WOF: Towards Behavior Analysis and Representation of Emotions in Adaptive Systems

Ilham Alloui, Flavien Vernier

WOF: Towards Behavior Analysis and Representation of Emotions in Adaptive Systems

Ilham Alloui and Flavien Vernier[0000−0001−7684−6502]

Univ. Savoie Mont Blanc, LISTIC,
F-74000 Annecy, France

Keywords: Object oriented design, Architecture models, Adaptive systems, Introspection, Decentralized control, Behavior analysis, Emotion representation

Abstract. With the increasing use of new technologies such as Communicating Objects (COT) and the Internet of Things (IoT) in our daily life (connected objects, mobile devices, etc.), designing Intelligent Adaptive Distributed software Systems (DIASs) has become an important research issue. Human face the problem of mastering the complexity and sophistication of such systems as those require an important cognitive load for end-users who usually are not expert.
Starting from the principle that it is to technology-based systems to adapt to end-users and not the reverse, we address the issue of how to help developers design and produce such systems. We then propose WOF, an object oriented Framework founded on the concept of *Wise Object* (WO), a metaphor to refer to human introspection and learning capabilities.
To make systems able to learn by themselves, we designed introspection, monitoring and analysis software mechanisms such that WOs can learn and construct their own knowledge. We then define a WO as a software-based entity able to learn by itself on itself (i.e. on services it is intended to provide) and also on the others (i.e. the way others use its services). A WO is seen as an avatar of either a physical or a logical object (e.g. device / software component).
In this paper, we introduce the main requirements for DIASs as well as the design principles of WOF. We detail the WOF conceptual architecture and the Java implementation we built for it. To provide application developers with relevant support, we designed WOF with the minimum intrusion in the application source code. Adaptation and distribution related mechanisms defined in WOF can be inherited by application classes. In our Java implementation of WOF, object classes produced by a developer inherit the behavior of *Wise Object* (WO) class. An instantiated system is a *Wise Object System* (WOS) composed of WOs that interact through an event bus. In the first version of WOF, a WO was able to use introspection and monitoring built-in mechanisms to construct knowledge on: (a) services it is intended to render; (b) the usage done of its services. In the current version, we integrated an event-based WO simulator and a set of Analyzer classes to provide a WO with the possibility to use different analysis models and methods on its data. Our major goal is that a WO can be able to identify common usage of its services and to detect unusual usage. We use the metaphor of emotions to refer to unusual behavior (stress, surprise, etc.). We show in the paper a first experiment based on a statistical analysis method founded on stationary processes to identify usual/unusual behavior.

1 Introduction

With the increasing use of new technologies such as Communicating Objects (COT) and the Internet of Things (IoT) in our daily life (connected objects, mobile devices, etc.), designing Intelligent Adaptive Distributed software Systems (DIASs) has become an important research issue. Human face the problem of mastering the complexity and sophistication of such systems as those require an important cognitive load for end-users who usually are not expert.
Multiplicity of users, heterogeneity, new usages, decentralization, dynamic execution environments, volumes of information result in new system design requirements: technologies should adapt to users more than users should do to technologies. In the domain of home automation for example, both end-users and system developers face problems:

- end-users: instructions accompanying the devices are too complex and it is hard for non-expert users to master the whole behavior of the system; such systems are usually designed to meet general requirements through a set of predefined configurations (a limited number of scenarios in the best case). A user may need a set of services in a given context and a different set of services in another context. A user does not need to use all what a system could provide in terms of information or services.
- developers lack software support that help them build home automation systems able to adapt to end-users. Self-adaptation mechanisms are not mature yet and most existing support approaches are either too specific or too abstract to be helpful as stated in [**?**].

All along the paper, we use a simple example of home automation system (see Figure 1)To illustrate our purposes. Let us consider a system composed of a roller shutter (actuator) and a control key composed of two buttons (sensors).
In the general case and in a manual mode, with a one-button control key, a person can: bring the shutter either to a higher or to a lower position. With a second button, the user can tune inclination of the shutter blades to get more or less light from the outside. As the two buttons cannot be activated at the same time, the user must proceed in two times: first, obtain the desired height (e.g. 70%) then the desired inclination (e.g. 45%). For such systems, three roles are generally defined: *system developer*, *system configurator* and *end-user*. Assume an end-user is at his office and that according to time and weather, his/her requirements for the shutter change (height and inclination). This would solicit the end-user all along the day and even more when there are several shutters with different exposure to the sun. From a developer's point of view, very few support is available to easily construct adaptive systems: when provided, such support is too specific and cannot be easily reused in another context. Adaptation mechanisms and intelligence are generally merged with the application objects which make them difficult and costly to reuse in another application or domain.
Starting from the principle that it is to technology-based systems to adapt to end-users and not the reverse, we address the issue of how to help developers design and produce such systems. We then propose WOF, an object oriented Framework founded on the concept of *Wise Object* (WO), a metaphor to refer to human introspection and learning capabilities.
To make systems able to learn by themselves, we designed introspection, monitoring and analysis software mechanisms such that WOs can learn and construct their own knowledge and experience.

Fig. 1. An example of home automation system

According to this approach (see Figure 2), ”wise” buttons and shutters would gradually construct their experience (e.g. by recording effect of service invocation on their state, statistics on invoked services, etc.) and adapt their behavior according to the context (e.g. physical characteristics of a room, an abstract state defined by a set of data related to the weather, the number of persons in the office, etc.). From the development perspective, we separate in the WOF objects' ”wisdom” and intelligence logic (we name *abilities*) from objects' application services (we name *capabilities*) they are intended to render.

To provide application developers with relevant support, we designed WOF with the minimum intrusion in the application source code. Adaptation and distribution related mechanisms defined in WOF can be inherited by application classes. In our Java implementation of WOF, object classes produced by a developer inherit the behavior of *Wise Object* (WO) class. An instantiated system is a *Wise Object System* (WOS) composed of WOs that interact through an event bus. In the first version of WOF, a WO was able to use introspection and monitoring built-in mechanisms to construct knowledge on: (a) services it is intended to render; (b) the usage done of its services. In the current version, we integrated an event-based WO simulator and a set of Analyzer classes to provide a WO with the possibility to use different analysis models and methods on its data. Our major goal is that a WO can be able to identify common usage of its services and to detect unusual usage. We use the metaphor of emotions to refer to unusual behavior (stress, surprise, etc.). We show in the paper a first experiment based on a statistical analysis method founded on stationary processes to identify usual/unusual behavior.

