Context-aware Self Adapting Systems: a Ground for the Cooperation of Data, Software, and Services

Fabio A. Schreiber
and
Emanuele Panigati
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano - Via Ponzio, 34/5 - 20133 Milan, Italy
{fabio.schreiber, emanuele.panigati}@polimi.it

Modern technologies related to Pervasive and Ubiquitous computing require software to self adapt to different operating environments and situations, in a word, to different contexts. Several approaches have been proposed to solve this problem both in the Data Management and in the Programming Languages communities, however these efforts have proceeded along separate paths with little, if no interaction.

We claim that there are complementary features which can bring different paradigms in the Data Management and Programming Languages domains to a fruitful cooperation in building Adaptive Systems. In fact, data collected by sensor networks can be directly used by application programs as well as used to determine the context the application is working in, so attaining a context-aware behavior obtained by triggering the execution of specific program modules or the connection to relevant web services.

In this paper, we use the PerLa pervasive data management language and JCop Context-Oriented Programming Java language extension to show the feasibility of this approach applied to the classical case of keeping an office room climate comfortable under several environmental constraints and to the management of a ski resort.

Keywords: Adaptive systems, Pervasive Data Management, Context awareness, Ubiquitous computing, Heterogeneous sensor networks, Middleware infrastructure, PerLa, Context Oriented Programming paradigms.

1. INTRODUCTION

Pervasive Computing is becoming one of the most wide-spread computing technologies. Today, the majority of computationally capable devices surround us hidden in everyday life objects and, with the advent of the Internet-of-Things (IoT), this trend will dramatically increase (Borgia, 2014), (Editorial, 2007), (Chui, Loffler, and Roberts, 2010). Among the properties of a Pervasive Computing System (PCS) (ubiquity, mobility, heterogeneity, ...) *self adaptation is a must*. In order to achieve self adaptation, a system must know the state of the environment, the state of the system itself and the behavior it must exhibit in this environment; changes in the environment entail changes in the system behavior, possibly causing actions on the environment itself.

The environment can be constituted by physical objects or variables or by human-generated input, including data generated by social networks: we shall refer to it as the PCS *working context*. The concept of *context*, describing the "situation of every entity in the environment" (Dey, 2001), becomes a first-class citizen since it allows to *tailor* the available data for the user, reducing the "information noise"(Bolchini, Curino, Orsi, Quintarelli, Rossato, Schreiber, and Tanca, 2009). Furthermore, this perspective opens the way to a *proactive* interaction with the environment, enabling the possibility of enacting, for each particular user in a particular situation, the actions to be performed by the PCS, possibly by means of intelligent actuators. In a physical system, the

This work was partially funded by the European Commission, Programme IDEAS ERC, Project 227977-SMSCom and Industria 2015, Programma n° MI01_00091 SENSORI.

data values defining a context can be obtained by sensors (*numerical observables*) and translated to the *symbolic observables* required by the specific context model (Coutaz, Crowley, Dobson, and Garlan, 2005)

As we show in Section 5, there are several projects that create pervasive systems environments, but there isn't any standardized, integrated framework out there for developing pervasive applications (Cheng, 2009), therefore work has still to be done for engineering such systems. In our previous work (Schreiber, Tanca, Camplani, and Viganó, 2012) we started focusing on this goal, using sensors to characterize and discover possible contexts, and defining suitable actions accordingly.

Like every computer system, besides the hardware resources, PCSs are constituted of software - developed in some programming language - and of data - managed by some data management system. Both the Programming Languages (PL) and the Data Management (DM) communities have developed *context-aware* paradigms and techniques following parallel lines, with little or no interaction between them. Taking the hint from a famous statement about the relationship between science and philosophy by Kant (Kant, 1787), we argue that *Programs without Data are empty, Data without Programs are blind*; therefore a convergence between the two fields is strongly needed.

Scientists and engineers have made significant efforts to design and develop systems able to adapt their behavior according to specific changes. These systems address adaptivity in various respects, including performance, security, fault management, control theory, etc. (IBM, 2006), (Diao, Hellerstein, Parekh, Griffith, Kaiser, and Phung, 2005). While adaptive systems are used in a number of different areas, software engineers focus on their application in the software domain, called *self-adaptive software*.

Software adaptivity requires that a system be able to modify itself based on observations or occurred events (*Intercession* property) (McKinley, Sadjadi, Kasten, and Cheng, 2004). This property is strictly connected to the basic idea of context. An adaptation mechanism is expected to trace system changes and to take appropriate actions according to the designed rules and this aim can be achieved through monitoring all the entities that can directly affect the behavior of the whole software system or a group of modules.

Summarizing, adaptive systems can be defined as follows (Cheng, 2009): "Adaptive systems are able to adjust their behavior in response to their perception of the environment and of the system itself".

Furthermore, a system able to retrieve data, to build and manage contexts and to properly reason on them, is called **context-aware**. In (Pascoe, 1998) a context-aware system is defined as follows: "A system that provides services or information to the users according to the context".

In context-aware systems context encompasses the following pieces of the whole environment in which it operates:

- *Computing environment*: available processors, devices accessible for user input and display, network capacity, connectivity and costs of computing.
- *User environment*: location, collection of nearby people, and social situation.
- *Physical environment*: all external phenomena relevant to the system.

Adaptivity and context-awareness are strictly related to each other and in many real situations are even interchangeable. However, context-awareness is more related to "information tailoring", i.e., it refers to the ability of the system to know exactly at any time the current contextual information and to provide it when required (Bolchini et al., 2009), while adaptivity refers to the execution of behavioral variations in response to changes of the entities that can affect the behavior of the system, also the internal software itself. Therefore adaptivity and context-awareness are complementary in building applications for PCS.

In this paper we strive for showing how combining the PerLa pervasive data management language and the Context-Oriented Programming (COP) paradigm makes the implementation of

context-aware and adaptive pervasive systems possible. We consider the classical application of keeping an office room climate comfortable under several environmental constraints as a running example.

The remainder of the paper is organized as follows: in Section 2, we introduce the background material on context management from the data as well as from the programming languages perspectives; in Section 3 we present our proposal for the integration of the Data Management and Programming Languages views; in Section 4 our approach is applied to the management of a ski resort; in Section 5 projects addressing similar issues are briefly surveyed; finally, we discuss the conclusions (Section 6).

2. BACKGROUND

As a first step toward the integration of data, services and programs into a general purpose development framework for adaptive pervasive systems, the PerLa system, designed for managing data in Wireless Sensor Networks (WSN) (Schreiber, Camplani, Fortunato, Marelli, and Rota, 2012), has been extended with the ability of declaring and managing contexts (Schreiber et al., 2012), thus allowing to apply context-awareness to generic system operations. Moreover, as shown in detail in Section 2.1.4, since the number of possible contexts can rapidly grow with the complexity of the application, the design phase is also supported by a tool that leverages the possibility to speed-up and modularize the definition of the data and operations associated with each specific context. In the following sections we shall discuss how operations can be expressed as *Layers* in a Context Oriented Programming language (Salvaneschi, Ghezzi, and Pradella, 2012a) and as calls to web services.

2.1 Context: the Data Management view

Reduction of the retrieved data volume and of the *information noise* have been the main reasons for adopting context-aware data management techniques in Pervasive Systems. A survey of some context models which have been proposed in the literature is presented by Bolchini et Al. in (Bolchini, Curino, Quintarelli, Schreiber, and Tanca, 2007); in this section we briefly introduce the context model as well as the language and the framework which we defined to specify the actual context and the consequent actions.

2.1.1 *Context Dimension Tree (CDT).* The Context Dimension Tree (CDT) (Bolchini et al., 2009), (Bolchini, Quintarelli, and Tanca, 2013) has been proposed as a model to represent context at a conceptual level and to derive the actual contexts definitions. According to the CDT model, the set of possible contexts of the environment can be modeled as a labeled tree composed of *dimension* (black) and *concept* (white) nodes. The former are used to capture the different characteristics of the environment, while the latter are used to represent the admissible values that can be assumed by each dimension. Both dimensions and concepts can be semantically enriched using attributes (square nodes) that are parameters whose values are provided at run-time. Moreover, the tree term suggests that the designer can model the environment using the preferred granularity, nesting more than one level of dimensions with the unique restriction that every dimension can only have concept children and vice versa. This constraint imposes that black and white nodes alternate while descending the tree, as in Figure 1, where the CDT of our running example is shown, containing the fundamental aspects of context in the office environmental condition management scenario. In this scenario sensors are located (`Location` dimension) in offices or meeting rooms, and they monitor several parameters, like the presence of smoke (`Smoke` dimension), the humidity (`Humidity` dimension) and others, in order to avoid the rise of stressful working conditions (e.g., overheating of the offices) or of dangerous situations (e.g., fire).

In order to denote that a dimension has assumed a certain value we use the $\langle Dimension = Value \rangle$ notation, called a **context element**. A context C can then be formalized as the conjunction of one or more context elements: $C \equiv \bigwedge_i \langle Dimension_i = Value_{i_j} \rangle$. It is worth noticing that

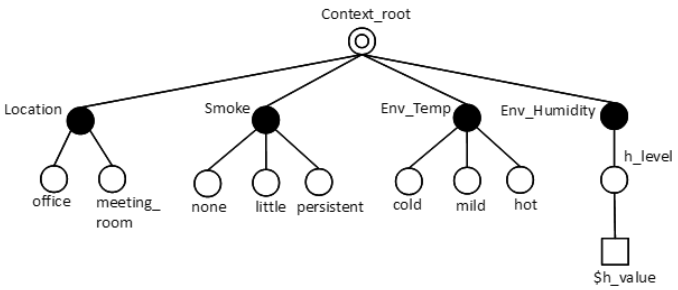

Fig. 1. Context Dimension Tree (CDT)

not all possible subtrees are valid contexts. Consider for instance the three possible values of the dimension `Env_Temp` in Figure 1: `cold`, `mild` and `hot`; being children of one dimension, they are always to be instantiated in mutual exclusion. The designer can specify further constraints, like the *useless context constraints*, forbidding some context elements to be used in the same context definition (Bolchini et al., 2009).

