SEVENTH FRAMEWORK PROGRAMME
THEME – Energy Efficient Buildings

EeB-ICT 2011.6.4. ICT for energy-efficient building and spaces of public use

Self learning Energy Efficient builDing and open Spaces

GA No.285150

Implementation and refinement of self-learning algorithms and global optimization in the two pilots

Work Package	WP5 - Self learning and global optimization		
Task	T5.5 Implementation and refinement of self-learning algorithms and global optimization		
Revision	0		
Due date	31/07/2014	**Submission date**	12/09/2014
Dissemination level	PU	**Deliverable type**	R

Authors	Rui Máximo Esteves (UiS); Shane Montage (USALF); Sunil Vadera (USALF) and Jia Wu (USALF)
Verification	Sunil Vadera (USALF)
Approval	Noemi Jimenez-Redondo (CEMOSA)

Table of Contents

List of Figures

List of Tables

1 Introduction

This report presents the outcome of Task 5.5, which aimed to refine Optimization methods on sample scenarios, consider their applicability for efficient energy management for buildings, and provide them for the SEEDS architecture. Task 5.5 was stated as follows:

> **Task 5.5: Implementation and refinement of self learning algorithms and global optimization into the two validation pilots**
> Participants: FHG-EAS, CEMOSA, UiS (lead), USALF, CIDAUT
>
> This task will provide the support for the implementation of the Self-learning energy management system developed within the SEEDS project into the two validation pilots. Some minor re-adjustment or refinement of the algorithms may be required.

The current deliverable presents a refinement of the algorithms for Self-Learning and Optimization that were previously described in the documents D5.2 (Vadera et al., 2013) and D5.3 (Esteves, 2013). The refinement aims to solve specific issues reported in the mentioned reports, namely with strategies to improve computational efficiency and accuracy of the algorithms.

In this document we outline necessary procedures to adopt the components to conditions in a particular building on two example pilot buildings: one in Madrid, Spain and a second in Stavanger, Norway. During the retrofitting process access might be limited to certain system components due to constructional, policy and security reasons. We aim to identify and address such major components from the perspective of SEEDS building management system.

2 Optimization

The OPT algorithm optimizes the energy consumption constrained by the following comfort variables: Temperature, Air quality and Illuminance. Contrary to the temperature and air quality, the illuminance has no inertia effect. The control of Illuminance is efficiently implemented by a set of straightforward conditions, which does not require the complexity of the PSO neither the need for SL. The computational resources necessary for determining the energetically optimized control setting for the illuminance are negligible, while for determining the control settings for the Temperature and Air quality can be a challenge.

Deliverable D5.3 highlighted the computational challenges tackled by SEEDs. In D5.3 the JAVA prototype developed for the Helicopter Garage (details presented in D2.8), was used to study the execution time of the different components. The conclusion of the D5.3 was that the majority of the execution time is consumed by the repeated calls required by optimisation when executing the SL component. Each call to SL component consumed 0.4 s and the execution time of the rest of the components is neglect able. The number of calls to the SL by the OPT is directly dependent on the number of iterations that are necessary for the OPT to converge and the complexity of the optimization problem that OPT has to solve is exponential in the number of control variables. Therefore D5.3 stressed the need to develop new strategies that could accelerate the convergence of the OPT algorithm and minimising the time SL takes to forecast parameters. A more efficient convergence of the OPT algorithm allows SEEDS to simultaneously optimize energy consumption using a wider forecast interval and use modest and inexpensive computational resources.

2.1 *Strategies for accelerating OPT convergence*

SEEDS OPT aims to obtain the optimal control settings for discrete time intervals. In relation to forecasting, OPT can take decisions based on just one forecast or based on multiple forecasts. In the SEEDS context the first situation is called single step[1] forecast optimization and the second situation is called multistep forecast optimization. Since the single step is a particularization of the multistep forecast optimization for just one forecast interval, OPT algorithms are designed for multistep forecast and it is decided by the CONTROLLER to call OPT for one or more forecast intervals. In the OPT algorithm one particle has $d=p*n$ dimensions, where p is the number of control signals and n the number of steps to forecast. OPT defines the number of dimensions dynamically in running time according to requested by the CONTROLLER. Therefore, if the CONTROLLER calls OPT for only one forecast interval, OPT will instantiate the particles with the number of dimensions $d=p$. This flexibility allows SEEDS to balance the number of steps of forecast according to the available time to converge and desired performance.

2.1.1 Smart initialization of the PSO particles

A PSO particle is an abstract concept that represents a point in a d-dimensional search space. The position of this point can be represented by a vector with d-elements, where each element of the

[1] The concepts of ”single step” and ”multistep” corresponds to the concepts of ”single level” and ”multilevel” presented in the D5.3 (Esteves, 2013) The designation was changed in order to be more intuitive for the reader.

vector define a coordinate for this point. The elements of the vector (i.e.: the dimensions of the particles) correspond to the signals that SEEDS aims to control. Therefore, one coordinate (i.e.: one element of the vector) of a particle position corresponds to the value of one control setting in a SEEDS actuator. The PSO algorithm requires initial values for the positions of the particles. The choice of the initial positions for the PSO particles can greatly influence the speed of the convergence, especially in a high-dimensional search space (Richards and Ventura 2004). In D5.3 (Esteves, 2013), the selection of the initial values of the PSO particles (i.e.: value of the actuators) was random. As a consequence, it was observed that the number of interactions required to converge was very high when using a random initialization method.
Since some of the SEEDS equipment has inertia (e.g.: thermal inertia) it makes sense to initialize some particles with positions that corresponded to the optimum computed in the last interval. We observed that in most circumstances, the solution for the next interval is not radically different from the optimum found in the previous interval.
Comparing with the random method, the initialization of particles with the optimum solution found in the last interval has another considerable advantage. This is explained by a characteristic of the PSO: it keeps in memory the best solution found during the execution of the algorithm. Therefore, the initialization with the best previous solution guarantees that OPT always return it in case no better solution was found. While using the random initialization method, OPT might return a worse solution specially if the time slot available to run OPT algorithm is heavily constrained.
On the other side, initializing all particles with the optimum solution found in the past reduces the possibility of the PSO scan several different regions of the search space. This would restrict the chances of the algorithm to search for a new optimum that is radical different from the one found in the past interval. Therefore, the best results are expected by arranging a set of initial particles that is a combination of some particles that are carefully initialized (e.g.: with the best previous solution) with other particles that are randomly initialized.

SEEDS use the following strategies to initialize the particles:

a) Initialise one particle using the last optimal values

The general idea is that one particle is initialized with the optimized control setting that were determined in previews OPT. If optimized control settings from previews OPT are missing, then it initializes using the control settings for equivalent conditions in the past (control settings at an equivalent time exactly one week before). The purpose of this instantiation method is to start the PSO closer to the optimal solution and guarantee that if no better solution is found, **OPT always return an optimized result**.

OPT initialize this particle by reading values from two tables in the MySQL database (SEEDS Archive) The table “real” is filled with values that were observed in reality from sensors or actuators. Therefore, the table “real” refers to values in the past or in the present. The table “calculated” is filled with values that were calculated as the output of OPT. Therefore, the table “calculated” refers to the forecasts.

Lets consider that the particle has $d = \{u_{t1}, u_{t2}, \dots, u_{tn}\}$ dimensions, where $u_{t1}, u_{t2}, \dots, u_{tn}$are the control signals for the steps 1 until n.

The initial values for $u_{t1}, u_{t2}, \dots, u_{tn}$ are computed in two different procedures:

a) u_{t1} - OPT reads the values from Archive table “real” with the newest timestamp for all actuators. These values correspond to the values of the actuators that are observed in the real equipment. These values are used to fill the dimensions of the particle that corresponds to the actuators for the first step of forecast.
b) $u_{t2}, \dots, u_{tn}$ - OPT reads the values from the Archive table “calculated” with the timestamps matching the correspondent timestamps of the forecasts. These values were calculated by running multistep OPT previously, therefore it is expected to be very close to the optimum.

In the advent of the table “calculated” is empty for a required timestamp, OPT searches the table “real” instead. Since the required timestamp refers to the future, OPT searches in the table “real” for the timestamp one week ago in relation to the desired timestamp.

b) Instantiate one particle with maximum values and another with minimum values

To guarantee that the PSO scans the two extreme poles of the search space, two particles are initialized differently. One particle is initialized with the maximum values possible for each actuator and another particle is initialized with the minimum values.

c) Instantiate one particle with medium values for the actuators

The medium value that each actuator can assume is used for initialize one of the particles. It is used the value of 0.5 for the actuators with binary signals.

d) Instantiate the other available particles randomly

The previous strategies for the initialization presented in this section reflect some assumptions given by experts in the domain. However, there are no guarantees that following this assumptions leads to the best solution. Therefore, a considerable amount of particles are still initialized by attributing values randomly chosen within the range of the actuators.

The total number of particles that we found best in SEEDS is 50.

2.1.2 Dynamic OPT call

The concept of dynamic OPT call is to allow the CONTROLLER to call OPT with the different periodicity and forecast horizon according to the usage of the building. The controller decides about the forecast horizons that OPT will run and the available time that OPT have to compute the results. During the regular occupancy period of the building the CONTROLLER should call OPT successively in short periods of time. During the long periods of non-occupancy the CONTROLLER should call OPT with less frequency and with a bigger time slot, which allows the OPT to compute the optimization of larger time horizons. In the context of the present section, the non occupancy refers to the time where the building is closed and not working.

The principle of the Dynamic call is illustrated in the next couple of figures. Figure 1 illustrates a call to OPT during the period of non-occupancy. In the given example, the CONTROLLER calls OPT at the time *t1*. The call of OPT (method *doOptimizing*) has two arguments T_n and T_return. The argument T_n represents the number of forecast intervals (i.e.: t1, t2, t3, …) required for the optimization to look ahead. In the given example the value of T_n is *n*. The second argument required by the OPT is T_return, which is the moment in time when OPT should finish and OPT returns a solution. The argument T_return determines the amount of time that OPT has available to compute and when T_now (time now) reaches T_return OPT should return the optimal values found so far. In the given example, T_return is *tx*, which corresponds to the beginning of the building occupancy period. Therefore, the time slot for OPT to compute the optimal values for the actuators is represented in green and in yellow is represented the forecast horizon to look ahead. When T_now reaches *tx*, OPT stops the PSO and return an array with the best set of control settings for every timestamp in the forecast horizon (i.e.: u_{t2}, u_{t3}, u_{t4}, …, u_{tx}, …, u_{tn}).

