A Wise Object Framework for Distributed Intelligent Adaptive Systems

Ilham Alloui, Flavien Vernier

A Wise Object Framework for Distributed Intelligent Adaptive Systems

Ilham Alloui[1], Flavien Vernier[1]
[1] *Univ. Savoie Mont Blanc, LISTIC, F-74000 Annecy, France*
{*ilham.alloui, flavien.vernier*}*@univ-smb.fr*

Keywords: Object oriented design, Architecture models, Adaptive systems, Introspection, Decentralized control

Abstract: Designing Intelligent Adaptive Distributed Systems is an open research issue addressing nowadays technologies such as Communicating Objects (COT) and the Internet of Things (IoT) that increasingly contribute to our daily life (mobile phones, computers, home automation, etc.). Complexity and sophistication of those systems make them hard to understand and to master by human users, in particular end-users and developers. Those are very often involved in learning processes that capture all their attention while being of little interest for them. To alleviate human interaction with such systems and help developers to produce them, we propose WOF, an object oriented framework founded on the concept of *Wise Object* (WO). A WO is a software-based entity that is able to learn on itself and also on the others (e.g. its environment). *Wisdom* refers to the experience (on its own behavior and on the usage done of it) such object acquires by its own during its life. In the paper, we present the WOF conceptual architecture and the Java implementation we built from it. Requirements and design principles of wise systems are presented. To provide application (e.g. home automation system) developers with relevant support, we designed WOF with the minimum intrusion in the application source code. The adaptiveness, intelligence and distribution related mechanisms defined in WOF are 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 then a *Wise Object System* (WOS) composed of *wise objects* that interact through an event bus according to *publish-subscribe* design pattern.

1 Introduction

Designing Intelligent Adaptive Distributed Systems is an open research issue addressing nowadays technologies such as Communicating Objects (COT) and the Internet of Things (IoT) that increasingly contribute to our daily life (mobile phones, computers, home automation, etc.). Complexity and sophistication of those systems make them hard to understand and to master by human users, in particular end-users and developers. Those are very often involved in learning processes that capture all their attention while being of little interest for them.

New usages, amounts of information, multiplicity of users, heterogeneity, decentralization, dynamic execution environments result in new system design requirements: new technologies should adapt to users more than users should do to technologies. If we take home automation systems for example, both end-users and system developers face problems:

- end-users encounter at least the following problems: instructions accompanying the devices are too complex and it is hard for non-expert users to master the whole behavior and capabilities provided by 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). Information needed by users is not necessarily the same from one to another. 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 of home automation systems lack software support for building such systems, in particular self-adaptation mechanisms are not mature yet: most existing approaches are either domain-specific or too abstract to be helpful as stated in (Abuseta and Swesi, 2015).

To illustrate our purposes, all along the paper we use a simple example in home automation domain. Let us consider a system composed of a roller shutter (actuator) and a control key composed of two buttons (sensors). In the very general case and in a manual mode, with a one-button control key, a person uses

the button to: 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 dedicated to easily construct adaptive systems: when provided, such support is limited to an application domain and cannot be reused with a minimum of constraints. Generally, adaptation mechanisms and intelligence are merged with the application objects which make them difficult and costly to reuse in another application or another domain.

To alleviate human interaction with such systems and help developers to produce them, we propose WOF, an object oriented framework founded on the concept of *Wise Object* (WO) (Alloui et al., 2015). A WO is a software-based entity that is able to learn on itself and also on the others (e.g. its environment). *Wisdom* refers to the experience (on its own behavior and on the usage done of it) such object acquires by its own during its life. According to this approach, "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 the "wisdom" and intelligence logic (we name *abilities*) of the objects from application services (we name *capabilities*) they are intended to render.

To provide application (e.g. home automation system) developers with relevant support, we designed WOF with the minimum intrusion in the application source code. The adaptiveness, intelligence and distribution related mechanisms defined in WOF are inherited by application objects. We realized a Java implementation of WOF and validate it on a home automation example.