2.1.2 *A Pervasive Data Management Framework.* As extensively presented in (Schreiber et al., 2012) PerLa is a framework to configure and manage modern pervasive systems and, in particular, wireless sensor networks. PerLa adopts the database metaphor of the pervasive system: such approach, already adopted in the literature (Madden, Franklin, Hellerstein, and Hong, 2005), is data-centric and relies on a SQL-like query language. PerLa queries allow to retrieve data from the pervasive system, to prescribe how the gathered data have to be processed and stored and to specify the behavior of the devices. The most relevant query types supported by PerLa are: i) *Low Level Queries (LLQ)*, which define the behavior of every single or of a homogeneous group of nodes, and specify the data selection criteria, the sampling frequency and the computation to be performed on sampled data; ii) *Actuation Queries (AQ)*, which provide the mechanisms to change parameters of the devices or to send commands to actuators. A SQL-like high level interface is also provided towards standard Data Stream Management Systems. The other fundamental component of PerLa is a middleware whose architecture exposes two main interfaces: a high-level interface, which allows query injection and a low-level interface that provides plug&play mechanisms to seamlessly add new devices and support energy savings. All nodes in the sensing network are abstracted by the PerLa middleware as logical devices called *Functionality Proxy Component* (FPC). The FPCs have common and homogeneous interfaces, and are used by PerLa queries to access the data gathered from the network nodes. No knowledge of the nodes hardware and computational characteristics is needed to perform a PerLa query. Moreover, thanks to the FPC abstraction, the language is not tied to any particular type of sensing device so allowing to gather data also from sources other than physical devices, such as social networks.

2.1.3 *Embedding context into PerLa.* In order to push the knowledge of context from the application down to the middleware level, in a previous work (Schreiber et al., 2012), we designed and implemented:

– an extension of the existing PerLa language syntax, called **Context Language (CL)**, in order to declare, inside PerLa, the CDT, the contexts as well as the actions to be performed accordingly;
– the **Context Manager (CM)**, able to maintain and manage the declared CDT, detect active contexts and perform the desired actions;

The syntax of the CL has been divided into two parts, called *CDT Declaration* and *Context Creation*, both presented in details in (Schreiber et al., 2012).

Listing 1: The Context Dimension Tree

```
CREATE DIMENSION Location
        CREATE CONCEPT office
                WHEN location = 'office'
        CREATE CONCEPT meeting_room
                WHEN location = 'meeting_room'
CREATE DIMENSION Smoke
        CREATE CONCEPT none
                WHEN smoke < 0.4
        CREATE CONCEPT little
                WHEN smoke >= 0.4 AND smoke <= 1
        CREATE CONCEPT persistent
                WHEN smoke > 1
CREATE DIMENSION Env_Temp
        CREATE CONCEPT cold
                WHEN temperature < 18
        CREATE CONCEPT mild
                WHEN temperature >= 18 AND temperature < 24
        CREATE CONCEPT hot
                WHEN temperature >= 24
CREATE DIMENSION Humidity
        CREATE CONCEPT h_level
                CREATE ATTRIBUTE $h_value
```

CDT Declaration. :

This part allows the user to specify the CDT. A set of *CREATE DIMENSION/CONCEPT* statements allows to declare the dimensions as well as their concept nodes. When creating a concept of a dimension, the designer must specify the name and the condition for assuming the specified values by means of numeric observables that can be measured from the environment (*WHEN* clause). When the design requires the presence of attributes, the *CREATE ATTRIBUTE* clause must be used, using the $ sign as a prefix before the name of the attribute, meaning that its value will be supplied by the application at runtime.

As an example of its usage we report in Listing 1 the syntax to completely define the CDT presented in Fig. 1. The CDT defines four dimensions: `Location`, that can assume '*office*' and '*meeting_room*' values; `Smoke`, `Env_Temp` and `Env_Humidity`, that represent the actual condition of the environment. Notice that, while `Smoke` and `Env_Temp` are quantified by symbolic observables, `Env_Humidity` values are directly used as numerical observables provided by sensor readings.

Labels of the fields in the `WHEN` clause (e.g. *location*, *smoke*, *humidity*, ...) refer to external input data (like data coming from the sensors or other kind of input) while the labels in the *CREATE DIMENSION* or *CREATE CONCEPT* clauses refer to the labels of the CDT nodes.

Context Creation. :

Listing 2 shows the part of the syntax which allows the designer to declare a context on a defined CDT and control its activation by defining a **contextual block**, which is composed by four **components**:

- **ACTIVATION component**: allows the designer to declare a context, using the *CREATE CONTEXT* clause and associating a name to it. The *ACTIVE IF* statement is used to translate the $Context \equiv \bigwedge_{i,j}(Dimension_j = Value_i)$ statement into PerLa.
- **ENABLE component**: introduced by the *ON ENABLE* clause, allows to express the actions that must be performed when a context is recognized as active;
- **DISABLE component**: introduced by the *ON DISABLE* clause is the counterpart of the previous one, allowing to choose the actions, if any, to be performed when the declared context is no longer active;
- **REFRESH component**: instructs the middleware on how often the state of the context variables in the *ACTIVE IF* statement is to be checked.

In Listing 2 we report the declarations for two possible contexts: the first one represents

Listing 2: Examples of contexts in PerLa

```
CREATE CONTEXT normal
ACTIVE IF Env_Temp = 'mild' AND Humidity.h_level >= 40
        AND Humidity.h_level <= 65
        AND Smoke = 'none'
ON ENABLE:
   SELECT humidity, temperature
   SAMPLING EVERY 1m
   EXECUTE IF location = 'office'
ON DISABLE:
   DROP normal
REFRESH EVERY 5m

CREATE CONTEXT fire
ACTIVE IF Env_Temp = 'hot' AND Smoke = 'persistent'
ON ENABLE:
   SET alarm = TRUE
ON DISABLE:
   DROP fire
   SET alarm = FALSE
REFRESH EVERY 5m
```

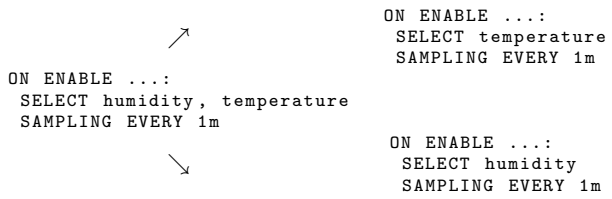

Fig. 2. Partial (ENABLE) components definition

the "normal" situation, in which the environment has comfortable values of temperature and humidity, while the second context represents the rise of a possible dangerous situation (a fire alarm).

2.1.4 *Contextual Block Composition.* In the previous sections we mentioned how a growing tree depth of the CDT allows the designer to capture the aspects of the environment with different granularities, since more dimensions (and thus concepts) allow to express more possible contexts.

The number of possible contexts that can be generated from a complete CDT, with a single dimension layer having D nodes and N concept nodes each, amounts to $\prod_{i=1}^{D} N_i$. Even if many of these contexts can be meaningless, nevertheless the designer is charged with the hard task of declaring: i) every single context and ii) a set of actions for each one of them; so her/his task becomes rapidly unfeasible. In the following we show the possibility of relieving the designer from this cumbersome task, enabling the middleware to automatically build the contextual block starting from the contextual block components (Schreiber et al., 2012).

Partial components. :

The syntax of the PerLa language allows to separate the block components into one or more *partials*, as shown in Figure 2. A *partial* contains a subset of the statements and clauses included in the original block, with the only constraint that this subset must be valid from the point of view of the PerLa QL syntax. This division is particularly meaningful for the ENABLE and DISABLE components; the only block that can not be divided is the ACTIVATION block since it deals with the definition of the context itself.

Automatic composition. :

With the introduction of partials, the concepts behind the automatic composition of contextual blocks can be described. The main idea, already adopted in (Bolchini et al., 2013) for the tailoring of data, is illustrated in Figure 3. The designer must only associate one or more *partials* with each context element of the CDT. When the system has to compose a contextual block, it starts

from the partials associated with the context elements which are part of the context and combines them by means of a generic operator, represented here by the symbol $\oplus$.

The association of the partials with the CDT context elements can be performed using the syntax in Listing 3, which enriches the CDT declaration section of the CL.

Listing 3: Modified CL syntax to support partial components

```
CREATE DIMENSION <Dimension_Name>
 [CHILD OF <Parent_Node>]
 [CREATE ATTRIBUTE $<Attribute_Name>]* |
 {CREATE CONCEPT <Concept_Name>
    [WHEN<condition>] [EVALUATED ON <"Low_Level_Query">]
  [WITH ENABLE COMPONENT: <"PerLa_Query">]
  [WITH DISABLE COMPONENT: <"PerLa_Query">]
  [WITH REFRESH COMPONENT: <Period>]
  [CREATE ATTRIBUTE $<Attribute_Name>
  [EVALUATED ON <"Low_Level_Query">]]*}*
```

The *WITH ENABLE COMPONENT* clause may contain any query expressed using PerLa. The same holds for the *WITH DISABLE COMPONENT* clause. The last clause (*WITH REFRESH COMPONENT*) allows to specify the time period (always using PerLa's syntax) to be used. Finally the composition can be carried out both at design and at run-time (Schreiber et al., 2012).

When the association phase is complete and before the system is put into an operational state, it is possible to combinatorially generate all the possible contexts that are defined by the CDT and that are not forbidden by the constraints. For each possible context the relative contextual block is then automatically generated composing the partials associated in the previous phase. In Algorithm 1 the composition algorithm is shown in pseudo-code (Schreiber et al., 2012).

This algorithm, as its first step, retrieves all the relative context elements, i.e., the couples $\langle Dimension_i = Value_j \rangle$ for all possible contexts (**getContextElements()** function, Algorithm 1 line 4).

With these context elements, the CM exploits three functions[1] in order to retrieve the partial components associated with every context element retrieved at the previous step (**getEnable Components()**, **getDisableComponents()** and **getRefreshComponents()** functions, Algorithm 1 line 6-8). When all these inputs have been retrieved a **composeBlock()** function is invoked. This function firstly creates an empty contextual block. All the retrieved partial components are chained (*attach()* function) to the empty block, as from the code in Algorithm 1, lines

[1]Algorithm 1 reports only the ENABLE function, the other two being identical from an operational point of view; the only difference is in the query statement block (ON ENABLE, ON DISABLE or REFRESH) on which they operate.

