UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

SEBD 2014

22ND ITALIAN SYMPOSIUM ON ADVANCED DATABASE SYSTEMS

June 16th - June 18th 2014

Towers Hotel Stabiae Sorrento Coast

EUSTEMA®
Making Innovation

CSP
thinking evolution

Context-Aware Software Approaches: a Comparison and an Integration Proposal

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

Discussion Paper

Abstract. In this paper, we claim that there are complementary features which can bring different paradigms in the DM and PL domains to a fruitful cooperation in building Adaptive Systems. The data tailoring capabilities embedded in the PerLa sensor data management language have been extended, thus applying context-awareness to generic system operations; these operations, in turn, can be expressed as *Layers* in a Context Oriented Programming language.

1 Introduction

Scientists and engineers have made significant efforts to design and develop *adaptive systems*. These systems are able to adapt their behavior according to specific changes to the running context [5]. There are two main categories of behavioral variations: *internal changes* include all preconceived events that occur within the system and are able to modify the system state; *external changes* are related to the environment and to user interactions and are caused by external events. These systems address adaptivity in various respects, including performance, security, fault management, etc... 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*. Self-adaptive software aims at adjusting various entities or attributes in response to changes in the self and in the context of a software system. By self, we mean the whole body of the software, while the context encompasses everything in the operating environment that affects the properties of the system and its behavior.

The central concept of adaptivity can be summarized by this definition [4]: "Adaptive systems are able to adjust their behavior in response to their perception of the environment and of the system itself".

On the other hand, a system able to retrieve data, to build and manage contexts and to properly reason on them, is called **context-aware**. In context-aware systems, context follows these pieces of the whole environment in which it operates: *a) Computing environment*: available processors, devices accessible for user input and display, network capacity, connectivity and costs of computing. *b) User environment*: location, collection of nearby people, and social situation. *c) Physical environment*: all external phenomena relevant to the system. In [11] a context-aware system is defined as follows: "A system that provides services or information to the users according to the context".

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 [3], while adaptivity refers to the execution of behavioral variations in response to changes of all the entities that can affect the behavior of the system, also the internal software itself. Therefore Adaptivity and context-awareness are complementary in building pervasive applications.

Starting from this premise, the PerLa language and middleware, designed for managing data in Wireless Sensor Networks (WSN) [14], have been extended with the ability of declaring and managing contexts [17], thus allowing to apply context-awareness to generic system operations; these operations, in turn, can be expressed as *Layers* in a Context Oriented Programming language [13]. Throughout this paper we consider the classical application of keeping an office room climate comfortable under several environmental constraints as an example of the possibility to combine the data and the software-oriented approaches in building adaptive systems. COP has been preferred to other approaches owing to its layered structure which better allows the integration process, as we shall see in Section 3.

The rest 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; finally, we discuss the conclusions in Section 4.

2 The background

2.1 Context: the Data Management view

2.1.1 A Pervasive Data Management Framework As extensively presented in [14] 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 [10], 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 behaviors of the devices. Perla currently supports three types of queries: *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; *High Level Queries (HLQ)*, which define the high level elaboration involving data streams coming from multiple nodes, and are equivalent to SQL operations on data streams and *Actuation Queries (AQ)*, which provide the mechanisms to change parameters of the devices or to send commands to actuators. 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 saving.

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

Listing 1.1. The Context Dimension Tree and the fire context

The introduction of the CDT context model within the PerLa framework makes it possible the description of how their combined action can be used to achieve a context-aware pervasive systems management.

2.1.2 Embedding context into PerLa As mentioned in the introduction, "pushing" the knowledge of context from the application down to the middleware level was the main contribution of our previous work [17], [16]. To achieve our goals we designed and implemented: *i)* 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; *ii)* the **Context Manager (CM)**, able to maintain and manage the declared CDT, detect active contexts and performs the desired actions.

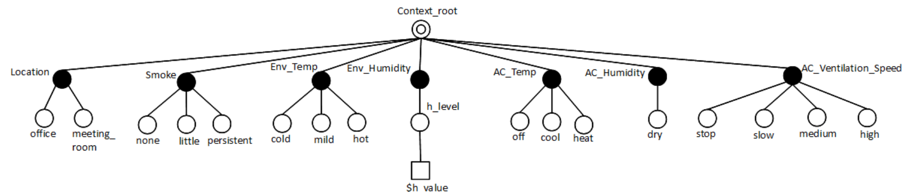

Fig. 1. Context Dimension Tree (CDT)

The syntax of the CL has been divided into two parts, called *CDT Declaration* and *Context creation*, both presented in details in [17].

CDT Declaration :

allows the user to specify the CDT, i.e. all the application-relevant dimensions and the values they can assume. The CDT for our example is shown in Fig. 1.

Context creation :

allows the designer to declare a context on a defined CDT and to control its activation by defining a **contextual block**, which is composed by four fundamental parts, called **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 chose the actions, if any, to be performed when the declared context is no more active;
- **REFRESH component**: instructs the middleware on how often the necessary controls must be performed.

In Listing 1.1 we report a block declaration for a possible context that represents the rise of a possible dangerous situation (a fire alarm).

Since the number of possible contexts has a combinatorial growth with the number of dimensions and concept nodes, the syntax of the PerLa language allows to separate the block components into one or more *partials*, thus relieving the designer from the task of declaring a very large number of context [17].

2.2 Context: the Programming Languages view

Several programming paradigms have been proposed in the literature [13] in order to support Context-awareness in object-oriented and modular programming languages.

In the following we briefly compare the features of three approaches: *Aspect-Oriented Programming (AOP)* [8], *Context-oriented Programming (COP)* [1], [7] and *Behavioral Programming (BP)* [6]; we shall refer, for each one of these paradigms, to the Java language and to its respective libraries: *AspectJ*, *Jcop* and *Bpj*.

Context Oriented Programming Context-oriented Programming [13], [2], [7] enables changes in the behavior of the application depending on the current context. 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 and deactivated at run-time, enacting a behavioral change; it represents the modularization unit of such piece of behavior

Aspect Oriented Programming Contrary to COP, which was born exactly to enable context-dependent adaptivity, the main goal of AOP is 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.

Behavioral Programming BP 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.

2.2.1 Paradigm Comparison

The main general features of the three paradigms are summarized in table 1 and shortly commented in the following.

Features / Language	Need of new syntactic constructs	Support for dynamic requirements	Scoping	Design simplification	Sw modularity	Incremental development	Triggering events	Context management	Behavioral variations
COP	X		X	X	X	X	sequence	multiple, internal, external	layer
AOP	X	X	X	X	X	X	single	multiple, internal, external	aspect
BP					X	X	sequence, internal	multiple, by events	B-thread

Table 1. Language paradigms comparison

Syntax: AOP and COP require special constructs (new language keywords introduction) to enforce their semantic features; their standard code is mixed respectively with "aspect" and "contextual" code. BP instead provides only new semantic features to the underlying programming language.

Support: AOP was born to deal with cross-cutting concerns through aspects weaved at compile-time; however, a lot of frameworks have been developed to make AOP dynamic. COP directly provides all the features necessary to perform context-based behavioral variations at run-time.

Scoping: With dynamic layers composition, behavioral variations activation in COP is dynamically scoped. In static AOP, scope is related to *pointcuts* and *join-points*. In dynamic AOP scope is also related to *aspects*. In BP, scope follows the same rules of the underlying programming language.

Design: COP avoids the definition of ad-hoc software components to perform context-based behavioral variations. AOP also allows to reduce the impact of application design through *aspects*. The design phase is a crucial part of the development of an application implemented with BP: it requires software engineers to correctly design the application scenarios.

Software modularity: AOP allows to encapsulate cross-cutting concerns in dedicated modules separate from others. COP is more suitable for performing behavioral variations, rather than for software structuring. BP allows for code scenarios (*behaviors*) in dedicated software components (*b-threads*), so it can be considered a modular programming technique.

Incremental development: One of the main advantages of BP is the ability to structure the application in modules and to incrementally implement it in agreement with requirements and scenarios. Also COP supports software evolution thanks to the possibility to embed layers in classes. Both static and dynamic AOP heavily focus on software modularization, they are a good solution for software evolution.

Triggers: BP provides the features to trigger behaviors by sequences of internal events while in AOP only a single event at a time can be considered, due to the restriction imposed by join-points. In COP the number of triggering events depend from the chosen framework.

Context COP is designed properly to combine both external and internal information in order to provide behavioral variations. AOP, in its dynamic

version, provides all features necessary to handle both external and internal information. Context for BP is instead totally determined by events.

Behavioral variations The main difference between COP and BP is that COP makes more or less deep changes to the normal behavior whereas BP enforces the synchronized alternation of different behaviors, which can also run simultaneously. Finally Dynamic AOP performs behavioral variations in a way that could be substantially equivalent to COP, also obtaining quite similar results.

3 Integrating PerLa and COP

COP focuses 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.

On the other hand, 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.