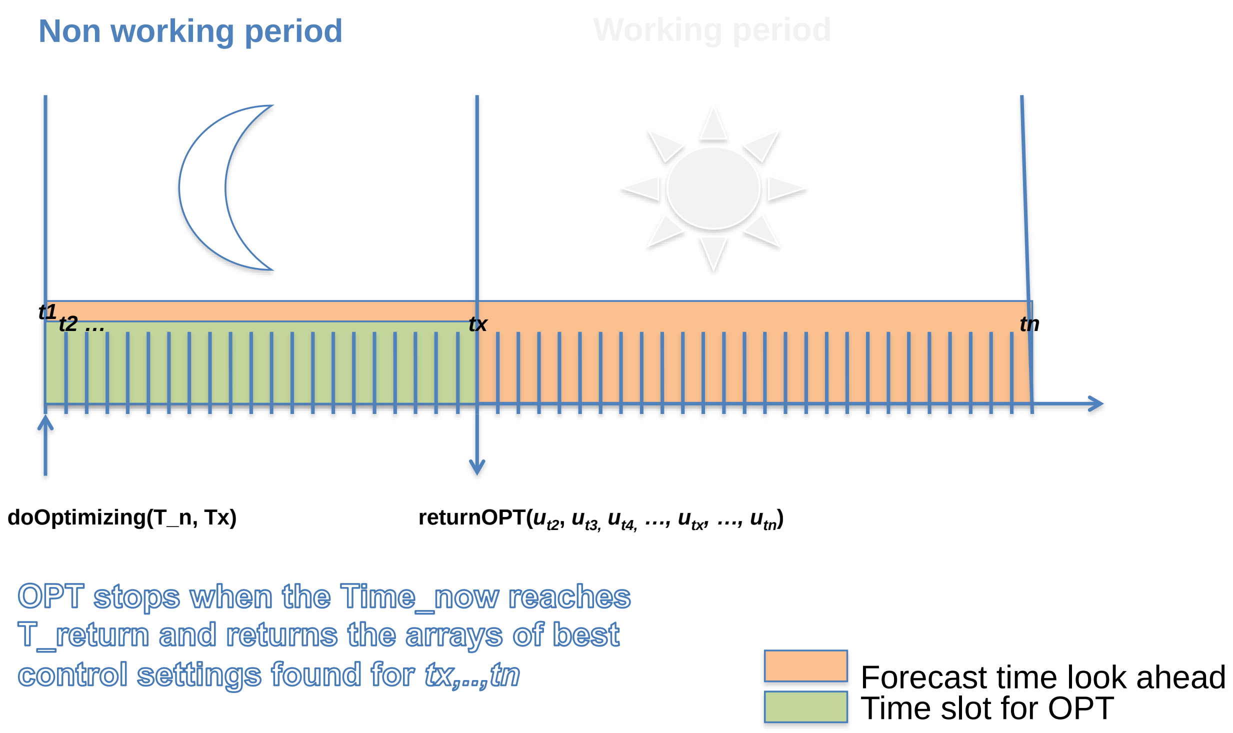

Non working period
Working period
t1
t2 …
tx
tn
doOptimizing(T_n, Tx)
returnOPT(u_{t2}, u_{t3}, u_{t4}, …, u_{tx}, …, u_{tn})
OPT stops when the Time_now reaches T_return and returns the arrays of best control settings found for tx,…,tn
Forecast time look ahead
Time slot for OPT

Figure 1. Example of Dynamic OPT call during the Non occupancy period.

Figure 2 illustrates a call to OPT during the period of occupancy. In the given example, when T_now reaches the moment *tx*, the CONTROLLER calls OPT assigning the number of forecast intervals as 3 and the return time as *tz*. When T_now reaches the moment *tz*, OPT stops the PSO and return an array with the best set of control settings for the 3 timestamps in the forecast horizon (i.e.: NEWu_{tz},…). In this case, the available time slot for OPT to compute can be very limited, however the algorithm uses the initialization strategies presented in 2.1.1 to accelerate the convergence. Since these strategies take into account the optimal results previously calculated, the optimum found during the occupancy of the building is expected to be a refinement from the solution determined previously in the non-occupancy period. This expectation is justified by the fact that the new PSO output is based upon more accurate predictions from SL. The prediction of SL has tendency to degrade when the forecast is too far from the present moment in time due to the fact that SL depends on the weather forecasts.

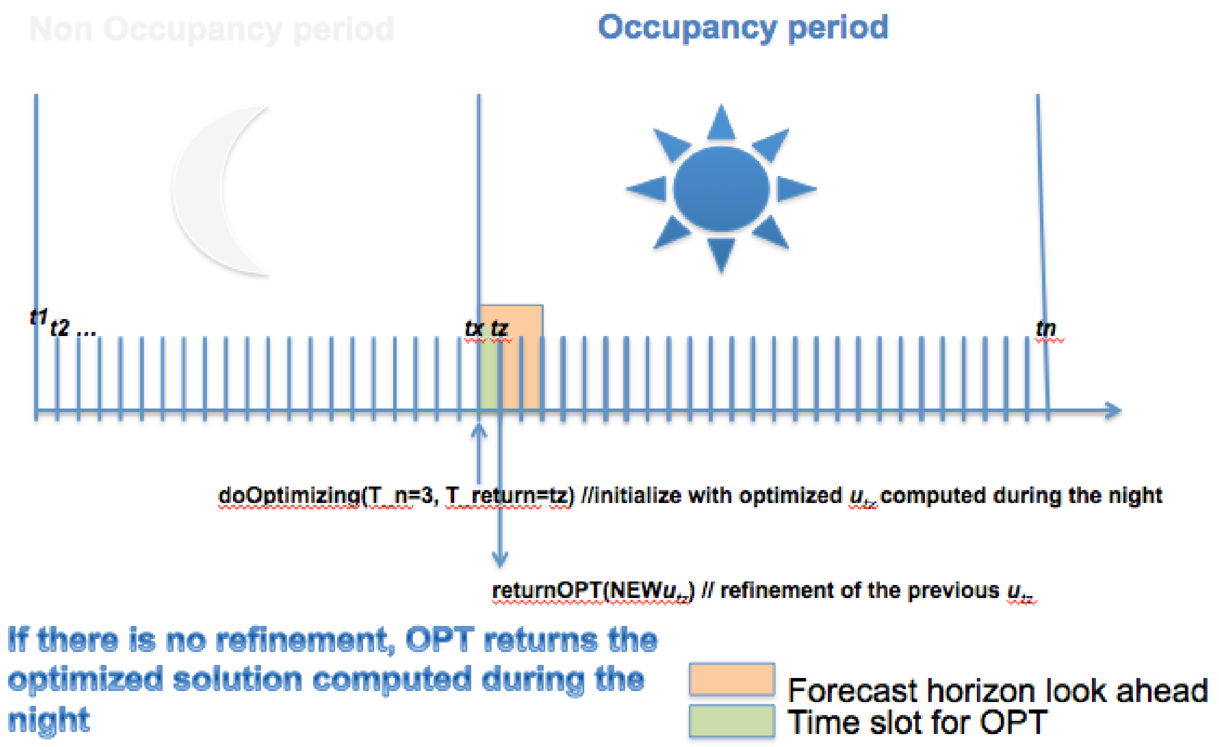

Non Occupancy period
Occupancy period
t1
t2 ...
tx tz
tn
doOptimizing(T_n=3, T_return=tz) //initialize with optimized u_{tz} computed during the night
returnOPT(NEWu_{tz}) // refinement of the previous u_{tz}
If there is no refinement, OPT returns the optimized solution computed during the night
Forecast horizon look ahead
Time slot for OPT

Figure 2. Example of Dynamic OPT call during the Occupancy period.

The setting of the regular occupancy period and the non-occupancy period varies with the building. The beginning of the period of occupancy should be defined some moments before the building start to be used. This way, the building can meet the comfort step when the users of the building arrive. In the case of the University of Stavanger the regular occupancy period corresponds to the interval in between 7:00 and 20:00 every working day. The non-occupancy period correspond to the interval in between 20:00 and 7:00 of every working day and includes the weekends and holidays. Obviously a building that has a different propose should have a different definition of these periods. The definition of the occupancy periods is set in the database and SEEDS OPT automatically adapt to the new definition. This allows the SEEDS system to be easily implemented in different types of buildings.

In the demonstrators of Madrid and Stavanger the CONTROLLER calls OPT during the occupancy periods with a periodicity of 10 min and with the total forecast horizon of one hour. During the non-occupancy periods the CONTROLLER calls OPT with a forecast horizon of 24 hours and with a time slot of more than 6 hours.

During the periods of non-occupancy since no person is expected to be in the building, meeting comfort is not a priority of SEEDS. Therefore, during the non-occupancy period the values sent to the actuators are the optimal values calculated during the previous non-occupancy period. This happens because during the non-occupancy the OPT is using all the available computational power to calculate the optimal values for the next 24 hours. The present solution is the best trade-off found to allow SEEDS algorithm to run in inexpensive commodity computer hardware and avoid the high costs of supercomputing, servers or computer clusters.

The combination of the Dynamic OPT strategy with the Smart Initialization of the PSO particles allows together to solve a complex optimization problem by dynamically adjusting the OPT to meet the limited availability of modest computational resources.

2.1.3 Dynamic restrictions of the search space

The building manager can change the range of values for the actuators in the Archive. This change can be a reduction of the range of the actuators, dynamically set according to the season and the characteristics of the equipment. OPT is prepared to seamlessly adapt to a change of a certain equipment as long as the range is changed in the Archive. Therefore, if the building manager finds it appropriated, he/she can reduce the range of values for each actuator in the Archive according to the season. OPT dynamically adopts the new range and therefore the size of the search space and the complexity of the optimization problem is reduced. If the building manager does not have the seasonal information, then it should set the range of actuators as provided by the manufacture of the equipment.

For the demonstrator of Stavanger the range of some actuators was defined according to this strategy. For the sake of simplicity the same values are used during the whole year. The values are presented in the next table.

Table 1. Range for the different actuators (Stavanger)

Actuator	Range
Air supply temperature in AHU	[16,24]
Volumetric flow values	[30,100]
Electric Radiators	{0,1}
Indoor light control	{0,1}

2.2 *Optimization of different types of energy*

SEEDS main aim is to optimize the energy consumption maintaining user comfort. Energy in a building can be supplied by different sources. . For example, the SEED demos use energy from different types: electrical, thermal and fuel. The administrator of the building may want to favour the optimization of the energy consumption of one type in detriment of another type. The criterion for weighting different energy types can be as simple as financial, environmental or it can be a complex combination of different criterions in different proportions. Criterions may be altered to follow politics, national strategies and new findings in science. For these reasons it is an advantage that OPT deals with the possibility of optimizing energies from different sources according to a criterion defined by the building administrator. To meet this requirement the OPT cost function presented in D5.3 (Esteves, 2013) was slightly modified. The modification is related with the calculation of ê and it is presented in the equation 1.

$$\hat{e} = \hat{e}_{\alpha} \times c_{\alpha} + \hat{e}_{\beta} \times c_{\beta} + \cdots + \hat{e}_{\gamma} \times c_{\gamma}$$
(eq. 1)

where:

$\hat{e}_{\alpha}, \hat{e}_{\beta}, \hat{e}_{\gamma}$ are 3 different energy types

$c_\alpha, c_\beta, c_\gamma$ are the weights for the corresponding energy types

The cost function is presented in the equation 2.

$$\alpha(\hat{e}, \hat{x}) = \begin{cases} \hat{e}, wl \leq \hat{x} \leq wh \\ f, \hat{x} < wl \vee \hat{x} > wh \end{cases} \quad \text{(eq. 2)}$$

where:

α is the cost function

ê is the predicted energy consumption for a candidate control settings

$\hat{x}$ is the predicted comfort level

wl is the lower comfort boundary

wh is the higher comfort boundary

f is the penalty score as defined in D5.3

In SEEDS the building administrator defines the weights of each different type of energy in the database. The building administrator can define different values of a certain weight to different moments in time (e.g.: a different weight for the energy type A for certain hours of the day). OPT is set by default with all weights with value one.