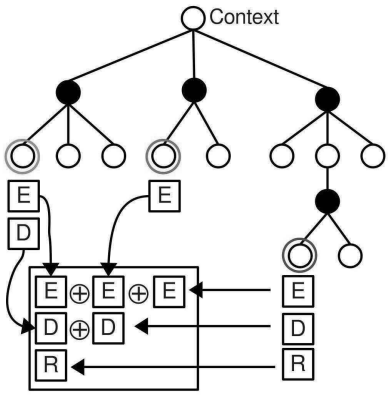

Fig. 3. Contextual block composition $\oplus_D$

```
Input  : The C set of all possible contexts
Output: BS set with the composed contextual blocks

1  BS=∅;
2  for (context c_i ∈ C) do
3  |  /*Context elements retrieval*/ ;
4  |  CE[ ] ← getContextElements (c_i);
5  |  /*Components retrieval*/ ;
6  |  E[ ] ← getEnableComponents (CE[ ]);
7  |  D[ ] ← getDisableComponents (CE[ ]);
8  |  R[ ] ← getRefreshComponents (CE[ ]);
9  |  B_i ← composeBlock (E[ ],D[ ],R[ ]);
10 |  optimizeBlock (B_i);
11 |  if (parseBlock (B_i)=='OK') then
12 |  |  BS = BS ∪{B_i};
13 |  end
14 |  else
15 |  |  return WARNING('Parse Error')
16 |  end
17 end
18 return BS;
19 Procedure composeBlock(E[ ],D[ ],R[ ]) ;
20 B = ∅;
21 for (enable comp. e ∈ E[ ], disable comp. d ∈ D[ ] ) do
22 |  attach (B.E, e);
23 |  attach (B.D, d);
24 end
25 B.R = min(R[ ])
26 return B;
27 /*Identical for Disable and Refresh*/ ;
28 Procedure getEnableComponents(CE[ ]) ;
29 EB = ∅;
30 i = 0;
31 for (ce ∈ CE) do
32 |  if (Context_Enable_Rel(ce) ≠ ∅ ) then
33 |  |  EB[i] = Context_Enable_Rel(ce);
34 |  |  i++;
35 |  end
36 end
37 return EB;
```

Algorithm 1: Composition Algorithm Pseudocode

19-26 (using the *dot* notation to indicate the access to a precise component of a contextual block). As far as the REFRESH component is concerned, the **composeBlock()** function computes (and attaches) the lowest refresh value among the ones contained in the R[ ] set. It seems reasonable, in fact, that the context whose state must be controlled with a higher frequency (smallest temporal values) is the most critical one and its refresh value is to be chosen during composition. Except for the discussed REFRESH component, the ENABLE and DISABLE components are formed by

multiple clauses expressed using PerLa syntax: simply appending, one after the other, all the clauses contained in this components is not enough. An **optimizeBlock()** function is in charge, acting on the composed ENABLE and DISABLE components, of placing every single PerLa clause in the ($\mathcal{E}$, $\mathcal{D}$, $\mathcal{R}$) blocks and put these blocks in the right position according to the specific query syntax, described in (Schreiber et al., 2012); some of the function tasks are: removing duplicate clauses and reordering and rearranging them in order to enhance the system performances.

The last step of the algorithm instructs the CM to inject the composed contextual block into the middleware QueryParser component using the **parseBlock()** function. The QueryParser validates the syntax and the semantics of the composed block and raises a warning message in case some inconsistencies are detected.

Alternatively, the composition of a contextual block can be carried out at run-time only when its relative context is recognized as active by the middleware (Schreiber et al., 2012), so avoiding the generation of many contextual blocks even if their actual activation happens very seldom. However, more than one context can be active simultaneously, and switching between contexts may be frequent; therefore the on-line composition of several context elements, possibly involving complex partials, could potentially slow down the whole system performance.

2.2 Context: the Programming Languages View

Several programming paradigms have been proposed in the literature (Salvaneschi et al., 2012a) in order to support context-awareness in object-oriented and modular programming languages. So far, the programming language level has been considered an alternative to a solution at the architectural level removing, in this way, the need of designing dedicated components to achieve the same functionality.

While in Section 5 we briefly review different approaches which have been proposed to manage context in programming languages, namely *Aspect-Oriented Programming (AOP)* (Kiczales and Al., 1997) and *Behavioral Programming (BP)* (Harel, Marron, and Weiss, 2012), in this section we analize *Context-oriented Programming (COP)* (Appeltauer, Hirschfeld, Haupt, Lincke, and Perscheid, 2009), (Hirschfeld, Costanza, and Nierstrasz, 2008), (Salvaneschi, Ghezzi, and Pradella, 2011) since the context model adopted by a program that follows the COP paradigm is the one that best fits the PerLa context model. In particular the COP concept of *layer* can be easily compared to the concept of *context element* in the CDT, while the *layer composition* procedure is the COP translation of the concept of *CDT context*: in PerLa, a context is the conjunction of different context elements while in COP it is the composition of several layers. Anyhow, we do not exclude the possibility to integrate other paradigms (AOP, BP, ...) in future works.

2.2.1 *Context-Oriented Programming.* Context-oriented Programming (Salvaneschi et al., 2012a), (Appeltauer, Hirschfeld, and Lincke, 2013), (Hirschfeld et al., 2008) enables changes in the behavior of the application depending on the current context. COP treats contextual information explicitly and provides a complete support to behavior adaptation at run-time; it introduces new keywords that may depend on the underlying language and the specific library, in order to provide complete support to the programming paradigm. Several libraries have been introduced to implement COP in most of common programming languages, such as Java, Python, and others.

Context for COP is simply intended as *any information which is computationally accessible by any part of the system and may determine behavioral variations*; when a context is active, a program must be able to behave in one of a determined set of alternative ways.

The main concept of COP is the *behavioral variation*. Each behavioral variation is related to a specific piece of context and can be dynamically activated or deactivated at run-time, enacting a behavioral change; it represents the modularization unit of such piece of behavior.
Many different solutions have been proposed to create a connection in the program between the contextual information and dynamic behavioral variations; since the layer-based model (Desmet, Vallejos, Costanza, and Hirschfeld, 2007) is the most widespread, we will refer to it in the following.

Layers

Layers are entities which group related context-dependent behavioral variations and contain *partial method* definitions that implement the functionality of behavioral variations. Layers can be dynamically activated and deactivated at run-time. Depending on the current context, different layers will be selected for further program executions. Several solutions have been proposed in COP for layers activation. In the JCop Java extension, layers are first-class entities, which can either be defined within the classes for which they provide behavioral variations (*layer-in-class*), or in a dedicated top-level layer similar to an aspect (*class-in-layer*) (Appeltauer et al., 2013). In the layer-in-class case all the possible variations are directly encapsulated in classes which need adaptations. Layer activation is dynamically scoped: it affects the behavior of the program not only for the method calls syntactically inside the code block, but also for all the calls triggered in turn. In the class-in-layer case it is possible to modularize adaptations independently of the base code, obtaining code that is free of context-specific concerns.

Other COP languages, namely EventJC, do not rely on a monolithic block structure, rather, they separate the layer activation control mechanism and the execution of the context-dependent behavior. Layer activation is controlled by a state transiton model; transitions between layers are triggered by *events* and are specified by a rule-based sublanguage (Kamina, Aotani, and Masuhara, 2011), (Kamina, Aotani, and Masuhara, 2013). This separation allows the changes of context and the execution of context-dependent behaviors to happen at different points in a program; however, the lack of a clearly defined dynamic scope can entail some consistency problem during the program execution (Salvaneschi et al., 2011).

Layers dynamic composition

COP directly provide all the features necessary to perform context-based behavioral variations at run-time, without using metaprogramming or other different frameworks. If the execution of a method is not affected by context, it is called *plain method* (Appeltauer et al., 2009). Context-dependent behavioral variations, called *layered methods* since they are involved in layers composition, (Appeltauer et al., 2013) are expressed as *partial method definitions*. In layered methods, a *base method* describes the normal behavior; it is executed when no active layer provides a corresponding partial method, and at least one partial method definition exists (Appeltauer et al., 2009). The dispatching mechanism is intuitive: when activated, layered method calls are dispatched to the partial method provided by the layer. So, partial methods provide a different behavior compared to base methods and they can be executed before, after, or around a base method.

Keywords ***with*** and ***without*** provide an explicit layer composition mechanism, because they specify which layers must be activated (deactivated) for the scoped block. However an explicit layer composition, based on ***with*** statements, is not enough in many cases; for this reason many different activation mechanisms have been proposed (Salvaneschi et al., 2012a).

Event-based composition

Dynamically scoped activation is a general model in which the ***with*** statements activate a sequence of layers in the code block.

In a pervasive environment, context changes are event-driven; events can be handled in a synchronous way, especially those bounded to changes in the environment and in the system, or asynchronously, such as those related to the user interactions. JCop provides specific constructs for declarative and event-based composition.

The *declarative layer composition* model consists of a logic concatenation of predicates, which represent events, and of a composition block, which contains some ***with*** (or ***without***) statements. Moreover, it introduces two new constructs in order to solve the problem of scattered ***with*** statements: i) the keyword ***on*** which identifies an event in the program execution flow; ii) an optional keyword ***in*** to bind the object on which the composition declaration should be evaluated.

This model is a good solution, in particular in case of composition based on predictable events. In situations that are connected with unexpected events, the explicit specification of those events

with ***on*** statements can be complex and could become really verbose.

For this reason JCop, in addition to ***on***, introduces the ***when*** statement. This statement specifies an expression that represents an event occurred in the program control flow and, if it is evaluated to *true*, layered methods in the code block are activated. Composition made with ***when*** statements is called *conditional*. The ***on*** statement specifies the points "where" the composition takes place, whereas the ***when*** statement allows to declare "when" the composition of layers must start. Furthermore, JCop provides a first-class ***context*** construct; ***contexts*** are special singleton types that can not be instantiated. The construct can host both declarative and event-based composition statements and auxiliary methods and fields. For a more complete explanation of layers composition models see (Appeltauer et al., 2013).

3. INTEGRATING PERLA AND OTHER PARADIGMS

In this section we provide some examples of how PerLa can be extended with other paradigms to apply it in a wider range of possible scenarios. We shall focus on two main examples: the possibility to integrate *Web Services* considering them as sensors, so expanding the sensing capability of the middleware, and on the integration between PerLa and COP, to enhance the expressive power of the system allowing more complex context-aware operations thanks to the extension of the query language with new semantics.

Figure 4 synthetically shows how PerLa can be used to monitor the context state by means of the *ACTIVE IF, ON ENABLE*, and *ON DISABLE* clauses within which context-aware actions can be performed on data, through native PerLa statements, or by web-services or COP programs, through existing APIs or layer activation mechanisms respectively.

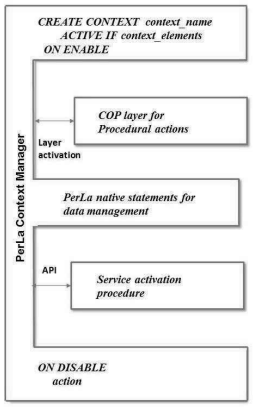

Fig. 4. General structure of a C-A self-adapting system

3.1 PerLa and Web Services

In its first versions PerLa connected only sensors through the TCP/IP protocol. In the current version (Rota, 2014) the communication structure has been enriched in order to allow the retrieval of information also from other systems.

In fact, many Web Services exist which offer APIs, like weather forecast services (providing temperature, pressure and other weather-related parameters). These APIs give information that PerLa can model like a virtual sensor.

The enabling feature for this new paradigm is the implementation of a *Channel* supporting the HTTP communication protocol. The real challenge is to create this channel as generic as possible, so it will be independent of the content-type used by the HTTP call for transferring data (in the *request* or in the *response*).

3.1.1 *PerLa Channel design.* The PerLa *Channel* is the component responsible for the communication between the FPC and a real device. In this case the device is represented by a Web Service with an HTTP interface (ex. SOAP or JSON/REST).

