

Project Number 723634

D5.8 Final Integrated Cloud Analysis and Reconfiguration Platform

**Version 1.0
31 October 2019
Final**

Public Distribution

ATB, IKERLAN, The Open Group, University of York

Project Partners: ATB, Electrolux, IKERLAN, OAS, ONA, The Open Group, University of York

Every effort has been made to ensure that all statements and information contained herein are accurate, however the SAFIRE Project Partners accept no liability for any error or omission in the same.

© 2019 Copyright in this document remains vested in the SAFIRE Project Partners.

PROJECT PARTNER CONTACT INFORMATION

| | |
|--|---|
| ATB Sebastian Scholze Wiener Straße 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de | Electrolux Italia Claudio Cenedese Corso Lino Zanussi 30 33080 Porcia Italy Tel: +39 0434 394907 E-mail: claudio.cenedese@electrolux.it |
| IKERLAN Trujillo Salvador P Jose Maria Arizmendiarieta 20500 Mondragon Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es | OAS Karl Krone Caroline Herschel Strasse 1 28359 Bremen Germany Tel: +49 421 2206 0 E-mail: kkrone@oas.de |
| ONA Electroerosión Jose M. Ramos Eguzkitza, 1. Apdo 64 48200 Durango Spain Tel: +34 94 620 08 00 jramos@onaedm.com | The Open Group Scott Hansen Rond Point Schuman 6, 5 th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org |
| University of York Leandro Soares Indrusiak Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325 570 E-mail: leandro.indrusiak@york.ac.uk | |



DOCUMENT CONTROL

| Version | Status | Date |
|---------|--|-------------------|
| 0.1 | Template creation | 1 July 2019 |
| 0.5 | Initial draft with all components | 18 September 2019 |
| 0.7 | Updates and further integration of content | 27 September 2019 |
| 0.8 | Minor updates from BC demonstrators | 4 October 2019 |
| 0.9 | Internal review version | 24 October 2019 |
| 1.0 | Final version | 31 October 2019 |

TABLE OF CONTENTS

| | |
|--|-----------|
| 1. Introduction..... | 1 |
| 1.1 Overview..... | 1 |
| 1.2 Document Structure..... | 1 |
| 2. SAFIRE Integrated Cloud Platform Functionality Description | 2 |
| 2.1 Overview of SAFIRE platform..... | 2 |
| 2.2 Description of the final SAFIRE platform features | 3 |
| 3. User Manual for the SAFIRE Platform | 4 |
| 3.1 Predictive Analytics..... | 4 |
| 3.1.1 Introduction..... | 4 |
| 3.1.2 Service access via MQTT | 4 |
| 3.1.3 Service access via REST Web Service | 12 |
| 3.1.4 Uploaded models | 13 |
| 3.1.5 Training new models..... | 13 |
| 3.2 Situational Awareness | 13 |
| 3.2.1 Situation Model..... | 13 |
| 3.2.2 Situation Monitoring | 14 |
| 3.2.3 Situation Determination | 19 |
| 3.3 Optimisation | 20 |
| 3.3.1 Generating FDL templates | 20 |
| 3.3.2 FDL template for ONA BC..... | 21 |
| 3.3.3 FDL template for OAS BC | 25 |
| 3.3.4 FDL template for Electrolux BC..... | 27 |
| 3.4 Security Framework | 29 |
| 3.4.1 Components of the Security Framework implementation..... | 30 |
| 3.4.2 Declarative Policy Language | 31 |
| 3.4.3 Using the 'ngac' Policy Tool | 32 |
| 3.4.3.1 Developing and testing policies | 32 |
| 3.4.3.2 Policy Tool interactive commands..... | 33 |
| 3.4.4 Using the 'ngac-server' Policy Server | 34 |
| 3.4.4.1 Policy Server startup options | 34 |
| 3.4.4.2 Policy Query API..... | 35 |
| 3.4.4.3 Policy Administration API..... | 36 |
| 3.4.4.4 Dynamic Policy Change | 38 |
| 3.4.4.5 Policy composition..... | 38 |
| 3.4.5 Policy Enforcement Point (PEP) / Resource Access Point (RAP) design pattern..... | 39 |
| 3.4.5.1 Policy Enforcement, PEP-to-RAP, and Resource Access APIs..... | 39 |
| 3.4.5.2 Web service PEPs | 40 |
| 3.4.5.3 | 40 |
| 3.4.6 Platform protections needed to achieve non-functional properties of TOG-NGAC | 40 |
| 3.4.6.1 Nonfunctional Reference Monitor properties | 40 |
| 3.4.6.2 NGAC component and interaction integrity | 41 |
| 3.4.6.3 Reliable identities..... | 42 |
| 4. Installation and Configuration of the SAFIRE Platform | 43 |
| 4.1 Installation and Configuration of the basic SAFIRE architecture components | 44 |
| 4.1.1 Apache NiFi..... | 44 |
| 4.1.2 Apache Kafka..... | 45 |
| 4.1.3 Docker, Docker Compose and Docker Registry | 49 |
| 4.1.3.1 Docker..... | 50 |
| 4.1.3.2 Docker Compose..... | 50 |
| 4.1.3.3 Docker Container Registry..... | 51 |
| 4.1.4 Security Framework..... | 51 |
| 4.1.4.1 Prerequisites and Dependencies..... | 51 |



| | | |
|-----------|---|-----------|
| 4.1.4.2 | Installation and build..... | 52 |
| 4.1.4.3 | Testing and Using NGAC | 53 |
| 4.1.4.4 | Operation of the Policy Server | 55 |
| 4.1.4.5 | Configuration of NGAC | 55 |
| 4.1.4.6 | Guidance for achieving non-functional properties for TOG-NGAC..... | 55 |
| 4.2 | <i>Installation and Configuration of the SAFIRE Dashboard</i> | <i>57</i> |
| 4.3 | <i>Installation and Configuration of Data Ingestion and Monitoring Services</i> | <i>59</i> |
| 4.4 | <i>Installation and Configuration of the Predictive Analytics Service</i> | <i>59</i> |
| 4.5 | <i>Installation and Configuration of the Situational Awareness Services</i> | <i>60</i> |
| 4.6 | <i>Installation and Configuration of the Optimisation Engine</i> | <i>62</i> |
| 5. | Conclusions..... | 64 |
| 6. | References..... | 65 |

TABLE OF FIGURES

| | |
|--|----|
| Figure 2-1 Main components of SAFIRE | 2 |
| Figure 3-1 Clients Registering/Unregistering in PA Service | 5 |
| Figure 3-2 Client / PA Service interaction use-case diagram (Ona) | 6 |
| Figure 3-3 Client / PA Service interaction use-case diagram (Electrolux) | 6 |
| Figure 3-4 Collection of Samples to be evaluated | 10 |
| Figure 3-5 PA Service produces a prediction per sample | 10 |
| Figure 3-6 Collection of predictions | 11 |
| Figure 3-7 NGAC - Functional Architecture | 31 |
| Figure 3-8 PEP / RAP design pattern for the resource access path..... | 39 |
| Figure 4-1 SAFIRE FICP Deployment Diagram | 43 |
| Figure 4-2 Confirmation of the successful ZooKeeper instalation with netstat | 46 |
| Figure 4-3 Possible unsuccessful execution of Kafka with Java 9 | 47 |
| Figure 4-4 Creation of a topic in Kafka | 47 |
| Figure 4-5 Listing Kafka topics | 48 |
| Figure 4-6 Creating Kafka producer in a console | 48 |
| Figure 4-7 Creating Kafka consumer in a console..... | 49 |
| Figure 4-8 SAFIRE Dashboard Screenshot | 58 |
| Figure 4-9 Deployment diagram of the Predictive Analytics Service | 59 |
| Figure 4-10 Deployment diagram of the Situational Awareness service | 60 |

EXECUTIVE SUMMARY

This deliverable presents the Final Integrated Cloud Platform of the SAFIRE project and describes the functionality of the integrated full prototypes.

The Final Integrated Cloud Platform (FICP) completes the previous early version with the description of the full prototype of the SAFIRE services, as well as with details for the configuration and installation of the different modules in the business case site.

The SAFIRE integrated cloud platform consists of the four main SAFIRE services, namely the Predictive Analytics (PA), the Situation Determination (SD), the Optimisation Engine (OE) and the Security Framework (SPT), and its operation is being monitored using the dashboard. All the modules have been developed using latest open source technologies, adequate for big data management in an industrial environment, and have been packaged using docker in order to be directly deployable in different operating systems. The data transfer within the SAFIRE modules and outside to its environment (and the business case legacy systems) is being utilised using NiFi templates. The communication between the modules has been established using the kafka messaging system. This technology is also being used by the dashboard to visualise the results of the module during the SAFIRE operation.

For each of the three business cases, for Electrolux, OAS and ONA, the SAFIRE integrated cloud platform has been configured and connected or integrated with the legacy systems for data exchange and process management. User installation and configuration guidelines have been provided to allow future users of SAFIRE to integrate the platform into their systems.

1. INTRODUCTION

1.1 OVERVIEW

The present deliverable describes the SAFIRE Final Integrated Cloud Analysis and Reconfiguration Platform (FICP), which includes SAFIRE adapted functionalities based on the main services (i.e. Situation Determination, Predictive Analytics and Reconfiguration & Optimisation) into a unique cloud platform that shares common features like data ingestion, message handling, security measures, distributed access control, monitoring or auditing. This cloud version of the platform implements SAFIRE concept in the cloud as this was adjusted after the validation of the preliminary version.

The document also describes the architecture of the SAFIRE services and provides step by step guidelines to install and configure the SAFIRE platform. It also describes how different instances of the FICP have been installed and configured at each of the project's business cases.

1.2 DOCUMENT STRUCTURE

The current deliverable is structured as follows:

- *Section 2* presents an overview of the services and features included in SAFIRE FICP, and how they are implemented at each of the business cases.
- *Section 3* provides the user manual of the SAFIRE FICP.
- *Section 4* provides guidelines on how to deploy the SAFIRE platform. It describes the installation and configuration procedures for the different modules of the SAFIRE platform as generic components.
- *Section 5* presents the main conclusions reached by the consortium in the integration of the SAFIRE platform.

2. SAFIRE INTEGRATED CLOUD PLATFORM FUNCTIONALITY DESCRIPTION

2.1 OVERVIEW OF SAFIRE PLATFORM

SAFIRE offers a solution that gathers data streams from the products/machines and their contexts to proactively make recommendations leading to the enhancement of the product performance, or the optimisation and reconfiguration of the production processes. These data streams provide important insights into the specific requirements for the factory infrastructure and configuration, and new opportunities for the improvement of products/machines, which implicitly has an enormous impact on the user satisfaction by improving the using experience of the product/machine.

SAFIRE has been implemented following the general architectural model in the picture below.

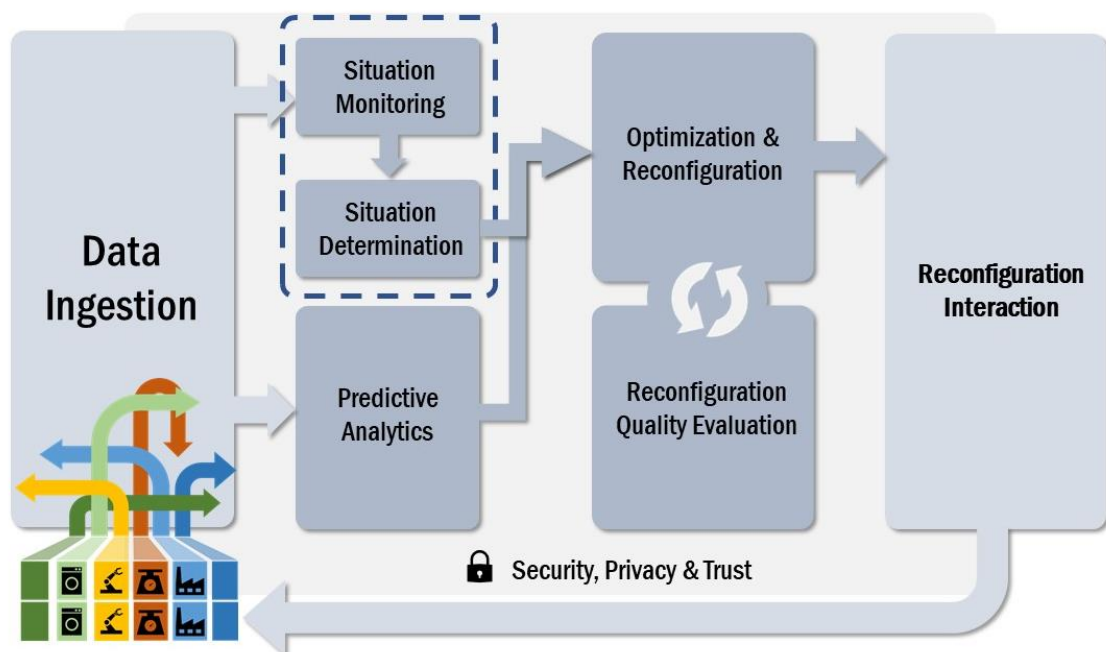


Figure 2-1 Main components of SAFIRE

As it can be observed, Data Ingestion services provide data to the Situational Determination Services (composed by Situation Monitoring and Situation Determination) and the Predictive Analytics Services. In general terms, data will be acquired through the Apache NiFi (<https://nifi.apache.org/>), a popular distributed streaming platform. NiFi is a solution that enables data routing and transformation, as well as the implementation of mediation logic among systems. In SAFIRE is used for the ingestion of data from the manufacturing systems and products to be optimised and reconfigured. Using this technology, data captured are being made available to all SAFIRE Services.

SAFIRE FICP implements all the components, including Apache NiFi and Kafka, as stand-alone services for data ingestion and exchange.

2.2 DESCRIPTION OF THE FINAL SAFIRE PLATFORM FEATURES

The SAFIRE solution provides a highly customisable software platform for production systems and smart products, offering:

1. Both reactive and predictive reconfiguration for both production systems and smart products
2. Flexible run-time reconfiguration decisions during production rather than pre-planned at production planning time
3. Real-time reconfiguration decisions for optimisation of performance and real-time production and product functions.

The SAFIRE project targets two related technology challenges for smart factories that present new opportunities for improving production, products and services:

1. Interconnected Systems of Production Systems (SoPS) within smart manufacturing environments, where individual production systems and the SoPS as a whole, have hardware (HW) and software (SW) requirements to be addressed to achieve specific business objectives, such as scheduling, power consumption, throughput, and maintenance.
2. Connected Product Networks (CPNs) where networked smart products collect data, can be adapted in the field, and can deliver extended services to customers through optimisation of smart product performance parameters and customisation of products to environments, usage patterns and other dynamic factors.

The advanced analytics and reconfiguration capabilities developed within SAFIRE are based on mastering the big data challenges associated with manufacturing (sensor and process data), enterprise and smart product data, to allow manufacturers to address production-system-behaviour forecasting, and to establish optimisation methods that are integrated in the design and product chain. Furthermore the platform provides big data analytic capabilities that meet real-time requirements such that dynamic run-time reconfiguration decisions are made during production time rather than pre-planned at production-planning time.

The following technologies and innovations developed in the SAFIRE project are supporting the above mentioned targets:

- New techniques for reconfiguration and optimisation of production systems and products based on predictive big data analytics of data generated by exploiting situational awareness during production and product use.
- A set of tools and services to support:

- Dynamic and predictable reconfiguration and optimisation
 - Predictive big data analytics
 - Cloud resource management
- New cloud based secure infrastructure for reconfiguration and optimisation of production systems / smart products.

The SAFIRE infrastructure is targeted to be provided as an add-on for an existing production system, or next generation smart factory operating system, allowing production systems to be transformed to include capabilities for dynamic real-time reconfiguration and optimisation.

3. USER MANUAL FOR THE SAFIRE PLATFORM

This section provides for every feature of the SAFIRE platform a brief overview of how to use and customize the specific component implementing this feature:

- Predictive Analytics (section 3.1)
- Situational Awareness (section 3.2)
- Optimisation (section 3.3)
- Security Framework (section 3.4)

An overview on how to install and configure the SAFIRE platform will be given in section 4.

3.1 PREDICTIVE ANALYTICS

3.1.1 Introduction

After installation and configuration, *PA Service* in the cloud can be accessed in two ways, via MQTT and via REST Web Service. The following sections describe how to invoke PA Service for requesting predictions.

3.1.2 Service access via MQTT

Introduction

PA Service can be accessed via MQTT broker that can be found in amazon cluster in the following address:

- **Address:** ec2-34-247-12-59.eu-west-1.compute.amazonaws.com
- **Username:** elec
- **Password:** safireproject
- **Port:** 1883

A client must connect to the MQTT broker and interact with the *PA Service* via publishing messages to topics (send message) and subscribing to topics to receive answers (receive message).

Topic Management

In order to interact with *PA Service*, a client must Register/Unregister via a common topic for all clients as shown in Clients Registering/Unregistering in PA Service (this figure shows an example with Electrolux BC).