In the paper, we focus mainly on the architecture of the WO and WOS, their global structure and behavior. In Section 2, we discuss the challenges and requirements for DIASs. Then we present design principles and fundamental concepts underlying WOF in Section 3. In Section 4 we detail the structure and behav-

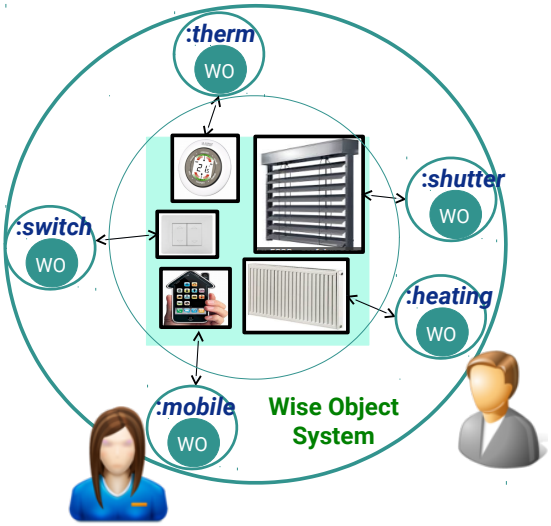


Fig. 2. A wise home automation system

ior of a WO and a WO System (WOS) and present the architectural patterns we adopted in our design. We focus in section Section 5 on WO knowledge analysis using statistical approaches, in particular to identify common usage and detect unusual behavior. To illustrate how to use the WOF, we give an example in the home automation domain in Section 6. Finally, in Section 7 we discuss our approach and conclude with ongoing work and some perspectives.

2 Requirements

A technology-based system should be able to: (1) *know by itself on itself*, i.e. to learn how it is intended to behaves, to consequently reduce the learning effort needed by end-users (even experimented ones); (2) *know by itself on its usage* to adapt to users' habits. In addition like any service-based system (3) such system should be able to improve the quality of services it is offering. WOF aims at helping developers producing such systems while meeting end-users' requirements:

- *Requirement 1*: We need *non-intrusive* systems that serve users while requiring *just some* (and not all) of their attention and only when necessary. This contributes to *calm-technology* [**?**] that *describes a state of technological maturity where a user's primary task is not computing, but being human*. As claimed in [**?**], new technologies might become highly *interruptive* in human's daily life. Though *calm-technology* has been proposed first by Weiser and Brown in early 90's [**?**], it remains a challenging issue in technology design.
- *Requirement 2*: We need systems composed of *autonomous* entities that are able to independently adapt to a changing context. If we take two temperature sensors installed respectively inside and outside the home, each one

reacts differently based on its own experience (knowledge). A difference in temperature that is considered as normal outside (e.g. 5 degrees) is considered as significant inside. Another situation is when an unexpected behavior occurs, for example a continuous switching on - switching off of a button. In such a case, the system should be able to identify unusual behavior according to its experience and to decide what to do (e.g. raising an alert);

– *Requirement 3*: In an ideal world, an end-user declares his/her needs (a goal) and the system looks for the most optimal way to reach it. This relates to goal-oriented interaction and optimization. The home automaton system user in our example would input the request "I want the shutter at height h and inclination i" and the system based on its experience would choose the "best" way to reach this state for example by planning a set of actions that could be the shortest one or the safest or the less energy consuming, etc. according to the non-functional quality attributes that have been considered while designing the system [**?**].

Many approaches are proposed to design and develop the kinds of systems we target: multi-agent systems [**?**], intelligent systems [**?**], self-X systems [**?**], adaptive systems [**?**]. In those approaches, a system entity (or agent) is able to learn on its environment through interactions with other entities. Our aim is to go a step forward by enhancing a system entity with the ability to learn by its own on the way it has been designed to behave. There are at least two advantages to this: (a) as each entity evolves independently from the others, it can control actions to perform at its level according to the current situation. This enables a decentralized control in the system; (b) each entity can improve its performance and then the performance of the whole system, i.e. a collaborative performance.

While valuable, existing design approaches are generally either domain-specific or too abstract to provide effective support to developers. The IBM MAPE-K known cycle for autonomic computing [**?**] is very helpful to understand required components for self-adaptive systems but still not sufficient to implement them. Recently, more attention has been given to design activities of self-adaptive systems: in [**?**], authors propose design patterns for self-adaptive systems where roles and interactions of MAPE-K components are explicitly defined. In [**?**] authors propose a general guide for developers to take decisions when designing self-adaptive systems. Our goal is to offer developers an object oriented concrete architecture support, ready to use for constructing wise systems. We view this as complementary to the work results cited above where more abstract architectures have been defined.

From a system development perspective, our design decisions are mainly guided by the following characteristics: software support should be non-intrusive, reusable and generic enough to be maintainable and used in different application domains with different strategies. Developers should be able to use the framework with the minimum of constraints and intrusion in the source code of the application. We consequently separated in the WOF the objects' "wisdom" and intelligence logic (we name *abilities*) from application services (we name *capabilities*) they are intended to render.

3 Fundamental Concepts of WOF

We introduce the fundamental concepts of WO and WOS from a runtime perspective. We adapt to this end the IBM MAPE-K known cycle for autonomic computing [**?**].

3.1 Concept of WO

We define a Wise Object (WO) as a software object able to learn by itself on itself and on its environment (other WOs, external knowledge), to deliver expected services according to the current state and using its own experience. *Wisdom* refers to the experience such object acquires by its own during its life. We intentionally use terms dedicated to humans as a metaphor. A *Wise Object* is able to learn on itself using introspection. A *Wise Object* is considered as a software avatar intended to "connect" to either a physical entity/device (e.g. a vacuum cleaner) or a logical entity. In the case of a vacuum cleaner, the WO could learn how to clean a room depending on its shape and dimensions. In the course of time, it would in addition improve its performance (less time, less energy consumption, etc.).

