



Deliverable 5.2

First version of the simulation environment and monitoring solutions

Technical References

Document version : 1.0
Submission Date : 31/08/2022
Dissemination Level : Public
Contribution to : WP5- Methods and Tools for Auditing ICT Ecosystems
Document Owner : CNR
File Name : BIECO_D5.2_31.08.2022_V1.0
Revision : 3.0

Project Acronym : BIECO
Project Title : Building Trust in Ecosystem and Ecosystem Components
Grant Agreement n. : 952702
Call : H2020-SU-ICT-2018-2020
Project Duration : 36 months, from 01/09/2020 to 31/08/2023
Website : <https://www.BIECO.org>

Revision History

REVISION	DATE	INVOLVED PARTNERS	DESCRIPTION
0.0	02/02/2022	CNR	TOC
0.1	30/03/2022	CNR	TOC revision
0.2	27/04/2022	CNR	Draft content
0.3	19/05/2022	CNR	Revision of the content
0.4	10/06/2022	CNR	Sections editor assignment
0.5	22/06/2022	CNR	Sections 2, 3.2, and 9.4 first version
0.6	27/06/2022	CNR+UNI	Section 3.4 and Section 4 first version
0.7	28/06/2022	IESE	Content Added
0.9	29/06/2022	HS	GUI content first draft
1.0	29/06/2022	CNR	Runtime monitor performance evaluation
1.1	04/07/2022	CNR	Section 8
1.2	06/07/2022	UMU	Section 2.2
1.3	07/07/2022	GRAD	Subsection 6.1
1.4	07/07/2022	CNR	Refinement of sections 1 introduction, 3.4 Ontology Manager, 7 Advancement about Runtime Monitoring
1.5	08/07/2022	IESE	Description of the Predictive Simulation Rest Interface (Chapter 3.3.3) Future Work for Predictive Simulation w.r.t. simulation models added.
1.6	08/07/2022	CNR	Executive Summary
1.7	11/07/2022	UNI	Revision section 3.4. Update sub-section 3.4.3 Update Acronymous list and section 9
1.8	15/07/2022	CNR	Overall check. Revision Section 3.4. Section 4, Section 8. and Section 9.
1.9	18/07/2022	IESE	Revision of section 1.1 challenges, Section 3.3 Predictive Simulation implementation, Section 6 advancement of the Predictive Simulation Section 9 Future work
2.0	20/07/2022	CNR	Final version for internal review
2.1	22/08/2022	CNR	UMU review comments integration.
2.2	22/08/2022	CNR	GRAD review comments integration and acronyms list finalized
2.3	25/08/2022	UNINOVA	Coordinator review and Edition
3.0	31/08/2022	UNINOVA	Coordination Finalization and Submission

List of Contributors

Deliverable Creator(s): Antonello Calabrò (CNR), Said Daoudagh (CNR), Felicita Di Giandomenico (CNR), Eda Marchetti (CNR), Rudolf Erdei (HS), Ana Inês Oliveira (UNI), Filipa Ferrada (UNI), Sara Matheu (UMU), Emilia Cioroai (IESE), Ioannis Sorokos (IESE), Eva Sotos (GRAD).

Reviewers: Adrian Sanchez Cabrera (UMU), Eva Sotos (GRAD), Sanaz Nikghadam-Hojjati (UNINOVA), José Barata (UNINOVA)

Disclaimer: The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

All rights reserved.

The document is proprietary of the BIECO consortium members. No copying or distributing, in any form or by any means, is allowed without the prior written agreement of the owner of the property rights.

This document reflects only the authors' view. The European Community is not liable for any use that may be made of the information contained herein.



BIECO project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952702.

Acronyms

Acronym	Term
API	Application Programming Interface
BIECO	Building Trust in Ecosystems and Ecosystem Components
CE	Controlled Environment
CEP	Complex Event Processor
ConSert	Conditional Safety Certificate
DSL	Domain Specific Language
DT	Digital Twin
EDR	Endpoint Detection and Response
GUI	Graphical User Interface
ICT	Information and Communications Technology
IETF	Internet Engineering Task Force
JMS	Java Message Service (JMS)
JSON	JavaScript Object Notation
KB	Knowledge Base
MENTORS	Monitoring ENvironment FOR Sos
MONTOLOGY	MONitoring onTOLOGY
MQTT	Message Queuing Telemetry Transport
MUD	Manufacturer Usage Description
OWL	Web Ontology Language
POM	Project Object Model
RDF	Resource Description Framework
REST	REpresentational State Transfer
RtE	Runtime Evidence
SIEM	Security Information & Event Management
SoS	System Of Systems
SUA	System Under Audit
UC	Use Case
XDR	eXtended Detection and Response
XML	Extensible Markup Language

Executive Summary

Work Package 5 goal is to define and develop techniques, methods, and tools supporting the audit activity in the BIECO project. Thus, the Auditing Framework has been developed as a constituent of Work Package 5 and integrated into the BIECO Runtime phase. The Auditing Framework exploits the field data produced by devices, components, sensors, services, Systems of Systems (SoS), or ecosystems involved in the auditing activity to predict and assess functional and non-functional properties and increase the overall ecosystem's trustworthiness.

This document reports the Work Package 5 activities performed to improve the Deliverable D5.1 preliminary framework proposal and realizes the current first version of the Auditing Framework and its main components. Specifically, the deliverable targets:

- The Runtime Monitoring is responsible for collecting on-the-field data events and assessing an established set of functional and non-functional properties.
- The Predictive Simulation provides the Runtime Monitor with suitable predictions about future systems or components' behaviours.
- The Ontology Manager manages the knowledge about the different Ecosystem entities (such as devices and components) and the specification process of the monitoring rules.
- The Auditing Framework GUI manages the interaction with the user. It also provides features for the Auditing Framework setup and execution and data storage and retrieving.

The deliverable reports also the research advancements performed during the project's second year (from M13 to M24) regarding the methodologies and features implemented in Auditing Framework components and its validation through one of BIECO's available use cases.

Considering the Work Package 5 objectives (as reported in the BIECO DoA), the deliverable targets the implementation of the first version of the Auditing Framework that enables:

1. The definition of the executable simulation models and the parameters against which the behaviour of the ICT systems and their interacting actors are judged as trustworthy or not.
2. The definition of monitoring methodologies and tools for detecting malicious behaviours of ICT systems and their interacting actors and assessing the validity of the simulation models.
3. The definition of monitoring tools able to validate the simulation decisions through real-time data of systems sensors and actuators.

In addition, to improve the usability of the Auditing Framework and the knowledge management, the following (extra DoA) objectives have been added and targeted during the second year of the BIECO project:

4. Definition of a dynamic, user-friendly, and adaptable methodology for specifying functional and non-functional properties and their management.
5. Definition of holistic support for knowledge management and data sharing within the overall BIECO Runtime Phase.
6. Definition of the Auditing Framework GUI for improving the usability in customizing the Auditing Framework and managing its executions.

The activities reported in this deliverable are strictly connected with Work Packages 3, 4, 6, 7, and 8.

Targeting reviews' comments, indications, and possible improvements

Considering the BIECO General Project Review Consolidated Report, we provide clarifications about the Work Package 5 comments in the following.

R.R.#1: Review Report - Section 1.3. - Auditing System Framework - page 4/17

Concerning Auditing Framework execution, in the Introduction, Section 2.2, and Section 8, the deliverable clarifies that execution of the Auditing Framework can be performed considering three different situations:

- a. using a simulation of the Controlled Environment (CE).
- b. using a testing environment (or forensic reconstructions) in which the CE components can either be executed in a real context, simulated models, or stubs.
- c. using operational systems where the CE components and the SUA are executed in the operating (real) environment.

R.R.#2: Review Report - Section 1.3. - Auditing System Framework - page 4/17

Concerning the Use of Extended MUD file, Section 2.2.1 details the use of the Extended Mud file.

R.R.#3: Review Report - Section 1.5. - 5. (Vulnerability detection and forecasting tool)- page 5/17

The Introduction, Section 2.2, and Section 8 of the deliverable clarify that the Auditing Framework can be executed considering different situations (see R.R.#1 above). The provided implementation can be executed at runtime in the production environment without adaptations or improvements.,

R.R.#4: Review Report - Annex 1- D5.1- page 13/17

This deliverable provides the technical description of the Auditing Framework and details about its validation. In particular:

- in Section 2 provides the improved Auditing Framework architecture and details about the Blueprints management.
- Section 3 presents the Auditing Framework (and its components) implementation. It also details the GitHub repository, technologies used, and licenses needed.
- Section 8 presents the Auditing Framework execution on one of the BIECO use cases.

Project Summary

Nowadays, most ICT solutions companies develop require the integration or collaboration with other ICT components, which third parties typically create. Even though these kinds of procedures are essential to maintain productivity and competitiveness, the fragmentation of the supply chain can pose a high-risk regarding security, as in most cases, there is no way to verify if these other solutions have vulnerabilities or if they have been built considering the best security practices.

Companies must change their mindset to deal with these issues, assuming an "untrusted by default" position. According to a recent study, only 29% of IT businesses know that their ecosystem partners are compliant and resilient to security. However, cybersecurity attacks have a high economic impact, and it is not enough to rely only on trust. ICT components need verifiable guarantees regarding their security and privacy properties. It is also imperative to detect vulnerabilities from ICT components more accurately and understand how they can propagate over the supply chain and impact ICT ecosystems. However, it is well known that most of the vulnerabilities can remain undetected for years, so it is necessary to provide advanced tools to guarantee resilience and better mitigation strategies, as cybersecurity incidents will happen. Finally, it is essential to expand the horizons of the current risk assessment and auditing processes, considering a much broader threat landscape. BIECO is a holistic framework that will provide these mechanisms to help companies to understand and manage the cybersecurity risks and threats they are subject to when they become part of the ICT supply chain. The framework, composed of tools and methodologies, will address the challenges related to vulnerability management, resilience, and auditing of complex systems.

Partners



Disclaimer

The publication reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Table of Contents

Technical References	1
Revision History.....	2
List of Contributors	2
Acronyms.....	4
Executive Summary.....	5
Project Summary.....	7
Partners.....	8
Disclaimer	8
Table of Contents.....	9
List of Figures.....	12
1. Introduction.....	14
1.1. Auditing Framework Challenges	15
1.2. Roadmap	17
2. Auditing Framework in the BIECO Runtime Phase.....	18
2.1. Using Inferred knowledge Auditing Framework’s main components	18
2.2. Auditing Framework Blueprints	19
2.2.1. Extended MUD File	19
2.2.2. Conditional Safety Certificate (ConSerts)	21
3. Auditing Framework Implementation	23
3.1. Auditing Framework Implementation Details	23
3.1.1. Communication Flows Intra-Nodes/Components.....	23
3.1.2. Technologies Used.....	23
3.1.3. Licenses	23
3.1.4. Github Details Repository	23
3.2. Runtime Monitoring	24
3.2.1. Internal Artifact’s Structure	24
3.2.2. Communication Flows Inter Artifacts.....	30
3.2.3. Exposed Interfaces	30
3.2.4. Exchanged Data Structure	31
3.2.5. Produced Data	32
3.2.6. Technologies Used.....	33
3.2.7. User Installation Guidelines.....	33

3.3.	Predictive Simulation	34
3.3.1.	Internal Artifacts Structure	34
3.3.2.	Communication Flows Inter Artifacts.....	35
3.3.3.	Exposed Interfaces	35
3.3.4.	Exchanged Data Structure	37
3.3.5.	Produces data	37
3.3.6.	Technologies Used.....	38
3.3.7.	User Installation Guidelines.....	38
3.4.	Ontology Manager	38
3.4.1.	Internal Artifacts Structure	39
3.4.2.	Communication Flows and Exposed Interfaces	41
3.4.3.	Exchanged Data Structure	43
3.4.4.	Produces Data	43
3.4.5.	Technologies Used.....	44
3.5.	Auditing Framework GUI	45
3.5.1.	Communication Flows Inter Artifacts.....	45
3.5.2.	Technologies Used.....	45
3.5.3.	User Installation Guidelines.....	45
4.	Advancements in Ontology Manager.....	46
4.1.	SoS Module	47
4.2.	Attributes Module	47
4.3.	Skills Module	48
4.4.	Rule Module	48
4.5.	Monitoring Module	50
5.	Advancement of the Auditing Framework Interface.....	51
6.	Advancement of the Predictive Simulation	52
6.1.	Derivation of Specialized Digital Twins	52
6.1.1.	The Process.....	53
6.1.2.	Derivation of Specialized Timing Digital Twins.....	53
7.	Advancements of Runtime Monitoring	55
7.1.	Runtime Monitor Innovation Aspects	56
7.1.1.	Leveraging the XDR.....	56
7.1.2.	Leveraging the EDR	57
7.1.3.	Leveraging the SIEM	57

8. Auditing Framework Execution.....	58
8.1. UC 4 Coppelia	59
8.1.1. Pre-Setup Phase	60
8.1.2. Offline Activities	63
8.1.3. Finish Pre-Setup	65
8.1.4. Start Auditing Framework.....	67
8.1.5. Validation Scenario	70
9. Conclusions	72
References.....	73
Appendix A. Runtime Monitoring Instrumentation Guidelines	74
A. Introduction	74
A.1 Overview	74
A.2 Input/Output Captured by the Specific Probe	75
A.3 Message-Based Notification (Made by the User)	77
A.4 Mentors Probe Code Injection	77
A.5 Automatic Instrumentation (ext. Service)	80
B. Event Description	81

List of Figures

Figure 1: An updated view of the Auditing Framework	18
Figure 2: Extended MUD model.....	20
Figure 3: Usage of the MUD file within BIECO.....	21
Figure 4: Threat MUD structure.....	21
Figure 5: Runtime Monitoring Components	24
Figure 6: Service Status from the BIECO project	25
Figure 7: Service Status from the BIECO project	26
Figure 8: Class diagram of the connections Register	27
Figure 9: JSON2MQTT Mediator behavior	28
Figure 10: Elements of the exposed REST interface	30
Figure 11 Elements of the exposed REST interface	31
Figure 12 Class diagram of Event Structure	31
Figure 13 Internal structure of the Predictive Simulation components.....	34
Figure 14 Flow between the internal artefacts of the Predictive Simulation.....	35
Figure 15 Structure of Predictive Simulation Rest Interface.....	37
Figure 16 Ontology Manager Reference Architecture	39
Figure 17 Screenshot of WebProtégé during the development of BIECO Ontology	40
Figure 18 Overview of Ontology Server RESTful Interface.....	41
Figure 19 Example of Ontology Server API endpoints for the SoS module	42
Figure 20 Example of GET and POST requests through the Ontology server	42
Figure 21 Ontology Manager Data Structure JSON Schema	43
Figure 22 An instance of SoS data.....	44
Figure 23 Ontology Modules	46
Figure 24 System of Systems (SoS) module	47
Figure 25 Attributes Module.....	47
Figure 26 Skills Module.....	48
Figure 27 Rule Module	49
Figure 28 Rule Transformation Process.....	49
Figure 29 From Abstract to Well-defined rule enrichment process	49
Figure 30 Drools Rule Skeleton	50
Figure 31 Monitoring Module	50
Figure 32 Execution of parallel abstraction	52
Figure 33 Coppelia Simulator.....	59

Figure 34 Runtime GUI: Auditing Framework Setup	59
Figure 35 Auditing Framework GUI: Auditing Framework Pre-Setup	60
Figure 36 Auditing Framework GUI: SoSs selection	60
Figure 37 Auditing Framework GUI: Device selection	61
Figure 38 Auditing Framework GUI: Components selection	61
Figure 39 Auditing Framework GUI: Skills selection	62
Figure 40 Auditing Framework GUI: Select/adapt abstract rules selection	62
Figure 41 Auditing Framework GUI: Refine Abstract Ruleset	63
Figure 42 Auditing Framework GUI: Refine Abstract Ruleset	64
Figure 43 Auditing Framework GUI: Probe Injection	64
Figure 44 Digital Twin Eclipse Profile: Digital Twin development	65
Figure 45 Auditing Framework GUI: Finish Pre-Setup phase	66
Figure 46 Auditing Framework GUI: Standard Well-defined rule refinement	66
Figure 47 Auditing Framework GUI: Pure predictive Well-defined rule refinement	67
Figure 48 Trace of Connection message sent by SUA_Probe	68
Figure 49 Trace of Velocity messages sent by SUA_Probe	68
Figure 50 Trace of forecast messages sent by DT_Probe	69
Figure 51 Trace of rule self-generated by the Runtime Monitoring	69
Figure 52 Coppelia Simulator: Execution of a malicious code attack	70
Figure 53 Runtime Monitoring Logger: Trace of rule violation raised	70
Figure 54 Coppelia Simulator: System turns back to a safe condition	71
Figure 55 Code Instrumentation process	74
Figure 56 Serial port probe example	77
Figure 57 example folder	78
Figure 58 SUA probe example	78
Figure 59 ConcernAbstractProbe	79
Figure 60 Simple Probe	80
Figure 61 Specific instrumentation tool	81
Figure 62 Specific BIECO event	82

1. Introduction

This deliverable focuses on developing techniques, methods, and tools supporting the audit activity in the BIECO framework. It describes the first version of the simulation environment and the monitoring solution and details their preliminary implementation and validation. It also details the additional components, the Ontology Manager and the Auditing Framework GUI (Graphical User Interface), included in the framework for making user interaction and knowledge management easier.

Indeed, the Auditing Framework is the BIECO dynamic mechanism for the online analysis of functional and non-functional properties of an entity against well-stated conditions, such as contractual conditions for trust. The Auditing Framework collects events at different specification levels and goals and from heterogeneous sources (such as applications, components, sensors, or devices). It uses the collected data to infer complex patterns that indicate specific functional and non-functional properties. Those patterns represent the observed behaviours and are compared with the trustable ones of the monitored entities to detect anomalies, vulnerabilities, or problems.

As introduced in the deliverable D5.1 [11], the Auditing Framework can:

- a) Collect and analyse data from the different SoS sources (e.g., applications, sensors, software, and hardware components or devices).
- b) Assess the run time behaviour of the SoSs (components or devices) based on the expected behaviour rules.
- c) Promptly raise alarms in case of violations, anomalies, or misbehaviours.

Within the BIECO project, the Auditing Framework leverages the current state of the practice in diverse ways:

- Providing the implementation of a dynamic, user-friendly, and adaptable methodology for the specification of functional and non-functional properties. It is also holistic support for knowledge management and data sharing within the overall BIECO runtime phase.
- Including a predicting simulation system for anticipating the behaviour of a trustable ecosystem (components). Working with the monitoring facilities represents a dynamic oracle defining the trustable patterns.
- Providing an integrated mechanism for assessing the correctness of the run time executions of the ecosystem and its components against the collected prediction without knowing the source code structure; and
- Providing a dynamic, user-friendly, and adaptable methodology for managing the alarms, triggering the corresponding notifications, and executing the associated countermeasures.

As detailed more in this deliverable, the Auditing Framework is fully integrated into the BIECO Framework. Indeed, it uses the Runtime Phase mechanism for retrieving and integrating the Design Phase data (hereafter called Blueprints) into its knowledge base. It also uses the BIECO framework execution environment for collecting events about the entities to be audited.

Following the terminology introduced in D5.1[11], in this deliverable, the System Under Auditing (SUA) indicates the device, the component, or the system that is the target of the auditing activity, and the Controlled Environment (CE) refers to the environment in which SUA is executed.

The Auditing Framework has been developed independently from the realization of the CE. Therefore, the auditing activity can be performed considering three different situations:

- a) CE executed in a simulation context, i.e., the CE is abstracted as a simulation model.
- b) CE executed in a testing context, i.e., the CE is executed in a testing environment, and the components, directly interacting with the SUA, are either executed in the real context or simulated model or executed using stubs.
- c) CE executed in a real context, i.e., the CE and its components are executed in a real environment.

In all the above situations, the required precondition for Auditing Framework execution is the instrumentation of the CE and SUA with probes, i.e., pieces of code injected in the entities for sending events about the execution.

1.1. Auditing Framework Challenges

This section provides the main challenges by referring to the deliverable D8.2 [17] for a detailed list of the Auditing Framework objectives. The remainder of the deliverable details how these challenges have been targeted inside Work Package 5.

CH1: Whitening the Black-Box Assessment Process

Using probes inside the CE and SUA lets the Auditing Framework collect internal execution data (white-box data) without knowing their source code structure (black-box data). Indeed, the implemented methodology makes the CE and the SUA more “transparent” for functional and non-functional properties assessment and prediction without revealing their internals. Data is collected through probes preserving the principles of loose coupling and implementation neutrality.

CH2: Separating Properties Predictions and Assessment

The implementation of the Auditing Framework follows the principle of independence between the components. All the conceived components have a specific role and contribute to the overall quality, usability, and effectiveness. However, the Auditing Framework can work with few adjustments, excluding some of its components:

- Predictive Simulation could be executed independently from the monitoring activity. In this case, predictions can be used by an external monitoring engine or for different purposes.
- Monitoring activity could run independently from the Knowledge Management process and the Predictive Simulation decisions. Indeed, the monitoring properties could be provided as an external data set, and the properties assessment executed without having the oracle decision (i.e., as any standard monitoring engine).
- Knowledge management processes could run independently from monitoring and Predictive Simulation activities. Indeed, the classification and collection of the specific peculiarities, properties, and quality attributes of the different

ecosystems and their components and devices is a starting point for any development and assessment activity.

CH3: Leveraging the Existing Monitoring Solutions

The Auditing Framework leverages the existing monitoring solutions considering several aspects. In particular:

- The implemented monitoring solution includes and leverages some of the features proposed by solutions like XRD (eXtended Detection and Response), EDR (Endpoint Detection and Response), and SIEM (Security Information & Event Management).
- The implemented monitoring component is an open-source component that can collaborate with other available monitoring solutions (like Ganglia¹, Zabbix², and Netdata³) with required adaptations.
- The implemented monitor solution lets diverse levels of reaction to the detected violations or misbehaviour. Indeed, notifications, execution termination, and application of suitable countermeasures to make the auditing execution continue in a safe mode are all possible.
- The functional and non-functional properties collected through the knowledge management process and assessed by the monitoring solution can be easily customized, enriched, or modified. The properties dataset also represents a shared dataset exploitable for further application or research activity.
- The implemented knowledge management process lets the user customize the manual/automatic countermeasures based on the risk analysis. This possibility lets mitigate the vulnerability detection risks and keep the human in the loop.
- The implemented monitoring solution can be easily modified by using smart agents to automatically apply the possible countermeasures on the CE or SUA in agreement with the loose coupling principles.

CH4: Leveraging the failure Prediction Methods

Auditing Framework leverages the prediction of malicious attack by implementing the Predictive Simulation components that enable evaluation of a software behaviour before it is executed thanks to the use of specialized Digital Twin (DT), i.e., abstract models representing the executable abstractions of the ecosystem components (ICT systems, ICT system components, and actors) and their interactions. In case of failure prediction caused by a malicious attack, the Predictive Simulation methods let discover performance degrading over time and sporadic and specific situations when a destruction target impact of an attacker is likely to be achieved.

CH5: Leveraging the Implementation of the Digital Twins

The Auditing Framework leverages the existing approaches for implementing the Digital Twins by creating a model that can capture the failure in the “pure Predictive Simulation phase,” where the conformity between the abstract models and the real-world digital

¹ <http://ganglia.info/>

² <https://www.zabbix.com/>

³ <https://www.netdata.cloud/>

asset is evaluated in a set of conformity testing. Further on, Predictive Simulation abstracts specifics of the models directed to the scope of the evaluation (timing synchronization, functional interaction) that needs to be defined before deployment.

1.2. Roadmap

The deliverable is structured as follows:

- Section 2 presents the Auditing framework and details the used Blueprints.
- Section 3 details the implementation of the Auditing Framework and its components. Specifically, Section 3.1 describes the overall implementation of the Auditing Framework and the interaction with the Runtime Phase. Sections 3.2, 3.3, 3.4, and 3.5 detail the Runtime Monitoring, the Predictive Simulation, the Ontology Manager, and the Auditing Framework GUI, respectively.
- In Sections 4, 5, 6, and 7 report advancements in the Ontology Manager, the Auditing Framework GUI, the Predictive Simulation, and the Runtime Monitoring specification with respect to deliverable D5.1 [11], respectively.
- Section 8 describes the application of the Auditing Framework to one of the BIECO use cases, while Section 9 provides discussion and future works.
- Finally, Appendix A provides additional material about the Auditing Framework implementation.

2. Auditing Framework in the BIECO Runtime Phase

By referring to deliverable D2.4 [9] for a detailed description of the Auditing Framework architecture, this section provides details about its execution inside the BIECO Runtime Phase. In particular, Section 2.1 illustrates the updated Auditing Framework architecture, whereas Section 2.2 discusses the definition and management of BIECO Blueprints.

2.1. Using Inferred knowledge Auditing Framework's main components

This section summarizes the main components of the Auditing Framework architecture already introduced deliverables D5.1 [11] and D2.4 [9]. In this deliverable, the interactions with the BIECO middleware and the additional components of the BIECO Runtime Phase are omitted to focus on the Auditing Framework activity.

As in Figure 1, the Auditing Framework includes five main components: (1) Runtime Monitoring, (2) Auditing Framework GUI, (3) Predictive Simulation, (4) Ontology Manager, and (5) Auditing Framework message BUS.

