Scalable Processing of Context Information with COSMOS

Denis Conan[1], Romain Rouvoy[2], and Lionel Seinturier[3]

[1] GET / INT, CNRS Samovar
9 rue Charles Fourier, 91011 Évry, France
Denis.Conan@int-evry.fr
[2] University of Oslo, Department of Informatics
P.O.Box 1080 Blindern, 0316 Oslo, Norway
rouvoy@ifi.uio.no
[3] INRIA-Futurs, Projet Jacquard / LIFL
Université des Sciences et Technologies de Lille (USTL)
59655 Villeneuve d'Ascq, France
Lionel.Seinturier@inria.fr

Abstract. Ubiquitous computing environments are characterised by a high number of heterogeneous devices that generate a huge amount of context data. These data are used to adapt applications to changing execution contexts. However, legacy frameworks fail to process context information in a scalable and efficient manner. In this paper, we propose to organise the classical functionalities of a context manager to introduce a 3-steps cycle of data collection, interpretation, and situation identification. We propose the COSMOS framework, which is based on the concepts of *context node* and *context management policies* translated into software components in software architecture. This paper presents COSMOS and evaluates its efficiency throughout the example of the composition of context information to implement a *caching/off-loading* adaptation situation.

Key words: Mobile computing, context, architecture, component.

1 Introduction

Ubiquitous computing environments are characterised by an high number of mobile devices, wireless networks and usage modes. Distributed applications for such environments must continuously manage their execution context in order to detect the conditions under which some adaptation actions are required [6]. This execution context contains various categories of observable entities, such as operating system resources, user preferences, or sensors. Data coming from these entities are often related and aggregated to provide a high-level and coherent view of the execution context. Besides, the management of such a view is under the responsibility of a context manager, which is furthermore in charge of identifying situations where applications need to be adapted.

Two categories of approaches exist in the literature for context management: The ones that are "user-centred", and those based on "system" supervision. This paper wishes to reconciliate both by proposing a component-based framework for context management.

With the "user-centred" approach, context includes the user terminal, nearby small devices, such as sensors and devices reachable through a network. Existing works in the literature [6,10,17] divide context management into four functionalities: *Data collecting*, *data interpreting*, *condition-for-change detection*, and *adaptation usage*. The central point of existing frameworks consists in computing high-level abstract information about the context from some low-level raw data. In our opinion, two weak points can be identified in these frameworks: *(i)* the difficulty for composing context information and *(ii)* scalability, either in terms of the volume of processed data and/or in terms of the number of supported client applications.

The "system" supervision approach has been studied thoroughly in the past [15]. This approach is gaining again some attention as clusters, grids [2,4] and ubiquitous computing [7,9] are becoming mainstream. Existing solutions consist in instrumenting operating systems and collecting data. The weak point of frameworks in this approach is often that the collected data are numerical and too low-level for being used efficiently by adaptation policies.

This paper proposes COSMOS (*COntext entitieS coMpositiOn and Sharing*), which is a component-based framework for managing context data in ubiquitous environments. The applications we are targeting are, for example, tourist computer-based guides with contextual navigation or applications with contextual annotations, such as multi-player games. The context management provided by the COSMOS framework is *(i)* user and application centred to provide information that can be easily processed, *(ii)* built from composed instead of programmed entities, and *(iii)* efficient by minimising the execution overhead. The originality of COSMOS is to combine component-based and message-oriented approaches for encapsulating context data, and to use an architecture description language (ADL) for composing these context data components. By this way, we hope to foster the design, the composition, the adaptation and the reuse of context management policies.

This paper is organised as follows. Section 2 motivates the definition of the COSMOS framework for composing context information. Section 3 presents the design of the COSMOS framework, starting from the concept of a context node, and then proceeding by presenting the design patterns that are proposed for composing context nodes. Section 4 presents the case study of a *caching/offloading* adaptation situation. Sections 5 and 6 reports on the implementation of the COSMOS framework and evaluates its performances, respectively. Section 7 presents some related work. Finally, Section 8 concludes this paper and identifies some perspectives.

2 Overview and Motivations

This section proposes a general overview of COSMOS, which is our framework for context management. The architecture of the COSMOS framework is illustrated in Figure 1. COSMOS is divided into three layers: the Context collector layer, the Context processing layer, and the Context adaptation layer.