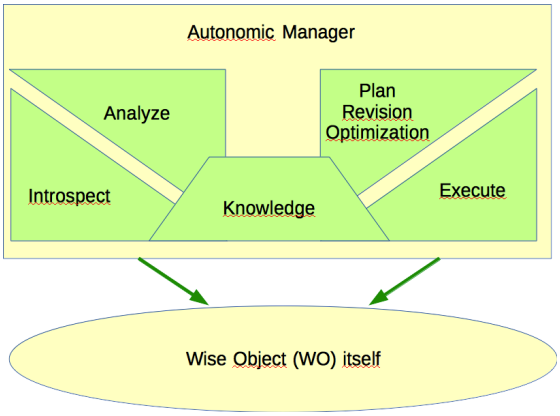
- its autonomy: it is able to behave with no human intervention;
- its adaptiveness: it changes its behavior when its environment changes;
- its intelligence: it observes itself and its environment, analyzes them and uses its knowledge to decide how to behave (introspection and monitoring, planning);
- its ability to communicate: with its environment that includes other WOs and end-users in a decentralized way (i.e. different locations)

We designed WO in a way its behavior splits into two states we named *Dream* and *Awake*. The former is dedicated to introspection, learning, knowledge analysis and management when the WO is not busy with service execution. The latter is the state the WO is in when it is delivering a service to an end-user or answering an action request from the environment. The WO then monitors such execution and usage done with application services it is responsible for. We use the word *Dream* as a metaphor for a state where services invoked by the WO do not have any impact on the real system: this functions as if the WO is disconnected from the application device/component/object it is related to.

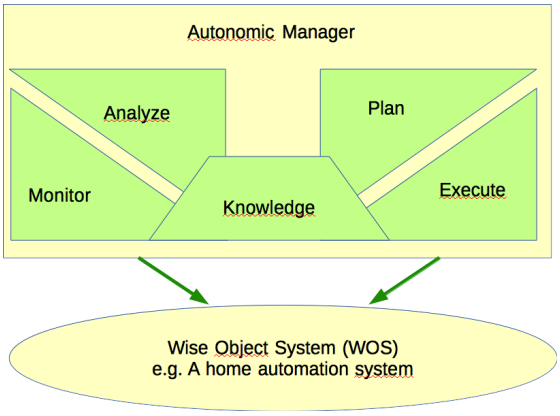
To ensure adaptiveness, each WO has a set of mechanisms that allow it to perform a kind of MAPE-K loops [**?**]. *Dream* and *Awake* MAPE-K are respectively depicted by Figure 3(a) and Figure 3(b). Let us call the dream MAPE-K a IAPE-K, due to the fact that in the dream case the Monitoring is actually Introspection.

When dreaming, a WO introspects itself to discover services it is responsible for, analyzes impact of their execution on its own state and then plans revision actions on its knowledge. WO constructs its experience gradually, along the dream states. This means that WO knowledge is not necessarily complete and is subject to revisions. Revision may result in adaptation, for instance recording a new behavior, or in optimization like creating a shortening among an action list to reach more quickly a desired state.

When awake, a WO observes and analyzes what and how services are invoked and in what context. According to its experience and to analysis results, a WO is able to communicate an *emotion* if necessary. We define a WO emotion as a distance between the common usage (usual behavior) and the current usage of its



(a) WO Dream IAPE-K



(b) WO Awake MAPE-K

Fig. 3. WO MAPE-Ks [**?**]

services. According to this metaphor, a WO can be surprised if one of its services is used while it has never been the case before. A WO can stress if one of its services is more frequently used or conversely, a WO can be bored. WO emotions are intended to be used as a new information by other WOs and/or the end-users. This is crucial to adaptation at a WOS level (e.g. managing a new behavior) and to attract attention on potential problems (e.g. alerts when temperature is unusually too high). With respect to its emotional state, a WO plans relevant actions (e.g. raising alarms, opening windows and doors, cutting off electricity, etc.).

3.2 Concept of WOS (WO System)

We define a WOS as a distributed object system composed of a set of communicating WOs. Communicated data/information (e.g. emotions) are used by the WOS to adapt to the current context. It is worth noting that each WO is not aware of the existence of other WOs. WOs may be on different locations and it is the charge of the WOS to handle data/information that coordinate WOs' behaviors. The way this is done is itself an open research question. In our case, we defined the concept of *Managers* (see Section 4) to carry out communication and coordination among WOs. This is close to the *Implicit Information Sharing Pattern* introduced in [**?**].

4 Design models of WO and WOS

WOF is an object oriented framework built on the top of a set of interrelated packages. This section introduces our design model of the concepts presented in the previous section.

4.1 Design model of WO

Figure 4 shows the UML Class diagram for WO. This model is intentionally simplified and highlights the main classes that compose a WO. WO Class is an abstract class that manages the knowledge of its sub-classes. Knowledge managed by a WO is of two kinds: capability-related (i.e. knowledge on application services) and usage-related (knowledge on service usage). In our present experiment, we have chosen a graph-based representation for knowledge on WO capabilities and usage done of them. Knowledge on WO capabilities is expressed as a state-transition graph while that on WO usage is expressed as a Markov graph where usage-related statistics are maintained. The Markov graph clearly depends on the usage of an object (a WO instance) as 2 WO instances of a same class may be used differently.
Let us recall that WO behavior is split into two states. The dream state and the awake state, see Figure 5. The dream state is dedicated to acquiring the capability knowledge and to analyzing the usage knowledge. The awake state is the state where the WO executes its methods invoked by other objects or by itself, and, monitors such execution and usage.
To build capability-related knowledge, the WO executes the methods of its subclass (i.e. the application class) to know their effect on the attributes of this subclass.This knowledge is itself represented by a state-transition diagram. Each set

