Project Number 723634

D3.5 Final Specification of Dynamic and Predictable Reconfiguration and Optimisation Engine

Version 1.0
18 October 2018
Final

EC Distribution

University of York

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

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 Arizmendiarrieta 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, 5th 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	First draft of the document.	20 August 2018
0.4	Added description of Factory Description Language.	15 September 2018
0.5	Added description of configuration deployment	17 September 2018
0.6	Update of the requirements coverage table	28 September 2018
0.7	Added relation to the generic ontology for modelling correlation	04 October 2018
0.8	Version sent to internal review	05 October 2018
0.9	Version after revision	12 October 2018
1.0	Final version	18 October 2018

TABLE OF CONTENTS

1. Introduction **1**

1.1 Progress beyond D3.2 Early Specification of Dynamic and Predictable Reconfiguration and Optimisation Engine *2*

2. Capabilities to be developed **2**

2.1 Metrics API *3*

2.2 Optimisation/Reconfiguration Engine *3*

2.3 Objective Function *3*

3. Capability State of the Art **3**

3.1 Metrics API *3*

3.2 Optimisation Engine *4*

3.3 Objective Function *5*

4. Capability New technologies / innovations **6**

4.1 Metrics API *6*

4.2 Optimisation Engine *6*

4.3 Objective Function *7*

5. Interface Specifications **8**

5.1 Metrics API *8*

5.1.1 Configuration Specification 8

5.2 Configuration Schema Interface *9*

5.3 Optimisation Engine Interface *11*

5.4 Objective Function Interface *11*

6. End-User Configuration **12**

6.1 Suitable Pre-existing Objective Function *12*

6.2 No Suitable Pre-existing Objective Function *12*

6.2.1 Explicit description of the Objective Function. 12

6.2.2 Predicted Objective Function from analysis of process data. 13

6.3 Plant Description for the Optimisation and Reconfiguration Purpose *13*

6.3.1 Optimisation and Reconfiguration Generic Ontology 13

6.3.2 Factory Description Language 18

6.4 OE and OF deployment *22*

6.4.1 Dynamic and scalable orchestration 22

6.4.2 OE configuration 24

6.4.3 OF configuration 25

6.4.4 Cooperation with other SAFIRE modules 26

6.4.5 Implementation of the SAFIRE security framework 28

7. BUSINESS USE CASES **31**

7.1 Explicit description of the Objective Function Example - OAS Use case *31*

7.2 Explicit description of the Objective Function Example - ONA use case *32*

7.3 Predicted Objective Function from analysis of process data - Electrolux use case *33*

8. Requirements coverage (table) **34**

8.1 Reconfiguration *34*

8.2 Optimisations *35*

8.3 Performance *35*

8.4 Interfaces *36*
8.5 Communications *36*
8.6 Hardware/Platform/Devices *37*
9. References **38**

EXECUTIVE SUMMARY

This document specifies the full prototype of the Reconfiguration and Optimisation Engine component of the 'SAFIRE' project. The key concepts underlying component dependencies are described, and the corresponding interfaces are specified. The mechanisms supporting genericity are specified, together with concrete relationships to Business Cases. Descriptions are given of requirements coverage and extensions to existing research/technologies. The progress beyond the early prototype of the same component is explained.

1. INTRODUCTION

The goal of the Reconfiguration and Optimisation Engine is to provide scalable, high-quality reconfiguration of industrial systems/manufacturing processes.

Such optimisation has been extensively studied in Operations Research and Computer Science for over half a century [Papadimitriou and Steiglitz, 1982], and the essential solution methods are well understood academically and have been widely used in industry.

Optimisation is concerned with finding good solutions to problems that can be formulated in a specific manner. Such formulations have two essential ingredients:

1. A specification that implicitly describes all potential *solutions* to the problem. This is known as the *state space* of the problem.

2. A specification of some numerical *quality* measure for a proposed solution. This is known as the *objective function* or *fitness function*: a value to be maximised or minimised, depending on the problem at hand. (In this documents, the terms objective and fitness function are used interchangeably.)

The traditional example of an optimisation problem is the well-known Travelling Salesperson Problem (TSP): given a list of cities [Papadimitriou and Steiglitz, 1982], each of which should be visited exactly once, optimise the order in which these cities are visited, so as to minimise the total distance travelled. The state space of the problem consists of all possible re-orderings of the list of cities. The obvious quality measure of a proposed reordering is just the associated distance travelled. The TSP is just one example of a very wide range of problems that can be formulated in this way: in particular, manufacturing process problems such as Job Shop/Flow Shop/Open Shop scheduling have been represented in this way and studied since the inception of optimisation methods.

Such problems are not in general easy to solve: for example, as the number of cities is increased, so the number of possible solutions to the TSP increases in a 'bigger than exponential' manner. This 'exponential complexity' is shared by problems in manufacturing process optimisation: it soon becomes impossible for even the fastest supercomputer (or even any future computer) to consider all possible solutions and pick the best one.

Optimisation has therefore been strongly concerned with the development of *heuristics* [Burke et al, 2003]: methods which produce solutions which are *"good enough, fast enough, cheap enough"* even though they consider only a tiny fraction of all possible solutions.

Over the last 40 years, many different heuristics approaches have been proposed in the scientific literature and subsequently compared in both laboratory conditions and real-world applications. An example of a heuristic approach that have proved ubiquitously successful is *Genetic Algorithms* [Holland, 1992]: based on notions of "survival of the fittest" from Darwinian evolution, potential solutions are *selected* (in proportion to their quality) from a *population* of solutions – selected solutions are then subject to *crossover*

(analogous to genetic crossover of DNA, in which parts of a pair of solutions are combined to create a new solution) and *mutation* (analogous to the random mutations which occur in nature). The mutated solutions (which heuristically tend to be of *better quality*) are then incorporated into the population for subsequent evaluation. This process (of creating successive 'generations' of the population) repeats until a solution of sufficient quality is found (or else some other budgetary condition is reached). Since its invention in 1976, the method has been extensively studied and applied in many thousands of applications. One of the particular advantages of Genetic Algorithms is that they have high *genericity* across different types of problem: they tend to perform reasonably well even if the model of the process to be optimised is not highly detailed.

1.1 PROGRESS BEYOND D3.2 EARLY SPECIFICATION OF DYNAMIC AND PREDICTABLE RECONFIGURATION AND OPTIMISATION ENGINE

This specification is consistent with deliverable D3.2 Early Specification of Dynamic and Predictable Reconfiguration and Optimisation Engine [SAFIRE D3.2, 2018] and includes the majority of its content. The main additions to D3.5 are:

- specification of the objective function configuration by means of Optimisation Engine Configurator,
- deployment configuration description, in particular including the architecture of co-operating containers executing distributed evolutionary algorithms (supporting so-called island mode),
- relation to the SAFIRE security framework,
- optional execution of predictive analytics, as developed in WP2, to evaluate the value of solutions alternatively to the automatically generated OF,
- update of the requirements coverage table in Section 8.

The 'Optimisation Engine Configurator' will generate both Configuration and Objective Function corresponding to the manufacturing process model selected/specified by the end user (e.g. Interval or Max-plus algebra: see deliverable D3.1 [SAFIRE D3.1, 2018] for a detailed description of Max-plus and Interval Algebras). Since both the Objective Function and the Metrics API are planned to be generated from the specification in the proposed Factory Description Language (FDL), the appropriate descriptions for all SAFIRE BCs are provided.

2. CAPABILITIES TO BE DEVELOPED

The key deliverable for York is the provision of the *Optimisation and Reconfiguration Engine* (OE) component of SAFIRE, responsible for real-time process reconfiguration on demand. The overall specification of capability to be delivered by York is most readily explained via a division into three component parts: the *Metrics API*, *Optimisation Engine* and *Objective Function* (OF). These are specified in more detail below, at first

informally, then in terms of their formal parameters and detailed capabilities. Requirements coverage of these components can be found towards the end of the document.

2.1 METRICS API

The Metrics API provides a complete *schema description* for the variables associated with process configuration. Taken together, these variables define the *state space*, as defined in the introduction, above. As specified in deliverable D1.2 [SAFIRE D1.2, 2017], the elements of the configuration data schema are termed *metrics*, i.e. either *measurable physical values* corresponding to industrial process sensors or else *key objective (quality) measures* derived from these.

2.2 OPTIMISATION/RECONFIGURATION ENGINE

The key capability of the Optimisation and Reconfiguration Engine (Optimisation Engine, hereafter) is the ability respond to dynamic reconfiguration requests. Functionally, the Optimisation Engine takes as input a configuration (an instantaneous description of the manufacturing process) as specified by the Metrics API, and outputs a reconfiguration containing the proposed control values to be applied to the process. In accordance with the description of optimisation given in the introduction: the proposed reconfiguration is one that is of high quality, as determined by the *Objective Function*.

2.3 OBJECTIVE FUNCTION

As described above, the *Objective Function* is the means by which the Optimisation Engine is informed of the quality of a proposed reconfiguration. The term *Objective Function* is well-known in Operations Research [Gendreau and Potvin, 2010], having been widely used for well over half a century.

3. CAPABILITY STATE OF THE ART

3.1 METRICS API

The Metrics API has been implemented by York as a means of defining and representing processes, serving as a basis for interoperability and communication between the Optimisation Engine and other parts of the SAFIRE system that need access to it. The API is implemented in the Java programming language. Support is provided for automatic serialisation to/from the popular human-readable JSON protocol. Support has also been implemented for sending/receiving serialised JSON over http.

In contrast to the genericity provided by the Metrics API (see Section 4.1, below), there is little in the state of the art for representing optimisation problems in a domain-independent manner. Indeed, the only initiative of which we are aware [Kronberger et al, 2013] is directly tied to a specific choice of optimisation algorithm (viz., Grammatical Evolution), which would be overly constraining for the SAFIRE implementation. In the absence of a generic method of representing (the state space of) problems, it is typical that the optimiser is configured a) using expert labour and b) by writing program code to represent problem specifics. Our proposed approach avoids the necessity for this, being instead schema-driven. In particular, being able to interoperate instances of

this schema directly via JSON over HTTP facilitates the federated hosting of solvers in the cloud that is a key part of the SAFIRE remit.

3.2 OPTIMISATION ENGINE

SAFIRE represents progress towards the 'Smart Factory' notion of "Industry 4.0". In contrast to third generation manufacturing systems, the notion is that manufacturing processes will operate in `just-in-time' mode, being rapidly responsive to the dynamic arrival of manufacturing orders. It is anticipated [Chen et al, 2017] that such factories will have a set of highly configurable machines with automated material handling systems and a cloud-based management system.

More specifically, a key motivation for cloud-based optimisation is that part of the SAFIRE remit is for the Optimisation Engine to operate 'with predictable upper bounds on execution time'. Combining such 'Worst Case Execution Time' guarantees with metaheuristic search draws on expertise particular to the University of York's Real-time Systems research group. By virtue of a cloud-based architecture (Section 6.4), optimisation can be scaled (by federating the workload over a larger number of computers) to solve larger problems or produce quality results more quickly.

Unlike SAFIRE, the current state-of -the-art in cloud-based optimisation is not typically concerned with execution-time guarantees. In addition (in contrast to our recent application of the Optimisation Engine [Dziurzanski et al, 2018]), we are not aware of any associated case studies for OEE.

A recent position paper [Salza et al, 2016] presents a conceptual workflow for the deployment and execution of distributed Genetic Algorithms (GAs), one of the most widely used search-based meta-heuristics for optimisation. The software container technology (Docker) and a lightweight Linux distribution created to execute containers (CoreOS) has been used for large and scalable deployments on different infrastructure, focusing on security, consistency and reliability. This idea has been further extended in Salza et al [Salza et al, 2017], where evolutionary machine learning classifiers have been deployed to the cloud. In the proposed solution, a similar architecture is used for performing evolutionary optimisation. In particular, Docker containers are used to execute instances of jMetal [Durillo and Nebro, 2011], a popular Java framework offering a variety of algorithms for single- and multi-objective metaheuristic optimisation.