For the Stavanger pilot OPT uses the default weight values, whilst for the Madrid pilot the weights are defined in the table 2. The coefficient used in Table 2 favour chemical energy (natural gas) versus electricity because of the lower primary energy and CO_2 emissions involved. Since hourly coefficients for primary energy or CO_2 have not been found, the coefficients in the table are based on cost which is closely related to the above parameters.

Table 2. Energy Weights for Madrid demo[2]

Summer	Winter	Energy Weights
11:00 - 15:00	18:00 – 22:00	$\hat{e} = \hat{e}_\alpha \times 3.89 + \hat{e}_\beta \times 1$
00:00 – 08:00	00:00 – 08:00	$\hat{e} = \hat{e}_\alpha \times 1.77 + \hat{e}_\beta \times 1$
Rest of the day	Rest of the day	$\hat{e} = \hat{e}_\alpha \times 3.04 + \hat{e}_\beta \times 1$

ê – total predicted energy; $\hat{e}_\alpha$ – predicted electrical energy; $\hat{e}_\beta$ – predicted chemical energy

2.3 OPT algorithm

This section starts by presenting a description of some values and variables relevant for the OPT bundle. It presents the implemented version of the OPT algorithm in pseudo-code. It does not always follow objected oriented convention in order to be understandable for non-programmers.

[2] Explanations for the values in this table are presented in the D8.1 (Jimenez et al., 2012).

Some explanatory comments are shown in green font. For the sake of simplicity, the description does not elaborate on methods that are specific for controlling the PSO.
In order to be clearer for the reader, the algorithm for optimization of the Temperature and Air Quality is presented separately from the Illuminance. A call to OPT runs the optimization of Temperature and Air Quality and the optimization of the Illuminance.

2.3.1 Temperature and Air Quality optimization

The OPT algorithm starts by fetching data from the database. The fetch data is a time series where the first timestamp corresponds to the moment when OPT is called. The following timestamps corresponds to the time steps required by optimization to look ahead. Examples of these time series are the boundaries for occupancy and no occupancy, comfort set points, status of the actuators, and environmental variables. The values correspondent to the timestamps in the present is read for the tables “real” existing in the database. The values for the timestamps that take place in the future are read from the tables “calculated”. Then the OPT proceeds with the initialization of the particles that constitutes the PSO swarm. Details about the initialization procedure can be found in the section 2.1.1 “Initialization of the PSO”. The PSO loop runs the first iteration. In each iteration of the loop, the PSO computes the fitness of the particles that constitutes the swarm. As explained earlier, in the SEEDS context a particle is a multi-dimensional vector where each value corresponds to the status of an actuator. The fitness is the value of the cost function and a lower value means a fitter particle. For computing the cost function of a particle, it is necessary that OPT has predicted values from SL for all the time steps required by optimization to look ahead. To get the predicted values it is necessary to chain the calls to SL for those time steps in a loop. In this loop, the output of the SL for a given step is the input for the SL in the next step. In each step of the chained loop, the output of the SL is also checked if it meets the comfort requirements for that time step. If the output of SL fails to meet the required comfort requirement for a certain time step, this chained loop terminates immediately and the particle is automatically considered unfit. It is worth mentioning that this chained loop has to be re-computed for every particle and for every iteration of the PSO. After computing the SL for the last time step (unless the particle was not prematurely rejected as unfit) the fitness for that particle is given by computing the cost function. After the fitness been evaluated for all the particles of the swarm, the fitness is used for computing the position of the particles for the next PSO iteration. At this point the first iteration of the PSO is over and the PSO loops until the time slot defined by the CONTROLLER is over. When the time slot is over, the PSO loop terminates and the coordinates of the best particle are the optimal control settings for temperature and air quality. OPT calls the method for optimize the illuminance and the results are added to the optimal control settings for Temperature and Air Quality. OPT returns the optimal values to the CONTROLLER and the call to OPT finishes. The CONTROLLER writes in the database the optimal values.

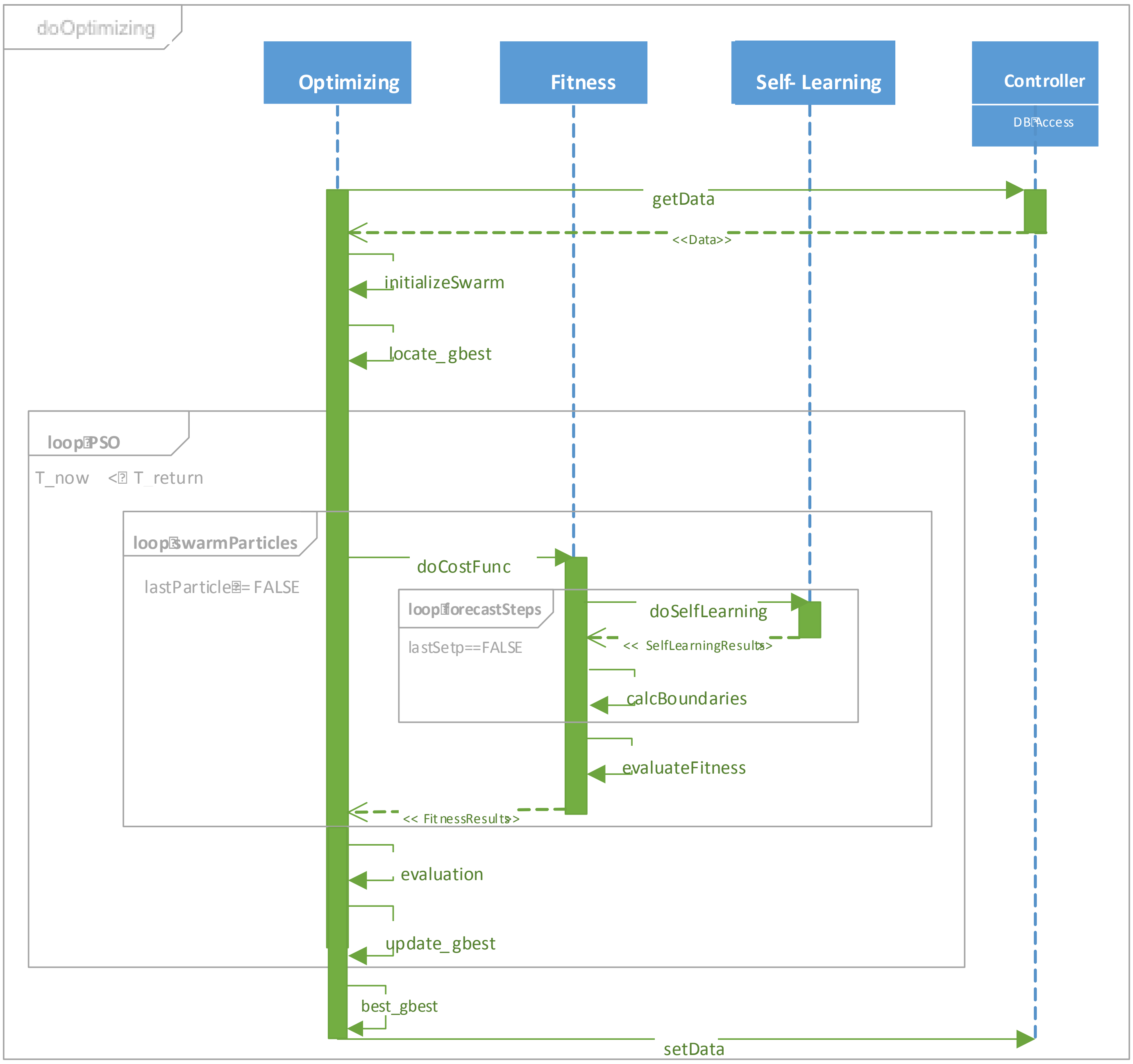


Figure 3. Optimization sequence diagram for SEEDS

Without being exhaustive the main methods used by OPT are presented in pseudo-code in the next pages, allowing a clear understanding of some details of the algorithm.

2.3.2 Pseudo code

2.3.2.1 Necessary control settings (UiS Pilot)

Electric Radiators

Uses java identifier Elrad_onoff.java

Possible values (0,1)

Volumetric flow

Uses java identifier Svfc.java

Possible values [30,100] **double**

Air supply temperature in AHU

Uses java identifier Sp_supair_temp.java

Possible values [16,24] **double**

2.3.2.2 Values to read from the database (UiS Pilot)

Room Temperature boundaries for no occupancy

(setup by the system administrator)

Upper and lower Temperature boundaries

In the following pseudo-code we use upper_temp_empty and lower_temp_empty

Variable type double

Room Temperature boundaries when occupancy

(setup by the system administrator to avoid malfunction by the user)

Upper and lower Temperature boundaries

In the following pseudo-code we use upper_temp_occu and lower_temp_occu

Variable type double

Room Temperature

The temperature read by the sensor

Uses java identifier In_temp.java

Variable type double

Room CO2 boundaries for no occupancy

(setup by the system administrator)

Upper CO2 boundary

In the following pseudo-code we use upper_CO2_empty

Variable type double

Room CO2 boundaries for occupancy

(setup by the system administrator)

Upper CO2 boundary

In the following pseudo-code we use upper_CO2_occu

Variable type double

Room CO2

The CO2 level read by the sensor

Uses java identifier In_airq.java

Variable type double

Room Occupancy

Provides info if the room is occupied or not for the moment Tx

Uses java identifier Movement.java

Variable type Boolean

It should use the info from the movement sensor or read the occupancy tables according to the moment in time

Temp User Comfort

Temperature increment/ decrement set by the user

Uses java identifier User_comf_level.java

Variable type double

Control Settings (CS)

Previously described in Control Settings

2.3.2.3 Arguments for calling OPT

T_n - the number of time steps required by optimization to look ahead

T_return – the time when OPT call should finish and OPT returns a solution. It determines the amount of time that OPT has available to compute and return results

2.3.2.4 Method doOptimizing

```
doOptimizing () {

    #CS_swarm is the set Control Settings (CS) for the several particles of the swarm.
    #CS_swarm = {CS for the particle1, particle 2, …};
    # CS = {Control Settings for time step 1, time step 2, …, last step}
    CS_swarm = smartInitPSO()

    SumEnerg = 0

    Time_begin = time_now() # Beginning time of the optimization call

    #Get the Boundaries defined by the BEM administrator for Occ and no Occ
    OB = getBEMboundariesDB()

    #Comfort  Set Point from DB (i.e.: desired Comfort defined by the user in the reality)
    SP_now = getComf_SP_DB()

    Occ_now = getOcc_DB(Time_begin) #Occupancy now

    #Compute PSO while there is still time left
    WHILE (time_now() < T_return) DO {

        FOREACH particle DO {

            Tx = Time_begin #Beginning time of the optimization call

            #Cost function evaluation for a candidate particle.
            C = doCostFunc(Tx, T_return, OB, CS)

            #Fitness of the several particles per iteration
            evaluateFitness(C)

            IF (time_now() > T_return) THEN
                BREAK FOREACH
        }

            updatePbest() #Updates the best CS found so far

            #Determines next set of CS to evaluate
            CS_swarm = updateParticlesPosition()
```

```
      }

     #Convert Continuous variables to discrete (only for the discrete actuators Electric Radiators
      Pbest = cont2disc(Pbest)

      #sets the value of the best CS found
      setDB(Pbest)

      RETURN(Pbest)
}
```