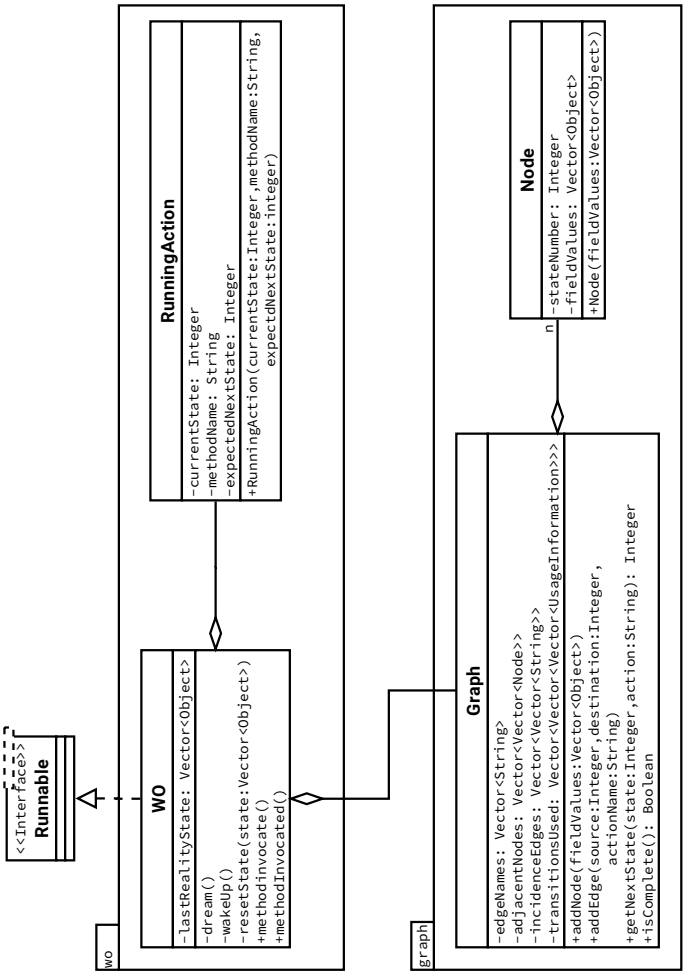
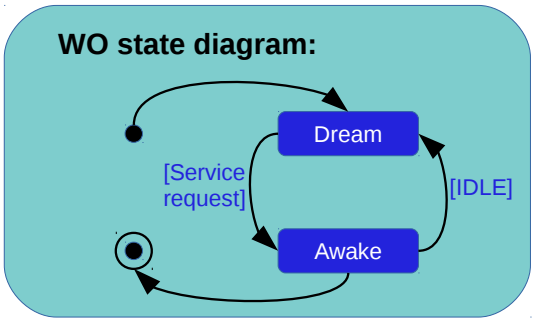
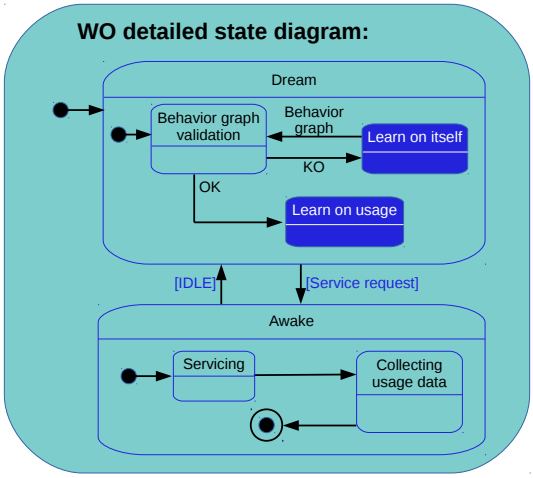


Fig. 4. UML class diagram of WO



(a) WO short state diagram



(b) WO detailed state diagram

Fig. 5. UML state diagram of WO built-in behavior [**?**]

of attribute values produces a state in the diagram and a method invocation produces a transition. The main constraint in this step is that method invocation must have no real effect on other objects of the application when the WO is dreaming. This is possible thanks to the system architecture described in Section 4.2.
Regarding usage-related knowledge on an application object, two kind of situations are studied: emotions and adaptation of behavior.
As introduced in section 3, an emotion of a WO is defined as a distance between its current usage and its common usage. WO can be stressed if one of its methods is more frequently used or conversely, a WO can be bored. WO can be surprised if one of its method is used and this was never happened before.
When a WO expresses an emotion, this information is caught by the WOS and/or other WOs and that may consequently lead to behavior adaptation. At the object level, two instances of the same class that are used differently – different frequencies, different methods... – may have different emotions, thus, different behavior and interaction within the WOS.
A WO uses its capability-related knowledge to compute a path from a current state to a known state [**?**]. According to the frequency of the paths used, a WO can adapt its behavior. For example, if a path is often used between non-adjacent states, the WO can build a shortcut transition between the initial state and the destination state; it then can also build the corresponding method within its subclass instance (application object). This modifies the capability-related graph of this instance.

4.2 Design model of WOS

As explained in Section 3, WOs are not aware of the existence of other WOs. They are distributed and communicate data/information towards their environment. WOs may be on different locations and one or many *Managers* carry out communication and coordination among them. In this paper, we propose a concrete architecture based on a bus system, where any WO communicates with other objects through the bus. This architecture has many advantages.
A first one is the scalability. It is easier to add WOs, managers, loggers... on this kind of architecture than to modify a hierarchical architecture. Moreover, this architecture is obviously distributed and enables distribution/decentralizion of WOs in the environment.
The third main advantage is the ability for a WO to disconnect/reconnect from/to the bus when needed. This makes it possible the implementation of the Dream state (see Section 3). Let us recall that in the dream state, a WO can invoke its own methods to build its capability-related graph, but these invocations must not have any effect on the subject system.
Thus, when a WO enters the Dream state, it disconnects itself from the bus and can invoke its methods without impact on the real world system. More precisely, the WO disconnects its ”sending system” from the bus, but it continues receiving data/information via the bus. Therefore, if a WO in Dream state receives a request, it reconnects to the bus, goes out from the Dream state to enter into Awake state and serves the request.
Figure 6 shows the UML class diagram of a bus-based WO system. This model is simplified and highlights the main classes. The system uses an Event/Action mechanism for WOs' interactions. On an event, a state change occurs in a WO, an action may be triggered on another WO. The peers “Event/Action” are defined