In Ma et al [Ma et al, 2017], a master-slave topology implementing a distributed evolution algorithm was employed. The master assigned the individuals from each generation to the slave nodes based on their load information and then collected the corresponding fitness values. The comparison with allocation of the same number of individuals to each node has been conducted for 32, 48 and 64 nodes. The obtained improvement of the computation time has ranged from 6% to 39% depending on the cluster size. While shortest time was achieved for the largest case, the strategy proposed in that paper has not considered heterogeneous architecture or various communication costs. The proposed solution also takes communication overhead into consideration. Since the proposed approach uses load balancers as detailed later in this document, the number of

nodes is decided dynamically between pre-defined boundaries. Leclerc et al [Leclerc et al, 2016] propose a cloud-based framework facilitating large scale evolutionary experiments. Their framework provides a master-slave architecture, with nodes communicating via JSON over HTTP. The applied scheduling policy aims to uniformly spread the load across peers. The slave node with minimal load is chosen for each incoming fitness function evaluation task. There is no possibility of sharing processing units between tasks. Consequently, if there is no slave with an idle processing unit, the task is placed in a FIFO queue. To guarantee the appropriate amount of computing resources, the framework is intended to be executed on virtual machines (VMs) whose number is steered by the cloud provider, using facilities such as Amazon Auto Scaling Group. When the smoothed expected time to empty the queue is larger or smaller than certain thresholds, a VM is added or removed, respectively. The motivation for the container-based approach is to provide both scalability and cost-effectiveness: payment is made only for actual replicated containers. In future, the serverless approach can be applied instead if the limits regarding the maximal execution time or memory footprints will be lowered by the major public cloud vendors. The current limits have been provided in SAFIRE deliverable D3.1 [SAFIRE D3.1, 2018] together with the requirements regarding the cloud platform.

3.3 OBJECTIVE FUNCTION

In order to determine the quality of a proposed reconfiguration, the Optimisation Engine requires some form of *model* of the system to be optimised.

The prevalent approach in optimisation is for the system to be explicitly modelled (typically in a mathematical form) by potentially lengthy and costly collaboration between optimisation experts (often academics) and experts for a specific Business Case.

In specific problem areas, some efforts have been made to reduce the necessity for domain experts by providing dedicated configuration tools. Such tools provide a predefined, problem-specific Objective Function and associated model, aspects of which can be modified by someone familiar only with the practicalities of the process. Such modification may be done via a human-readable textual description or GUI. One example of a commercial package of this nature is RosterViewer (http://www.staffrostersolutions.com), applicable to personnel scheduling problems.

Such tools are built with expert knowledge and are restricted to particular problem areas. In contrast, the role of the Objective Function within SAFIRE is to support a diverse range of manufacturing processes in a manner that is as problem agnostic as possible. Since this has implications for ‘End User Configuration’, the supported options for this are discussed in more detail that Section.

4. CAPABILITY NEW TECHNOLOGIES / INNOVATIONS

4.1 METRICS API

Working in combination with the interface specifications of the Optimisation Engine and Metrics API, the Objective Function is key to SAFIRE *genericity*, maximising the ability to describe problems in a manner that is independent of Business Case specifics.

In accordance with the descriptions of D1.2 [SAFIRE D1.2, 2017], the Metrics API supports a variety of different types of metric value types (integer, real-valued, nominal), and allows checkable constraints to be specified on their associated ranges. It also allows the specification of process-specific quality indicators (the key objective metrics) and whether they are to be maximised or minimised.

The Metrics API and the means by which such open ended support for Business Cases is provided is described in more detail in the subsequent sections.

4.2 OPTIMISATION ENGINE

The capability of 'predictable, dynamic response' is provided via a scalable, cloud-based architecture for the Optimisation Engine.

The proposed optimisation process is implemented using a master-slave paradigm: the master executing a single instance of OE awaits manufacturing orders. Upon arrival of some observed change in plant configuration (e.g. arrival of a manufacturing order), configuration data is sent to a certain number of slave nodes, where, at each stage, (an instance of) the Objective Function is executed.

The current implementation of the Optimisation Engine uses a Genetic Algorithm (as described in the Introduction Section), implemented within the `jMetal` framework (https://github.com/jMetal) as well as the framework written in York in the course of the SAFIRE project and placed inside a Docker container. This container acts as a REST-compliant Web service, awaiting input in the form of a population of proposed plant re-configurations to be optimised. On timeout, the container returns the best of the proposed re-configurations. Communication between the master and slave nodes is performed via JSON over HTTP.

A recent conference publication [Dziurzanski et al, 2018] describes the above in full implementation detail, and gives experimental results showing successful application of the scalable, serverless implementation of the Optimisation Engine to a ubiquitous class of manufacturing problems, producing near-optimal solutions in terms of OEE. However, as discussed in SAFIRE Deliverable D3.1 [SAFIRE D3.1, 2018], the current limits related to the serverless container execution in the most popular public clouds are too strict to execute OF and thus the only possibility to remain serverless for OF is to use Apache OpenWhisk (https://openwhisk.apache.org), but this solution is mainly applicable in private clouds. At the current moment of public cloud computing evolution, the container services such as Amazon Elastic Container Service (ECS) or Amazon Elastic Container Service for Kubernetes (EKS) are more appropriate to execute computation-

ally-intensive containers, as discussed in SAFIRE Deliverable D3.1 [SAFIRE D3.1, 2018].

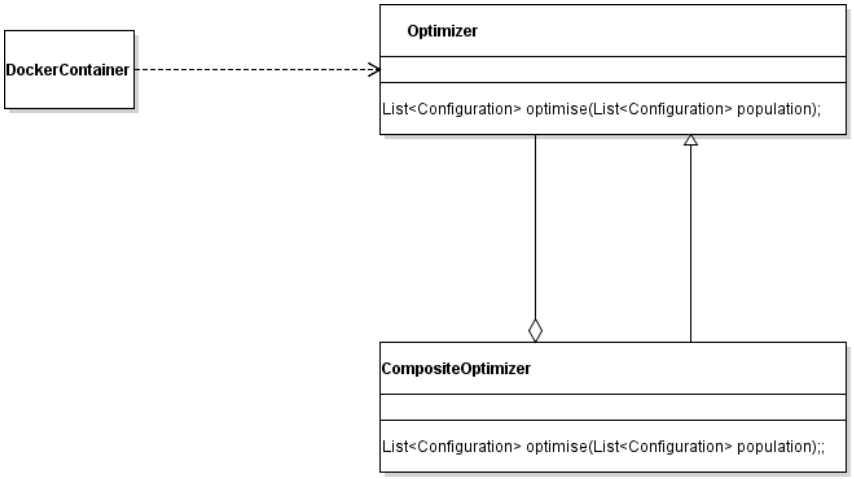

Figure 1. Running multiple Optimisation Engines via Docker.

The architecture of the Optimisation Engine (Figure 1) allows to optimisers to be hierarchically aggregated and/or federated across a distributed system to yield solvers of increasing power. A recent research community initiative [Swan et al, 2015] advocates this approach as being vital for large scale improvements in both research practice and solver performance. In particular, the industrial application of these methods is novel. There also novelty in using such a decomposition to create solver ensembles which are both cloud-based and highly parallel [Dziurzanski et al, 2018].

4.3 OBJECTIVE FUNCTION

In contrast to the 'always bespoke' approach described in the 'Capability State of the Art' approach, above, our approach to Objective Function definition is to provide a range of options:

- The option to model a business case using a pre-existing model.
- The traditional bespoke option, as above.

As detailed in deliverable D3.1, two models (the 'Max-plus' and 'Interval' algebras) of OF have been provided, which are applicable to a very wide range of production scenarios. The Objective Function represents a key variation point for End-User customisation, and the attendant 'End-User Responsibilities' for Business Case are detailed in the corresponding Section.

5. INTERFACE SPECIFICATIONS

As described in [Dziurzanski et al, 2018], a preliminary version of the Optimisation Engine is in operation, with interfaces as described below.

5.1 METRICS API

The reconfiguration process performed by the Optimisation Engine can be achieved in a generic fashion, driven by a per-Business Case specific specification of:

- A *Configuration Specification* specifying the *data schema* for the system to be optimised.
- A *quality measure* (known as an *Objective Function*) to determine the value of the proposed reconfiguration.

These specifications also suffice to determine the interfaces by which the wider SAFIRE system (most specifically, Situation Determination and Predictive Analytics) communicate with the Optimisation Engine.

We now describe each in more detail.

5.1.1 Configuration Specification

The configuration schema is a complete description of the metrics by which the system is observed, controlled and evaluated, in terms of their types (e.g. integer, real-valued, string) and the associated valid ranges (e.g. 1-11,0.0-1.0, {“cool”,”warm”,”hot”} etc).

In SAFIRE Deliverable D1.2 [SAFIRE D1.2, 2017] these metrics were specified as being divided into 3 categories: ‘Observable Metrics’; ‘Control Metrics’ and ‘Key Objective Metrics’. By way of example, the corresponding metrics for BC2-UC1 (OAS) are given in Table 1-Table 3.

Table 1. Key Objective Metrics of Business Case BC2-UC1

Key Objective Metrics			
Metric	*Unit*	*Objective*	*Description*
performance	%	min	Depends on the speed of the weighting process.
quality of dosing	%	max	Target-actual comparison. The dosing should be precise. 100 % is optimal.
OEE	%	max	Combination of quality and performance

Table 2. Controlled Metrics of Business Case BC2-UC1

Controlled Metrics			
Metric	*Unit*	*Range*	*Description*
Auger conveyor speed for coarse dosing	RPM	0-max	Set RPM of the auger conveyor Coarse Dosing mode

Auger conveyor speed for fine dosing	RPM	0-max	Set RPM of the auger conveyor Fine Dosing mode
Time frame for Auger conveyor coarse dosing	Sec	0-max	Set time frame for coarse dosing
Time frame for Auger conveyor fine dosing	Sec	0-max	Set time frame for fine dosing

Table 3. Observable Metrics of Business Case BC2-UC1

Observable Metrics			
Metric	*Unit*	*Sample rate*	*Description*
Scale weight	Int gram	sec	The current weight measurement of a scale
Auger conveyor speed	Int RPM	sec	The speed of the auger conveyor
Auger conveyor status	Int	sec	Fine Dosing mode or Coarse Dosing mode or off
production step	int	1-x	Current production step of a production order. To identify if the weighting process is enabled
Kind of raw material	String	Sec	Identify the kind of raw material currently in the weighting process.
Amount of raw material needed	kg	One time for each weighting step	Amount of raw material which has to be weighed.

5.2 CONFIGURATION SCHEMA INTERFACE

In order to define the required inputs to the Optimisation Engine, York has developed a 'Metrics API', allowing the definition and validation of configuration schema. This API is written in the Java programming language and is made available to all partners via the SAFIRE Git repository. To ensure that remote invocation of the Optimisation Engine can be performed in a consistent manner by all project partners, the classes of the Metrics API provide support for serialisation via JSON.

There are several dozen classes in the Metrics API. The key ones are given in the UML diagram presented in Figure 2.