The lower layer of the COSMOS framework defines the notion of a context collector. Context collectors are software entities that provide raw data about the environment. These pieces of data come from operating system probes, network devices (*e.g.*, sensors), or any other kind of hardware equipment. The notion of a context collector also encompasses information coming from user preferences. The rationale for this choice is that context collectors should provide all the inputs needed to reason about the execution context.

The middle layer of COSMOS defines the notion of a context processor. Context processors filter and aggregate raw data coming from context collectors. The purpose is to compute some high-level, numerical or discrete, information about the execution environment. The status of the network link (*e.g.*, strongly connected, weakly connected, or disconnected) is an example of the piece of information outputted by a context processor. Data provided by context processors are fed into the adaptation layer.

The upper layer of COSMOS is concerned with the process of decision making. The purpose is to be able to make a decision on whether or not an adaptation action should be planned. The adaptation layer is thus a service that is provided to applications and that encapsulates the situations identified by context nodes and processors.

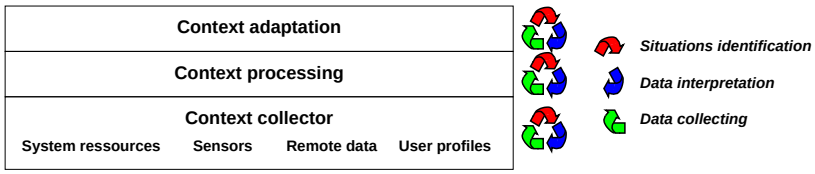

Fig. 1. Architecture of a COSMOS context manager

To provide a scalable context processing framework, the design of COSMOS has been motivated by three founding principles: *separation of concerns*, *isolation* and *composability*. We elaborate on these principles in the next paragraphs.

The notion of separation of concerns promotes a clear separation of functionalities into different modules. In the case of the COSMOS framework, the activities we want to separate are related to the grabbing of context information, the interpretation of this information, and the decision making process. The actions undertaken in these three cases correspond to three separate software engineering domains. The context collector layer addresses issues that are related to network technologies with solutions, such as UPnP for discovering and

connecting devices, to distributed systems with, for example, data consistency protocols and network failure detectors, and to operating systems for information about hardware devices. Although separate, these three domains (network, distributed systems and operating systems) are close. The context processor layer addresses issues that are quite different. The techniques used to aggregate, filter, and reason about context data are related to domains, such as software engineering, databases, or information systems. One can also envision case studies where inference engines are used to implement the process of decision making. Finally, the context adaptation layer is directly related to the application being developed. The adaptation scenarios which are handled by this layer are domain-specific. The fact that all these concerns are quite different motivated the definition of the three above-mentioned layers.

The second principle which motivated the definition of a 3-layers architecture for the COSMOS framework, is to isolate the part that interacts with the operating system, from the rest of the framework and of the application. Although adaptation actions should not be too frequent, processing context information is an activity that must be conducted more often, while data gathering is a third activity that must be continuous. Thus, we have three different activities with different frequencies. We decouple as much as possible these activities in order to obtain a non-blocking and usable framework. Each activity is conducted in one of the three layers, which has its own autonomous life cycle: Each layer performs a 3-steps cycle of data collection (from its lower layer), processing, and decision making (for its upper layer). This principle is illustrated on the right side of Figure 1.

Composability is the third principle that motivated the design of the COSMOS framework. We want to obtain a solution where context information can be easily assembled. By being able to compose context information, we hope to foster the reuse of context management policies. For this, we adopt a component-based software engineering approach: As explained in the next section, context information is reified into software components. By connecting these components, we define assemblies that gather all the data needed to implement a specific policy.

3 Building Context Management Policies from Context Nodes

In this section, we present the composition of context information with COSMOS. Sections 3.1 and 3.2 introduce the concept of context nodes, their properties and parameters. Next, Section 3.3 defines the generic architecture of context nodes. Finally, Section 3.4 is focused on the design of the overall architecture of COSMOS, that is the relationships between the context nodes.

3.1 Concept of context node

The basic structuring concept of COSMOS is the *context node*. A context node is a context information modelled by a component. Context nodes are organised

into hierarchies with the possibility of sharing. The graph of context nodes represents the set of context management policies defined by client applications. The sharing of a context node (and by implication of a partial or complete hierarchy) corresponds to the sharing (of a part or the whole) of a context management policy.