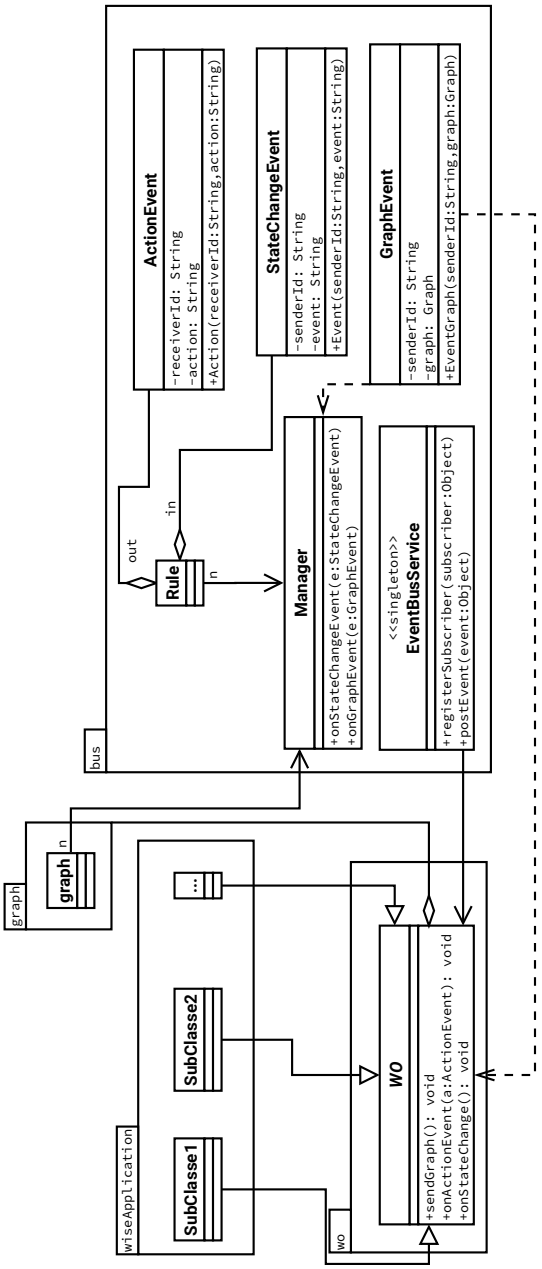


Fig. 6. UML class diagram of a bus-based WO system

by Event, Condition, Action (ECA) rules that are managed by a Manager. When this latter catches events (StateChangeEvent), it checks the rules and conditions and posts a request for action (ActionEvent) on the bus. From the WO point of view, if one of its subclass instance state changes at Awake state, it posts a StateChangeEvent on the bus. When a WO receives an ActionEvent, two cases may occur: either the WO is in Awake state or in Dream state. If the WO is in Awake state, it goes to the end of its current action and starts the action corresponding to the received request. If the WO is in Dream state, it stops dreaming and enters into the Awake state to start the action corresponding to the received request.
In our Java implementation of WOF, object classes produced by a developer inherit the behavior of *Wise Object* (WO) class. An instantiated system is defined as a *wise system* composed of *Wise Objects* that interact through a (or a set of distributed) *Manager(s)* implemented by an event-based bus according to *publish-subscribe* design pattern.

4.3 Design model of WO Data Analyzers

As stated all along the paper, a WO is able to collect and analyze usage-related data. To enrich the WOF framework and to offer the possibility to use different analysis models and methods on the same data, we associate a WO with a set of Analyzer classes according to a Factory-like design pattern [**?**]. This design decision aims at defining a set of analyzers on collections of data (i.e. instances of ”Graph”) with the possibility to compare analysis results.
Therefore an Analyzer class named ”daClass” is statically registered into the WO class using ”registerDAClass(daClass: Class<DataAnalyzer>)” static method, where ”DataAnalyzer” is the abstract class for analyzers. Analyzers of a WO – the analyzers of knowledge graphs of the WO – are then instantiated in the WO constructor. Figure 7 illustrates this design model. This approach is close to factory pattern where the factory is the WO class, the ”DataAnalyzers” are the products and the WO instances are the clients.
The ”DataAnalyzer” abstract class implements the ”Runnable” interface so that an analyzer is implemented by an independent thread. This abstract class defines 3 abstract methods: ”DataAnalyzer(g:Graph)”, ”resume()” and ”suspend()”. The first ”DataAnalyzer(g:Graph)” is the default constructor that requires the graph of knowledge to analyze. The second ”resume()” starts or resumes the analyzer. Let us recall that the analysis – the learning activity – only occurs within the dream state of a WO. Therefore, the analyzers must be stopped and resumed accordingly. The last method ”suspend()” suspends the analyzer. As depicted by Figure 7, we realized an implementation of the ”DataAnalyzer” abstract class: ”StatAnalyzer”. The ”StatAnalyzer” performs a statistical analysis of events: occurrences of graph transitions. It stores for each transition the dates of its occurrences in a ”VectorAnalyzer”. Each ”VectorAnalyzer” of a ”StatAnalyzer” is characterized by a ”windowSize” that represents the memory size of the vector. When the vector is full, if a new event occurs, the oldest is forgotten: removed from the vector. Moreover, as the analyzer only performs the analysis during the dream states of a WO, it also stores information about the data that has already been analyzed to resume analysis from where it stopped at the last suspend. Regarding the ”VectorAnalyzer”, this class extends the descriptive statistics library of the ”The Apache Commons Mathematics Library”. The descriptive statistics provides a dataset of values of a single variable and computes descriptive statistics based on stored