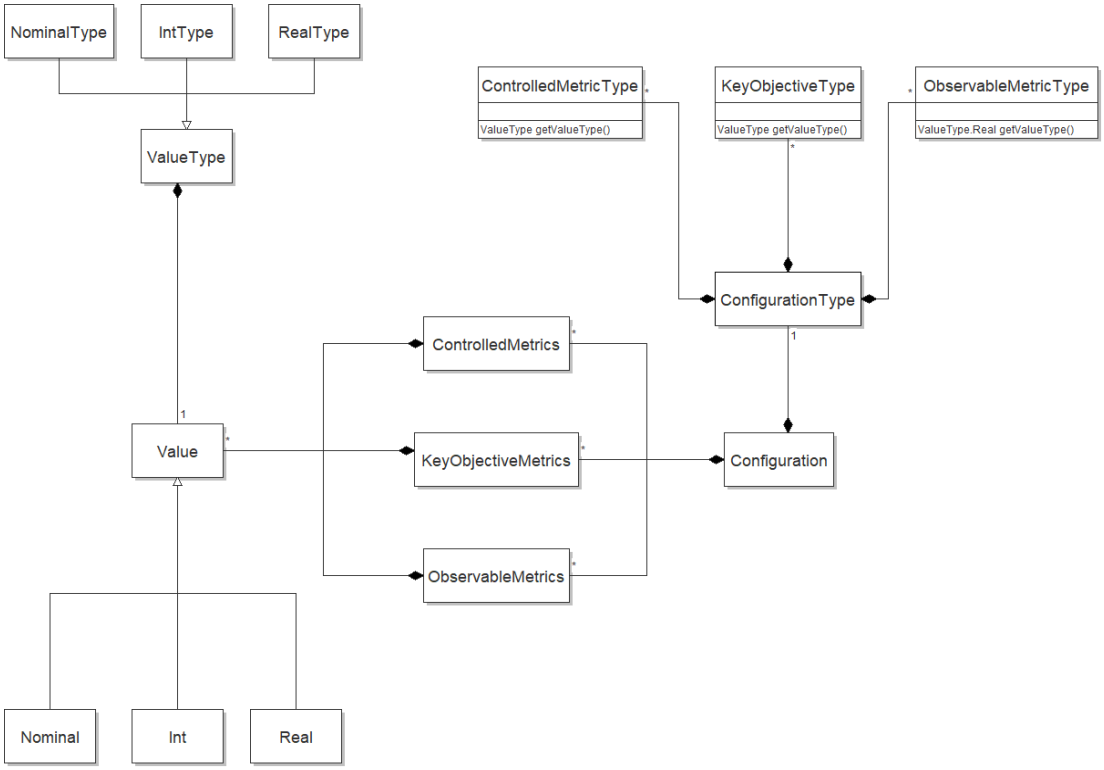

Figure 2. Class diagram of Metrics API

For summary purposes, the key aspects of the Metrics API can be described in terms of the Configuration class, which is used as both input to and output from the Optimisation Engine:

```
package uk.ac.york.safire.metrics;

public final class Configuration {

    public Configuration(ConfigurationType configurationType,

     Map<MetricName, Value > controlledMetrics,

        Map< MetricName, Value > observableMetrics,

     Map< MetricName, Value > keyObjectives );


    /* Other implementation details omitted... */

};
```

A `Configuration` is thus the Optimisation Engine's representation of a manufacturing process, containing values for the controlled, observable and key objective metrics. Each of these are indexed by name (`Map<MetricName,Value>`). `Value` is a vari-

ant record type, capable of representing either Integer, Real or String values (examples of each can be seen in the Units columns of the **BC2-UC1** metrics, above).

The `ConfigurationType` provides the associated configuration schema, i.e. the required types and ranges of the associated `Values`, thereby allowing runtime validation of a `Configuration`.

5.3 OPTIMISATION ENGINE INTERFACE

The Optimisation Engine can be invoked by other parts of the SAFIRE system via the following simple API call:

```
interface OptimisationEngine {

   Configuration
   optimise(Configuration args);
}
```

Input: args -- a *configuration with observable and Control metrics* (D1.2 [SAFIRE D1.2, 2017]) corresponding to the *real-world* physical state of the process to be optimised.

Output: a high-quality proposed *re-configuration:* specifically, a configuration containing Control metrics to be applied to the real-world process. In order to relieve the invoker of the Optimisation Engine (e.g. other SAFIRE modules such as Situation Determination) from concerns implementation details, we provide a variety of alternative implementations of the Optimisation Engine interface: for example allowing transparent use of either locally hosted Optimisation Engines or ones hosted remotely via a Docker container. In the latter case, the arguments and results to the call to `optimise` are automatically communicated via JSON over HTTP. The latter version will be used in the full prototype of the Reconfiguration and Optimisation Engine.

5.4 OBJECTIVE FUNCTION INTERFACE

The long-established practice of optimisation proceeds by computationally generating (potentially many thousands) of possible proposed reconfigurations, choosing high-quality ones on the basis of some Business Case-specific quality measure. This quality measure is traditionally known as an *Objective Function*. For SAFIRE purposes, we can consider an Objective Function to be a function which assigns a numerical value to (the control metrics of) a proposed configuration:

```
double valueOfReconfiguration(Configuration proposedReconfiguration);
```

BC-specific objective functions are made available to the Optimisation Engine via configuration file at setup time (e.g. as output from the 'OE Configurator Program' – see Section 6 below for more details of the latter): as mentioned above, this can transparently be specified as either running locally (i.e. running within the same JVM as the Optimisation Engine) or remotely (i.e. using a designated service endpoint via JSON-over-HTTP).

6. END-USER CONFIGURATION

We detail End-User configuration options from the perspective of the Optimisation Engine component:

As previously discussed, it is a necessity in computation optimisation for the optimiser to be provided with an Objective Function: a means of evaluating configuration quality. As described in SAFIRE Deliverable D3.1 [SAFIRE D3.1, 2018], we have provided two associated process models (Max-Plus and Interval algebras) that can cover a wide class of manufacturing problems, including specifically ONA and OAS Business Cases. End-user configuration of SAFIRE for some other Business Case falls into one of two main categories, depending on whether these pre-existing objective functions are suitable:

6.1 SUITABLE PRE-EXISTING OBJECTIVE FUNCTION

Assuming that this new Business Case can be described in terms of the above models, there is still a requirement for the end-user to explicitly define their process in terms of one of the above models. In the Full Prototype of Reconfiguration and Optimisation Engine, a standalone 'OE Configurator Program' will be provided to facilitate this.

6.2 NO SUITABLE PRE-EXISTING OBJECTIVE FUNCTION

If the new Business Case *cannot* readily be described by the above models then there are two options, with attendant pros and cons. These options are illustrated with suitable uses cases defined by SAFIRE project industrial partners in Section 7.

6.2.1 Explicit description of the Objective Function.

This option is the one traditionally followed in optimisation and requires the objective function to be described in explicitly mathematical terms. For example, as described in the Introduction, the Objective Function for the well-known Travelling Salesman problem is the sum of the distances between the cities visited.

Advantages:

- Can take advantage of a priori expert insight into the business process being controlled.
- Allows the modelling of almost any optimisation problem.

Disadvantages:

- Requires each Business Partner to explicitly formulate their requirements in mathematical terms.
- To the extent that genericity beyond the 3 current BCs is a goal of SAFIRE, the labour cost (for both Business Partner and SAFIRE Configuration Expert) of this process should be borne in mind.

6.2.2 Predicted Objective Function from analysis of process data.

The second option would be to for the Predictive Analytics module to provide the Optimisation Function. This could be done either:

1. At configuration time from historical data.
2. Adaptively at runtime by observing the correlation between observable/control metrics and key objectives.

Advantages:

- Does not require Business Partner engagement to specify the Objective Function.
- If runtime adaptivity is supported, then this has the potential to respond intelligently to changing production environments.

Disadvantages:

- Success depends on the predictability of the production environment: many observations may be required before prediction quality converges. This implies the need for either representative historical process data (for configuration-time learning of the objective function) or else (for production-time learning) an inexpensive production process/ human veto of the optimizer while the predictions are initially learned.

6.3 PLANT DESCRIPTION FOR THE OPTIMISATION AND RECONFIGURATION PURPOSE

In SAFIRE, the plants to be optimised and reconfigured, together with the production processes (recipes) to be applied in the manufacturing process, are described with the ontology briefly described in Subsection 6.3.1, whereas Subsection 6.3.2 introduces the Factory Description Language (FDL).

6.3.1 Optimisation and Reconfiguration Generic Ontology

In SAFIRE, the plants to be optimised and reconfigured, together with the production processes (recipes) to be applied, are described with the ontology presented in Figure 3. This ontology is a subontology of the SAFIRE ontology and complements the ontology for modelling correlation between information sources, products & situations, as presented in deliverable D4.1 [SAFIRE, D4.1, 2018], as shown later in this section. In this subontology, *Order* is a primary actor describing request for producing a certain amount of a certain commodity, represented by entity *Product*. The pair of *Product* and the amount that can be produced by a certain *ProductionProcess* is represented by entity *ProductAmount*. The rationale for introducing this entity is the fact that, in general, a production process can lead to manufacturing more than one commodity at the same time. *ProductionProcess* describes operations that shall be executed for producing a certain commodity. *Subprocess* models a certain stage of a production process. *SubprocessRelation* describes a relation between two *Subprocesses* in a production process.

For example, if *Subprocess_B* has to be executed directly after *Subprocess_A*, relation *Meet* (M) shall be selected. *SubprocessRelationType* describes the relation (based on Allen's algebra) between two *Subprocesses* (Source and Destination) in the same production process. This relation belongs to set {EQ , F , LT , M , O , S} (see deliverable D3.1 [SAFIRE D3.1, 2018] for details). *SubprocessRelationUsage* models a relation between a certain *Subprocess*, selected *ProcessingDevice*(*s*) for executing this *Subprocess*, its *ProcessingDeviceMode* and the corresponding *ProcessingTime*, *EnergyConsumption* and *MonetaryCost*. *ProcessingDevice* is a primary actor describing a certain processing resource (e.g. machine) in a plant. *ProcessingDeviceMode* specifies a mode that a certain resource can operate in. The modes can model an explicit resource mode (as Eco mode), or e.g. different commodities/tools using for manufacturing (like wire in the ONA BC). *AbstractProcessingDevice* is a set of plant's *ProcessingDevice* that subprocesses can be allocated to. *SequenceDependentSetup* models extra costs when two certain *Subprocesses*, specified with attributes *hasSequenceDependentSetupSource* and *hasSequenceDependentSetupDestination*, are performed subsequently using the same processing device. This extra cost can refer to time, energy or monetary cost, so three data properties are provided: *hasSequenceDependentSetupEnergyConsumption*, *hasSequenceDependentSetupMonetaryCost* and *hasSequenceDependentSetupProcessingTime*. The remaining object and data properties are self-explanatory. The object and data properties of this ontology are enumerated in Table 4 and Table 5, respectively. The complete SAFIRE ontology comprised of two subontologies: the described ontology for plant description and the generic ontology from D4.1 is depicted in Figure 4.

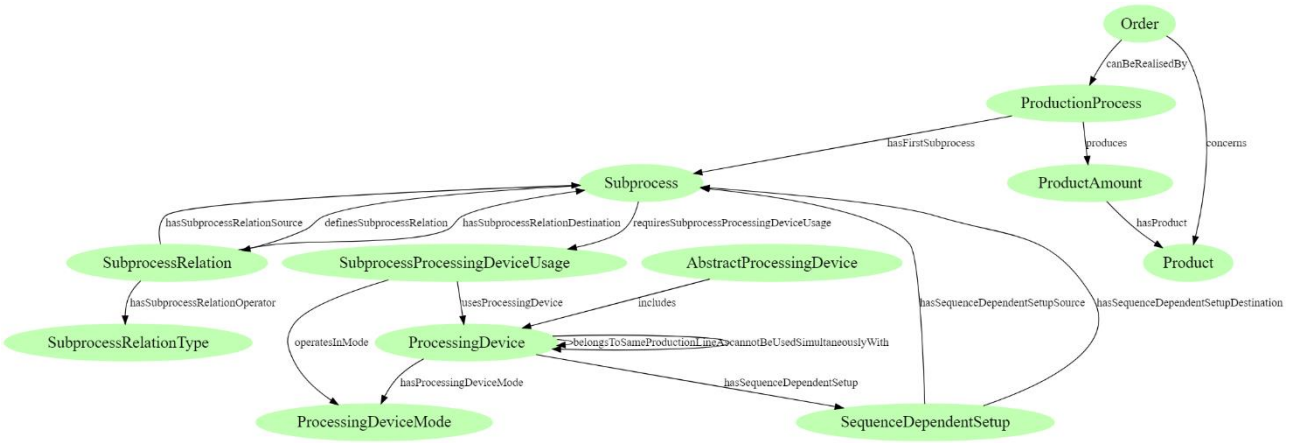

Figure 3. Ontology for plant description

Table 4. Object properties in the ontology for plant description

Name	Domain	Range	Description
hasProduct	ProductAmount	Product	associates manufactured amount of commodity with the corresponding commodity
produces	ProductionProcess	ProductAmount	associates a production process with commodity amounts manufactured with this process
canBeRealisedBy	Order	ProductionProcess	associates an order with a production process that can manufacture commodities satisfying this order