2.3.2.5 Computation of the Cost Function

```
#To be easier to understand, some variables are not explicitly defined as arguments. If some
variables are not defined here, it is considered to be implicitly passed from the calling function
doCostFunc (Time Tx, Time T_return, BEMboundaries OB, ControlSettings CS) {

        Ti= Tx

        #Loop of calls to SL chained for every time steps of the forecast horizon
        WHILE (Ti  < T_return) DO {

            P = doSelfLearning(Ti, CS.Ti) #Predicts Temp,CO2,E,Occ,SP

            #If Ti is now then use the SP and Occ from DB
            IF (Ti == Tx) THEN
                B = calcBoundaries(SP_NOW, Occ_Now, OB)

            #ELSE use the SP and Occ from SL
            ELSE
                B = calcBoundaries(Ti, P.SP, P_Occ, OB)

            #Sums all the different energy types and apply coefficients (a,b,etc) to weight the
energies. Note: by now a and b are 1
                P.E = a * P.E.Ele + b*P.E.Chemic + …

            #Checks if Comfort is met
            IF ((B.lower_temp<P.Temp< B.UPPER_TEMP_USER)
                AND (P.CO2< B.UPPER_CO2_USER))
                                THEN SumEnerg=SumEnerg+P.E
           ELSE SumEnerg=SumEnerg+doPenalty()

           Ti=Ti+deltaT
        }

        RETURN(SumEnerg)
}
```

2.3.2.6 Method to get the Boundaries for occupancy and no occupancy from DB

```
#Boundaries defined by the BEM administrator

getBEMboundariesDB (Time T) {

    GET from DB for all rooms for the time T

        {

            Room Temperature boundaries for no occupancy
            Room Temperature boundaries when occupancy
            Room CO2 boundaries for no occupancy
            Room CO2 boundaries for occupancy

        }

        RETURN boundaries

}
```

2.3.2.7 Method to get Comfort_SP from DB for the present moment

```
getComf_SP_DB (Time Tnow) {

    GET from DB for all rooms for the time Tnow

        {

            Room User_comf_level + In_temp
            Room In_airq

        }

        RETURN SP

}
```

2.3.2.8 Method to compute Comfort Boundaries

```
calcBoundaries (Comfort_SP SP, Occupancy Occ, OccBoundary OB) {

    DECLARE B #Structure that will contain the comfort Boundaries
    IF (Occ  == TRUE)  {

        IF (SP. Temp + 1 < OB.upper_temp_occu) THEN
            B.upper_temp= SP.Temp + 1
        ELSE
            B.upper_temp= OB.upper_temp_occu
        IF (SP.Temp - 1 > OB.lower_temp_occu) THEN
            B.lower_temp= SP.Temp - 1
        ELSE
            B.lower_temp= OB.lower_temp_occu
        B.upper_CO2_user = B.upper_CO2_occu
    }
    ELSE {

        B.upper_temp= OB.upper_temp_empty

        B.lower_temp= OB.lower_temp_empty

        B.upper_CO2= OB.upper_CO2_empty

    }

    RETURN(B)

}
```

2.3.2.9 *Method for the PSO initialization*

# Set initial values of the PSO particles. It helps the PSO to converge faster to the optimum. It also guaranties that OPT will return the previous optimized Control Settings if no better solution is found.

```
smartInitPSO() {

    #T_last_step = Time that last step shall start (T_return – deltaT)

    CS = GET from DB the Control Settings for the time (Tnow) … (T_last_step)

    #If any of CS values does not exist in the DB, use a value for the equivalent time from the
    previous week
    IF (CS == EMPTY) THEN
        CS = GET from DB for T (Tnow – 1week)…(T_last_step - 1week)

    Instantiate one particle with CS values
    Instantiate one particle with MAX CS possible values
    Instantiate one particle with MIN CS possible values
    Instantiate the other available particles randomly
```

```
        #set other required parameters for PSO
        setParamPSO()

        #CS_Swarm are the initial control settings for the several particles of a SWARM
        RETURN (CS_swarm)
}
```

2.3.2.10 Method to set other required parameters for PSO

```
setParamPSO() {

        Number of particles = 10 #should be posterior re-adjusted according to the results

        Local exploration constant= 0.5+log(2) #should be posterior re-adjusted according to the results

        Global exploration constant= 0.5+log(2) #should be posterior re-adjusted according to the results

        Number of particles informants= 3 #should be posterior re-adjusted according to the results

        Maximum velocity not clamped #should be posterior re-adjusted according to the results

        Number of iterations = the maximum amount that can be run during the OPT call

}
```

2.3.3 Illuminance comfort optimization

The control of illuminance is only present in Stavanger pilot. The method for optimizing the illuminance is called by OPT. This call occurs after computation of the optimal control settings for the Temperature and Air quality.

2.3.3.1 Necessary control settings for illuminance

Indoor light control
Uses java identifier Inlight_onoff.java
Possible values (0,1)

2.3.3.2 Values to read from the database

Room Last Occupancy
Provides the timestamp when the last movement was observed in the room
In the following pseudo-code we use Last_Occ
Variable type double
The value is obtained through a query to the DB

Room Light level
Luminance inside of a room

Uses java identifier Lighting.java
Variable type double

Room Light boundaries

(Max and minimum values of light indoor when the room has occupancy
(setup by the system administrator)
In the following pseudo-code we use LUX_MAX and LUX_MIN
Variable type double

Indoor light control

Uses java identifier Inlight_onoff.java
Possible values (0,1)

2.3.3.3 *Method for optimization of the illuminance*

```
doLightOPT (Time T) {
    GET from DB for all rooms for the time T {
        LUX_MAX
        LUX_MIN
        Inlight_onoff
    }
    QUERY DB about Last_Occ for all rooms

    FOREACH room {
        IF ((Inlight_onoff ==ON) {
            Delta_T = time_now() – Last_Occ    # Delta_T is the time interval between
        the last observed user in the room and the present time

            If (Delta_T < TMD )  # TMD is a Maximum time interval threshold for a
        room to be occupied
                Occ= TRUE
            ELSE Occ = FALSE

            IF (Occ==FALSE))
                Inlight_onoff =OFF
            IF (Occ==TRUE) AND (lighting>LUX_MAX+200))
            Inlight_onoff =OFF
            }
    }
    RETURN(Inlight_onoff)
}
```

3 Prediction of User and Process Behaviour and Energy Consumption

3.1 Architecture of the Self-Learning components

Figure 4 presents the main sub-components of the Self-learning component. The Self-learning component, like Controller, adopts the Mediator design pattern (Gamma, et al. 1995). The process of forecasting is handled by the SelfLearingForecast bundle and the process of training is handled by the SelfLearningTraining bundle. The interfaces ISelfLearningForecast and ISelfLearningTraining are described in detail in Sections 3.3 and 3.4, respectively.

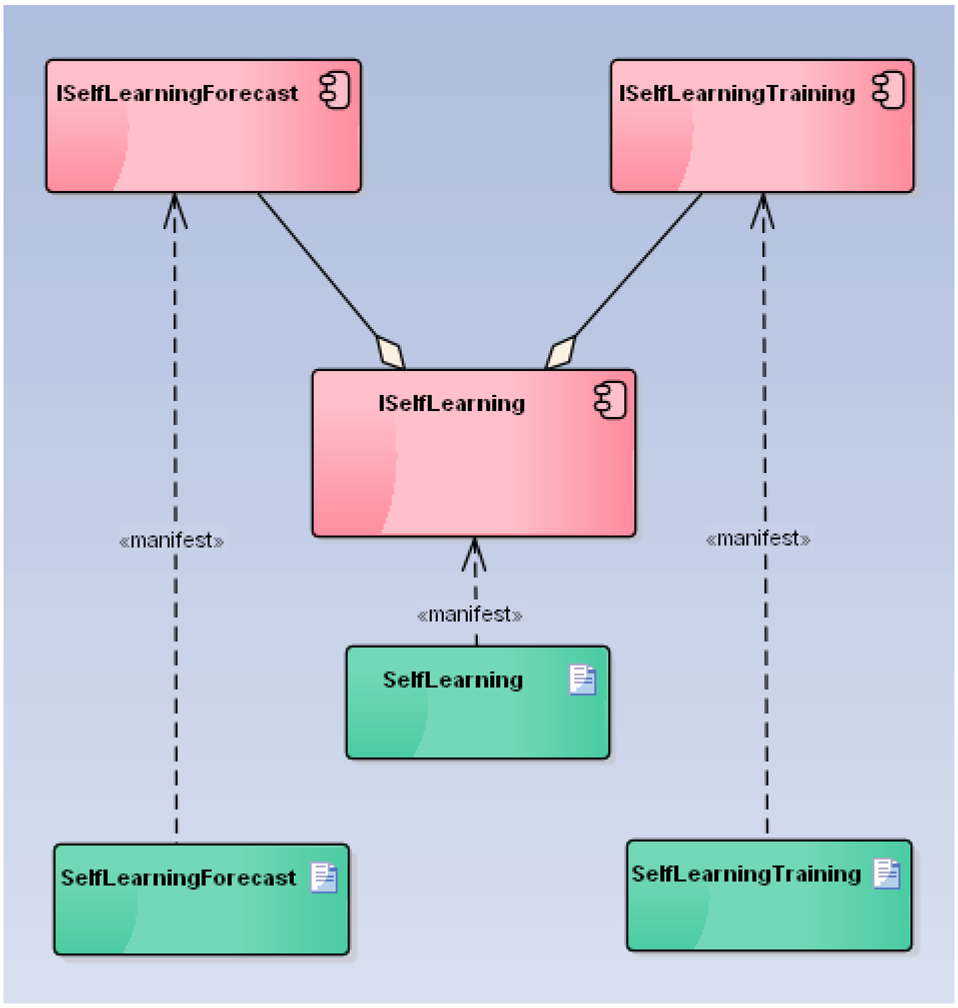


Figure 4. Component diagram for SelfLearning

3.2 Configuration

The Self-learning component is configurable. This eases the configuration for both the UiS and Madrid demonstrators. Self-learning has been developed in a modular manner whereby its configuration is modularized into a single bundle. This enables easy adaption and configuration. For example, one only needs to change the variables used in Self-learning in a single place: the configuration bundle.

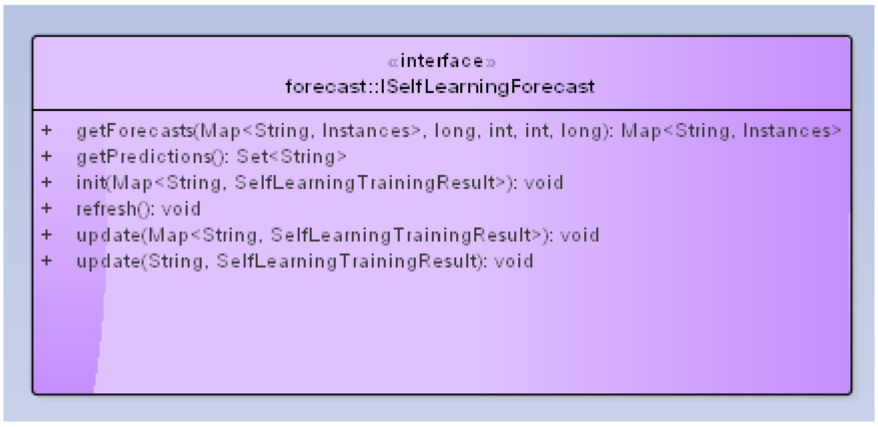
In order to configure the Self-learning bundle for a particular building, the input and output variables must be specified. Details of the variables can be found in Deliverable 2.9 (Donath, Wurm, et al. 2014). These are dependent on factors such as the number of rooms and the type of equipment in use. The output variables that are forecast by Self-learning for the UiS and Madrid demonstrators include the following:

- Set point of indoor air temperature
- Indoor air temperature
- Indoor air quality level
- Lighting level
- Room use (occupied or unoccupied)
- Electrical energy required
- Chemical energy required
- Thermal energy provided
- Luminous energy provided.