For each of them, here below, a brief description is provided. We will discuss those components in detail in the remainder of the deliverable.

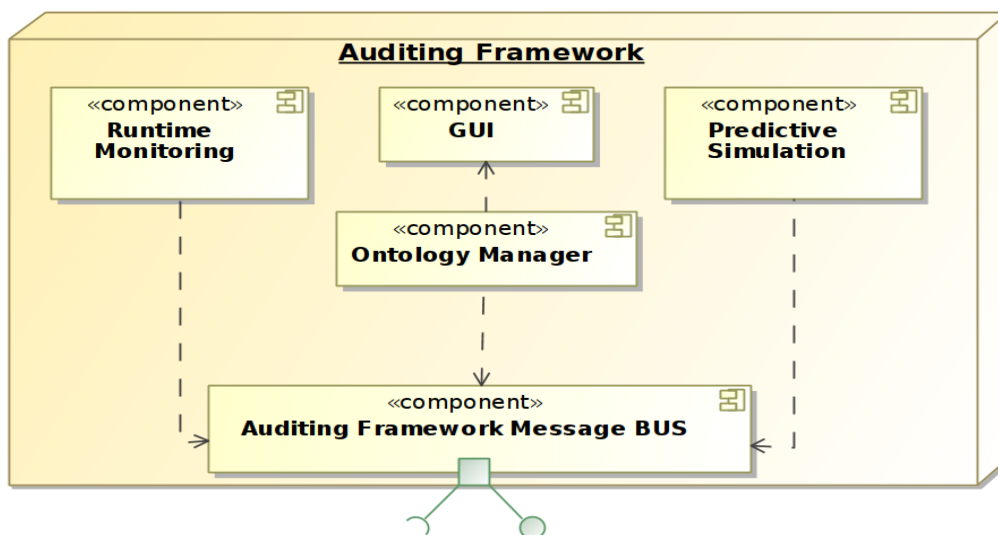


Figure 1: An updated view of the Auditing Framework

(1) Runtime Monitoring: It uses SUA and CE events to match a predefined set of functional and non-functional properties that the CE and the SUA should satisfy. Runtime Monitoring works in collaboration with the Predictive Simulation component for assessing specific rules focused on SUA behaviour predictions; Ontology Manager for receiving the set of properties (rules) to be monitored during the execution; and the Auditing Framework Message Bus for receiving the events.

(2) GUI: The GUI component, hereafter called Auditing Framework GUI, resides within the BIECO User Interface. Its primary purpose is to let the user interact with the Framework. That includes the Auditing Framework setup and settings for the project level. The Auditing Framework GUI interacts with the Ontology Manager, requesting available setup

information and providing the user input. The Auditing Framework GUI also manages the project state, saving the user-provided information for later use.

(3) Predictive Simulation: The Predictive Simulation component resides on the BIECO server. It targets the definition of the Digital Twin, i.e., abstract models representing the executable abstractions of the ecosystem components (ICT systems, ICT system components, and actors) and their interactions. These will be used to predict the future ecosystem components' behaviour.

(4) Ontology Manager: This component is the manager of the knowledge for the classification and categorization of the different Systems of Systems, their devices, and components, as well as the relative skills and functional and non-functional properties. The ontology manager is responsible for the management of the specification process of the monitoring rules: from the abstract to well-defined, and finally to instantiated rules.

(5) Auditing Framework Message BUS: This component is the backbone of the Auditing Framework. It manages all the communication between all the framework parts and relies on two technologies: Java Message Service (JMS) Messages and REpresentational State Transfer (REST) interfaces. Those technologies are exploited using Apache Artemis⁴ and Java Spring Boot with Thymeleaf⁵.

2.2. Auditing Framework Blueprints

This section describes the Blueprints provided by the Design Phase and used inside the Auditing Framework. They are the Extended MUD File and the ConSerts data (detailed in Sections 2.2.1 and 2.2.2, respectively). The Data Collection Tool (see deliverable D2.4 [1] for a detailed description) stores the Blueprints, while the Ontology Manager component (detailed in Sections 3.4 and 4) integrates them into the ontology used by the Auditing Framework.

2.2.1. Extended MUD File

As detailed in deliverable D6.2 [12], the MUD file is an IETF⁶ standard that the manufacturer can use to describe in a homogeneous the expected network behaviour of a particular device in terms of Access Control Lists. Dealing with its lack of expressiveness beyond the network layer and communication aspects, the BIECO Task T6.2 defined an extended MUD model. It can describe more fine-grained aspects beyond the network layer (see Figure 2), such as cryptographic configuration, the number of communications allowed, REST services offered and accessed by the device, and even known vulnerabilities and weaknesses associated with the device (see deliverable D6.2 for more details).

The extended MUD file is created from the standard MUD during the BIECO Design Phase, using the Resilblockly tool⁷ [12] (developed inside Work Package 6) for preliminary risk analysis (see Figure 2). At the end of the security evaluation methodology developed in BIECO Work Package 7, the extended MUD is updated,

⁴ <https://activemq.apache.org/components/artemis/>

⁵ <https://spring.io/projects/spring-boot>

⁶ <https://www.ietf.org/>

⁷ <https://resilblockly.resiltech.com:8407/#/>

reflecting the actual values and configuration obtained from the security testing. This updated MUD is intended to be used to deploy the device securely and to detect suspicious behaviours not reflected by the MUD.

During the auditing phase setup, as depicted in Figure 3, the updated MUD, among others, can be used as a Blueprint and integrated into the ontology data to define the functional or non-functional monitoring rules. Indeed, during the Auditing Framework execution, the Runtime Monitoring can detect deviations or violations of the established rules. Consequently, it can promptly apply mitigation actions or update the MUD policies in case of new functionality or configuration deviations.

<pre> 1 module: ietf-access-control-list 2 +--rw acls! 3 +--rw acl* [name] 4 +--rw acls 5 +--rw access-list* [name] 6 +--rw name 7 +--rw type? 8 +--rw aces 9 +--rw ace* [name] 10 +--rw name 11 +--rw matches 12 +--rw mud 13 +--rw manufacturer? 14 +--rw same-manufacturer? 15 +--rw model? 16 +--rw local-networks? 17 +--rw controller? 18 +--rw my-controller? 19 +--rw database? 20 +--rw eth? 21 +--rw ipv4? 22 +--rw ipv6? 23 +--rw tcp? 24 +--rw udp? 25 +--rw icmp? 26 +--rw egress-interface? 27 +--rw ingress-interface? 28 +--rw [keys]?* 29 +--rw kty? 30 +--rw alg 31 +--rw crv? 32 +--rw length 33 +--rw key_ops 34 +--rw purpose? 35 +--rw x5u? 36 +--rw x5c? 37 +--rw [application-protocol]?* 38 +--rw protocol 39 +--rw version 40 +--rw num-connections 41 +--rw [resource]?* 42 +--rw url 43 +--rw [method] * 44 +--rw auth? 45 +--rw key 46 +--rw value 47 +--rw keepAlive? 48 +--rw actions 49 +--rw attachment--points </pre>	<pre> 1 module: ietf-mud 2 +--rw mud! 3 +--rw mud-version 4 +--rw mud-url 5 +--rw last-update 6 +--rw mud-signature? 7 +--rw cache-validity? 8 +--rw is-supported 9 +--rw systeminfo? 10 +--rw mfg-name? 11 +--rw model-name? 12 +--rw firmware-rev? 13 +--rw software-rev? 14 +--rw documentation? 15 +--rw extensions* 16 +--rw from-device-policy 17 +--rw to-device-policy 18 +--rw [weaknesses]?* 19 +--rw id 20 +--rw name 21 +--rw description 22 +--rw date? 23 +--rw last_modified? 24 +--rw likelihood 25 +--rw impact 26 +--rw risk 27 +--rw [vulnerabilities]?* 28 +--rw id 29 +--rw name 30 +--rw description 31 +--rw date? 32 +--rw likelihood 33 +--rw cvss 34 +--rw risk </pre>
--	---

Figure 2: Extended MUD model

If the security issues detected by the Auditing Framework cannot be mitigated, it may imply an update of the system and even a re-evaluation of the system security. In this case, the security evaluation methodology (described in the Work Package 7 D7.2 [15] and D7.3 [16] deliverables) should be re-executed, focusing on the affected component and security property, which could also imply an update of the MUD file based on the test results.

A further (future) investigation for the Auditing Framework is the possibility to let vulnerability detection information sharing. Indeed, when the SUA component is executed in a real context, the Auditing Framework could provide facilities for sharing information about the affected components with the interested parties. In this sense, a threat MUD file could be created to define mitigation policies to restrict access to the compromised component or service. For this purpose, deliverable D6.3 [13] provides more details about the structure of the threat MUD (see Figure 4), and the architecture needed to share and obtain it.

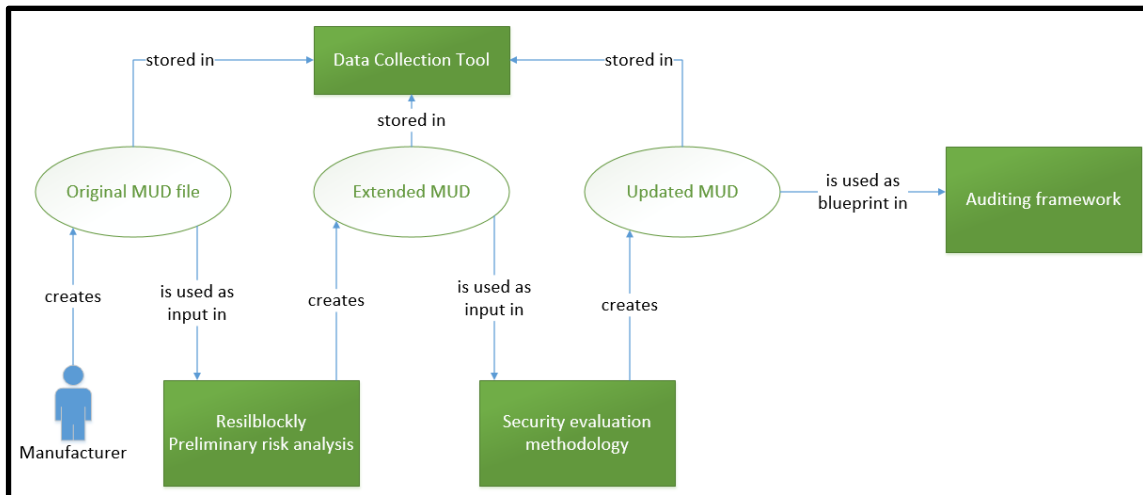


Figure 3: Usage of the MUD file within BIECO

```

1  module: ietf-threat-mud
2  +--rw threatmud!
3  +--rw threat-mud-version
4  +--rw threat-mud-url
5  +--rw last-update
6  +--rw threat-mud-signature?
7  +--rw cache-validity?
8  +--rw is-supported
9  +--rw threat-intelligence-provider
10 +--rw threat-name
11 +--rw cvss-vector?
12 +--rw documentation?
13 +--rw extensions*?
14 +--rw from-device-policy
15 +--rw to-device-policy
  
```

Figure 4: Threat MUD structure

2.2.2. Conditional Safety Certificate (ConSerts)

A Conditional Safety Certificate (ConSert) [22], [24] is a modular pre-assured safety concept. ConSerts are engineered during the safety development lifecycle as models of reconfiguration, allowing safety-related properties to be guaranteed under variable systemic and environmental conditions. Generalizing ConSerts to be applied within BIECO, we consider how safety relates to the more general concept of dependability,

which simultaneously encompasses security. In doing so, we can specify non-functional requirements related to safety-related security requirements, which may be dynamically evaluated at the time of system composition rather than the time of specification.

ConSerts can be engineered during the systems development process (BIECO deliverable D6.4 [14] provides a more detailed discussion). This engineering process results in a ConSerts model, which can be exported as a blueprint in a specified file format (based on YAML⁸).

ConSerts considers two sources of information: the level of quality available from demanded external services and evidence collected internally by the system to determine the quality at which a given service can be provided. Such 'Runtime Evidence' (RtE) can be acquired during system operation and trigger updates to the quality of the provided service. What kind of information is relevant as RtE depends on the safety (or dependability, e.g., security) concept upon which the ConSert is specified.

ConSerts is flexible in incorporating an open domain of runtime information to ascertain the operational context and recommend a system adaptation. Inside BIECO, the following possibilities have been considered:

- Exploiting the vulnerability detection, forecasting, and propagation data collected inside the Work Package 3. In this case, ConSerts can use the Work Package 3 models to determine the services' acceptable levels of quality and (or) the vulnerability risks to conceive service reconfiguration or service halt if necessary.
- Exploiting the failure prediction methods developed inside Work Package 4 to anticipate subsystem/component failures and trigger ConSert re-evaluation (reconfiguration).

On the technical level, using the approach already described for the extended MUD file (see the previous section), the ConSerts blueprints can be integrated into the ontology data. They can provide either the composition and adaptation checking mechanisms to be used in case of violations or the countermeasures to be activated to force the SUA or CE reconfiguration functions.

⁸ <https://yaml.org/>

3. Auditing Framework Implementation

The implementation of the Auditing Framework architecture presented in Section 2 and Figure 1 has been provided as a virtual framework component. Therefore, the Runtime Monitoring, Ontology Manager, Predictive Simulation, and the Auditing Framework GUI communicate through the Auditing Framework Message BUS for transferring data and managing the process phases.

In the following, Section 3.1 presents the overall implementation of the Auditing Framework and the interaction with the Runtime Phase. Sections 3.2, 3.3, 3.4, and 3.5 detail the implementation of the Runtime Monitoring, the Predictive Simulation, the Ontology Manager, and the Auditing Framework GUI, respectively. Each section has almost the same structure i) first, it introduces the internal components and their implementation details; ii) then, it presents the (intra- and inter-) communication details and the exposed interfaces; iii) successively, the section details the (input and output) data structures, iv) and finally it focuses on the technologies used and the installation guidelines.

3.1. Auditing Framework Implementation Details

To satisfy the Section 1.1 challenges and, in particular, “CH2: *Separating properties predictions and assessment*”, the Auditing Framework components are not strictly connected. This also provides the possibility to either deploy them on different machines or clouds; or replace them with more performant components or services.

The virtual framework has been merged within a typical docker composition using the technologies that docker-compose described in a *ym/* file. Specifically, the implementation of the Predictive Simulation was supported by the concept of the triple modular framework as detailed in deliverable D2.4 [1].

3.1.1. Communication Flows Intra-Nodes/Components

The entry point for the virtual component is the messageBus. Every component in the virtual framework exposes its REST interface for communicating directly with the BIECO platform.

3.1.2. Technologies Used

The technologies used are

- Docker for packaging the framework.
- Apache ActiveMQ or Apache Artemis, or Mosquitto for the communication bus.

3.1.3. Licenses

No licenses are needed.

3.1.4. GitHub Details Repository

Auditing Framework is exposed as a single docker artifact using docker-compose.

3.2. Runtime Monitoring

The Runtime Monitoring component has been developed to target the challenges mentioned in Section 1.1 and in particular: *CH1: Whitening the black-box assessment process; CH2: Separating properties predictions and assessment; CH3: Leveraging the existing monitoring solutions.*

The Runtime Monitoring splits functionalities into independent modules that can be enacted according to the desired configuration/operational profile. The components can also be instantiated as stand-alone nodes or containers on top of a Docker architecture.

This choice provides a more substantial decoupling and lets the components deploy on different machines for improving resources management. In particular, the Complex Event Processor (CEP), the core part of the Runtime Monitoring, can be deployed in multiple instances on several nodes.

Development relies on Java technologies (OpenJDK⁹ 16 and 17) for the core functionalities and Drools¹⁰ for the principal Complex Event Processor.

The project and the dependencies between libraries are managed: for the JMS through Maven¹¹, i.e., the message broker embedded into the system is ActiveMQ¹² (an open-source multi-protocol Java-based message broker); for the interface that exposes the interaction with REST interfaces, through Mosquitto¹³. More details about technologies are in Section 3.2.6.

3.2.1. Internal Artifact's Structure

The core modules of the Runtime Monitoring are depicted in Figure 5 and described hereafter.

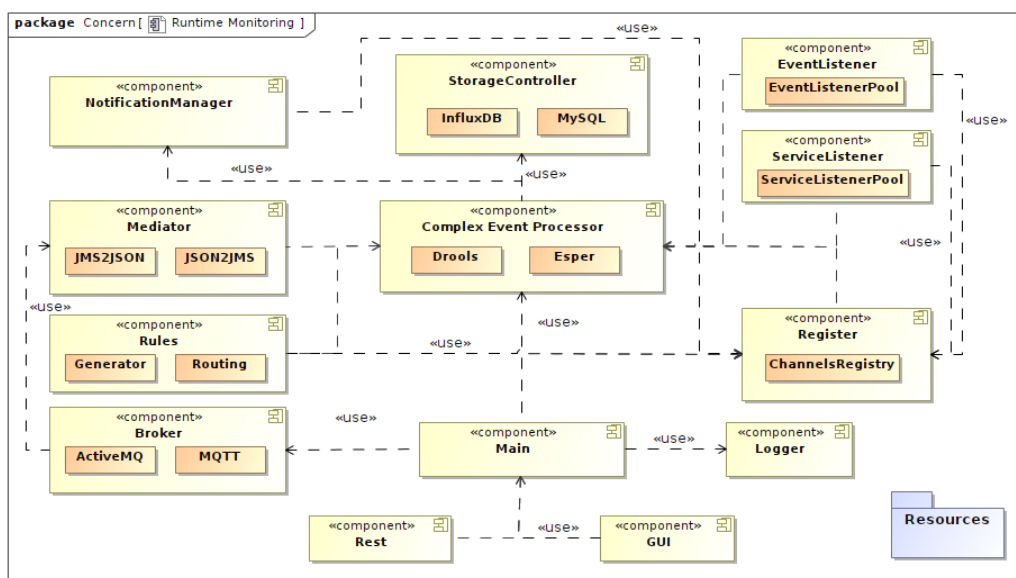


Figure 5: Runtime Monitoring Components

⁹ <https://openjdk.org/>

¹⁰ <https://www.drools.org/>

¹¹ <https://maven.apache.org/>

¹² <https://activemq.apache.org/>

¹³ <https://mosquitto.org/>

3.2.1.1. Main

It is the launcher of the architecture; it oversees executing the components required for a specific profile in the correct order. The Runtime Monitoring can be performed with or without a Rest component. This allows managing the Runtime Monitoring lifecycle according to what is required to interact with the BIECO platform.

3.2.1.2. Rest

This component will start a Glassfish Grizzly Server¹⁴ that exposes:

- a basic web page for publishing the status of the Runtime Monitoring.
- a set of POST methods for interacting within the Runtime Monitoring according to the lifecycle specified inside the BIECO Project and reported in Figure 6.

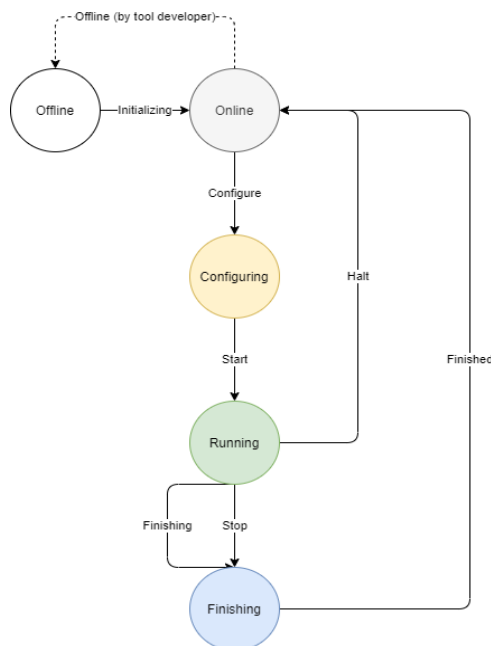


Figure 6: Service Status from the BIECO project

Moreover, the **Rest** component provides facilities to authenticate the messages sent or received by the REST interface. According to the agreed lifecycle, the developed authentication mechanism is based on two tokens (one for incoming and one for outgoing messages).

Figure 7 shows the web interface, the logs, and the related activities.

¹⁴ <https://javaee.github.io/grizzly/>

```

Runtime Monitoring
Status: Running
Monitoring logs:
-----
2022-06-23T12:13:32.900+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.ConcernApp -
- NEW MONITORING SESSION - ALL COMPONENTS RESTARTED -
-----

2022-06-23T12:13:32.902+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.ConcernApp - Starting components
2022-06-23T12:13:32.909+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Start SSL Broker
2022-06-23T12:13:32.925+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Creating TrasportConnector
2022-06-23T12:13:33.134+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Waiting for broker start
2022-06-23T12:13:33.134+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - broker started
2022-06-23T12:13:33.134+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Enabling SSL ConnectionFactory
2022-06-23T12:13:33.186+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Connection successfully created
2022-06-23T12:13:33.186+0200 [grizzly-http-server-1] DEBUG it.cnr.isti.labsedc.concern.broker.ActiveMQBrokerManager - Component ActiveMQBrokerManager loa
2022-06-23T12:13:33.190+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.storage.MySQLStorageController - Setting up storage with default p
2022-06-23T12:13:33.580+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.storage.MySQLStorageController - Connected successfully to concern
2022-06-23T12:13:33.794+0200 [grizzly-http-server-1] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - CEP creation
2022-06-23T12:13:33.819+0200 [Thread-5] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne creates a lis
2022-06-23T12:13:34.120+0200 [Thread-5] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne created Sessi
/home/acalabro/workspace/ConcernMonitoringRest/src/main/resources/startupRule.drl with knowledgePackages: [[Package name=it.cnr.isti.labsedc.concern.ever

```

Figure 7: Service Status from the BIECO project

3.2.1.3. Logger

It provides common logging facilities across the Runtime Monitoring components. Logging is implemented using Log4j¹⁵.

3.2.1.4. Broker

The component executes an internal instance of a message broker and provides facilities for sending messages (events and requests) between the artifacts involved in a monitoring session. The execution of this component is optional. Depending on the technology used for the communication, two solutions can be supported (also at the same time):

- 1) Using a configuration that runs an ActiveMQ instance. In this case, the JMS Message Broker is exposed on port 61616. The instance execution is managed following the pattern Singleton, and the artifact is included using maven dependencies in the Project Object Model (POM) of the project.
- 2) Using Mosquitto, i.e., an open-source message broker that implements MQTT protocol, allows light transmission between low-power devices using publish/subscribe messaging techniques.

3.2.1.5. Register

This component traces and stores all the information related to the connections between the internal Runtime Monitoring components and external artifacts involved in a monitoring session. For instance, the **Register** collects the list of the Complex Event Processor instance executed, the channel used for listening for events, and the parameters they use. This information is collected using the object *TopicAndProperties* as detailed in Figure 8.

As in the figure, the class provides the information needed for: connecting to a specific channel; routing a message or a request to the channel on which a dedicated Complex Event Processor instance is running.

¹⁵ <https://logging.apache.org/log4j/2.x/>

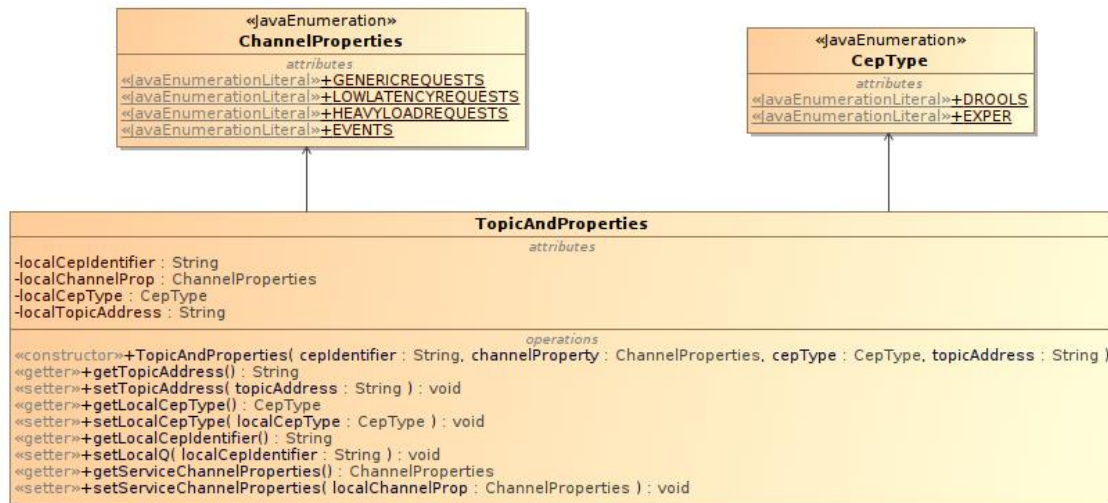


Figure 8: Class diagram of the connections Register

3.2.1.6. EventListener and ServiceListener

EventListener and ServiceListener components automatically execute the routing. Both the components execute a thread poll. Each thread listens on a specific channel: the event messages generated by probes (EventListener Task); the requests for the evaluation of one or more rules on (generic or specific) Complex Event Processor (ServiceListener Task).

These two components guarantee the scalability of the Runtime Monitoring: if necessary, generating more channels or more complex event processor instances for specific scopes can be generated.