concerns	Order	Product	associates an order with the commodity manufactured by this order
belongsToSameProductionLineAs	ProcessingDevice	ProcessingDevice	models the situation when there exists a few production lines in a plant and a certain commodity can be produced in any of them; however, when a certain resource is selected to be used at the first stage, it limits the choice of the resources used at further steps
cannotBeUsedSimultaneouslyWith	ProcessingDevice	ProcessingDevice	models mutual exclusiveness of the resources; for example, two processing devices (e.g. pipes) cannot be used at the same time as they are connected to the same resource (e.g. scale)
definesSubprocessRelation	Subprocess	SubprocessRelation	associates a subprocess with a certain relation whose this subprocess is an object
hasFirstSubprocess	ProductionProcess	Subprocess	associates a production process with the first subprocess of this process
hasSubprocessRelationDestination	SubprocessRelation	Subprocess	associates a relation between a subprocess with its second parameter
hasSubprocessRelationOperator	SubprocessRelation	SubprocessRelationType	associates a relation between a subprocess and its relation type (e.g. meet or overlap)
hasSubprocessRelationSource	SubprocessRelation	Subprocess	associates a relation between a subprocess and its first parameter
hasProcessingDeviceMode	ProcessingDevice	ProcessingDeviceMode	associates a resource with a mode this resource can operate in
hasSequenceDependentSetup	ProcessingDevice	SequenceDependentSetup	associates a resource with its setup depending on the sequence of subprocesses executed on this particular resource
hasSequenceDependentSetupDestination	SequenceDequenceDependentSetup	Subprocess	associates a sequence dependent setup with its second parameter, i.e. the subprocess that is scheduled to be executed subsequently on a given resource
hasSequenceDependentSetupSource	SequenceDequenceDependentSetup	Subprocess	associates a sequence dependent setup with its first parameter, i.e. the subprocess that is scheduled to be executed (directly) earlier on a given resource
operatesInMode	SubprocessProcessingDeviceUsage	ProcessingDeviceMode	associates a certain resource usage during a certain subprocess with a mode that the processing resource operates in
requiresSubprocessProcessingDeviceUsage	Subprocess	SubprocessProcessingDeviceUsage	associates a subprocesses with a certain resource usage during this subprocess execution
usesProcessingDevice	SubprocessProcessingDeviceUsage	ProcessingDevice	associates a resource usage during a certain subprocess with this resource
includes	AbstractProcessingDevice	ProcessingDevice	associates an abstract processing device (e.g. a production line, treated as a set of concrete processing devices) with the concrete processing devices

Table 5. Data properties in the ontology for plant description

Name	Domain	Range	Description
hasAmount	ProductAmount	xsd:int	associates an amount of commodity with its integer

			value
hasEnergyConsumption	SubprocessProcessingDeviceUsage	xsd:int	associates a processing device usage during a production process while executing a certain subprocess with its integer value of energy consumption
hasMonetaryCost	SubprocessProcessingDeviceUsage	xsd:int	associates a processing device usage during a production process while executing a certain subprocess with its integer value of monetary cost
hasProcessingTime	SubprocessProcessingDeviceUsage	xsd:int	associates a processing device usage during a production process while executing a certain subprocess with its integer value of processing time
hasProcutionProcessName	ProductionProcess	xsd:string	associates a production process with its name
hasProcutionProcessPriority	ProductionProcess	xsd:int	associates a production process with its priority
hasProcessingDeviceModeName	ProcessingDeviceMode	xsd:string	associate a processing device mode with its name
hasProcessingDeviceName	ProcessingDevice	xsd:string	associates a processing device with its name
hasSequenceDependentSetupEnergyConsumption	SequenceDependentSetup	xsd:int	associates a sequence dependent setup with is integer value of energy consumption
hasSequenceDependentSetupMonetaryCost	SequenceDependentSetup	xsd:int	associates a sequence dependent setup with is integer value of monetary cost
hasSequenceDependentSetupProcessingTime	SequenceDependentSetup	xsd:int	associates a sequence dependent setup with is integer value of processing time

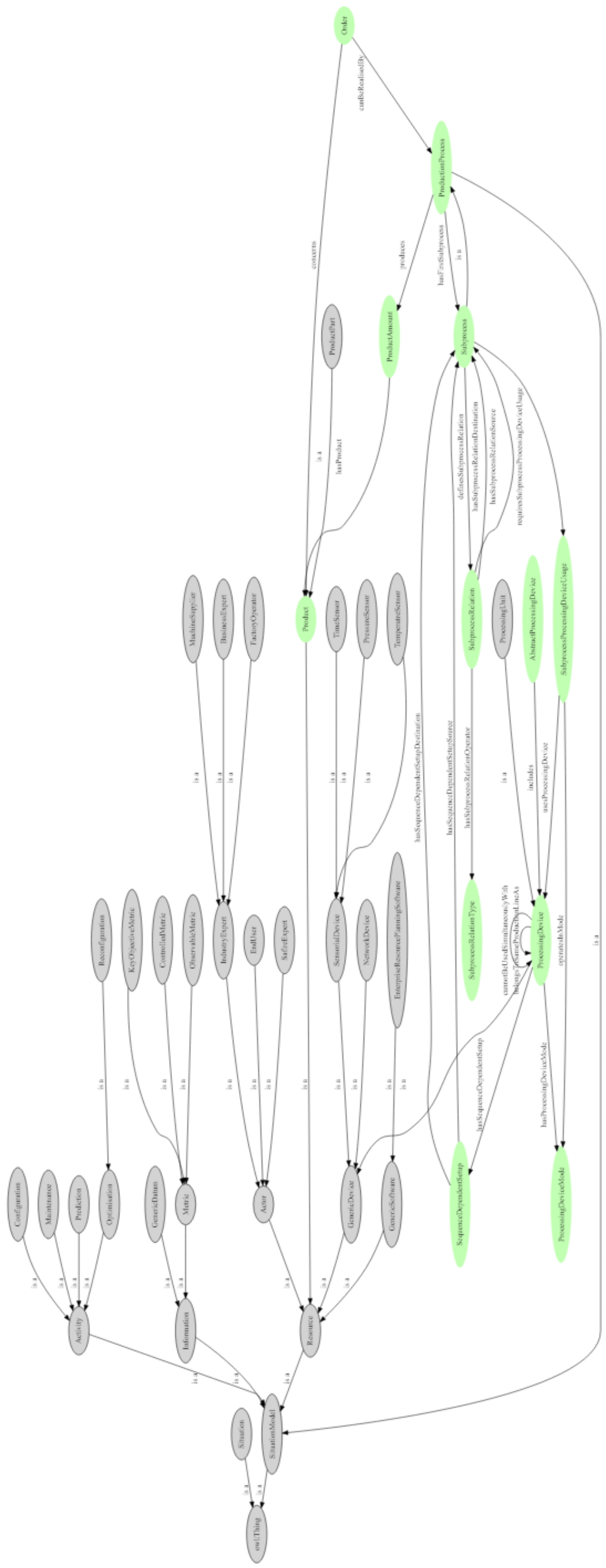

Figure 4. SAFIRE ontology comprised of subontologies: the ontology for plant description (green nodes and generic ontology for modelling correlation between information sources, products & sit-uations

6.3.2 Factory Description Language

Based on the ontology presented earlier in this section, an XML-based language for factory description, named Factory Description Language (FDL), is proposed. This language is expressive enough to describe all SAFIRE business cases, as defined in SAFIRE Deliverable D1.1 [SAFIRE D1.1, 2017] and other similar scenarios. The key idea behind FDL creation is to make the language human-readable, so that a person outside the project can relatively easily define the problem to be optimised, including a plant architecture, products, production processes etc. The elements (tags) of FDL directly correspond with the ideas from the ontology described earlier in this section. Below, the most important elements of FDL are discussed and a number of examples has been provided.

Element *processingDevices* includes a set of elements named *processingDevice*, representing all processing resources (e.g. machines) in a plant. A *processingDevice* element requires the *name* attribute. As a resource can operate in a number of various operation modes, a *processingDevice* element includes the nested *modes* element, which in turn includes a set of *mode* elements with the mandatory *name* argument. Each resource has to include at least one mode.

Example:

```
<processingDevices>
    <processingDevice name="Conveyor1">
            <modes>
                    <mode name="Standard"/>
            </modes>
    </processingDevice>
    <processingDevice name="Mixer1">
            <modes>
                    <mode name="Standard"/>
                    <mode name="Economic"/>
            </modes>
    </processingDevice>
</processingDevices>
```

The *productionLines* element describes all production lines in a factory, introduced as nested *productionLine* elements. The *name* attribute in the *productionLine* element is mandatory. The *productionLine* element includes a nested *productionLineProcessingDevices* element, which in turn includes nested *productionLineProcessingDevice* elements. Each *productionLineProcessingDevice* start-tag includes two attributes, *order* and *name*. The former attribute values are consecutive numbers that identify the resource order in a production line, whereas the latter attribute values have to be equal to the resource names introduced in element *processingDevice*. Each production line is linear and thus each possible split of processing results in creating a new production line, from the production line source to its sink. In example below, the two production lines starts with the same resource (Scale), but as two routes are possible starting from

Converyor1 or Conveyor2, two *productionLine* elements starting from the Scale resource are generated.

Example:

```
<productionLines>
    <productionLine name="ProductionLine1">
            <productionLineProcessingDevices>
                    <productionLineProcessingDevice order="1" name="Scale"/>
                    <productionLineProcessingDevice order="2" name="Conveyor1"/>
                    <productionLineProcessingDevice order="3" name="Mixer1"/>
                    <productionLineProcessingDevice order="4" name="Tube1"/>
                    <productionLineProcessingDevice order="5" name="Pipeline1"/>
            </productionLineProcessingDevices>
    </productionLine>
    <productionLine name="ProductionLine2">
            <productionLineProcessingDevices>
                    <productionLineProcessingDevice order="1" name="Scale"/>
                    <productionLineProcessingDevice order="2" name="Conveyor2"/>
                    <productionLineProcessingDevice order="3" name="Mixer2"/>
                    <productionLineProcessingDevice order="4" name="Tube2"/>
                    <productionLineProcessingDevice order="5" name="Pipeline2"/>
            </productionLineProcessingDevices>
    </productionLine>
</productionLines>
```

Element *productionProcesses* includes a set of production processes that need to be scheduled in the considered plant. Each *productionProcess* element, nested in *productionProcesses*, includes the mandatory *name* attribute and one or more *alternative* sets of subprocesses leading to manufacturing a certain commodity.

Each *subprocess* element requires the name attribute and a set of nested *subprocessProcessingDevice* elements. Unique names of subprocesses are required to refer to them unambiguously from other elements, e.g. *sequenceDependentSetup* (explained later). If more than one *subprocessProcessingDevice* elements are provided, they are treated as alternative ones and producing the same commodity.

In the *subprocessProcessingDevices* element, all processing devices that have to be allocated *simultaneously* to execute the given subprocess are listed with elements *subprocessProcessingDevice*. The mandatory argument of this tag is *processingDeviceName*, whose value shall be found in the *processingDevice* element described earlier. Then *subprocessProcessingDevicesMode* elements follow with the mandatory *modeName* attribute whose value shall be listed into the corresponding *processingDevice* element, as described earlier. The *subprocessProcessingDevicesMode* element includes at least one of the three elements: *processingTime*, *energyConsumption* and *monetaryCost*. These three elements specify the corresponding numeric costs of using the particular pro-

cessing device in the particular mode and as such can be later used to define a fitness function of a factory scheduling.