At the current development stage, PerLa contextual features mainly focus on data, and the actions that can be performed in response to contextual changes are limited to the execution of PerLa statements [17]. 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.

Context management

As mentioned in Sections 2.1.1, PerLa with its queries allows to monitor the entire life cycle of the information working with several data streams, produced by the sensing devices. With CL, the designer can define a sort of *contextual dynamic view* on a data stream. On the contrary, COP is not directly aware of how information is provided; in fact it is not directly responsible of sensors, but it uses the information provided by them, to perform behavioral variations.

Both PerLa and COP are suitable for context distribution: the CDT model deals naturally with contexts belonging to different groups of sensors and to distributed instances of the application; in COP several threads may exist for each local instance and they adapt their behavior differently. In COP it may become necessary to introduce dedicated components to monitor continuous data sources in order to provide contextual information, starting from rough data provided by sensors, and to decide which layers activate on the application.

Enacting behavioral variations

The semantics of the *ON ENABLE* and *ON DISABLE* clauses of PerLa CL could apparently look similar to that of the ***with*** and ***without*** statements of COP. Considering the *ON ENABLE* clause, it sets which data must be provided, and then it utilizes these data to change some parameters, according to the corresponding concept in the CDT.

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 context *SmokeMonitoring* is actually an extension of the *fire* context in Listing 1.1 since it performs additional actions (e.g. it stops the ventilation not

```
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 speed = 0
ON DISABLE (SmokeMonitoring) :
DROP SmokeMonitoring
SET PARAMETER air_outlet = FALSE
SET PARAMETER alarm = FALSE
```

Listing 1.2. The SmokeMonitoring example in PerLa

```
context SmokeRisk {
   in(SmokeMonitoring sm) && when(SmokeMonitoring.smokeRisk()) {
     with(SmokeLayer);
   } }
class ActiveActuators {
   public activeAlarm()  {
     // When this method is called by thread FireMonitoring,
     // the fire alarm is activated
   }
   ...
   layer SmokeLayer {
     activeAlarm() {
       // If the layer is activated, the air outlet will be opened
       Outlet.sendOpenCmd();
     } } }
```

Listing 1.3. The SmokeMonitoring example in COP

to spread the smoke and it opens the outlet to let it flow away) in order to be more effective in case of fire (Listing 1.2).

To implement the example in COP, 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. With COP, we can only define a ***context*** structure to encapsulate changes related to the smoke detector. In fact, we can assume that the activation of the fire alarm is the "normal" behavior, whereas the activation of both actuators represents a variation (Listing 1.3).

The architecture of PerLa is shown in Fig. 2; a monitor thread checks only the part of the global context useful for the *SmokeMonitoring* context activation; so the *SmokeMonitoring* context stream contains all and only the values related to temperature, humidity and air conditioner fan speed of the sensors located in the offices. If a sensor is not located in an office, its data are completely discarded in this context stream (Since the *EXECUTE IF* clause tells the system to check data gathered from offices' sensors). If a sensor gathers more data than

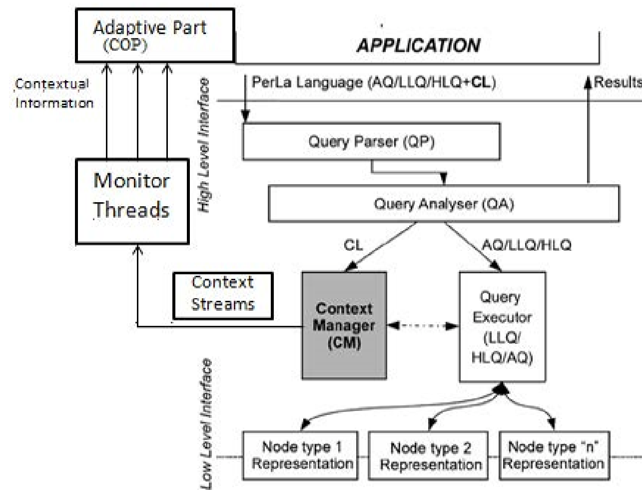

Fig. 2. The system architecture

those required, the context unrelated data are discarded from the *SmokeMonitoring* context stream (another context stream must be declared if this data are still needed by another context). This solution however does not provide a real integration between the two actors.

An alternative solution may be to extend the PerLa CL with the option of directly launching scripts, programs, functions or even applications from the context queries, with a mechanism analogous to a Remote Procedure Call (RPC). In this way, a direct connection between context information and behavioral variations is created. The application designers can define a different independent behavior for each context, not needing to implement intermediate components for data and context management. The PerLa middleware takes the responsibility of every aspect related to context management. The CM acquires a decisive role: it must supervise the context activation state in order to avoid any possible inconsistent situation, which, in this scenario, could be impossible to handle.