Self-learning is also generic in that a range of Self-learning methods can be used. The Self-learning component relies on the Weka framework for the implementation of the various methods. The configuration specifies the Self-learning method used and the parameters associated with that method. For example, in the UiS and Madrid demonstrators, feed-forward neural networks were used. However, other methods such as decision tree learning and Bayesian networks could also be selected.

3.3 Forecasting

Self-learning is configured so that each output variable described above has a corresponding model. The ISelfLearningForecast interface (Figure 5) is implemented by the SelfLearningForecast bundle. A model is encapsulated in the SelfLearningTrainingResult class. This class is described in Section 3.4.



```
«interface»
forecast::ISelfLearningForecast

+ getForecasts(Map<String, Instances>, long, int, int, long): Map<String, Instances>
+ getPredictions(): Set<String>
+ init(Map<String, SelfLearningTrainingResult>): void
+ refresh(): void
+ update(Map<String, SelfLearningTrainingResult>): void
+ update(String, SelfLearningTrainingResult): void
```

Figure 5. ISelfLearningForecast interface

An implementation of ISelfLearningTraining can return a model that has been trained (Section 3.4). The interface contains methods to initialise the models (Table 3), to update a model(s) (Table 4, Table 5) and to return forecasts based on the models that are currently in use (Table 9).

Table 3. init method

public void init(Map<String, SelfLearningTrainingResult> trainingResults)	
Initialise one or more models.	
trainingResults:	A mapping from an identifier to the model used to make a forecast for that identifier.

Table 4. update method

public void update(Map<String, SelfLearningTrainingResult> trainingResults)	
Update one or more models.	
trainingResults:	A mapping from an identifier to the model used to make a forecast for that identifier.

Table 5. update method

public void update(String identifier, SelfLearningTrainingResult trainingResult)	
Update the model for an identifier.	
identifier: trainingResult:	An identifier. A model used to make forecasts for that identifier.

Table 6. refresh method.

public void refresh()
Use the model that have recently been updated

Table 7. getPredictions method.

public Set<String> getPredictions()	
The identifiers about which forecasts can be made.	
returns:	The identifiers about which forecasts can be made.

Table 8. getForecasts method.

public void getForecasts(Map<String, Intances> controlSettings, long actualTime, int cycleNumber, int stepNumber, long stepTime)	
Return forecasts based on the control setting, optimisation cycle for a given step.	
controlSettings:	The control settings.
actualTime:	The actual timestamp of the Optimization cycle.
cycleNumber:	The number of the Optimization cycle.
stepNumber:	The number of the step within an Optimisation cycle.
stepTime:	The timestamp of the step within an Optimisation cycle.

Consider Figure 6. ISelfLearning can get the latest models from ISelfLearningTraining by invoking the getTrainingResults method (Section 3.4). ISelfLearningForecast separates the updating of a model from its subsequent use to make forecasts. This is achieved by calling the update and refresh methods, respectively.

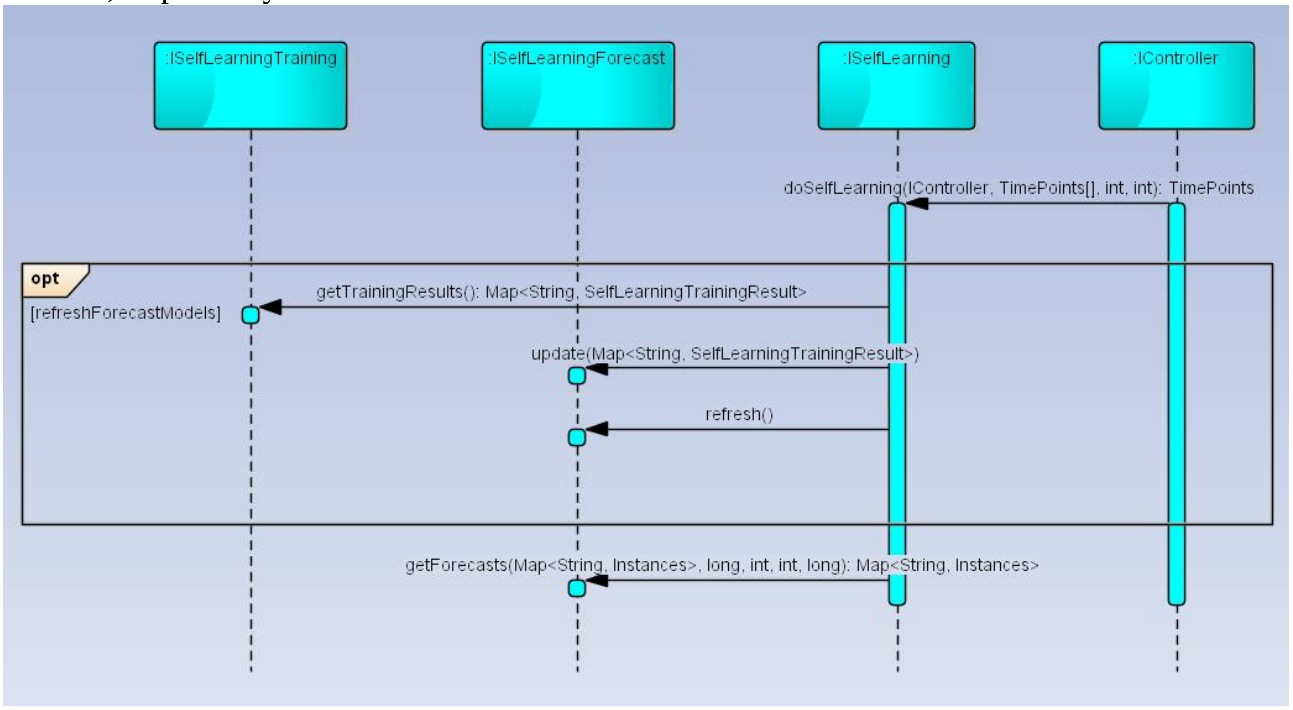

:ISelfLearningTraining
:ISelfLearningForecast
:ISelfLearning
:IController
doSelfLearning(IController, TimePoints[], int, int): TimePoints
opt
[refreshForecastModels]
getTrainingResults(): Map<String, SelfLearningTrainingResult>
update(Map<String, SelfLearningTrainingResult>)
refresh()
getForecasts(Map<String, Instances>, long, int, int, long): Map<String, Instances>

Figure 6. Updating of models for Forecasting

Self-learning must decide *when* to update forecasting. In Figure 6, this is represented by the guard condition refreshForecastModels. The strategy adopted by Self-learning is to use the same model for forecasts *within an Optimisation cycle*. However, a model can be updated *between Optimisation cycles*.

3.4 Training

The ISelfLearningTraining interface (Figure 7) is implemented by the SelfLearningTraining.

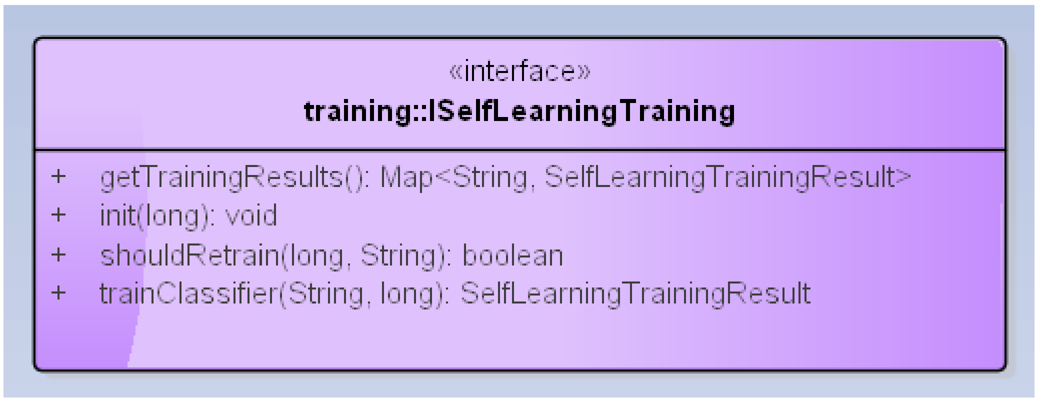

«interface»
training::ISelfLearningTraining
+ getTrainingResults(): Map<String, SelfLearningTrainingResult>
+ init(long): void
+ shouldRetrain(long, String): boolean
+ trainClassifier(String, long): SelfLearningTrainingResult

Figure 7. ISelfLearningTraining interface

Table 9. getTrainingResults method

public void getTrainingResults(): Map<String, SelfLearningTrainingResult> trainingResults
For each identifier, return the associated model that has been trained.
returns: A mapping from identifier to models.

In practice, the SelfLearningTraining bundle returns the *latest models* that have been trained. However, an alternative implementation of ISelfLearningTraining could use criterion such as ongoing performance as a basis for determining what should be returned.

SelfLearningTraining evaluates the performance of each model using the shouldRetrain method and can advise as to whether or not training is needed (Table 10). The forecasts made by a model play a role in the evaluation. The evaluation of the performance of a model will be discussed further in Section 3.4.1.

Table 10. shouldRetrain method

public boolean shouldRetrain(long trainingTime, String identifier)	
Advise whether or not training should occur.	
trainingTime:	The training time of the model under consideration.
identifier:	The identifier of the model under consideration.
return:	Whether training is either advised or not advised.

Table 11. getTrainingResults method

public void getTrainingResults(): Map<String, SelfLearningTrainingResult> trainingResults	
For each identifier, return the associated model.	
returns:	A mapping from identifier to models.

A model is encapsulated within the SelfLearningTrainingResult class (Figure 8). The model itself takes the form of a Classifier as implemented by Weka (Table 12).