In order to define and configure a channel, PerLa offers three base classes: *i)* **AbstractChannel** is the abstraction of any communication channel between the PerLa FPC and the real device (through a TCP socket, a WebSocket or the HTTP protocol), *ii)* **ChannelRequestBuilder** is the Java object used by the FPC for creating the requests to be passed to the channel and *iii)* **Request** is the Java object created at run-time when a FPC needs to retrieve some data from a device.

The channel provides synchronous requests which are added to the queue shared between the channel and the FPC. The FPC can process the response immediately (synchronous call) or it can postpone the answer and process it just when it is needed (asynchronous call).

PerLa *Channel*s are dynamically built at run-time through the translation of a Descriptor (Java representation of XML descriptor) into PerLa objects.

This translation is performed by some Java objects: the *ChannelFactory* is responsible for the XML channel descriptor validation; the *ChannelRequestBuilderFactory* is responsible for the validation of the XML request descriptor; the *ChannelDescriptor* is the Java representation of *channel* XML tag and the *RequestDescriptor* is the Java representation of the *request* XML tag.

The HTTP Channel implementation has been realized as a specialization of the generic classes described earlier in this subsesction, providing a set of Java objects and methods that extends the basic ones with all the features required to properly handle the HTTP channel.

The inclusion of Context-aware Services (Maamar, Benslimane, and Narendra, 2006) in the picture, as described above is a further step towards the building of comprehensive Context-aware systems. A first prototype which integrates PerLa and web services in order to optimise energy consumption in buildings has been implemented in the SENSORI project[2].

3.2 PerLa and COP

In this section we are going to present how the context is managed in PerLa and, in particular, how its components perform behavioral variations, comparing it with the COP paradigm. However, it is not in the aim of this work to review all the possible languages that refer to the COP paradigm; such comparison can be found in (Salvaneschi et al., 2011), (Salvaneschi et al., 2012a).

We choose the COP paradigm, and in particular the JCop Java extension, as a first integration attempt because the core of the PerLa middleware is deployed in Java and JCop includes all the features needed to build the first system prototype.

As presented in Section 2.2.1, COP languages focus on the activation at run-time of context-dependent behavioral variations; in particular, it provides features to directly perform the variation of the involved modules, starting from some significant contextual information. COP can provide behavioral variations for events generated by the program code of the application, i.e. for events related to the interactions among its internal objects or modules. If the goal is the implementation of the adaptive part of an application, we think that using the COP paradigm is the easiest solution.

Context management

As mentioned in Section 2.1.2, the PerLa framework already provides a general context model: the CDT. The designer must only declare a dedicated CDT and write the application related CL queries and the middleware will be responsible for its management. This reduces the chances that, for large and distributed applications, some kind of inconsistencies occur or some kind of contextual information be not properly modeled. Moreover, PerLa CL has been designed for the creation of a direct inter-relationship between the context model, data, and adaptations; it allows to define: i) which data affect a context; ii) how sub-contexts can be composed to create higher level contexts; iii) which actions must be executed when changes in active contexts are detected.

[2]SENSORI has been funded by Industria 2015 program n° MI01_00091 of the Italian government

At the current development stage, besides the web services APIs, PerLa provides contextual features that mainly focus on data, and the actions that can be performed in response to contextual changes are limited to the execution of PerLa SQL-like statements (Schreiber et al., 2012). Context has the role of a data "tailor", i.e., it allows the user to define which data, retrieved by sensors, must be selected in a specific situation. The actions permitted are essentially limited to the activation or deactivation of devices, to the setting of some sampling parameters or to a limited range of tasks performed by some actuators, as presented in the office example of Section 2.1.

Moreover, PerLa has been designed to work with several data streams, produced by the sensing devices; the Low Level Queries allow to set their working mode: the sampling intervals, which data must be selected and the computation to perform on the sampled data. With CL, the designer can define a *contextual dynamic view* on a data stream. On the contrary, COP languages are not directly aware of how context information is provided; in fact they do not directly manage sensors, but they use the information provided by them, to perform behavioral variations. Adopting JCop, the developers must implement a mechanism to monitor continuous data sources, in order to provide contextual information; other approaches (e.g., Flute (Bainomugisha, Vallejos, Roover, Carreton, and Meuter, 2012)) instead provide also mechanisms to directly monitor continuous data sources.

The CDT model deals naturally with contexts belonging to different groups of sensors and to distributed instances of the application. The nodes at the lower level of the tree could be used to abstract several instances of a dimension, creating a *local* CDT for each instance. In this way, context data can be distributed to different locations, leading to the introduction of a *combined* CDT comprising a primary CDT and one or more *local* CDTs. PerLa is very suitable for context distribution and this feature has been used in the Green Move application, in which - for privacy - a portion of the CDT is maintained locally to the user's devices and is used to complete the context-based data filtering (Panigati, Rauseo, Schreiber, and Tanca, 2012).

Even if several threads may exist for each local instance and each thread adapts its own behavior w.r.t. the instance in a different way, with JCop it is only possible to implement the behavioral variations of the instances; mechanisms to compose information coming from different local contexts, in order to infer a higher level context, and for data sources monitoring are not supported. For this reason, it may become necessary to introduce dedicated components to generate significant contextual information, starting from rough data provided by sensors, and to decide which layers must be activated on the application.

The PerLa middleware has the components already designed for this purpose: the CM for what concerns the context and the application actions management, and other low level components for what regards the data sources monitoring, their operational impact being hidden to the user. Designers, through the PerLa CL, are relieved of the responsibility to develop dedicated components for context management which could result a rather complex task. With the possibility to define new contexts by composing other contexts at run-time, it becomes easier to specify the desired number of contextualized actions.

Let us now suppose that in the office rooms a new air conditioner and an air outlet have been installed; the devices functions, shown in Figure 5, are available as soon as we plug them into the system, without the necessity of applying further changes to it. The air conditioner can work in different modes depending on the room environment status: it can lower or rise the room temperature, it can dry off humidity and it can set a fan speed. The PerLa code in Listing 4 shows that, by installing the new air outlet in the offices, its activation becomes possible in case of persistent smoke.

The context *SmokeMonitoring* presented in Listing 4, is actually an extension of the *fire* context (Listing 2) since it performs additional actions (e.g. it stops the ventilation not to spread the smoke and it opens the outlet to let it flow away) in order to be more effective in case of fire. PerLa CL creates a direct connection between monitoring and adaptations: it checks for the

Listing 4: The SmokeMonitoring example in PerLa

```
CREATE CONTEXT SmokeMonitoring
ACTIVE IF Location = 'office' AND Smoke = 'persistent'
REFRESH EVERY 1h
ON ENABLE (SmokeMonitoring) :
SELECT smoke
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (smoke)
SET PARAMETER air_outlet = TRUE
SET PARAMETER alarm = TRUE
SET PARAMETER fan_speed = 0
ON DISABLE (SmokeMonitoring) :
DROP SmokeMonitoring
SET PARAMETER air_outlet = FALSE
SET PARAMETER alarm = FALSE
```

amount of smoke in an office and the context *SmokeMonitoring* is automatically activated if the smoke is persistent.

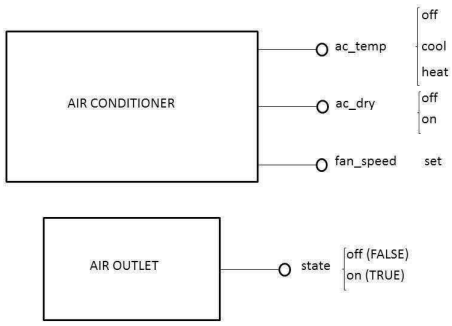

Fig. 5. The office new equipment

Enacting behavioral variations

To implement the office example with JCop, an external mechanism to monitor changes in context must be specified in addition to the definition of layers and when they have to be activated. An intuitive solution could be the introduction of a thread to monitor the temperature and the risk of fire, and another thread to monitor the smoke level in the room. The ***context*** construct of JCop allows to encapsulate changes related to the smoke detector. In fact, we can assume that, in case of fire, the activation of the fire alarm is the "normal" behavior, whereas the activation of both the fire alarm and the air outlet represents a variation.

In order to provide a comparison between PerLa and COP, we adopt the conditional composition approach for the next examples, due to the presence of the ***context*** construct and the ***when*** clause. If the context is active and therefore the condition in the ***when*** clause of the corresponding concept is satisfied, an established action is performed.

The proposed JCop code (Listing 5) translates the PerLa example of Listing 4: the main *SmokeMonitoring* thread instance `sm` sleeps for ten minutes[3] (cf. the PerLa *SAMPLING EVERY 10m* clause) and, when resumed, it controls the smoke level in the room; if it detects persistent smoke within the *smokeRisk* method of the *SmokeMonitoring* thread (cf. the PerLa *ACTIVE IF* clause), it activates context *smokeRisk()*, equivalent to PerLa *Location = "office" AND Smoke = "persistent"* (Listing 4), in which the ***with*** statement will activate the *SmokeLayer* in the class *ActiveActuators*, so executing the same actions that the PerLa code invokes in the *ON ENABLE* clause. The *ON ENABLE* and *ON DISABLE* clauses of PerLa CL could apparently have a

[3]The `sleep(600000)` is part of the thread instance code and it is not shown in the listing.

Listing 5: The SmokeMonitoring example in JCop

```
context SmokeRisk {
   in(SmokeMonitoring sm) && when(SmokeMonitoring.smokeRisk()) {
     with(SmokeLayer);
   }
}

class ActiveActuators {
   public activeAlarm()  {
     // When this method is called by thread
    // SmokeMonitoring,
     // the air outlet will be opened
     Alarm.activateFireAlarm();
   }
       ...

   layer SmokeLayer {
     activeAlarm() {
       // If the layer is activated, the air outlet will be opened
       Outlet.sendOpenCmd();
     }
   }
 }
```

Listing 6: The OverheatMonitoring example in PerLa

```
CREATE CONTEXT OverheatMonitoring
ACTIVE IF Location = 'office'
       AND Env_Temp = hot AND Env_Humidity.h_level > 0.65
REFRESH EVERY 30m
ON ENABLE (OverheatMonitoring) :
SELECT temperature, humidity
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (temperature, humidity)
SET PARAMETER ac_temp = 'cool'
SET PARAMETER ac_dry = 'on'
SET PARAMETER fan_speed = 0.65
ON DISABLE (OverheatMonitoring) :
SET PARAMETER ac_temp = 'off'
DROP OverheatMonitoring
```

behavior similar to that of the ***with*** and ***without*** statements of JCop mentioned in Sect. 2.2.1, since they enable (or disable) a given procedure, retrieving some data from the sensors and thus operating on them.

If *SmokeLayer* has been activated and a high amount of smoke is detected, the partial method *activeAlarm()* switches to "open" the state of the air outlet. When the layer is deactivated, the state of the actuator is switched to "close" (*SET PARAMETER ac_temp = "off"*), going back to the previous situation.

This example shows that context in JCop is driven more by events than by rough data; it also introduces other similarities between PerLa and JCop, at least from a conceptual point of view. The concepts of partial components and partial methods could be considered based on the same idea of composing different entities to provide a new behavior; the PerLa context query and the COP **context** construct contain dedicated statements to declare when context changes and which actions must be executed in response.

PerLa partial components and COP context entities refer to context composition and adaptations as a whole, i.e., from the composition of different basic data in order to obtain a new higher concept, to the execution of combined actions, covering both data management and behavioral variations.

As an example of how the coordination of several actions related to different contexts can be achived by the CL in a simple way, Listing 6 shows how the context *OverheatMonitoring* checks every thirty minutes if, in the set of data provided by a group of devices, there is a value in

the temperature field higher than the settled threshold, and if it finds it, the air conditioner switches on the cooling function. Now, suppose that a context *VentilationMonitoring* (Listing 7) is declared in order to monitor when the office temperature lowers too much: in this case there would be a partial overlap between the data required by contexts *VentilationMonitoring* and *OverheatMonitoring*. In fact, both contexts need the current temperature value as a partial context while only *OverheatMonitoring* requires the humidity value (see Fig. 2); the PerLa middleware deals with both the composed contexts to provide data and to perform actions in order to set the air conditioner variables: the fan speed, the humidity and the temperature control.

Listing 7: The VentilationMonitoring example in PerLa

```
CREATE CONTEXT VentilationMonitoring
ACTIVE IF Location = 'office'
        AND Env_Temp = cold
REFRESH EVERY 30m
ON ENABLE (VentilationMonitoring) :
SELECT temperature
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (temperature)
SET PARAMETER ac_temp = 'heat'
SET PARAMETER fan_speed = 0.65
ON DISABLE (VentilationMonitoring) :
SET PARAMETER fan_speed = 0.3
DROP VentilationMonitoring
```

The separation of context definition and activation management sections from the code specifying the context-aware behavior of the system, and the contextual block composition mechanism, introduced in Section 2.1.4, are similar to the *composite layers* mechanism introduced in EventCJ (Kamina et al., 2013) to overcome the linguistic problems connected to units of behavior which can correspond to different contexts and, possibly, can be executed under a combination of contexts.

In general, by adding the context management features, besides for sensor data collection and querying, PerLa can be adopted for applications performance monitoring. Sensors can be configured for this purpose and the collected data can be used to change the working parameters of the monitored application, e.g., in order to increase its efficiency; the application developers must declare the significant context changes and which actions must be performed in different situations, with no need to implement anything at the operational level.

3.3 Implementation issues

In the previous sections, we saw how the COP paradigm introduces the problem of choosing the most appropriate way to compose the application, including the choice of the layers composition mechanism. For example, *dynamically scoped activation* approaches are convenient when all the entities in the control flow are context-dependent, while *per-object activation* is a suitable solution when behavioral variations intersect only a little part of the application structure (Salvaneschi et al., 2012a).

A solution to the implementation of a Context-Aware self-adapting system can be to delegate everything related to the configuration of sensors, the intermediate computations on data, and contexts declaration and management to PerLa which, as shown in Section 2.1, provides a complete support for managing all the entities involved in the pervasive environment, and to use COP languages to perform behavioral variations, using the contextual information provided by PerLa; a context can be viewed as a particular *context data stream*, i.e. a normal PerLa stream, properly adjusted to provide only data related to the defined context. This operation is performed by the CM: it manages the CDT and the actual contexts, and creates the contextual streams accordingly. Some simple actions, such as those of the office examples, can be still performed by the CM.

COP, in turn, can handily use the information inserted in context data streams to implement the desired behavioral variations. The programmer has only to care about the definition of layers, with statements, partial methods, etc., for the entities whose normal behaviors must be dynamically adapted.

Moreover, the activation of a layer at the client application can be seen as a particular case of *firing external events* from the PerLa middleware. Therefore, since the concept of event represents a general notification the system sends to a remote application, we extended the Context Language, the CDT declaration language and the representations of these constructs with the possibility of declaring events by means of the *EVENT* interface. This clause, which can be inserted both in the *ON ENABLE / ON DISABLE* and in *WITH ENABLE COMPONENT / WITH DISABLE COMPONENT* clauses, is composed of three parts:

Type of event This part is a token that defines which event will be fired. In our implementation we define two types of events: i) in order to detect the change of the activation status of a COP layer the token *COPLAYER*; ii) the second one - *REST* - represents a remote request to a REST web service. Both events are represented by an object that inherits from the event interface, in particular the Layer class and the Rest class.

Command This part declares the actions that must be performed when the event is fired. It is specific of the type of event.

Attributes In this part, all the parameters (mandatory or optional) that are needed to further specify the event are declared.

As to the communication pattern between the PerLa middleware and the client, we chose a lightweight implementation of the publish-subscribe mechanism which is fit for sending a large number of messages to subgroups of connected applications, based on the type of message and its content, without directly managing the connections.

The resulting architecture is shown in Figure 6, while in Figure 7 the sequence diagram of the interactions among the components is presented.

Referring to (Angaroni, 2015)[4] for a detailed description of the extended system, we mention here only the most relevant updates to the PerLa language and middleware.

Language clauses

In order to bind the change of status of a context in PerLa to the activation or deactivation of layers the *EVENT* clause is used, followed by the *COPLAYER* token for which three types of commands have bee defined:

LOAD This action defines that the specified layer has to be activated and that the activation command will be sent to the interested applications.

DROP This action is the opposite of the previous one and it is used to deactivate a layer; it can be declared only if a layer has been already activated using the *LOAD* clause.

TOGGLE This action specifies that a layer will switch from active to inactive when the context it is associated with changes its status.

We must notice that the uncontrolled use of *LOAD* and *DROP* actions can cause inconsistencies and an unforeseable behavior as, for instance, in the case a layer is activated an unbounded number of times. Anyhow, we decided to keep them in order to leave complete freedom to the designer, but to address this problem, we defined the attribute *MAX*, whose default value is "unbounded", for the *LOAD* command in order to set an upper bound to the maximum times a layer can be activated; if the *TOGGLE* command is used you need not care about this problem.

Other attributes for the commands are:

[4]http://perlawsn.sourceforge.net/files/documentation/2016_04_Angaroni.pdf

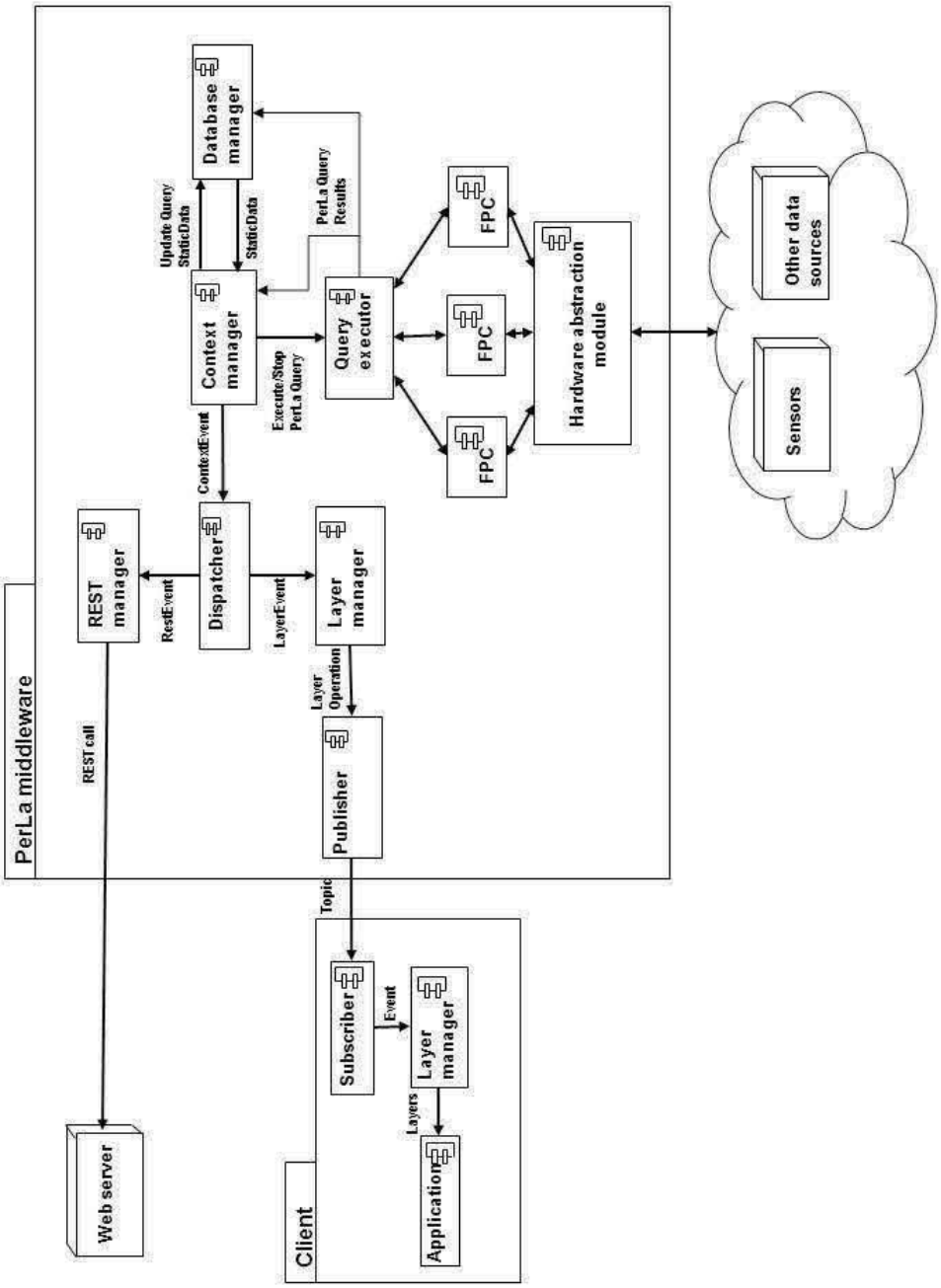

Fig.6. The system architecture

Identifier This attribute is mandatory and defines the unique identifier common to the system and the remote applications. Therefore, if two layers are declared with the same identifier the system will treat them as the same layer. This value is also used to link the particular event fired by PerLa and the COP layer in the remote application.