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 nowadays self-adaptive systems. Then we present design principles and fundamental concepts underlying WOF in Section 3. In Section 4 we detail the structure and behavior of a WO and a WO System (WOS). To illustrate how to use the WOF, we give an example in the Home automation domain in Section 5. Finally, in Section 6 we discuss our approach and conclude with ongoing work and some perspectives.

2 Requirements

A system based on new technologies should be able at runtime to: (1) *know by itself on itself*, i.e. to learn how it behaves, to consequently reduce the understanding effort needed by end-users (even experimented ones); (2) *know by itself on its usage* to adapt to users according to the way and to the context it is used in. In addition like any service-based system (3) such system should be capable of improving the quality of services it is offering. WOF aims at producing such systems while meeting those end-user related 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 in a sense contributes to *calm-technology* (Weiser and Brown, 1996) that *describes a state of technological maturity where a user's primary task is not computing, but being human*. As claimed in (Amber, 2010), 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 (Weiser and Brown, 1996), it is more than ever, 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 lives its life and reacts differently based on its experience (knowledge). Typically a difference in temperature that is considered as normal outside (e.g. 5 degrees) may be considered as significant inside. This means that collecting and analyzing contextual information is also a key requirement for those systems. Another situation is when unexpected behavior occurs: 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 consequently (e.g. raising an alert to the end-user);

- *Requirement 3*: In an ideal world, a system 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. depending on non-functional criteria taken into account in the quality of service support (Bass et al., 1998).

Many approaches are proposed to design and develop the kinds of systems we target: multi-agent systems (Wooldridge, 2009), intelligent systems (Roventa and Spircu, 2008), adaptive systems (Salehie and Tahvildari, 2009), self-X systems (Huebscher and McCann, 2008). In all those approaches, a system entity (or agent) is able to learn on its environment (including other entities) through its interactions. Our intention is to go a step forward by enhancing a system entity with the capability of learning by its own on the way it has been designed to behave. We see at least two benefits to this: (a) a decentralized control: as each entity evolves independently from the others, it can control actions to perform at its level according to the current situation; (b) each entity can improve its performance and then the performance of the whole system.

On another hand, 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 (IBM, 2005) 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: authors in (Brun et al., 2013) proposed a *design space* as a general guide for developers to take decisions when designing self-adaptive systems. In (Abuseta and Swesi, 2015), authors propose design patterns for self-adaptive systems where roles and interactions of MAPE-K components are explicitly defined. Our primary aim is to offer developers an object oriented concrete architecture support, ready to use for constructing wise systems. We view this as complementary to 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 "wisdom" and intelligence logic (we name *abilities*) of the objects 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 (IBM, 2005).

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. When human better succeed in observing the others, a *Wise Object* would have more facilities to observe itself by introspection. A *Wise Object* is intended to "connect" to either a physical entity/device (e.g. a vacuum cleaner) or a logical entity (e.g. software component) (see Figure 1). 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.).

A WO is thus characterized by:

- 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

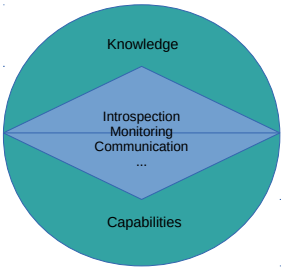
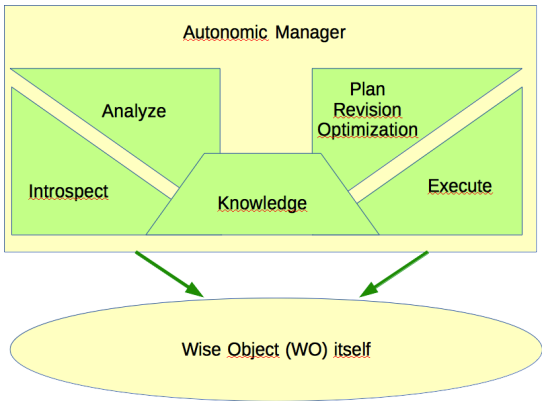