3.2.1.7. Complex Event Processor

The Complex Event Processor manages the automatic rule execution and assessment. It can be executed considering two different engines in parallel: Drools¹⁶ or Esper¹⁷.

In particular, Drools is an engine based on the RETE algorithm, based on analysis executed on forward and backward chains. It can be instructed through a specific Domain-Rule-Language. Drools guarantee high speed and scalability, thanks to the rule engine engendered by the RETE algorithm (RETE-OO).

Esper uses the Event Processing Language rule language that implements and extends the SQL-standard language. Usually, the Esper engine has more performant memory management than Drools, which relies on a garbage collector that is not performant in case of millions of events.

3.2.1.8. Rules

This component enacts either the rules generation or the self-injection process when meta-rules are invoked during the Runtime Monitoring execution.

¹⁶ <https://www.drools.org/>

¹⁷ <https://www.espertech.com/esper/>

The Rules component is also in charge of routing the messages that contain rules to the target Complex Event Processor. It uses CepType, the parameter stored on the event, and the analysis of the ActiveCep list stored in the Register.

3.2.1.9. Mediator

This component guarantees the interoperability between MQTT messages (mostly sent using JSON - JavaScript Object Notation) and the JMS Messages.

The Mediator uses messages from/to two kinds of brokers: ActiveMQ and Mosquitto. This component executes two main functions:

1. converting a JSON message into a JMS Message and injecting it into one of the dedicated channels.
2. converting a JMS message into JSON when a notification needs to be sent to an entity that does not support the JMS message paradigm.

The high-level behaviour is described in Figure 9.

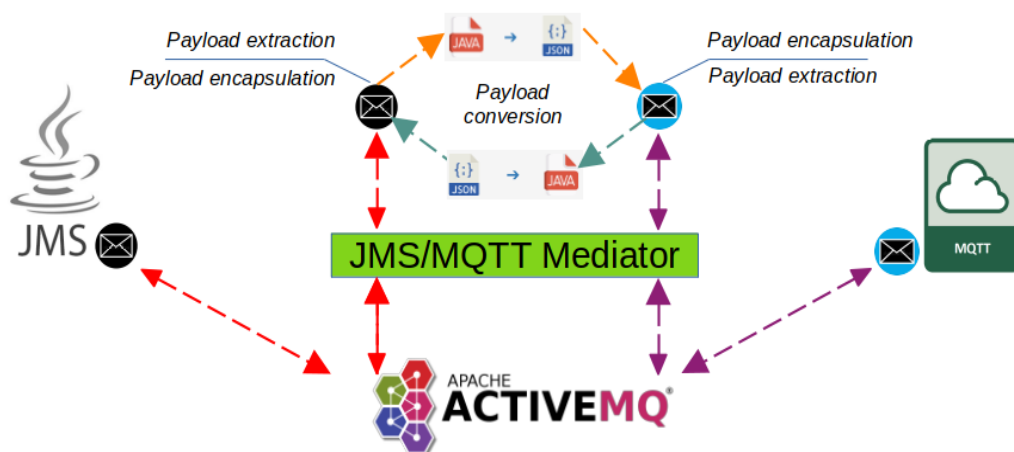


Figure 9: JSON2MQTT Mediator behavior

3.2.1.10. StorageController

This component manages to store all the events and data generated by the Runtime Monitoring. The StorageController allows (simultaneously) two implementations:

- 1) using a generic MySQL¹⁸ server deployed outside Runtime Monitor to store simple events.
- 2) Using a database structured for time series, such as InfluxDB¹⁹, allows for managing vast amounts of events faster.

¹⁸ <https://www.mysql.com>

¹⁹ <https://www.influxdata.com/>

3.2.1.11. Notification Manager

This component manages the notification of failure sent by the Complex Event Processor. It forwards the failure notification to the specific channel gathering the correct information (channels details) from the ChannelRegistry component.

3.2.1.12. Execution Flow

This section details the basic start-up procedure to clarify the communications between the Runtime Monitoring artifacts.

Startup: In particular, when the **Main** component is invoked, it checks if the parameters related to the execution of the **Rest** component, *ActiveMQ*, and *Mosquitto* broker are set to true or false. Indeed, according to the parameters set, the Runtime Monitoring Engine can be executed according to a specific operation profile.

If the **Rest** component is invoked, a GrizzlyServer is run so that the REST interface for invoking the Runtime Monitoring by external components is exposed.

At this point, the **Main** component invokes the execution of one (or more) **Broker** for the communication according to the specific message paradigm. The first broker is *ActiveMQ* for local instance execution, and the second is *Mosquitto*. This can be executed locally within the Runtime Monitoring to raise the backbone on which messages can flow.

Successively, the **Register** component can start-up, and the *ChannelRegistry* be created to enable the storage of data related to components, Complex Event Processor (CEP), and the created channels.

In this stage, an instance of the **StorageController** component is created as a data structure on which the received events are stored. This is a connection to a *MySQL* server or *InfluxDB* and includes a set of internal interfaces for storing and reading data for the other components.

At this point, the channels on which the events flow are created. Thus, facilities have been developed for managing the amount of data that this infrastructure may receive and the possibility to categorize data among them: i.e., address it to a specific CEP, manage load balancing between multiple instances of Complex Event Processor, or simply separate notification from requests or simple events.

The **EventListener** and the **ServiceListener** components expose a pool of threads. Each thread generated by those components is opening a channel listening for a specific kind of event. The **EventListener** will receive and manage events related to the execution of a System Under Audit (events generated by probes); the **ServiceListener** will generate channels dedicated to the evaluation requests (loading of rules by an external component).

Following the same paradigm, another component has been developed to avoid confusion between Framework and notifications: the **NotificationManager**, in charge of dispatching the notification sent from the CEP to the correct recipient/channel.

The last component that can be executed in single or multiple instances is the **ComplexEventProcessor**. The instances can rely on *Drools* engine or *Esper* engine. The

information about the channel used for CEPs execution is stored in the **Register** component.

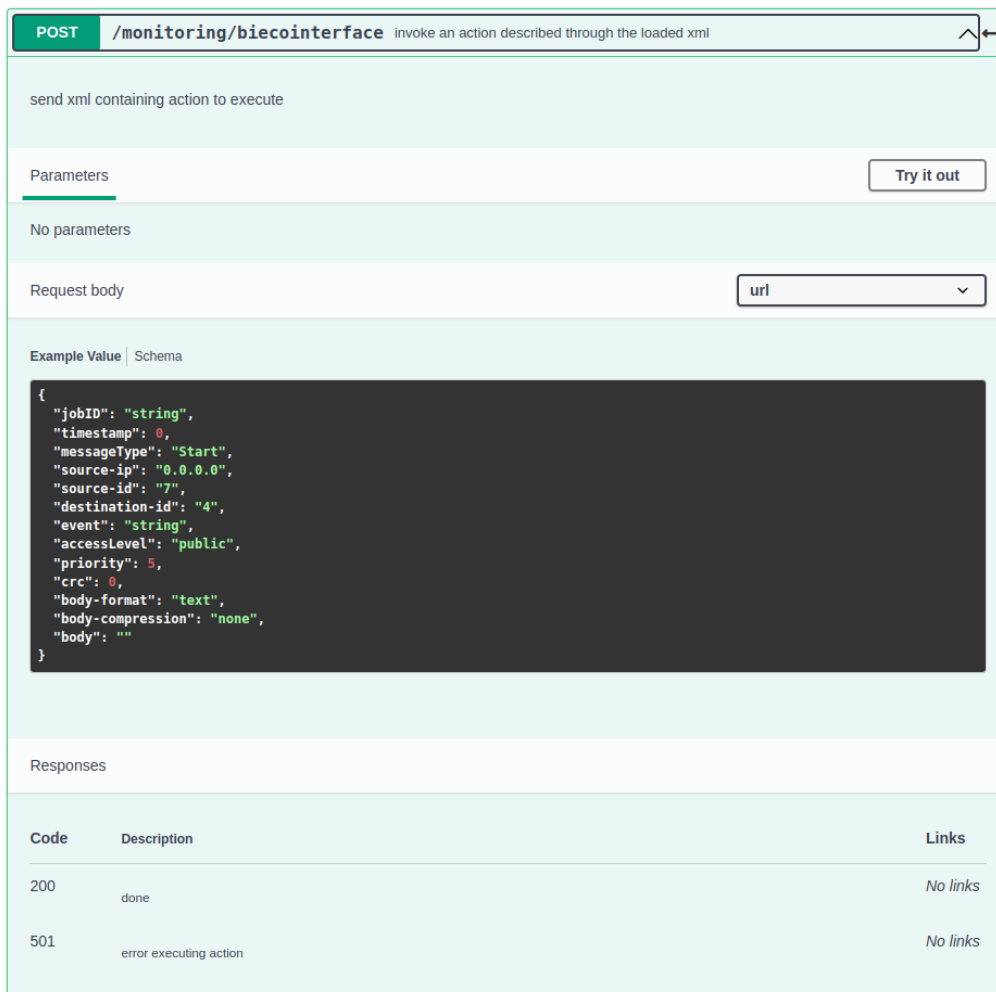
Runtime Monitoring is ready to receive events or evaluation requests on the exposed channels.

3.2.2. Communication Flows Inter Artifacts

Within the Runtime Monitoring, the components exchange data related to the events notified by external entities or rules: received or generated during the execution. The database can be instantiated within the Runtime Monitoring or can be an external entity. It receives data managed by the StorageController component, converts the execution to information, and stores it.

3.2.3. Exposed Interfaces

Runtime Monitoring exposes different interfaces: JMS, REST, and MQTT.



POST /monitoring/biecointerface invoke an action described through the loaded xml

send xml containing action to execute

Parameters Try it out

No parameters

Request body url

Example Value | Schema

```

{
  "jobID": "string",
  "timestamp": 0,
  "messageType": "Start",
  "source-ip": "0.0.0.0",
  "source-id": "7",
  "destination-id": "4",
  "event": "string",
  "accessLevel": "public",
  "priority": 5,
  "crc": 0,
  "body-format": "text",
  "body-compression": "none",
  "body": ""
}

```

Responses

Code	Description	Links
200	done	No links
501	error executing action	No links

Figure 10: Elements of the exposed REST interface

- REST interfaces are provided for interacting with the BIECO Platform to manage the lifecycle of the Runtime Monitoring node. An example is reported in Figure 10. More details are provided at the following link: https://app.swaggerhub.com/apis/acalabro/Auditing_Framework/1.0.2.
- MQTT interfaces expose an open channel on which the JSON messages can be sent according to a schema that reflects the Event message described in Section 3.2.4. An example of a message is reported in Figure 11.
- JMS interface is exposed using ActiveMQ Broker. It is a set of Topics and channels on which asynchronous messages can be sent to the Runtime Monitoring (see Section 3.2.4).

```

1 {"senderID":"MobileApp","dataParameter":"","data":"3",
2 "anotherParamater":2,"name":"8c:8d:ab:10:40:bd",
3 "checksum":"none","sessionID":"idOfTheSession",
4 "destinationID":"monitoring","cepType":{"DROOLS},"timestamp":1629356448048}

```

Figure 11 Elements of the exposed REST interface

3.2.4. Exchanged Data Structure

Runtime Monitoring is based on event messages. The payload of those messages contains valuable information for identifying the event that occurred in the system under audit and executing the complex event processing operation against rules and properties to be verified.

An event is an immutable statement of fact, reporting something that happened in the system under auditing. Probes generate events and mark them with a timestamp related to the last occurrence (see Section 3.2.6.1). Figure 12 provides the Runtime Monitor event customization inside BIECO Project Architecture.

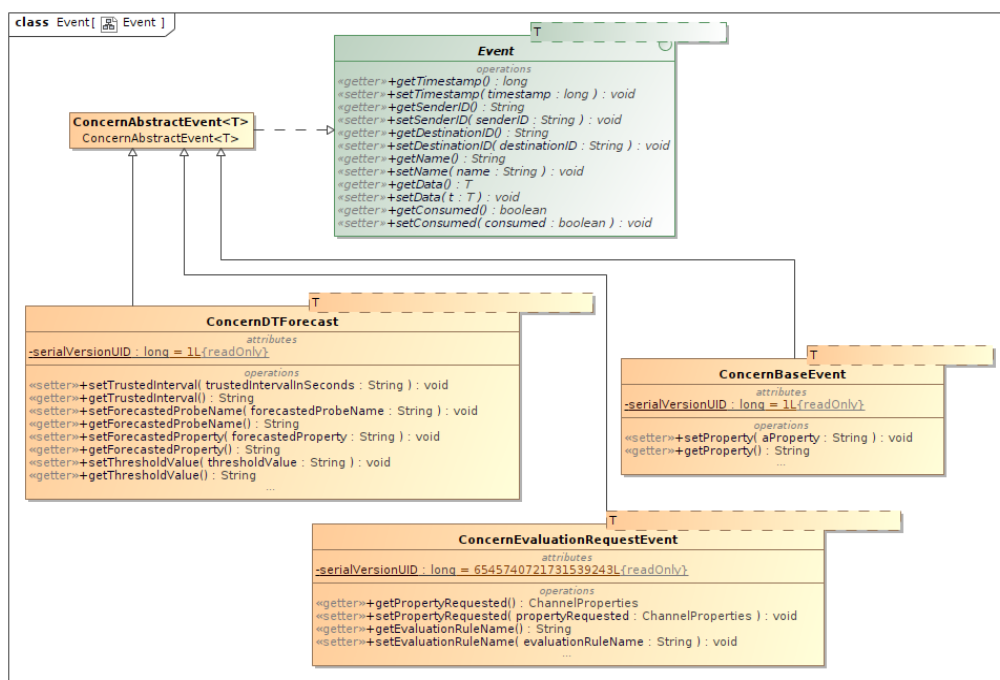


Figure 12 Class diagram of Event Structure

As in the Figure, the main parameters of the `Event<T>` interface class are:

- *Timestamp* refers to the time an event has been generated (when it occurs in the system under audit).
- *SenderID*, the identification of the entity generating and sending the event.
- *DestinationID*, the identifier of the CEP that should manage this event.
- *Name* is a String that contains the name or parameter name the event is referring to.
- *Data<T>*, a variable type that contains the value of the event that occurred; for example, an event with *Name* = "Temperature" and *Data* = 35.0f.
- *Consumed*, a Boolean value that indicates if the event has been already managed or not. The CEP can use this parameter to mark an event already managed (in case it remains inside the event cloud or stream).

As shown in Figure 12, the generic implementation of the interface `Event<T>` interface is proposed in the abstract class `ConcernAbstractEvent<T>`.

An extension of this class is represented by the `ConcernBaseEvent<T>` that includes an extra parameter called *Property*.

Using the same pattern, two specific types of events are generated for the correct analysis of events generated in BIECO.

The first is the `ConcernEvaluationRequest<T>` class. It contains the basic structure of the `ConcernAbstractEvent<T>` extended with two specific parameters: *PropertyRequested* and *EvaluationRuleName*.

The *PropertyRequested* object refers to the message properties as detailed in Section 3.2.1.5. The rule can be routed to the correct Complex Event Processor using this parameter.

The `ConcernDTForecast<T>` is also an extension of `ConcernAbstractEvent<T>`. It contains the forecast generated by the Digital Twin that will enact the self-rule generation process.

This object contains extra fields:

- *TrustedInterval*: the amount of time-related to the validity of the forecast.
- *ForecastedProbeName*: the name of the Probe in the SUA to which the forecasted property is referring.
- *ForecastedProperty*: the property forecasted (ex: latency, connectivity).
- *ThresholdValue*: a threshold value used, for example, in case the property is related to latency or a scalar value.

3.2.5. Produced Data

The output data are represented by:

- Events, in terms of storing the data flowing into the Monitoring Engine for being analysed.
- Complex Events are events created by the Monitoring Engine by correlating simple Events according to the rules loaded into the CEP.
- Notifications/Violations that represent the action executed after the triggering of

3.2.6. Technologies Used

The software has been developed in Java, version 17, using OpenJDK.

The project is available on GitHub, and dependencies are solved through the Maven framework.

The communications are managed using JMS or JSON messages transmitted on top of ActiveMQ or Mosquitto broker.

The interfaces exposed are realized using REST running on Grizzly Server.

Storage is realized using MySQL Server or InfluxDB time-series database.

The first version of the Complex Event Processor engine has been developed using Drools. Another version is ongoing and relies on the Esper engine.

The overall Runtime Monitoring is going to be dockerized for the second release of the overall platform.

3.2.6.1. Probes

As described in Deliverable D5.1 [11], tools for instrumenting code are currently available. Details related to the instrumentation procedures and probes injection are reported in Appendix A.

3.2.7. User Installation Guidelines

- 1) svn checkout <https://github.com/acalabro/ConcernMonitoringRest.git>
- 2) import the project into Eclipse
- 3) Create a configuration that executes as the main class:
 - a) the `it.cnr.isti.labsedc.concern.rest.Main` class OR
 - b) the `it.cnr.isti.labsedc.concern.ConcernApP`
- 4) export the project as a Runnable Jar File and select "Extract required libraries into the generated JAR."
- 5) execute the exported jar simply by running "java -jar exportedjarfile.jar."

3.2.7.1. Hw/Sw Requirements

For being executed, as a Jar file, the Runtime Monitoring requires:

- OpenJDK Runtime 17.
- no restriction on port 61616 for ActiveMQ.
- no restriction on port 8181 (if the Rest component is needed).
- no restriction on port 1883 for Mosquitto (if needed).

Required libraries have been included within the jar file.

3.2.7.2. Licenses

The code developed for the Runtime Monitoring has been released following GPL3²⁰.

²⁰ <https://www.gnu.org/licenses/gpl-3.0.html>

The libraries involved (listed below) are open source and will follow their respective licensing:

- drools-*
- javax.jms-api.
- activemq-broker.
- mysql-connector-java.
- log4j-core.
- org.eclipse.paho.client.mqttv3.
- jersey-container-grizzly2-http.
- any other minor library included within the pom.

3.2.7.3. Github and DockerHub Repository

The monitoring project is available on Github at the following link:
<https://github.com/acalabro/ConcernMonitoringRest>

3.3. Predictive Simulation

This section details the Predictive Simulation internal structure, its artifacts, and implementation details.

3.3.1. Internal Artifacts Structure

Figure 13 depicts the internal structure of the Predictive Simulation components. The Predictive Simulation consists of jar files executed within the BIECO framework that works on a Publish Subscriber basis. Specifically, it contains an *amq Consumer* that consumes real-time data received via a specialized message defined for the Auditing Framework.

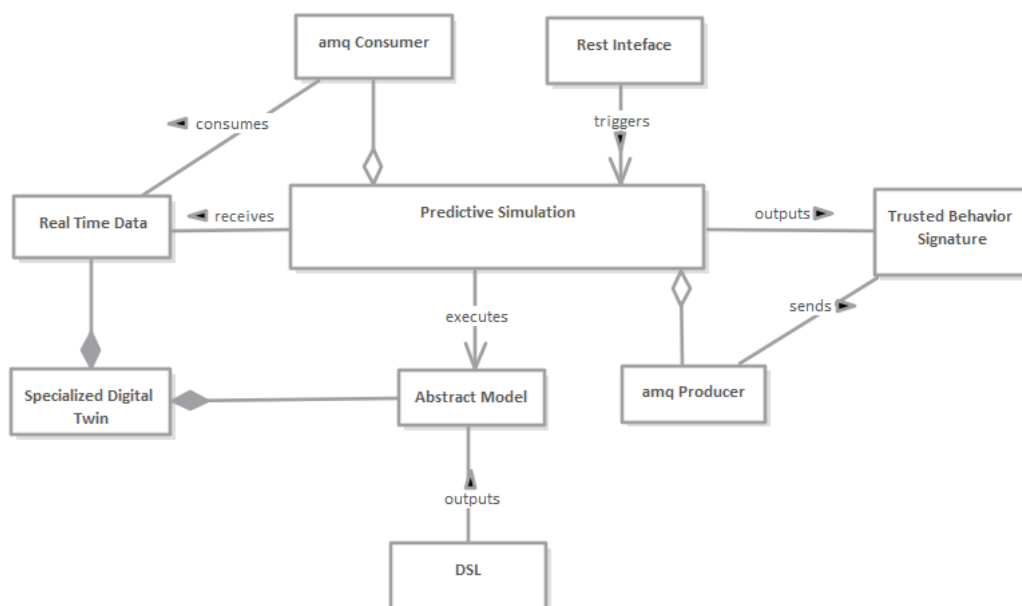


Figure 13 Internal structure of the Predictive Simulation components

The Predictive Simulation module also contains an *amq Producer* that sends the Trusted behaviour Signature back on the auditing bus.

The triggering of the simulation is performed by receiving specialized commands from the REST interface. The models that the Predictive Simulation executes are abstract models which are outputted by the Domain Specific Language (DSL). When fed with Real-Time Data, these abstract models become specialized Digital Twins.

3.3.2. Communication Flows Inter Artifacts

The figure below depicts the flow between the internal artefacts of Predictive Simulation: The Rest Interface that connects the Predictive Simulation and the BIECO Orchestrator triggers the execution of the jars that incorporate the abstract models and the *amq Consumer* that enables the collection of the real-time data. Once these two parts are put together (abstract models and real-time data), specialized Digital Twins are created and executed. For sending the resulting Trusted Behaviour Signature to the auditing bus, the Predictive Simulation triggers the execution of an *amq Producer*.

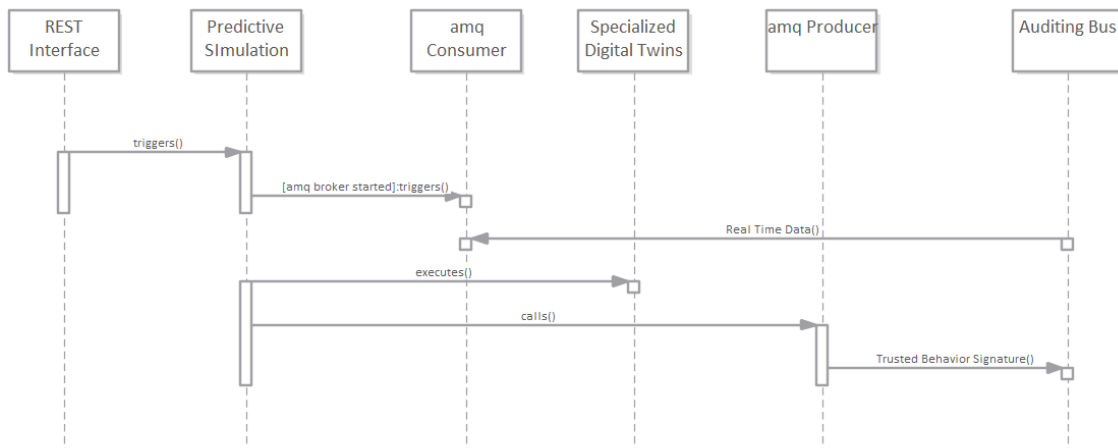


Figure 14 Flow between the internal artefacts of the Predictive Simulation

3.3.3. Exposed Interfaces

The Predictive Simulation provides a rest interface to communicate with the BIECO orchestrator. The interface is a stand-alone application implemented in Java. Its goals are to receive BIECO messages and to process them for the Predictive Simulation. The interface manages especially the control and monitoring messages. The Predictive Simulation is another standalone application, which is started, stopped, and halted by the rest interface.

The structure of the Predictive Simulation Rest Interface is shown in Figure 15. The rest interface component allows a decoupling of the Predictive Simulation from the orchestrator to allow reuse of the Predictive Simulation in other projects and increase the maintainability. The component is structured into the four sub-components: ApplicationController, OrchestratorSendMessage, AppRegistry, and PSBridge. The first three sub-components are generic for each BIECO application and require adaption to the

specific application. The sub-component PSBridge is introduced to consider the specific requirements for the Predictive Simulation.

The first sub-component is the AppController, which provides the rest interface and maps HTTP post requests on the sub-URL “/BIECOinterface” to the function processMessage. Each HTTP post message on this interface is checked for the correct token. The HTTP post requests are answered either by “401 Unauthorized” or “200 OK”. A reaction to the BIECO message content is not foreseen, as separating the HTTP rest interface layer with the technical implementation and the BIECO messaging layer is wanted. In the case of authorized BIECO messages, the AppController processes the messages based on their type. The possible types and reactions are:

- GETSTATUS: the state of the AppRegistryService is sent, which is BIECOToolStatuses.ONLINE
- HEARTBEAT: the tool id is set by the heartbeat
- CONFIGURE: no specific configurations for the Predictive Simulation are foreseen.
- DATA, EVENT: an infrastructure with a queue to provide messages to Predictive Simulation is created. The DATA/ EVENT message is added to a queue of the PSBridge class.
- START, STOP, HALT: The respective PSBridge class functions are called.

The AppController uses the class AppRegistry to store the state of the communication. The attributes orchestratorURL, orchestratorToken, and the token of the Predictive Simulation are statically defined in the program code of the class AppRegistry.

As the AppController also sends the response messages to the Orchestrator, it uses the class OrchestratorSendMessage to encapsulate the communication. This encompasses transforming messages to JSON objects, calculating the Cyclic Redundancy Check, and sending the respective HTTP post request.