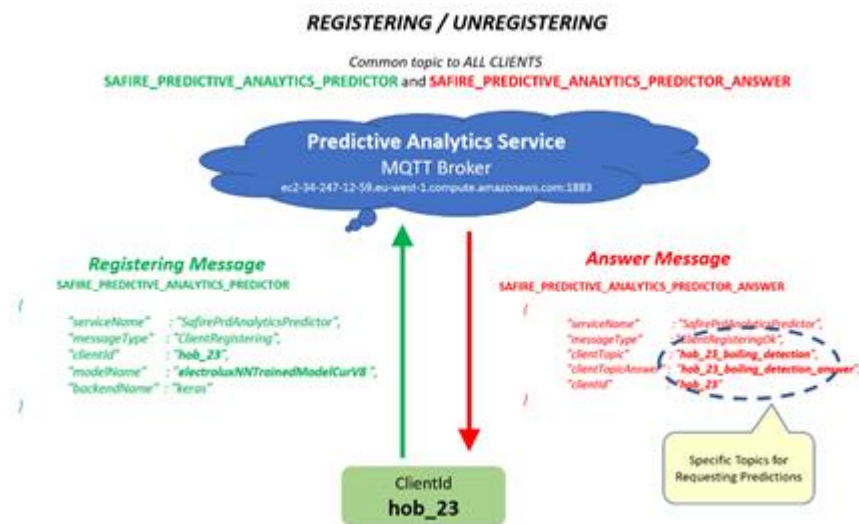


Figure 3-1 Clients Registering/Unregistering in PA Service

In this case, the client is a *hob* from Electrolux (id: hob_23) that sends a JSON message to the *PA Service* (described later in detail) and receives back an answer about the registration. Common topics for Register/Unregister are the following:

- Registering/Unregister – send message to topic:
SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR
- Answer Registering/Unregister – receive message through subscription to:
SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR_ANSWER

In the answer to the registering message, the client receives two **private topics** that will be used for requesting predictions (in Clients Registering/Unregistering in PA Service these private topics are *hob_23_boiling_detection* for further requests and *hob_23_boiling_detection_answer* for the answers)

Typical Use-Case Flow Diagram

Figure 3-2 shows a typical use-case flow diagram of a client, composed by three phases: (1) *registering*, (2) *requesting predictions* and (3) *unregistering*. In this example an

Ona cutting machine (id: *wedm_12.12.345*) client may be requesting predictions of *thickness change event*.

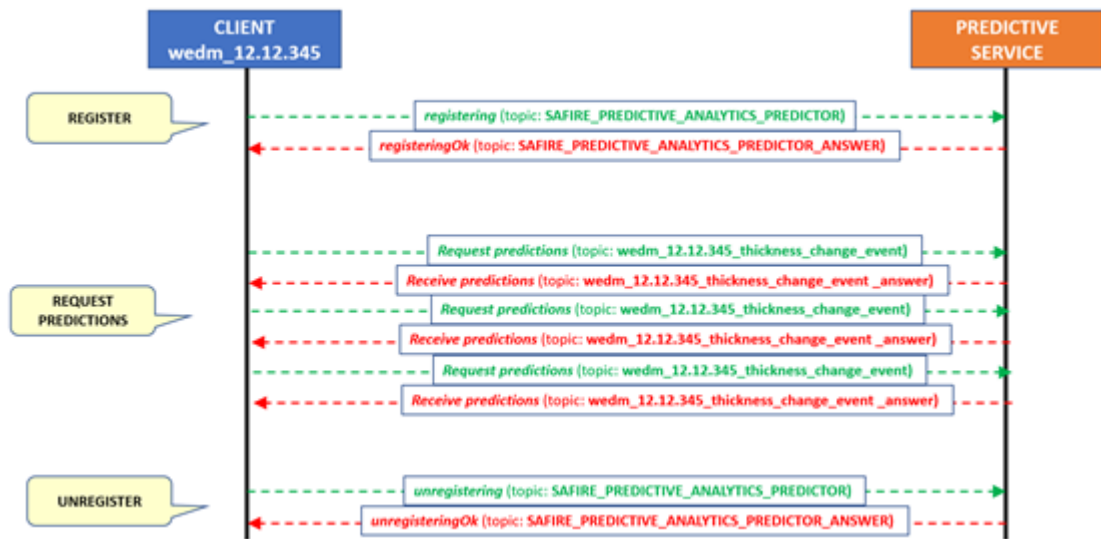


Figure 3-2 Client / PA Service interaction use-case diagram (Ona)

After (1) a registering process (register and wait to receive a registering ok answer via common topics), (2) the client requests and receive predictions and (3) finally, after interacting with PA Service the client unregisters.

As another example, in Figure 3-3 a hob in Electrolux may be requesting boiling predictions.



Figure 3-3 Client / PA Service interaction use-case diagram (Electrolux)

JSON Message to Register

The message to be sent to MQTT is a JSON message with the following format and must be published to the fix topic **SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR**. Two cases (Electrolux and Ona) are shown for clarity:

- *Electrolux* example:

```
{
  "serviceName" : "SafirePrdAnalyticsPredictor",
  "messageType" : "ClientRegistering",
  "clientId" : "hob_2341",
  "modelName" : "electroluxNNModelCurFeaturesF04",
  "backendName" : "keras"
}
```

- *Ona* example:

```
{
  "serviceName" : "SafirePrdAnalyticsPredictor",
  "messageType" : "ClientRegistering",
  "clientId" : "wedm_12.12.345",
  "modelName" : "onaNNOsciloThicknessModel_OneVsRest_5",
  "backendName" : "keras"
}
```

Fields in **bold** are specific to the client. *ClientId* must uniquely identify the client, *modelName* specifies the predictive model to be loaded (in this case a *h5 neural network) and *backendName* specifies the predictive backend that the PA Service must invoke to produce predictions (in this case *Keras*).

Another example of requests, shown below, corresponds to a client registering to request predictions of an **spark** model called **receiptboilingmodel** that predicts receipts cooking times.

```
{
  "serviceName" : "SafirePrdAnalyticsPredictor",
  "messageType" : "ClientRegistering",
  "clientId" : "hotel_1",
  "modelName" : "receiptboilingmodel",
  "backendName" : "spark"
}
```

Notes:

- *clientId* , in this case **hotel_1**, may be any string identifying uniquely the client (for example, identifiers like *eu.de.bremen.hotels.hotel421* are also valid).
- *modelName* , represents the name of machine learning trained model. In this case **receiptboilingmodel** represents an spark's model.

JSON Answer Message received to confirm Registering

PA Service answers to the Registering message with another JSON message as the one shown below. This message is received via subscription in fix topic:

SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR_ANSWER

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"       : "ClientRegisteringOk",
  "clientId"          : "hob_2341",
  "clientTopic"       : "hob_2341_boil_detection",
  "clientTopicAnswer" : "hob_2341_boil_detection_answer"
}
```

If *PA Service* is able to load the *modelName* (for example a neural network) with the specified *backendName* it returns a **ClientRegisteringOk** type message (as shown above), otherwise returns a **ClientRegisteringFail** type message. In the answer message, if registering was ok, *clientTopic* and *clientTopicAnswer* fields contain the private topics to be used by the client for prediction requests.

As another example, answer message shown below represents the answer to the registering message to request predictions of an **spark** model called **receiptboilingmodel** that predicts receipts cooking times.

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"       : "ClientRegisteringOk",
  "clientId"          : "hotel_1",
  "clientTopic"       : "hotel_1_cooking_scheduling",
  "clientTopicAnswer" : "hotel_1_cooking_scheduling_answer"
}
```

Notes:

- When receiving the answer, *PA Service* provides two specific topics (private to the client), in this case, **hotel_1_cooking_scheduling** and **hotel_1_cooking_scheduling_answer**. First topic will be used by client to request predictions (publish). Second topic will be used by client to receive answers with the predictions (subscribe).
- When receiving the answer, *messageType* field value may be:
 - *ClientRegisteringOk* – Registering was Ok.
 - *ClientRegisteringResetOk* – The client was already registered. Re-registering was ok.
 - *ClientRegisteringFail* – Registering failed.

JSON Message to Unregister

The message to be sent to MQTT is a JSON message with the following format and must be published to the fix topic **SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR**.

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
```



```
"messageType"      : "ClientUnregistering",
"clientId"         : "hob_2341"
}
```

JSON Answer Message received to confirm Unregistering

PA Service answers to the Registering message with another JSON message as the one shown below. This message is received via subscription in fix topic:

SAFIRE_PREDICTIVE_ANALYTICS_PREDICTOR_ANSWER

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"       : "ClientUnregisteringOk",
  "clientId"         : "hob_2341"
}
```

Notes:

When receiving the answer, *messageType* field value may be:

- *ClientUnregisteringOk* – Unregistering was Ok.
- *ClientUnregisteringResetOk* – The client was already unregistered. Re-unregistering was ok.
- *ClientUnregisteringFail* – Unregistering failed.

JSON Message to Request Predictions

The message to be sent to MQTT is a JSON message with the following format and must be published into the private topic received in the registering message (for example in topic *hob_2341_boil_detection*)

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"       : "ClientRequestingPrediction",
  "timeStamp"        : "1517927276069",
  "clientId"         : "hob_2341",
  "dataFrameColNames" : ["Time [s]","Cur_F08 [A]","Cur_F09 [A]","Cur_F10 [A]"],
  "dataFrameColTypes" : ["double","double","double","double"],
  "dataFrameRowData" : [[0.0, 52.60463, 49.25544, 46.69432]
                        [0.7, 45.345, -1.45, 0.09]]
                        [0.6, 0.5, 0.4, 0.3]]
}
```

Requesting predictions involves passing to the *PA Service* a **collection of samples** for which the predictive model has to produce a prediction (the *PA Service* will answer with a collection of predictions, described later). The key fields, *dataFrameColNames*, *dataFrameColTypes*, *dataFrameRowData* in this message represent a data table with a collection of samples as shown in Figure 3-4:

| | | | | | |
|-------------------|----------|-------------|-------------|-------------|-----------|
| dataFrameColNames | Time [s] | Cur_F08 [A] | Cur_F09 [A] | Cur_F10 [A] | |
| dataFrameColTypes | double | double | double | double | |
| dataFrameRowData | 0.0 | 52.60463 | 49.25544 | 46.69432 | sample #1 |
| | 0.7 | 45.345 | -1.45 | 0.09 | sample #2 |
| | 0.6 | 0.5 | 0.4 | 0.3 | sample #3 |

Figure 3-4 Collection of Samples to be evaluated

In this specific case, the predictive model *expects samples with four fields of type double*. If one or more of the samples do not meet the sample type expected by the model invoked, the prediction will fail.

The *PA Service* invokes the predictive model backend (i.e. Keras) for each sample and produces and returns a prediction for each sample, as shown in Figure 3-5.

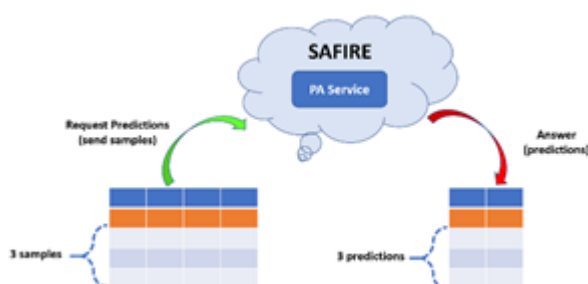


Figure 3-5 PA Service produces a prediction per sample

JSON Message to Receive answer predictions

PA Service answers to the Requesting Predictions message with another JSON message as the one shown below. The Answer will be received via subscription in the topic_answer specified in the answer to the registering message, in this case **hob_2341_boil_detection_answer**

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"      : "ClientReceivingPrediction",
  "timeStamp"        : "1517927276069",
  "clientId"         : "hob_2341",
  "dataFrameColNames" : ["probBoiling"],
  "dataFrameColTypes" : ["double"],
  "dataFrameRowDataPrediction" : [[0.201, 0.234, 0.312]],
  "errorDescription"  : "",
  "retCode"          : 0
}
```

Important Notes:

- Doubles (i.e 1.1513697245391086E-4) are returned as double values without “”.
- RetCode is returned as an integer without “”;

The key fields, `dataFrameColNames`, `dataFrameColTypes`, and `dataFrameRowDataPredictions` in this message represent a data table with a collection of predictions (one prediction per sample) as shown in Figure 3-6. Note that in this case the predictions consist of a single *double* value that, for example, represents the probability of being boiling:

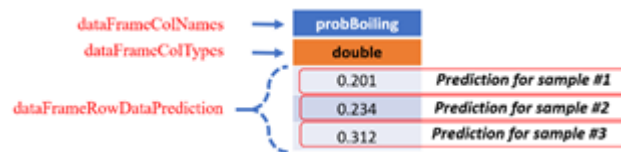


Figure 3-6 Collection of predictions

As another example, request message below (and the answer) represents the case of a model that predicts the *processingTime*, *monetary* cost and *quality* of a receipt expressed with the input fields of *processTypeName*, *amountProduced* and *energy*.

Request JSON message:

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"      : "ClientRequestingPrediction",
  "timestamp"        : "1517927276069",
  "clientId"         : "hotel_1",
  "dataFrameColNames" : ["processTypeName", "amountProduced", "energy"],
  "dataFrameColTypes" : ["string", "string", "integer"],
  "dataFrameRowData"  : [["Beef B", "250g", 6650]]
}
```

Answer JSON message:

```
{
  "serviceName"      : "SafirePrdAnalyticsPredictor",
  "messageType"      : "ClientReceivingPrediction",
  "timestamp"        : "1517927276069",
  "clientId"         : "hotel_1",
  "dataFrameColNames" : [["processingTime", "monetary", "quality"]],
  "dataFrameColTypes" : [["integer", "integer", "integer"]],
  "dataFrameRowDataPrediction" : [[115, 18, 9]],
  "errorDescription"  : "",
  "retCode"           : 0
}
```

Description of fields in the messages

This section summarises the meaning of all fields in the JSON messages described above.

- **"serviceName"** Always must be set to **"SafirePrdAnalyticsPredictor"** because is calling this service. Same value sent in Request is received in Answer.

- **"messageType"** Can be one of the following values:
 - *ClientRegistering, ClientRegisteringOk, ClientRegisteringResetOk, ClientRegisteringFail*
 - *ClientUnregistering, ClientUnregisteringOk, ClientUnregisteringOkFail*
 - *ClientRequestingPrediction, ClientReceivingPrediction*
- **"timeStamp"** It recommended to be the value returned by function now() in the computer when the client requests the service. It is needed to order the messages on reception side. Same value sent in Request is received in Answer.
- **"clientId"** An identifier of the client that is calling the service. Same value sent in Request is received in Answer.
- **"clientTopic"** The message will be sent to this topic. Same value sent in Request is received in Answer.
- **"clientTopicAnswer"** The answer to the message will be sent to this topic. Same value sent in Request is received in Answer.
- **"modelName"** Trained machine learning model to be invoked for prediction.
- **"backendName"** Machine learning module backend to be invoked, currently must be keywords "keras" or "spark".
- **"dataFrameColNames"** List ([]) of data field names provided by the request.
- **"dataFrameColTypes"** List ([]) of data field types provided by the request. Must be keywords "integer", "double", "string", "arrayDouble" or "arrayInteger".
- **"dataFrameRowData"** List of lists ([]) of data field values provided by the request. One or more rows can be provided.
- **"dataFrameRowDataPrediction"** List of lists ([]) of predicted values. There is one list of predicted values for each row.
- **"errorDescription"** Description of the error returned by the service when no prediction is returned (retCode <> 0).
- **"retCode"** 0 if the service success (and predicts), and <> 0 otherwise (and no prediction was produced).

3.1.3 Service access via REST Web Service

Access via REST Web Service is described in detail in SAFIRE Deliverable *D2.4 Full Prototype of Predictive Analytics Platform*, section *9.3.2 Templates to Develop Predictive Analytics REST Web Service and Clients*.

3.1.4 Uploaded models

The following models are currently uploaded in the PA Service:

- *onaNNOsciloThicknessModel_OneVsRest_5.h5*. This specific predictive model (neural network) is capable of detecting when the wire is 1 mm ahead of the thickness change (see deliverable D2.4 section 5.1 devoted to Ona BC for details).
- *electroluxNNModelCurFeaturesF*.h5*. This collection of predictive models (neural networks) are capable of predicting if the water on a pot is boiling. Models have been trained for currents F04, F05, F06, F07, F08, F09 and F10.
- *receiptboilingmodel*. This model returns the processing time, cost and quality of a collection of simple cooking receipts. In this case is a plain table for demonstrative purposes.

3.1.5 Training new models

The steps to be followed to train new predictive models is described in detail in SAFIRE Deliverable *D2.4 Full Prototype of Predictive Analytics Platform*, section 9.3.1 *Templates to Define and Train Predictive Models*. This section explains how to define basic new Spark/Keras models.

3.2 SITUATIONAL AWARENESS

The Situation Determination component allows for identifying changes in the situations of the environment. The current identified situation is used to support the optimisation / reconfiguration. It uses monitored “raw data” provided by the SAFIRE data ingestion NiFi templates, which get data directly from the legacy systems, or the predictive analytics for the product and processes, as well as knowledge available in different systems, to derive the product/machine/process current situation. Using the situation model the monitored data are being evaluated and the situation determined. The module also fulfils its role as a central broker between the other SAFIRE modules like the Optimisation Engine or Predictive Analytics module.

Business case specific customisation

In order to adapt the Situation Awareness services within a concrete Business case, customisations for the following software components have to be done:

- Situation model
- Situation monitoring service
- Situation determination service

3.2.1 Situation Model

In order to provide a custom solution for the different business cases and the company specific scenarios, a different situation model should be created for each purpose. The

custom situation models will include additional entities that extend the generic ones, and respective relations as needed. As it is foreseen, the generic entities that will be mainly extended to the customised situation models, are the Generic Datum and the Metric, since the specific use-case scenarios will require additional input data and evaluation measurements. The entities and the respective additional relations are being described in detail in the early and full prototype documents.

3.2.2 Situation Monitoring

For customising the Situation Monitoring for a specific application scenario, one has to implement the following business case specific “plugins”, which will be included in the Situation Monitoring Service via its configuration:

System Monitors:

In order to be able to monitor a specific system, one has to implement a Monitor to ingest data into the Situational Monitoring. For implementation, templates are available to allow for an easy integration into the Situation Monitoring Service (e.g. FileSystemMonitor, WebServiceMonitor, DatabaseMonitor).

Parsers:

For each specific monitor that was implemented, a corresponding parser has to be implemented to be able to parse the content and prepare it for analysing. For implementation, templates are available to allow for an easy integration into the Situation Monitoring Service (e.g. FileParser, WebServiceParser, DatabaseParser).

Analysers:

For each specific monitor that was implemented, a corresponding analyser has to be implemented to be able to analyse the monitored content. The analyser is responsible for filling the specific monitoring data model. For implementation, templates are available to allow for an easy integration into the Situation Monitoring Service (e.g. FileAnalyser, WebServiceAnalyser, DatabaseAnalyser).

Monitoring Data Models:

In order to be able to monitor a specific system, one has to implement a Monitor to ingest data into the Situational Monitoring. For implementation, templates are available to allow for an easy integration into the Situation Monitoring Service.

The individual implementations have to be addressed in a monitoring configuration to define bundles of classes which are responsible for the monitoring of a specified data source.

In the following the xml schema (monitoring-config.xsd) for the situational monitoring configuration is listed. The configuration elements are described in detail.

monitoring-config.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://www.atb-bremen.de" >
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="indexes"/>
        <xs:element ref="datasources"/>
        <xs:element ref="interpreters"/>
        <xs:element ref="monitors"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="indexes">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" ref="index"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="index">
    <xs:complexType>
      <xs:attribute name="id" use="required" type="xs:ID"/>
      <xs:attribute name="location" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="monitors">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" ref="monitor"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="monitor">
    <xs:complexType>
      <xs:attribute name="id" use="required" type="xs:ID"/>
      <xs:attribute name="datasource" use="required" type="xs:IDREF"/>
      <xs:attribute name="index" use="required" type="xs:IDREF"/>
      <xs:attribute name="interpreter" use="required" type="xs:IDREF"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="datasources">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" ref="datasource"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="datasource">
    <xs:complexType>
      <xs:attribute name="class" use="required"/>
      <xs:attribute name="id" use="required" type="xs:ID"/>
      <xs:attribute name="monitor" use="required"/>
      <xs:attribute name="options" use="required"/>
      <xs:attribute name="type" use="required" type="xs:NCName"/>
      <xs:attribute name="uri" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="interpreters">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" ref="interpreter"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="interpreter">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" minOccurs="1" ref="configuration"/>
        </xs:sequence>
        <xs:attribute name="id" use="required" type="xs:ID"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="configuration">
      <xs:complexType>
        <xs:attribute name="analyser" use="required"/>
        <xs:attribute name="parser" use="required"/>
        <xs:attribute name="type" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:schema>