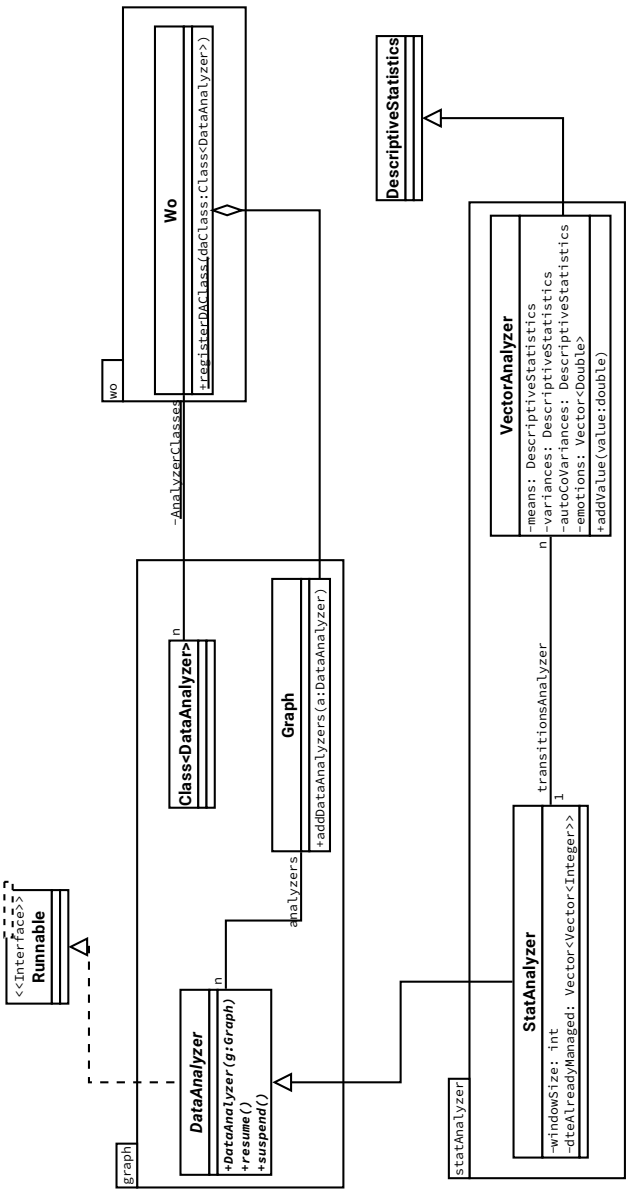


Fig. 7. UML class diagram of WO Data Analyzer Pattern

data. The number of values that can be stored in the dataset may be limited or not. The "VectorAnalyzer" extends this class to provide statistics about the evolution of means, variances and autocorrelations, when a new value is added into the dataset. A representation of this evolution – a distance with the stationarity – is stored in the "emotion" vector. The next section describes this analysis approach.

5 Statistical analysis and WO emotions

The first analyzer we implements is based on the weaker forms of stationarity of the process [**?**]. The stationarity study focuses on the occurrences of the event. We call event the invocation of a method from a given state (i.e. the execution of a transition of the knowledge graph). Figure 8 gives a graph of knowledge, with 4 possible events: $t1, t2, t3$ and $t4$. This events occur at different times, for example, on Figure 8 the event $t1$ occurs at times $\left[e^1_{t1}, e^2_{t1}, e^3_{t1} \ldots\right]$ In this first analysis, we

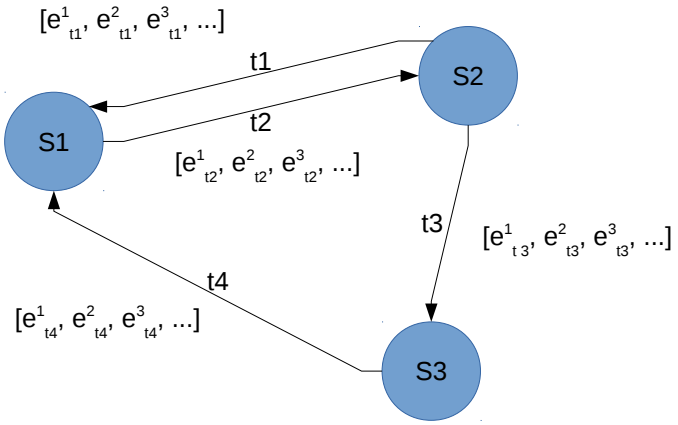


Fig. 8. Example of time series for knowledge graph analysis

do not study the correlation between the events.
Let $x(i)$ a continuous and stationary time random process. The weaker forms of stationarity (WSS) defines that the mean $E[x(i)]$ and variance $Var[x(i)]$ do not vary with respect to time and the autocovariance $Cov[x(i), x(i-k)]$ only depends on range k.
This process is a WSS process if and only if:

$$\begin{aligned} &E[x(i)] = \mu \quad \forall i, \\ &Var[x(i)] = \sigma^2 \neq \infty \quad \forall i, \\ &Cov[x(i), x(i-k)] = f(k) = \rho_k \quad \forall i \forall k. \end{aligned}$$

This definition implies the analysis of the whole time series. In our case, the common usage can change and we define it by the stationarity. Therefore, we

compute the stationarity – the common usage – on a sliding window of size w:

$$
\begin{aligned}
&E[x(i)] = \mu \quad \forall i \in [t-w,t],\\
&Var[x(i)] = \sigma^2 \neq \infty \quad \forall i \in [t-w,t],\\
&Cov[x(i),x(i-k)] = f(k) = \rho_k \quad \forall i \in [t-w,t] \forall k,
\end{aligned}
$$

where the time series $x(i)$ are the occurrences $\left[e_\tau^{t-w} \dots e_\tau^i \dots e_\tau^t\right]$ of a given event – i.e. transition – τ between t-w and t.
According to this definition of the stationarity, we define an emotion as the distance between the current usage and the common usage, in other words the distance with the stationarity measure. We define this distance $d(x(i))$ by the following centered normalized scale where:

$$
d(x(i)) = \begin{cases}
d(E[x(i)]) = \frac{E[x(i)] - \overline{E[x(j)]}}{(\max(E[x(j)]) - \min(E[x(j)]))/2},\\
d(Var[x(i)]) = \frac{Var[x(i)] - \overline{Var[x(j)]}}{(\max(Var[x(j)]) - \min(Var[x(j)]))/2},\\
d(Cov[x(i),x(i-k)]) = \frac{Cov[x(i),x(i-k)] - \overline{Cov[x(j),x(j-k)]}}{(\max(Cov[x(j),x(j-k)]) - \min(Cov[x(j),x(j-k)]))/2},
\end{cases}
$$

where $j \in [t-w,t]$ and $\overline{E[x(j)]}$, $\overline{Var[x(j)]}$ and $\overline{Cov[x(j),x(j-k)]}$ are respectively the means of means, variances and autocovariances on the range $[t-w,t]$.
Thus, when a new event occurs at $t+1$, we compute the distance with the common usage between $t-w$ and t. If all values of the distance – $d(E[x(i)])$, $d(Var[x(i)])$ and $d(Cov[x(i),x(i-k)])$ – are in $[-1,1]$ this is considered as a common behavior, otherwise this is identified as a behavior change (unusual usage) relatively to the knowledge on the common usage.

6 An illustrating example "Home automation"

The concept of WO has many scopes of application. It can be used to adapt an application to its environment, to monitor an application from inside, to manage an application according to the usage done of it... In this section, we highlight the WO behavior within a home automation application. This choice is justified by the fact that:

- home automation systems are usually based on a bus where many devices are plugged on;
- home automation devices have behavior that can be represented by a simple state diagram.

According to the first point, a home automation system can be directly mapped onto a WO system based on a bus where the home automation devices are related to WOs. The second point avoids the combinatorial explosion that can appear due to a large number of states to manage in a state diagram.
Let us take a simple example of a switch and a shutter. The switch is modeled by 2 states "on" and "off" and 3 transitions "on()" , "off()" and "switch()".