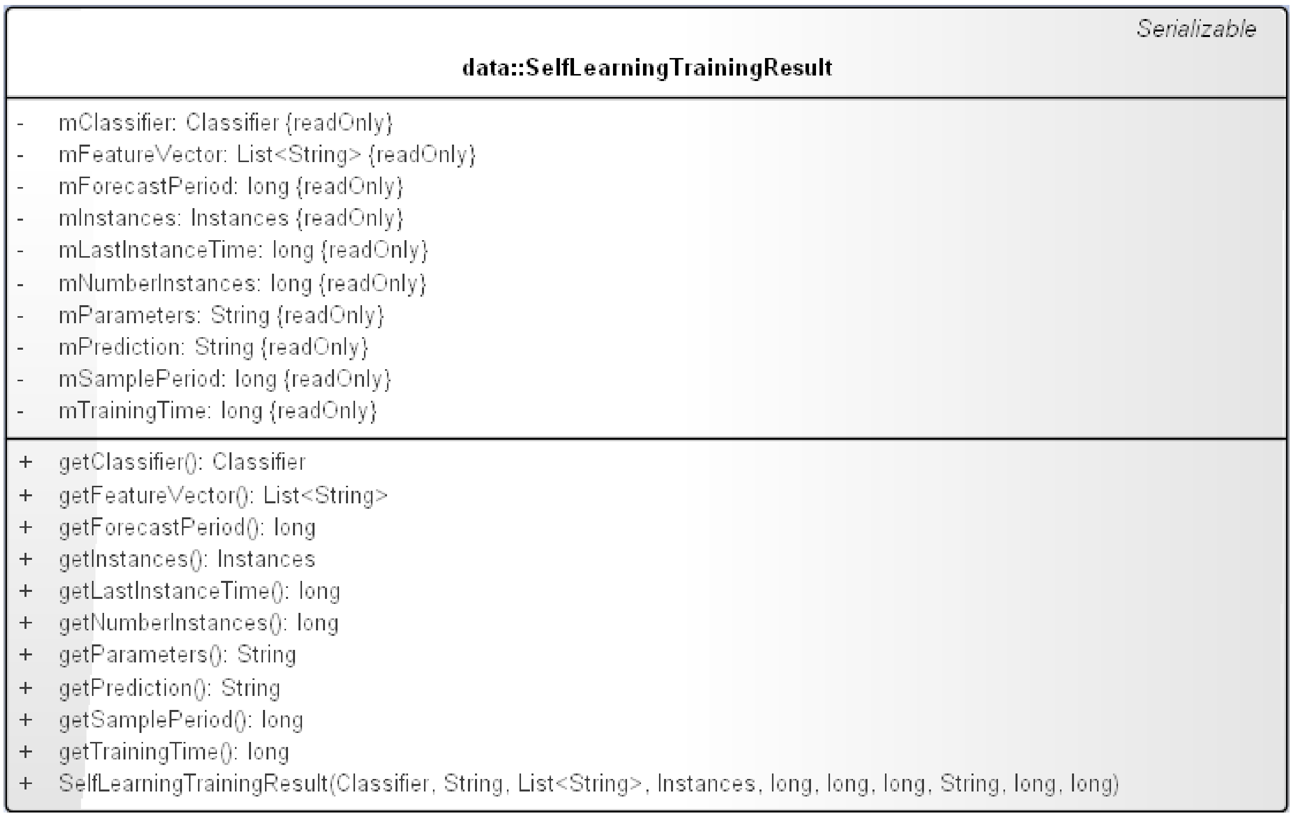

Serializable
data::SelfLearningTrainingResult
- mClassifier: Classifier {readOnly}
- mFeatureVector: List<String> {readOnly}
- mForecastPeriod: long {readOnly}
- mInstances: Instances {readOnly}
- mLastInstanceTime: long {readOnly}
- mNumberInstances: long {readOnly}
- mParameters: String {readOnly}
- mPrediction: String {readOnly}
- mSamplePeriod: long {readOnly}
- mTrainingTime: long {readOnly}
+ getClassifier(): Classifier
+ getFeatureVector(): List<String>
+ getForecastPeriod(): long
+ getInstances(): Instances
+ getLastInstanceTime(): long
+ getNumberInstances(): long
+ getParameters(): String
+ getPrediction(): String
+ getSamplePeriod(): long
+ getTrainingTime(): long
+ SelfLearningTrainingResult(Classifier, String, List<String>, Instances, long, long, long, String, long, long)

Figure 8. SelfLearningTrainingResult class

The SelfLearningTrainingResult class also includes information about the training data that was used to train the model (Table 12).

Table 12. Fields of the SelfLearningTrainingResult class

Field Name	Description
mClassifier:	The Weka model used to make forecasts.
mFeatureVector	The feature vector of the model, where the feature names are based on SEEDS identifiers.
mForecastPeriod	The forecast period of the model.
mInstances	The dataset that Weka uses to represent the features, include feature name and type.
mLastInstanceTime	The timestamp of the last instance used for training.
mNumberInstances	The number of instances in the training data.
mParameters	The parameters used for trained as defined by Weka and MOA
mPrediction	The SEEDS identifier that is forecast by the model.
mSamplePeriod	The time period between successive instances in the training data.
mTrainingTime	The timestamp when the model was trained.

Thus, given the data used to train the model, the information about training can be used to reproduce the model from scratch.

As discussed in Section 3.2, the choice of Self-learning method is stored in the configuration. In the UiS and Madrid demonstrators, feed-forward neural networks were used, but other methods such as decision tree learning and Bayesian networks could also be selected. The precise parameters can be found in Deliverable 5.2 (Vadera, et al. 2013).

The features used by a model are specified in a feature vector which specifies the input features and output feature of a model. Each feature can be categorized into three distinct categories: provided features, derived features and predicted features (Donath, et al. 2013) A provided feature reads a value directly from the Archive. A derived feature, in addition to reading a value from the Archive, also applies a filter. Finally, a predicted feature outputs the value of a forecast. This predicted feature is set as the class for the model.

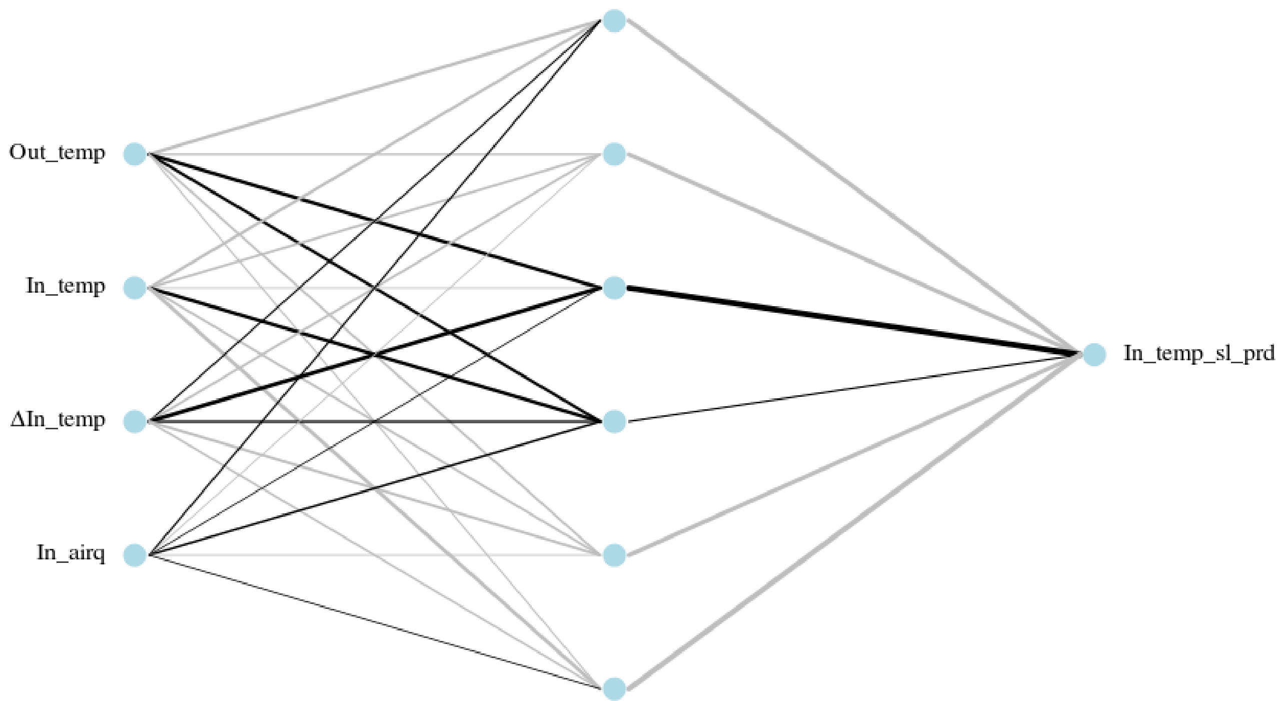


Figure 9. Feed-forward Neural Network for forecasting Indoor Temperature

Table 13 shows the features of a model. For the sake of clarity, for the feature names we use the class names that correspond to an identifier.

Table 13. Feature Vector for a model

Feature	Description	Feature Category
Out_temp	Outdoor air temperature from weather station	Provided
In_temp	Indoor air temperature (air temperature sensor)	Provided
ΔIn_temp	Change in Indoor air temperature (air temperature sensor) over a given time interval	Derived
In_airq	Indoor air quality level / Rooms CO2 concentration (CO2 sensor)	Provided
In_temp_sl_prd	Indoor air temperature predicted (air temperature sensor)	Predicted

3.4.1 Evaluating the performance of model

A key question for SelfLearning is at what point should a new model be produced? SelfLearningTraining becomes aware of the forecasts that have been made by querying the Archive. The shouldRetrain method uses this information to then make an informed decision based on the performance of the model to date.

In practice, the implementation of the shouldRetrain method by SelfLearningTrainings adopts a metric to continuously evaluate the accuracy of model. The metric compares the values of the forecasts made by a model with the actual values read from the sensors and stored in the SEEDS

Cache. As soon as the value of the metric deviates from a given threshold, then shouldRetrain method advises that training should occur. SelfLearning then follows this advice and initialises training (Figure 10).

As shown below, there are various standard metrics that can be adopted:

Mean absolute error (MAE)	$\sum \frac{\lvert predicted - actual \rvert}{N}$
Mean squared error (MSE)	$\sum \frac{(predicted - actual)^2}{N}$
Root mean squared error (RMSE)	$\sqrt{\sum \frac{(predicted - actual)^2}{N}}$
Mean absolute percentage error (MAPE)	$\sum \frac{\left\lvert \frac{(predicted - actual)}{actual} \right\rvert}{N}$

where *N* is the number of observations, *predicted* is the forecasted value and *actual* is the actual value. The metric adopted by Self-Learning is the Root Mean Squared Error (RMSE). In each case, only the forecast made in the first step of the multi-step forecast are considered.

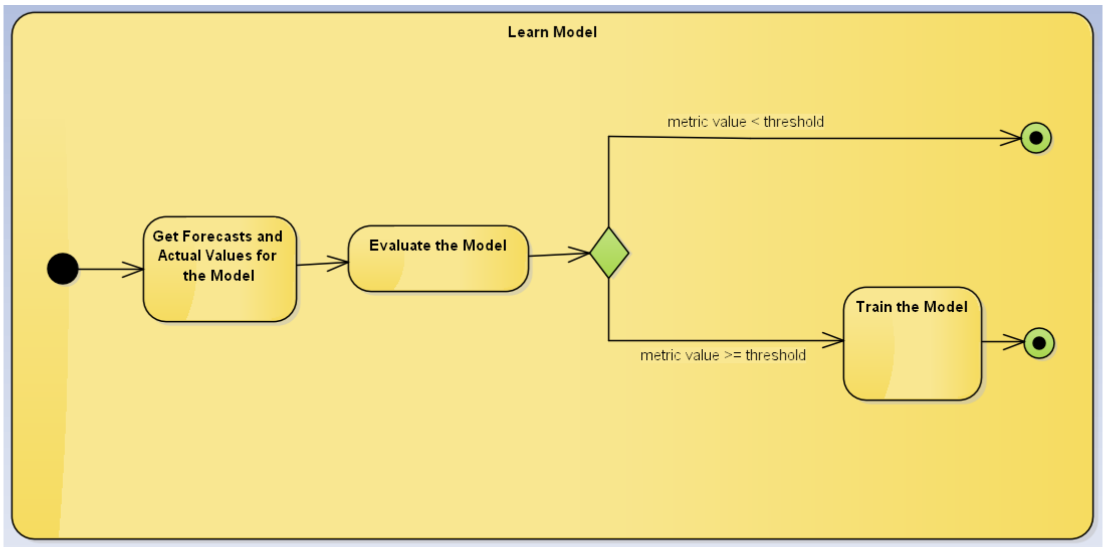

Learn Model
Get Forecasts and Actual Values for the Model
Evaluate the Model
metric value < threshold
metric value >= threshold
Train the Model

Figure 10. Evaluate the performance of a model.