Figure 1: Conceptual structure of a WO

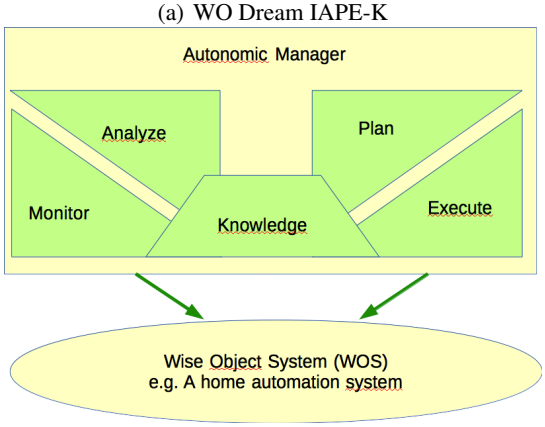
when it is delivering a service to an end-user or answering an action request from the environment. The WO then monitors such execution and usage done with application services it is responsible for. We use the word *Dream* as a metaphor for a state where services invoked by the WO do not have any impact on the real system: this functions as if the WO is disconnected from the application device/component it is related to.

To ensure adaptiveness, each WO incorporates mechanisms that allow it to perform a kind of MAPE-K loops (IBM, 2005). *Dream* and *Awake* MAPE-K are respectively depicted by Figure 2(a) and Figure 2(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 (experience). WO constructs its experience gradually, along the dream states. This means that WO knowledge is not necessarily complete and is subject to revisions. Revision actions may relate to adaptation, for instance recording a new behavior, or to optimization like creating a shortening of a list of actions 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, the WO is able to communicate an *emotion* if necessary. We define a WO emotion as a distance between a current usage of its services and its common usage (usual one). With this metaphor, a WO can be stressed if one of its services is more frequently



(a) WO Dream IAPE-K



(b) WO Awake MAPE-K

Figure 2: WO MAPE-Ks

used or conversely, a WO can be bored. It can be surprised if one of its services is used while it has never been used before. WO emotions are intended to be used as a new information by the system (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 an alert, opening the windows and doors, cutting off the 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* Section 4 to carry out

communication and coordination among WOs. This is close to the *Implicit Information Sharing Pattern* introduced in (Weyns et al., 2013).

4 Design models of WO and WOS

WOF is an object oriented framework build 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 3 shows the UML Class diagram of 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 subclasses. Knowledge of a WO is both a capability-related knowledge model and a usage-related knowledge model. In our present experiment, we have chosen a graph-based representation for knowledge on WO capabilities (i.e. application services) and its usage. Knowledge on WO capabilities is stored in a state-transition graph while that on WO usage is stored in a Markov graph where usage-related statistics are maintained. The Markov diagram clearly depends on the usage of an Object (a WO instance). It is important to note that even if initially, the state diagram depends only on a class (WO subclasses), according to its usage an object (WO subclass instance) can add transitions into its state diagram. Therefore, the state diagram is considered as an instantce attribute rather than a class attribute.

Let us recall that WO behavior is split into two states. The dream state and the awake state, see Figure 4. 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 its state diagram – its capability knowledge – the WO executes the methods of its sub-class (i.e. an application class) to know the effect on the attributes of this sub-class. Each set of attribute values produces a state in the diagram and a method invocation produces a transition. The main constraint in this step is that the method invocation must have no effect on other objects of the application when the WO is dreaming. This is solved thanks to the system architecture described in Section 4.2.

Regarding knowldge on an application object usage, two kind of situations are studied: emotions and adaptation of behavior.