Listing 1.1. Switch Java code

```
public class Switch extends Wo {
  public boolean position;

  public Switch() {
```

```
    super();
  }
  public void on() {
    invoke();
    position = true;
    invoked();
  }
  public  void off() {
    invoke();
    position = false;
    invoked();
  }
  public  void switch() {
    invoke();
    if(position){
      position = false;
    }else{
      position = true;
    }
    invoked();
  }

}
```

The shutter is modeled by n states that represent its elevation between 0% and 100%. If the elevation is 0%, the shutter is totally closed and if the elevation is 100%, the shutter is totally open. To avoid a continuous system, the shutter can only go up or down step by step.

Listing 1.2. Shutter Java code

```
public class RollingShutter extends Wo {
  private int elevation = 0;
  private static int step = 20;

  public RollingShutter() {
    super();
  }

  public void down(){
    methodInvocate();
    if(this.elevation>0){
      this.elevation -= RollingShutter.step;
    }
    if(this.elevation <= 0){
      this.elevation = 0;
    }
    methodInvocated();
  }

  public void up() {
```

```
    methodInvocate();
    if(!this.elevation < 100){
      this.elevation += RollingShutter.step;
      if (this.elevation >= 100){
        this.elevation = 100;
      }
    }
    methodInvocated();
  }
```

As one design principle behind WOF is to minimize intrusion within the application source code, we have succeeded to limit them to the number of two "warts". The examples highlight those 2 intrusions in the code. They are concretized by two methods implemented in the WO Class – methodInvocate() and methodInvocated() – and must be called at the beginning and the end of any method of the WO subclass (application class). Those methods monitor the execution of a method on a WO instance. We discuss about these "warts" in the last section.

In our example, an instance of Switch and another of RollingShutter are created. Two ECA rules are defined to connect those WOs:

- [Switch*Instance*.on? / True / RollingShutter*Instance*.up()]
- [Switch*Instance*.off? / True / RollingShutter*Instance*.down()]

They define that when the event "on" occurs on the switch, the action – method – "up" must be executed on the rolling shutter and that when the event "off" occurs on the switch, the action "down" must be executed on the rolling shutter. For the experiment and feasibility study, the action on the Switch*Instance* – "on()" and "off()" invocations – are simulated using the WO simulator we are developing. The actions "on" and "off" occur according to a Poisson distribution and depend on the elevation of the rolling shutter. The likelihood of action "off" occurrence is $RollingShutterInstance.elevation/100$, the likelihood of action "on" occurrence is inversely proportional. When an action occurs, "on" or "off", it can occur x times successively without delay, where x is bounded by the number of occurrences to reach the bound of shutter elevation, respectively 100% and 0%.

Presently, a WO acquires knowledge about its capabilities using a graph representation. The knowledge about its usage is the logs of all its actions/events and can be presented by a Markov graph. The logging presented in Log 1 shows the events occurred on each WO of the system. This information is collected from each WO. With this information each WO can determine its current behavior and a manager can determine the system behavior. This is discussed in Section 7. Log 2 gives Markov graph logging representation. Let us note that the Markov graph representation hides time-related information as it is based on the frequency of occurrences. Log 2 shows that the wise part of the Switch instance detects the 2 states and the 6 transitions. It also shows a 2x2 adjacency matrix followed by a description of the 6 transitions including their usage-related statistics.

Log 2 shows for instance that from the state 0 with the position attribute at false, the Switch*Instance* may execute method "on()" or "switch()" and go to state 1 or execute method "off()" and remain in the same state 0. Usage-related statistics show that method "switch()" is never used from the state 0 all along the 1000 iterations.

Regarding the RollingShutter instance, the logging after the 2nd iteration (Log 3) and the last Log 4 are given. Log 3 shows that the wise part of the RollingShutter

Switch*Instance*		RollingShutter*Instance*	
on	off	up	down
0	1769	3	1774
1	6015	5	6016
1	6624	5	6625
4263	10435	4264	10436
8523	10435	8525	10444
9963	11026	9968	11028
9964	11026	9966	11028
10994	12615	10997	12616
10995	15811	10996	15816
...	...	...	...

Log 1: First event log stored on each WO.

```
Graph:
2 States, 6 Transitions
   0 1
0: 1 1
1: 1 1
State [0 , false] :
      Adjacency on->[1 , true] - 0.313,
                switch->[1 , true] - 0.0,
                off->[0 , false] - 0.687,
State [1 , true] :
      Adjacency off->[0 , false] - 0.311,
                on->[1 , true] - 0.689,
                switch->[0 , false] - 0.0,
Current State: 1
```

Log 2: Switch log after 1000 iterations.

instance detects 6 states and 10 transitions (green values of adjacency matrix). Consequently, it has not detected all the possible transitions yet. This incomplete

```
Graph:
6 States, 10 Transitions
   0 1 2 3 4 5
0: 1 1 0 0 0 0
1: 1 0 1 0 0 0
2: 0 1 0 1 0 0
3: 0 0 1 0 1 0
4: 0 0 0 1 0 1
5: 0 0 0 0 0 0
State [0 , 0 , 20] :
      Adjacency down->[0 , 0 , 20] - 0.0,
                up->[1 , 20 , 20] - 1.0,
State [1 , 20 , 20] :
      Adjacency up->[2 , 40 , 20] - 1.0,
                down->[0 , 0 , 20] - 0.0,
State [2 , 40 , 20] :
      Adjacency down->[1 , 20 , 20] - 1.0,
                up->[3 , 60 , 20] - 0.0,
State [3 , 60 , 20] :
      Adjacency up->[4 , 80 , 20] - 0.0,
                down->[2 , 40 , 20] - 0.0,
State [4 , 80 , 20] :
      Adjacency down->[3 , 60 , 20] - 0.0,
                up->[5 , 100 , 20] - 0.0,
State [5 , 100 , 20] :
      Adjacency ,
Current State: 1
```