3.4.2 Data Flow into a Model

A significant advance in SEEDS is that Self-learning can change a model on-the-fly at run-time. In particular, the features of a model can change. As discussed above, this occurs when a new model is produced. Self-learning uses the features of a model to determine the data that flows into that model.

Consider Figure 11. At start-up, the feature vector is initialized from a file on disk. This file is provided by a human. Subsequently, the feature vector is read from memory and it is provided by the SelfLearningTraining component itself.

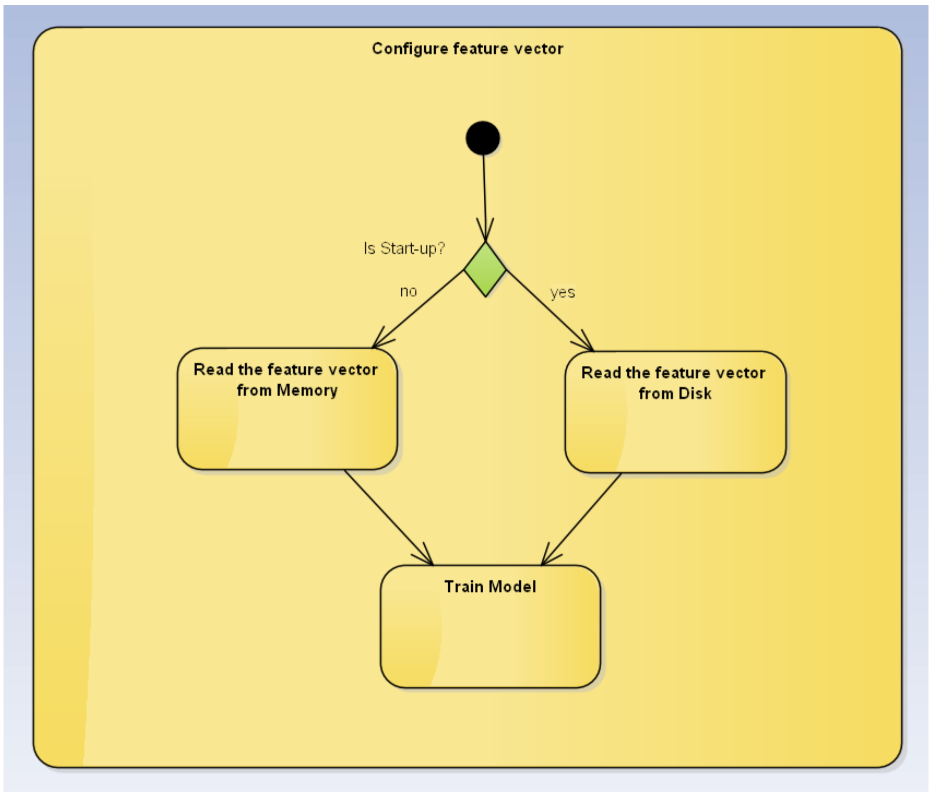

Configure feature vector
Is Start-up?
no
yes
Read the feature vector from Memory
Read the feature vector from Disk
Train Model

Figure 11. Configuration of a feature vector.

Table 13 and Table 14 show the features of a model, before and after a change has taken place. We omit the class feature as its values are dependent on whether the model is used for forecasting or training. As before, we use the class names that correspond to an identifier for the feature names.

Table 14. Model after change

Feature	Description	Feature Category
Out_temp	Outdoor air temperature from weather station	Provided
In_temp	Indoor air temperature (air temperature sensor)	Provided
ΔIn_temp	Change in Indoor air temperature (air temperature sensor) over a given time interval	Derived
In_airq	Indoor air quality level / Rooms CO2 concentration (CO2 sensor)	Provided
In_hum	Return air relative humidity (Humidity in the room) Indoor relative humidity (RH sensor)	Provided
In_temp_sl_prd	Indoor air temperature predicted (air temperature sensor)	Predicted

The data flow into a model is represented as a data stream. Table 15 and Table 16 show the data stream into the model before and after the change above. The model is updated at 10:30 and this change comes into effect immediately. We can see that the addition of the indoor humidity feature drives a corresponding change in the data flow into the model.

Table 15. Data stream before change

Time	Out_temp	In_temp	ΔIn_temp	In_airq
10:10	13	15	0	20
10:20	13	15.5	0.5	20

Table 16. Data stream after change

Time	Out_temp	In_temp	ΔIn_temp	In_airq	In_hum
10:30	13	15.5	0	21	71
10:40	13.1	15.5	0	21	71

Self-learning relies on the MOA (Massive Online Analysis) framework for the implementation of a data stream. Indeed, MOA and Weka are closely related. This is an advantage as MOA can seamlessly interface with the learning method provided by Weka. More information on the use of Weka with MOA can be found in the MOA documentation.

3.5 Performance improvement through Advanced ICT solutions

3.5.1 Caching

As identified in Deliverable 5.3 (Esteves 2013) , the time taken to make forecasts was considered a bottleneck for Optimisation in the early implementations of Self-Learning.

Although the time taken for each call was 0.4s, the number of calls made by Optimisation meant that this needed to be reduced. Hence, several solutions were considered; the most effect approach was to use caching. Its introduction has lead to a significant speed up.

A feature can be shared by more than model and between different steps in the multi-step forecast process. In the previous implementation, a feature was retrieved from the Archive multiple times and its value recomputed. However, with the introduction of a cache a feature is retrieved once from the Archive and its value computed.

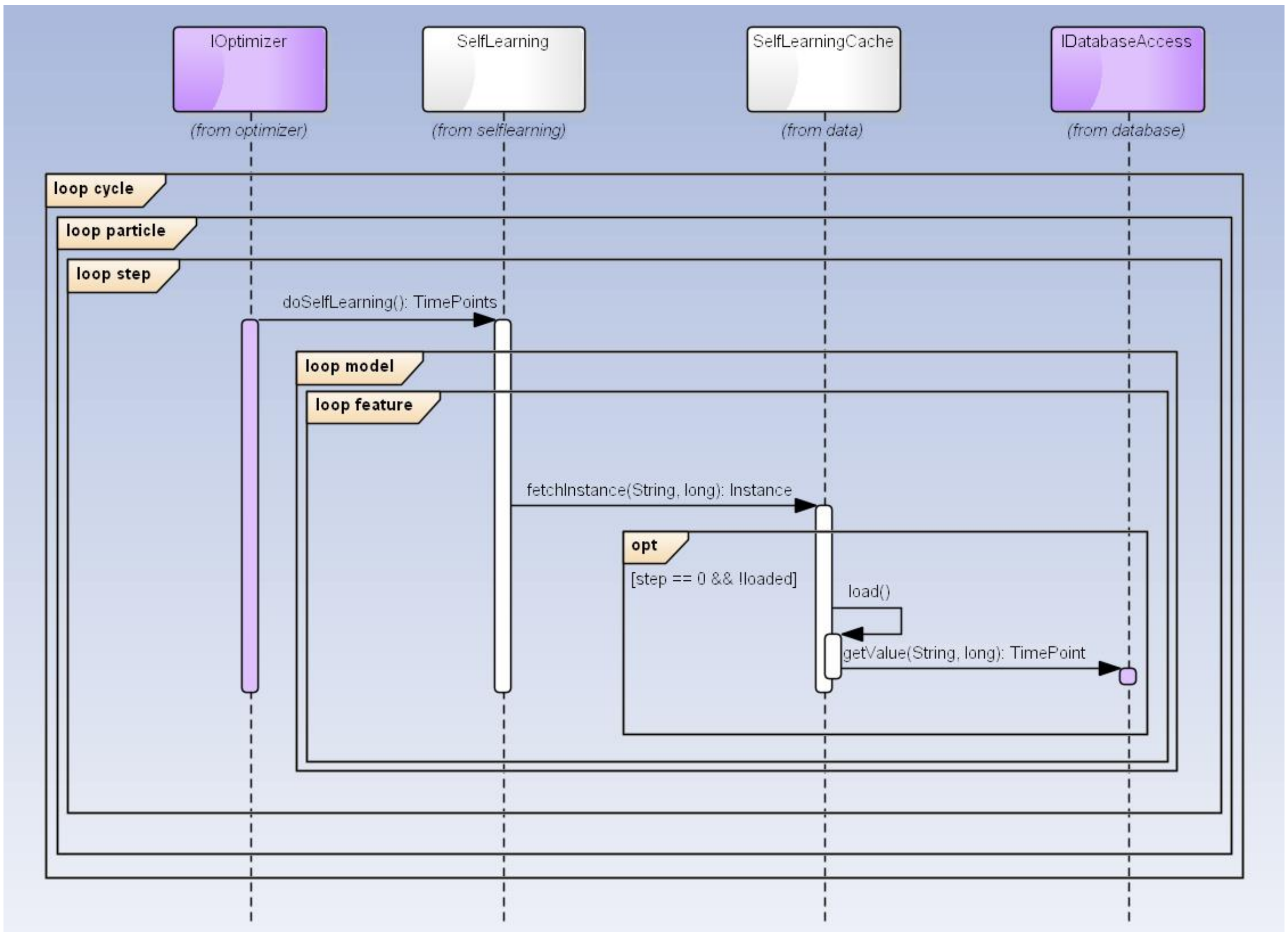


Figure 12. Example of Caching in SelfLearning

Figure 12 illustrates one such case. In the first step of a multi-step forecast, the value of a feature is taken from the Archive. If not already loaded, the cache will query the Archive and compute the feature. This occurs only once. Subsequent requests to fetch the feature will use the already loaded value and not query the Archive.

The SelfLearning bundle uses caching mechanism provided by the Google guava library. As shown in Figure 13, both SelfLearningTraining and SelfLearningForecast use this library. In particular, they use the class LoadingCache. This class returns the value associated with a feature in the cache, first loading that value if necessary.