```

Indexes - Each index entry has the following mandatory attribute

- id: The unique name of the index
- location: The URI of the location the index is stored

Datasources - Each datasource entry has the following mandatory attributes

- id: The unique name of the datasource
- type: The type of the datasource. Possible values are: file, webservice, database
- monitor: The class of the monitor to be used. Possible values are:

```

package de.atb.context.monitoring.monitors.database.DatabaseMonitor
package de.atb.context.monitoring.monitors.file.FileSystemMonitor
package de.atb.context.monitoring.monitors.file.FilePairSystemMonitor
package de.atb.context.monitoring.monitors.file.FileTripletSystemMonitor
package de.atb.context.monitoring.monitors.webservice.WebServiceMonitor

```

- options: Options for the datasource can be entered using this value. The options are dependent on the datasource to be used
- uri: The uri of the data source to be monitored
- class: The following datasource implementations are available

```

package de.atb.context.monitoring.config.models.datasources.DatabaseDataSource
package de.atb.context.monitoring.config.models.datasources.FilePairSystemDataSource
package de.atb.context.monitoring.config.models.datasources.FileSystemDataSource
package
de.atb.context.monitoring.config.models.datasources.FileTripletSystemDataSource
package de.atb.context.monitoring.config.models.datasources.WebServiceDataSource

```

Interpreters - Each interpreter entry has the following mandatory attributes

- id: The unique name of the interpreter

- configuration
 - analyser: The analyser class to be used. The following implementations are available:

```
package de.atb.context.monitoring.analyser.database.DatabaseAnalyser
package de.atb.context.monitoring.analyser.file.FileAnalyser
package de.atb.context.monitoring.analyser.file.FilePairAnalyser
package de.atb.context.monitoring.analyser.file.FileTripletAnalyser
package de.atb.context.monitoring.analyser.webservice.WebServiceAnalyser
```

- parser: The parser class to be used. The following implementations are available:

```
package de.atb.context.monitoring.parser.database.DatabaseParser
package de.atb.context.monitoring.parser.file.FileParser
package de.atb.context.monitoring.parser.file.FilePairParser
package de.atb.context.monitoring.parser.file.FileTripletParser
package de.atb.context.monitoring.parser.webservice.WebServiceParser
```

- type: Currently only used for File analyser and parser. Defines the file extensions to be used.

Monitors - monitors are grouping datasources, interpreters and indexes for the monitoring of a specific datasource (a monitor for a data source). Each monitor entry has the following mandatory attributes

- id: The unique name of the monitor
- datasource: Id of one datasource which was configured.
- interpreter: Id of one interpreter which was configured.
- Index: Id of one index which was configured.

An example of a whole configuration is provided in the following:

```
monitoring-config.xml
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="http://www.atb-bremen.de"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.atb-bremen.de monitoring-config.xsd">

  <indexes>
    <index id="index-dummy" location="indexes/dummy"></index>
    <index id="index-safire" location="indexes/safire"></index>
  </indexes>

  <datasources>
    <datasource
      id="datasource-dummy"
      type="file"
      monitor="de.atb.context.monitoring.monitors.file.FilePairSystemMonitor"
      uri="target/test-classes/filepairmonitor"
      options="extensionOne=1&extensionTwo=2"
      class="de.atb.context.monitoring.config.models.datasources.FilePairSystemDataSource"
    />
  </datasources>
```

```

    <interpreters>
      <interpreter id="interpreter-dummy">
        <configuration
type="*"
parser="de.atb.context.monitoring.parser.file.DummyFilePairParser"
analyser="de.atb.context.monitoring.analyser.file.DummyFilePairAnalyser" />
      </interpreter>
    </interpreters>

    <monitors>
      <monitor
        id="monitor-dummy"
        datasource="datasource-dummy"
        interpreter="interpreter-dummy"
        index="index-dummy" />
    </monitors>
  </config>

```

Finally, a service configuration has to be created which defines the services to be deployed enriched with additional network information (as host, location, name, server, proxy).

```

services-config.xml
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="http://www.atb-bremen.de"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <services>
    <service id="AmIMonitoring">
      <host>localhost</host>
      <location>http://localhost:19001</location>
      <name>AmIMonitoringService</name>
      <server>de.atb.context.services.AmIMonitoringService</server>
      <proxy>de.atb.context.services.IAmIMonitoringService</proxy>
    </service>
    <service id="AmI-repository">
      <host>localhost</host>
      <location>http://localhost:19002</location>
      <name>AmIMonitoringDataRepositoryService</name>
    </service>
    <server>de.atb.context.services.AmIMonitoringDataRepositoryService</server>
    <proxy>de.atb.context.services.IAmIMonitoringDataRepositoryService</proxy>
  </service>
  <service id="PersistenceUnitService">
    <host>localhost</host>
    <location>http://localhost:19004</location>
    <name>PersistenceUnitService</name>
    <server>de.atb.context.services.PersistenceUnitService</server>
    <proxy>de.atb.core.services.IPersistenceUnitService</proxy>
  </service>
</services>
</config>

```

3.2.3 Situation Determination

For customising the Situation Determination for a specific application scenario, one has to implement the following business case specific “plugins”, which will be included in the Situation Determination Service via its configuration:

Situation Identifiers

The main classes of the Situation Identifiers are:

- *IContextIdentifier*: The interface defining a situation identifier. A situation identifier is a wrapper that is used to identify a situation based on monitored data and the situation model. In each concrete implementation of a situation identifier the usage of a reasoner can be defined.

ContextContainer: This class is a wrapper object that holds an identified situation during run-time.

Finally, the service configuration the situational determination related services to be deployed have to be added, enriched with additional network information (as host, location, name, server, proxy).

```
services-config.xml
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="http://www.atb-bremen.de"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <services>
    <service id="ContextExtractionService">
      <host>localhost</host>
      <location>http://localhost:19001</location>
      <name>ContextExtractionService</name>
      <server>de.atb.context.services.ContextExtractionService</server>
      <proxy>de.atb.context.services.IContextExtractionService</proxy>
    </service>
    <service id="ContextRepositoryService">
      <host>localhost</host>
      <location>http://localhost:19002</location>
      <name>ContextRepositoryService</name>
      <server>de.atb.context.services.ContextRepositoryService</server>
      <proxy>de.atb.context.services.IContextRepositoryService</proxy>
    </service>
  </services>
</config>
```

3.3 OPTIMISATION

The entire optimisation process starts with describing a scenario to be optimised using Factory Description Language (FDL), briefly described in deliverable D5.6. This can be performed in two ways: either by configuring template generator tool or just working directly with a configuration template in the FDL formal. Below, both these possibilities are explained. In particular, the crucial extracts of FDL templates for the three SAFIRE industrial cases are discussed to facilitate the manual modification of the plant or order parameters.

3.3.1 Generating FDL templates

To facilitate the preparation of an FDL description for a certain factory, which for larger plants can be relatively long, a software tool generating such description has been written in the Java language and provided in the official SAFIRE Git repository named *optimisation engine*. Its branch named *formatted_outputs* contains the template generator based on each SAFIRE BC. It is located in package `uk.ac.york.safire.factoryModel`. In this section, the process of the FDL description generation is described based on the OAS BC. The FDL templates for ONA and Electrolux BCs are simpler due to the lack of production lines.

The template generator for scenarios similar to the OAS BC is located in package `uk.ac.york.safire.factoryModel.OAS`. This package contains the following classes:

- *DependentSetupPattern* which describes the sequence dependent setup,
- *Device* which describes the plant resources and all their parameters such as resource type, working modes, availability, etc,
- *ProductionLine* which describes the set of resources that performs a complete production manufacturing. In the current version of this class, each production line is composed of a silo, a mixer and a tank. However, the number and names of these resources can be changed by an end-user.
- *ProductionProcess* which describes the recipes to produce commodities, e.g. paints. This description includes the compatible production lines, the produced amount and the subprocesses (with their processing time, economic cost, etc.),
- *SubProcess* which describes the subprocesses of a production process,
- *SubProcessRelation* which describes the execution order of subprocesses together with the temporal relations between them,
- *OASFactoryModel* which generates a model of the OAS factory,
- *OASXMLReader* and *OASXMLWriter* which read from and write an FDL file that describes the scenario.

The starting point of preparing a new FDL template is to modify the *OASFactoryModel* class. In this class, the factory is specified by providing the resource types (enum *DeviceType*), working modes of devices (enum *ModeType*), the number of machines of each type available in the factory (int[] *NumberOfDevices*, i.e., 5 silos, 10 mixers and 2 storage tanks). The processing time for each recipe depending on the machine type and its mode is also provided. Similarly, the template of recipes is generated by providing the commodities names (String[] *Productions*), the ordered amounts (int[] *AmountRequired*), priority of each commodity (int[] *Urgency*) working modes of each machine type, the required production and their amount. Finally, the objectives to be minimised during the optimisation process are specified as well (String[] *Objectives*).

With these definitions, we can then generate the factory model, via the method *getOASConfiguration()*. Firstly, the devices are generated based on the *DeviceType* enum and assigned with their cost for processing in a time unit. Then, the related production lines are obtained based on these devices. The recipes (i.e., production processes) are then generated for each required commodity. With each recipe, the compatible production lines and commodity amount is assigned, as well as subprocesses, their device affinities and costs. Then, based on *SubProcessRelationTemplate*, which defines the execution order of these sub-processes, the execution order for the subprocesses is imposed. Finally, the sequence dependent setup is generated via iterating all possible execution order of those subprocesses and assign an extra cost to the ones that belong to different paint types (mainly for cleaning the machine). This class is invoked by the *XMLWriter* class can invoke *getOASConfiguration()* method to obtain those 5 lists and then write them into XML as an OAS factory description (see input folder or simply execute the main method in the XML Writer). A configuration edited by a user can be read by executing the main method in the *XMLReader* class.

3.3.2 FDL template for ONA BC

The considered ONA business case is an example of a discrete manufacturing scenario. It is related to wire electrical discharge machining (WEDM), where a thermo-electric sparking process removes material using a wire to cut the desired shape of a part. Complex profiles with tight tolerances in hard conductive materials can be obtained. The objectives are to minimise the makespan and the monetary cost per part. This cost can be obtained by summarising all values of *monetaryCost* of the subprocesses involved in producing a given part. If applicable, the values of *extraMonetaryCost* of *sequenceDependentSetups* should be added as well.

Table 1. Example parameters of cutting parts in the considered discrete manufacturing scenario

| Part | Machine Size | Mode | Cutting time (min) | Wire Cost per part (EUR) | Machine cost per part (EUR) | Total cost per part (EUR) |
|------|--------------|------|--------------------|--------------------------|-----------------------------|---------------------------|
| | | | | | | |

| | | | | | | |
|---------|--------|-----|--------|-------|--------|--------|
| Part 01 | Small | 1 | 2833.5 | 28.1 | 164.0 | 192.1 |
| | Small | 2 | 2956.2 | 28.1 | 140.3 | 168.4 |
| | Small | 3 | 3042.1 | 28.1 | 147.8 | 175.9 |
| | Small | 4 | 3174.1 | 30.2 | 136.8 | 167.0 |
| | Medium | 1 | 2033.5 | 30.2 | 242.9 | 273.1 |
| | ... | ... | ... | ... | ... | ... |
| | Large | 4 | 1974.1 | 53.7 | 408.4 | 462.1 |
| ... | | | | | | |
| Part 20 | Large | 1 | 5341.3 | 335.2 | 5866.7 | 6201.9 |
| | Large | 2 | 5505.1 | 383.1 | 8381.0 | 8764.1 |
| | Large | 3 | 5191.7 | 482.1 | 5673.8 | 5673.8 |
| | Large | 4 | 4106.6 | 648.3 | 4754.9 | 4754.9 |

The resource allocation consists of selecting processing devices (and thus production line) for cutting the part (product). The selected processing devices (machines) can process parts of various sizes (small, medium or large) and operate in a number of modes, each related to, e.g., a different wire type. Consequently, all possible modes for processing each considered part have to be explicitly specified using element *subprocessProcessingDeviceMode*. Table 1 presents an example of parameters of manufacturing parts in the considered factory.

With FDL, each part in the considered discrete manufacturing scenario is characterised by its name, its priority (in terms of urgency), the number of cuts required to produce the part and the list of compatible devices of the given production. Besides, a production contains a set of subprocesses representing each cut operation, where a production procedure can be pre-empted between cuts. Each subprocess (a cut) contains the information of processing time, energy consumption and monetary cost for executing on a given machine. Note, the above configuration is built based on the consideration that users may need to configure each cut operation manually (e.g., adjust the processing time). In the case where manually configuring cut operations is not necessary, a user can also describe a production without providing information of subprocesses, where the system generates the corresponding subprocesses automatically based on the given number of cuts.