Example:

```
<productionProcesses>
    <productionProcess name="White50l">
            <subprocesses>
                    <subprocess name="White50lTask1">
                            <subprocessProcessingDevices>
                                    <subprocessProcessingDevice processingDevice-
Name="Silo">
                                            <subprocessProcessingDeviceMode mode-
Name="Standard">
                                                    <processingTime>1</processingTime>
                                                    <energyConsump-
tion>1</energyConsumption>
                                                    <monetaryCost>100</monetaryCost>
                                            </subprocessProcessingDeviceMode>
                                    </subprocessProcessingDevice>
                                    <subprocessProcessingDevice processingDeviceName
="Scale">
                                            <subprocessProcessingDeviceMode mode-
Name="Standard">
                                                    <processingTime>2</processingTime>
                                                    <energyConsump-
tion>1</energyConsumption>
                                                    <monetaryCost>100</monetaryCost>
                                            </subprocessProcessingDeviceMode>
                                    </subprocessProcessingDevice>
                            </subprocessProcessingDevices>
                    </subprocess>
            </subprocesses>
    </productionProcess>
</productionProcesses>
```

Another element that is mandatory in a *productionProcess* element as long as that element includes more than one *subprocess* element is *subprocessRelations*, using *subprocessRelation* to describe relations between *subprocesses* in the considered *productionProcess*. Three arguments are mandatory: *source* and *destination* requires a proper name of *subprocess* introduced in the considered *productionProcess*, whereas *allensOperator* requires any relation from the interval Allen's algebra that describes the temporal relation between the source and the destination. The following *allensOperator* values are possible: *LT* for source earlier than destination, *S* for source since destination, *F* for finish destination, *EQ* for source equal to destination, *O* for source overlapping destination, *M* for source meeting destination and *D* for source during destination.

Example:

```
<subprocessRelations>
    <subprocessRelation source="Task1" destination="Task2" allensOperator="M"/>
    <subprocessRelation source="Task2" destination="Task3" allensOperator="M"/>
    <subprocessRelation source="Task3" destination="Task4" allensOperator="M"/>
    <subprocessRelation source="Task4" destination="Task5" allensOperator="M"/>
</subprocessRelations>
```

Element *sequenceDependentSetup* determines extra costs when two certain subprocesses, specified with attributes *source* and *destination*, are performed subsequently using the same processing device, specified with attribute *processingDevice*. This extra cost can refer to time, energy or monetary cost, so three elements are provided: *extraProcessingTime*, *extraEnergyConsumption* and *extraMonetaryCost*.

Example:

```
<sequenceDependentSetups>
    <sequenceDependentSetup source="White50lTask3" destination="Yellow50lTask3"
processingDevice="Tube1">
                                                    <extraProcessing-
Time>2</extraprocessingTime>
                                                    <extraEnergyConsump-
tion>1</extraenergyConsumption>
                                                    <extraMonetary-
Cost>100</extramonetaryCost>
    </sequenceDependentSetup>
</sequenceDependentSetups>
```

To explain the application of FDL to SAIFRE BCs, its relation to the ONA BC follows. In ONA BC, the objective is to minimise the monetary cost per part. This cost can be obtained by summarising all values of *monetaryCost* of the subprocesses used to produce a given part. If applicable, the values of *extraMonetaryCost* of *sequenceDependentSetups* should be added as well. FDL can be applied to other SAFIRE BCs in a similar manner.

The resource allocation consists of selecting processing devices (and thus production line) for processing the part (product). The selected processing devices (machines) can operate in a number of modes, each related to, e.g., different wire type. Consequently, all possible modes for processing each considered part have to be explicitly specified using element *subprocessProcessingDeviceMode*.

6.4 OE AND OF DEPLOYMENT

6.4.1 Dynamic and scalable orchestration

Both OE and OF are planned to be available in the forms of Docker containers, to be deployed in a local or remote cloud, communicating each other over HTTP. Such software architecture not only provides the full flexibility in selecting the data centre for deployment, but also benefits from numerous cloud-computing features, such as load balancing or autoscaling. It means that the number of OF containers can be dynamically changed to answer the current users' requirements. In case of a higher demand caused by e.g. a growing number of active users, the number of containers is seamlessly scaled up to keep the appropriate computing power, and similarly scaled down in case of a lower demand. Such an approach not only guarantees the high system uptime, theoretically equal 100% but also increases the responsiveness of OE and lowers the monetary cost in case of the deployment in a public cloud. The auto-scaling feature is utterly transparent to the user. Even in an unlikely case of any container crash, the load balancer immediately switches the user to another container build from the same image in a way that no user data nor session details are lost. This feature is possible due to the fact that the OE and OF containers are designed to be stateless, i.e., they do not store any session-related data that shall be persistent. Such a distributed, container-based architecture of the proposed solution is in line with the state-of-the-art software design and deployment.

As mentioned in the previous section, the number of the containers implementing the OF evaluation functionality is scaled automatically using the standard public cloud autobalancing facilities. However, the container with OE also will be scaled, facilitating a massive parallel optimisation process performed by a distributed evolutionary algorithm. In this approach, each OE container is viewed as so-called island. The optimisation starts with a single OE container (island). Whenever OE does not improve the result for a predefined interval, it initiates the connection to the same container with the same configuration scheme it got earlier from SD.

It is assumed that containers with OE and OF will be deployed in a cluster with the Kubernetes container-orchestration system, which is available in all major cloud facilities, including AWS, Azure, CloudStack, GCE, OpenStack, OVirt, Photon, VSphere, IBM Cloud Kubernetes Service, as well as can be installed locally. Both OE and OF will be provided as separate pods, which means that they will not share volumes and thus can communicate only by using a network. Since only a single program is planned to be executed in a single pod, the software dependencies will be highly decoupled. Similarly, the number of instances of OE will not directly influence the number of instances of OF, as both of these cardinalities will be decided independently by so-called Horizontal Pod Autoscaler, whose role is to scale automatically the number of pods based on observed CPU utilization of the nodes executing the corresponding containers. This autoscaler is implemented as a controller executed periodically (with a period controlled by the controller manager’s --horizontal-pod-autoscaler-sync-period flag, 30 seconds by default). The controller fetches the per-pod resource metrics for each targeted pod and calculates the utilisation value as a percentage of the equivalent resource request on the containers in each pod. The mean of the utilisation across all

targeted pods is used to compute a ratio for scaling the number of desired replicas, according to equation

$$\text{desired replicas} = \text{current replicas} \cdot \left\lceil \frac{\text{current CPU utilization value}}{\text{desired CPU utilization value}} \right\rceil.$$

In future, Kubernetes is likely to include other metrics, such as based on memory footprint or network traffic.

Since the number of pods can change periodically, the pods should not been accessed directly by other SAFIRE components. Instead, a Kubernetes *service* needs to be created to define a related micro-service. Each service is assigned with an IP address and a port. A service can be discovered with either an environment variables or DNS. The latter option is planned to be used for OE deployment. It requires an installation of CoreDNS, a flexible and extensible DNS server, in the cluster. This server is available in a form of so-called Kubernetes Addons. It observes the cluster using Kubernetes API and creates a set of the corresponding DNS records for each new services found, including an appropriate DNS A record. For example, *fitnessfunction.optimisation* DNS record will be created for a service named *fitnessfunction* in namespace *optimisation*. Thus, the OE containers can access OF without knowing its IP addresses, just by querying the appropriate domain name. The OF service, as accessible only by (OE) containers inside the same cluster, should be published with behaviour *ClusterIP*. It makes the service only reachable from within the cluster. However, the OE service should be invoked by the SAFIRE SD module, that can be installed outside the cluster. Thus, it must be exposed using an external (public) IP address, which can be obtained using one of 3 publishing behaviours: *NodePort*, *LoadBalancer* and *ExternalNode*. The behaviour to be selected depends on the target cloud and, for example, if a cloud provides an external load balancer, the *LoadBalancer* behaviour should be selected. Regardless the choice, an internal load balancer is available to split evenly the inbound transfer into the available replicas of OE, as generated by Horizontal Pod Autoscaler.

The OF containers will be executed independently from each other, but the OE containers are expected to share the best individuals found so far during the certain optimisation process, according to the so-called island model of a parallel evolutionary algorithm. Each container corresponds to one island and evolves for most of the time. However, periodically the best solutions are exchanged between islands in a process which is known as *migration*. Various migration topologies have been studied for example a ring, as discussed in [Sudholt, 2015]. Since the number of islands in the proposed implementation is dynamic and the islands would have the same IP address, the direct communication between them is hindered. Instead, they can contact another container, sending it the individuals together with their fitness values and receiving individuals generated by other OEs. This communication scheme is logically identical with a complete graph. As the container storing the individuals needs to store them in the key-value form, it may be realised with one of the popular in-memory NoSQL databases, such as Redis, MongoDB (Percona memory storage engine) or Memcached. As the islands execute the same optimisation algorithm, they follow the homogeneous island model.

The parameters that will need to be evaluated in the full prototype are:

- emigration policy, including the number of migrants, their quality (best, worst or random) and the decision whether the migrants are removed from the island or copied (pollination),
- immigration policy, including whether the migrants are selected as parents or replace the worst or random individuals,
- migration interval (periodic or random),
- number of migrants.

The architecture of the Kubernetes cluster for optimisation and reconfiguration engine is presented in Figure 5.

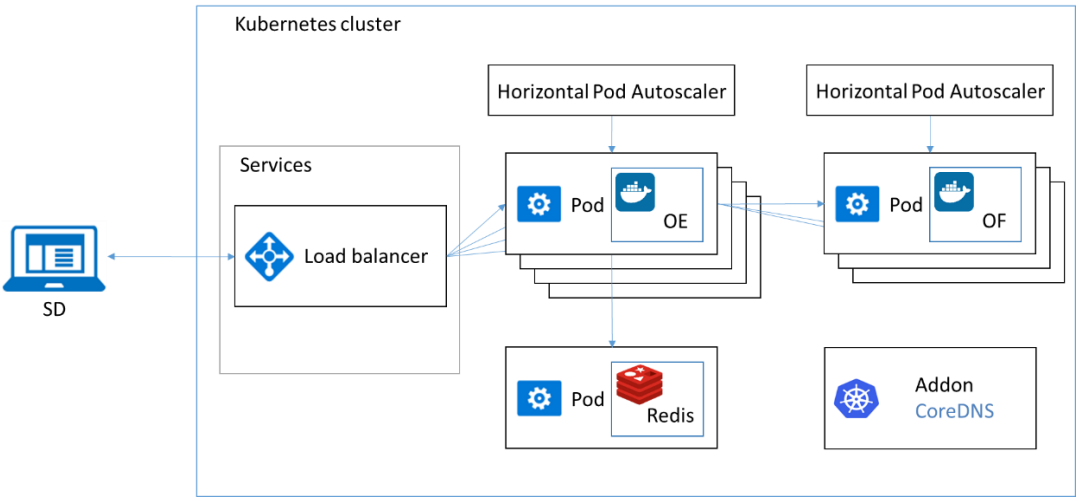

Figure 5. Kubernetes cluster for optimisation and reconfiguration engine

6.4.2 OE configuration

In line with the SAFIRE assumptions, the OE is developed in a way to enlarge its generality. OE is agnostic with respect to the provided metrics, described with the Metrics API, specified earlier in this document. Nevertheless, some minor customisation is needed before the OE compilation and cloud deployment. It includes selecting of the engine type among the versions implementing various algorithms specified in deliverable D3.1 [SAFIRE D3.1, 2018]. Three parameters need to be selected:

- single objective vs multi objectives,
- bounded execution time vs unbounded execution time,

- selecting production processes (recipes) among alternatives vs applying all specified recipes.

Except from the above enumerated parameters, the only custom information needed is the domain address of the OF block that needs to be invoked in order to obtain fitness value for the individuals created during the search-based optimisation process (or, alternatively, the domain address of PA if it is planned to be used instead of OF).

The implementation details of this configuration is planned to be described in future SAFIRE Deliverable D3.4 Full Prototype of Dynamic and Predictable Reconfiguration and Optimisation Engine.