The PSBridge sub-component processes the START, STOP and HALT messages. The START message causes the start of the process by using the JAR file of the Predictive Simulation. A STOP message is interpreted to use the Java process interface to call the destroy () function. After termination of the process, a FINISHED message is sent to Orchestrator. In contrast, the HALT message calls the function destroyForcibly(). Furthermore, the PSBridge class provides a queue of DATA and EVENT messages, which the Predictive Simulation may access.

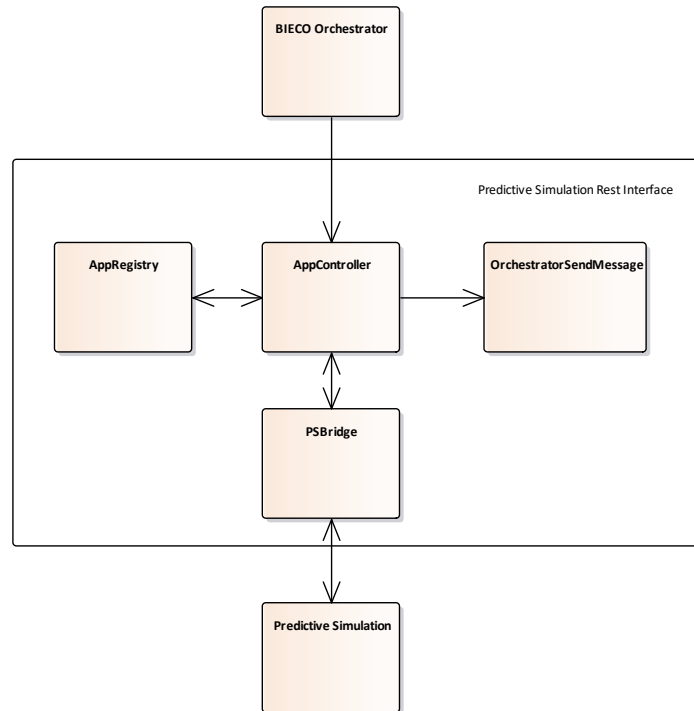


Figure 15 Structure of Predictive Simulation Rest Interface

3.3.4. Exchanged Data Structure

The structure of the data sent to the auditing bus is exemplified below and consists of:

- 1) Definition of the timing interval of the behaviour for which the prediction is being performed "Trusted Interval."
- 2) Definition of the timing interval in which the behaviour is expected: Prediction Window
- 3) Trusted order of events
- 4) Trusted type of events

```
{
  "Trusted Interval": "1s",
  "Prediction Window": "5s",
  "Trusted Order": {
    "1": "Velocity Command",
    "2": "Velocity Command",
    "3": "Velocity Command",
    "4": "Score Event",
    "5": "Velocity Command",
    "6": "Velocity Command"
  },
  "Trusted Type": {
    "Trusted type": "Score Event"
  }
}
```

3.3.5. Produces data

One data point within the signature contains information about the behavior interval for prediction, the window of prediction, and the Trusted Behavior Signature having the order, the name, and the type of events.

Trusted Interval: in terms of seconds, for example: 1s

Prediction Window: in terms of seconds, for example: 5s,

Trusted Order: of events, including the number of events, for example:

1 Velocity Command, 4 Score Events, 5 Velocity Commands

Trusted Type: Score Event, Velocity Command

3.3.6. Technologies Used

Java and RESTful Web Service have been used to define the REST interface.

The definition of the abstract model has been performed using the Xtext framework²¹ (as presented in D5.1 [11]). For the implementation of the producer and the consumer, amq has been used.

3.3.7. User Installation Guidelines

The .jar for the Predictive Simulation connection to the BIECO orchestrator is already provided. For keeping a loose coupling of artifacts and enabling their exchangeability, this .jar calls another jar that contains the definition of the abstract models and the internal amq producer and consumer. The current version of the DSL holds the definition of an entity as a list of entities and a list of features. Consequently, we require that all entities need to be declared before dealing with the features of the entity.

For creating Models of the twins using the DSL4Twins (see deliverable D4.2 [10]), the following configuration is necessary:

- Xtext v2.24
- Junit test 5
- Java 1.8.0
- And IDE of your preference (for example, Eclipse)

3.3.7.1. Hw/Sw Requirements

Java-compliant machine, internet connection

3.3.7.2. Licenses

Under BIECO licensing.

3.4. Ontology Manager

Ontology Manager is the component responsible for managing and supporting the implementation and the BIECO Ontology. It has been introduced in deliverable D5.1 [11] and refined and specialized as described in Section 4. The implementation of the Ontology Manager targets the challenges *CH2: Separating properties predictions and assessment* and *CH3: Leveraging the existing monitoring solutions* presented in Section 1.1.

Thus, its architecture is voluntarily conceived as abstract as possible to instantiate its components with available tools. In the following, the components of Ontology Manager are introduced.

²¹ <https://www.eclipse.org/Xtext/>

3.4.1. Internal Artifacts Structure

Ontology Manager is composed of different components collaborating to achieve a common goal, i.e., providing functionalities and means to share information and knowledge about SoS and Ecosystem within BIECO.

Figure 16 illustrates the updated supporting architecture, an enhancement of the one introduced in Deliverable D.5.1 [11], in which two new components (Ontology Mapper and Ontology Population) have been introduced. They allow the management of the BIECO Blueprints and the Data Provider's data. Unlike the previous proposal, the Visualization Component has been removed as an internal component and substituted with the external Auditing Framework GUI (described in Section 3.5). It allows the Ontology Final User to interact with the Ontology Manager for ontology navigation and management. In the following, more details about each component are provided.

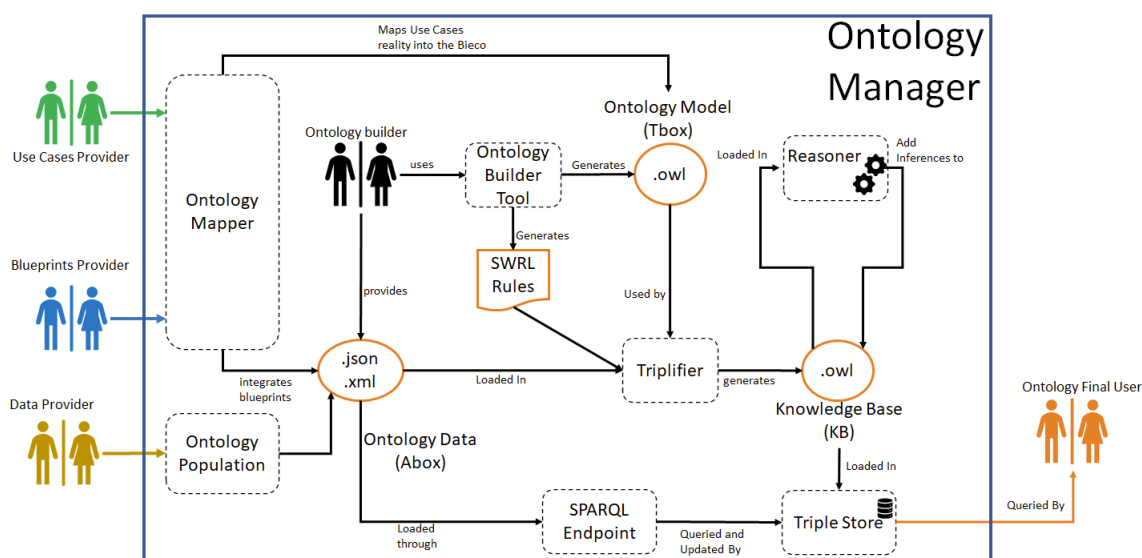


Figure 16 Ontology Manager Reference Architecture

3.4.1.1. Ontology Mapper

Ontology Mapper manages the BIECO blueprints contributing to the connection between the Design and Runtime Phases. Ontology Mapper has the following responsibilities:

- It lets the integration of the blueprints that are Extended MUD File and ConSerts data with the reference context.
- It lets the specification of ontology entities and presentation of specific use case domain. Figure 16 shows that the Use Cases provider is the primary target end-user group at this stage.

3.4.1.2. Ontology Population

It allows the manual definition of values for ontology population by a Data Provider. It is composed of a dedicated user interface that lets definition and the insertion of all the required individuals for populating the ontology. It is also possible to upload the JSON

file directly according to the format defined in Section 3.4.3, which contains the data to be transmitted to the Triplifier component.

3.4.1.3. Ontology Tool Builder

Ontology Tool Builder is used for creating, modifying, and visualizing the ontology according to the representation detailed in Section 4. In the BIECO Project, the Ontology Tool Builder is instantiated with Protégé because it provides a friendly Graphical User Interface (GUI) for the definition of ontologies; it can be adapted to build even complex ontology-based applications thanks to its modular architecture. The BIECO customization of the Ontology Tool Builder can be used offline as a standalone solution and online as a web-based solution called WebProtégé²². This lets a more dynamic sharing of ontologies for collaborative viewing and editing. Figure 17 shows a screenshot of the adopted WebProtégé.

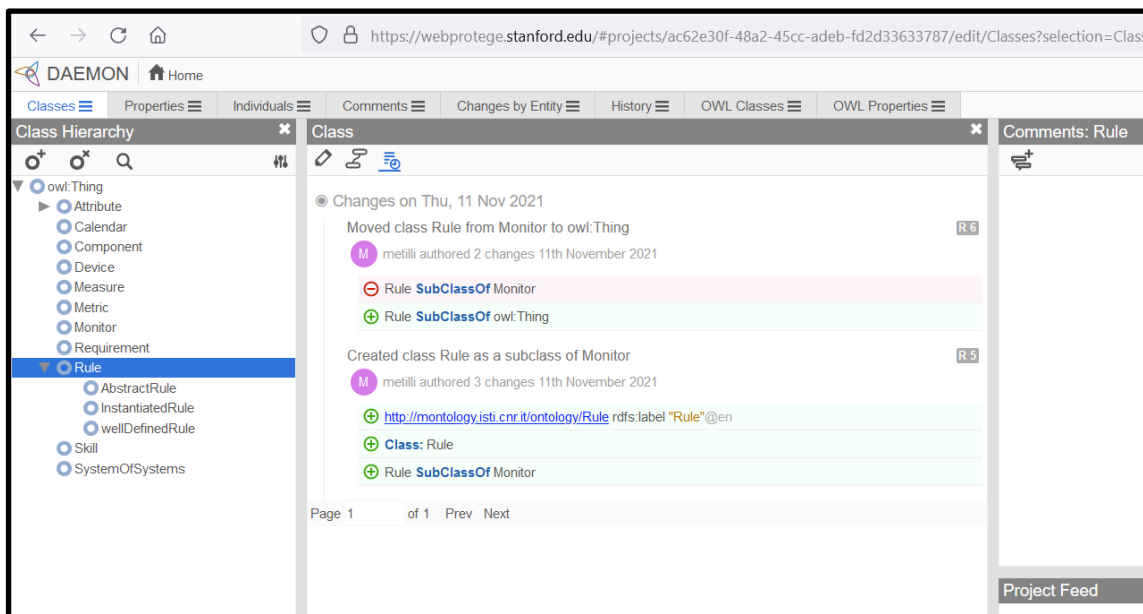


Figure 17 Screenshot of WebProtégé during the development of BIECO Ontology

3.4.1.4. Triplifier

It is a triplifier based on OWL (Web Ontology Language)²³ developed in Java. Triplifier takes the ontology data as input, consisting of the individuals, and it is specified in JSON or XML format. It also takes the rules the Ontology Builder Tool defines as input to allow the reasoning and inference of new knowledge.

²² <https://webprotege.stanford.edu/>

²³ <https://www.w3.org/OWL/>

3.4.1.5. Semantic Reasoner

Semantic Reasoner is used to inferring new knowledge: the reasoning is performed for consistency checks and definition of inferences. In the BIECO Project, the implementation of the Semantic Reasoner relies on OpenIlet because:

- it is Java-based that can be easily integrated with OWL API.
- is an open-source software actively maintained, providing functionality to check the consistency of ontologies, among other functionalities.

3.4.1.6. Triple Store

In the BIECO Project, GraphDB has been selected as the reference triple store, a free-to-use graph database and knowledge discovery tool compliant with RDF and SPARQL and available as a high-availability cluster. Technically the Auditing Framework GUI interacts with the Ontology Manager by employing well-defined SPARQL queries.

3.4.1.7. SPARQL Endpoint

This component allows specifying and executing specific SPARQL queries to retrieve knowledge from the triple store and dynamically update the KB content. In Section 3.4.3, details about both Triple Store and SPARQL Endpoint are also provided from the behavioural point of view.

3.4.2. Communication Flows and Exposed Interfaces

The proposed framework to manage the communication between the Auditing Framework GUI and the Ontology Manager component is shown in Figure 18. It provides services for querying/interacting with the ontology. The Auditing Framework GUI uses the provided information during the user interaction. The server handles GET, POST, PUT and DELETE requests, which are addressed with SELECT/CONSTRUCT, INSERT, UPDATE and DELETE SPARQL queries, respectively.

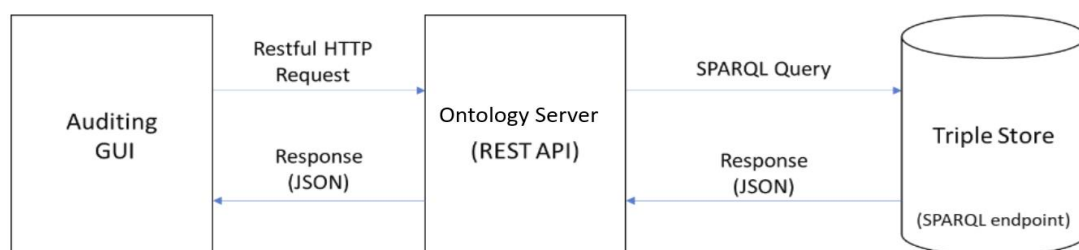


Figure 18 Overview of Ontology Server RESTful Interface

The model for the specification of the available endpoints follows the OpenAPI Specification (OAS)²⁴. Figure 19 shows some examples of the proposed endpoints that

²⁴ <https://swagger.io/docs/specification/paths-and-operations/>

the Ontology server API exposes, in this case for the SoS module of the developed Ontology (see Section 4 for more details).

Example of Ontology Server API endpoints for the SoS module.

```

/ontology/sos
/ontology/sos/{id}
/ontology/sos/{id}/environment
/ontology/sos/{id}/system
/ontology/sos/{id}/system/{id}
/ontology/sos/{id}/system/{id}/device
/ontology/sos/{id}/system/{id}/device/{id}
/ontology/sos/{id}/system/{id}/device/{id}/component
/ontology/sos/{id}/system/{id}/device/{id}/component/{id}

```

Figure 19 Example of Ontology Server API endpoints for the SoS module

The HTTP operations are defined for each endpoint (path) (GET, POST, PUT, and DELETE). A single path can support more than one operation, and an operation would have one path, except for the GET operation.

For instance, 'GET /ontology/sos' returns all available systems of systems in the ontology, whereas 'GET /ontology/sos/{id}' returns information about a particular system of systems. For the POST, PUT and DELETE operations the 'POST /ontology/sos', 'PUT /ontology/sos/{id}' and 'DELETE /ontology/sos/{id}', are used respectively. These last operations must be managed carefully, according to the ontology constraints.

Figure 20 shows the interactions between Auditing Framework GUI (Auditing GUI in the Figure) when it performs GET and POST requests for returning a particular system of systems and inserting a new one. When the Ontology Server receives the requests, it creates the corresponding SPARQL query to retrieve/insert the results from/in the knowledge graph database. The results are retrieved in JSON format.

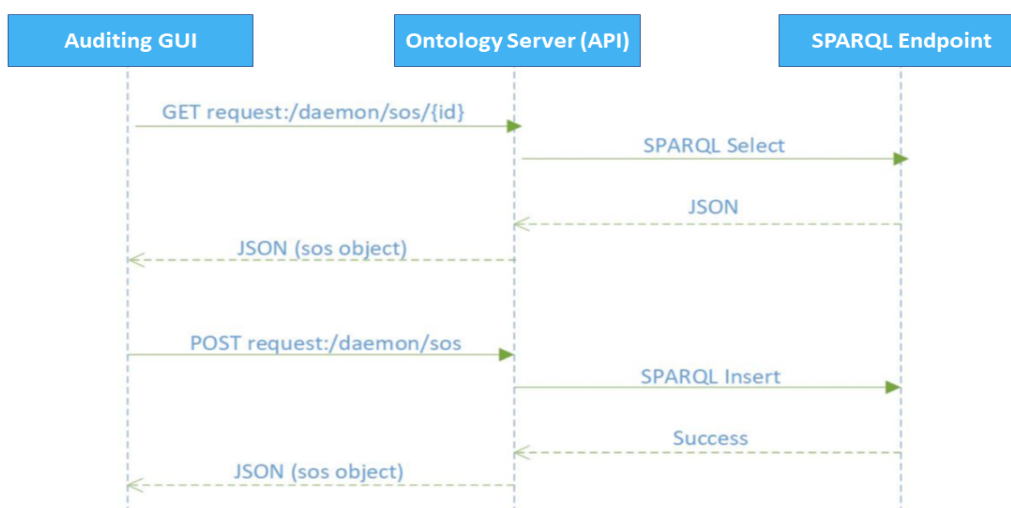


Figure 20 Example of GET and POST requests through the Ontology server

3.4.3. Exchanged Data Structure

Ontology Manager is based on RESTful service and communicates with components by exchanging data. To facilitate that communication, there is a specific JSON schema. It represents the data exchanged and allows interoperability with the current (and additional) components integrated within the BIECO Auditing Framework or Platform. Figure 21 reports the conceived JSON data structure schema, and it highlights the structure of the SoS JSON object, which has an SoS ID, a UseCaseID that identifies the use case the SoS is related to, the SoS Name, a Description of the SoS, and a Justification field containing the reason of choosing that SoS.

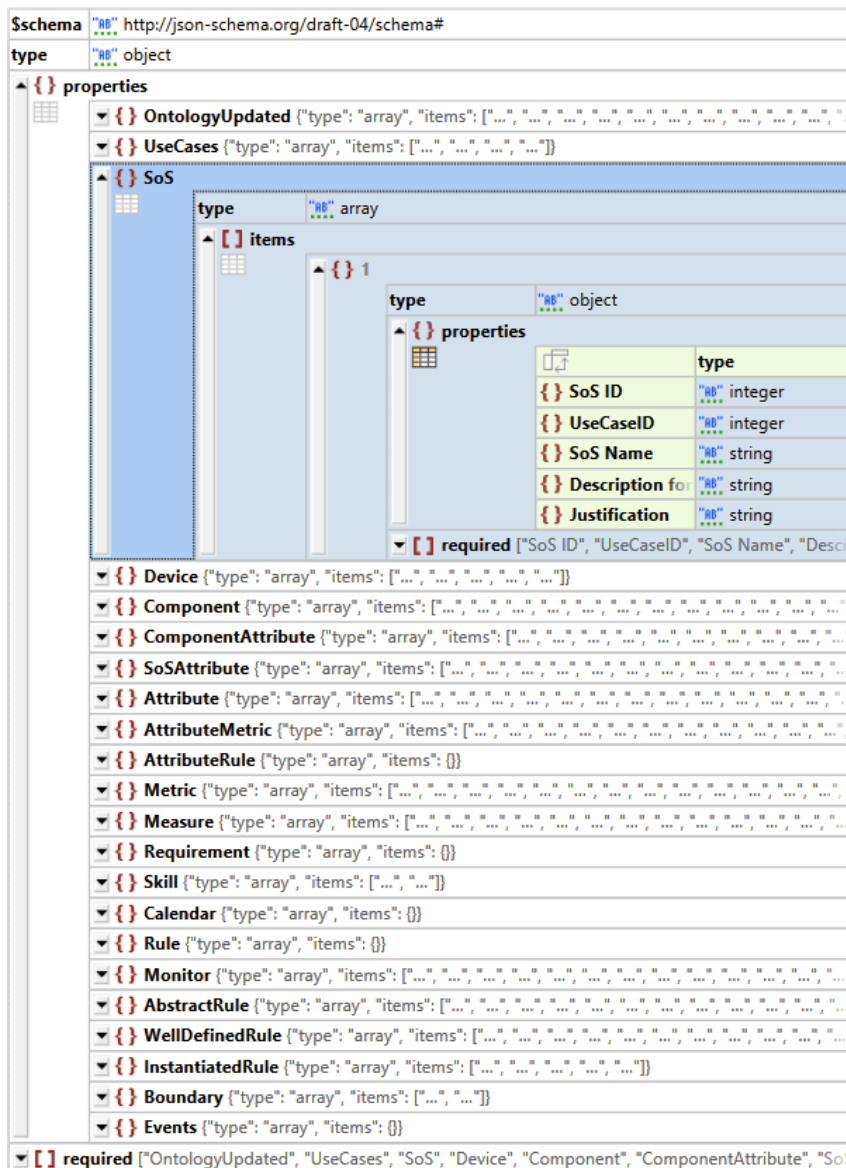


Figure 21 Ontology Manager Data Structure JSON Schema

3.4.4. Produces Data

All the data produced by the Ontology Manager must comply with the schema mentioned above. An instance of that schema containing data related to the System of Systems associated with the BIECO four use-cases is reported in Figure 22.

```

SoS
[
  {
    "SoS ID": 1,
    "UseCaseID": 1,
    "SoS Name": "ICT GW",
    "Description for GUI DEMO18": null,
    "Justification": null
  },
  {
    "SoS ID": 2,
    "UseCaseID": 2,
    "SoS Name": "AI Investment",
    "Description for GUI DEMO18": null,
    "Justification": null
  },
  {
    "SoS ID": 3,
    "UseCaseID": 3,
    "SoS Name": "Smart Microfactory",
    "Description for GUI DEMO18": null,
    "Justification": null
  },
  {
    "SoS ID": 4,
    "UseCaseID": 4,
    "SoS Name": "Autonomous Navigation",
    "Description for GUI DEMO18": "Autonomous navigation environment composed of 2 robotic units + 2 stations ",
    "Justification": "Common use case for design and runtime"
  }
]

```

Figure 22 An instance of SoS data

3.4.5. Technologies Used

The following provides the technologies for developing the Ontology Manager and distinguishing between tools and languages.

3.4.5.1. Tools

Protégé is a free, open-source platform that provides a suite of tools to construct domain models and knowledge-based applications with ontologies. Specifically:

- Protégé Desktop v.5.5.0²⁵, for windows, a platform-independent version that requires a Java Runtime Environment.
- WebProtégé is an online version of Protégé, and It requires an internet connection and a browser. After registering to the platform at <https://webprotege.stanford.edu/>, a specific workspace is available where it is possible to create

GraphDB supports:

- GraphDB uses RDF4J²⁶ as a library and its APIs for storage and querying.
- It supports the GraphQL, SPARQL, and SeRQL languages and RDF (e.g., RDF/XML, N3, Turtle) serialization formats.
- OWL 2 RL profile is fully supported and QL partially.
- It integrates OpenRefine for the ingestion of tabular data and provides semantic similarity search at the document level

3.4.5.2. Languages

Java: Java is the primary programming language for developing the Ontology Manager's

²⁵ <https://protege.stanford.edu/products.php#web-protege>

²⁶ <https://rdf4j.org/>

component. For the Auditing Framework GUI component, details are in Section 3.5.

Web Ontology Language (OWL) represents the complex knowledge about the system of systems and monitoring concepts and relations between them, i.e., the ontology used in the BIECO project.

SPARQL is used for retrieving and manipulating data stored in GraphDB.

3.5. Auditing Framework GUI

The Auditing Framework GUI manages the interaction between the user and the Ontology Manager component.

3.5.1. Communication Flows Inter Artifacts

The Auditing Framework GUI exposes a single interface, the actual graphical user interface, which presents data/information to the user and collects input. Examples of Auditing Framework GUI usage are provided in Section 8.1.

Communication with the Ontology Manager is done via the REST API that it exposes, so for this, the Auditing Framework GUI does not expose any interface. Details of the data structure are provided in Section 3.4.

3.5.2. Technologies Used

The Auditing Framework GUI is implemented in Java, HTML/CSS, and Javascript, using the following libraries:

- Java Spring Boot with Thymeleaf.
- VueJS.
- Bootstrap CSS Framework.

3.5.3. User Installation Guidelines

Considering the installation guidelines, specific details are provided in the following sections.

3.5.3.1. Hw/Sw Requirements

HW/SW requirements include:

- Java Runtime Environment 11
- Access to CDN (Content Delivery Network) resources for frontend components like VueJS and Bootstrap CSS.
- no restriction on port 8080 for the Auditing Framework GUI Component of the platform.

3.5.3.2. Licenses

No licenses are needed for using this component

4. Advancements in Ontology Manager

This section reports the evolution of the initial core ontology (i.e., MONTOLGY-MONitoring onTOLOGY), managed by the Ontology Manager introduced in deliverable D5.1 [11], to make it more modular, manageable, and comprehensive. In particular, the content has been re-organized into five modules, each containing a set of correlated concepts and relations between them. Therefore, the conceived enhanced ontology²⁷ is related to the challenges CH2 and CH3 by proposing a well-defined Taxonomy used along the BIECO lifecycle. It allows Knowledge derivation reasoning and inference of new knowledge.

The core ontology and its evolution aim to help the different SoS stakeholders gather functional and non-functional properties related to the various parts of SoS. Consequently, that enables the definition of concrete monitoring rules associated with a specific property to demonstrate compliance (non-compliance) with the selected properties.

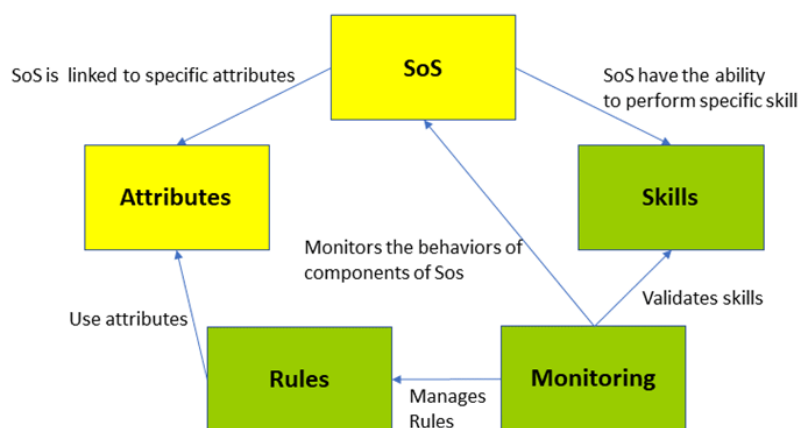


Figure 23 Ontology Modules

The core ontology defined in deliverable D5.1 [11] is composed of two main modules: the System of Systems (SoS) Module (containing eight concepts) and the Monitoring Module (which includes five concepts), with a total of 13 (thirteen) concepts.

The advancement consists in adding new concepts and reorganizing the content in a more manageable and modular way to enable interoperability and facilitate both extensibility and maintainability. More precisely, as reported in Figure 23, the new shape of MONTOLGY is divided into five modules: SoS, Attributes, Skills, Monitoring, and Rules. The remainder of this section briefly describes each module and points out the new concepts/classes by colouring the shape outline in red.

²⁷ The enhanced ontology is called DAEMON in the joint paper of CNR and UNI BIECO partners, accepted in the 5th IFIP International INTERNET OF THINGS (IoT) Conference [8].

4.1. SoS Module

Differently from MONTOLGY, the System of Systems is modelled as a composition of System, and it is influenced by a specific environment in which it operates and is executed. Therefore, a System is a collection of Devices representing the object of the monitoring activities. As in MONTOLGY, each Device is composed of a specific set of Components, as shown in the figure below.

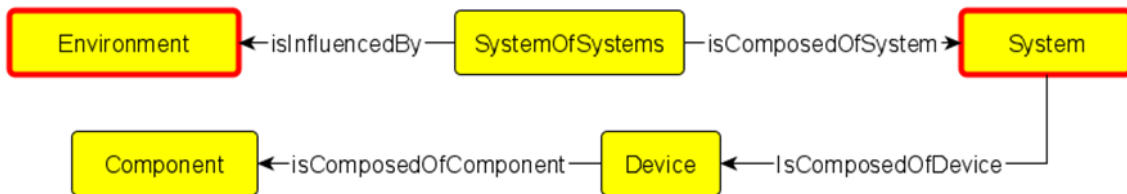


Figure 24 System of Systems (SoS) module

4.2. Attributes Module

An Attribute is a functional and non-functional property related to a specific SoS concept (see Figure 25). And this module contains all the concepts related to the observable properties of the classes in the SoS module. As in Figure 23, this module introduces two specific concepts: QualitativeAttribute, and ObservableAttribute, i.e., quantitative attributes used to define both the Measure and Metric used to define monitoring rules. The Attribute hierarchy is also expanded by adding three sub-classes: EnvironmentAttribute, SystemAttribute, and DeviceAttribute, enabling monitoring of their behaviour through specific monitoring rules.

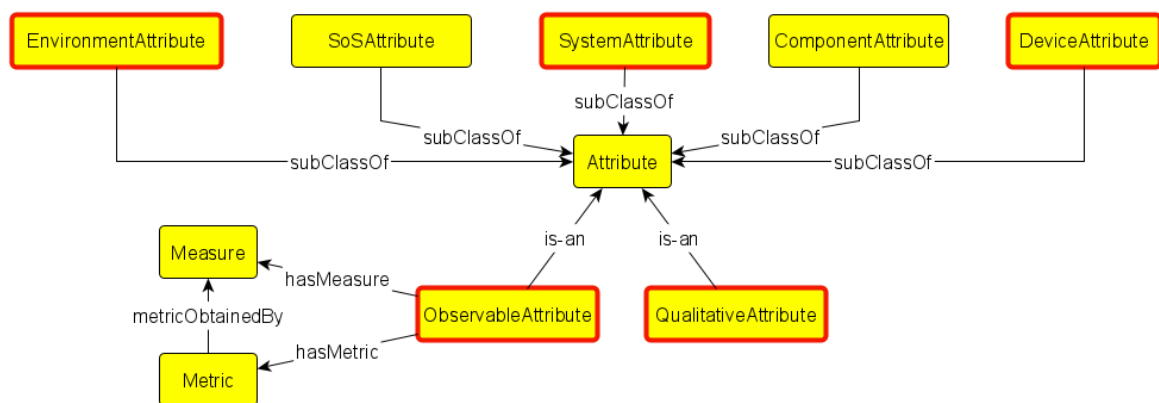


Figure 25 Attributes Module

4.3. Skills Module

The original concept of Skill has been extended in two ways. Firstly, a skill hierarchy is created by leveraging the actual concept of Skill as a super-class of the hierarchy. Two specific sub-classes (BasicSkill and ComplexSkill) are connected through the relation *isComposedBy*.

As shown in

Figure 26, a ComplexSkill is composed both through a set of BasicSkill, or/and iteratively throughout a set of ComplexSkill. Secondly, the concept of ObservableSkill is introduced, i.e., the observed ability related to the SoS concept that can be validated through the monitoring facilities.

Differently from the core ontology described in deliverable D5.1 [11], the Requirement class is connected directly to ObservableSkill through the *isRelatedToSkill* association. Therefore, each ObservableSkill, specified as a set of Requirements, can be verified through a specific Rule.

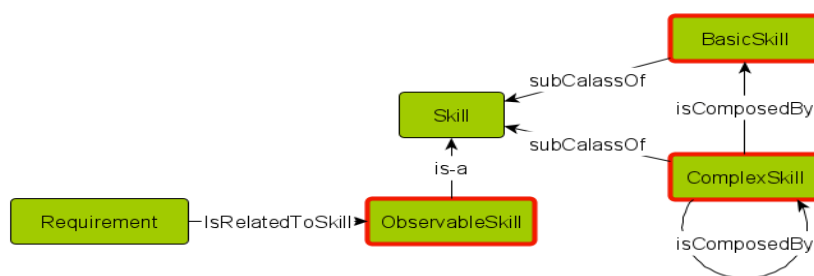


Figure 26 Skills Module

4.4. Rule Module

The advancement of the core ontology leverages the concept of *Rule* by providing a well-formed hierarchy with the following sub-classes (see Figure 28):

- *AbstractRule* points out a generic rule that is not yet instantiated within the execution context and gathered from the navigation of the ontology.
- *WellDefinedRule* refers to a rule ready for being translated to the destination language of the Complex Event Processor and related to the monitoring of a specific device.
- *InstantiatedRule* is a rule written using the language understandable by a monitor engine.
- *Boundary* contains specific values that express the applicability ranges of the rule.

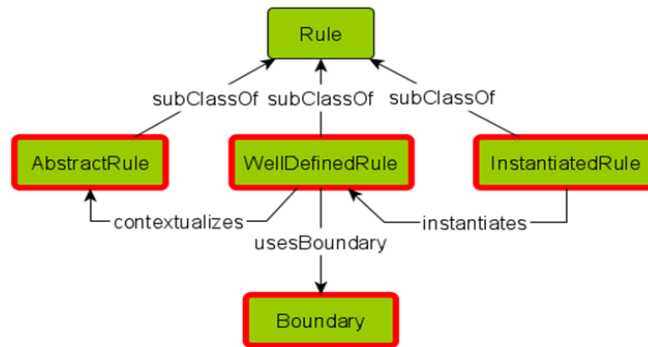


Figure 27 Rule Module

To better clarify the complexity of the process involved in obtaining a processable rule, in Figure 28, a graphical representation of the evolution of the rule: from abstract to instantiated one can be found.



Figure 28 Rule Transformation Process

In particular, the abstract rule is a generic natural language description of the objective of the auditing activity that is easily understandable by non-expert users, for instance, such as the maximum number of established simultaneous connections between two components. The abstract rule is then refined into the well-defined rule, a semi-structured and implementable rule. The users need to add specific details about the context, for instance, the maximum number of established simultaneous connections. An example of abstract and well-defined rules can be found in Figure 29.

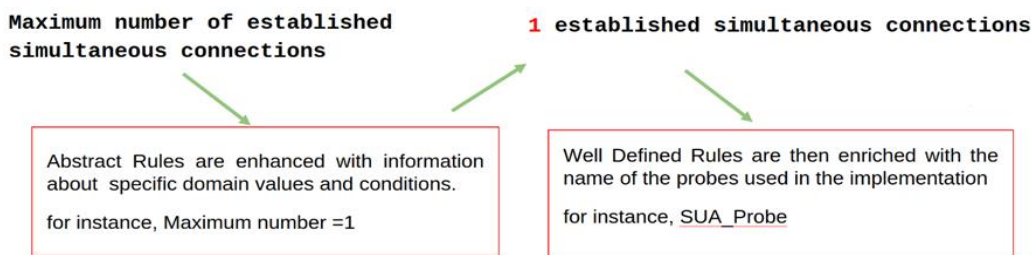


Figure 29 From Abstract to Well-defined rule enrichment process

Finally, the well-defined rule, enriched with the name of the probes used by the user, will be automatically translated into an instantiated rule according to the monitoring language used. The Runtime Monitoring uses this during the Auditing Framework execution.

A typical structure of an instantiated rule can be summarized as follows:

```

1 declare // Optional
2 rule "rule name"
3 // Attributes
4 when
5 // Conditions
6 then
7 // Actions
  
```

Figure 30 Drools Rule Skeleton

It can contain one or more rules that define the rule conditions (when) and actions (then) at a minimum.

4.5. Monitoring Module

The core class of the Monitoring module is the Monitor, which observes rules organized in the Calendar, i.e., an ordered set of rules (see Figure 31). Each Calendar can validate a specific *ObservableSkill* at run-time defined in the Skills module. The Monitor has a specific EntryPoint used to communicate with the Probe.

A Probe is a piece of software code that can be injected into an observed/monitored component, device, or system, and it can send Events according to a specific format. The probes can send events at regular intervals or in a particular situation. The sent events contain information related to the occurrence of actions on the observed SoS entity.

The term Event defines the change of a state within a system. This state change is generated when a method call is executed, or internal action is enacted. The injected Probe will pack this atomic action into an event and notify the Monitor to perform the processing action on the event stream. To be correctly managed by a concrete monitor, the event should contain several pieces of information needed for analyzing a snapshot of what is happening within the System Under Audit.

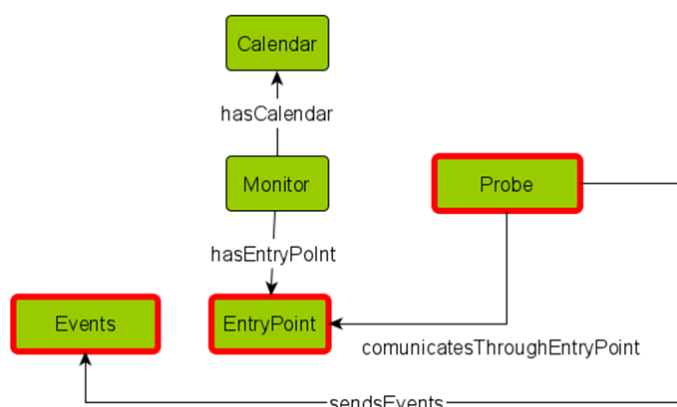


Figure 31 Monitoring Module

5. Advancement of the Auditing Framework Interface

Considering the Deliverable D5.1 [11], the conceptual structure of the Auditing Framework interface has been improved considering the four different steps:

1. Pre-setup.
2. Artifacts preparation.
3. Finalization of the Pre-setup.
4. Execution.

The following describes the steps and their interaction.

Pre-Setup

Once the Design Phase concludes, and BIECO's Runtime Phase is due to start, the user sends a notification to trigger the *Pre-setup* phase. Then retrieval of the domain-specific language for the specification of the Digital Twins takes place and includes the creation of the Digital Twins and the instrumentation of the CE/SUA with the probes.

For this reason, the Auditing Framework Interface needs to support the freezing/resuming of the ongoing setup session. The initial auditing rules refinement occurs, and the components' execution starts.

After starting the Runtime Phase, the execution pattern requires the Auditing Framework Setup to be enabled. This process is exemplified in Section 8.

Through the Auditing Framework Interface exposed within the BIECO Orchestrator, the user may execute the operations needed for the Auditing Framework *Pre-setup*. They include browsing the ontology data for classifying the type of CE and SUA to get the subset of the relative rules to be assessed during the Auditing Framework execution.

Artifacts Preparation

This phase includes getting probes information or artifacts for instrumenting CE, SUA, and DT. It also gets DSL related to the Digital Twin that the user needs to instantiate. The information acquired through Ontology Navigation can be saved for recovery in the future, and guidelines about instrumentation with probes of the CE/SUA and the setup of the DT through the DSL development are provided to the user.

Finalization of the Pre-setup

The auditing activity can be restarted as soon as the instrumentation and the DT definition are completed. After completing this process, the session previously saved can be recovered and the abstract rules refined or completed.

In this case, the user can modify or confirm the subset of rules selected for the monitoring procedures.

Execution

Once confirmed or updated, those rules can be executed by the monitoring platform component of the Auditing Framework.

6. Advancement of the Predictive Simulation

This section targets the Challenges CH4 and CH5 introduced in Section 1.1 about detecting malicious deviations through specialized Digital Twins. Concerning D5.1 [11], the development has been improved by providing a solution based on simplified concerns that could prevent an intelligent software component from detecting it is under evaluation. Initially designed for achieving a clear understanding of either functional or timing behaviour of real-time control systems, this approach enables a concern-directed prediction of the trustworthiness of intelligent software behaviour.

By focusing the scope of the evaluation on either schedule, function interaction, or communication protocol between the intelligent software and interacting entities (such as software, hardware, or subsystem), specialized and faster evidence of trust can be achieved. As depicted in Figure 32, runtime evidence of trust can be provided through the execution of **horizontal abstractions** of a software component or systems behaviour, which are directed towards a specific scope of the evaluation and can be executed at every level of **vertical abstraction**.

From top to bottom, vertical abstractions can be defined with a range of details varying from a very high level where they take the shape of input/output tables or state charts to very concrete levels when fully implemented. The horizontal abstraction of the intelligent software behaviour can be executed to provide specialized evidence of trust. Further on, vertical abstractions can propagate evidence of trust between different horizontal levels for assuring the satisfaction of specific system-level goals and ongoing coalitions.

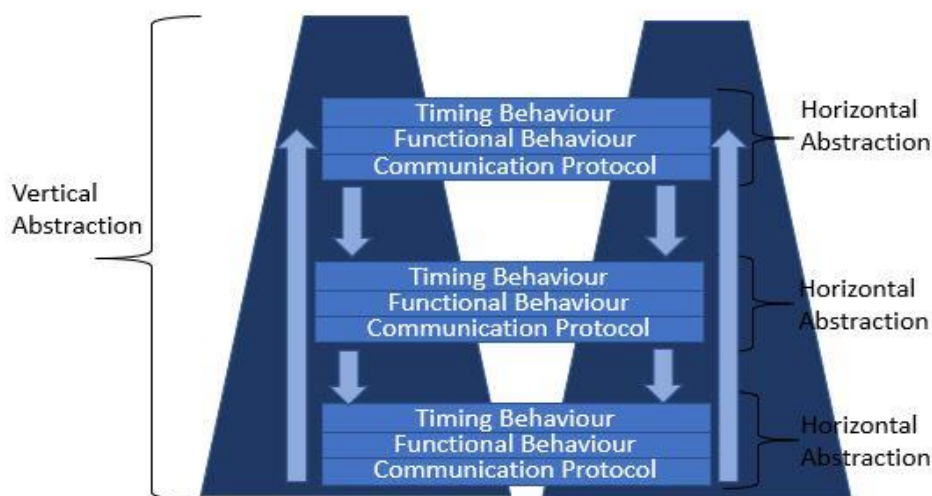


Figure 32 Execution of parallel abstraction

6.1. Derivation of Specialized Digital Twins

The provided approach leverages the principle of simplification in the design of software functions, creating more superficial behavioural structures capable of avoiding uncontrolled feature interaction. Concretely, the process of simplifying concerns to define concern-oriented abstract models has been redirected. In deriving abstract models, it is essential to identify the main concerns and leave the ones subordinated to them apart. The model derivation process needs to capture the characteristics of a

system and the behaviour of its components at a specific level of abstraction. This process varies in complexity by the nature of the system to be abstracted.

The approach that enables runtime detection of malicious deviations based on Predictive Simulation requires a Design Phase for engineering systems artefacts that support the later runtime prediction and conformity monitoring.

In this phase, different models of the system behaviour are created, including functional models that enable runtime evaluation of functional interaction, temporal models that enable timing predictions used in evaluating a software smart agent's synchronization capability, and models that will allow the runtime evaluation of the communication protocol.

As discussed in Deliverable D4.2 [10], for enabling the runtime prediction of timing behaviour, in the pure Predictive Simulation phase, the temporal logic model is used to validate the accuracy of the Digital Twins that provide the timing abstractions.

The development of the two artifacts: the temporal model and the software smart agent, can lead to a set of situations, namely:

- 1) No faults in either of the artifacts: Ideal situation
- 2) Same fault in both artefacts: Fair prediction
- 3) Different faults between artifacts: a situation that leads to dishonest trust

6.1.1. The Process

To address these deviations, during pure Predictive Simulation, the behaviour of a smart software agent, subject to trust evaluation and the corresponding temporal logic model, is evaluated for consistency before being deployed. Then, during runtime within a simulation environment, before the execution of the smart software agent, the corresponding temporal model is fed with real-time data and executed much faster. During this phase, the specialized Temporal Digital Twins are executed with other interacting components of the software. Smart software can be either other software components, hardware resources, or system platforms.

6.1.2. Derivation of Specialized Timing Digital Twins

The model is specifically targeted toward capturing untrusted deviations from the minimum and maximum delay of execution. For the experienced reader, it is evident that the model is based on a fragment of Linear Temporal Logic, but with a refined validation clause regarding the temporal connective. These models traditionally provide means for formally checking events' occurrence over time captured in traditional connectives of *until* and *since*, our models differ from general Linear Temporal Logic models in the sense that the temporal connective is limited in its future scope, and the validity of statements is to be contained in said scope. Besides this restriction for the temporal connective, the model provides customized predictive-simulation restrictions that check for future events. Overall, the model is highly expressive for timing considerations of behaviour evaluated within the Predictive Simulation paradigm.

For any simple statements p, q, \dots , any complex statements A, B, \dots , the unary connectives \neg (Negation), \diamond (In the future), and the binary connectives \wedge (Conjunction), \vee (Disjunction), \rightarrow (Entailment), the following recursive forming rules apply:

- a) For any simple statement p , p is a well-formed statement. Furthermore, if $A = p$, then A is well-formed statement.
- b) If A is a well-formed statement and $*$ is a unary connective, then $*A$ is a well-formed statement.
- c) If A and B are well-formed statements and $*$ a binary connective, then $A * B$ is a well-formed statement.
- d) There are no more well-formed statements than those defined by the clauses (a), (b) and (c).

Simple and complex statements refer to any data generated by events. The array of connectives excludes any quantifiers connectives (e. g., $\forall x$, for all x) and focuses on the propositional fragment rather than the first or higher order ones. This helps keep the forthcoming model to a minimum, making its implementation easy as only simple operations would be required. It also reduces the computational complexity and makes its implementation in resource-constrained devices much easier.

A model M is the structure $M = \langle K, T, |= \rangle$, where K is for instance a set of robots a, b, c, \dots ; i. e., $K = \{a, b, c, \dots\}$; each element of K , each robot, is a set in itself that includes a minimum and maximum time delay, m and h respectively, among other optional characteristics o_1, o_2, o_3, \dots ; i.e., $a = \{m, h, o_1, o_2, o_3, \dots\}$. T is a set of temporal points t_1, t_2, t_3, \dots ; i. e., $T = \{t_1, t_2, t_3, \dots\}$. Finally, $|=$ is a relation from K to the set of statements such that the following clauses apply:

- (1) $a |= A \wedge B$ if and only if (iff) $a |= A$ and $a |= B$
- (2) $a |= A \vee B$ iff $a |= A$ or $a |= B$
- (3) $a |= \neg A$ iff $a \not|= A$
- (4) $a |= A \rightarrow B$ iff $a \not|= A$ or $a |= B$
- (5) $a, t |= \diamond A$ iff $h = t + d_1$, $m = t + d_2$ & $\exists s, s \in T$, with $t < s$, $m < s < h$, and $a, s |= A$, and $\forall u, u \in T$, if $t < u < s$, then $a, u \not|= A$.

7. Advancements of Runtime Monitoring

Considering the complexity of the ecosystem, problems caused by a single constituent (hardware or software) piece could either compromise the entire system or ecosystem or expose it to hidden faults, malicious behaviour, or vulnerabilities able to impact or propagate to the other interconnected parties.

Thus, techniques for efficiently and effectively assessing and preventing anomalies and dangerous situations are required, especially when a new device, software, or system component is integrated into an existing IoT system or ecosystem.

Among them, focus on using a runtime monitoring approach to detect, trace, and notify security and privacy threats during the development of the online execution [21].

Monitoring approaches have been recognized in the literature as practical solutions that provide dynamic mechanisms for analysing functional and non-functional properties against well-stated conditions, such as contractual conditions for trust.

Indeed, a monitor engine can collect events for different goal evaluations (strategic-tactical-operational) and from various systems and system components (including sensors). The collected data are then used for inferring complex patterns, each associated with specific functional and non-functional properties. In practice, the monitoring activities involve the collection and analysis of different data sources (e.g., sensors, software, and hardware components or devices); the assessment of functional and non-functional properties relative to components or devices of the system of ecosystems, the detection of properties violation; the rising of specific alarms and the actuation of countermeasures if necessary.

At the state of the practice, there are three main trends for detecting or predicting runtime vulnerabilities or violations [21].

Using (previous) knowledge: In this case, previous data collections or past examples of failures are used to predict the output's quality. The proposals are usually based on either deep learning approaches as in [23] or rely on a separate system to monitor and predict a target model's failure (as in [19], or on a perceptions system[20], or methods for prediction learning (as in [18]).

Using input data: This group includes monitoring methods based on the analysis of the stream of input data coming from different sources, like, for instance, sensors, components, devices, systems, or models (as in [1][2][3]).

Using confidence estimations: In this case, confidence learning and uncertainty estimation are used for the output evaluation (as in [5],[4], and [7])

Starting with the proposal provided in deliverable D5.1 [11], the Runtime Monitoring has been refined and finalized as described in Section 3.2 to target the second trend. Indeed, Runtime Monitoring uses the input data for violation detection. In particular, the proposed infrastructure has been improved by considering aspects such as:

- possibility of managing the heterogeneity of events producers.
- possibility of interacting with different contexts and environments.
- ability to manage multiple CEP (Esper, Drools) instances.
- ability to deal with Multiple data storage (influxdb, MySQL).
- possibility to communicate through a Rest interface and JSON messages.
- mediation of messages through JMS2JSON mediator.

- possibility to scale the amount of CEP and events listener channels.
- optimizing message processing and improving the quality of services by executing routing techniques to the most suitable CEP.
- improving the possibility to scale monitoring instances using docker container deployment (ongoing activity).

All these facilities have been developed according also to the principles specified in the *Reactive manifesto*²⁸:

- **Responsiveness** for guaranteeing a consistent quality of services.
- **Resilience**: trying to manage all possible exceptions and interruptions that may occur during the execution to provide a highly available system.
- **Elasticity**: allowing the number of complex event processors and channels communication scales to avoid central bottlenecks.
- **Message driven**: all the messages are asynchronous and loosely coupled between components involved in the evaluation.

7.1. Runtime Monitor Innovation Aspects

The BIECO project focuses on integrating monitoring facilities that provide a predictive engine for functional or non-functional property definitions, dynamic implementation of monitoring rules, and an adaptive approach for functional and non-functional property verification and assessment.

As described in Section 3.2, the available monitoring infrastructures have been analysed to improve the Runtime Monitoring. In particular, the current implementation of the Runtime Monitoring leveraged some of the features proposed by existing solutions such as XDR (eXtended Detection and Response), EDR (Endpoint Detection and Response), and SIEM (Security Information & Event Management).

The following details of the improvement concerning each of them are provided.