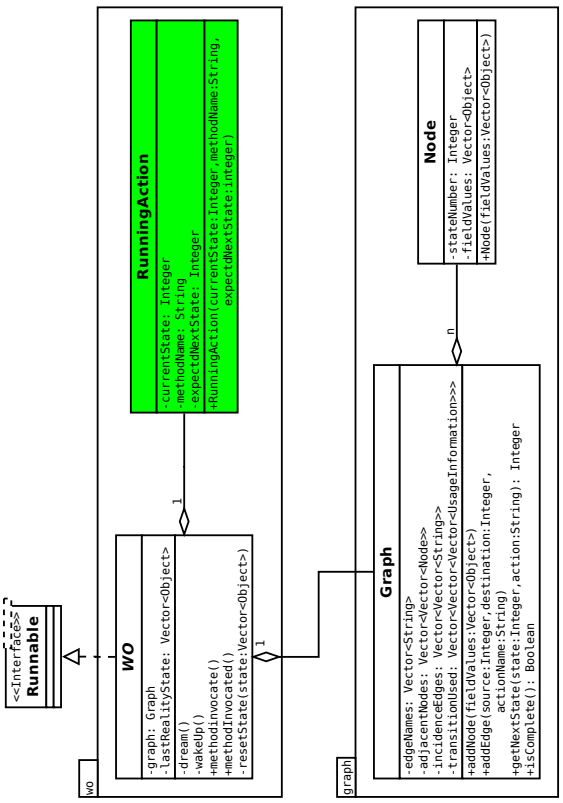
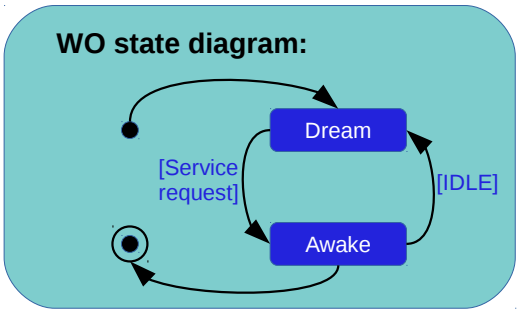


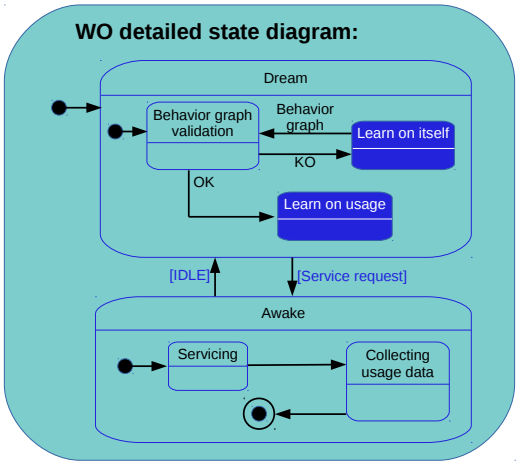
Figure 3: Class diagram of a WO

As introduced in 3, an emotion of WO is a distance between its current usage and its common usage (i.e. unusual 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. Emotions of WO are a projection of its current usage with regard to its common usage. When a WO expresses an emotion, this information is catched by the WOS 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 in the WO system.

A WO uses its state diagram to compute a path from a current state to a known state (Moreaux et al.,



(a) WO short state diagram



(b) WO detailed state diagram

Figure 4: WO state diagram

2012). According to the frequency of the paths used, a WO can adapt its behavior. For instance, if a path is often used between non-adjacent states, the WO can build a shortcut transition between the initial and destination states and then build the corresponding method within its subclass instance (application objet). This consequently modifies the state diagram 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 Section 3. Let us recall that in the dream state, a WO can invoke its own methods to build its state diagram, 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 for another object, it reconnects to the bus, goes out from the Dream state to enter into Awake state and serves the request.

Figure 5 shows the Class diagram of a WO system based on the bus. This model is simplified and highlights the main classes that compose a WO bus system. The system uses an Event/Action mechanism for WOs' interactions. On an event, a state change occurs in WO, an action may be triggered on another WO. These peers "Event/Action" are defined 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 its state changes during its 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 bus according to *publish-subscribe* design pattern.