Domain This optional attribute is used by the *LOAD* and *TOGGLE* commands to declare to which domain of the publish-subscribe it will be distributed. It is particularly useful when the designer wants to define different types of applications that can be connected to PerLa; it can also be used to define a security policy in which only the applications that connect to

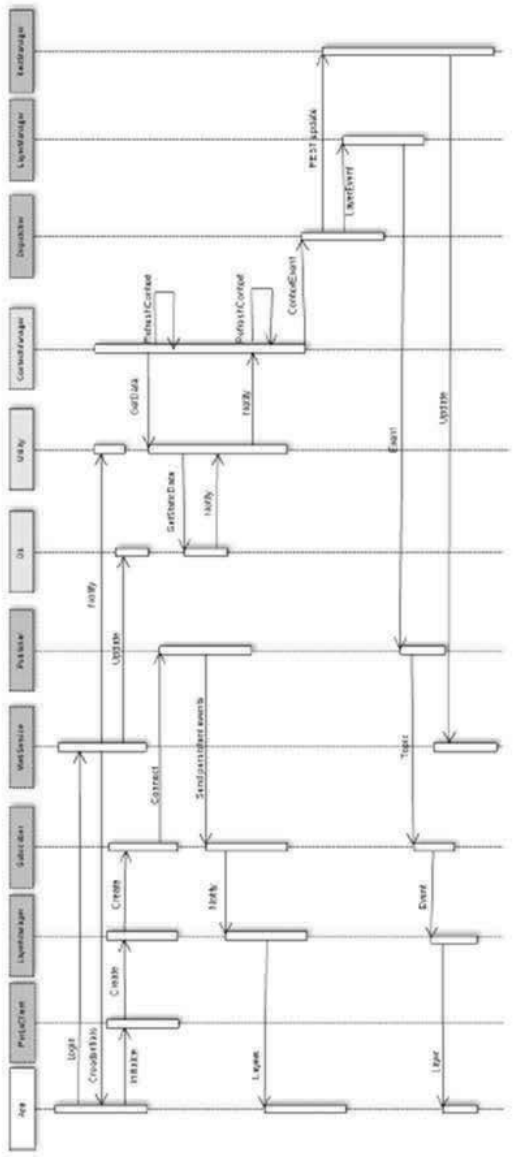

Fig.7. The components interactions sequence diagram

the specific domain can receive that event. If not specified, the event will be published on the *public* domain.

Shared This is an optional boolean attribute used by the *LOAD* and *TOGGLE* commands. If set to TRUE, it signals to the client's layer manager that this type of layer can be activated by the client application independently of PerLa. The default value is FALSE.

PerLa can also call a REST service for performing the *POST, GET, PUT*, and *DELETE* operations in order to obtain some environmental information and possibly to update the web site when a context changes its status. The only constraint for this type of operations is that they must refer to an existing web service; this condition is checked at run-time by the *RestManager*.

The attributes that can be specified in a REST event are:

URI This attribute specifies which web service the action is directed to.

Content In this attribute the content to be sent to the web service is specified. It is available only for the *POST* and the *PUT* commands; its value is a string, but the user, by inserting the name of a concept or of an attribute preceded by the dollar sign, can specify the name of some values that will be inserted when the event is fired. This attribute must be declared in the CDT and it must be used in the current context declaration.

Middleware components

In order to manage the new event related language clauses, the *ContextEvent* class has been defined. It inherits from the class *Context*, allowing the context manager to treat in the same way a normal context and a context that contains some events. Then we created an interface named *EVENT* that defines a generic event; as previously seen, it can be specialized for every type of event that can be fired. In our case, we created the Layer and the Rest entities.

The **Dispatcher** is the main entry point to define a context that needs to fire some events. It manages and keeps track of the different *ContexEvent* which have been created. It receives the notification from the Context Manager and it distributes the events to the event-specific managers.

When the context changes its status, the Dispatcher notifies the **EventManager** interface the event with the specific actions, which will be executed by the appropriate event manager. The ability of the **EventManager** to keep track of all the declared and activated *ContextEvent* objects has two main advantages: i) it centralizes the management of these objects and avoids useless replication in the other managers; ii) it allows a general consistency check. A UML schema of the middleware components is shown in Figure 8

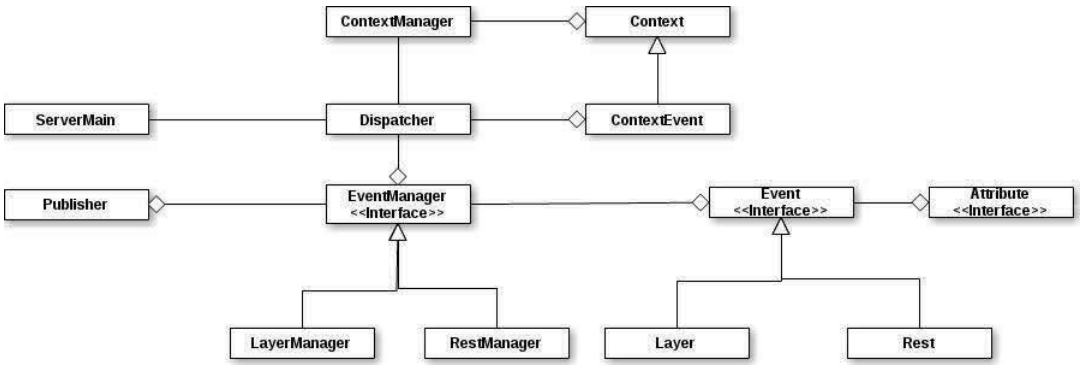

Fig.8. The middleware UML diagram

Events distribution

The publish/subscribe mechanism has been implemented by using Apache ActiveMQ which has all the features that are deemed important for this kind of application, namely:

- Loose coupling between the applications and the middleware;
- Assurance of delivery;
- Capability to deal with a large number of connections;
- Ability to manage heterogeneous types of messages;
- Easily extendable to new types of messages;
- Ability to manage clients that disconnect and reconnect;
- Ability to send the client only the messages it is interested to;
- Having a security policy;
- Being lightweight.

Moreover, it allows to implement the subscriber in different languages.

Since not all the applications must receive all the events that can be fired, we need some kind of security policy in the distribution of the events. Therefore, we give the system the possibility to divide the different connected applications into domains on the basis of the topics of interest; if a topic is published in one domain, it will be invisible to the others. Additionally, in the different domains some encryption mechanism can be set.

Two domains are originally created: i) a *public domain* in which the unprotected topics are published. This domain contains all the available messages. The applications that subscribe to this publisher need not provide any credentials. ii) a *system domain* in which the information of the PerLa system, if any, will be published. Other domains will be created only when the system is deployed in a specific environment.

On the server side, the **MainServer** class creates the publisher; it receives the subscriptions by the applications, then it checks if the application needs to subscribe to a specific domain and if it has the right credentials. Then the **Publisher**, specific for each type of event, is assigned to a domain. It dispatches the messages and assures that all of them are received by the subscribers. Once created by the MainServer, the Publisher is passed to the specific EventManager that receives the notifications of the events from the Dispatcher. Then the EventManager selects the right Publisher, creates the message and sets some parameter in the header in order to allow filtering at the client side.

On the client side, the **Subscriber** receives a specific kind of event and notifies it to the appropriate **EventManager**. When the user initializes the **PerLaClient** class it selects in which type of events it is interested in order to allow the PerLaClient to create the proper managers and the subscribers. Additionally, the domain from which the applications must receive the events must be set, and, if required, some credentials must be provided. Moreover, in order to filter the received messages, some user information can be set by using the JMS Message Selector class. If some consistency on the reception of the events is to be assured, such as those referring to layer activation, the subscriber must be set as *durable*.

The client

The client, which receives the event notifications, is made of three components:

the **Subscriber** classes which receive the messages;

the **EventManager** classes that manage the events and create the **Subscriber** classes for the different kinds of events and domains.

The **PerLaCLient** class that creates the different **EventManager** classes.

In general, an application can decide directly at which kind of events it has to subscribe; however, for the specific case of layers activation and deactivation, the client application delegates the management of some layers to PerLa. Thus, the PerLaCLient automatically configures the **LayerManager** and the **Subscriber** classes.

As to the domain, the application designer must specify at which domain the application must connect. If this parameter is not set, the EventManager will automatically connect to the *public* domain.

The **LayerManager** allows to activate or deactivate layers and modify the behavior of the targeted application. When the layer manager receives, as a parameter provided by the application, the **JCOP** class, which is the main controller of the composition of layers in JCop, it checks if the defined layers are actually declared in the application itself and retrieves them. Then it creates the **Subscriber** and sets the specified layers as a filter, in order to receive only the relevant messages.

When a layered event arrives, the **LayerManager** decides which action is to be performed based on the command attribute. If it is a LOAD it adds the layer to the current composition of layers enabled by PerLa, if it is a DROP it will remove the specified layer from the current

composition and if it is not present it will raise an exception. Other consistency constraints are detailed in (Angaroni, 2015). In Fig. 9 the UML schema of the client is shown.

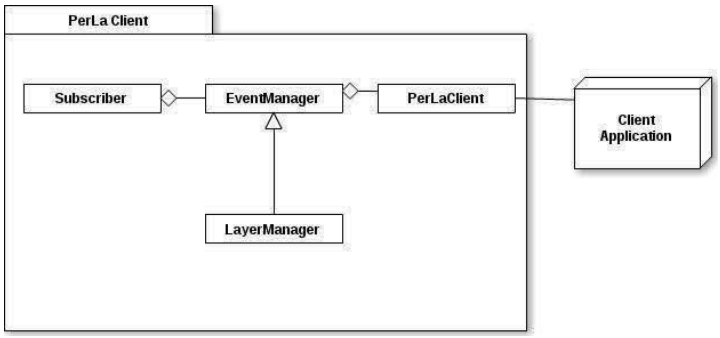

Fig.9. The Client UML diagram

4. A CASE STUDY

In order to show the possible applications of PerLa we shortly describe a real-world example where this system can make a difference in the control of the environment and the prevention of risks. The chosen site to monitor is a well known ski resort in the Italian Alps (Angaroni, 2015). In such environment, there is a large number of parameters which are normally controlled by the staff, but the automation of such functions can make a real difference, in particular as to risk monitoring.

By allowing every person (either staff member or customer) to know in real-time what is happening in the resort, if risky conditions arise, it can alert users to be careful, and possibly it can suggest the staff to shut down the chair lifts and evacuate the customers. Moreover, it allows to automatize some operations; for example, if there is reduced visibility due to harsh weather, the system can automatically switch on the light signals that are placed on the pistes.

Among the physical environment status indexes, the most important one is the snow conditions. A block of snow can be characterized by its depth, temperature, density and snow water equivalent (SWE, water content obtained from melting a sample of snow). These measures tell how much snow is on the slopes and its conditions and can be used to decide whether artificial snow should be added. Another important measure is the snow density: if it is too low, the intervention of a snow-groomer is needed in order to prepare a surface suitable to ski, and it can also alert for an increased avalanche risk; if it is too high, it signals the presence of ice and, also relying on their skiing ability, a possible augmented risk for the users. The weather conditions such as the temperature, humidity, wind force and direction, snowfall or fog must be also considered.