Then, the Optimisation Engine Configurator (OEC) is used to generate both the configuration template and the objective function evaluator in the following way. Depending on the input factory parameter, OEC locates to the correct XML factory modelling file and reads the corresponding optimisation parameters and factory descriptions, which include optimisation objectives, factory resources with their availability, production processes with their subprocesses, subprocess relations of subprocesses and dependent setups for production processes. An example of the FDL-based factory model for the considered discrete manufacturing scenario is given below, starting with the objective description.

```
<objectives>
  <objective name="makespan" />
  <objective name="monetary" />
</objectives>
```

The objectives are titled as *objective* with a name specifying the metric for optimisation with the assumption of minimisation. For the considered case, two objectives are

supported: *makespan* and *monetary*. However, one can configure the objectives to make the optimisation engine to focus only on one of the objectives by simply removing the other objective.

```
<processingDevices>
  <processingDevice name="Small 1" availability="1">
    <unavailableTimes>
      <unavailableTime>50,100</unavailableTime> <unavailableTime>250,300</unavailableTime>
    </unavailableTimes>
  </processingDevice>
  <processingDevice name="Small 2" availability="1">
    <unavailableTimes>
      <unavailableTime>0,20</unavailableTime>
    </unavailableTimes>
  </processingDevice>
  <processingDevice name="Small 3" availability="0">
    <unavailableTimes>
      <unavailableTime>25,30</unavailableTime>
    </unavailableTimes>
  </processingDevice>
</processingDevices>
```

The resources in a factory are modelled as a list of *processingDevice*, where each device is specified with a unique name, availability and unavailable times (in the case where the device is available only in certain time intervals). As given in the example, device *Small 1* is unavailable during periods 50-100 and 250-300 (from the starting point of manufacturing, in minutes) while *Small 2* is available during the entire manufacturing process. For the considered process manufacturing scenario, each device is also associated with an operating mode (i.e., *economy*, *standard* or *performance*), which can be switched dynamically during run-time with various costs imposed during manufacturing.

The production model is presented below. This model contains each part to be produced with a set of subprocesses required to produce it. Each subprocess is then modelled by OEC as an individual task associated with a specified resource allocation (among all its compatible resources) and a unique priority for schedule by a fixed priority preemptive scheduler.

```
<productionProcess name="P15" priority="15" cuts="10">
  <comptiableDevices>
    <comptiableDevice name="Large 1" processingTime="5505" energy="120" monetary="4651" />
    <comptiableDevice name="Large 2" processingTime="5341" energy="120" monetary="6573" />
    <comptiableDevice name="Large 3" processingTime="7421" energy="120" monetary="3566" />
    <comptiableDevice name="Large 4" processingTime="6205" energy="120" monetary="4255" />
  </comptiableDevices>
  <subProcesses>
    <subProcess name="P15 cut 1">
      <subProcessProcessingDevice name="Large 1" processingTime="550" energy="12" monetary="465" />
      <subProcessProcessingDevice name="Large 2" processingTime="534" energy="12" monetary="657" />
      <subProcessProcessingDevice name="Large 3" processingTime="742" energy="12" monetary="356" />
    </subProcess>
  </subProcesses>
</productionProcess>
```



```

    <subProcessProcessingDevice name="Large 4" processingTime="620"
energy="12" monetary="425" /> </subProcess>
    <subProcess name="P15 cut 2">
        <subProcessProcessingDevice name="Large 1" processingTime="550" energy="12" mon-
etary="465" />
        <subProcessProcessingDevice name="Large 2" processingTime="534" energy="12" mon-
etary="657" />
        <subProcessProcessingDevice name="Large 3" processingTime="742" energy="12" mon-
etary="356" />
        <subProcessProcessingDevice name="Large 4" processingTime="620"
energy="12" monetary="425" /> </subProcess>
    <subProcess name="P15 cut 3">
        <subProcessProcessingDevice name="Large 1" processingTime="550" energy="12" mon-
etary="465" />
        <subProcessProcessingDevice name="Large 2" processingTime="534" energy="12" mon-
etary="657" />
        <subProcessProcessingDevice name="Large 3" processingTime="742" energy="12" mon-
etary="356" />
        <subProcessProcessingDevice name="Large 4" processingTime="620"
energy="12" monetary="425" />
    </subProcess>
</subProcesses>
</productionProcess>

```

To ensure that the subprocesses of a production process are executed in the correct order (if necessary), the notion *subprocessRelation* is introduced to describe the execution sequence of subprocesses. For either the considered discrete or process manufacturing scenario, the notion *M* is used to describe that a subprocess *SP2* is executed immediately after another subprocess *SP1*. This information is used by OEC for generating the objective function.

```

<subprocessRelations>
    <subprocessRelation source="P1 cut 0" destination="P1 cut 1" allensOperator="M" />
    <subprocessRelation source="P1 cut 1" destination="P1 cut 2" allensOperator="M" />
    <subprocessRelation source="P2 cut 0" destination="P2 cut 1" allensOperator="M" />
    <subprocessRelation source="P2 cut 1" destination="P2 cut 2" allensOperator="M" />
    <subprocessRelation source="P3 cut 0" destination="P3 cut 1" allensOperator="M" />
    <subprocessRelation source="P3 cut 1" destination="P3 cut 2"
allensOperator="M" />
</subprocessRelations>

```

At last, during scheduling, multiple productions could execute on one resource, which could cause the extra cost for the machine to be cleaned and/or reset for a different product. The FDL models treat such cost as a set of potential independent tasks for sequence-dependent setup, where each of such tasks describes the source production (the product being processed), the target production (the product to be processed), the resource, time consumption and the corresponding costs. The objective function will be generated by OEC based on the list of dependent setup tasks and applies these tasks dynamically where appropriate (i.e., when a dependent setup is necessary) during the scheduling process.

```

<sequenceDependentSetups>
    <sequenceDependentSetup source="P1" destination="P2" processingDevice="Small 4" ex-
traProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
    <sequenceDependentSetup source="P1" destination="P2" processingDevice="Medium 1" ex-
traProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
    <sequenceDependentSetup source="P1" destination="P2" processingDevice="Small 3" ex-
traProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
    <sequenceDependentSetup source="P1" destination="P2" processingDevice="Medium 2" ex-
traProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
    <sequenceDependentSetup source="P1" destination="P2" processingDevice="Small 2" ex-
traProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />

```



```
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Small 1" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Large 3" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Large 4" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Medium 3" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Large 1" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Medium 4" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
<sequenceDependentSetup source="P1" destination="P2" processingDevice="Large 2" extraProcessingTime="10" extraEnergyConsumption="10" extraMonetaryCost="1000" />
</sequenceDependentSetups>
```

FDL can be applied to other discrete manufacturing scenarios in a similar manner.

3.3.3 FDL template for OAS BC

The description of the factory provided by SAFIRE industrial partner OAS is similar to that of the considered discrete manufacturing scenario, but with the notion *productLine* introduced. The reason to introduce this notation is that, as a typical process manufacturing plant, commodities produced often require several devices working in collaboration, with an explicit execution order enforced.

The considered chemical plant produces paints by mixing/dispersion of powdery, liquid and paste recipe components, following a stored recipe. Table 2 gives an example of parameters for producing several paints the considered process manufacturing scenario. Below we provide an example FDL for describing the production line.

```
<productionLines>
  <productionLine name="ProductionLine 1">
    <productionLineProcessingDevices>
      <productionLineProcessingDevice order="0" name="Silo 1" />
      <productionLineProcessingDevice order="1" name="Mixer 1" />
      <productionLineProcessingDevice order="2" name="Tank 1" />
    </productionLineProcessingDevices>
  </productionLine>
  <productionLine name="ProductionLine 2">
    <productionLineProcessingDevices>
      <productionLineProcessingDevice order="0" name="Silo 1" />
      <productionLineProcessingDevice order="1" name="Mixer 2" />
      <productionLineProcessingDevice order="2" name="Tank 1" />
    </productionLineProcessingDevices>
  </productionLine>
  <productionLine name="ProductionLine 3">
    <productionLineProcessingDevices>
      <productionLineProcessingDevice order="0" name="Silo 2" />
      <productionLineProcessingDevice order="1" name="Mixer 3" />
      <productionLineProcessingDevice order="2" name="Tank 1" />
    </productionLineProcessingDevices>
  </productionLine>
</productionLines>
```

Table 2. Example of parameter for producing several paints in the considered OAS BC scenario

| Paint Types | Compatible Production Lines | Amount of Produced Commodity | Recipe Execution Time |
|-------------|-------------------------------|------------------------------|-----------------------|
| Std White | $\{P_1, P_2, P_3, P_4, P_5\}$ | 5 t | 60 min. |
| | $\{P_6, P_7\}$ | 10 t | 45 min. |
| | $\{P_8, P_9\}$ | 10 t | 30 min. |
| Super White | $\{P_1, P_2, P_3, P_4, P_5\}$ | 6 t | 90 min. |
| | $\{P_6, P_7\}$ | 12 t | 60 min. |
| | $\{P_8, P_9\}$ | 12 t | 45 min. |
| Std Blue | $\{P_1, P_2, P_3, P_4, P_5\}$ | 4 t | 100 min. |
| | $\{P_6, P_7\}$ | 8 t | 80 min. |
| | $\{P_8, P_9\}$ | 8 t | 60 min. |
| Std Green | $\{P_1, P_2, P_3, P_4, P_5\}$ | 4 t | 120 min. |
| | $\{P_6, P_7\}$ | 8 t | 90 min. |
| | $\{P_8, P_9\}$ | 8 t | 60 min. |

As given above, the products in the considered process manufacturing scenario are manufactured by production lines, where each line contains certain devices. All lines are executed following a strict execution order: Silo, Mixer and Tank. Below presents the example describing the production process and related-cost for producing the paint *Std Weiss*.

```
<productionProcess name="Std Weiss 45t" priority="1">
  <processType name="Std Weiss A" amountProduced="5t">
    <comptiableProductionLines>
      <comptiableProductionLine>ProductionLine 1</comptiableProductionLine>
      <comptiableProductionLine>ProductionLine 2</comptiableProductionLine>
      <comptiableProductionLine>ProductionLine 3</comptiableProductionLine>
    </comptiableProductionLines>
```

```
  <subprocesses>
    <subprocess name="Std Weiss A Task 1">
      <subprocessProcessingDevicesGroup processingTime="15">
        <subprocessProcessingDevice name="Silo 1" mode="Ecomony"/>
        <subprocessProcessingDevice name="Mixer 1" mode="Ecomony"/>
      </subprocessProcessingDevicesGroup>
      <subprocessProcessingDevicesGroup processingTime="15">
        <subprocessProcessingDevice name="Silo 1" mode="Ecomony"/>
        <subprocessProcessingDevice name="Mixer 1" mode="Standard"/>
      </subprocessProcessingDevicesGroup>
      <subprocessProcessingDevicesGroup processingTime="15">
        <subprocessProcessingDevice name="Silo 1" mode="Ecomony"/>
        <subprocessProcessingDevice name="Mixer 1" mode="Power"/>
      </subprocessProcessingDevicesGroup>
    </subprocess>
    <subprocess name="Std Weiss A Task 2">
      <subprocessProcessingDevicesGroup processingTime="120">
        <subprocessProcessingDevice name="Mixer 1" mode="Ecomony" />
      </subprocessProcessingDevicesGroup>
      <subprocessProcessingDevicesGroup processingTime="80">
        <subprocessProcessingDevice name="Mixer 1" mode="Standard" />
      </subprocessProcessingDevicesGroup>
      <subprocessProcessingDevicesGroup processingTime="40">
        <subprocessProcessingDevice name="Mixer 1" mode="Power" />
      </subprocessProcessingDevicesGroup>
    <subprocess name="Std Weiss A Task 3">
      <subprocessProcessingDevicesGroup processingTime="10">
        <subprocessProcessingDevice name="Mixer 1" mode="Ecomony" />
        <subprocessProcessingDevice name="Tank 1" mode="Standard" />
      </subprocessProcessingDevicesGroup>
      <subprocessProcessingDevicesGroup processingTime="10">
        <subprocessProcessingDevice name="Mixer 1" mode="Standard" />
        <subprocessProcessingDevice name="Tank 1" mode="Standard" />
      </subprocessProcessingDevicesGroup>
```

```

</subprocessProcessingDevicesGroup>
<subprocessProcessingDevicesGroup processingTime="10">
  <subprocessProcessingDevice name="Mixer 1" mode="Power " />
  <subprocessProcessingDevice name="Tank 1" mode="Standard"
/>
</subprocessProcessingDevicesGroup>

```

As described in the FDL extract above, an order of 45 tonnes of *Std Weiss* is executed by several sub-orders, where each sub-order produces the maximum amount that the production line can produce in one execution e.g., *Std Weiss A* can produce 5 tonnes in each execution on production line P_1 .

```

<subprocessRelations>
  <subprocessRelation source="Std Weiss A Task 1" destination ="Std Weiss A Task 2" al-
lensOperator="M" />
  <subprocessRelation source="Std Weiss A Task 2" destination ="Std Weiss A Task 3" al-
lensOperator="M" />
</subprocessRelations>

```

The Element *subprocessRelations* contains a set of sub-elements describing the execution order between subprocesses in a given process, where Allen's temporary operator *M* indicates that the source subprocess must be executed directly before the destination.

FDL can be applied to other process manufacturing scenarios in a similar manner.

3.3.4 FDL template for Electrolux BC

The scenario specified by SAFIRE industrial partner Electrolux is similar to OAS BC as in both of them, an ordered amount of certain commodities (food and paint, respectively) needs to be produced following stored recipes. In contrast to OAS BC, some recipes result in inferior quality of the commodities and hence a trade-off between makespan and quality can be observed. In this BC, the optimisation is then performed for these two objectives, as can be described in FDL in the following way.

```

<objectives>
  <objective name="makespan" />
  <objective name="quality" />
</objectives>

```

Table 3 provides example parameters of recipes for the considered scenario. There can be alternative recipes to produce the same commodity type, possible in different amount, requiring different processing time or quality. Below, two recipes for boiled water are shown in FDL.

```

<productionProcesses>
  <productionProcess name="Boiled Water 3000g" priority="1">
    <processTypes>
      <processType name="Boiled Water A" amountProduced="1000g" predecessor="" ener-
gy="350" processingTime="15" monetary="3" quality="8">
        <comptiableCookingZones>
          <comptiableCookingZone>CookingResource 1</comptiableCookingZone>
          <comptiableCookingZone>CookingResource 2</comptiableCookingZone>
          <comptiableCookingZone>CookingResource 3</comptiableCookingZone>

```

```

        <comptiableCookingZone>CookingResource 4</comptiableCookingZone>
    </comptiableCookingZones>
</processType>
<processType name="Boiled Water B" amountProduced="2000g" predecessor="" energy="1400" processingTime="10" monetary="6" quality="8">
    <comptiableCookingZones>
        <comptiableCookingZone>CookingResource 5</comptiableCookingZone>
        <comptiableCookingZone>CookingResource 6</comptiableCookingZone>
    </comptiableCookingZones>
</processType>
</processTypes>
</productionProcess>

```

These recipes include also information about energy usage and economical cost. Optimisation with respect to these objectives is also possible and would require adding the appropriate keywords inside the *objectives* section, as shown below for energy usage (it is assumed that all the objectives are minimised).

```

<objectives>
    <objective name="makespan" />
    <objective name="quality" />
    <objective name="energy" />
</objectives>

```

In the example above, the first recipe (Boiled Water A) can be processed in cooking resources 1-4, whereas the second recipe (Boiled Water B) requires either cooking resource 5 or 6. The cooking resources are formed as a Cartesian product of cooking zones available in hob and compatible pots. Both these resource types are specified as processing devices.

```

<processingDevices>
    <processingDevice name="CZ 1" availability="1">
        <unavailableTimes>
            <unavailableTime />
        </unavailableTimes>
    </processingDevice>
    ...
    <processingDevice name="Pot 1" availability="1">
        <unavailableTimes>
            <unavailableTime />
        </unavailableTimes>
    </processingDevice>
</processingDevices>

```

Pots are paired with cooking zones using the `<groupedResource>` tag.

```

<groupedResources>
    <groupedResource name="CookingResource 1">

```

```

<groupedProcessingDevices>
  <groupedProcessingDevice name="CZ 1" />
  <groupedProcessingDevice name="Pot 1" />
</groupedProcessingDevices>
</groupedResource>
</groupedResources>

```

Table 3. Example of parameter for cooking meals in the considered Electrolux BC scenario

| Food type | Recipe name | Predecessor | Amount (g) | Cooking zone | Pot type | Energy (kJ) | Cooking time (min) | Monetary cost (€) | Deficiency (inverse of quality) |
|-------------|-------------|----------------|------------|--------------------------------|----------|-------------|--------------------|-------------------|---------------------------------|
| Pasta | A | Boiled water A | 100 | Hob(1), Hob(2), Hob(3), Hob(4) | Pot 1 | 840 | 30 | 0.021 | 2 |
| | B | Boiled water A | 100 | Hob(1), Hob(2), Hob(3), Hob(4) | Pot 1 | 770 | 25 | 0.018 | 9 |
| | C | Boiled water B | 200 | Hob(5), Hob(6) | Pot 2 | 1120 | 20 | 0.021 | 14 |
| | D | Boiled water B | 200 | Hob(5), Hob(6) | Pot 2 | 1190 | 15 | 0.018 | 19 |
| | E | Boiled water C | 300 | Hob(7) | Pot 3 | 1520 | 10 | 0.021 | 22 |
| | F | Boiled water C | 300 | Hob(7) | Pot 3 | 1590 | 5 | 0.018 | 25 |
| Meat (beef) | A | Boiled water A | 250 | Hob(1), Hob(2), Hob(3), Hob(4) | Pot 1 | 4550 | 120 | 0.27 | 5 |
| | B | Boiled water A | 250 | Hob(1), Hob(2), Hob(3), Hob(4) | Pot 1 | 6650 | 110 | 0.18 | 9 |
| | C | Boiled water B | 500 | Hob(5), Hob(6) | Pot 2 | 6900 | 90 | 0.27 | 12 |
| | D | Boiled water B | 500 | Hob(5), Hob(6) | Pot 2 | 7000 | 85 | 0.18 | 16 |
| | E | Boiled water C | 750 | Hob(7) | Pot 3 | 7350 | 60 | 0.27 | 21 |
| | F | Boiled water C | 750 | Hob(7) | Pot 3 | 7550 | 55 | 0.18 | 27 |

3.4 SECURITY FRAMEWORK

The SAFIRE SPT Framework explores coherent system-wide security policy and enforcement in IIoT systems. To that end it provides an expressive, dynamic, and comprehensible security policy description and enforcement vehicle. The base requirements are met by the policy modelling and enforcement framework expressed in the Next Generation Access Control (NGAC) Functional Architecture and related standards. [1]

NGAC can provide a unifying approach to policy definition and access control in a dynamic FoF IIoT environment such as SAFIRE. It was necessary to make NGAC more usable, so as to be deployable in more diverse environments than its past reference

implementations. It was also necessary to make it more easily extensible for new kinds of protected objects, and supportive of more complex policies and composed policies.