COSMOS provides the developer with pre-defined generic context nodes: *Elementary nodes* for collecting raw data, *memory nodes*, such as averagers, translation nodes, *data mergers* with different quality of service, *abstract or inference nodes*, such as additioners, thresholds nodes, etc. Note that in a classical context manager architecture the first nodes constitute the collectors, most of the other ones are part of the interpretation layer, while the last thresholds based ones serve to identify situations. In COSMOS, each class of nodes can be used in every layers, hence leveraging the expressiveness power of context policies.

3.2 Properties of a context node

Passive vs. active. A passive node obtains context information upon demand. A passive node must be invoked explicitly by another context node (passive or active). An active node is associated to a thread and initiates the gathering and/or the treatment of context information. The thread may be dedicated to the node or be retrieved from a pool. A typical example of an active node is the centralisation of several types of context information, the periodic computation of a higher-level context information, and the provision of the latter information to upper nodes.

Observation vs. notification. The observation reports containing context information are encapsulated into messages that circulate from the leaves to the root of the hierarchies. When the circulation is initiated at the request of parent nodes or client applications, it is an observation. In the other case, this is a notification.

Blocking or not. During an observation or a notification, a node that treats the request can be blocking or not. During an observation, a non-blocking context node begins by requesting a new observation report from each of its child nodes, and then updates its context information before answering the request of the parent node or the client application. During a notification, a non-blocking node computes a new observation report with the new context information just being notified, and then notifies the parent node of the client application. In the case of a blocking node, an observed node provides the most up-to-date context information that it possesses without requesting child nodes, and a notified node updates its state without notifying parent nodes. In addition, a node can be configured for a unique observation or notification if its state is immutable. Finally, the observation of a node can raise exceptions, for instance when the physical resource is not present or in case of a configuration problem. On demand, the thrown exception can be masked to parent nodes or client applications, and default values can be provided in that case.

3.3 Architecture of a context node

The architecture of a context node is component-based. This architecture is implemented with the FRACTAL component model [3] and its associated tools: the FRACTAL ADL architecture description language, and the DREAM [13] message-oriented component library. We take advantage of the two main characteristics of FRACTAL which are to provide a hierarchical component model with sharing. However, nothing is specific to FRACTAL in our design and COSMOS could be implemented with any other component model supporting these two notions.

Each context information is a context node which extends the composite abstract component ContextNode (see Figure 2). Pull and Push are interfaces for observation and notification. A ContextNode contains at least an operator (primitive abstract component ContextOperator), and is connected to the message-oriented communication service provided by the DREAM framework. The properties introduced in Section 3.2 become component attributes of ContextOperator. By default, nodes are passive (isActiveXxx = $false$), non-blocking (xxxThrough = $true$), and the observation reports are mutable (xxxOnlyOnce = $false$). The attributes nodeName and catchObservationException serve to name the context node, and to specify whether the exceptions which may be thrown must be forwarded to parent nodes (the default value is $false$), respectively.

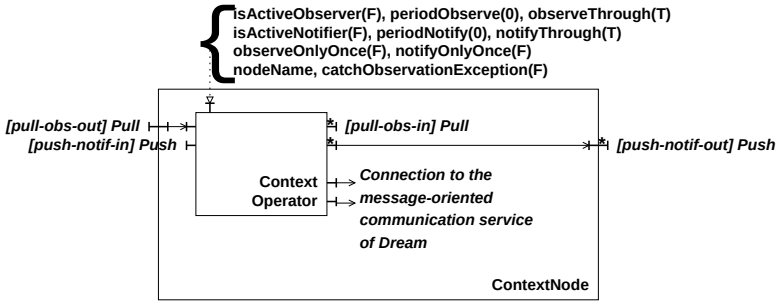

Fig. 2. Abstract Composite ContextNode.

Context nodes are then classified into two categories. Leaves of the hierarchy import context information from a lower layer of the context management architecture. This lower layer may be the operating system or another framework, built with COSMOS or not, component-oriented or not. For instance, a WiFi resource manager can obtain the corresponding context information directly from the operating system (through system calls) or can encapsulate a (legacy) framework dedicated to the reification of system resources. Nodes of the graph that are not leaves, contain one or several other context nodes. For instance, a context node may compute the overall memory capacity of a terminal by encapsulating

two other context nodes, the first one computing the average free memory and the second one computing the average free swap.