Other factors to be accounted for are, for example, the chair lifts state, the presence of equipment or rescue squads on the slopes, and others that are useful to give a view of the system to the staff and to make customers to be aware of the risks in order to prevent accidents.

4.1 System architecture

In our scenario, smart sensor networks, composed of devices having the ability to sense, compute and communicate the environment temperature, light intensity, pressure, and other snow properties, are used for monitoring the snow composition on the pistes, data which, suitably aggregated, can be used for many purposes.

In order to measure the snow conditions, the CIT-1 snow sensor is a reliable and sensible sensor with automatic snow melting control, while the 260-700 Ultrasonic Snow Depth one is an inexpensive solution for remotely measuring snow depth or water levels. In order to connect them directly to the electrical grid, most of the weather sensors have been located near the Base

Listing 8: The ski resort CDT

```
...
CREATE DIMENSION SnowCondition

CREATE CONCEPT Icy WHEN temperature: < -5 AND snow_density > 0.30
EVALUATED ON EVERY ONE SELECT temperature, snow_density
SAMPLING EVERY 2 h
EXECUTE IF EXISTS (temperature) AND (snow_density)

CREATE CONCEPT Slushy WHEN temperature > 8 AND snow_density BETWEEN 0.12 AND 0.29
EVALUATED ON EVERY ONE SELECT temperature, snow_density
SAMPLING EVERY 2 h
EXECUTE IF EXISTS (temperature) AND (snow_density)
...
```

Stations - which are to receive the data from the deployed sensors - at the start and arrival point of the chair lifts. This allows the use of more powerful devices and reduces the cost of deployment and maintenance. Other devices are to be powered by batteries and solar panels and communicate by radio channels with the base stations.

All users in the resort have their own devices - equipped with a WiFi or GSM receiver - running the client PerLa system which connects to the *FPCs* through the base stations for getting the needed data.

A database is used in order to manage the static data of the resort.

4.2 The context

Using the CDT model explained in Section 2.1, we consider as primary context dimensions the *Role* of the user, his/her *Location*, the *WeatherCondition*, the *SnowCondition*, and the possible *Risk*. In Listing 8 an excerpt of the CDT declaration is shown:

Once the context model has been declared, contexts have been defined on it as explained in Section 2.1; in Listing 9 a couple of contexts which use the *SnowCondition* dimension are shown:

Icy Pistes. Icy pistes can be a problem for the skiers because they have less control in the presence of ice. For this reason we created a context that represents this information which could then be displayed on the website of the sky resorts and on the skiers devices in order to make them more careful.

Slushy Pistes. The opposite situation than icy pistes are slushy pistes. When the snow keeps melting and refreezing, the snowflake structure is lost and many little lumps of ice are left. This context is defined in order to inform the skiers to be careful since the pistes are not in optimal conditions.

4.3 Application programs

There are two main types of applications that use the data provided by the system: the first is a mobile application that runs on the devices of the users (see the clauses EVENT COPLAYER TOGGLE ... in Listing 9); the second one is the resort's web services (see the clauses EVENT REST PUT ... in Listing 9).

From the point of view of the mobile application, PerLa provides the commands to activate or deactivate the layers which enact the behavioral variations in the application program. For example, if the application knows that the user is an amateur skier and the layer IcySlopes is activated, it can send him an alert message to slow down due to the presence of ice in the pistes. In order to identify the user from the credentials he/she provides to the applications, a connection to the web services is needed; his/her role can completely modify the behavior of the application.. For example, the application used by a snow lift controller will be focused on the wind conditions

Listing 9: Context declaration example

```
CREATE CONTEXT IcyPistes
ACTIVE IF Place = Slope AND SnowCondition = Icy AND Role = Customer
ON ENABLE: EVERY 30 m SELECT temperature, snow_density
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (temperature) OR EXISTS (snow_density)
REFRESH EVERY 1 h
EVENT COPLAYER TOGGLE Icypistes ,
EVENT REST PUT  www.AlpSkyResort.com/SlopesState/IcyPistes

CREATE CONTEXT SlushyPistes
ACTIVE IF Place = Slope AND SnowCondition = Slushy
AND Role = Customer
ON ENABLE: EVERY 30 m SELECT temperature, snow_density
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (temperature) OR EXISTS (snow_density)
REFRESH EVERY 1 h
EVENT COPLAYER TOGGLE SlushyPistes ,
EVENT REST PUT  www.AlpSkyResort.com/SlopesState/SlushyPistes
```

Listing 10: Example of behavioral variation in the client application

```
when (Device.inSlope() && Device.Role().equals("AmateurSkier")) :
with(LayerManager.GetLayers() );
layer Icypistes {
public void DeviceAlert.update() {
// Alarm beep
}
public void Display.update() {
// Message: Slow down the snow is too icy
}
}

class Device {
public static boolean inSlope() {
/* checks GPS and RFID data */
}
public static String Role(){
/*Return the role of the user */
}
}

class DeviceAlert {
public void update() {
// Do nothing.
}
}
class Display {
public void update() {
// Message: The snow condition are optimal.
}
}
```

and the congestion on the chair lifts, while for a rescue squad it will be focused on showing the possible dangers that can show up in the resort area, e. g.: a possible avalanche or a rescue request.

Listing 10 refers to a fragment of a JCop application program showing a behavioral variation in case of difficult snow conditions for an amateur skier.

5. RELATED WORK

A very comprehensive analysis of other projects active in the context-aware systems research field can be found in (Hong, Suh, and Kim, 2009). From these works a similar approach to context-aware systems management emerges, in which context is mainly analyzed at the *application* level; a dedicated language is used to retrieve all the necessary information from the sensors (Reichle, Wagner, Khan, Geihs, Lorenzo, Valla, Fra, Paspallis, and Papadopoulos, 2008). This is the

case of *CIS* (Judd and Steenkiste, 2003), where contextual data is stored in a central database later queried using SQL. *CASS* (Fahy and Clarke, 2004) adopts a similar centralized database approach. In CoBrA the context is represented as a Context Knowledge Base (Chen, 2004) for the specific application of event/meeting management. On top of this knowledge base temporal, spatial and event/meeting reasoners (based on contextual rules) operate to deduce more abstract contextual information. *iQueue* (with the *iql* language (Cohen, Purakayastha, Wong, and Yeh, 2002)), on the contrary, allows to compose contextual information according to data specification requirements and provides a library that can be used to build context-aware applications. The SOCAM project (Gu, Pung, and Zhang, 2005) proposes an extremely general ontology-based context model. Sets of extensible ontologies are exploited to express contextual information about user, environment and platform In (Schreiber et al., 2012) we presented a middleware-based approach in which the overall computational complexity grows linearly with the number of deployed sensors.

The literature also contains some detailed surveys on the different models adopted to represent context as well as on available pervasive management system frameworks. The work presented in (Bolchini et al., 2007) defines a framework used to compare and classify sixteen different context models, while (Bettini, Brdiczka, Henricksen, Indulska, Nicklas, Ranganathan, and Riboni, 2010) provides a historic overview. As far as pervasive management system frameworks are concerned a detailed comparison can be found in (Schreiber et al., 2012).

Until 10 years ago, only solutions at the architectural level dealt with context and software adaptivity. In the recent past, researchers tried to find other ways to deal with those issues and, particularly, the focus has shifted on solutions at the programming language level. For this reason researchers started to move in the direction of embedding context management and context-dependent behavioral variations directly in the program code.

The capability of the system to adapt its behavior according to its knowledge of the whole global context is the most important aspect. The more information a system is able to retrieve, aggregate and manage, the more accurate its reasoning will be. Hence, every solution at the language level must allow the programmer to develop a full context-aware application, which can properly achieve three type of behavioral variations: *User-, Environment- or System-dependent*.

Many different models for the development of context-aware applications have been proposed. One of the most widespread frameworks created to support context management and adaptivity is *Context Toolkit* (Salber, Dey, and Abowd, 1999); it simplifies the activities of building context-aware applications using the Java programming language.

This framework represents an example of the relationship between adaptations and context. In fact, it provides specific components for context management, but also specific components for the encoding of behaviors, called *Services*.

Its current main features are:

- *Abstraction* of contexts, sensors, and actuators using *Widgets*.
- *Resource Discovery* of distributed components.
- *Rule-based* reasoning for context-aware applications through *Enactors*, components which encapsulate the application logic and simplify the acquisition of context data.
- *Machine learning* reasoning for context-aware applications.
- *Explanation* of application behavior and reasoning through the Intelligibility Toolkit.
- *Control* facility through *Enactors*.

The most important components are the *Widgets*, which are responsible for encapsulating the details of the devices for sensing context while providing applications with the needed context information.

Moreover, pervasive and ubiquitous applications have often to deal with distributed computing environments, and the problem of uniformly reasoning on different separated contexts is one of

the main problems of a distributed architecture. Context Toolkit introduces a rough model of context distribution related to two different concepts:

(1) the context model distribution, i.e., how the global context is split in many sub-contexts, each one with specific attributes and properties.
(2) how to reason on the distributed context, i.e., how the system utilizes sub-contexts to form a bigger piece of context since each context Widget is responsible for a sub-context captured by a sensor.

In Lee et Al. (Lee, Pham, Kim, and Youn, 2011) a hierarchical context model is proposed for reasoning on distributed context. It is composed by several layers: the highest level represents the global context of the whole system, while the lower levels represent the contexts of any local source of information. This abstract model shows a tree structure, which is very useful and manageable in many real situations. Higher level contexts are inferred from lower levels, without directly considering the original local information. This model allows to delegate the management of different pieces of context to different parts of the system and also to distribute the reasoning process to many independent entities.

We think that the approach presented in this paper, which merges the power of an object oriented programming language and of web services with the flexibility of a context-management system, provides a very general and application independent solution to the problem of building context-aware adaptive software.

Leaving the details to (Salvaneschi et al., 2012a), we present in the following some other context-aware programming paradigm; further possible approaches are described in (Bainomugisha et al., 2012), (Salvaneschi, Ghezzi, and Pradella, 2012b) and (González, Cardozo, Mens, Cádiz, Libbrecht, and Goffaux, 2010).

Aspect Oriented Programming (AOP) has as main goal the modularization of orthogonal functionalities in software by allowing a clear separation of *cross-cutting concerns*, where a *concern* is a set of information which affect program code. It might also happen that a concern implementation affects other concerns, creating code duplication and/or dependencies (*cross-cuts*) (Fabry, Dinkelake, Noy, and Tanter, 2015). *Aspects* are features which, while being not related to the program primary function, affect many part of the program; aspects are defined and designed not to violate the *separation of concerns* principle. The place in the main program where the code implementing an *aspect* is to be executed is specified by a *join point*; a set of join points constitute a *pointcut*. When, during the main program execution, a join point belonging to the pointcut is reached, an additional code - an *advice* - is executed in order to modify the standard behavior, so attaining a context-dependent behavior.