SAFIRE has implemented its own version of the **NGAC functional architecture** with “unbundled” PEP and RAP. This architecture enables a more extensible implementation of NGAC by easing the addition of new protected object kinds. Another important feature of SAFIRE’s version is a lightweight Policy Server that places fewer demands on the operating environment and thus is more portable.

SAFIRE’s NGAC implementation includes its own simple **declarative policy specification language** that is easily extensible in future versions.

The “users” of the SAFIRE security framework are not SAFIRE “end-users” but rather system architects and developers of SAFIRE components and client applications, as well as system and security management administrators. Normal SAFIRE end-users may be unaware of the security functions as long as their usage of SAFIRE features is consistent with the configured security policies. Administrative and security management users have a role in the configuration and operation of the system. Before a SAFIRE system is made operational there are ICT architects and application implementers involved in the application of the Security Framework. This User Manual section is intended to address these audiences. Consequently, this manual will also cover the subject of integrating SAFIRE security components, along with other SAFIRE functional components, with the underlying platform and security components of a new or existing ICT environment that is intended to support a SAFIRE deployment.

This User Manual for SAFIRE security should be used in conjunction with the Security Methodology section of the SAFIRE Integrated Methodology document D5.6, which provides more background and rationale for the use of the features described in this User Manual, and with the installation and configuration guidance for the security framework contained in Section 4.1.4 of this document. This manual provides an update to some of the information provided in D5.4 Full Prototype of the SPT Framework.

3.4.1 Components of the Security Framework implementation

The SAFIRE Security Framework is based on an implementation developed by The Open Group of the NGAC standard [1]. It is very portable, with its only external dependency being that SWI-Prolog must be available for the platform’s operating environment.

The functional architecture of NGAC is depicted in Figure 3-7. Components coloured in blue are trusted components. Together these components mediate access by the Client Application (CA) to the Protected Resources by determining and enforcing the policy provided by security administrators.

Although they are trusted components, the Policy Enforcement Point (PEP) and the Resource Access Point (RAP) are “unbundled” from the implementation of the core NGAC components, which include the Policy Tool and the Policy Server. Specifically, the PEP and RAP are developed or provided by the developer of the Client Application

to complete the Resource Access Path. Prior to using NGAC the CA would access the resource servers directly through the Resource Access Interface. With NGAC the Protected Resources are accessible only to the NGAC components.

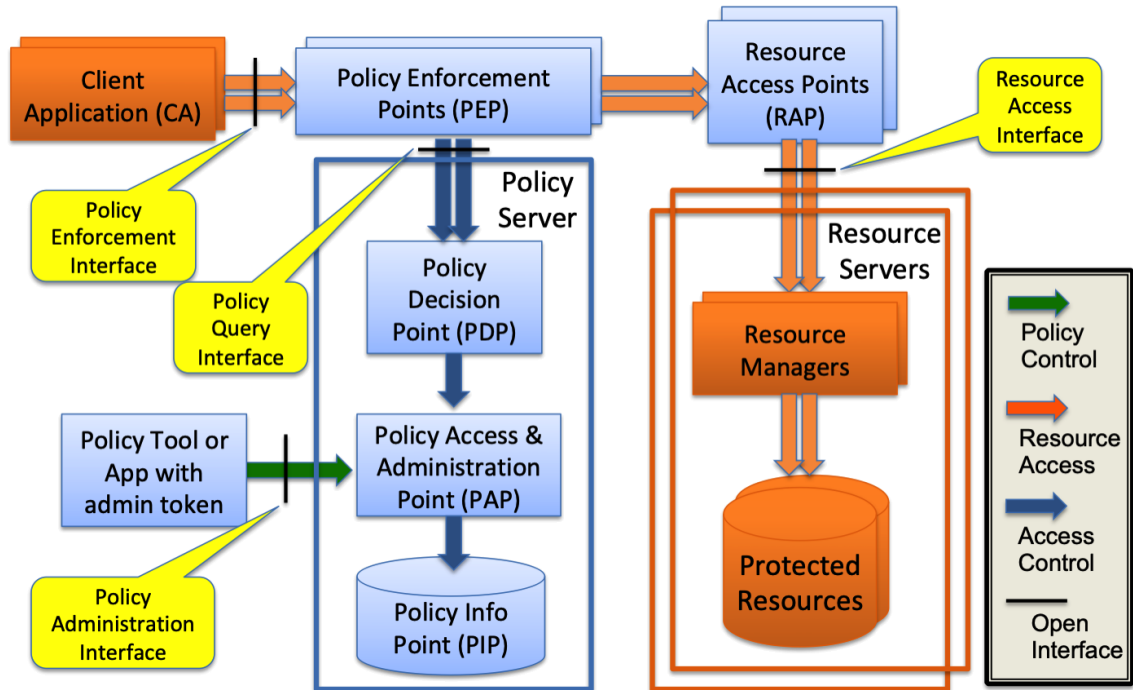


Figure 3-7 NGAC - Functional Architecture

3.4.2 Declarative Policy Language

A declarative policy specification is of the form:

policy(<policy name>, <policy root>, <policy elements>).

where,

<policy name> is an identifier for the policy definition

<policy root> is an identifier for the policy class defined by this definition

<policy elements> is a list [<element>, ... , <element>]

where each <element> is one of:

user(<user identifier>)

user_attribute(<user attribute identifier>)

object_class(<object class identifier>, <operations>)

object(<object identifier>)

```

object( <object identifier>, <object class identifier>, <inh>, <host name>,
<path name>, <base node type>, <base node name> )

object_attribute( <object attribute identifier> )

policy_class( <policy class identifier> )

composed_policy( <new policy name>, <policy name1>, <policy name2> )

operation( <operation identifier> )

assign( <entity identifier>, <entity identifier> )

associate( <user attribute id>, <operations>, <object attribute id> )

```

where <operations> is a list:

```
[ <operation identifier>, ... , <operation identifier> ]
```

```
connector( ' PM ' )
```

The initial character of all identifiers must be a lower-case letter or the identifier must be quoted with single quotes, e.g. smith or 'Smith' (identifiers are case sensitive so these examples are distinct). Quoting of an identifier that starts with a lower-case letter is optional, e.g. smith and 'smith' are not distinct.

Additionally: < inh > can be **yes** or **no**.

< host name> contains the name of the host where the corresponding file system object resides.

< path name> is the complete path name of the corresponding file system object.

3.4.3 Using the 'ngac' Policy Tool

The 'ngac' Policy Tool is a command driven application that includes the policy language interpretation and decision logic that is also used in the Policy Server's Policy Decision Point.

3.4.3.1 Developing and testing policies

A policy in the declarative policy language is just a text file containing a policy specification as defined above. Any text editor can be used to create a policy file. By convention, but not by necessity, the file name has the extension ".pl" (which can be remembered by the mnemonic "policy language"). In fact, a benefit of not using the extension is that it is then not necessary to put the file name in single quotes when using the 'ngac' tool commands. However, in any event, an unquoted file name must start with a lowercase letter and may only contain underscore (" _ ") and no white space. A quoted file name does not have these restrictions, and can be any name that is acceptable to the file system.

The policy file may be loaded into the Policy Tool using the `import_policy` command. The policy may be queried using the `access` command, yielding the same result that the Policy Server would return from a Policy Query API access call. Subsequent imports of a policy by the same name (not necessarily from a file of the same name) will replace the loaded named policy. The cycle of edit, import, query may be repeated without leaving the Policy Tool until the user is satisfied that the policy behaves as intended.

3.4.3.2 Policy Tool interactive commands

After starting 'ngac' it offers the prompt "ngac>". There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command "help" will list the available commands in the current mode. Only the most commonly needed normal mode (admin) commands are introduced here.

- `access (<policy name>, <permission triple>)` .
Check whether a permission triple is a derived privilege of the policy.
- `admin.`
Switch to admin (normal) user mode.
- `advanced.`
Switch to advanced user mode.
- `aoa (<user>)` .
Show the user accessible object attributes of the current policy.
- `combine (<policy name 1>, <policy name 2>, <combined policy name>)` .
Combine two policies to form a new combined policy with the given name.
- `echo (<string>)` .
Print the argument string, useful in command procedures.
- `halt.`
Exit the policy tool. (Will also terminate spawned server.)
- `help.`
List the commands available in the current mode.
- `help (<command name>)` .
Give a synopsis of the named command.
- `import_policy (<policy file>)` .
Import a declarative policy file and make it the current policy.
- `newpol (<policy name>)` .
Set the named policy to be the new current policy.
- `nl.`
Print a newline, useful in command procedures.

- `proc (<procedure name> [, step]) .`
Run the named command procedure, optionally pausing after each command.
- `proc (<procedure name> [, verbose]) .`
Run the named command procedure, optionally verbose.
- `regtest .`
Run built-in regression tests.
- `script (<file name> [, step]) .`
Run the named command file, optionally pausing after each command.
- `script (<file name> [, verbose]) .`
Run the named command file, optionally verbose.
- `selftest .`
Run built-in self tests.
- `server (<port>) .`
Start the policy server on the given port number.
- `version .`
Display the current version number and version description.
- `versions .`
Display past and current versions with descriptions.

3.4.4 Using the ‘ngac-server’ Policy Server

Installation and operation of the Policy Server are described in Section 4.1.4.3.

The Policy Server offers two APIs, the Policy Query API and the Policy Administration API.

3.4.4.1 Policy Server startup options

When a compiled version of the policy tool or the policy server is started from the command line, several command line options (and synonyms) are recognized.

- `--token, -t <admintoken>` use the token make authenticated requests to the paapi
- `--deny, -d` respond to all access requests with deny
- `--permit --grant -g` respond to all access requests with grant
- `--import --policy --load -i -l <policyfile>` import the policy file on startup
- `--port --portnumber --pqport -p <portnumber>` server should listen on specified port number

- `--selftest -s` run self tests on startup
- `--verbose -v` show all messages

The policy administration functions should not be made available to the normal object PEPs. Rather the Policy Administration API should be accessible only to an administratively authorised user through the policy administration tool or a process with the same authorisation, and some functions such as *setpol/getpol* should be accessible only to the “shell” program that executes the NGAC client application. In this way, the “shell” that controls execution of the application would also determine the user/session and policy under which the application should execute. Only these programs should be given the administrative token that enables calls to the Policy Administration Interface to be made. If the token option is not specified when the Policy Server is started it will use a default administrative token that is set in the param.pl source file. Currently this default token is not a secret (it is ‘admin_token’) but operating in the mode with the default token is useful for development, testing and demonstrations.

3.4.4.2 Policy Query API

This interface is used by a Policy Enforcement Point to determine whether a client-requested operation is supported by the associated user’s permissions on the requested object under a particular policy, and if the operation is permitted where may the object be accessed through an appropriate Resource Access Point (RAP).

It is a relatively simple interface, in the form of RESTful APIs. The APIs are invoked by making an HTTP GET request to a URL formed from the host address and a path terminated by, e.g. ‘pqapi/access’ and with the arguments appended to the URL with URL-encoding after a ‘?’ separator. To illustrate, the following curl command invokes the access API of a Policy Server running on the local host at port 8001:

```
curl 'http://127.0.0.1:8001/pqapi/access?user=u1&ar=r&object=o1'
```

A “failure” response by one of these APIs is typically preceded by a string indicating the reason for the failure.

pqapi/access – test for access permission under current policy

Parameters

- user = <user identifier>
- ar = <access right>
- object = <object identifier>

Returns

- “permit” or “deny” based on the current policy
- “no current policy” if the server does not have a current policy set

Effects

- none

pqapi/getobjectinfo – get object metadata

Parameters

- object = <object identifier>

Returns

- “object=<obj id>,oclass=<obj class>,inh=<t/f>,host=<host>,path=<path>,basetype=<btype>,basename=bname>”

Effects

- none

An active session identifier may be used as an alternative to a user identifier in an access query made to the Policy Query Interface.

3.4.4.3 Policy Administration API

This relatively simple interface, in the form of RESTful APIs, is invoked by making an HTTP GET request to a URL formed from the host address and a path terminated by, e.g. ‘paapi/getpol’ and with the arguments appended to the URL with URL-encoding after a ‘?’ separator. To illustrate, the following curl command invokes the getpol API of a Policy Server running on the local host at port 8001 using the default administrative token to access the administration API:

```
curl -G "http://127.0.0.1:8001/paapi/getpol" -data-urlencode "token=admin_token"
```

A “failure” response by one of these APIs is typically preceded by a string indicating the reason for the failure.

paapi/getpol – get current policy being used for policy queries

Parameters

- token = <admin token>

Returns

- <policy identifier> or “failure”

Effects

- none

paapi/setpol – set current policy to be used for policy queries

Parameters

- token = <admin token>
- policy = <policy identifier>

Returns

- “success” or “failure”
- “unknown policy” if the named policy is not known to the server

Effects

- sets the server’s current policy to the named policy

paapi/add – add an element to the current policy

Parameters

- token = <admin token>
- policy = <policy identifier>
- polycylement = <policy element> only user, object, and assignment elements as defined in the declarative policy language; restriction: only user to user attribute and object to object attribute assignments may be added. Elements referred to by an assignment must be added before adding an assignment that refers to them.

Returns

- “success” or “failure”

Effects

- The named policy is augmented with the provided policy element

paapi/delete – delete an element from the current policy

Parameters

- token = <admin token>
- policy = <policy identifier>
- polycyelement = <policy element> permits only user, object, and assignment elements as defined in the declarative policy language; restriction: only user-to-user-attribute and object-to-object-attribute assignments may be deleted. Assignments must be deleted before the elements to which they refer.

Returns

- “success” or “failure”

Effects

- The specified policy element is deleted from the named policy

paapi/load – load a policy file into the server

Parameters

- token = <admin token>
- policyfile = <policy file name>

Returns

- “success” or “failure”

Effects

- stores the loaded policy in the server
- does NOT set the server’s current policy to the loaded policy

paapi/combinepol – combine policies to form new policy

Parameters

- token = <admin token>
- policy1 = <first policy name>
- policy2 = <second policy name>
- combined = <combined policy name>

Returns

- “success” or “failure”
- “error combining policies” if the combine operation fails for any reason

Effects

- the new combined policy is stored on the server

paapi/unload – unload a policy from the server

Parameters

- token = <admin token>
- policy = <policy name>

Returns

- “success” or “failure”
- “unknown policy” if the named policy is not known to the server

Effects

- the named policy is unloaded from the server
- sets the server’s current policy to “none” if the unloaded policy is the current policy

paapi/initsession – initiate a session for user on the server

Parameters

- token = <admin token>
- session = <session identifier>
- user = <user identifier>

Returns

- “success” or “failure”
- “session already registered” if already known to the server

Effects

- the new session and user is stored

paapi/endsession – end a session on the server

Parameters

- token = <admin token>
- session = <session identifier>

Returns

- “success” or “failure”
- “session unknown” if not known to the server

Effects

- the identified session is deleted from the server

3.4.4.4 Dynamic Policy Change

The Policy Server does support *dynamic total policy change*: the ability to load new policies, to form new policies composed of already loaded policies, and to select from among the loaded or composed policies that policy which is to serve as the policy used to make policy decisions. A policy selection remains in effect until a subsequent policy selection. The server retains all of the loaded and composed policies for the duration of its execution.

In addition, the ‘ngac-server’ offers *limited dynamic selective policy change* after a policy is loaded or formed by combining policies. The *add* and *delete* APIs provide this capability. Details of the limitations are provided in the description of the APIs.

The current implementation of policy storage in the PIP is ephemeral. There is no persistence of the policy database except in the original policy file(s) used to initialize the server and the sequence of commands issued to the server to incrementally modify policies after loading of policy files. Policies and policy modifications should be managed accordingly externally to the NGAC software.

3.4.4.5 Policy composition

The policy server supports two forms of policy composition. The first is achieved with the *combinepol* API. It forms the composition of policies as described in the NGAC literature and examples.