7.1.1. Leveraging the XDR

Considering the XDR, the Runtime Monitoring leveraged the mechanism of the data-lake to make it more suitable for prompt analysis and reactions. XDR uses a comprehensive approach based on a data-lake paradigm for detection and response. It collects and secures data on activities on multiple levels and provides automated analysis of this data to detect threats. As a result, security analysts are equipped to conduct deeper investigations and adopt more rapid responses. The Runtime Monitoring implemented in BIECO leveraged the multiple levels of automated data analysis of the XDR by exploiting a federated set of Complex Event Processors (CEPs). Indeed, each CEP is dedicated to either a specific threat of functional or non-functional property verification or to analysing a particular type of data. The CEPs are strongly interconnected and able to exchange complex events. This allows evaluation to be executed faster than XDR to speed up the detection of emerging threats and possible malicious behaviour.

²⁸ <https://www.reactivemanifesto.org/>

7.1.2. Leveraging the EDR

Considering the EDRs, Runtime Monitoring leveraged their countermeasures management by providing a more flexible and customizable mitigation approach. Indeed, Runtime Monitoring lets the users define their countermeasures to avoid stopping the runtime activities as a unique alternative.

The customizable countermeasures stored in the BIECO Ontology can be associated with the Runtime Monitoring rules and applied during the monitoring activity in case of violation detection.

Various countermeasures have been considered and are currently partially integrated into the Runtime Monitoring. The Complex Event Processor component triggers the countermeasures. This may involve the endpoint violating the expected behaviour. Ordered by increasing complexity, the countermeasure implementation is

- 1) Notify a specific message in which logs and data of the detected violation are reported.
- 2) Stop the execution of the component or device responsible for the violation (the SUA or one of the CE elements).
- 3) Execute a countermeasure to mitigate the violation and bring the system back safely. This includes the possibility of integrating *Smart Agents* and artifacts into the probes. The Smart Agent can get a security issue as a counter effect: it can be considered a potential backdoor. Therefore, specific risk mitigation strategies should be adopted.

7.1.3. Leveraging the SIEM

Considering the SIEM solutions, the Runtime Monitoring includes mechanisms for gathering all the events generated across the System Under Audit (SUA). However, Runtime Monitoring leverages it to manage events not only for storage and post-analysis activity and for online validation of possible functional and non-functional property violations. Indeed, BIECO Runtime Monitoring exploits the knowledge about the correct SUA behaviour for evaluating the ongoing SUA behaviour to detect and notify threats promptly.

8. Auditing Framework Execution

As part of the BIECO Runtime phase execution, the Auditing Framework targets the monitoring of functional and non-functional properties of the System Under Auditing (SUA) and the Controlled Environment (CE). It involves the interactions between the System of System, the Controlled Environment, and the new component or device (the SUA).

The BIECO Runtime phase lets the execution of the Auditing Framework in three different situations:

- a) **CE is simulated**, i.e., the CE is a simulation model able to represent the real environment. In this case:
 - i) SUA can be simulated or real
 - ii) Models or Stubs can be used to simulate the environment components in which SUA is executed.
- b) **CE is executed in a testbed environment**, i.e., the CE is a representation of the real environment but executed in a testbed framework to have the possibility to control the internal status of each CE component and to manage violations safely. In this case:
 - i) The SUA is a real component
 - ii) The CE components directly interacting with the SUA can be
 - real components
 - simulated models,
 - executed using stubs.
- c) **CE is executed in a real context**, i.e., the CE and its components are executed in a real (operational) environment. In this case,
 - i) The SUA is a real component.
 - ii) The CE components directly interacting with the SUA are real components.

As detailed more in deliverable D8.2 [17], the BIECO project provides four Use Cases (UCs) to validate the Auditing Framework activity. In particular, the use cases let the validation in the a) and b) situations described above. Specifically:

UC 1 - ICT Gateway: provides an example of the CE being executed in a testbed environment (situation b)). In this case, the ICT gateway is the SUA, while the other CE components directly interacting with the SUA are executed using stubs.

UC 2 - AI Investment Platform: provides an example of the CE executed in a testbed environment (situation b)). In this case, SUA is a real component, and the CE components directly interacting with the SUA are real.

UC 3 - EV Smart Microfactory: provides an example in which the CE is executed in a testbed environment (situation b)). In this case, SUA is a real component, and the CE components directly interacting with the SUA are real.

UC 4 Coppelia: provides an example in which the CE is simulated (situation a)). In this case, the CE (i.e., the Autonomous Navigation) and the SUA (i.e., a Robot Unit 1) is simulated.

The following sections describe the Auditing Framework execution on the UC 4. It focuses on the interaction between the four main components: the Auditing Framework GUI, the Ontology Manager, the Runtime Monitoring, and the Predictive Simulation.

8.1. UC 4 Coppelgia

In UC 4, the Controlled Environment consists of a CoppeliaSim simulation representing the multi-robot navigation scenario for intralogistics as depicted in the overlay (see Figure 33). As better specified in deliverable D2.4, the System Under Audit is part of the robot, with probes being injected into the local planner component for navigation. Under normal conditions, both robots depicted in Figure 33 should follow the computed path while avoiding environmental objects and each other.

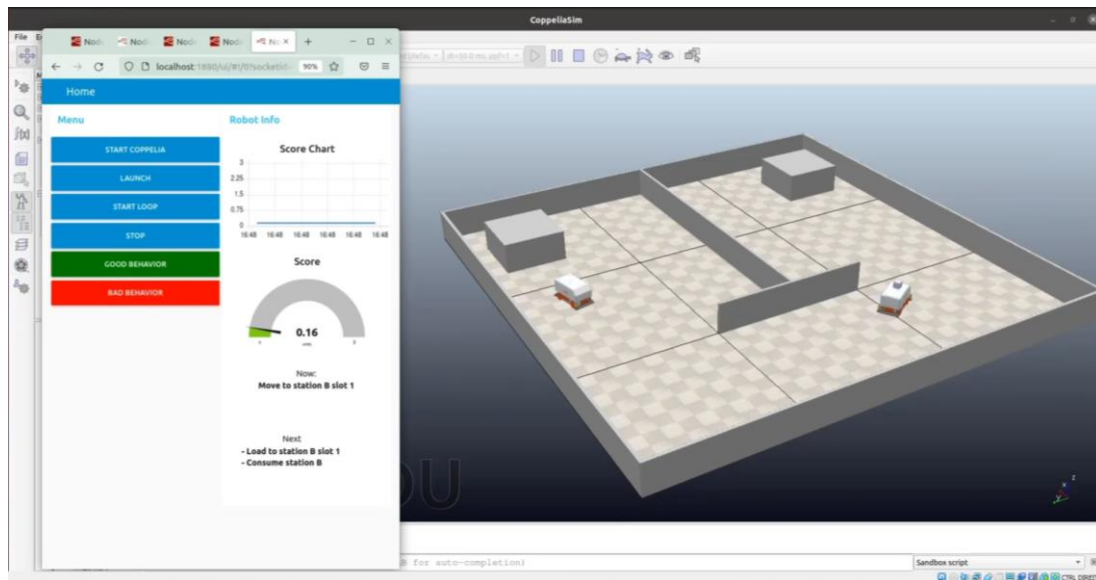


Figure 33 Coppelia Simulator

The user can start the auditing activity by interacting with the Runtime GUI²⁹. As shown in Figure 34, the Auditing Framework begins with selecting the Auditing Framework *Setup* features (see details in Section 3.5). The following sections detail the execution of each of them, starting from the first step, i.e., the *Pre-setup Phase*.

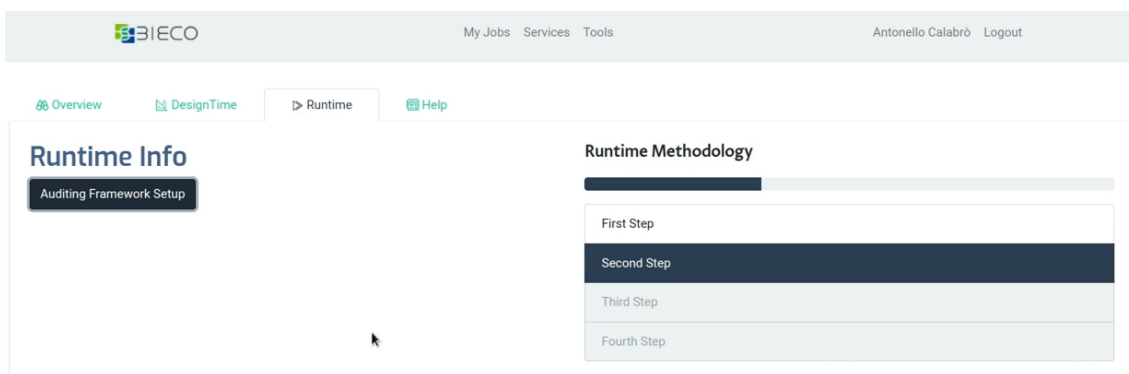


Figure 34 Runtime GUI: Auditing Framework Setup

²⁹ Note that the Runtime GUI is the BIECO framework interface for the overall Runtime Phase management. The Auditing Framework GUI is instead the GUI specifically developed for the management of the Auditing Framework activities.

8.1.1. Pre-Setup Phase

As shown in Figure 35, the activity allows the user to explore the classification and categorization of the different Systems of Systems, their devices, and components. The *Pre-setup Phase* involves the collaboration between the Auditing Framework GUI (see Section 3.5) and the Ontology Manager (see Section 3.4).

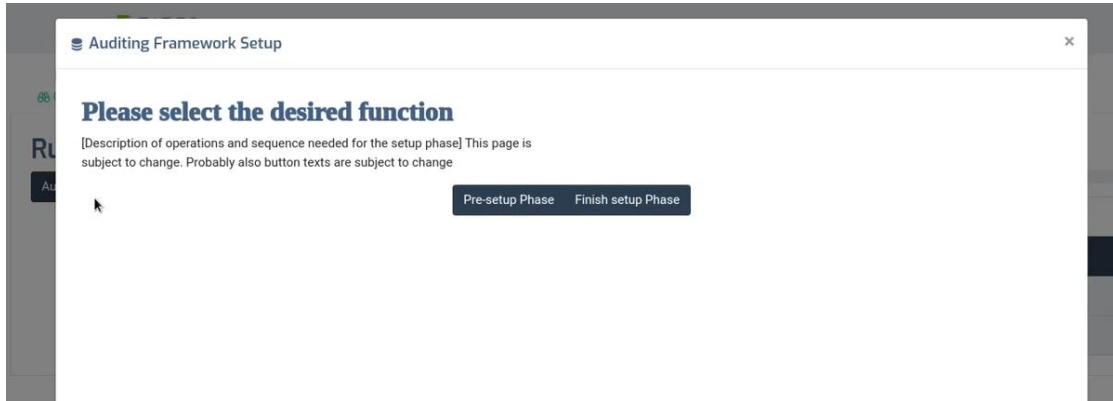


Figure 35 Auditing Framework GUI: Auditing Framework Pre-Setup

In particular, the Auditing Framework GUI provides the user easy-to-use means for navigating the ontology. Indeed, the user selection forces a suitable ontology query to guide the definition of the rules to be used during the auditing stage (see Section 3.4).

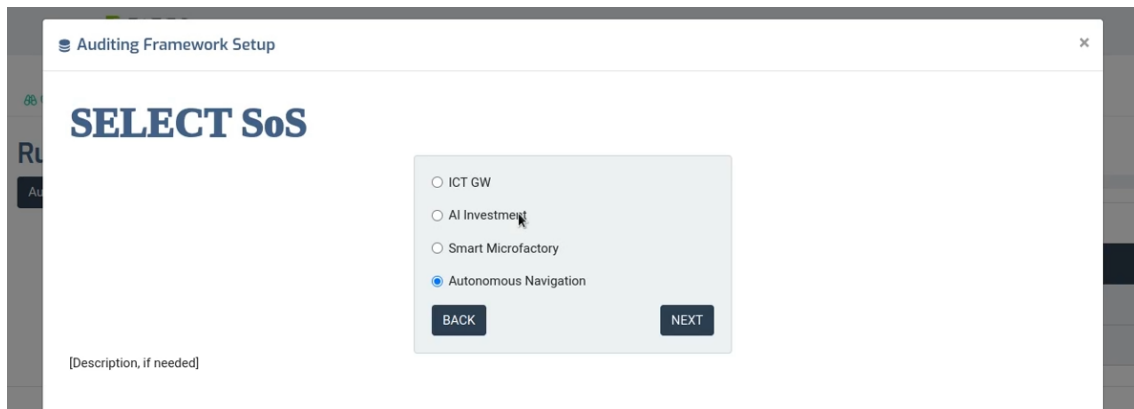


Figure 36 Auditing Framework GUI: SoSs selection

As shown in Figure 36, a list of possible controlled environments is provided. In this case, Autonomous Navigation can be selected as the target CE for UC 4. Successively, the list of the CE devices is visualized through the Auditing Framework GUI to let the user select the target SUA. Figure 37 shows that *Robot Unit 1* is chosen as the target SUA for the UC4 experiment.

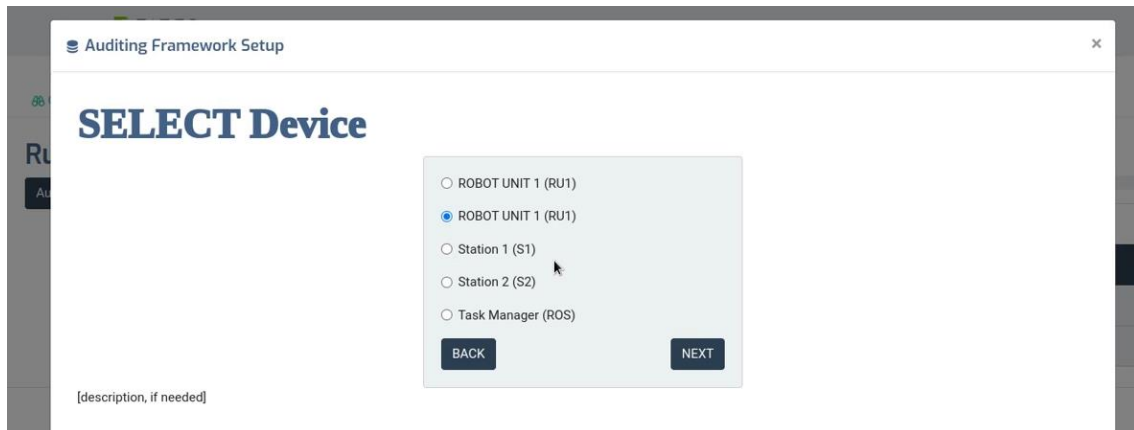


Figure 37 Auditing Framework GUI: Device selection

This selection forced the Auditing Framework GUI to query the Ontology Manager with the name of the components to be visualized.

As soon as this data is available, the Auditing Framework GUI visualizes the components list to the user to let them select the suitable one. As shown in Figure 38, for UC 4, the selected component is *Local Planner_1*.

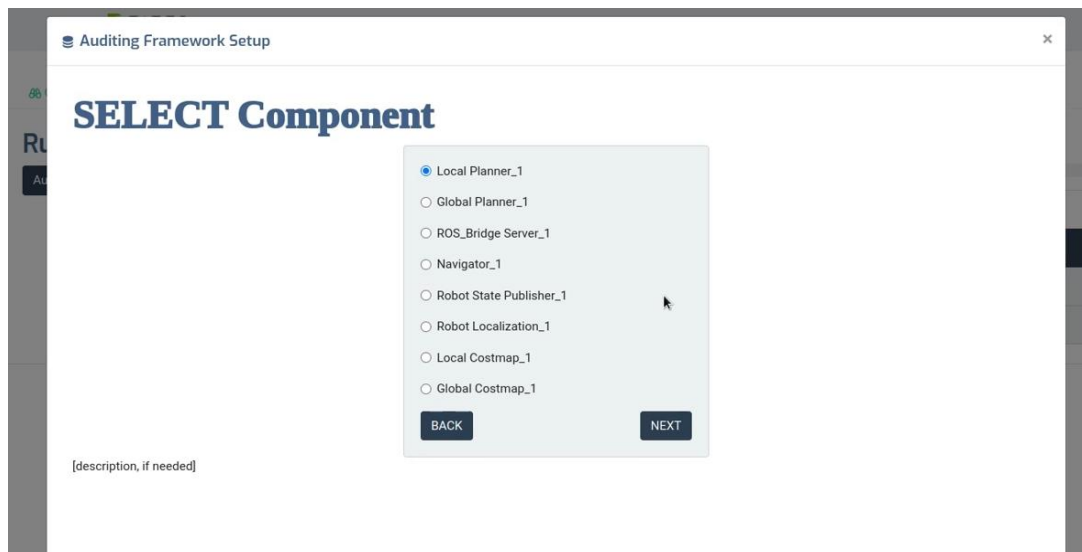


Figure 38 Auditing Framework GUI: Components selection

As before, through the collaboration between the Auditing Framework GUI and the Ontology Manager, the visualization of the specific component skills is provided: *connectivity* and *movement* (see Figure 39).

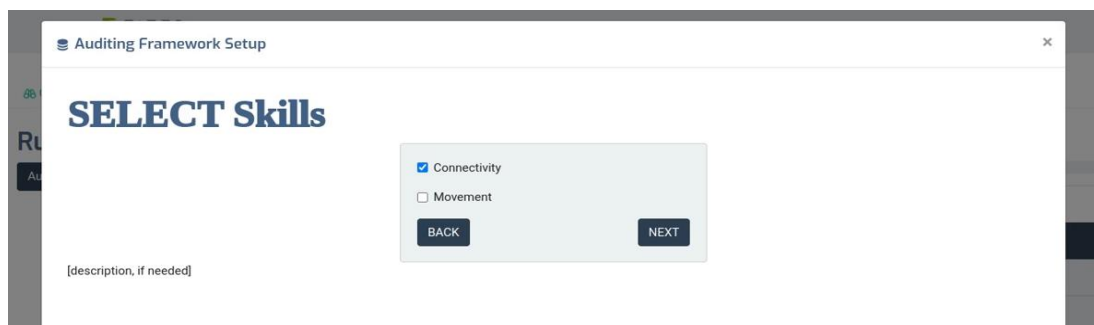


Figure 39 Auditing Framework GUI: Skills selection

In UC 4, connectivity is the skill considered for experimentation because it is regarded as the most critical from the security point of view.

Again, the collaboration between the Auditing Framework GUI and the Ontology Manager provided the user with the lists of the most suitable functional and non-functional properties for the selected SUA and CE.

As shown in Figure 40, the properties are presented as high-level specifications and correspond to the ontology abstract rules (see Section 4.4). According to the ontology representation, the abstract rules are classified as standard rules, i.e., non-functional properties that can be assessed through the Runtime Monitoring, and Pure Predictive rules, i.e., functional properties that can be predicted using Digital Twin in the Predictive Simulation.

As shown in Figure 40 in UC 4, one of the standard rules that the user could select is the *“Maximum number of established simultaneous connections.”* The rule targets the mutual interaction between Local Planner_1 and Autonomous Navigation.



Figure 40 Auditing Framework GUI: Select/adapt abstract rules selection

As shown in Figure 41, the rule boundaries can be established either using the Blueprints data collected during the design time phase execution or can be provided by the user. In both cases, the values are managed through the ontology and remain valid for all runtime execution.

Considering the pure predictive rules instead, in UC 4, one of the properties considered is the “*Expected communication pattern through an ordered list of message types*” (see Figure 40). This rule is visualized as an Ontology Manager query and focuses on the behavior of the Local Planner_1. It requires that the Digital Twin predicts and forecasts to monitor the specific message order.

The last interaction between the Auditing Framework GUI and the Ontology Manager concerns the definition of boundaries. Indeed, if not provided by the Blueprint data analysis, the user needs to insert the boundaries for the selected abstract rules as the last step. As shown in Figure 41, the value of the “Maximum number” is set to 1 for the standard rule selected.

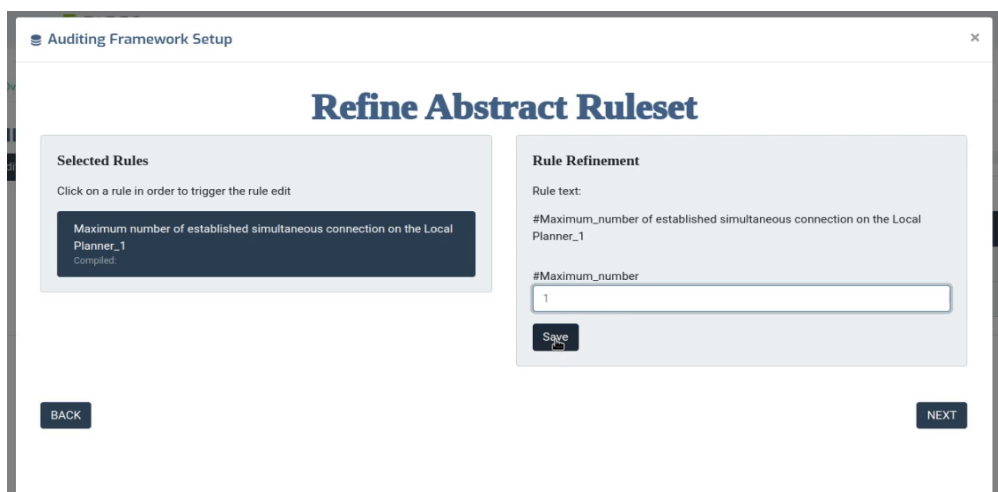


Figure 41 Auditing Framework GUI: Refine Abstract Ruleset

This concludes the Pre-Setup phase and starts preparing the following steps described in the following sections.

8.1.2. Offline Activities

When the pre-setup phase ends, the interaction between the Auditing Framework and the Runtime GUI provides the user with downloadable artifacts helpful in preparing for the following auditing activities.

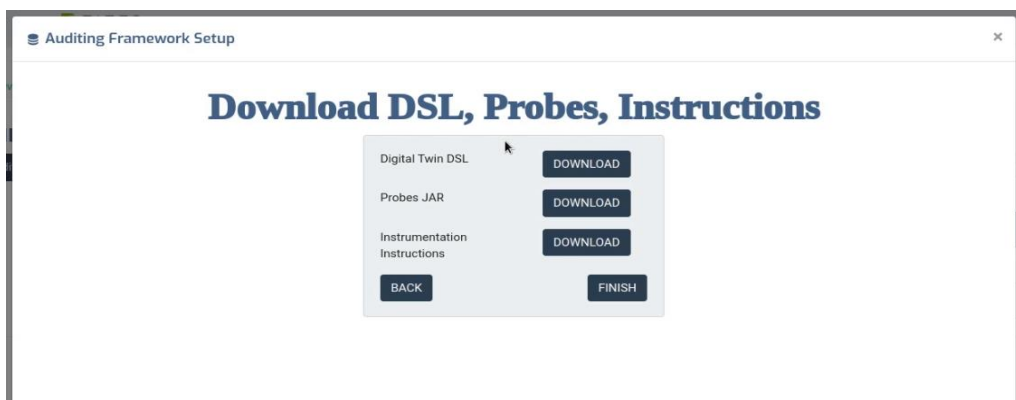


Figure 42 Auditing Framework GUI: Refine Abstract Ruleset

As shown in Figure 42, the artifacts include

- A set of guidelines for using the DSL language and deriving the Digital Twin.
- A jar file of the executable probe.
- A set of guidelines helpful in instrumenting the code.

The guidelines for instrumenting both the SUA and CE are provided in Appendix A.