6.4.3 OF configuration

As stated before in this document, OF is defined using Factory Description Language (FDL). The FDL specification is to be parsed by Optimisation Engine Configurator available in the full prototype of OE, generating two artefacts: a OF container to be deployed in a cloud and a Metrics API template to be used by other SAFIRE modules, in particular SD. SD module is planned to use that template by filling the values of the observable metrics and then to send such filled template to OE. OE executes the operators on the solutions generated according to that template and forwards them, in the Metrics API format, to the OF (or PA) module for evaluation of the value of these solutions. This flow is presented in Figure 6.

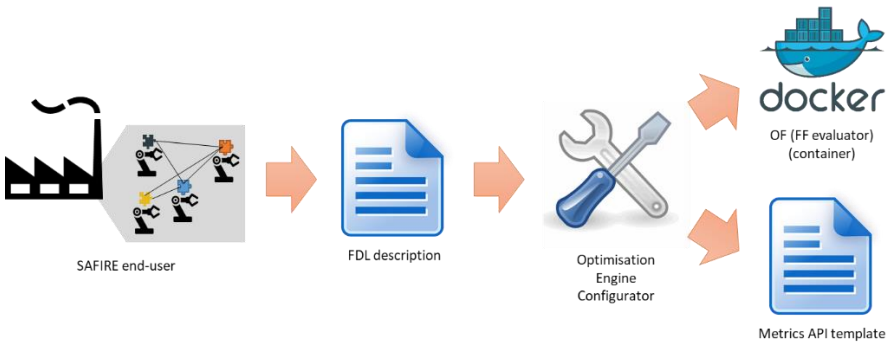

Figure 6. Design flow for OF configuration building

The container with a customised OF (fitness function evaluator) will be automatically built out of a code in the Scala language, automatically generated from FDL. The code to be generated includes one class, inheriting from the *IAObjectiveFunction*, present in the early prototype. The following functions need to be overwritten: *predictKeyObjectives* and *toReportString*. The *predictKeyObjectives* function takes two parameters: a configuration (of type *Configuration*, as defined in Metrics API) and a map from control metrics' names (strings) to their values (of type *Value*, as defined in Metrics API) and returns a map from objectives' names (strings) to their values (of type *Value*, as defined in Metrics API), as computed by OF. The body of this function is expected to assign task to the compatible factory resources and schedule them, using one of the schedulers provided in the interval algebra implementation, developed in the

course of the SAFIRE project, e.g. *FIFOScheduler*. After scheduling, the key objective values (e.g. makespan) should be updated and these new values returned to the caller.

The Metrics API template is built using class *ConfigurationType*, as defined in Metrics API. It should include three array lists: ControlledMetricType, ObservableMetricType and KeyObjectiveType. All these lists store key-value tuples, where the first element of the touple is the metrics name (string) and the second one is the appropriate value, as defined in Metrics API.

6.4.4 Cooperation with other SAFIRE modules

As shown in Figure 7, the modules implementing the optimisation and reconfiguration engine communicate with two other SAFIRE modules: Situation Determination (SD) and Predictive Analytics (PA).

The role of Situation Determination (SD) is to identify the presence of particular patterns in factory configuration state and communicate this to the SAFIRE OE component. Such communication results in execution of the optimisation process.

Since the Objective Function is where Business Case-specific intelligence resides, the outputs from SD will be passed on to the OF via the Optimisation Engine. The OF module computes the fitness value for the solutions found during the search-based optimisation process performed by OE. The best solutions with respect to the key objectives provided in the initial HTTP POST issued by SD, are returned to SD in order to be applied in the manufacturing process under optimisation.

PA plays two roles in the presented communication scheme. The first role is to query it by OE instead of OF to provide a fitness value of a certain individual. This role is relevant in situation when the nature of the process to be optimised is difficult to be described by analytic formulas and hence to the proposed max-plus and interval algebras. In such cases, the fitness values can be learn by PA by analysing inputs and outputs of the plant or device. This role is denoted with the arrow between OE and PA in Figure 7. The second role of PA is to serve as an aim in the value evaluation by OF. In certain scenarios, it may be the case that the model used for evaluation returns slightly different values than the ones obtained by the real plant or device. For example, the OF model can be built based on a certain set of target devices or plants and do not consider the variability of the certain device or plant under optimisation. Such discrepancy can be noted by PA and over time the PA module can learn more accurate properties of the optimised device or plant. This aiding role of PA is denoted with the dashed line between OF and PA in Figure 7.

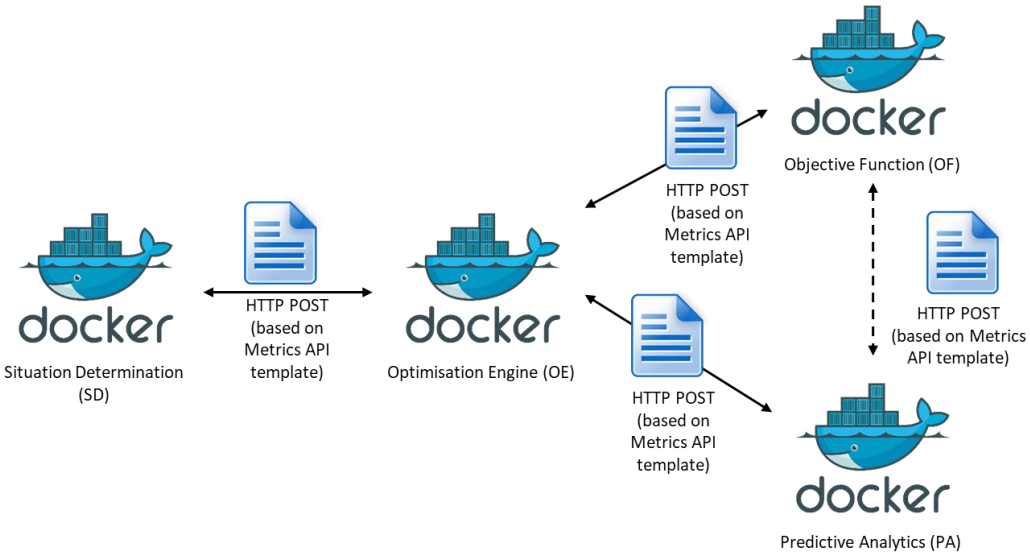

Figure 7. Cooperation between SAFIRE modules

All HTTP POST shown in the figure include a configuration in a JSON form. An extract from such message content is shown in Figure 8. The sections for key objective metrics, observable metrics and controlled metrics can be easily identified. All three possible value types: Real, Integer and Nominal, are shown. For example, the presented controlled metrics is named *Std Weiss A 6 allocation*. Its type is nominal and can assume values from set {*Mixer 1*, *Mixer 2*, *Mixer 3*, *Mixer 4*, *Mixer 5*}, denoting the resource that the particular recipe can be processed with. In this particular example, *Mixer 3* has been selected.

The possible situations are problem-dependent, but it is assumed that at least unavailability of resources or impossibility of executing certain tasks in a provided timeframe are determinable in any plant or device. This information is transmitted in a form of observable metrics, as shown in Figure 8. The related observable metrics is named *Mixer 1 availability* and is of type INT, that can assume either 0 or 1 for not availability or availability, respectively. In the example case, *Mixer 1* is available.

keyObjectiveMetricTypes=[KeyObjectiveType[name=makespan,valueType=ValueType.Real[min=0.0,max=1.7976931348623157E308,typ=REAL],units=n/a,searchDirection=MINIMIZING]]

(...)

ObservableMetricTypes=[ObservableMetricType[name=Mixer 1 availability,valueType=ValueType.Integer[min=1,max=1,typ=INT],units=n/a,sampleRate=SampleRate.EventDriven[]],

(...)

ObservableMetricType[name=Weiss Basis A 9 start,valueType=ValueType.Integer[min=0,max=0,typ=INT],units=n/a,sampleRate=SampleRate.EventDriven[]], ObservableMetricType[name=Weiss Basis A 9 end,valueType=ValueType.Integer[min=2335,max=2335,typ=INT],units=n/a,sampleRate=SampleRate.E

```
ventDriven[]]

(...)

controlledMetrics={Std Weiss A 6 allocation=Nominal(value=Mixer
3,type=ValueType.Nominal[name=Std Weiss A 6 allocation type ,values={Mixer 1,Mixer 2,Mixer
3,Mixer 4,Mixer 5},typ=NOMINAL]),

(...)
```

Figure 8. Extracts from an example configuration for OAS use case

6.4.5 Implementation of the SAFIRE security framework

The compliance with the Safire security framework (SSF) requires trusted channels, which can be obtained with TLS-based connections using the pre-generated keys and certificates. Such channels represent the *communication integrity* aspect of security. The Optimisation Engine (OE) container can be invoked only by the SAFIRE Situation Determination (SD) module, authorised earlier following the SSF. Similarly, the connection between OE and Objective Function (OF) uses TLS and, when exists, the connection between OF and Predictive Analytics (PA) as well. Notice that OE and OF containers are generated independently for each end user, thus there is no possibility of accessing other user data as long as end users do not reuse the same certificates or keys.

OE is planned to use TLS certificates on both the server and the client side to provide a proof of identity. Consequently, it requires both the client and server to own a certificate signed by a specific Certificate Authority (CA), for example using OpenSSL. This way of securing the connection is widely available, supported by Docker and the majority of public cloud vendors. Similarly, Kubernetes supports TLS and even each Kubernetes cluster has its own root CA that can be used by the cluster components to validate the clients and servers' certificates. If deployed to a Kubernetes cluster, OE, OF and PA can request a certificate signing using the certificates.k8s.io API. Similarly, a root CA (named AWS Certificate Manager - ACM) is available when using Amazon Web Services (AWS). TLS is also supported by the Network Load Balancer, which can be used with both EC2 and Fargate Launch Style. TLS is also available in IBM Cloud, Azure, OpenStack and other public clouds so choosing this protocol does not limit the future deployment options. The independence of the execution containers and of the data they employ (subject to disciplined use of access keys, explained later in this section), along with the guarantee that the above-enumerated containers can be invoked only by a *trusted invocation path*, covers the *architectural integrity* aspect of security with the combination of *communication integrity*.

The Next Generation Access Control (NGAC) methodology and configurable policy, as described in SAFIRE deliverable D5.5 [SAFIRE D5.5, 2018], are responsible for the above-mentioned *discipline in the use of access keys*. The components of the Reconfigurable and Optimisation engine, OE and OF, include the appropriate PEP (Policy Enforcement Point) subcomponent for communication with Policy Server (PS). The possibility of including PEPs into OE and OF has been raised thanks to The Open Group's

modification of the NGAC functional architecture, which unbundled the PEPs and Resource Access Points (RAPs) from the NGAC perimeter. As it states in D5.5, the appropriate PEP subcomponents for OE and OF are developed by following simple templates that include calls to the Policy Server through the RESTful Policy Query Interface. The PEP subcomponents consist of a single decision based on calling the Policy Server and either returning an error condition to its caller if the Policy Decision Point (PDP), a module of PS, returned deny or completing the access operation if the PDP returned *permit*.

The Policy Server API includes functions *initsession* and *endsession* which allow a session identifier to be registered as a proxy for a user identifier, where *user* denotes trusted components such as OE and OF. Then the *access* checks are made with the session identifier instead of a user identifier (i.e., its key). A particular SAFIRE component execution can be associated with a *user id* under a *session id*, which is a long string that is infeasible for an imposter to guess. The *session id* is passed to the OE (*user* according to the definition from D5.3) component to use when it makes requests to a policy enforcement point for OF (*object* according to the definition from D5.3) access. When the OE container instance is initiated by the SD component (as described earlier in this document), the initiator registers a *session id* to the policy administration API and passes that *session id* to the OE container. Only the initiator (SD), not the OE container, has the authorisation to call this API (enforced by its authorised TLS connection to the server). When the containers want to access their data, they make a request to a PEP for that data kind, supplying the *session id* in the request. The PEP, in turn, asks the PDP through the Policy Query Interface *access* request, using the *session id* instead of a *user id*, whether the access is permitted according to the policy and then enforces this decision accordingly. In this scheme, the PEPs own the data, that is, they are granted exclusive access to all the data so that they can enforce access according to the policy decisions of the PDP. The same scheme is followed when OE initiates the OF container.