The ‘all’ policy composition is a distinct form of policy composition. When the policy server’s current policy is set to ‘all’ through the *setpol* API, all currently loaded policies are automatically combined for every *access* request. The manner in which the policies are combined is as follows:

- Every policy is first qualified to participate in computing the verdict of an *access* request. To qualify a policy must be defined to have explicit jurisdiction over both the user and the object specified in the *access* request. There must be at least one qualifying policy.

- All qualified policies are queried with the triple (user, access right, object) specified in the *access* request. If any qualified policy returns ‘deny’ then the *access* request returns ‘deny’.

Sets of policies to be combined according to the ‘all’ policy composition should be designed with the foregoing runtime semantics taken into consideration.

3.4.5 Policy Enforcement Point (PEP) / Resource Access Point (RAP) design pattern

Figure 3-8 highlights the PEP/RAP design pattern for the resource access path that is part of the earlier NGAC functional architecture figure. The developer is able to construct all of the NGAC components along the resource access path and test the application without involvement of the NGAC server, if necessary. The necessary components, the Policy Enforcement Points (PEP) and Resource Access Points (RAP), are small trusted components that consist primarily of code that probably exists already in some form within the application. The developer must extract these relatively small bits of code from the application and place them into separate executables for PEPs and RAPs. In some cases it may be convenient to combine the PEP and RAP into a single executable, particularly when they are paired to deal with a single resource kind.

The PEP must invoke the Policy Decision Point (PDP) through its Policy Query Interface to make policy-based decisions about requests for operations on resources that a Client Application (CA) presents to the PEP.

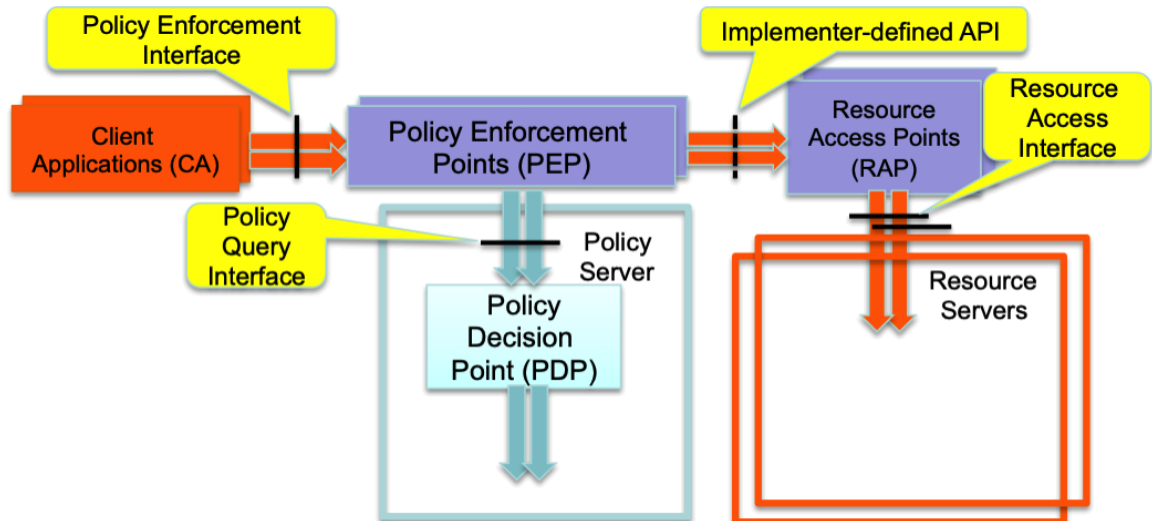


Figure 3-8 PEP / RAP design pattern for the resource access path

3.4.5.1 Policy Enforcement, PEP-to-RAP, and Resource Access APIs

The Policy Enforcement Interfaces and interfaces between the PEP and the RAP are under the control of the developer. The Resource Access Interfaces are those provided by the existing Resource Servers.

Our NGAC implementation provides simple examples for **PEPs and RAPs** that application developers can use as guidance to add new kinds of protected resources.

3.4.5.2 Web service PEPs

Other arrangements of the PEP/RAP and resource server may be useful, for example in the case that the resource is a Web service. In one such arrangement the PEP and RAP are packaged with the Web service (which is the resource server in this case) and the RAP is degenerate. In another arrangement the PEP is presented as a proxy for the actual Web service and the RAP is packaged with the Web service. Only the proxy may have permission to call the actual RAP and Web service. This may be accomplished by establishing a trusted channel between the proxy PEP and the RAP.

In every case, though, however the components or the resource access path are packaged, it is the PEP that calls the PDP and is responsible to enforce its decision. The PEP/proxy must have a reliable way of determining the identity of the invoking CA so that the query to the PDP is meaningful.

3.4.5.3

3.4.6 Platform protections needed to achieve non-functional properties of TOG-NGAC

Our goal in the SAFIRE project is to demonstrate a flexible attribute-based access control scheme within the SAFIRE architecture. The prototype assumes a benign environment without hostile actors. For deployment in a production environment, where hostile actors must be assumed, the previously discussed platform protections for NGAC must be implemented within the IT environment. The details of the approach to achieve this will vary according to the specific platform hardware and software components and choices of security controls made by the security architects and administrators of the enterprise system that will host SAFIRE. We now discuss the broad objectives needed to protect NGAC in a potentially hostile environment and approaches that may be taken to achieve this.

3.4.6.1 Nonfunctional Reference Monitor properties

NGAC applies the concept of a Reference Monitor, which beyond the correctness of its functional properties, should exhibit the nonfunctional properties: 1) that it be tamperproof, and 2) that it must always be invoked for any operation on a protected resource. (This property is sometimes referred to as non-bypassability.) Both of these properties are achieved not within the reference monitor implementation itself, but by the architecture within which the reference monitor is embedded, and by support by its operating environment.

Tamperproofness of the NGAC implementation guarantees that the behaviour of the reference monitor cannot be subverted. It requires that unauthorized parties can not modify the NGAC implementation, cannot modify the source, rebuild and install the executables, or otherwise modify the installed executables to change their behaviour.

Non-bypassability of the NGAC implementation requires that the features and configuration of the operating environment guarantee an architecture that precludes the

possibility that any subject executing within the operating environment may access the resources to be protected by NGAC except by a pathway in which NGAC is consulted to make a determination, according to its configured policy, whether or not to allow the access, and that the parameters of such queries be reliable and unmodified in a way that would compromise the integrity of the query, and that the verdict of that determination is faithfully carried out (by the Policy Enforcement Point).

3.4.6.2 NGAC component and interaction integrity

The Functional Architecture of NGAC identifies a collection of components that provide specific functions and interact in specific ways to achieve the specified functional behaviour of the reference monitor. In addition to the integrity of the components and of their operation, the integrity of the interactions are necessary to guarantee the integrity of the overall operation of NGAC.

When an individual component is packaged as a unit of execution provided by the operating environment, such as a process, then the property of process isolation and integrity of process execution can provide integrity of the component.

When components are aggregated within a unit of execution, then the property of process isolation and integrity of process execution can provide integrity of component interaction. However, it is neither practical nor desirable to aggregate all the components for a variety of reasons. For example, by factoring the functional architecture into multiple processes, it is possible to employ the principle of compartmentalisation to the reference monitor implementation, insulating the components from one another and thereby reducing the complexity of each execution unit and mitigating the damage that may result from a component failure. Further, the weaker coupling facilitates components that are individually implementable and maintainable.

As soon as, and to the extent that, NGAC components are allocated to distinct processes then the interactions among the components must occur over inter-process communication channels. Operating environments may provide inter-process communication mechanisms that are protected by process isolation features, such as pipes. In this case the operating environment typically provides integrated features, such as maintaining process and object attributes and ownership, and access control features based on those attributes, and services for identification and authentication of users in conjunction with ownership.

However, it is often the case that components may need to run on different systems depending upon where the subjects run and the objects are maintained. When the processes are distributed over multiple systems the inter-process communication involves networking. Network programming techniques have overtaken the use of strictly local inter-process communication mechanisms in many cases, since network-based inter-process communication, such as sockets, can be used independently of whether the communicating processes are on the same or different systems. Such mechanisms, such as sockets and ports, introduce new communication integrity issues when the environment is not benign.

There are multiple approaches to achieving communication integrity using network sockets for example, but these introduce entirely new layers of complexity and mechanisms to the setup and administration of systems that utilize secured sockets. For example, the generation of cryptographic keys, the selection of cryptographic mechanisms, and the distribution and management of keys are necessary.

3.4.6.3 *Reliable identities*

The issue of network communication integrity is entwined with the services of identification and authentication across systems, which may be handled by entirely separate third-party mechanisms that are merely supported by the operating system.

While the choice of *identification* and *authentication* solutions, as well as that of network integrity solutions, and their deployment is complex, costly, and labor-intensive to deploy, it is also routinely the province of enterprise ICT architecture and operations departments to perform this activity, and appropriate products and techniques are well understood by those practitioners. Consequently, on the SAFIRE project we focused on the novel aspects of *authorisation* as provided by NGAC, and did not pursue identification, authentication, identity management, cryptographic functions, and key management, because while these are complex and costly, they are routinely handled by an enterprise ICT department in ways that are done consistently across the enterprise as part of the platform that hosts SAFIRE and its NGAC-based policy features. Section 4.1.4.6 provides some guidance for achieving the desired non-functional properties of NGAC through functional requirements on its ICT environment.

4. INSTALLATION AND CONFIGURATION OF THE SAFIRE PLATFORM

This section includes the guidelines to deploy (i.e. install and configure) the Final Integrated Cloud Platform (FICP) of SAFIRE. The following diagram shows the different elements and services to deploy:

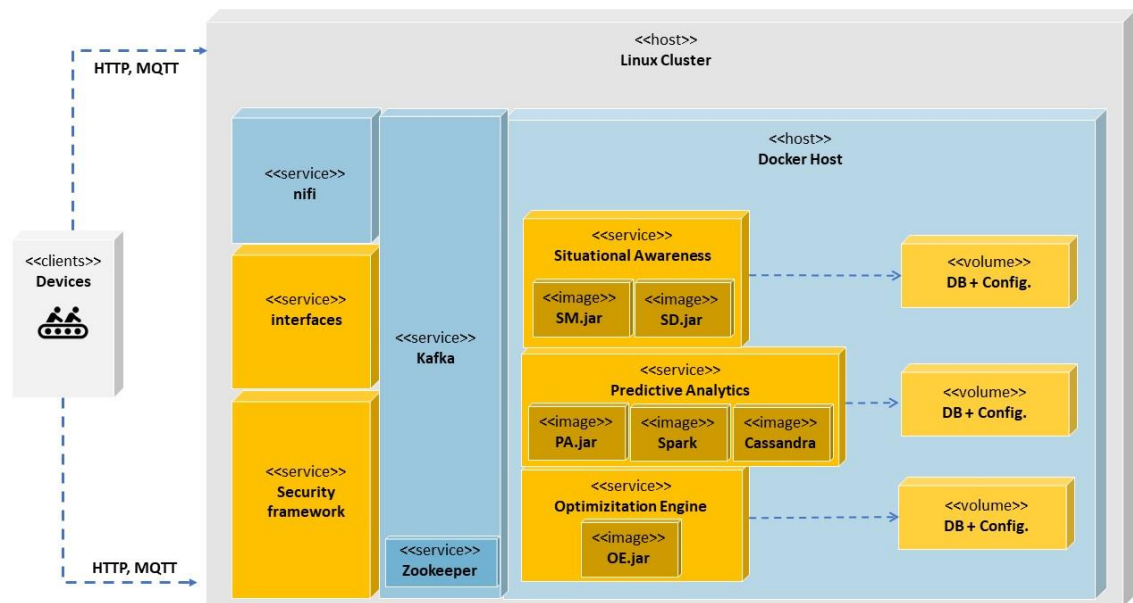


Figure 4-1 SAFIRE FICP Deployment Diagram

Firstly, the section describes how to proceed with the installation of the basic components of the SAFIRE cloud architecture, namely **NiFi**, **Docker** (including Docker compose) and **Kafka** (blueish components in the previous diagram). Then, the different subsections include guidelines for the deployment of each SAFIRE service (orange components).

The equipment used to host the FICP is a 16 core server processor with 32 GB RAM , RAID 5 with 500GB for the OS and an iSCSI-connected NAS that enables a maximum storage capacity of 12 TB. The OS installed in the server is Linux Ubuntu Bionic 18.04.1 LTS.

For the setup and configuration of the FICP environment the following resources are required:

- an industry expert responsible for the equipment operation to configure the equipment (either bare metal or virtualized) for the installation of the operating system, required services and the environment of the container.
- an industry expert for operations (or development) responsible for the containerisation of the applications, i.e. responsible for the elaboration of the YAML files that allow the deployment of the containers.

- an industry expert responsible for adapting SAFIRE ontology to each of the business cases.
- an industry expert responsible part of the development staff responsible for the adaptation of the data ingestion service to the specific requirements of the business cases.

4.1 INSTALLATION AND CONFIGURATION OF THE BASIC SAFIRE ARCHITECTURE COMPONENTS

The general architectural components to install are *Apache NiFi*, *Docker*, *Apache Kafka* and the *Security Framework* that will run as stand-alone services in the FICP. Docker will be used in the FICP to host the SAFIRE services.

4.1.1 Apache NiFi

NiFi is installed as a stand-alone service directly in the server. In order to install NiFi the following steps shall be followed:

1. Download the “tarball” NiFi file for Linux systems from NiFi downloads page:

<https://nifi.apache.org/docs/nifi-docs/html/getting-started.html#downloading-and-installing-nifi>

2. Extract the downloaded file to a selected directory

```
user@ubuntu:~$ tar -xvf nifi-1.7.1-bin.tar.gz etc/safire/bin
```

3. Configure NiFi modifying the *nifi.properties* file found in the installation folder.

- Configuration best practices for linux can be found here:
<https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#configuration-best-practices>
- Configuration of security parameters can be found here:
<https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#security-configuration>

4. To run NiFi as a service, execute the following command from the /bin directory of the installation:

```
user@ubuntu ./bin:~$ ./nifi.sh install
```

5. NiFi service can be stopped with:

```
user@ubuntu ./bin:~$ service nifi stop
```



6. NiFi service can be restarted with:

```
user@ubuntu ./bin:~$ service nifi start
```

7. Or status queried with:

```
user@ubuntu ./bin:~$ service nifi status
```

4.1.2 Apache Kafka

This section includes step-by-step guidelines to have Kafka running as a service in Ubuntu Linux.

1. As Kafka is written in Java, JDK needs to be installed first. After updating and upgrading a fresh installation of Ubuntu, by issuing:

```
user@ubuntu:~$ sudo apt update
user@ubuntu:~$ sudo apt upgrade
```

The user should install Standard Java or Java compatible Development Kit:

```
user@ubuntu:~$ sudo apt install default-jdk
```

2. As Kafka uses ZooKeeper for cluster manager, it has to be installed and started. After its installation ZooKeeper will be automatically started as a daemon.

```
user@ubuntu:~$ sudo apt-get install zookeeperd
```

3. To verify the success of the installation, the tool named netstat can be used to monitor the activity on default ZooKeeper port 2181. In Figure 4-2, an expected output of the appropriate netstat command is presented:

```
user@ubuntu:~$ netstat -ant | grep :2181
```

4. However, netstat is not present in a fresh installation of Ubuntu 18.04 by default and should be installed first:

```
user@ubuntu:~$ sudo apt install net-tools
```

```
leandro@safirevm: ~
File Edit View Search Terminal Help
leandro@safirevm:~$ netstat -ant | grep :2181
tcp6      0      0 :::2181          :::*              LISTEN
leandro@safirevm:~$
```

Figure 4-2 Confirmation of the successful ZooKeeper installation with netstat

5. Kafka can be downloaded to e.g. folder *kafka* in the home directory from Apache website using *wget*:

```
user@ubuntu:~$ mkdir kafka && cd kafka && wget
http://apache.mirrors.nublu.co.uk/kafka/1.1.0/kafka_2.12-1.1.0.tgz
```

6. Decompress the downloaded file:

```
user@ubuntu:~$ tar -xvf kafka_2.12-1.1.0.tgz && cd kafka_2.12-1.1.0
```

7. And start the service:

```
user@ubuntu:~$ sudo bin/kafka-server-start.sh config/server.properties
```

Due to a bug in the latest version in the script located in `bin/kafka-run-class.sh` Kafka may not start properly with Java 9, what is shown in Figure 4-3. In this situation, one line in file `bin/kafka-run-class.sh` needs to be modified from:

```
JAVA_MAJOR_VERSION=$(($JAVA -version 2>&1 | sed -E -n 's/.* version
"([^-]*)\.*/\1p')
```

To:

```
JAVA_MAJOR_VERSION=$(($JAVA -version 2>&1 | sed -E -n 's/.* version
"([^-]*)\1p')
```

i.e. the second quotation mark needs to be removed.

```
leandro@safirevm: ~/kafka/kafka_2.12-1.1.0
File Edit View Search Terminal Help
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-server-start.sh /opt/kafka/
kafka_2.12-1.1.0/config/server.properties
/home/leandro/kafka/kafka_2.12-1.1.0/bin/kafka-run-class.sh: line 252: [: 10 20
18-04-17: syntax error in expression (error token is "2018-04-17")
[0.000s][warning][gc] -Xloggc is deprecated. Will use -Xlog:gc:/home/leandro/kaf
ka/kafka_2.12-1.1.0/bin/../logs/kafkaServer-gc.log instead.
Unrecognized VM option 'PrintGCDateStamps'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$
```

Figure 4-3 Possible unsuccessful execution of Kafka with Java 9

8. Successful execution of Kafka can be confirmed by creating a testing topic, e.g. with:

```
user@ubuntu:~$ bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic testing
```

which should be notified with information *Created topic "testing"* (see Figure 4-4).