3.4 Architecture of COSMOS

COSMOS proposes three design patterns to compose context nodes. These are architectural design patterns which organise the collaboration between context nodes to implement the context management policy. The four patterns that are used by COSMOS are: Composite, Factory method, Flyweight and Singleton.

The hierarchical composition of context nodes is achieved with the "Composite" [11] design pattern. This design pattern homogenises the definition of the architecture and allows defining elements composed of several sub-elements, which may be themselves either composite or primitive elements. Hierarchies built in COSMOS take advantage of nodes composition for inferring higher-level context information. The Composite pattern simplifies the composition of context nodes and the management of their dependencies.

Each node of the hierarchy encapsulates a particular treatment on the information provided either by child nodes or by encapsulated primitive components in the case of leaves. The context nodes apply a component-oriented version of the design pattern "Factory method" [11]. The skeleton of a context node is defined as the assembly of a context operator (extension of ContextOperator) with, on the one hand, the components for the extra-functional services and on the other hand, the child nodes. Thanks to this approach, the definition of a context node remains simple. In addition, the internal object-oriented design of the primitive component ContextOperator also follows the design pattern "Factory method" (the object-oriented version). Through its server interfaces, this component defines generic (resp. abstract) methods to overload (resp. implement). The algorithms for observing and notifying are always the same. Thus, the skeletons of theses algorithms are generic and delegate specific treatments to sub-classes.

The system resources reified in the nodes of the hierarchy can be shared by several context nodes since the leave nodes may contain lots of elementary context data. This is precisely the purpose of the design pattern "Flyweight" [11] to efficiently share numerous fine-grained objects. By applying a component-oriented version of this design pattern, context nodes in COSMOS can efficiently share any child node of the hierarchy.

4 Case study

In this section, we assess the expressiveness and the quality of context composition using COSMOS with a scenario from the domain of ubiquitous computing: Caching/off-loading (see Section 4.1) which is implemented with context nodes (see Section 4.2).

4.1 Caching/off-loading scenario

The scenario of the case study follows. We assume that the user of a mobile terminal executes a distributed application while roaming. The WiFi connection of the mobile terminal is subject to disconnections. In order to tolerate such disconnections, the middleware platform can be augmented with the capabilities of importing/caching application entities into a software cache. Another issue is the capability of exporting/off-loading application treatments on (more powerful) hosts of the wired network. In order to choose between caching and off-loading, the context manager computes the memory capacity as the sum of the average free memory plus the average free swap. The context manager also monitors the connection to the WiFi network. It detects disconnections and computes the adjusted bit rate (average bit rate during periods of strong connectivity). When the memory capacity is sufficient, but the adjusted bit rate low, caching is preferred. When the memory capacity is low, but the adjusted bit rate sufficient, off-loading is preferred. In the two other cases, the end-user or the middleware platform give their preferences (caching or off-loading). Once the decision is taken, connectivity information is used to detect the activation instants for caching/off-loading when the connectivity mode changes (from strongly connected to disconnected and *vice versa*).

4.2 Implementation with COSMOS context nodes

The implementation with context nodes of the above described scenario is illustrated in Figure 3. Every node is given an intuitive name expressing the context operator it contains. The edges of the graph model the composition and the sharing relationships. When the value of a property differs from the default case, this value is indicated next to the node: Active observations and notifications, blocking or non-blocking, etc. In the example, most of the actives nodes are observers; only the nodes that detect state changes (User preference's change detector and Connectivity detector) and decision changes (Decision stabilisation) notify their changes to parent nodes. Note that the Connectivity detector node is shared by two parents, one of them being not a direct parent. The WiFi manager is shared by three parent nodes. This is a blocking node. This choice has been made to avoid emitting system calls too frequently and thus to avoid freezing the user device.

The decision When caching/off-loading? requires a graph of approximately twenty context nodes. In COSMOS, developers have at their disposal raw numerical data: Swap size, free swap, free memory, WiFi link quality, etc., plus composition facilities that help in declaratively composing these data. The resulting solution is thus reusable for other use cases. Furthermore, developers are assisted in the management of extra-functional concerns: These tasks prove to be cumbersome, and indeed even not completely manageable. The strength of COSMOS thus lies into the separation of concerns: Separation of business concerns (relevant raw data and inference treatments) from extra-functional ones (system resource management for performance).