5 An illustrating example "Home automation"

The concept of WO has many scopes of application. It can be used to adapt an application to its en-

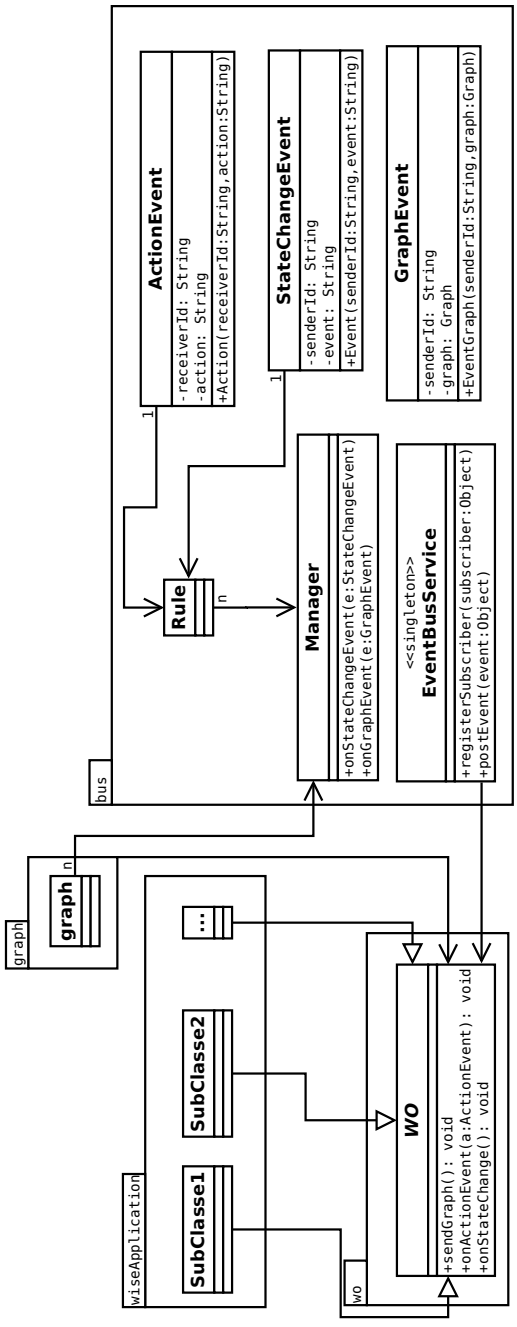


Figure 5: Class diagram of a WO system based on a bus

vironment, to monitor an application from inside, to manage an application according to its uses... 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 a switch and a shutter. The switch is modeled by 2 states “on” and “off” and 3 transitions “on()” , “off()” and “switch()”.

Listing 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 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 will 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 its 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 6. Log 2 gives Markov graph logging representation. Let us note that the Markov graph representation hides time-related information as it is based on 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 1 10 first event log stored on each WO.

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
13243	20015	13244	20017
...	...	...	...

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 instance detects 6 states and 10 tran-

Log 2 Switch log after 1000 iterations.

```
*****************************************
Switch

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
*****************************************
```

sitions (green values of adjacency matrix). Consequently, it has not detected all the possible transitions yet. This incomplete knowledge is not a problem,

Log 3 Rolling shutter log after the 2nd iteration.

```
*****************************************
RollingShutter

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
*****************************************
```

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).

Log 4 Rolling shutter log after the last iteration.

```
*****************************************
RollingShutter

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
*****************************************
```

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. State diagrams and usage logging are the knowledge base for WOs. We discus about management and use of this knowledge in Section 6.