Using the above information, the user can work offline to prepare the required Digital Twin and instrument the Local Panner_1, Controlled Environment, and Digital Twin itself with suitable probes. As an example, Figure 43 shows the probe inserted into Local Panner_1.

```
public class SUAProbe extends ConcernAbstractProbe {
    public SUAProbe(Properties settings) {
        super(settings);
    }

    public static void main(String[] args) throws UnknownHostException, InterruptedException {
        //creating a probe
        SUAProbe aGenericProbe = new SUAProbe(
            ConnectionManager.createProbeSettingsPropertiesObject(
                "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
                "tcp://localhost:61616","system", "manager",
                "TopicCF","DROOLS-InstanceOne", false, "SUA probe",
                "it.cnr.isti.labsedc.concern.java.lang.javax.security.java.util",
                "vera", "griselda"));

        //sending events
        try {
            DebugMessages.line();
            DebugMessages.println(System.currentTimeMillis(), SUAProbe.class.getSimpleName(),"Sending SUA messages");

            sendConnectionEventMessage(aGenericProbe);
            sendDisconnectionEventMessage(aGenericProbe);

            sendVelocityMessage(aGenericProbe, 0.1f);

        } catch (IndexOutOfBoundsException | NamingException e) {} catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 43 Auditing Framework GUI: Probe Injection

The development of the models for the Digital Twins is provided using the DSL (see Figure 44). For creating the executable DT, the user creates a file with the extension that refers to the language (.nv3). Automatically the pop-out menu enables the usage of pre-

defined structures like packages and entities. The user then starts to write the behaviour by structuring it accordingly.

In BIECO UC 4, the user defines the behaviour of the Local Planner that guides the navigation of the robot. With every build, the code for the executable models is generated. The navigation module interacts with other entities. This interaction needs to be specified in the behavioural description.

Interactions are captured in dedicated constructs that gather information being exchanged. This information is later encapsulated within specific events monitored for conformity with the execution in the real world or controlled environment. Decision events will be those events that cross the architectural boundaries of an internal component, whereas events describe normal input/output interaction with components within the same architectural structure. In the end, the complete behaviour is declared, and corresponding models are created. These models are then packed as .jar files and executed in a Predictive Simulation environment.

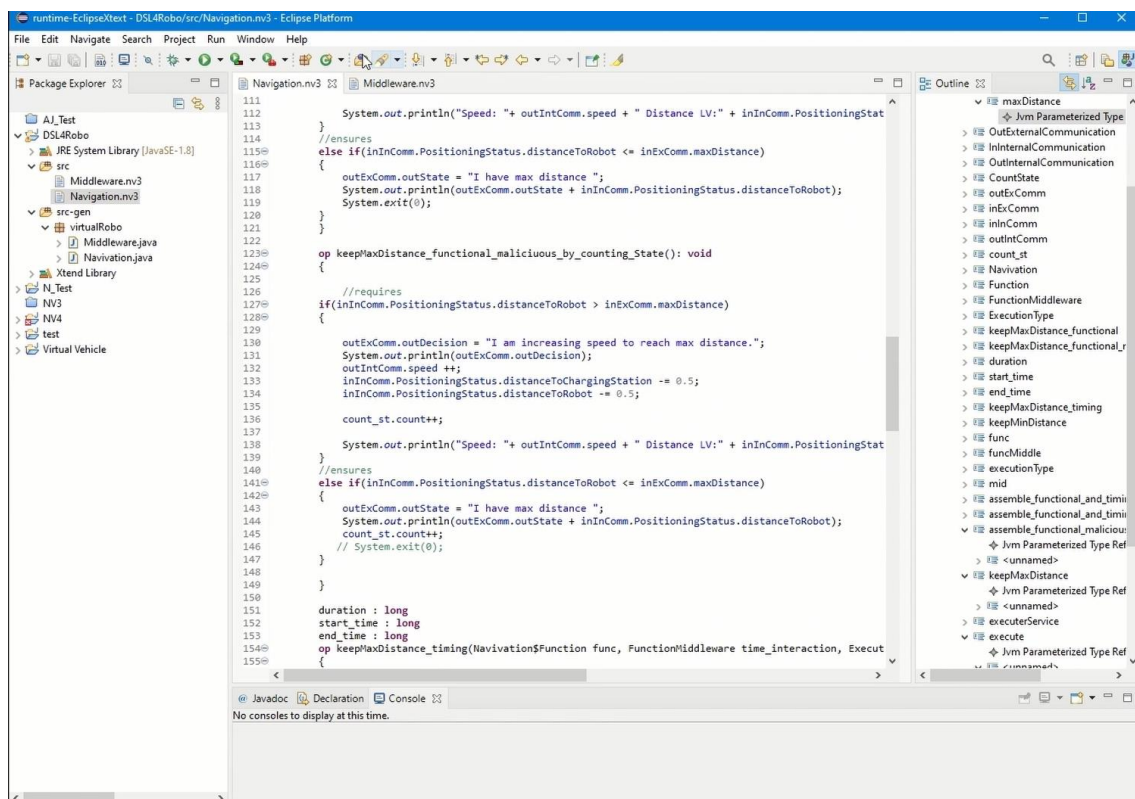


Figure 44 Digital Twin Eclipse Profile: Digital Twin development

8.1.3. Finish Pre-Setup

Through Runtime GUI, the user can finalize the auditing activity pre-setup phase. As shown in Figure 45, the Auditing Framework continues with the *Finish setup phase step*.

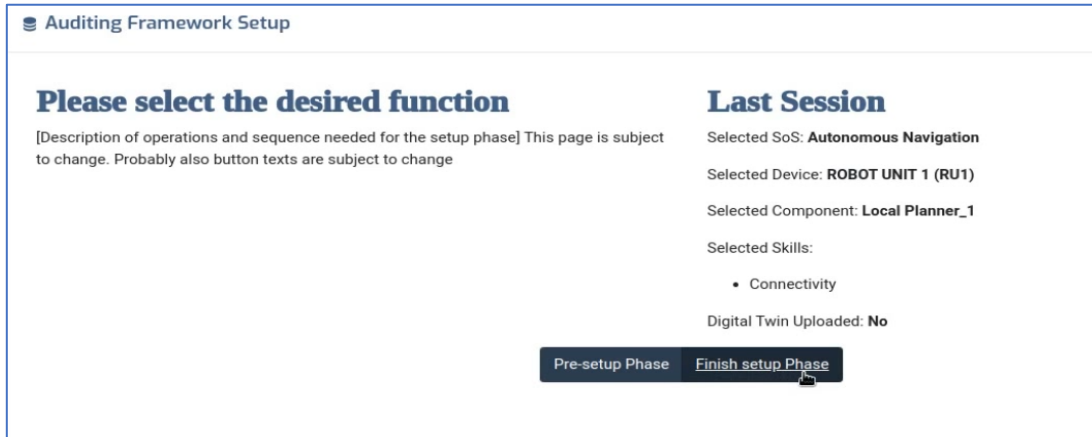


Figure 45 Auditing Framework GUI: Finish Pre-Setup phase

In these steps, the user first uploads the prepared Digital Twin, then finalizes the well-defined rule. This activity involves the Auditing Framework GUI and the Ontology Manager components. It focuses on the identifiers of the probes injected into SUA, Digital Twin, and Controlled Environment.

In Figure 46 and Figure 47, examples taken from the UC4 are shown. In this case, for the rule named "*Maximum number of established simultaneous connections*," the user inserts the identifier "SUA_PROBE"; for the pure predictive rule called "*Expected communication pattern through an ordered list of message types*" (see Figure 40) the user inserts "DT_PROBE."

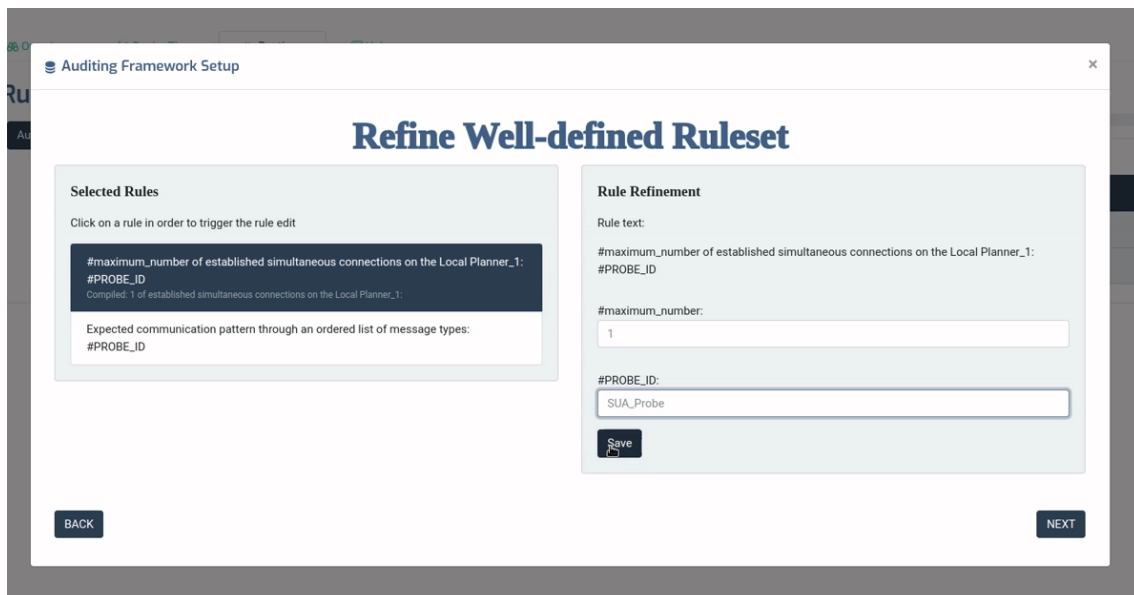


Figure 46 Auditing Framework GUI: Standard Well-defined rule refinement

Once finalized, the well-defined rules are translated into instantiated rules and provided to the Runtime Monitoring component for execution.

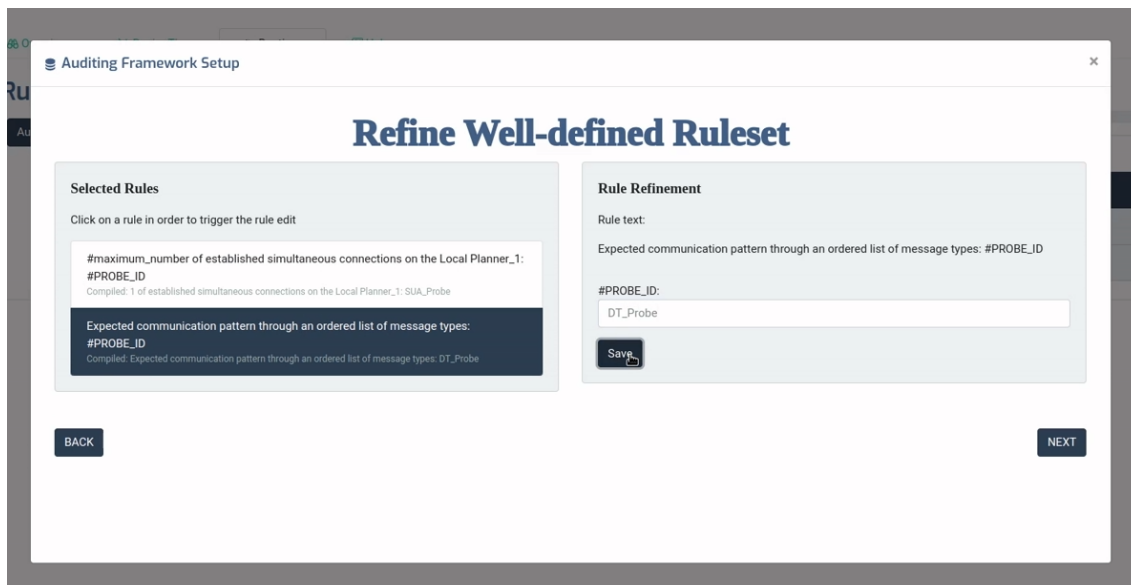


Figure 47 Auditing Framework GUI: Pure predictive Well-defined rule refinement

This ends the pre-setup phase, and the control goes back to the Runtime GUI to start the core of the Auditing Framework activity.

8.1.4. Start Auditing Framework

As soon as the user presses the “Start Auditing Framework” button in the Runtime GUI, a start command is sent by the Orchestrator to the Auditing Framework to begin the collaboration between the Runtime Monitoring and the Predictive Simulation Component. The start button caused the Runtime Monitoring to pass from the “online” state to “running” status. In this stage, the Predictive Simulation component is activated only in case one or more pure predictive rules have been selected. Otherwise, only the Runtime Monitor component is started.

In receiving the start command described in Section 3.2, the Runtime Monitoring raises the Complex Event Processor and accepts the instantiated rules defined in the Pre-Setup phase. Then, it compiles the rules into meta-rules and Instantiated rules.

At this point, the Runtime Monitoring starts listening to the events sent by the probes on a dedicated channel. As examples, Figure 48 and Figure 49 show the *SUA_Probes* injected in the *Local_planner_1* to send events related to velocity and score, respectively.

```

Runtime Monitoring
Status: Running
Monitoring logs:
-----
this.getMessage()
this.getDestinationID == "Monitoring",
this.getData == "established",
this.getConsumed == false,
this after $aEvent);
then
KielLauncher.printWarning("More than one connection between LocalPlanner and Global Planner established.\nShut down systems");
retract($aEvent);
retract($bEvent);
end and load it into the knowledgeBase
2022-04-13T11:01:54.732+0200 [ActiveMQ Session Task-2] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernBaseEvent in the stream, sent from SUA Probe
2022-04-13T11:01:54.732+0200 [ActiveMQ Session Task-2] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - with data:
Name: Connection
Destination: Monitoring
Data: established
SenderID: SUA Probe
Timestamp: 1649840514711
SessionID: sessionID
Checksum: noChecksum
CepType: DROOLS
2022-04-13T11:01:54.733+0200 [ActiveMQ Session Task-2] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernBaseEvent in the stream, sent from SUA Probe
2022-04-13T11:01:54.733+0200 [ActiveMQ Session Task-2] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - with data:
Name: Velocity
Destination: Monitoring
Data: 0.1
SenderID: SUA Probe
Timestamp: 1649840514727
SessionID: sessionID
Checksum: noChecksum

```

Figure 48 Trace of Connection message sent by SUA_Probe

In parallel, listening from the same channel, the Predictive Simulation, if previously activated, receives the events useful for fed abstract models.

These are executed faster than the Controlled Environment and can therefore provide predictions about trusted behaviour.

Considering the “*Expected communication pattern through an ordered list of message types*” rule, Figure 50 provides the DT_probe injected into the Digital Twin. It includes a DTForecast event containing a Digital Twin prediction regarding a velocity and score events sequence.

```

Runtime Monitoring
Status: Running
Monitoring logs:
-----
Destination: Monitoring
Data: 0.1
SenderID: SUA Probe
Timestamp: 1649840783810
SessionID: sessionID
Checksum: noChecksum
CepType: DROOLS
2022-04-13T11:06:24.112+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernBaseEvent in the stream, sent from SUA Probe
2022-04-13T11:06:24.113+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - with data:
Name: Score
Destination: Monitoring
Data: 0.1f
SenderID: SUA Probe
Timestamp: 1649840784111
SessionID: sessionID
Checksum: noChecksum
CepType: DROOLS
2022-04-13T11:06:24.613+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernBaseEvent in the stream, sent from SUA Probe
2022-04-13T11:06:24.614+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - with data:
Name: Velocity
Destination: Monitoring
Data: 0.1
SenderID: SUA Probe
Timestamp: 1649840784612
SessionID: sessionID
Checksum: noChecksum
CepType: DROOLS
2022-04-13T11:05:39.339+0200 [ActiveMQ Session Task-4] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received

```

Figure 49 Trace of Velocity messages sent by SUA_Probe

Using the data of the DTForecast event, the monitor instantiates the meta-rule into a new rule and injects it into the CEP.

```

Runtime Monitoring
Status: Running
Monitoring logs:
2022-04-13T11:06:24.613+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernBaseEvent in the stream, sent from SUA Probe
2022-04-13T11:06:24.614+0200 [ActiveMQ Session Task-6] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - with data:
Name: Velocity
Destination: Monitoring
Data: 0.1
SenderID: SUA Probe
Timestamp: 1649840784612
SessionID: sessionID
Checksum: noChecksum
CepType: DR00LS
2022-04-13T11:05:39.339+0200 [ActiveMQ Session Task-4] INFO it.cnr.isti.labsedc.concern.cep.DroolsComplexEventProcessorManager - ...CEP named InstanceOne received
it.cnr.isti.labsedc.concern.event.ConcernDTForecast in the stream, sent from DT_probe
.....
[received forecast time from DT]
.....
package it.cnr.isti.labsedc.concern.event;
import it.cnr.isti.labsedc.concern.event.ConcernAbstractEvent;
import it.cnr.isti.labsedc.concern.event.ConcernBaseEvent;
import it.cnr.isti.labsedc.concern.utils.KieLauncher;

dialect "java"

declare ConcernBaseEvent
    @role( event )
    @timestamp( timestamp )
end

rule "autogen-rule-noSession-DT_probe-rule"

```

Figure 50 Trace of forecast messages sent by DT_Probe

As shown in Figure 51, the boundary value for the period validity of the new rule is also provided. In this case, the boundary is set to 5 seconds, as established in the DTForecast event.

```

Runtime Monitoring
Status: Running
Monitoring logs:
.....
dialect "java"

when

    $0Event : ConcernBaseEvent(
        this.getName == "Velocity",
        this.getSenderID == "SUA Probe",
        this.getSessionID == "noSession");

    $1Event : ConcernBaseEvent(
        this.getName == "Velocity",
        this.getSenderID == "SUA Probe",
        this.getSessionID == "noSession",
        this after $0Event);

    $2Event : ConcernBaseEvent(
        this.getName == "Score",
        this.getSenderID == "SUA Probe",
        this.getSessionID == "noSession",
        this after $1Event);

    $3Event : ConcernBaseEvent(
        this.getName == "Velocity",
        this.getSenderID == "SUA Probe",
        this.getSessionID == "noSession",
        this after $2Event,
        this.getTimestamp() < ($0Event.getTimestamp+5000));

then

```

Figure 51 Trace of rule self-generated by the Runtime Monitoring

On the Runtime Monitoring side, it continuously receives the SUA_probe events containing Score, Velocity, and connection status and checks the set of instantiated rules. Predictive Simulator and Runtime Monitoring continue collaboration till the user decides to stop the auditing activity or as soon as a rule violation is experienced.

8.1.5. Validation Scenario

In the UC4 execution, a malicious code attack has also been simulated for validation purposes. Thus, a malicious code injection has been performed through the UI. As shown in Figure 52, this caused an increase in the connections between the *Local planner_1* and the Global planner.

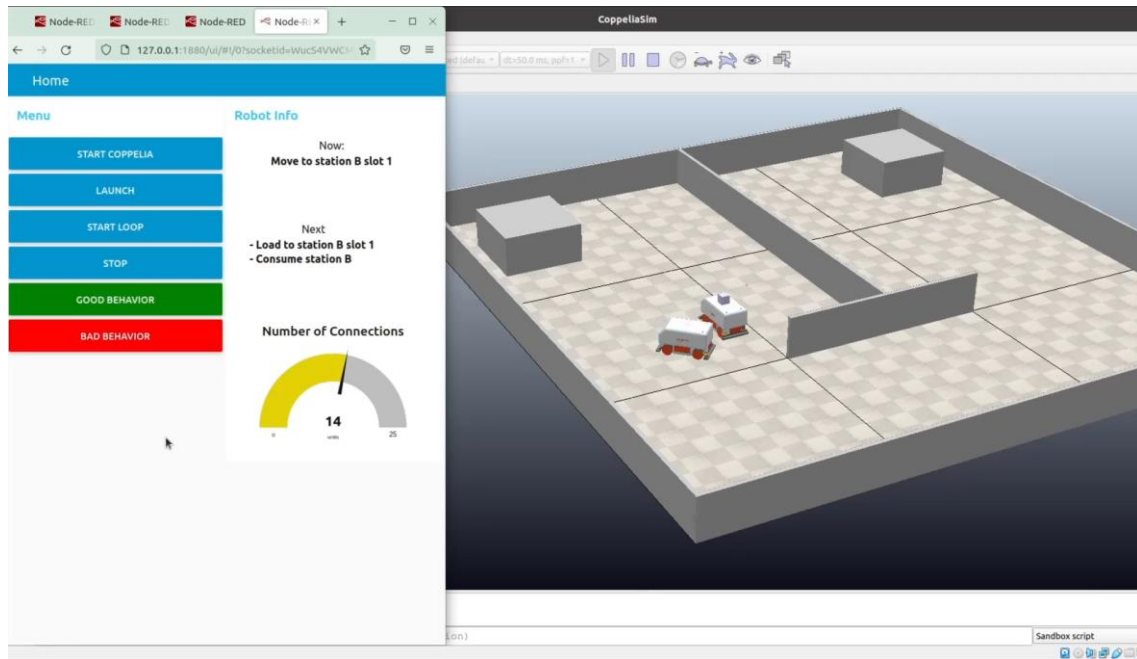


Figure 52 CoppeliaSimulator: Execution of a malicious code attack

Consequently, as in Figure 53, the Runtime Monitoring observed the violation of the “*Maximum number of established simultaneous connections*” rule, immediately notified the BIECO platform, and triggered the associate countermeasure. In case the Auditing Framework activity was stopped, the system’s dynamic configuration performed offline.



Figure 53 Runtime Monitoring Logger: Trace of rule violation raised

Once the system has been reconfigured, another round of the Auditing Framework execution has been performed. As shown in Figure 54, the system returned to a safe and trusted condition.

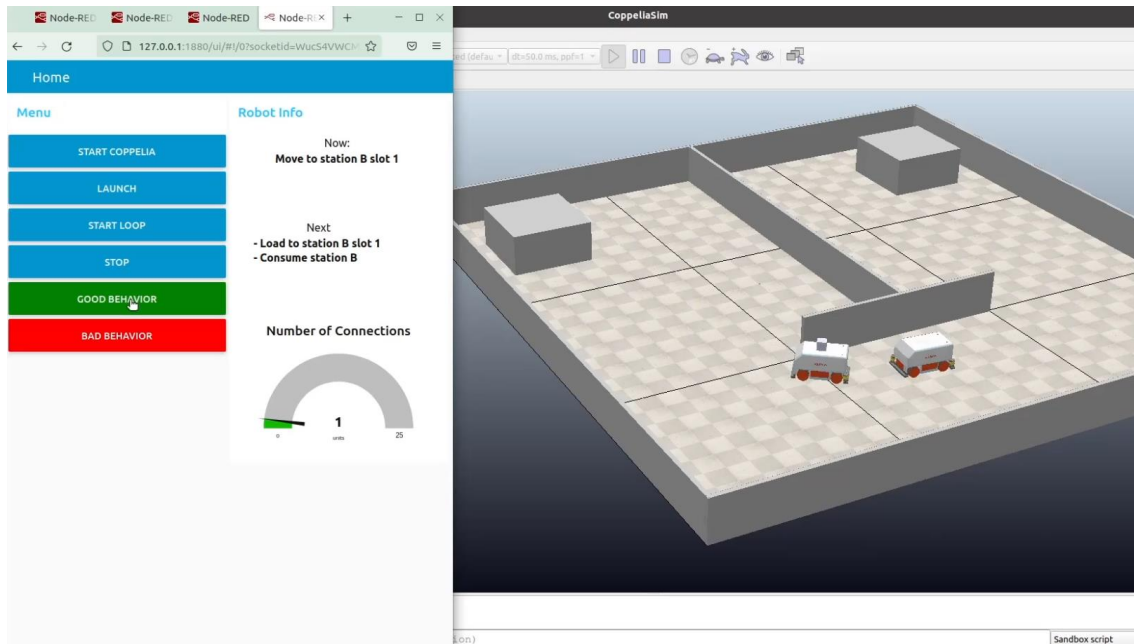


Figure 54 Coppelia Simulator: System turns back to a safe condition

9. Conclusions

The deliverable reported the Work Package 5 activities performed over the second years (M13 to M24) to improve and implement the Auditing Framework. Thus, it provided technical details about implementing the overall framework and its components. Additionally, it presented the (technological and research) advancements concerning the previous deliverable (D5.1 [11]). The validation of the Auditing Framework with the UC 4 - Coppelia has also been described.

The deliverable fulfilled all the future works listed in the previous deliverable, focused on the improvements of the proposed ontology, the implementation of the Auditing Framework and its components, the integration of the Blueprints, and the validation of the through the BIECO Use Cases.

As future general works, the following activities will be considered and reported in deliverable D5.3.

1. Implementation of the final version of the Auditing Framework and its components.
2. Validation of the Auditing Framework with all the BIECO Use cases.
3. Exploitation and dissemination of the Auditing Framework.

As specific future works, the following will be considered:

For the Runtime Monitoring: include features for using smart agents instead of the proposed probes. The new probe should be capable of sending formal events, receiving notifications from the CEP, and activating countermeasures. Therefore, the probe could be used to change its host's behaviour while running to reduce the risk of the detected violation. Solutions to be analysed are lowering a transmission rate to avoid collision or congestion; or executing an alternative activity during the system running.

For the Ontology Manager: Finalize the components' implementation by identifying the alternative open-source tools to be used and customized. This activity will also consider interoperability and extensibility to improve the overall implementation performance.

For the Predictive Simulation: explore how autoencoder-based classification of situations may be used for training and using models. Specific attention will be dedicated to investigating which requirements a learnt model should fulfil to be applied for Predictive Simulation and which opportunities and limitations exist. Additionally, the different possibilities for the DT derivation will be considered: (1) in-house, in parallel with the source code of the smart software agent by starting from the same specifications, (2) from the abstract behaviour of DTs, the source code of the smart software agent can be derived, or (3) in house, from the source code, specialized abstract models can be derived. In this last case, the possibility of extracting from AST (Abstract Syntax Tree) and CFG (Control Flow Graph) the meaningful information to be included in the definition of SDT (specialized Digital Twins) will be considered.

References

- [1] P. Antonante, D. I. Spivak and L. Carlone, Monitoring and diagnosability of perception systems, arXiv:2005.11816, 2020, [online] Available: <http://arxiv.org/abs/2005.11816>.
- [2] Paolo Barsocchi, Antonello Calabrò, Antonino Crivello, Said Daoudagh, Francesco Furfari, Michele Girolami, Eda Marchetti: A Privacy-By-Design Architecture for Indoor Localization Systems. QUATIC 2020, pp. 358-366.
- [3] Paolo Barsocchi, Antonello Calabrò, Erina Ferro, Claudio Gennaro, Eda Marchetti, Claudio Vairo: Boosting a Low-Cost Smart Home Environment with Usage and Access Control Rules. Sensors 18(6): 1886 (2018).
- [4] Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, Eda Marchetti: Towards Business Process Execution Adequacy Criteria. SWQD 2016, pp. 37-48.
- [5] Antonello Calabrò, Francesca Lonetti, Eda Marchetti, Giorgio Oronzo Spagnolo: Enhancing Business Process Performance Analysis through Coverage-Based Monitoring. QUATIC 2016 pp. 35-43.
- [6] Emilia Cioroica, Said Daoudagh, Eda Marchetti (2022) Predictive Simulation for Building Trust Within Service-Based Ecosystems. PerCom Workshops 2022, pp. 34-37.
- [7] C. Corbière, N. Thome, A. Bar-Hen, M. Cord and P. Pérez, Addressing failure prediction by learning model confidence, Proc. Adv. Neural Inf. Process. Syst., vol. 32, pp. 2902-2913, 2019.
- [8] Said Daoudagh, Eda Marchetti, Antonello Calabrò, Ana Inês Oliveira, Filipa Ferrada, Francisco Marques, José Barata and Ricardo Peres (2022) An Ontology-based Solution for Monitoring IoT Cybersecurity. 5th IFIP International INTERNET OF THINGS (IoT) Conference 2022.
- [9] Deliverable D2.4, Overall system architecture Update (Final), BIECO project.
- [10] Deliverable D4.2, Report on methods and tools for the failure prediction, BIECO project.
- [11] Deliverable D5.1, Definition of the Simulation Model and Monitoring Methodologies, BIECO project.
- [12] Deliverable D6.2, Blockly4SoS user guide, BIECO project.
- [13] Deliverable D6.3, Risk Assessment, and additional requirements, BIECO project.
- [14] Deliverable D6.4, Mitigations identification and their design, BIECO project.
- [15] Deliverable D7.2, Security certification methodology definition, BIECO project.
- [16] Deliverable D7.3, Security certification methodology development, BIECO project.
- [17] Deliverable D8.2, BIECO Assessment methodology, BIECO project.
- [18] S. Hecker, D. Dai and L. van Gool, Failure prediction for autonomous driving, Proc. IEEE Intell. Vehicles Symp. (IV), pp. 1792-1799, Jun. 2018.
- [19] S. Mohseni, A. Jagadeesh and Z. Wang, Predicting model failure using saliency maps in autonomous driving systems, arXiv:1905.07679, 2019, [online] Available: <http://arxiv.org/abs/1905.07679>.
- [20] S. Rabiee and J. Biswas, IVOA: Introspective vision for obstacle avoidance, arXiv:1903.01028, 2019, [online] Available: <http://arxiv.org/abs/1903.01028>.
- [21] Q. M. Rahman, P. Corke and F. Dayoub, Run-Time Monitoring of Machine Learning for Robotic Perception: A Survey of Emerging Trends, in IEEE Access, vol. 9, pp. 20067-20075, 2021, doi: 10.1109/ACCESS.2021.3055015.
- [22] Reich Jan, Schneider Daniel, Sorokos Ioannis, Papadopoulos Yiannis, Kelly et al. Engineering of Runtime Safety Monitors for Cyber-Physical Systems with Digital Dependability Identities, Computer Safety, Reliability, and Security 2020", Springer International Publishing, pp. 3–17.
- [23] D. M. Saxena, V. Kurtz and M. Hebert, "Learning robust failure response for autonomous vision-based flight", Proc. IEEE Int. Conf. Robot. Automat. (ICRA), pp. 5824-5829, May 2017.
- [24] Schneider, Daniel, and Trapp, Mario, Conditional Safety Certification of Open Adaptive Systems, ACM Transactions on Autonomous Adaptive, York, NY, USA July 2013, Vol 8, n.2 .

Appendix A. Runtime Monitoring Instrumentation Guidelines

A. Introduction

This document provides the methodologies that can be used for instrumenting software to be monitored through the Auditing Framework.

The instrumentation relies on the concept of Probe, which is a library that sends a selected set of data (like methods execution, variable value, and execution time values) to the Auditing Framework. The Runtime Monitoring receiving this data, and through analysis driven by rules, will be able to infer behavioural patterns or check the conformance of functional and non-functional properties.

To enhance compatibility and security, the System Under Audit (SUA) can be instrumented with Probes in several ways proposed in the following. A library that contains software artifacts for the instrumentation is also provided.

A.1 Overview

The simpler version of the probe in BIECO is represented by a piece of code capable of sending events according to a specific format.

This probe can send events regularly, or every time a specific situation occurs. An example of this probe is described in the Section A.4.

The following guidelines provide information for executing the Instrumentation process depicted in the figure below.

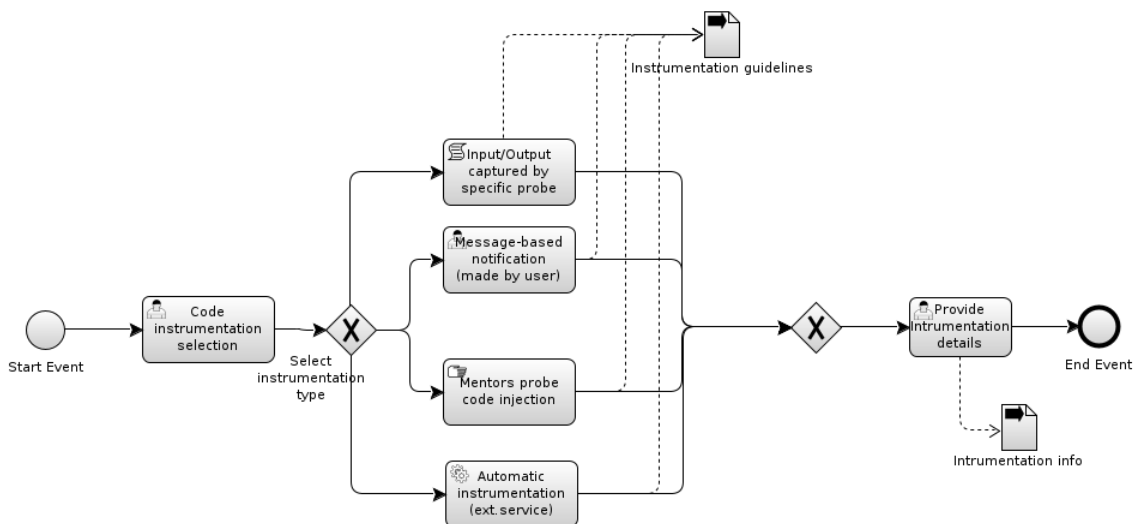


Figure 55 Code Instrumentation process

The user can choose among a bouquet of instrumentation mechanisms:

- **Input/Output captured by specific probe:** through this approach, the user must instrument/adapt its code with a mechanism that will provide the information needed for monitoring activities in a particular port/socket/channel. Once

offered, on the same port, an artifact capable of converting those raw data to the format understandable by the Runtime Monitoring must be configured to transform and send the data to the Runtime Monitoring. The artifact capable of reading from a stream and converting to Runtime Monitoring format is provided within the released library.

- **Message-Based notification (made by the user):** another approach lets the user instrument their device (or CE) with their mechanism to send messages directly to the message broker exposed by Runtime Monitoring. In this case, messages should be structured according to a predefined format and protocol. Details are provided in the Event Description section.
- **Mentors probe code injection:** in this case, the user can use a pre-build artifact provided in conjunction with these guidelines. For Java, this is represented by a jar library that can be directly used within the device (or CE) code for implementing event-message notifications.
- **Automatic instrumentation (ext. service):** The latter option is related to the possibility of exploiting an external service for device (or CE) instrumentation that can wrap components and trigger actions. At the same time, specific methods are invoked during the execution. In such a situation, overload risk should be evaluated because it could bias performances/communications. However, the message's structure must comply with what was described in Appendix B.

A.2 Input/Output Captured by the Specific Probe

The *Input/output captured by a specific probe* mechanism allows instantiating an external agent that reads data from a serial port, or a socket exposed by the device or system under audit on which it writes data useful for the monitoring activities.

An example of the data that can be sent on this port is shown in the following example:

```
#START#TIMESTAMP:12123;EVENTNAME:theName;EVENTDATA:data#END#
```

The external agent receiving a message like this will take care to convert it into the format described in the Appendix B and forward it to the monitoring platform.

This approach has been proposed by considering low-power devices or devices developed with software language that does not allow the creation of HTTP connections, rest, or MQTT channels for providing information directly to the Runtime Monitoring.

Depending on the implementation of this approach, some delays in notifications may be encountered; for this reason, developers must evaluate the efficiency of this approach.

The following Section proposes an example of this type of probe.

We are supposed to have a GPS device that provides data captured on a serial port.

The probe software will:

- connect on the GPS device com port.
- capture the raw data from the GPS device (in NMEA format).
- filter the part of interest (latitude and longitude).
- encapsulate it into a ConcernBaseEvent object message.
- sent it to the monitoring for the analysis.

```

package it.cnr.isti.labsedc.concern;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.jms.Topic;
import org.apache.activemq.ActiveMQConnectionFactory;
import com.fazecast.jSerialComm.SerialPort;
import it.cnr.isti.labsedc.concern.cep.CepType;
import it.cnr.isti.labsedc.concern.event.ConcernBaseEvent;

public class GPSProbe {
    static String deviceGPS = "ttyACM0";
    static SerialPort comPort;
    static OutputStream out;
    static String brokerUrl = "tcp://0.0.0.0:61616";
    public static String lastGPSpos = null;

    public static void main(String[] args) throws InterruptedException {
        loopThreadGPS();
    }

    private static void loopThreadGPS() {
        try {
            Process p = Runtime.getRuntime().exec("cat /dev/" + deviceGPS);
            new Thread(new Runnable() {
                public void run() {
                    System.out.println("GPS Probe started");
                    BufferedReader input =
                    new BufferedReader(new InputStreamReader(p.getInputStream()));
                    String line = null;
                    String[] results;
                    try {
                        while ((line = input.readLine()) != null)
                            if (line != null && line.startsWith("$GPGLL")) {
                                results = line.split(",");
                                if (results[6].compareTo("A") == 0) { //gps signal is valid
                                    testProbe(brokerUrl, "DROOLS-InstanceOne", "vera",
                                    "griselda", "Robot-TWO", results[1]+","+results[3]);
                                }
                            }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();

            p.waitFor();
        } catch (InterruptedException | IOException e1) {
            e1.printStackTrace();
        }
    }

    public static void testProbe(String brokerUrl,
        String topicName,

```

```

String username,
String password,
String eventData,
String eventName) {
    try {
        ConnectionFactory = new
        ActiveMQConnectionFactory(username, password, brokerUrl);
        Connection = connectionFactory.createConnection();
        Session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
        Topic = session.createTopic(topicName);
        MessageProducer producer = session.createProducer(topic);
        ObjectMessage msg = session.createObjectMessage();

        ConcernBaseEvent<String> event = new ConcernBaseEvent<String>(
            System.currentTimeMillis(),
            new Exception().getStackTrace()[1].getClassName(),
            "AuditingSystem-Monitoring", "sessionA",
            "checksum",
            eventName, eventData, CepType.DROOLS, false,"extension");

        msg.setObject(event);
        producer.send(msg);
    } catch (JMSEException e) {
        e.printStackTrace();
    }
}
}
}

```

Figure 56 Serial port probe example

A.3 Message-Based Notification (Made by the User)

The instrumentation based on *Message-Based notification made by the user* relies on the user self-implementation of the message notification mechanism.

The main constraint that users must respect for executing this approach are:

- create a secure connection to the message broker, for example, *tcp://BIECO.holisun.com:61616* using the security credentials provided.
- create an object Event according to what is described in Appendix B.
- mark the timestamp of this event once the events in the system under audit occurs.
- send messages using a mechanism that relies on JMS or through JSON format to the endpoint on which the monitoring is listening for post messages (see Section 3.2)

To clarify the mechanism of connection/disconnection and sending of a message, the developer can look at the code of SUAProbe, DTProbe, or the file ConcernAbstractProbe provided in the package.

A.4 Mentors Probe Code Injection

In conjunction with these guidelines, a jar artifact containing the code of a generic probe that can be used for the purpose is provided.

Opening the *mentorsProbe.jar* file as an archive into the folder */it/cnr/isti/labsedc/concern*, you can find a set of folders that defines the structure of the probe, as shown in the figure below.

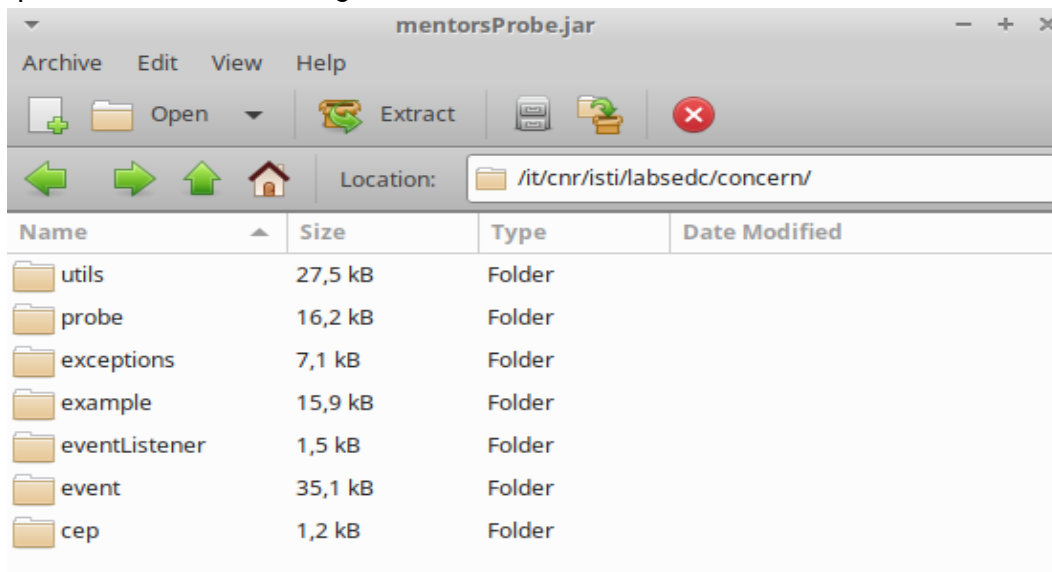


Figure 57 example folder

Two executable Java classes are reported in the example folder: *SUAProbe* and *DTProbe*.

SUAProbe represents an executable probe implementation related to a generic system under auditing; it inherits the constructor for the class *ConcernAbstractProbe* that contains the generic behaviour of a probe.

The usage of the *SUAProbe* example is shown in the example below:

```
SUAProbe          aGenericProbe          =          new          SUAProbe(
    ConnectionManager.createProbeSettingsPropertiesObject(
        "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
        "tcp://localhost:61616","system",
        "TopicCF","DROOLS-InstanceOne", false, "SUA_probe",
        "it.cnr.isti.labsedc.concern.java.lang.javax.security.java.util",
        "vera", "griselda"));
    "manager",
```

Figure 58 SUA probe example

Once the object is generated, the method inherited from the class *ConcernAbstractProbe* can be invoked for sending a generic *ConcernBaseEvent<T>*.

The method is called *sendMessage* contained within *ConcernAbstractProbe*.

```

protected void sendEventMessage(
ConcernBaseEvent<?> event, boolean debug)
throws JMSException, NamingException {
    if (debug) {
        DebugMessages.print(System.currentTimeMillis(),
        this.getClass().getSimpleName(),
        "Creating Message ");
    }

    try
    {
        ObjectMessage messageToSend =
publishSession.createObjectMessage();
        messageToSend.setJMSPayloadID(String.valueOf(MESSAGEID++));
        messageToSend.setObject(event);
        if (debug) {
            DebugMessages.ok();
            DebugMessages.print(System.currentTimeMillis(),
this.getClass().getSimpleName(),
        "Publishing message ");
        }
        mProducer.send(messageToSend);
        if (debug) {
            DebugMessages.ok();
            DebugMessages.line();
        }
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
}

```

Figure 59 ConcernAbstractProbe

Through these facilities, the user may decide to use the generic send events mechanism or to implement its own.

In the rest of the *SUAProbe* file, some methods have been presented to show the possibility of sending a specific type of event directly. In this case, the event *Score*, *Velocity*, *Connection*, and *Disconnection* related to the UC 4 already described have been presented:

- sendVelocityMessage(SUAProbe aGenericProbe, String speed).
- sendScoreMessage(SUAProbe aGenericProbe, String score).
- sendConnectionEventMessage(SUAProbe aGenericProbe).
- sendDisconnectionEventMessage(SUAProbe aGenericProbe).

Those method calls, or similar ones personalized according to the monitoring needs, must be placed in correspondence with the real event occurrence within the System Under Auditing.

The libraries allow the developer to implement all the connection and sending actions by itself. Figure below provides an example.

All the actions needed for establishing the connection are executed manually, creating a producer (the object capable of sending the message on a channel) and the sending actions.

```

package it.cnr.isti.labsedc.concern;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSEException;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.jms.Topic;
import org.apache.activemq.ActiveMQConnectionFactory;
import it.cnr.isti.labsedc.concern.cep.CepType;
import it.cnr.isti.labsedc.concern.event.ConcernBaseEvent;
import it.cnr.isti.labsedc.concern.event.ConcernNetworkEvent;

public class Probe {

    public static void testProbe(String brokerUrl, String topicName,
                                String username, String password,
                                String eventData, String eventName,
                                String extension, String checksum,
                                String sessionId, String sender,
                                String destination) {

        try {
            ConnectionFactory = new
                ActiveMQConnectionFactory(username, password, brokerUrl);
            Connection = connectionFactory.createConnection();
            Session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
            Topic = session.createTopic(topicName);
            MessageProducer producer = session.createProducer(topic);
            ObjectMessage msg = session.createObjectMessage();

            ConcernBaseEvent<String> event = new
                ConcernBaseEvent<String>(
                    System.currentTimeMillis(),
                    sender, destination, sessionId,
                    checksum, eventName, eventData,
                    CepType.DROOLS, false,extension);
            msg.setObject(event);
            producer.send(msg);
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}

```

Figure 60 Simple Probe

A.5 Automatic Instrumentation (ext. Service)

The *Automatic instrumentation (ext.service)* mechanism relies on an external instrumentation service for capturing events occurring in the device (or CE).

Some (free) services provide features like wrapping components or triggering specific calls when a method is called/executed. Those services will put an overload to the Device under test and may bias performances/communications.

Information captured by probes is then sent to the Runtime Monitoring using the mechanism provided in *Input/output captured by a specific probe* section. In Figure

below shows an example of one of the possible tools that can be used to instrument your code: *JProfiler*³⁰.

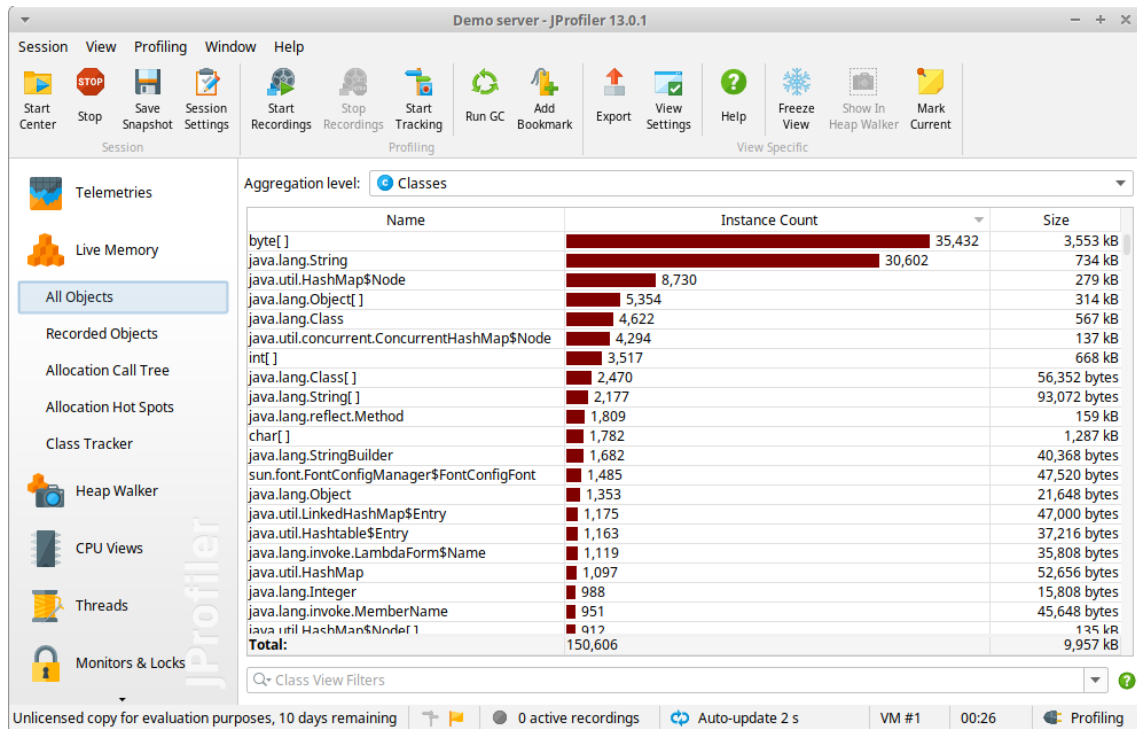


Figure 61 Specific instrumentation tool

B. Event Description

The figure below details the representation of an event compatible with the complex event processor deployed within the BIECO Auditing Framework.

³⁰ <https://www.ej-technologies.com/products/jprofiler/overview.html>

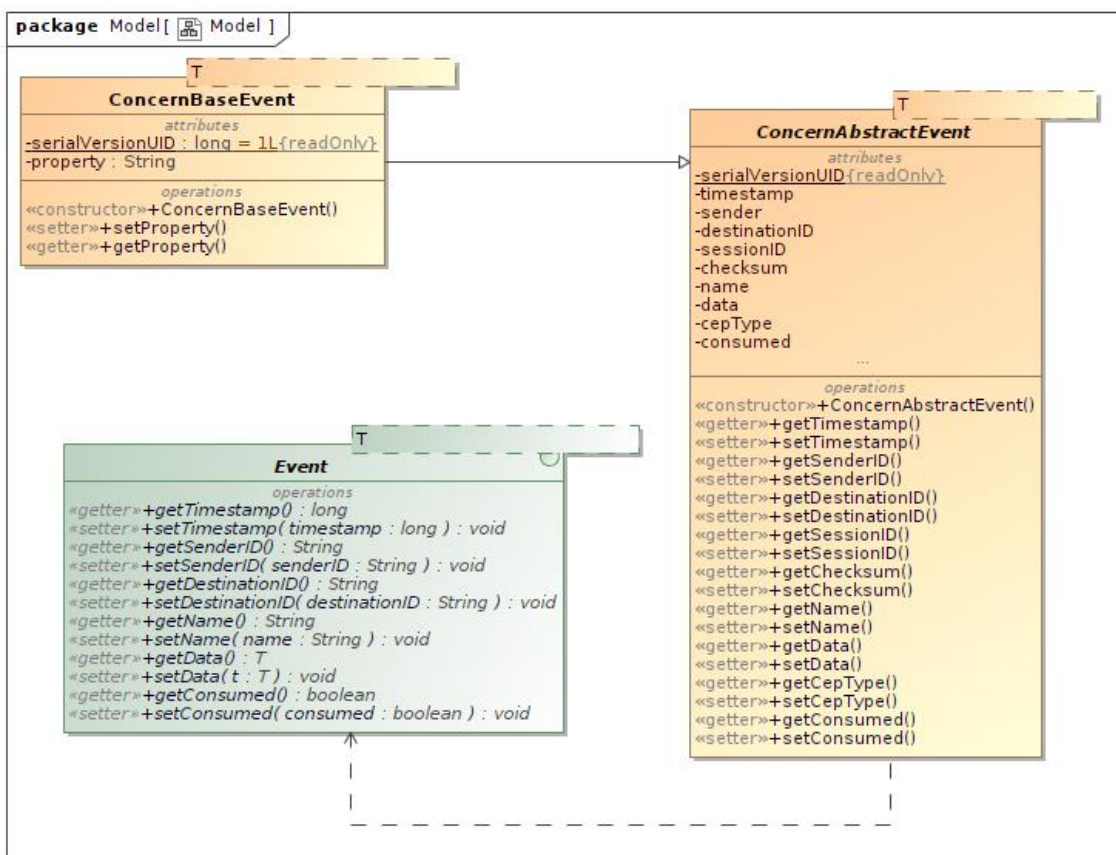


Figure 62 Specific BIECO event

The interface *Event* proposes a set of parameters and methods signatures for setting it up.

The *ConcernAbstractEvent* abstract class represents a usage of the event with the minimum subset of parameters needed.

Those parameters can be extended with the classical extension procedure of the Java programming language, allowing users to include more parameters that can be interesting for the analysis. In the *ConcernBaseEvent* example, the property parameter has been included to extend the *ConcernAbstractEvent*.

Below, more details about each field of the event object are provided.

- timestamp: the instant the event is generated on the been captured.
- senderID: the identifier of the sender.
- destinationID: the identifier of the receiver (for example, monitoring).
- sessionId: the identifier of the monitoring session.
- checksum: the computed Cyclic Redundancy Check of the payload.
- name: the name of the probe that is sending the data.
- data: the payload to notify, the format can be set up according to the parameter T of the *ConcernBaseEvent*<T>.
- cepType: the identifier of the Complex Event Processor (CEP) that should process this data: possible types are ESPER, DROOLS, and Enum *CepType* is available.
- consumed: a parameter used by the CEP for analysing the events.