Behavioral Programming (BP) (Harel et al., 2012) is a fully scenario-based approach based on the concept of *behavior threads* (*b-threads*). Each thread models a specific scenario (e.g. a specific usage context), by defining the sequence of events that must be detected in order to identify it. A sequence may contain also negations or be indifferent to the presence of specific events. There may also be a set of conditions to be verified together with the sequences of events to recognize the context, and each b-thread may output other events during its execution. A thread may be totally, partially or incompletely independent from other threads; the interaction among them gives the integrated system behavior. All the threads are synchronized and a thread may forbid other threads to generate their events by generating and emitting its own events. Inter-thread communication is event-mediated (a direct message exchange between threads is not possible), so a thread sequence detection may depend from the generation of a particular event by another thread. b-threads are orthogonal to objects, and may not be anchored to a given scenario or to a particular physical object.

6. CONCLUSIONS AND FUTURE WORK

In this work we propose a hybrid solution to the problem of building context-aware adaptive systems based on the PerLa framework to design, declare and manage context; Context Oriented

Programming Java extension JCop is used to write complex layered procedures where each layer is bound to a specific context; the invocation of web-services allows the inclusion of *virtual* sensing devices.

After building a first prototype, the context management modules of the system are currently under implementation; at the end, an evaluation campaign is planned in order to asses the performance of the system itself as well as to compare both technical performance and effectiveness of our approach against those of other systems proposed in the literature.

Further work is going on in order to provide a full-fledged software system which can be used in many different application areas spanning from healthcare systems to intelligent energy management in smart buildings. This extension consists of embedding a context manager and a part of the CDT in the client software so enabling context inference to be made directly on the user device so obtaining a better response. Given the current technological developments, which embed different sensor types on devices such as smartphones, the possibility emerges of extending the sensing network without deploying it by simply inserting an XML device descriptor in the user device.

ACKNOWLEDGMENTS

We acknowledge the work of Ing. Matteo Rovero in surveying the Aspect-, Behavior-, and Context-oriented programming paradigms (Rovero, 2013) and of Ing. Andrea Angaroni in developing the PerLa – JCOP integration system (Angaroni, 2015) in fulfillment of their Master theses.

References

Angaroni, A. 2015. Extending perla with context oriented programming. M.S. thesis, Politecnico di Milano, Italy.

Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. 2009. A Comparison of Context-oriented Programming Languages. In *COP '09: International Workshop on Context-Oriented Programming*. ACM Press, New York, NY, USA, 1–6.

Appeltauer, M., Hirschfeld, R., and Lincke, J. 2013. Declarative layer composition with the jcop programming language. *Journal of Object Technology 12,* 2, 4: 1–37.

Bainomugisha, E., Vallejos, J., Roover, C. D., Carreton, A. L., and Meuter, W. D. 2012. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, G. T. Leavens and J. Edwards, Eds. ACM, 67–84.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. 2010. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing 6,* 2, 161 – 180.

Bolchini, C., Curino, C., Orsi, G., Quintarelli, E., Rossato, R., Schreiber, F. A., and Tanca, L. 2009. And what can context do for data? *Commun. ACM 52,* 11, 136–140.

Bolchini, C., Curino, C. A., Quintarelli, E., Schreiber, F. A., and Tanca, L. 2007. A data-oriented survey of context models. *SIGMOD Rec. 36*, 19–26.

Bolchini, C., Quintarelli, E., and Tanca, L. 2013. Carve: Context-aware automatic view definition over relational databases. *Information Systems 38,* 1, 45–67.

Borgia, E. 2014. The internet of things vision: Key features, applications and open issues. *Computer Communications 54,* 0, 1 – 31.

Chen, H. 2004. An Intelligent Broker Architecture for Pervasive Context-Aware Systems. Ph.D. thesis, University of Maryland, Baltimore County.

Cheng, B. H. e. a. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, vol. 5525. Springer Berlin / Heidelberg, 1–26.

Chui, M., Loffler, M., and Roberts, R. 2010. The internet of things. *McKinsey Quarterly*.

Cohen, N. H., Purakayastha, A., Wong, L., and Yeh, D. L. 2002. iqueue: A pervasive data composition framework. In *Proceedings of the Third International Conference on Mobile Data Management*. MDM '02. IEEE Computer Society, Washington, DC, USA, 146–.

Coutaz, J., Crowley, J. L., Dobson, S., and Garlan, D. 2005. Context is key. *Commun. ACM 48,* 3, 49–53.

Desmet, B., Vallejos, J., Costanza, P., and Hirschfeld, R. 2007. Layered design approach for context-aware systems. In *VaMoS*. 157–165.

Dey, A. 2001. Understanding and using context. *Personal Ubiquitous Comput. 5,* 1, 4–7.

Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. 2005. Self-managing systems: A control theory foundation. In *Proc. 12th IEEE-ECBS*. 441–448.

Editorial. 2007. When everything connects. *The Economist*.

Fabry, J., Dinkelake, T., Noy, J., and Tanter, E. 2015. A taxonomy of domain-specific aspect languages. *ACM Computing Surveys 47,* 3, Art. 40.

Fahy, P. and Clarke, S. 2004. Cass: Middleware for mobile, context-aware applications. In *Workshop on Context Awareness at MobiSys 2004*.

González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J., and Goffaux, J. 2010. Subjective-c - bringing context to mobile platform programming. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, B. A. Malloy, S. Staab, and M. van den Brand, Eds. Lecture Notes in Computer Science, vol. 6563. Springer, 246–265.

Gu, T., Pung, H., and Zhang, D. 2005. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl. 28,* 1, 1–18.

Harel, D., Marron, A., and Weiss, G. 2012. Behavioral programming. *Commun. ACM 55,* 7 (July), 90–100.

Hirschfeld, R., Costanza, P., and Nierstrasz, O. 2008. Context-oriented programming. *Journal of Object Technology 7,* 3, 125–151.

Hong, J., Suh, E., and Kim, S.-J. 2009. Context-aware systems: A literature review and classification. *Expert Syst. Appl. 36,* 4, 8509–8522.

IBM. 2006. An architectural blueprint for autonomic computing. IBM Autonomic Computing white paper.

Judd, G. and Steenkiste, P. 2003. Providing contextual information to pervasive computing applications. In *Proc. of IEEE International Conf. on Pervasive Computing and Communications (PerCom'03), Fort Worth, Texas, USA*. 133–142.

Kamina, T., Aotani, T., and Masuhara, H. 2011. Eventcj: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. AOSD '11. ACM, New York, NY, USA, 253–264.

Kamina, T., Aotani, T., and Masuhara, H. 2013. Introducing composite layers in eventcj. *Information and Media Technologies 8,* 2, 279–286.

Kant, I. 1787. *Kritik der reinen Vernunft (Critique of pure reason) 2nd Ed.* Riga.

Kiczales, G. and Al. 1997. Aspect-oriented programming. In *ECOOP*. 220–242.

Lee, C. K., Pham, T. H., Kim, H. S., and Youn, H. Y. 2011. Similarity based distributed context reasoning with layer context modeling. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. IEEE, 320–325.

Maamar, Z., Benslimane, D., and Narendra, N. C. 2006. What can context do for web services? *Commun. ACM 49,* 12, 98–103.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2005. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30,* 1, 122–173.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. 2004. Composing adaptive software. *IEEE Computer 37,* 7, 56–64.

Panigati, E., Rauseo, A., Schreiber, F. A., and Tanca, L. 2012. Aspects of pervasive information management: an account of the green move system. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*. IEEE, 648–655.

Pascoe, J. 1998. Adding generic contextual capabilities to wearable computers. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*. IEEE, 92–99.

Reichle, R., Wagner, M., Khan, M., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., and Papadopoulos, G. 2008. A comprehensive context modeling framework for pervasive computing systems. In *Distributed applications and interoperable systems*. Springer, 281–295.

Rota, G. 2014. Design and development of an asynchronous data access middleware for pervasive networks: the case of perla. M.S. thesis, Politecnico di Milano, Italy.

Rovero, M. 2013. Context-aware application development: A comparison of different approaches. M.S. thesis, Politecnico di Milano, Italy.

Salber, D., Dey, A. K., and Abowd, G. D. 1999. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*. ACM, 434–441.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2011. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR abs/1105.0069*.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2012a. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software 85,* 8, 1801–1817.

Salvaneschi, G., Ghezzi, C., and Pradella, M. 2012b. Contexterlang: introducing context-oriented programming in the actor model. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, R. Hirschfeld, É. Tanter, K. J. Sullivan, and R. P. Gabriel, Eds. ACM, 191–202.

Schreiber, F. A., Camplani, R., Fortunato, M., Marelli, M., and Rota, G. 2012. Perla: A language and middleware architecture for data management and integration in pervasive information systems. *IEEE Trans. Software Eng. 38,* 2, 478–496.

Schreiber, F. A., Tanca, L., Camplani, R., and Viganó, D. 2012. Pushing context-awareness down to the core: more flexibility for the PerLa language. In *Electronic Proc. 6th PersDB*. 1–6.

Prof. Ing. Fabio A. Schreiber is Professor Emeritus (retired since November 2015) of Pervasive Data Management and of Database Systems at Politecnico di Milano. From 1981 to 1986 he was full professor of Computer Science at the Mathematical Department of the Universit di Parma. He received the Dr. Ing. degree in Electronic Engineering from Politecnico di Milano in 1969. His main research interests are in the fields of Distributed Informatics and Information Systems. His current research topics include: Database Systems for context-aware applications, Very Small and Mobile Database design methodologies and applications; Pervasive Information Systems. Other research interests include: dependability evaluation of computing and information systems and distributed information systems design. On these and other topics he authored more than hundred papers published in international journals and conferences. He organized the "3rd International Seminar on Distributed Data Sharing Systems" (Parma, 1984), is member of the editorial boards of Data and Knowledge Engineering, and, from 1975 to 1993, he has been Editor-in Chief of Rivista di Informatica - the journal of the Italian Association for Informatics (AICA). He participated to several research projects funded by Italian research institutions and by the ESPRIT and HORIZON 2020 European actions. He also partecipated to several projects on Information Systems and Telematics for Public Administrations and Local Authorities; for this activity he was bestowed the title of "Commendatore" of the Order "Al Merito della Repubblica Italiana". He is member of AICA, Senior member of ACM, and Life Senior member of IEEE.

Dr. Emanuele Panigati holds a Ph.D. in Informatic Engineering from Politecnico di Milano. In 2010 he received the M.Sc. degree in Computer Engineering and, since then, collaborates with the PErvasive Data GRoup of the Department of Electronic, Information, and Bioengineering at Politecnico. His research interests include context-aware systems, complex event processing systems, data stream management systems, and RDF/OWL reasoning systems.