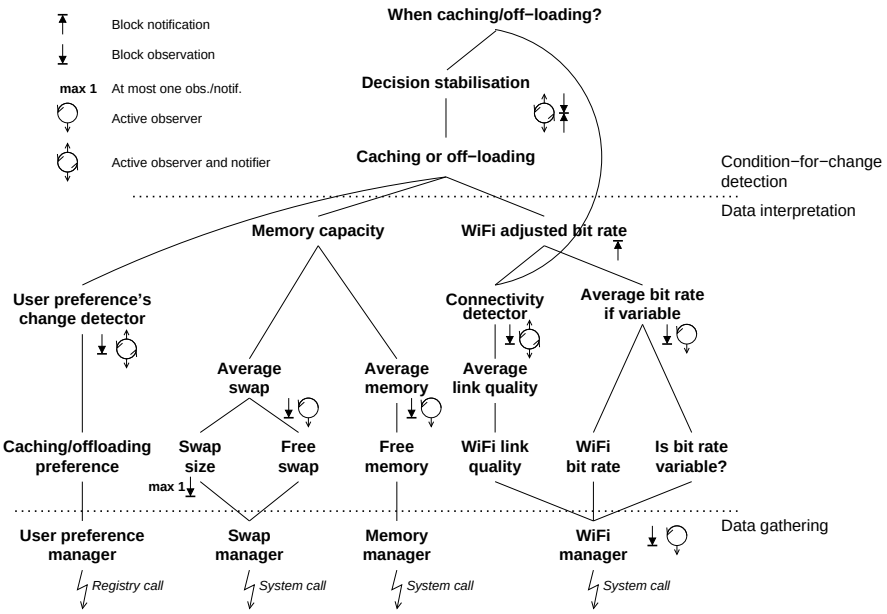

Fig. 3. Example of Composition of Context Nodes.

5 Implementation of COSMOS

The implementation of the COSMOS framework is based on three existing frameworks: FRACTAL, DREAM, and SAJE. FRACTAL [3] is the component model of the ObjectWeb consortium for open-source middleware. FRACTAL defines a lightweight, hierarchical and open component model (see `http://fractal.objectweb.org`). We use the Julia [3] version, which is a Java implementation of FRACTAL. We also take advantage of the numerous tools available for this component model, such as FRACTAL ADL, FPath, and Fraclet (a lightweight programming model). DREAM [13] is a library composed of several FRACTAL components. DREAM allows the construction of message-oriented middleware (MOM) and the fine-grained control of concurrency management with thread pools and message pools. Finally, SAJE [5] is a framework for gathering data from system resources, either physical (battery, processor, memory, network interface, etc.) or logical (sockets, threads, etc.). SAJE supports several operating systems: GNU/Linux, Windows XP, Windows 2000 and Windows Mobile 2003.

Implementing context adaptation policies with COSMOS consists in conducting two activities: *(i)* developing FRACTAL components for the context nodes that are resource managers linked with SAJE and for the context operators, and *(ii)* composing these components by using the FRACTAL ADL language. Furthermore, as described in Section 3.2, context nodes are defined to be highly configurable through numerous attributes (about ten attributes). The inherent drawback is the complexity of the configuration of a graph of context nodes, such

as the one presented in the example of Section 4.2 which contains about twenty nodes. To address this complexity, we use FPath, a language inspired from XPath and dedicated to the navigation into hierarchies of FRACTAL components.

A first version of COSMOS is available under the GNU LGPL license and can be downloaded from `http://picolibre.int-evry.fr/projects/cosmos`.

6 Performance Evaluation of the Prototype

The objective is to confirm experimentally the appropriateness of the component-based approach. Therefore, we make the distinction between the costs introduced by the reification of system resources by the framework SAJE and the costs due to the composition with COSMOS.

We have conducted performance measurements on a laptop PC with the following software and hardware configuration: 1.8GHz processor, 1GB of RAM, Compaq IEEE 802.11b WL110 card at 11Mbps, GNU/Linux Debian Sarge with the kernel 2.6.15, Java Virtual Machine Sun JDK 1.5 Update 6, and FRACTAL implementation Julia 2.1.3 (none of the execution optimisations activated). The results are presented in Table 1. Each test was run 10,000 times in order to obtain meaningful averages. A garbage collection and a warm-up phase occurred before each run. The unit of measure is the millisecond. When the measured values are less than one millisecond, the iterations number becomes 1,000,000. The configuration is the default one: passive nodes and non-blocking observations.