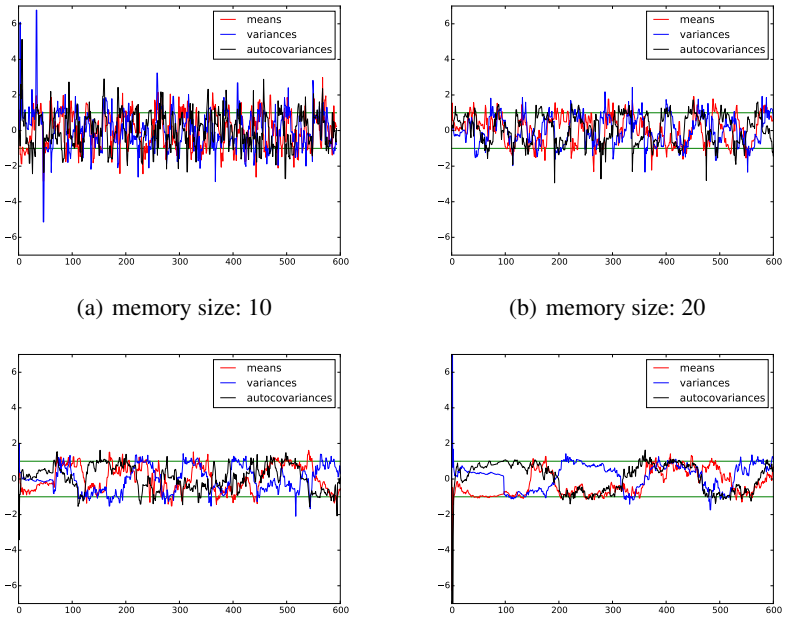
Log 3: Rolling shutter log after the 2nd iteration.

knowledge is not a problem, during the next Dream state or if it uses those transitions during the Awake state, the WO part of the application object will update its knowledge. The last Log 4 shows that all states and transitions are detected (learnt).

Another analysis of the log is given by Figure 9. This figure presents the ”emotion” for the event ”on” from state 0, computed from the statistical analysis presented in section 5. Figures 9(a), 9(b), 9(c) and 9(d) present the analysis of the common usage with different sizes of memory (window size of ”VectorAnalyzer”). Between -1 and 1, the behavior is considered as usual, because it already has appeared in the past stored in the memory. Out of the range $[-1..1]$ an unusual behavior is detected relatively to the knowledge in memory and the more important the distance with the range is, the more important the emotion is. Those results are consistent from the data analysis point of view: the bigger the time window is, the smoother the result is. From the emotion point of view, it means that the wider the memory is, the less unusual behavior is detected.

```
Graph:
6 States, 12 Transitions
   0 1 2 3 4 5
0: 1 1 0 0 0 0
1: 1 0 1 0 0 0
2: 0 1 0 1 0 0
3: 0 0 1 0 1 0
4: 0 0 0 1 0 1
5: 0 0 0 0 1 1
State [0 , 0 , 20] :
      Adjacency down->[0 , 0 , 20] - 0.0,
                up->[1 , 20 , 20] - 1.0,
State [1 , 20 , 20] :
      Adjacency up->[2 , 40 , 20] - 0.653,
                down->[0 , 0 , 20] - 0.347,
State [2 , 40 , 20] :
      Adjacency down->[1 , 20 , 20] - 0.456,
                up->[3 , 60 , 20] - 0.544,
State [3 , 60 , 20] :
      Adjacency up->[4 , 80 , 20] - 0.443,
                down->[2 , 40 , 20] - 0.557,
State [4 , 80 , 20] :
      Adjacency up->[5 , 100 , 20] - 0.375,
                down->[3 , 60 , 20] - 0.625,
State [5 , 100 , 20] :
      Adjacency up->[5 , 100 , 20] - 0.0,
                down->[4 , 80 , 20] - 1.0,
Current State: 1
```

Log 4: Rolling shutter log after the last iteration.



(a) memory size: 10 (b) memory size: 20

(c) memory size: 40 (d) memory size: 80

Fig. 9. Evolution of emotions according to memory size

It is worth noticing that this example is intentionally simple as our goal is to highlight the kind of knowledge a WO can currently acquire and analyze. Capability graphs and usage logging are the knowledge base for WOs. We discus the management and use of this knowledge in Section 7.

7 Discussion and concluding remarks

Our work addresses the issue of designing distributed adaptive software systems through WOF: a software object-based framework implemented in Java. At a conceptual level, WOF is built around the concept of "wise object" (WO), i.e. a software object able to: (a) learn on its capabilities (services), (b) learn on the way is being used and (c) perform data analysis to identify common usage and detect unusual behavior. At a concrete level, a WO uses: (a) introspection and monitoring mechanisms to construct its knowledge, (b) an event-based bus to communicate with the system and (c) a set of analyzers to identify both usual and unusual behaviors.

Regarding data analysis, we implemented a first statistical analyzer, based on the theory of stationary processes. This experiment is a step forward towards behavior analysis and emotion representation in adaptive systems. As shown in the paper, an emotion is defined as a distance between an unusual behavior and a common behavior.

Our work and experiments around WOF raised many research issues and perspectives. A first main perspective is to use other knowledge aggregation theories/techniques to represent *emotions* of a *Wise Object*: solutions may involve techniques from information fusion, multi-criterion scales or fuzzy modeling. A second one is to generalize behavior analysis and emotion representation to a WOS (Wise Object System): this requires knowledge aggregation to extract relevant information on the whole system starting from individual WOs.

In the present version of WOF, intrusiveness in application source code is limited to the inheritance relationship and two warts: the WO methods methodInvocate() and methodInvocated() that must be called at the beginning and the end of an application method. Regarding this issue, we envisage different solutions in the next version of WOF: (a) add dynamic Java code on-the-fly at runtime; (b) use Aspect Oriented Programming [**?**]; (c) use dynamic proxy classes.

We are convinced that *wise systems* are a promising approach to help humans integrate new technologies both in their daily life as end-users and in development processes as system developers. We use home automation to illustrate our work results but those can also apply to other domains like health that heavily rely on human expertise. Authors in [**?**] and [**?**] propose interactive Machine Learning (iML) to solve computationally hard problems. With regard to this, WOF puts the "human-in-the-loop" in two cases: when defining ECA rules to connect distributed WOs and when validating capability-related knowledge constructed by WOs.

To validate our approach, we recently initiated a new internal project called COMDA whose aim is to study and test our research ideas on a real system. The latter consists of a set of connected objects (sofa, chairs, etc.) to identify unusual situations like "no life sign in the living-room this morning" which is crucial in the domain of person ageing in place.

References