The whole communication scheme based on SSF is presented in Figure 9. In the figure, two session ids are generated: *session_id* is generated by the SD module using the SD's user key (*sd_user_id*) and *session2_id* is generated by OE using the OE's user key (*oe_user_id*). The *access* call to PS checks whether the user (either SD or OE) is permitted to *execute* the object (OE and OF, respectively). If the access is permitted, the component functionality is executed and the results are returned to the invoker. Then the session is ended. In the figure, the optional PA module is not present. Its presence would require establishing the third session based on the OF's user key (*of_user_id*) and then using it for invoking the functionality of PA.

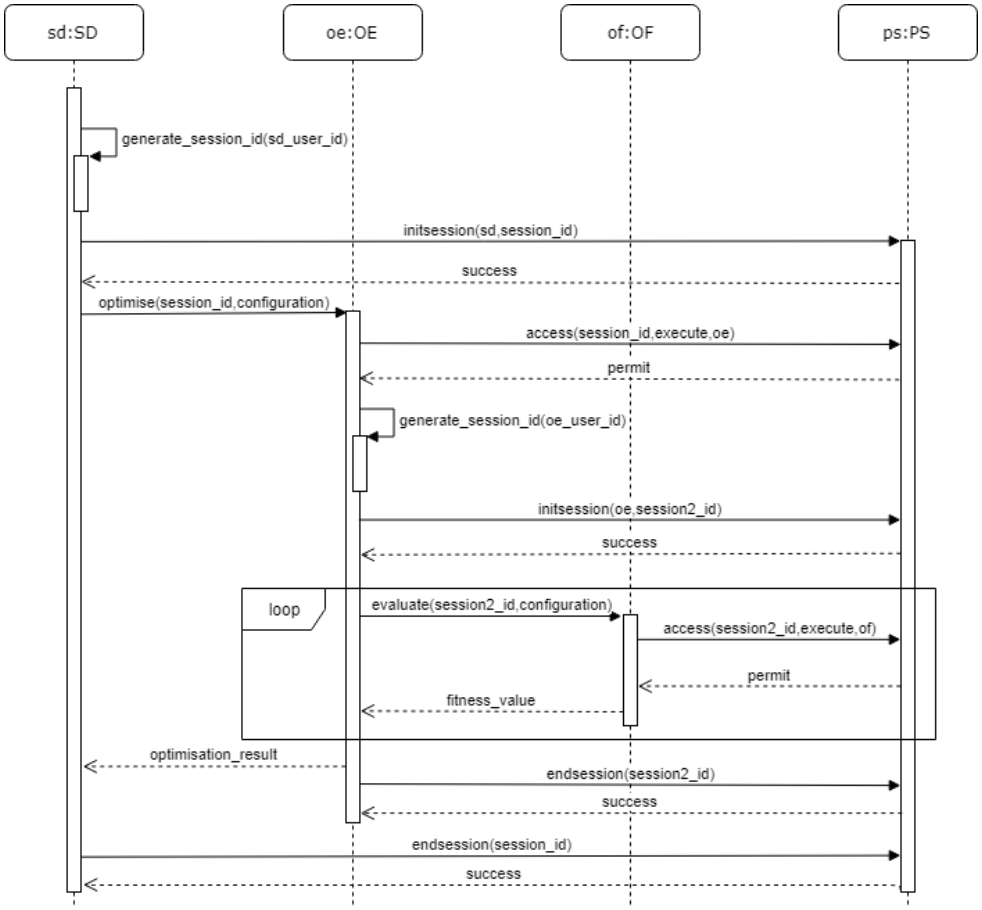

Figure 9. Extracts from an example configuration for OAS use case

7. BUSINESS USE CASES

7.1 EXPLICIT DESCRIPTION OF THE OBJECTIVE FUNCTION EXAMPLE - OAS USE CASE

This section describes modelling of a small plant with a fixed architecture with respect to the number & types of resources (silos, conveyors, tubes, scales, pipelines, mixers etc.) and connections between them. The plant to be modelled is shown in Figure 10.

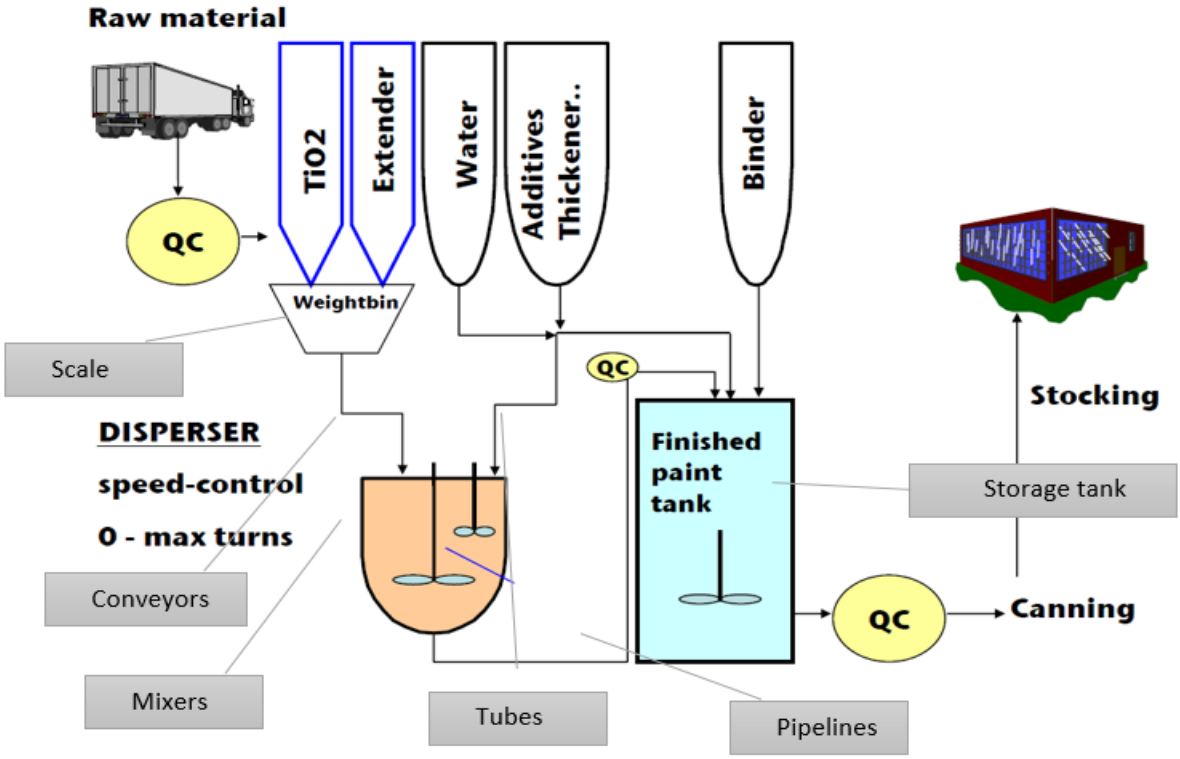

Figure 10. Example of a simple plant

As shown in the above Figure, several resources have been identified, namely *Silo*, *Scale*, *Conveyor1*, *Conveyor2*, *Mixer1*, *Mixer2*, *Tube1*, *Tube2*, *Pipeline1*, *Pipeline2*, *Storage tank*. An example of a recipe for manufacturing certain amount of a certain commodity is shown in Table 6. The resources enumerated as elements in a single set have to be allocated simultaneously. The tasks in this recipe forms a manufacturing job whose tasks have to processed in a sequence and are related with the immediate precedence relationship relation between neighbouring tasks, equivalent to the meet relation in Allen's interval algebra. The last task in the recipe is tagged with the manufactured products and its amount.

Assuming that Conveyors are connected to the same silos, it may be the case that they cannot be used simultaneously. Then, a mutex has to be created to guarantee their mutual exclusiveness. Similarly, due to the limited connectivity of the resources in the plant, resource affinities between conveyors, mixers and pipelines with the same indices have to be defined. Our extension to the state-of-the-art in Interval Algebra modelling

(see deliverable D3.1 for a detailed description of Max-plus and Interval Algebras) is sufficient to model this.

Table 6. Example recipe of a manufacturing job

Task name	Compatible resources	Processing time	Relations with other tasks	Tag
Task 1	{Silo,Scale}	{Silo,Scale}: time1	Task1 m[1] Task2	
Task 2	{Scale,Conveyor1,Mixer1}, {Scale,Conveyor2,Mixer2}	{Scale,Conveyor1,Mixer1}: time2A {Scale, Conveyor2,Mixer2}: time2B	Task1 m Task2, Task2 m Task3	
Task 3	{Tube1,Mixer1}, {Tube2,Mixer2}	{Tube1,Mixer1}: time3A {Tube2,Mixer2}: time3B	Task2 m Task3, Task3 m Task4	
Task 4	{Mixer1},{Mixer2}	{Mixer1}: time4A {Mixer2}: time4B	Task3 m Task4, Task4 m Task5	
Task 5	{Mixer1,Pipeline1},{Mixer2,Pipeline2}	{Mixer1,Pipeline1}:time5A {Mixer2,Pipeline2}:time5B	Task4 m Task5	Commodity1: Amount1

7.2 EXPLICIT DESCRIPTION OF THE OBJECTIVE FUNCTION EXAMPLE - ONA USE CASE

The beneficiaries of optimisation in the ONA use case are customers of ONA: typically SMEs with a certain number of WEDM machines of various models. The Key Objective metrics, Controlled Metrics and Observable Metrics have been identified, as listed in Table 7.

The considered WEDM machines are characterised with the maximum size of the processed part that can be processed and their usage cost per hour. Large and special machine models are more expensive than the standard and small series. Consequently, the cost per hour of the machine usage must be affected by the machine model and considered as the optimisation problem. Similarly, the cost and time of processing depends on the wire type and its consumption (related to the wire speed), that is set according to the

[1] In the applied Allen's Interval Algebra, m operator denotes the *meet* relation, which means that the second task is invoked directly after the first one.

application and its conditions. The machine can operate in so-called eco mode, which can save consumables cost (especially the wire) with a possible penalty in cutting speed.

The part processing time in each machine using wire of a certain type and diameter can be estimated with sufficient accuracy based on data provided by ONA for both standard and eco modes. Due to this possibility of determining the interval lengths for a single part processing, this optimisation problem can be described with Interval Algebra, mentioned in the previous subsection. Each task corresponds with a single part processing and is independent from other tasks. Such tasks are assigned to the available machines considering their processing time for the given application and its conditions, selecting both the wire and the processing mode, so that the chosen key objective metric (OEE or cost/part) is optimal in a given time window.

Our extension to the state-of-the-art in Interval Algebra modelling (see deliverable D3.1 for a detailed description of Max-plus and Interval Algebras) is sufficient to model this use case.

Table 7. Metrics in the ONA use case

Key objective metrics	OEE (maximise) Cost/part (minimise)
Controlled metrics	Wire type Wire diameter Machine model Machine eco-mode
Observable metrics	Actual process time per job/part Actual wire composition Energy consumption

7.3 PREDICTED OBJECTIVE FUNCTION FROM ANALYSIS OF PROCESS DATA - ELECTROLUX USE CASE

In the Electrolux use case, the Key Objective metrics, Controlled Metrics and Observable Metrics have been identified, as listed in Table 8. The influence of the controlled metrics on key objective and observable metrics can be determined by Predictive Analytics module based on the values obtained experimentally. The proposed Metrics API, Schema Interface and the Objective Function interface are capable of expressing this optimisation problem.