				Observation (ms)
a	SAJE	Free memory	Memory	0.038
	COSMOS	Memory manager	PeriodicMemory	0.045
b	SAJE	Quality of the WiFi link	WirelessInterface	14.0
	COSMOS	WiFi manager	PeriodicWireless	33.8
c	COSMOS	Example of Figure 3	WhenCachingOffloading—default config.	163.7
	COSMOS	Example of Figure 3	WhenCachingOffloading—Figure 3 conf.	4.7

Table 1. Performances of SAJE and COSMOS

The first series of measurements (see Table 1-a) concerns the extraction of the free memory information. With SAJE, the observation of the Memory object corresponds to an access to the Unix `/proc` file system (present in RAM) and to the initialisation of the data structures storing the information, that is to say less than 1ms. The differences between the observations with SAJE and with COSMOS (PeriodicMemory), which is evaluated to approximately 7μs, is the sum of (1) the cost of the calls to FRACTAL components (crossing the membrane and interception by controllers), (2) the extraction of context information from the SAJE object, and (3) the filling of the DREAM message chunk via the message manager component.

The second series of measurements (see Table 1-b) concerns the extraction of the quality of the WiFi link. The observation of the WirelessInterface SAJE object lasts longer than the observation of the Memory SAJE object because

the data of the WiFi interface are not present in RAM, but must be read from the network device. The observation of a PeriodicWireless component lasts longer since the context node extracts automatically all the available atttibutes (more than 30).

The last series of measurements (see Table 1-c) is the observation of the example of Figure 3 (component WhenCachingOffloading). It takes 163ms in the worst case: Every component is non-blocking. If the components are configured as presented in Figure 3, since the child components of WhenCachingOffloading block the observations, the observation time of WhenCachingOffloading becomes neglible (less than 5ms). This concludes that the component-based composition of context data not only pertinent but also efficient while preserving the context information accuracy.

7 Related Work

In this section, we compare COSMOS with the legacy frameworks dedicated to context monitoring, such as Phoenix and LeWYS. Then, we compare COSMOS with several middleware frameworks for context management.

Phoenix is a software framework for the observation of system resources for distributed applications deployed on clusters [2]. The architecture of Phoenix is composed of four parts: Observation agents, probes, broadcast primitives (into local networks), and a tool library. Observation agents can configure the observation frequency and multiplex the observations (by adjusting the frequency to the lowest requested value). Phoenix provides a dedicated language for describing an observation: Observable resource identifiers, comparison operators, first order logic and DELTA operators to measure the amplitude of variations. Phoenix provides only elementary operators: No memory or threshold operators, format translation, data merging, etc. However, the dedicated language approach for expressing observation requests could be used in the future evolution of COSMOS. In addition, Phoenix does not support the easy introduction of new operators.

LeWYS is a middleware framework for the supervision of clusters [4]. Its architecture encompasses probes that are deployed on all the computers of the cluster and a distributed system for notifying events. Even if LeWYS is built using FRACTAL, it does not support the composition of context data. For example, all the data retrieved by the probes are propagated without being filtered.

Context Toolkit is one of the first work on context management that was based on event programming and widget concepts introduced by GUI (*Graphical User Interfaces*) [10]. In the same framework, all the following functionalities are grouped: The interpreter for composing and abstracting context information, the aggregator for the mediation with the application, the service for controlling application actions performed on the context, and the discoverer that acts as a registry. Following the same philosophy, interpretation and aggregation functionalities have to be programmed in monolithic blocks: One interpreter and one aggregator per application, independently of the number of widgets and the level

of abstraction requested by the application. Finally, the management of system resources consumed by treatments is not addressed.

MoCA Context Service architecture [8] defines an access interface, an event manager, a context-type manager, and a context repository. The event manager design highlights the need for technical services, called orthogonal services, to improve performance. In addition, context data are typed and described using an XML-based model that builds a type system implemented as Java objects. Similarly to our work, the authors describe the need for using meta-information in order to leverage performance and scalability. However, since the authors transpose an ontology-based approach to an object-oriented one, the MoCA architecture does not separate the context management functionalities. For instance, the source of context data (local or remote) is described via an attribute rather than being described in the architecture. Contrariwise, with COSMOS, we apply the component-oriented approach both at the context manager architecture level and at the context node definition level. The XML-based model of MoCA is similar to a component descriptor with its attributes. But, since COSMOS uses an ADL, the specification becomes explicit and benefits from the expressiveness of the language and its tools. Finally, the authors propose to partition the context data space into views for improving the performance. In a component model with hierarchy and sharing, this feature is automatically available.