In Listing 1.4, the new clauses *LOAD COP CONTEXT/DROP COP CONTEXT* have been introduced, in order to tell the system to activate/deactivate the related COP context (in this example, the COP code refers to Listing 1.3). The activated COP context is then responsible of the relevant layer activation (e.g. *SmokeLayer*) and of managing all the related threads and procedures, while PerLa only manages the COP context activation/deactivation and controls the switching among different contexts. Notice that the COP context activation is not the only action performed by the system, since it firstly sends the air conditioner the “stop” (setting its speed to “0”) command using the default PerLa *SET PARAMETER* clause.

```
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)
```

```
LOAD COP CONTEXT SmokeRisk
SET PARAMETER speed = 0
ON DISABLE (SmokeMonitoring) :
DROP SmokeMonitoring
DROP COP CONTEXT SmokeRisk
```

Listing 1.4. The integrated COP-PerLa approach

This mechanism can be compared to that of BP, i.e. activating a new independent behavior when required. However, while in BP the alternation of different behaviors is based on the occurrences of events or sequence of events, in PerLa context and behavior can be regarded as a unique indivisible "block", entirely driven by sensor data.

Further examples and details about the PerLa and COP integration can be found in [15].

4 Conclusions and future work

After a review of the approaches taken by the Data Management and by the Programming Languages communities to the problem of building Context-aware Self-adapting software, in this work we propose a hybrid solution based on the PerLa framework and language to design, declare and manage Context, and on Context Oriented Programming language JCOP to write complex layered procedures where each layer is bound to a specific context.

A further step towards the building of comprehensive Context-aware systems could be the inclusion of Context-aware Services [9] in the picture, as synthetically depicted in Figure 3.

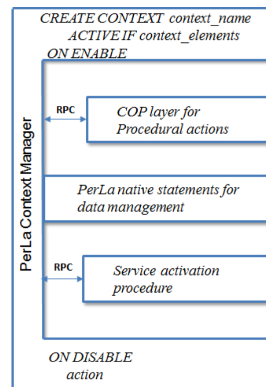

Fig. 3. General structure of a C-A self-adapting system

Acknowledgments

We acknowledge the work of Ing. Matteo Rovero in surveying the Aspect-, Behavior-, and Context-oriented programming paradigms in fulfillment of his Master thesis [12]. This work was partially funded by the European Commission, Programme IDEAS ERC, Project 227977-SMSCom and Industria 2015, Programma n° MI01_00091 SENSORI.

References

1. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-oriented Programming Languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM Press.
2. Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, pages 50–65, 2010.
3. Cristiana Bolchini, Carlo Curino, Giorgio Orsi, Elisa Quintarelli, Rosalba Rossato, Fabio A. Schreiber, and Letizia Tanca. And what can context do for data? *Commun. ACM*, 52(11):136–140, 2009.
4. Betty HC Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.
5. A.K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, 2001.
6. David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.
7. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
8. Gregor Kiczales and Al. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
9. Zakaria Maamar, Djamal Benslimane, and Nanjangud C. Narendra. What can context do for web services? *Commun. ACM*, 49(12):98–103, 2006.
10. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
11. Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, pages 92–99. IEEE, 1998.
12. Matteo Rovero. Context-aware application development: A comparison of different approaches. Technical report, Master Thesis, DEIB, Politecnico di Milano, 2013.
13. Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.
14. Fabio A. Schreiber, Romolo Camplani, Marco Fortunato, Marco Marelli, and Guido Rota. Perla: A language and middleware architecture for data management and integration in pervasive information systems. *IEEE Trans. Software Eng.*, 38(2):478–496, 2012.
15. Fabio A. Schreiber and Emanuele Panigati. Context aware data management and context oriented programming: is convergence possible? Technical Report 2014.7, Politecnico di Milano – Dipartimento di Elettronica, Informazione e Biongegneria, March 2014.
16. Fabio A. Schreiber, Letizia Tanca, Romolo Camplani, and Diego Viganó. Towards autonomic pervasive systems: the PerLa context language. In *Electronic Proc. 6th NetDB*, pages 1–7, 2011.
17. Fabio A. Schreiber, Letizia Tanca, Romolo Camplani, and Diego Viganó. Pushing context-awareness down to the core: more flexibility for the PerLa language. In *Electronic Proc. 6th PersDB*, pages 1–6, 2012.