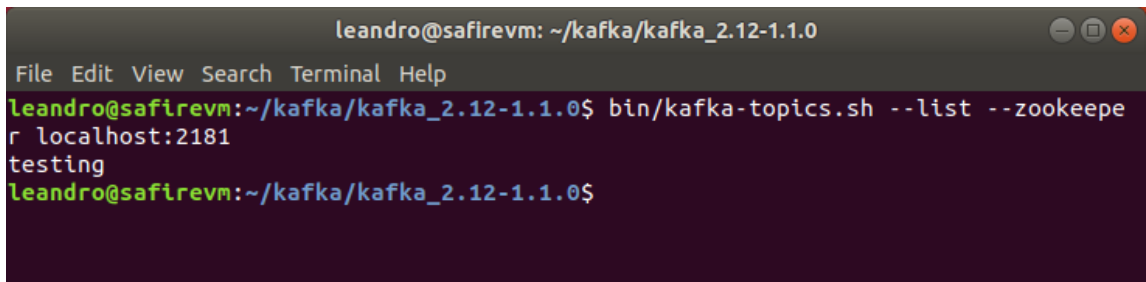
```
leandro@safirevm: ~/kafka/kafka_2.12-1.1.0
File Edit View Search Terminal Help
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-topics.sh --create --zookee
per localhost:2181 --replication-factor 1 --partitions 1 --topic testing
Created topic "testing".
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$
```

Figure 4-4 Creation of a topic in Kafka

9. The Kafka topics can be listed using the following command:

```
user@ubuntu:~$ bin/kafka-topics.sh --list --zookeeper localhost:2181
```

as shown in Figure 4-5.



```
leandro@safirevm: ~/kafka/kafka_2.12-1.1.0
File Edit View Search Terminal Help
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-topics.sh --list --zookeeper localhost:2181
testing
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$
```

Figure 4-5 Listing Kafka topics

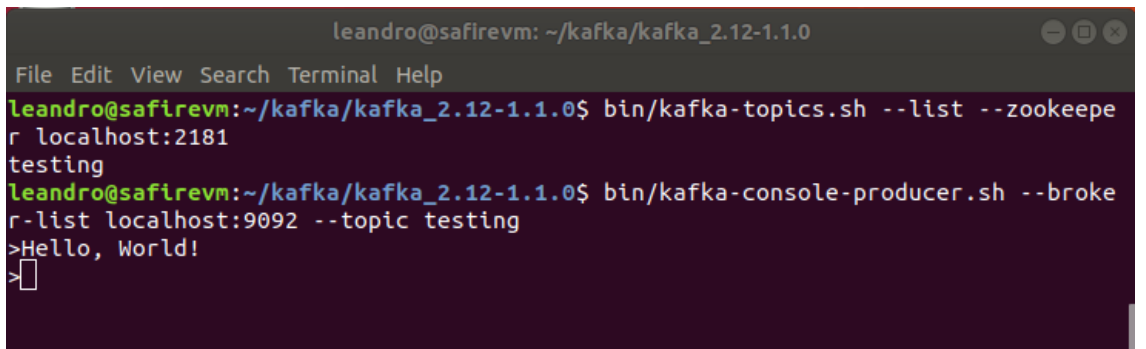
10. A console-based Kafka producer can be created using the command:

```
user@ubuntu:~$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic testing
```

whereas the corresponding consumer with:

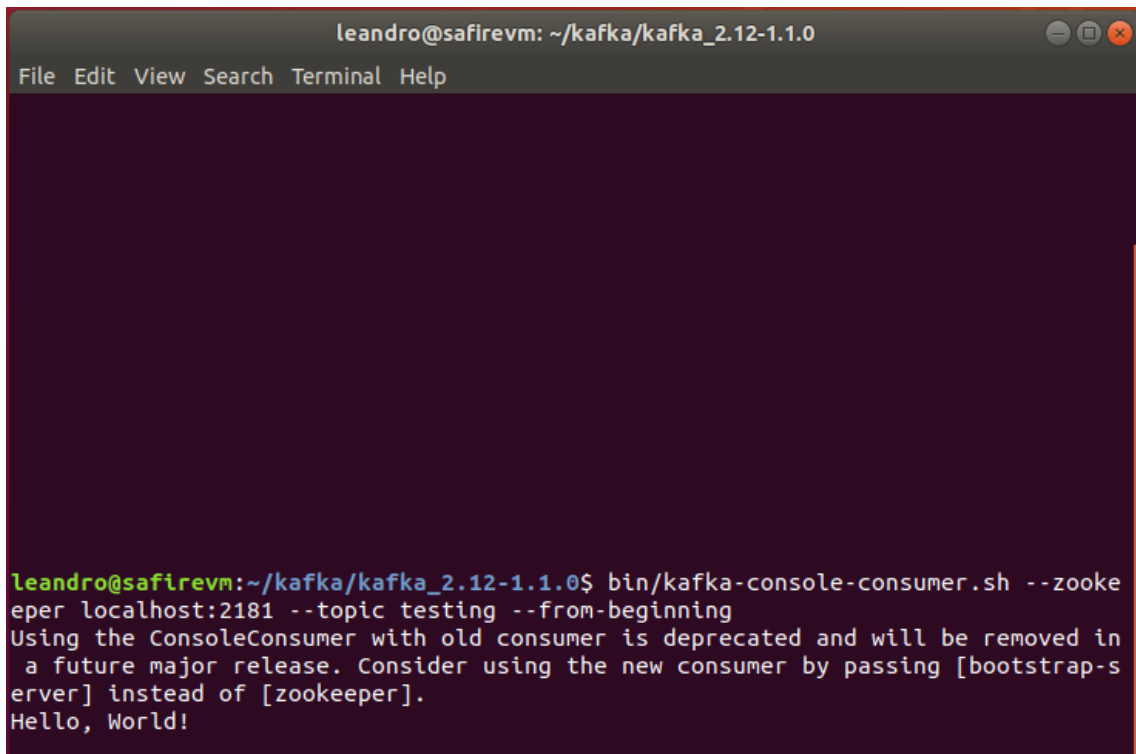
```
user@ubuntu:~$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic testing --from-beginning
```

what is shown in Figure 4-6 and Figure 4-7, respectively.



```
leandro@safirevm: ~/kafka/kafka_2.12-1.1.0
File Edit View Search Terminal Help
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-topics.sh --list --zookeeper localhost:2181
testing
leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic testing
>Hello, World!
>
```

Figure 4-6 Creating Kafka producer in a console



```
leandro@safirevm: ~/kafka/kafka_2.12-1.1.0
File Edit View Search Terminal Help

leandro@safirevm:~/kafka/kafka_2.12-1.1.0$ bin/kafka-console-consumer.sh --zooke
eper localhost:2181 --topic testing --from-beginning
Using the ConsoleConsumer with old consumer is deprecated and will be removed in
a future major release. Consider using the new consumer by passing [bootstrap-s
erver] instead of [zookeeper].
Hello, World!
```

Figure 4-7 Creating Kafka consumer in a console

4.1.3 Docker, Docker Compose and Docker Registry

Docker (<https://www.docker.com>) is a platform that can run in bare-metal or virtualised servers and allows the creation and management of application containers. A container is a lightweight, standalone, executable package of software that includes everything it needs to run an application. Containers are volatile, meaning that data and states are not stored unless specific volumes are created for the containers.

All SAFIRE services are deployed in a docker host as docker containers together with their associated volumes. Additionally, other general-purpose services (such as Kafka) are also be deployed as containers in the full integrated cloud prototype. Since most of SAFIRE services are multi-container docker applications, it is necessary to install Docker Compose.

Containers executed in a docker host must be previously registered either in a public or a private registry, so it is be necessary to understand how to reach the registry, create images in it and make use of them when necessary.

The present section shows the steps required to install docker and docker compose, and how to manage the images from the container registry (available in of GitLab repository the project).

4.1.3.1 Docker

The Docker Community Edition (Docker CE) for Ubuntu is manually installed after downloading the latest available package from: <https://download.docker.com/linux/ubuntu/dists/bionic/stable/>

1. To install docker and automatically execute it as a daemon, execute the following command in the desired path:

```
user@ubuntu ./docker:~$ sudo dpkg -i package.deb
```

2. Verify the installation by running the hello-world image included:

```
user@ubuntu ./docker:~$ sudo docker run hello-world
```

3. Configure docker to start as a service on every new boot with:

```
user@ubuntu ./docker:~$ sudo systemctl enable docker
```

4.1.3.2 Docker Compose

In order to install Docker Compose the following steps shall be followed:

1. Run this command to download the latest version (available in [Compose repository release page on GitHub](#)) of Docker Compose:

```
user@ubuntu:~$ sudo curl -L "https://github.com/docker/.../docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. Apply executable permissions to the binary:

```
user@ubuntu:~$ sudo chmod +x /usr/local/bin/docker-compose
```

3. Test the installation:

```
user@ubuntu:~$ docker-compose --version
```

It should retrieve the docker-compose version installed.

4.1.3.3 Docker Container Registry

As mentioned, all SAFIRE Services will be deployed as containers. Deployment will be made from a common repository of containers (public docker registry) available from the project's GitLab. To manage images, the user must login first:

```
user@ubuntu:~$ sudo docker login ikerlan.github.io:4678
```

Once logged in, the user can create images:

```
user@ubuntu:~$ sudo docker build -t ikerlan.github.io:4678/public-repos/<name-of-service>
```

Or update them:

```
user@ubuntu:~$ sudo docker push ikerlan.github.io:4678/public-repos/<name-of-service>
```

Each SAFIRE Service has its own `docker-compose.yml` file, which indicates the source of the necessary images for the service to operate. To deploy each service simply execute the following command from each the folder of each SAFIRE Service:

```
user@ubuntu:~$ docker-compose up
```

4.1.4 Security Framework

This installation and configuration guide provides an update to and elaboration of some of the related information in D5.4 Full Prototype of the SPT Framework.

4.1.4.1 Prerequisites and Dependencies

The 'ngac' Policy Tool and the 'ngac-server' Policy Server are implemented in the Prolog language and require the SWI-Prolog environment to run. SWI-Prolog is available for several operating environments, including Microsoft Windows (64 bit) and (32 bit), MacOS X 10.6 and later on Intel, and several Linux versions including Ubuntu. It is also available in Docker containers and as a source distribution that one can build locally.

Our NGAC implementation uses *only* the libraries that come with the SWI-Prolog distribution. Furthermore we've organised the functional architecture so as to place adaptation into the hands of application developers without requiring modification to the core implementation. This is in contrast to past reference implementations that have

had many external dependencies (some now obsolete) so that they were very cumbersome to work with and not very portable or adaptable.

SWI-Prolog is available from www.swi-prolog.org; we are currently using version 8.0.3. The installation and build instructions below include the installation of SWI-Prolog.

4.1.4.2 Installation and build

The TOG-NGAC software is provided as a set of Prolog source files from which can be built an “executable” that has the compiled code packaged with the Prolog runtime environment. The shell script `mkngac` compiles the code and creates the executables for the ‘ngac’ Policy Tool and for the ‘ngac-server’ Policy Server. To build the executables requires that the SWI-Prolog environment is installed. The executable is not distributed since it should be created in the target environment.¹ An executable may be deployed to other systems that offer the same target environment.

The following elements shall be installed and configured:

1. **Install SWI-Prolog** stable version from Linux Ubuntu Personal Package Archive (<http://www.swi-prolog.org/build/PPA.html>)

```
user@ubuntu:~$ sudo apt-get install software-properties-common
user@ubuntu:~$ sudo apt-add-repository ppa:swi-prolog/stable
user@ubuntu:~$ sudo apt-get update
```

2. Install the **NGAC source files**. The current version of the ‘ngac’ software consists of a directory tree including source files, example policy files, example data files, and test files. The distribution is provided as a zip file available from SAFIRE’s GitLab repository.
3. **Build the NGAC executables**. In the root directory of the TOG-NGAC release run the shell script `mkngac`.

```
user@ubuntu:~$ chmod +x /ngac/directory/
user@ubuntu ngac/directory:~$ ./mkngac
```

A listing of the directory should show that new ‘ngac’ and ‘ngac-server’ files have been created. These files are shell scripts that should be given execute permission.

¹ If the distribution should be found to contain copies of ‘ngac’ and ‘ngac-server’ these should be discarded and rebuilt.

4.1.4.3 Testing and Using NGAC

4. **Initiate the ‘ngac’ Policy Tool.** The executable can be executed directly from a command shell prompt.

```
user@ubuntu:~$ chmod +x /ngac/directory/ngac
user@ubuntu:~$ cd /ngac/directory
user@ubuntu ngac/directory:~$ ngac
```

The ‘ngac’ prompt “ngac> “ should appear. Note that ‘ngac’ commands entered at this prompt should be terminated with a full stop (“.”).

5. **Test** the installed NGAC tool. The ngac tool has built in self-tests to ensure that the algorithms are working correctly. Start self-test executing the following command in NGAC’s command line:

```
ngac> selftest.
```

All tests run should indicate success (the expected result was detected).

6. NGAC installation can also be tested by executing the example procedures found in the “procs.pl” file, or by adding new procs there, and then running:

```
ngac> proc(MyProc) .
or
ngac> proc(MyProc, verbose) .
```

7. **Import** one or more policy files:

```
ngac> import(policy(PolicyFileName)) .
```

where PolicyFileName is the name of the policy language .pl file relative to the execution directory. Place the PolicyFileName in single quotes if it begins with a non-lowercase letter or contains punctuation other than underscore (“_”).

8. **Select** a previously imported policy to be the current policy using the command newpol.

```
ngac> newpol(PolicyName) .
```

where `PolicyName` is the name of the policy occurring in the policy specification contained in the policy file (it is *not* the file name).

9. Queries to determine whether a particular user may perform a particular operation on a particular object may be run against the current policy using the `access` command.

```
ngac> access(PolicyName, (User,Op,Object)) .
```

where `(User,Op,Object)` is a permission triple to be checked against the named policy.

10. See all available ‘ngac’ commands by entering the command `help` or get further details on a single command using the command `help(CommandName)`.

```
ngac> help(access) .
```

11. Start the Policy Server using the command `server(PortNumber)`.

```
ngac> server(PortNumber) .
```

where `PortNumber` is an unused TCP port number.

12. Alternatively, start the Policy Server directly, apart from the ‘ngac’ Policy Tool, by executing the ‘ngac-server’ file with chosen command-line options.