MoCoA provides an environment for building context-aware applications for ad hoc networks based on sentient objects [16]. Sentients objects have most of the characteristics of components. The low-level inference treatments are organised as data merging pipes. MoCoA only allows notifications, contrary to COSMOS that add observations. Pipe treatments are complemented with inference ones with facts and rules, which are inspired from artificial intelligence. The pipes are logically enclosed in sentients objects, including for the control of system resources' consumption. But, contrary to COSMOS, MoCoA neither details nor provides any means to externally specify these controls. Finally, the authors of MoCoA express the useness of an ADL to describe the composition of pipes and sentients objects as we propose in COSMOS.

The context manager of Draco [14] is organised around a database and an ontology broker. The component-based approach is chosen for its ability to dynamically adapt the context management system to changing conditions of applications' requirements and context devices. The objective is to deploy / undeploy on demand functional context management components, such as filtering, history or transformation. The drawback of this use of the Singleton design pattern for functional context management services is that it does not scale. On the contrary, in COSMOS, these fine-grained functional services are replicated and integrated into context nodes when necessary. Concerning the ontology orientation, the evaluation concludes *(i)* to the difficulty to define an optimal deployment due to the difficulty to estimate of the processing time for all context management activities, and *(ii)* to the difficulty to use an ontology broker on small devices.

In *Le Contexteur* [7], *Contexteurs* are software entities similar to data components, and their meta-data (describing the data quality) as well as their con-

trollers (modifying the configuration) are available for both inputs and outputs. A *Contexteur* is a Java class that is associated to an XML descriptor. Thus, the software framework builds, in an *ad hoc* manner, a container around the *Contexteur* component. This *ad hoc* component model is implicit and not configurable (*e.g.*, for managing system resources). For each *Contexteur* using at least an activity, the local resource consumption can not be controlled. Furthermore, the sharing of context nodes supported by COSMOS is not addressed by *Le Contexteur*. In addition, *Contexteurs* exchange control information in order to ask to stop or force the data notification for example. However, given that there is no explicit component model, it is impossible to introduce new configuration possibilities, such as some new attributes or control modes. In COSMOS, the structure and the life-cycle of components is finely managed by the FRACTAL controllers.

Last but not least, RCSM [17] is an object-oriented framework with an architecture similar to ours. Every context source (users, sensors, operating system, remote hosts) is separated. But, the authors do not tackle the issues of the synchrony of the treatments or of the control of system resources for context management. PACE [12] presents a different architecture in which context data are stored in a database. The meta-data (temporality, quality, etc.) are added either to context data or to relations between them. The authors indicate clearly that they did not have a look at issues such as scalability or performance. Concerning context modelling, the same authors prone the object or the ontology orientations as the two acceptable alternatives among the myriad of modelling methods. With COSMOS, we add the component orientation, which raises a limitation of the object orientation: A more formal specification of the dependencies between context entities thanks to the usage of an ADL.

8 Conclusion

Ubiquitous environments put some constraints on the design and the implementation of applications. Among other requirements, applications for such environment must be highly adaptable. Before adapting, the decision making process that leads to adaptation is a difficult issue for which few efficient solutions exist. This process is based on gathering, analysing and treating vast amount of physical and logical data produced by the execution environment. In this article, we propose the COSMOS framework for managing such context information.

The COSMOS framework introduces the notions of context nodes and context policies (see Section 3). Context nodes are designed and implemented as software components, and can be composed and assembled to form complex context management policies. The goal of such an assembling is to drive the adaptation of an application.

The COSMOS framework is architectured around three principles: the separation of context data gathering from context data processing, the systematic use of software components, and the use of software patterns for composing these components. The first principle allows proposing new scalable context manage-

ment architectures with several levels of cycles, each one being composed of successive "gathering / interpretation / situations identification" phases. The second principle, software components, allows reusing more easily context nodes and the processors in the context nodes. The third principle allows composing rather than programming context management policies. For that, we have selected, in Section 3.4, three well-know design patterns [11] that are recurrently used when designing adaptation policies: the Composite, the Factory method, the Flyweight and the Singleton design patterns. The novelty of our approach is to use these patterns for composing software components which represent context nodes and context processors.