6 Discussion and concluding remarks

Our current research addresses the problem of how to design self-adaptive intelligent systems that limit the involvement of their users and their developers to what is necessary. We designed an object oriented framework (WOF) with the concept of *Wise Object* (WO) as the building block of such systems. As proof of concept, we started developing a Java framework for implementing these kinds of systems

with the minimum intrusion in the application code. Object classes produced by a developer inherit the behavior of *Wise Object* (WO) class. An instantiated system is then a *wise system* composed of *wise objects* that interact through an event bus according to the *publish subscribe* design pattern. We believe that *wise systems* is a promising approach to help humans serenely integrate new technologies both in their daily life as end-users and in development processes as system developers.

In our first experiment, we limited intrusion to the inheritance and two warts: the WO methods methodInvocate() and methodInvocated() that must be called at the beginning and the end of application methods. Different ways can be used to go further and remove those warts. A first way is to add dynamic java code on-the-fly at runtime. A second one is to use dynamic proxy classes. A third one is based on Aspect Oriented Programming (Kiczales et al., 1997). Those solutions solve the problem and we plan to implement them in future versions of WOF. The objective is to restrict the intrusion to the inheritance relationship between the WO Class and application classes.

In this paper, we focused in particular on adaptation, monitoring and communication mechanisms and showed how *Wise Object* behave according to their experience. There are still many research issues we are investigating: A first issue to handle is how to represent the knowledge in a *Wise Object*. State diagram and Markov graphs are used in our first approach, but other approaches like ontologies are envisaged. Knowledge, specific to each *Wise Object*, represents an amount of information that can be big but not necessarily relevant. A second issue relates then to knowledge aggregation by *Wise Object* so that they can extract relevant information to the whole system. This issue may involve techniques from information fusion approaches, multi-criterion scales and fuzzy modeling. Knowledge aggregation allows us to represent *emotion* of a *Wise Object*, namely the distance from its current behavior to its usual behavior (*surprise*, *stress*, etc.). A last issue is related to the use of aggregated knowledge within the system during its execution. This is typically a problem of information fusion. The goal is to generate a (sub-)system knowledge/emotion from the knowledge/emotion translated by the *Wise Objects*.

REFERENCES

Abuseta, Y. and Swesi, K. (2015). Design patterns for self adaptive systems engineering. *CoRR*, abs/1508.01330.

Alloui, I., Esale, D., and Vernier, F. (2015). Wise Objects for Calm Technology. In *10th International Conference on Software Engineering and Applications (ICSOFT-EA 2015)*, ICSOFT-EA 2015, pages 468–471, Colmar, France. SciTePress 2015.

Amber, C. (2010). Amber case 2011, we are all cyborgs now ted talk.

Bass, L., Clements, P., and Kazman, R. (c1998.). *Software architecture in practice /*. Addison-Wesley,, Reading, Mass. :. Online version: Bass, Len. Software architecture in practice. Reading, Mass. : Addison-Wesley, c1998 (OCoLC)605442178 Online version: Bass, Len. Software architecture in practice.

Brun, Y., Desmarais, R., Geihs, K., Litoiu, M., Lopes, A., Shaw, M., and Smit, M. (2013). A design space for self-adaptive systems. In de Lemos, R., Giese, H., Müller, H. A., and Shaw, M., editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 33–50, Dagstuhl Castle, Germany. Springer.

Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28.

IBM (2005). An architectural blueprint for autonomic computing. Technical report, IBM.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP*, pages 220–242.

Moreaux, P., Sartor, F., and Vernier, F. (2012). An effective approach for home services management. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 47–51, Garching. IEEE.

Roventa, E. and Spircu, T. (2008). *Management of Knowledge Imperfection in Building Intelligent Systems*. Studies in Fuzziness and Soft Computing. Springer Berlin Heidelberg.

Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42.

Weiser, M. and Brown, J. S. (1996). Designing calm technology. *In PowerGrid Journal*, 1.01.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., and Goeschka, K. (2013). On Patterns for Decentralized Control in Self-Adaptive Systems. In de Lemos, R., Giese, H., Müller, H., and Shaw, M., editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*, pages 76–107. Springer.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition.