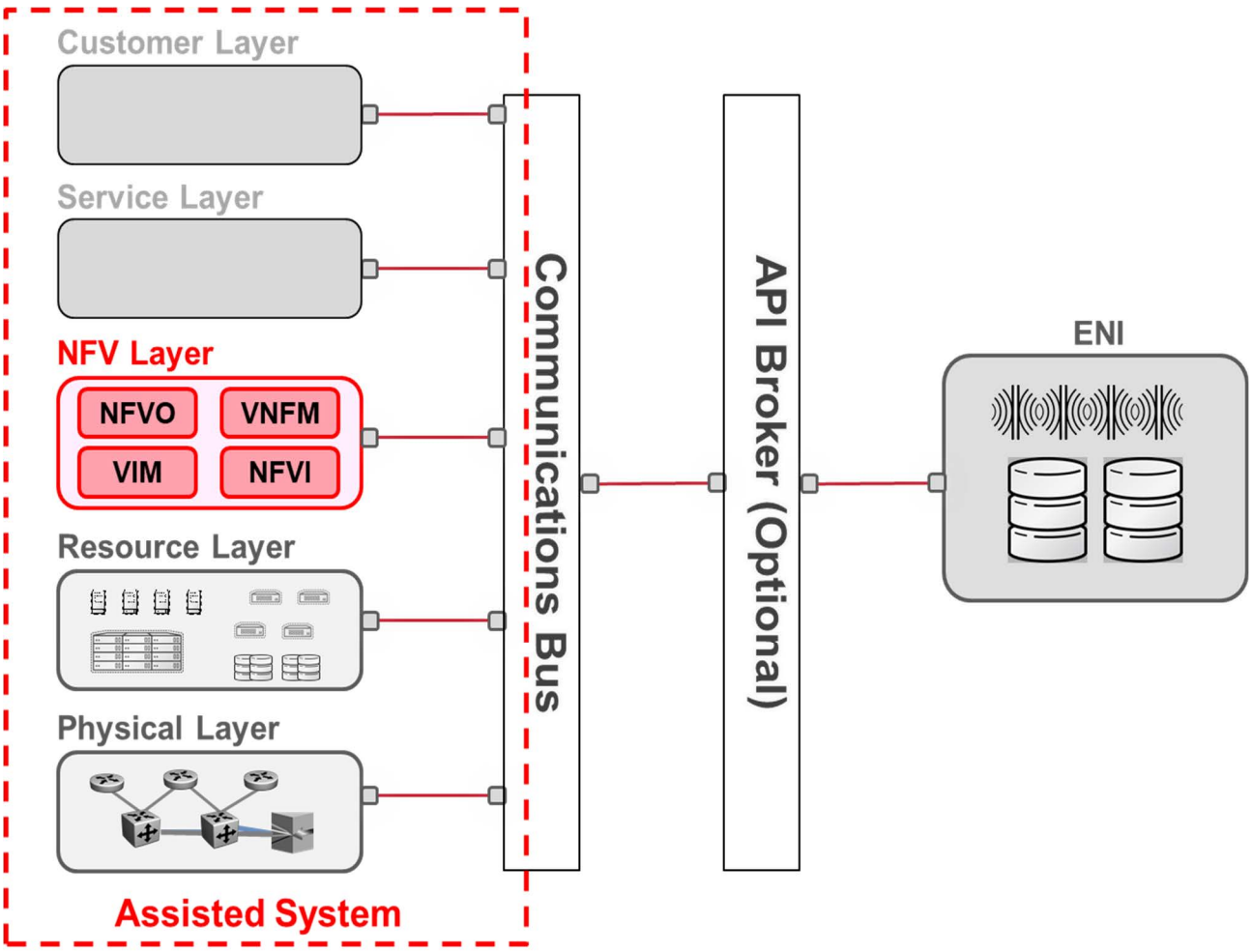

Figure 9-2: Interaction between the NFV MANO and ENI

The ENI System shall connect to the NFV MANO framework through specific External Reference Points in the following four functional areas:

- NFVO (NFV Orchestrator).
- VNFM (VNF Manager).
- VIM (Virtualised Infrastructure Manager).
- NFVI (NFV Infrastructure).

This will enable the collection of appropriate telemetry for ENI to analyse. However, it is unclear, at this point in time, how ENI can provide recommendations and/or commands to the NFV MANO Assisted System (or its Designated Entity), as the existing NFV MANO Reference Points are not suitable for conveying such information.

NOTE 2: Appropriate Reference Points are not currently defined by NFV MANO. Hence, this will be work for Release 3 of the present document (see clause 9), after a liaison has been created between ISG ENI and ISG NFV.

The purpose of the NFV MANO System is to provide orchestration, management, and control functions. In the envisaged interaction, the ENI System may support and help to improve these NFV MANO functionalities. ENI may also enhance the effectiveness of NFV MANO by providing AI-based functionalities as well as context- and situation-awareness. There are several options for NFV MANO and ENI to interact. One example is using the API Broker to translate between ENI and NFV MANO APIs. However, equivalent External Reference Points will first need to be defined by the NFV MANO System for at least the NFV Orchestrator, VNF Manager, and VIM Functional Blocks. The desire to implement this interaction could also imply the addition and/or modification of External ENI Reference Points to facilitate this interaction. The advantage of this approach is that the API Broker decouples the ENI and NFV MANO Systems, so that either can change without directly affecting the other.

The NFV MANO System has specific management responsibilities that are different in scope than those of the ENI System. The interaction between the NFV MANO and the ENI Systems may take on a number of permutations; exemplary scenarios are described in the following clauses. The behaviour of both Systems to the messages that are exchanged in each scenario being proposed shall be defined in clauses 4.4.2.1 and 4.4.3 of the present document.

9.4.2 Initial proposals for interaction scenarios

9.4.2.1 Introduction

The following clauses describe four scenarios of interaction between the ENI and Assisted Systems, where the latter has an NFV MANO System embedded in it. The first three scenarios limit the ENI System to interacting with the NFV MANO part of the Assisted System only, while the last scenario removes these restrictions and allows the ENI System to interact with the entire NFV MANO System.

9.4.2.2 Scenario 1: Passive Notification to NFV MANO

This scenario assumes that interaction between ENI and the Assisted System shall be limited to ENI and NFV MANO.

The ENI System observes the interaction between the Assisted System and the NFV MANO System. The ENI System may then place information describing the results of analysis performed by ENI in a repository shared by the NFV MANO and ENI Systems for review and future decision support. In this option, the ENI System shall not send recommendations or commands to either the Assisted System or the NFV MANO System. The ENI System can only send notifications to the NFV MANO System when it adds or changes data or information in the shared repository.

9.4.2.3 Scenario 2: Active Data Analysis for NFV MANO

This scenario assumes that interaction between ENI and the Assisted System shall be limited to ENI and NFV MANO.

The ENI System may take on the role of augmented data analysis, and enhance the MANO subsystem via delivery of decision support to the NFV MANO orchestrator in real time. In this option, the ENI System only sends recommendations or commands to the NFV MANO System that affect data analysis and related functions, such as trend prediction. It may also send repository change notifications to the NFV MANO System.

9.4.2.4 Scenario 3: Active Assistance to the NFV MANO System

This scenario assumes that interaction between ENI and the Assisted System shall be limited to ENI and NFV MANO.

The ENI System may take on an expanded role, assisting the NFV MANO System where appropriate. In this option, the ENI System sends recommendations and commands to the NFV MANO System for all functions that both the ENI and NFV MANO Systems have. However, the ENI System will not send recommendations and commands to the NFV MANO System or the Assisted System that affect functionality that NFV MANO does not currently have in its published specifications. It may also send repository change notifications to the NFV MANO System.

9.4.2.5 Scenario 4: Active Assistance to the Assisted System

This scenario assumes that interaction between ENI and the Assisted System, including NFV MANO, shall be permitted.

The ENI System may take on a further expanded role, assisting the NFV MANO System where appropriate as well as providing functionality that the NFV MANO System does not offer to the rest of the Assisted System. In this option, the ENI System sends recommendations and commands to the NFV MANO System for all functions that both the ENI and NFV MANO Systems have as well as recommendations and commands to the Assisted System that are from functions that only the ENI System has. This scenario shall use explicit communication, via ENI External Reference Points, to ensure that the ENI System, the Assisted System, and the NFV MANO System shall agree on which control, management operations are done by the ENI System, and which are done by the NFV MANO System. Similarly, the ENI System, the Assisted System, and the NFV MANO System shall agree on which operations require which type of messaging protocol (e.g. request-response, or one-way notification, or something else).

This scenario is applicable to situations where there is an evolution in the network that requires new functionalities that NFV MANO does not currently address. As an example, this may happen for situations where future network resource models will require more granular and multi-faceted analysis (mobile devices as cloud resource nodes, etc.), and will require a complex management capability, which is ideally augmented via ENI.

The ENI System may inform the NFV MANO System of recommendations and commands that it has made to other portions of the Assisted System, as this may influence the decision process of the NFV MANO System. It may also send repository change notifications to the NFV MANO System.

NOTE: Work on how decisions based on the closed loop architecture of ENI and the open loop architecture of NFV MANO is for study in Release 4 of the present document (see clause 9).

9.4.3 Interaction Scenarios for Assisted Policy Management in NFV MANO

NOTE: This is for further study in Release 4 (see clause 9).

9.5 Interaction with the MEF LSO RA

NOTE: This is for further study in Release 4 (see clause 9).

10 Areas for Future Study

10.1 Open Issues for the Present Document

Void.

10.2 Issues for Future Study

From clause 6.3.2.3.7.4 (Semantic Annotation):

NOTE 1: Release 4 will compare vectorized representations based on word frequencies (e.g. bag-of-words), word embeddings (e.g. words are mapped into a vector space where the distance between words in the space is related to the syntactic ad semantic features of the words; and example is word2vec), and document-level mechanisms (e.g. bi-directional encoder representations from transformers, or BERT) to determine which (if any) of these approaches could be used effectively in active learning.

From clause 6.3.2.4.2.2 (Detecting Anomalies):

NOTE 2: The following will be examined in Release 4 of the present document.

- **Statistical.** This uses historical data to model the expected behaviour of a system.
- **Probabilistic.** Similar to statistical, but uses probabilities or fuzzy algebra to do the comparison.
- **Distance-based Metrics.** This uses the distance between a newly measured datum and previous data, and defines the datum as an anomaly if the distance is larger than a pre-defined value.
- **Pattern Matching.** This compares each new measurement against a database of known anomalies, and classifies measurements that are more similar to known anomalies than to correct data as anomalies.
- **Structural Matching.** This compares the structure of the data to known data fields from models and/or ontologies. If the data matches correct or anomalous data, then a corresponding decision is made.
- **Clustering.** This projects measured data into a multi-dimensional space. Measurements that do not belong to, or are too far from a cluster, are classified as anomalies.

- **Ensemble Matching.** This approach uses a number of different algorithms to analyse each measurement, and then defines a collective vote from each method.
- **Machine Learning.** Different types of machine learning algorithms can be used. For example, deep neural networks (e.g. LSTM, or Long Short-Term Memory, which are a type of recurrent neural network capable of learning order dependence in sequence prediction problems) could be compared against autoencoders (e.g. a type of neural network that can be used to learn a compressed representation of raw data, which learns a condensed representation of the input data).

From clause 6.3.7.4.4 (Architecture of a Cognitive Functional Block):

NOTE 3: Information processing is for further study in Release 4 of the present document.

From clause 6.3.9.6.3.4 (A Single Unified Policy Management Architecture):

NOTE 4: This subject is for further study in Release 4 of the present document.

From clause 6.3.9.6.4 (Policy Management Federation):

NOTE 5: This is for further study in Release 4.

From clause 6.3.11.4.2 (Treating Output Generation as the Inverse of Normalization)

NOTE 6: This is for further study in Release 4.

From clause 6.4.3 (Function of the API Broker):

NOTE 7: An API Broker can also be used for API composition. This item is for further study in Release 4.

From clause 6.5.4 (Recommended Communication Patterns to be Used Between ENI and External Systems):

NOTE 8: This is for further study in Release 4.

From clause 7.3.6 (Reference Point $E_{app\text{-}eni\text{-}kno}$):

NOTE 9: These data and information will require a special protocol to exchange their content. That protocol will be defined in Release 4 of the present document.

From clause 9.2 (Generic Architecture):

NOTE 10: This topic will be further specified in Release 4 of the present document.

From clause 9.4.3 (Interaction Scenarios for Assisted Policy Management in NFV MANO):

NOTE 11: This is for further study in Release 4.

From clause 9.5 (Interaction with the MEF LSO RA):

NOTE 12: This is for further study in Release 4.

From clause A.1 (Integration with Other SDOs and Open Source Communities):

NOTE 13: The present document, along with work on other architectures (e.g. ZSM), is for further study in Release 4.

From clause A.2 (Integration with BBF CloudCO):

NOTE 14: The interaction between the ENI System and various open source communities is for further study in Release 4.

Annex A (informative): SDO and Open Source Interactions

A.1 Integration with Other SDOs and Open Source Communities

A.1.1 Introduction

This informative annex describes preliminary thoughts on integrating ENI with MEF-LSO and the BBF CloudCO architectures.

NOTE: The present document, along with work on other architectures (e.g. ZSM), is for further study in Release 4 (see clause 9).

A.1.2 Interaction with BBF CloudCO

The BBF CloudCO (Cloud Central Office) reference architecture is a Central Office Domain that is:

1) leveraging SDN and NFV techniques;

2) running on a cloud-like infrastructure deployed at Central Offices; and

3) accessed through a Northbound API, allowing Operators, or third parties, to consume its functionality, while hiding how the functionality is achieved from the API consumer.

Figure A-1 shows the interaction between the High-Level ENI Architecture and the BBF CloudCO architecture, see BBF TR-384 [i.39]. All the functional components that are shown in Figure A-1 are defined in BBF TR-384 [i.39] and ETSI GS NFV-MAN 001 [1]. The reference point between the ENI System and the BBF CloudCO architecture is being specified by ETSI ISG ENI. All the other reference points/interfaces are specified in BBF TR-384 [i.39] and ETSI GS NFV-MAN 001 [1].

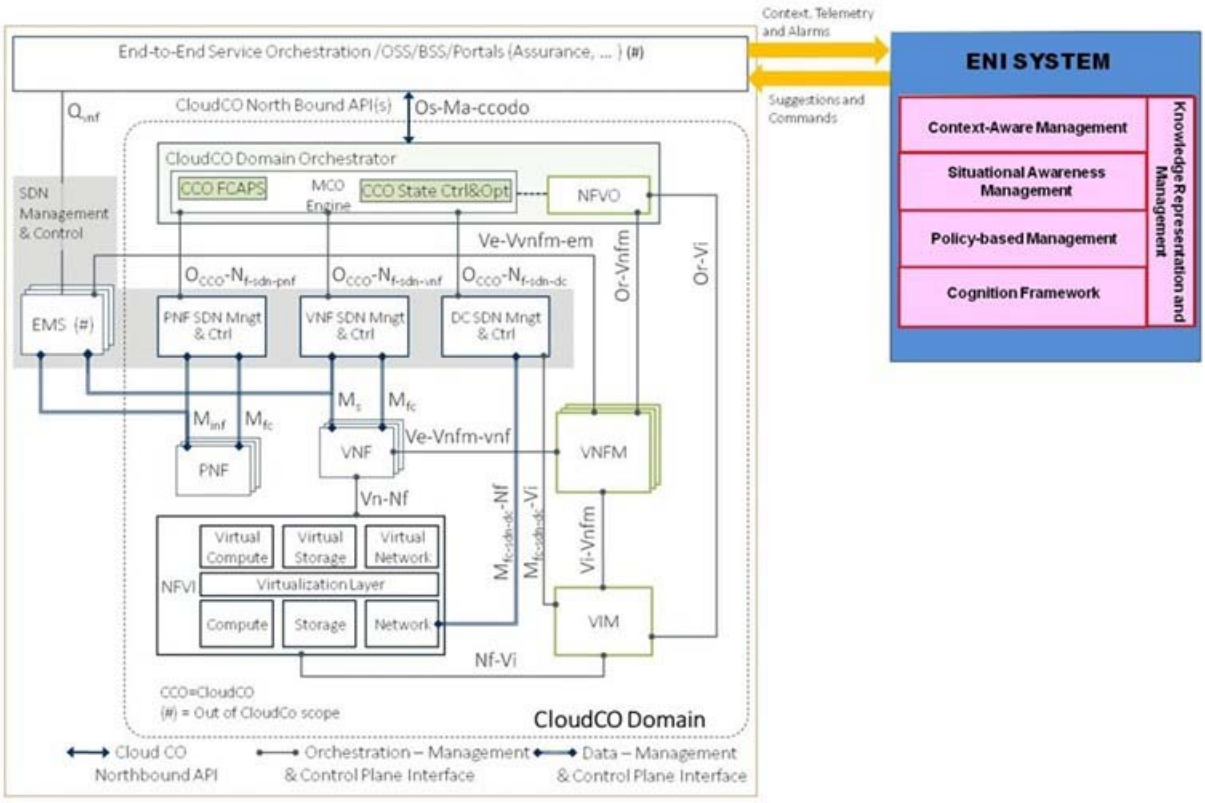

Figure A-1: Interaction between the BBF CloudCO and ENI

In Figure A-1, the shown End to End Service Orchestrator coordinates all client interfaces with regard to their privileges and views, and resolves any contentions that may arise. Furthermore, the E2E Service Orchestrator coordinates multiple CloudCO Domain Orchestrators, deployed on specific nodes. The CloudCO Domain Orchestrator manages, controls and orchestrates each CloudCO Domain that spans over portions of the whole access/edge network. Part of the E2E Service Orchestrator function is to as well coordinate user plane handoffs when services cross domain boundaries.

In this scenario the ENI System interacts with the CloudCO End-to-End Service Orchestration/OSS/BSS/Portals functional component via a reference point that is being specified by ISG ENI. The ENI System collects the necessary information from the CloudCo Domain Orchestrator on topics such as:

1) supported tenants;

2) supported use cases;

3) used and available resources per tenant and use case; and as well;

4) information related to lifecycle management of the virtual environment.

The output of the ENI System can be used for several purposes such as acting on the service/policy/resource module and engineering rules, recipes for various actions, policies and processes.

A.2 Interaction with Open Source Communities

NOTE: The interaction between the ENI System and various open source communities is for further study in Release 4.

Annex B (informative): ENI Architectural Evolution

B.1 ENI Architecture Evolution Motivation

Technology is evolving especially when it comes to Cognition, near natural language processing, deep reasoning and even situation awareness and context awareness.

Industry agreement as to Policy semantics (not DSL) and data normalization and tools is still work in progress, at best.

Multiple PoC are emerging at ETSI ENI and elsewhere in the industry. These PoC are using off the shelf technology with limited use of afore mentioned advanced technologies.

Need to coordinate architecture with related SDOs and Industry Consortia.

Need to coordinate Reference Points with other SDOs before finalizing the model. Additional challenges.

Data ingestion: use different tools today, need to be normalized into one common and COORDINATED data with agreed representation!

Infrastructure (as known as "Existing System") is still silo'ed hard to accomplish coordinated configuration, control and management of Network with Compute + Storage.

B.2 ENI Architecture Evolution Proposal

- Define few evolutional phases.
- Align PoC, Open Source and other SDO and industry consortia to ENI phases.
- Interact with open source communities!
- Update ENI Architecture based on industry progression.
- The above is designed to make ENI the focal of ML/AI evolution for the Telco industry and for its use of various Orchestration and Cloud Operation Environments (and network industry?) by ensuring tight coupling.
- Time Limit: limit evolution and publication of successive ENI phases, to no more than 18 months after the first publication of the ENI Architecture.

B.3 Proposed Definition of the ENI Phases

An "ENI Phase" is defined as compliance with a set of ENI Reference Points and its specific version, as set by the ENI WG from time to time.

For clarity, it is expected that a Reference Point may be versioned as the ENI architecture evolves.

The architecture document (i.e. the present document) will continue as currently planned and will not slow down or change course.

For each "ENI Phase", ENI will consider inputs from its documents and members, ongoing PoC, interaction and liaison with SDO and collaboration with open source communities to determine the set of Reference Points required to be supported.

An ENI implementation, MAY claim compliance for a given ENI Phase when it is in compliance with a set of Reference Points set by the ENI WG, as aforementioned.

It is recommended that the Architecture WG will provide specific recommendation of the phases using Reference Points.

Annex C (informative): Bibliography

- The Moral Machine.

History

Document history		
V1.1.1	September 2019	Publication
V2.1.1	December 2021	Publication
V3.1.1	June 2023	Publication