Scalability has been a driving factor for the design of COSMOS. We believe that several elements participate to this result: the composability brought by software components, the fact that COSMOS is divided in three independent layers, the fact that components can be shared and can have different properties to reduce their intrusiveness (see Section 3.2) and that the execution overhead have been kept as low as possible (see Section 6). The COSMOS framework is implemented on top of the FRACTAL [3] component model and the DREAM component library [13].

As a matter of future work, we plan to adopt three directions. First, we believe that the COSMOS framework is one of the main services that lies at the core of a platform for adapting distributed applications in a mobile environment. We could therefore think of integrating COSMOS in such a platform. A second direction concerns the composition of context management policies. The issue is to be able to address situations where two or several policies have to cohabit in a same platform for a same set of applications. As the intersection between these policies may not be empty, it is then necessary to provide tools to detect and solve the conflicts that arise between these policies. A direction that can be investigated consists in defining a type system [1] such as the one existing for the DREAM component library [13]. A related issue consists also in the possibility of setting up repositories for context collector components in order to facilitate their sharing. Finally, a third research direction consists in defining a domain specific language (DSL) for designing the composition of context nodes and context processors. Such a DSL could reuse ideas from the WildCAT [9] framework.

Acknowledgements

The authors wish to thank the anonymous reviewers and (in alphabetical order) Djamel Belaïd, Sophie Chabridon, Bertil Folliot, Pierre Sens and Chantal Taconet for their detailed reading and their numerous remarks on this paper.

References

1. P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.-B. Stefani. Dream Types: A Domain Specific Type System for Component-Based Message-Oriented Middle-

ware. In *4th ESEC/FSE Workshop on Specification and Verification of Component-Based Systems*, Lisbon (Portugal), Sept. 2005.
2. C. Boutros Saab, X. Bonnaire, and B. Folliot. PHOENIX: A Self Adaptable Monitoring Platform for Cluster Management. *Cluster Computing*, 5(1):75–85, Jan. 2002.
3. É. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL Component Model and Its Support in Java. *Software—Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11):1257–1284, Sept. 2006.
4. E. Cecchet, H. Elmeleegy, O. Layaïda, and V. Quéma. Implementing Probes for J2EE Cluster Monitoring. *Studia Informatica*, 4(1):31–40, May 2005.
5. L. Courtrai, F. Guidec, N. Le Sommer, and Y. Mahéo. Resource Management for Parallel Adaptive Components. In *IEEE IPDPS Workshop on Java for Parallel and Distributed Computing*, pages 134–141, Nice, France, Apr. 2003.
6. J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. The disappearing computer: Context is Key. *Communications of the ACM*, 48(3):49–53, Mar. 2005.
7. J. Coutaz and G. Rey. Foundations for a Theory of Contextors. In *4th International Conference on Computer-Aided Design of User Interfaces*, pages 13–34, Valenciennes (France), May 2002. Kluwer.
8. R. da Rocha and M. Endler. Context Management in Heterogeneous, Evolving Ubiquitous Envrionments. *IEEE Distributed Systems Online*, 7(4), Apr. 2006.
9. P. David and T. Ledoux. WildCAT: a generic framework for context-aware applications. In *3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 1–7, Grenoble (France), Nov. 2005.
10. A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Special issue on context-aware computing in the Human-Computer Interaction Journal*, 16(2–4):97–166, 2001.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
12. K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam. Middleware for Distributed Context-Aware Systems. In *7th International Symposium on Distributed Objects and Applications*, LNCS, Agia Napa (Cyprus), Nov. 2005. Springer-Verlag.
13. M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), Sept. 2005.
14. D. Preuveneers and Y. Berbers. Adaptive context management using a component-based approach. In *5th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 3543 of *LNCS*, pages 14–26, Athens (Greece), June 2005. Springer-Verlag.
15. B. Schroeder. On-Line Monitoring: A Turorial. *IEEE Computer*, pages 72–78, June 1995.
16. A. Senart, R. Cunningham, M. Bouroche, N. O'Connor, V. Reynolds, and V. Cahill. MoCoA: Customisable Middleware for Context-Aware Mobile Applications. In *8th International Symposium on Distributed Objects and Applications*, volume 4275 of *LNCS*, pages 1722–1738, Montpellier (France), Nov. 2006. Springer-Verlag.
17. S. Yau, F. Karim, Y. Wang, B. Wang, and S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, July 2002.