```
user@ubuntu ngac/directory:~$ ngac-server -p PortNumber -i PolicyFile-Name
```

This is the preferred way to initiate the server for production use. Consult the user manual for the command-line options available when starting the server in this way. Options include those shown here for specifying a port and loading of a policy, or setting the current policy to a built-in ‘grant’ or ‘deny’ policy that always returns ‘grant’ or ‘deny’ (respectively) to any access query.

13. Test the running Policy Server (started by either of the methods above) by running the shell script `servercurltest.sh` found in the TEST subdirectory. This script sends a sequence of requests to the server to test for known correct answers. The script `serverCombinedtest.sh` also found in the TEST subdirectory tests a

composed policy example. The script uses the `curl` utility to form and send the requests to the server's Web APIs.

The Policy Server is typically used by first configuring its policy, either through the command line option or using the Policy Administration Interface, and then by creating Policy Enforcement Points that query the Policy Query Interface for individual access requests made to the PEP by a client application.

4.1.4.4 Operation of the Policy Server

For productions use, the Policy Server executable may be started directly by an initialisation script that invokes it, along with chosen command line options, in the background (e.g. using a `'&'` in a shell script). The server logs messages to its standard output stream, which may be redirected to a file if desired.

4.1.4.5 Configuration of NGAC

TOG-NGAC can be easily extended in several ways.

- Commands can be added to the 'ngac' Policy Tool by modifying the `command` module to add `syntax`, `semantics` (optional), `help`, and `do` clauses for the new command. A `syntax` clause must be added for the command. This clause declares the command name and parameters, and what mode the command belongs to, `admin` or `advanced`. Admin commands are available in admin mode, but also accessible in the advanced mode but not vice versa.
- The self-test framework is implemented in the `test` module. Tests for specific new modules can be added in the `TEST` subdirectory. An example of a test definition file for the `spld` module is implemented in `TEST/spld_test.pl`.
- New predefined 'ngac' command procedures can be added to the `procs` module. A `proc` clause is added for each new procedure to be defined. There are examples in the `procs.pl` file.

Global parameters are set in the `param` module. Settable parameters (those that can be changed from the 'ngac' command line with the `set` command) are itemized in the list `settable_params`. Adding new settable parameters requires the new parameter name to be added to this list and to the `dynamic` directive above it in a fashion similar to the other entries.

4.1.4.6 Guidance for achieving non-functional properties for TOG-NGAC

Earlier in the User Manual, section 3.4.6, we discussed the non-functional requirements of an NGAC deployment in a non-benign environment. NGAC's non-functional properties may be achieved through *functional* requirements on NGAC's *operating environment*.

Operating Environment Protections

Since NGAC is not a native feature of the operating environment, it is dependent on basic features of the operating environment for its operational integrity within a computer system. Such features include:

- File system attributes, ownership, and permissions
- Process isolation and integrity

When components of the NGAC functional architecture are packaged together within a process, as are the PDP, PAP, and PIP components within the Policy Server, the operating environment's process isolation features are relied upon to prevent tampering with the running process and to protect intra-process communication among the components.

When components of the NGAC functional architecture access resources to which NGAC is mediating access, for example, a RAP accessing a local file through the file system interfaces, the file system access controls are leveraged. This is done in a very simple and fixed way, to limit access to the files exclusively to the NGAC components, and the RAP in particular. This delegates the flexibility of access control to the unified NGAC mechanism and the policies with which it is configured. This may be achieved, for example, by setting the ownership of the protected resources to a unique ngac-rap owner and by setting the executable containing the RAP (may be bundled with a PEP) to be set UID (set user ID on execution attribute of the file containing the executable) to the user ngac-rap. Since the PEP-RAP combination, invoked for resource operations by a client application (CA), is protected by process isolation, and since the PEP consults the Policy Server's PDP before carrying out a resource operation, the access to the resource by the CA is effectively mediated by NGAC.

Trusted Channels

By using TCP sockets for the inter-component interactions among process units in our implementation of NGAC, such as between a CA and a PEP and between the PEP and the Policy Server, we have set the stage for the extension to NGAC of enterprise solutions for establishing trusted channels among processes on the same or different systems. We defer to the enterprise ICT department for the choice and deployment of features to provide communication integrity in non-benign environments.

Identities of Users and Objects

When a solution for inter-system identity, authentication, and secure communication has been selected and deployed, the user identities within the identity management system should be reconciled with the users and objects declared and used in NGAC policies.

4.2 INSTALLATION AND CONFIGURATION OF THE SAFIRE DASHBOARD

The SAFIRE dashboard aims to allow the user to observe the operation of the different services of the platform. For each one of the main modules of the platform, namely the Situation Determination (SD), the Predictive Analytics (PA), the Optimisation Engine (OE) and the Security Framework (SPT), a card-like screen appears in the main screen of the dashboard, showing live information coming from the running services of the modules.

In order to install and run the SAFIRE dashboard the same procedure for installing and running any other SAFIRE docker image, as described before in section 4.1, should be followed. The kafka messaging system should run already prior to starting the dashboard service, since the dashboard observes the messages exchanged between the modules. For each one of the modules, a dedicated kafka topic is being used (namely SAFIRE_DASHBOARD_SD for the SD module, SAFIRE_DASHBOARD_PA for the PA module, SAFIRE_DASHBOARD_OE for the OE module, SAFIRE_DASHBOARD_SPT for the SPT module) where the modules publish their operations statuses and results. The messages published in those topics have the following json structure:

```
{
  "ServiceId": "number_Id_of_the_module"
  "ServiceName": "name_of_the_module",
  "ServiceStatus": "module_event_status",
  "ServiceMessage": "event_message ",
  "TimeStamp": "datetime_of_the_event"
}
```

An example of a json message for each one of the modules, in the related kafka topic, follows:

- SAFIRE_DASHBOARD_SD

```
{
  "ServiceId": "1",
  "ServiceName": "Situation Determination",
  "ServiceStatus": "monitoring",
  "ServiceMessage": "The message [monitoring] just received",
  "TimeStamp": "08.08.2019 03:23:09"
}
```
- SAFIRE_DASHBOARD_PA

```
{
  "ServiceId": "2",
  "ServiceName": "Predictive Analytics",
  "ServiceStatus": "waiting",
  "ServiceMessage": "Predictive Analytics is waiting for a new request",
  "TimeStamp": "08.08.2019 03:23:10"
}
```

- SAFIRE_DASHBOARD_OE


```
{
    "ServiceId": "3",
    "ServiceName": "Optimisation Engine",
    "ServiceStatus": "optimisation requested",
    "ServiceMessage": "Optimisation Engine is optimising",
    "TimeStamp": "08.08.2019 03:18:36"
  }
```
- SAFIRE_DASHBOARD_SPT


```
{
    "ServiceId": "4",
    "ServiceName": "Security Privacy Trust",
    "ServiceStatus": "access denied",
    "ServiceMessage": "Security [access denied]",
    "TimeStamp": "08.08.2019 03:23:12"
  }
```

Figure 4-8 SAFIRE Dashboard Screenshot shows the SAFIRE dashboard on a test execution.



Figure 4-8 SAFIRE Dashboard Screenshot

4.3 INSTALLATION AND CONFIGURATION OF DATA INGESTION AND MONITORING SERVICES

The full prototype of the integrated cloud platform implements a MQTT brokerage service to gather data directly from the connected devices and push them into the NiFi service. The final version of the cloud platform implements specific interfaces to the data repositories of each pilot, where data from the different devices are conveniently stored. The configuration of the access to each pilot's repository is described in Section 3.1.

4.4 INSTALLATION AND CONFIGURATION OF THE PREDICTIVE ANALYTICS SERVICE

The Predictive Analytics Service in the FICP is deployed as a docker application with several containers, including specific volumes to persist configuration data and data in Cassandra and PostgreSQL data bases.

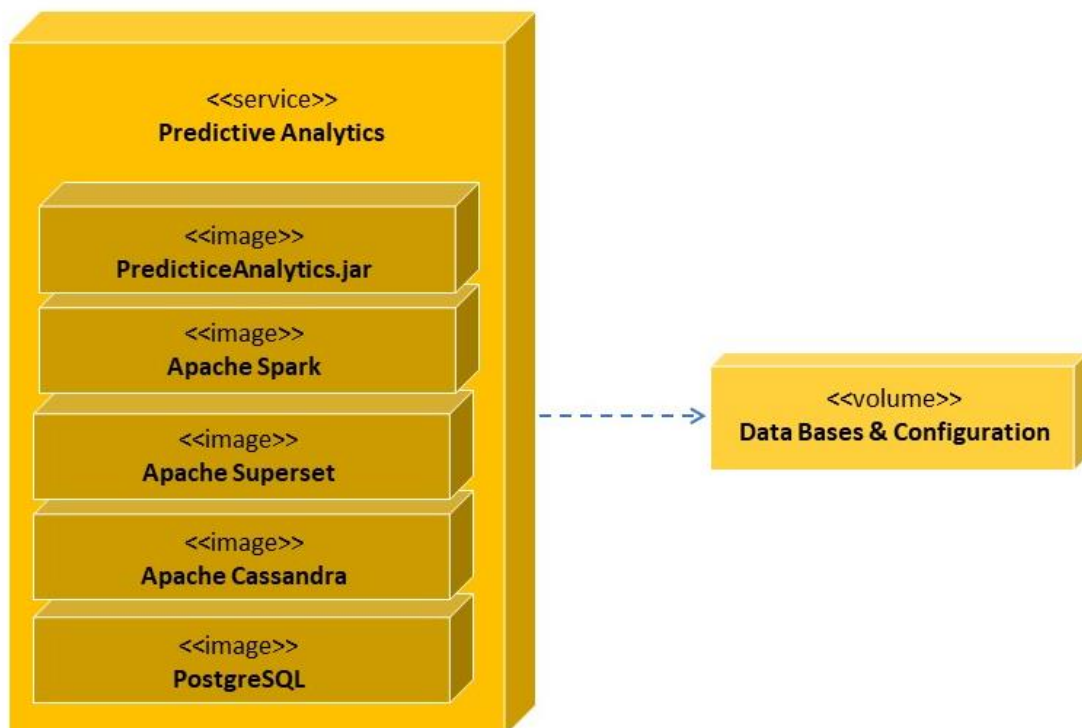


Figure 4-9 Deployment diagram of the Predictive Analytics Service

In order to deploy the Predictive Analytics service in the FICP, the following steps are followed:

1. Obtain the Predictive Analytics image from the registry:

```
user@ubuntu:~$ sudo docker pull ikerlan.github.io:4678/public-repos/safire-predictive-analytics
```

2. Deploy the service with docker-compose:

```
user@ubuntu:~$ docker-compose up
```

4.5 INSTALLATION AND CONFIGURATION OF THE SITUATIONAL AWARENESS SERVICES

Situational Awareness is deployed in the FICP as two Docker applications (Situation Monitoring and Situation Determination) with several containers, including specific volumes to persist XML configuration data and data in a H2 Database.

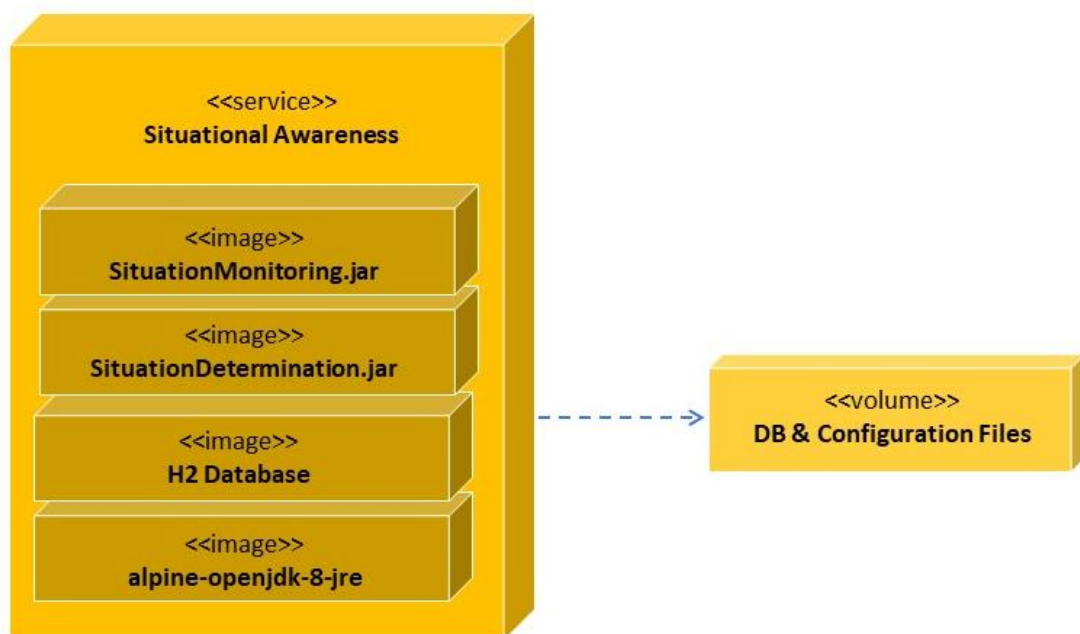


Figure 4-10 Deployment diagram of the Situational Awareness service

In order to deploy these services in the FICP, the following steps are followed:

1. Obtain the images of the business case specific situation monitoring and determination services from the registry (*\$bc* shall be substituted by one of the respective business cases *oas*, *ona* or *electrolux*):

```
user@ubuntu:~$ docker pull gitlab.atb-bremen.de:5555/safire/situation-  
monitoring-$bc
```

```
user@ubuntu:~$ docker pull gitlab.atb-bremen.de:5555/safire/situation-  
determination-$bc
```

2. Before executing the services, they are configured in a common XML configuration file. An example for such a configuration is included below:

```
<?xml version="1.0" encoding="utf-8"?>  
<config xmlns="http://www.atb-bremen.de"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  <services>  
    <service id="Monitoring">  
      <host>localhost</host>  
      <location>http://localhost:19001</location>  
      <name>MonitoringService</name>  
      <server>de.atb.context.services.MonitoringService</server>  
      <proxy>de.atb.context.services.IMonitoringService</proxy>  
    </service>  
    <service id="MonitoringRepository">  
      <host>localhost</host>  
      <location>http://localhost:19002</location>  
      <name>MonitoringDataRepositoryService</name>  
      <server>de.atb.context.services.MonitoringDataRepositoryService</server>  
      <proxy>de.atb.context.services.IMonitoringDataRepositoryService</proxy>  
    </service>  
    <service id="SituationDeterminationService">  
      <host>localhost</host>  
      <location>http://localhost:19004</location>  
      <name>ContextExtractionService</name>  
      <server>de.atb.context.services.ContextExtractionService</server>  
      <proxy>de.atb.context.services.IContextExtractionService</proxy>  
    </service>  
    <service id="SituationDeterminationRepositoryService">  
      <host>localhost</host>  
      <location>http://localhost:19005</location>  
      <name>ContextRepositoryService</name>  
      <server>de.atb.context.services.ContextRepositoryService</server>  
      <proxy>de.atb.context.services.IContextRepositoryService</proxy>  
    </service>  
  </services>  
</config>
```

3. After downloading the container image, and configuring the services, they can be started in isolated containers using the command “run”. By using this command there is the possibility to add a restart policy to allow automatic restart of the services after failure as mentioned below:

```
user@ubuntu:~$ docker run -restart unless-stopped situation-monitoring  
  
user@ubuntu:~$ docker run -restart unless-stopped situation-  
determination
```

4.6 INSTALLATION AND CONFIGURATION OF THE OPTIMISATION ENGINE

In this section, the process of building a container out of the optimisation engine sources, as stored in the SAFIRE public repository named "optimisation-engine", is described.

Fitness function evaluator, being a part of OE, is almost entirely stateless. In particular, the information about factory architecture, available recipes or ordered commodities is not stored in OE, but sent as a message triggering the optimisation process by the SD module instead. The only element that needs to be configured is related to security. As OE needs to communicate with the SPT Framework to check the access rights of each incoming optimisation request, the location of the NGAC server needs to be provided. It is done by using two string variables: *ipAddress* and *port*. The access right to be forwarded to the NGAC server is stored in string variable *ar* and the OE service name as an object in the SPT Framework is stored in the *UoYOptimisationEngine* class in package *uk.ac.york.safire.optimisation.mitm.atb*, as shown below.

```
public final static String ipAddress = "127.0.0.1";
public final static String port = "8001";
public final static String ar = "r";
public final static String object = "oe";
```

As the optimisation engine has been written in both the Scala and Java programming languages, the most popular open-source build tool for these languages is used, named "sbt". It requires Java 1.8 or later. The build instructions are given in the "build.sbt" file which ensures that the Docker image is built with all necessary dependencies. If one intends to use this file to compile the project, the following commands need to be issued:

```
user@ubuntu:~$ sbt clean
user@ubuntu:~$ sbt compile
```

To create a runnable jarfile, the project needs to be packaged with the command:

```
user@ubuntu:~$ sbt package
```

The class to be executed in the jarfile is specified in the "build.sbt" file. For example, to create an executable jarfile containing a remotely-invokable OE module, the following line should be uncommented:

```
mainClass in Compile := Some
```

```
("uk.ac.york.safire.optimisation.HttpRemoteOptimisationEngineServer")
```

In the repository, there is the file named "dockerize.sh". When executed, it dockerizes the executable jarfile, executing the following command

```
docker run --rm -p8080:8080 safire-optimisation-engine:0.4-snapshot
```

where "safire-optimisation-engine:0.4-snapshot" is the image name, derived from that of the executable jarfile.

The OE service is Business Case dependant and the corresponding FFs are hardcoded. However, the generic solution to build a customised FF for a smart factory or device based on its human-writable description is available using Factory Description Language (FDL) as explained in deliverable D5.6. The FDL description extracts for the three SAFIRE BCs are provided in section 3.3 in this deliverable.

5. CONCLUSIONS

The Final Integrated Cloud Platform (FICP) completes the previous early version with the description of the full prototype of the SAFIRE services, as well as with details for the configuration and installation of the different modules in the business case site.

The SAFIRE integrated cloud platform consists of the four main SAFIRE services, namely the Predictive Analytics (PA), the Situation Determination (SD), the Optimisation Engine (OE) and the Security Framework (SPT), and its operation is being monitored using the dashboard. All the modules have been developed using latest open source technologies, adequate for big data management in an industrial environment, and have been packaged using docker in order to be directly deployable in different operating systems. The data transfer within the SAFIRE modules and outside to its environment (and the business case legacy systems) is being utilised using NiFi templates. The communication between the modules has been established using the kafka messaging system. This technology is also being used by the dashboard to visualise the results of the module during the SAFIRE operation.

For each of the three business cases, for Electrolux, OAS and ONA, the SAFIRE integrated cloud platform has been configured and connected or integrated with the legacy systems for data exchange and process management. User installation and configuration guidelines have been provided to allow future users of SAFIRE to integrate the platform into their systems.

As future work after the end of the project, the SAFIRE consortium aims to continue testing and adjusting the operation of the platform to the respective business case needs in order to validate its operation and identify potential new opportunities for adjustment and expansion, inside and outside the selected business cases and their clients.



6. REFERENCES

- [1]. InterNational Committee for Information Technology Standards (INCITS). Information technology—Next Generation Access Control—Functional Architecture (NGAC-FA), INCITS 499-2018, ANSI 2018.