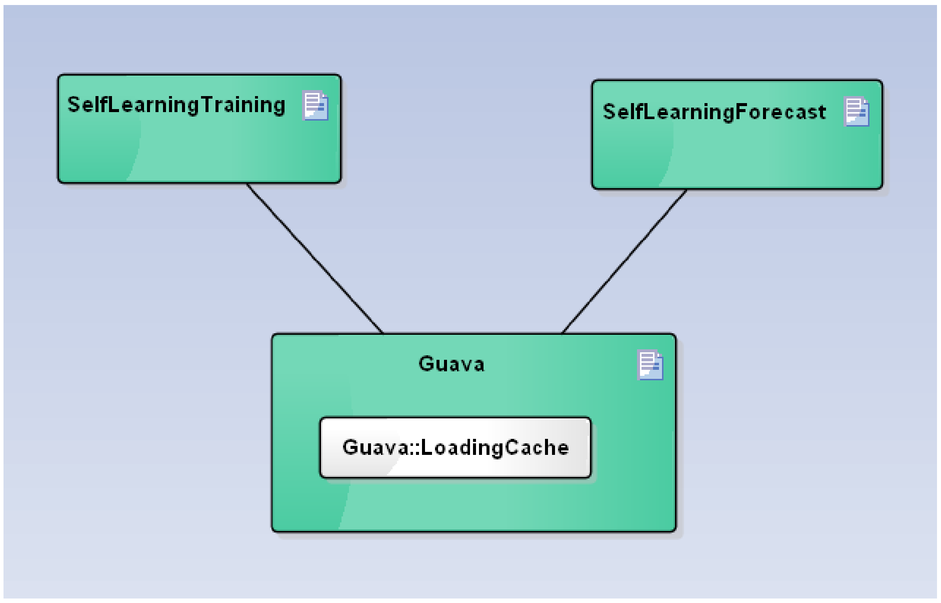


Figure 13. Component diagram for Caching in SelfLearning

3.5.2 Concurrency

There is scope for using concurrency in Self-learning. The models used by Self-learning are mostly independent. This fact can be used SelfLearningForecast and SelfLearningTraining, respectively. Self-learning uses the concurrency library provided by Java. SelfLearningForecast can make multiple forecasts in parallel and SelfLearningTraining can train multiple models in parallel. At run-time, a model can share features. However, the implementation of the LoadingCache class can handle multiple threads requesting the value for the same feature. The use of concurrency has resulted in a significant speed up for both the forecasting and training processes.

3.5.3 Deployment

SelfLearningTraining is not dependent on SelfLearningForecast. As such, SelfLearningTraining can be deployed on a separate instance of the Eclipse Equinox framework. This has the added advantage that competition between SelfLearningTraining and SelfLearningForecast for resources is reduced.
Figure 14 shows the deployment of the SelfLearningForecast and SelfLearningTraining components on separate instances of Eclipse Equinox. Both instances of Eclipse Equinox framework communicate by Remote Services provided by the Eclipse Communication Framework (ECF).

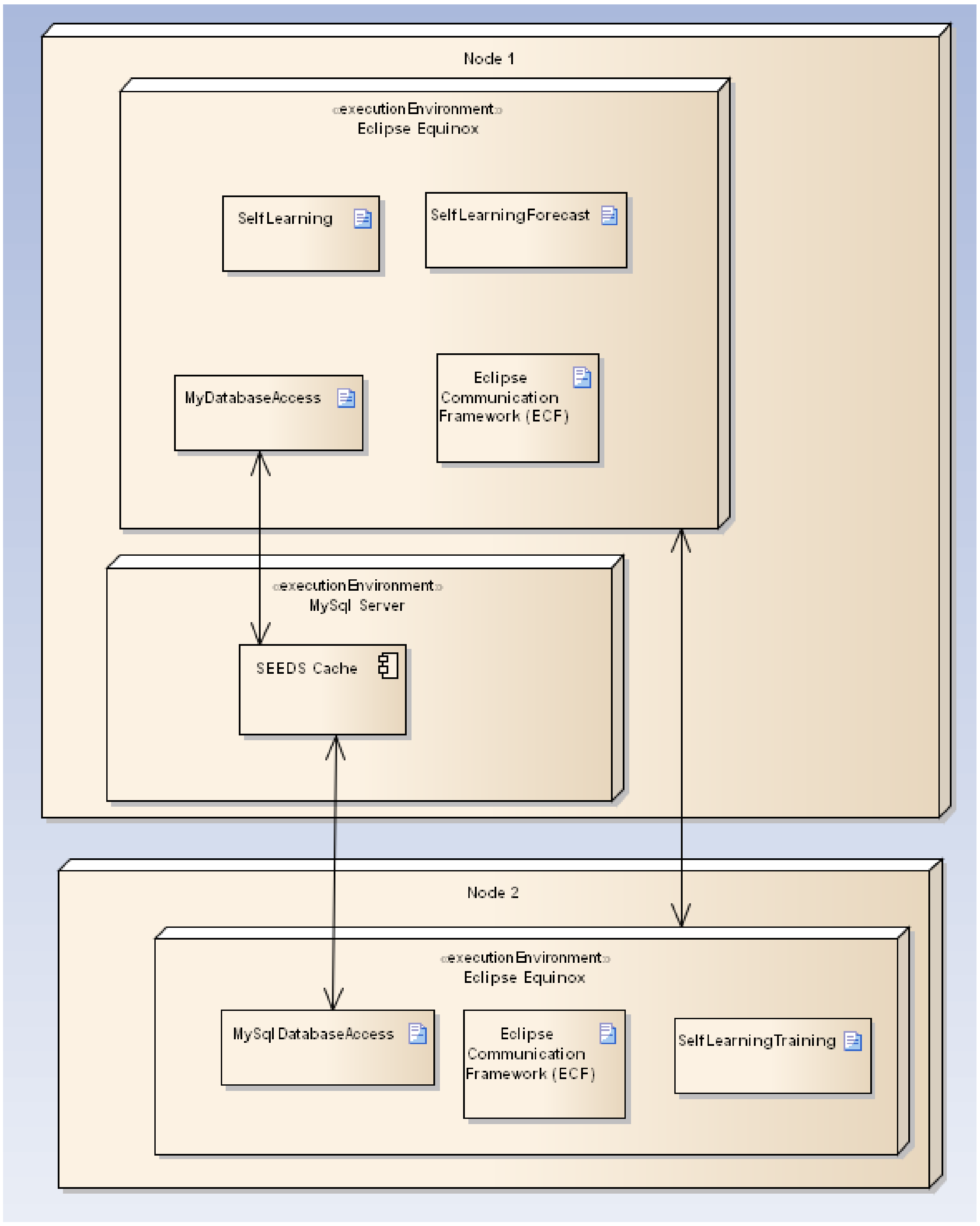


Figure 14. Deployment diagram for the SelfLearning, SelfLearningForecast and SelfLearningTraining components.

4 Tests

During the days 23 and 24 of April of 2014, the developers of the modules OPT SL and CONTROLLER met together for code integration, debugging and testing the performance of the improved algorithms.

All tests were performed on the same computer with similar performance as the pc installed in the pilots. The computer has the following characteristics:

- Processor: i7-2640M CPU 2.8GHz
- Memory: 8 GB
- Hard disk: SSD
- OS: Windows7, Java 7 32bit

Due to the missing of a real WISAN installation and real data, all tests were performed with a WISAN Mock-up. The data generated by WISAN Mock-up are randomly data within realistic ranges (e.g.: temperature values within 20 and 25ºC). For this reason, all experiments aim only to test the integration and performance of the several algorithms.

T1) Integration and function test of SL and OPT with the CONTROLLER

This test consisted in executing the SEEDS bundles together. The aim was to check if the JAVA OSGI bundles are able to intercommunicate and successfully interface with each other. To check the status of the test, debugging checkpoints were strategically set in the code in a way that the values of the variables were controlled in run time.
After some minor debugging the Self-Learning and OPT was successfully integrated with the Controller.

T2) Parallel execution of OPT and Building Model

To test if there are concurrency problems, the OPT and the Building Model were called simultaneously. The parallel execution of Optimizer and Building Model was successfully completed without any errors.

T3) Performance of the SEEDS software

This test consisted in executing two distinct experiments:

a) 30 min forecast optimization cycle

OPT was called to optimize control setting with a 30 min of forecast to look ahead. The duration of the interval step was 10 min, meaning that OPT was performing a 3 step optimization. To test the performance of SEEDS software, the OPT was set with a PSO with 15 particles and 15 iterations.
The time consumed was 1 min 30 sec.

b) 60 min forecast optimization cycle

OPT was called to optimize control setting with a 60 min of forecast to look ahead. The duration of the interval step was 10 min, meaning that OPT was performing a 6 step optimization. To test the performance of SEEDS software, the OPT was also set with a PSO with 15 particles and 15 iterations.
The time consumed was 4 min 30 sec.

c) 120 min forecast optimization cycle

OPT was called to optimize control setting with a 120 min of forecast to look ahead. The duration of the interval step was 10 min, meaning that OPT was performing a 12 step optimization. To test the performance of SEEDS software, the OPT was also set with a PSO with 15 particles and 15 iterations.
The time consumed was 7 min 20 sec.

d) Self-Learning Retraining

SL was called to retrain the neural network in parallel. The integration and parallel execution of the The Self-Learning retraining of the neuronal networks went successful and it took 1 min and 8 sec to retrain one network.

e) Optimization 8 hours running test

OPT was called to optimize control setting setup with similar conditions as for the previous test. The aim was to check the SEEDS Software for stability issues as memory leaks or execution time problems.
As result we verified that the execution time of the Building Model and Optimization remained stable. A memory leak was identified and fixed.

The overall results from the improvements introduced in the algorithms are very satisfactory and demonstrated that the concerns mentioned in the D5.3 (Esteves, 2013) are effectively minimized.

5 Conclusions

Deliverable 5.3 revealed concerns regarding high execution times for the optimization cycle. The identified issues were due to the high computational calls of SL and the high search space of the optimization problem.
SL and OPT implemented several strategies to reduce the execution time:

- Smart initialization of the PSO particles
- Dynamic OPT call
- Dynamic restrictions of the search space
- Caching
- Concurrency

By the time of the conclusion of the task 5.5 there is not enough data to train the algorithms and test the performance in the demonstrators. However, the improvements introduced in the algorithms were tested in a meeting by the partners: UiS, USALF, Fraunhofer, Softcrits and Cemosa..

These tests were successful in terms of interfacing and the results show that the execution time and search space are likely to have been resolved but it is worth noting that at the time of the tests, the data available was mock-up data.

In summary, the SL and Optimisation bundles offer an implementation of the SEEDs architecture that allows easy adaptation to forecasting for different building types and a specialised use of Swarm Optimisation suitable for Energy Optimisation.

Results of the performance of the whole SEEDS system (and therefore, the performance of Optimization and Self-Learning) in the SEEDS plot will be reported in D2.9.

6 References

Richards, M., and D. Ventura. 2004. “Choosing a Starting Configuration for Particle Swarm Optimization.” In , 3:2309–2312 vol.3. doi:10.1109/IJCNN.2004.1380986.

Donath, Ulrich, Jens Wurm, Richard Meyer, Francisco Javier Marquez Pradas, Rui Esteves, and Shane Montague. “D2.9 Energy Control Strategy. Final Version.” Project Deliverable, 2014.

Donath, Ulrich, Jürgen Haufe, Rui Esteves, Tomasz Wlodarczyk, Shane Montague, and Sunil Vadera. “D2.8 Energy Control Strategy. First Version.” Project Deliverable, 2013.

Esteves, Rui Máximo. “D5.3 Optimisation algorithms that have been tested on sample scenarios based on planned use cases.” Project Deliverable, 2013.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA.: Addison-Wesley Longman Publishing Co., Inc., 1995.

Vadera, Sunil, Jia Wu, Shane Montague, and Farid Meziane. “D5.2 Self-learning algorithms that have been verified with sample scenarios and test data.” Project Deliverable, 2013.

Jimenez, Noemi, A Barragán, Luis Neto, and Valentin Castaño “D8.1. Energy balance and environment impact. Best practives and lessons learned.” Project Deliverable, 2012.