Table 8. Metrics in the Electrolux use case

Key objective metrics	Boiling detection error Temperature estimation error
Controlled metrics	Power/energy profile Data acquisition period

Observable metrics	Type of pot Type of food/water amount Cooking speed Expected cooking level Power Current Voltage Temperature Vibration

8. REQUIREMENTS COVERAGE (TABLE)

This section presents the coverage of the related requirements by the functionalities of the three components developed in the course of WP3 Dynamic and Predictable Reconfiguration & Optimisation Engine. The requirements presented in this section are consistent with the ones enumerated in section 5 of SAFIRE deliverable D1.1 [SAFIRE D1.1, 2017]. In all the tables presented in this section, two columns including the state of the requirements coverage for the early (EP) and full (FP) prototypes are provided; the former has been taken from SAFIRE deliverable D3.2 [SAFIRE D3.2, 2017].

8.1 RECONFIGURATION

No.	Requirement	Overall Priority	EP	FP
U27	Able to reconfigure the selection of production line	SHALL	+	+
U28	Able to reconfigure composition of production batches	SHALL	+	+
U29	Able to reconfigure priority management functions	SHALL	+	+
U30	Able to reconfigure part express management	MAY	+	+
U31	Able to suggest process improvements	SHALL	+	+
U32	Able to reconfigure job scheduling	SHOULD	+	+
U33	Able to reconfigure process adaptation in response to operator actions	SHOULD	+	+
U34	Able to reconfigure the communication nodes	SHOULD	+	+
U35	Able to reconfigure Hob accessory express management / suggestions	SHOULD	+	+
U36	Able to reconfigure Pot position suggestions	SHOULD	+	+
U37	Able to reconfigure Hob recipes manager and suggestions	SHOULD	+	+

8.2 OPTIMISATIONS

No.	Requirement	Overall Priority	EP	FP
U39	Able to use knowledge from previous monitoring and analysis for optimisations	SHALL	+	+
U41	Able to optimise based on KPIs configured based on data analytics, situational monitoring & analysis results	SHOULD	+	+
U42	Able to optimise overall equipment effectiveness (OEE)	SHALL	+	+
U43	Able to optimise process speed	SHALL	+	+
U44	Able to optimise energy consumption	SHOULD	+	+
U45	Able to optimise part quality	SHOULD	+	+
U46	Able to optimise consumables consumption	SHALL	+	+
U47	Able to optimise tool wear	MAY	+	+
U49	Able to optimise batch scheduling in proNTo	SHOULD	+	+

8.3 PERFORMANCE

No.	Requirement	Overall Priority	EP	FP
U115	Does not negatively affect the usual production processes	SHALL	+	+
U116	Support for scalability in the size of cloud and computing resources	SHALL	+	+
U117	Support for horizontal scalability to many machines	SHALL	+	+
U118	Capable of real time data ingestion (registering data)	SHALL	+	+
U119	Capable of batch processing of data (offline analysis)	SHALL	+	+
U120	Capable of real time data processing	SHALL	+	+
U121	Capable of providing real time reconfigurations / optimisations (subject to network throughput limits)	SHALL	+	+
U122	Able to analyse relevant data within a given timeframe	SHALL	+	+
U123	Capable of storing up to 5 TB/year/machine with resource recycling facilities	SHALL	+	+

U124	Provides support for Machine Learning (Supervised / Unsupervised / Anomaly Detection)	SHALL	+	+
U125	Able to achieve required precision on cooking process estimation / optimisations	SHALL	+	+

8.4 INTERFACES

No.	Requirement	Overall Priority	EP	FP
U127	Provides a web based user interface	SHALL	+	+
U128	Implemented as a set of web services / web based solution	SHALL	+	+
U129	Able to customise the interfaces	SHALL	+	+
U132	Supports human-machine interface on multiple devices	SHALL	+	+
U133	Able to interface with third-party reporting systems	SHOULD	*[2]	+[3]
U134	Able to interface with third-party dashboards	SHALL	+	+
U136	Supports reconfiguration of the process / machine by remote commands	SHALL	+	+
U137	Able to be integrated with proNTo	SHALL	*[4]	+[5]
U138	Supports the ONA Machine Protocols	SHALL	*[6]	+[7]

8.5 COMMUNICATIONS

No.	Requirement	Overall Priority	EP	FP
U142	Support for Internet/Ethernet communications	SHALL	+	+
U143	Support for VPN connectivity	SHOULD	+	+
U144	Support for Machine / Cloud protocols	SHALL	+	+
U145	Support for Machine / Fog computing	SHOULD	+	+[8]

[2] The outputs from OE can be used with the third-party reporting systems, but a customised bridge is needed.
[3] The outputs from OE can be used with the third-party reporting systems, but a customised bridge is needed.
[4] The proNTo software can be used for data acquisition which, in turn, can be provided by SD to OE.
[5] The proNTo software can be used for data acquisition which, in turn, can be provided by SD to OE.
[6] The ONA Machine Protocol can be used for data acquisition which, in turn, can be provided by SD to OE.
[7] The ONA Machine Protocol can be used for data acquisition which, in turn, can be provided by SD to OE.
[8] As described in paper P. Dziurzanski, J. Swan and L.S. Indrusiak, Smart factories scheduling using edge computing and clouds, submitted to the 1st International Workshop on Trustworthy and Real-time Edge Computing for Cyber-Physical Systems (TREC4CPS).

U146	Support for communications between operators and platform	SHOULD	+	+
U147	Support for communications with business analytics	MAY	+	+

8.6 HARDWARE/PLATFORM/DEVICES

No.	Requirement	Overall Priority	EP	FP
U148	Supports continuous operation (24h per day, 7 days per week)	SHALL	+	+
U149	As platform independent as possible	SHALL	+[9]	+[10]
U150	Supports multiple deployment scenarios including Cloud / On Premise / Hybrid / Third-party Managed	SHOULD	+	+
U151	Provided connector with proNTo system operates under Windows	SHALL	*	+

[9] As a container, possible to be executed on any system supporting Docker.
[10] As a container, possible to be executed on any system supporting Docker.

9. REFERENCES

[Allen, 1983] James F. Allen, Maintaining knowledge about temporal intervals. Commun. ACM 26, 11 (November 1983), 832-843

[Burke et al, 2003] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In Handbook of metaheuristics, pages 457–474. Springer, 2003

[Burns et al, 2000] Alan Burns, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John Stankovic, and Lorenzo Strigini. 2000. The meaning and role of value in scheduling flexible real-time systems. Journal of systems architecture 46, 4 (2000), 305–325.

[Chen et al, 2017] Baotong Chen, Jiafu Wan, Lei Shu, Peng Li, Mithun Mukherjee, and Boxing Yin. 2017. Smart Factory of Industry 4.0: Key Technologies, Application Case, and Challenges. IEEE Access (2017).

[Diaz et al, 2011] Nancy Diaz, Elena Redelsheimer, and David Dornfeld. 2011. Energy consumption characterization and reduction strategies for milling machine tool use. Glocalized solutions for sustainability in manufacturing (2011), 263–267.

[Durillo and Nebro, 2011] Juan J Durillo and Antonio J Nebro. 2011. jMetal: A Java framework for multiobjective optimization. Advances in Engineering Software 42, 10 (2011), 760–771.

[Dziurzanski et al, 2018] Piotr Dziurzanski, Jerry Swan, Leandro Soares Indrusiak, 2018. “Value-Based Manufacturing Optimisation in Serverless Clouds for Industry 4.0”, Genetic and Evolutionary Computation Conference (GECCO) 2018, Kyoto, Japan (submitted)

[Gendreau and Potvin, 2010] Michel Gendreau, Jean-Yves Potvin. 2010. Handbook of Metaheuristics, Springer Science & Business Media, 2010.

[Goto, 2014] Hiroyuki Goto. 2014. Introduction to max-plus algebra. In Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation. ACM, 21–22.

[Holland, 1992] John H. Holland. 1992. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge, MA, USA.

[Khemka et al, 2015] Bhavesh Khemka, Ryan Friese, et al. 2015. Utility functions and resource management in an oversubscribed heterogeneous computing environment. IEEE Trans. Comput. 64, 8 (2015), 2394–2407

[Kronberger et al, 2013] Gabriel Kronberger, Michael Kommenda, Stefan Wagner, and Heinz Dobler. 2013. GPDL: a framework-independent problem definition language for grammar-guided genetic programming. In Proceedings of the 15th annual conference

companion on Genetic and evolutionary computation (GECCO '13 Companion), Christian Blum (Ed.). ACM, New York, NY, USA, 1333-1340. DOI: http://dx.doi.org/10.1145/2464576.2482713

[Leclerc et al, 2016] Guillaume Leclerc, Joshua E Auerbach, Giovanni Iacca, and Dario Floreano. 2016. The seamless peer and cloud evolution framework. In Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. ACM, 821–828.

[Papadimitriou and Steiglitz, 1982]Christos H. Papadimitriou and Kenneth Steiglitz. 1982. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Ma et al, 2017] Ning Ma, Xiao-Fang Liu, Zhi-Hui Zhan, Jing-Hui Zhong, and Jun Zhang. 2017. Load balance aware distributed differential evolution for computationally expensive optimization problems. In GECCO Proceedings Companion, 2017. ACM, 209–210.

[Mendez et al, 2007] Carlos A. Méndez, Jaime Cerdá, Ignacio E. Grossmann, Iiro Harjunkoski, Marco Fahl, State-of-the-art review of optimization methods for short-term scheduling of batch processes, Computers & Chemical Engineering, Volume 30, Issues 6–7, 2006, Pages 913-946.

[SAFIRE D1.1, 2017] Electrolux, OAS and ONA, D1.1 Application Scenarios Requirements Analysis, SAFIRE project deliverable, 2017.

[SAFIRE D1.2, 2017] University of York, D1.2 Optimisation Metrics and Benchmarking, SAFIRE project deliverable, 2017.

[SAFIRE D3.1, 2018] University of York, D3.1 Methodology for Dynamic and Predictable Reconfiguration and Optimisation Engine, 2018.

[SAFIRE D3.2, 2018] University of York, D3.2 Early Specification of Dynamic and Predictable Reconfiguration and Optimisation Engine, SAFIRE project deliverable, 2018.

[SAFIRE D4.1, 2018] ATB, D 4.1 Methodology for Situational Awareness, SAFIRE project deliverable, 2018.

[SAFIRE D4.3, 2018] ATB, D 4.3 Early Prototype of Situational Awareness Services, SAFIRE project deliverable, 2018.

[SAFIRE D5.5, 2018] The Open Group, D 5.5 Full Specification of SPT Framework, SAFIRE project deliverable, 2018.

[Salza et al, 2016] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. 2016. Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud. In GECCO Proceedings Companion, 2016. ACM, 121–122

[Salza et al, 2017] Pasquale Salza, Erik Hemberg, Filomena Ferrucci, and Una-May O'Reilly. 2017. Towards evolutionary machine learning comparison, competition, and collaboration with a multi-cloud platform. In GECCO Proceedings Companion, 2017. ACM, 1263–1270.

[Simons et al, 2017] Chris Simons, Jerry Swan, Krzysztof Krawiec, and John Woodward. "Metaheuristic Design Patterns". In: Emergent Research on the Application of Optimization Algorithms. IGI Global, 2017.

[Soares Indrusiak and Dziurzanski, 2015] Leandro Soares Indrusiak, Piotr Dziurzanski, "An interval algebra for multiprocessor resource allocation". SAMOS 2015: 165-172

[Swan et al, 2015] Jerry Swan, Steven Adriaensen, Mohamed Bishr, Edmund K. Burke and others. "A Research Agenda for Metaheuristic Standardization". In: Proceedings of the Eleventh Metaheuristics International Conference (MIC), Agadir, Morocco. 2015. url: https://goo.gl/kC06p5.

[Sudholt, 2015] Dirk Sudholt, Parallel evolutionary algorithms, Springer Handbook of Computational Intelligence, 2015, 929-959

[Woodward et al, 2014] John Woodward, Jerry Swan, and Simon Martin. "The 'Composite' Design Pattern in Metaheuristics". In: Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion. GECCO Comp '14. Vancouver, BC, Canada: ACM, 2014, pp. 1439–1444. ISBN: 978-1-4503-2881-4. DOI: 10.1145/2598394.2609848.