Software Architecture Patterns for a Context-Processing Middleware Framework

Romain Rouvoy, Denis Conan, Lionel Seinturier

Software Architecture Patterns for a Context Processing Middleware Framework

Romain Rouvoy* Denis Conan† Lionel Seinturier‡

March 27, 2008

Abstract

Ubiquitous applications are characterised by variations of their execution context. Their correct operation requires some continual adaptations based on the observation of their execution context. The design and the implementation of these observation policies is then the cornerstone of any ubiquitous applications. In this article, we propose COSMOS which is a framework for the principled specification and composition of context observation policies. With COSMOS, these policies are decomposed into fine-grained units called *context nodes* implemented as software components. These units perform basic context-related operations (*e.g.*, gathering data from a system or network probe, computing threshold or average values) and are assembled with a set of well-identified architectural design patterns. In this article, COSMOS is motivated and illustrated with an example from the domain of mobile e-commerce applications.

Key words: Context management, software architecture, software components, design patterns.

1 Introduction

This article presents the insights of COSMOS, a component-based framework for managing context information in ubiquitous context-aware applications. COSMOS aims at supporting the design and the development of applications reacting to changes in their execution environment. Examples of such context changes are the (dis-)appearing of hardware or software resources, or the modifications in the user preferences. Due to the high diversity of context information required by this kind of applications, COSMOS relies on the *Component-Based Software Engineering* (CBSE) principles [15] to ensure the integration of context information. In particular, the framework combines the concepts of *software components* and *architectural design patterns* to define the foundations of its architecture. Software components provide an efficient encapsulation of the context information diversity while architectural design patterns define the skeleton of the context management policies.

In [3], we presented how context management policies have been introduced in COSMOS to identify contextual situations for which a reaction of the application is expected. The context management policies are described as hierarchies of context nodes using a dedicated composition language. The contribution of this article is to present the mapping of the composition language constructions to architectural design patterns used in COSMOS. While the use of a *Domain-Specific Language* (DSL) leverages the definition of context management policies, components and design patterns supports the dynamic reconfiguration and evolution of the context management policies once specified and deployed.

Although components are gaining more and more attention for designing and implementing middleware platforms [4, 12, 16], the identification of architectural design patterns has not been investigated in a similar way to object-oriented middleware [1, 14]. Thus, we propose to illustrate that well-known design patterns can be reused and applied at the architectural level to offer a better control over the architecture of

*University of Oslo, Department of Informatics — E-mail: rouvoy@ifi.uio.no
†Institut TELECOM, SudParis / UMR CNRS Samovar — E-mail: denis.conan@it-sudparis.eu
‡University of Lille, INRIA ADAM — E-mail: lionel.seinturier@inria.fr

COSMOS. In particular, we can use these design patterns to separate the various extra-functional concerns (*e.g.*, memory footprint, resource consumption, instance management) involved in a context management policy from the business concerns of context management, that is composition of context information.

This article introduces the foundations of a context node in Section 2 and a motivating scenario inspired from mobile computing is described in Section 3. Next, Section 4 describes the reification of well-known design patterns in the architecture of COSMOS. The framework is evaluated in Section 5, while Section 6 discusses some related work. Finally, Section 7 concludes and identifies some perspectives.

2 Foundations of COSMOS

This section summarises the basis of the proposed approach by presenting properties related to the basic building units of composition of a context policy, that is the context nodes (cf. Section 2.1), and an overview of the core micro-architecture of context nodes (cf. Section 2.2).

2.1 Concepts and properties of a context node

The basic structuring concept of COSMOS is the *context node* [3]. A context node is a context information modelled by a software component. Context nodes are organised into hierarchies to form context management policies. The relationships between context nodes are encapsulation and sharing. 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 policy.

Context nodes at the leaves of the hierarchy encapsulate raw context data obtained from collectors (*e.g.*, operating system probes, sensors in the vicinity of the device, user preferences in profiles, remote devices). The rationale for this choice (notably the fact that user preferences are considered as context data to be collected) is that context nodes should provide all the inputs necessary for reasoning about the execution context. The role of a context node is thus to isolate the inference of high-level context information from lower architectural layers responsible for the collection of context data.

Context nodes are also equipped with properties which define their behaviour with respect to the context management policy.

Passive or active. Each context node can be passive or active. An active node is equipped with an activity to execute a given task. A typical example of an active node is a node in charge of 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 and/or notification. Communication into the hierarchy of context nodes may be bottom-up or top-down. The former case corresponds to notifications sent by context nodes to their parents, whereas the latter case corresponds to observations triggered by a parent node.

Blocking or pass-through. A context node which receives data transmitted by a notification or an observation may be blocking or pass-through. Non-blocking nodes propagate observations and notifications. Pass-through nodes stop the traversal: For observations the most up-to-date context information is transmitted without polling child nodes, and for notifications, context data is used to update the state of the node but parent nodes are not notified.

COSMOS provides the developer with pre-defined generic context operators. They are organised following a typology: *Elementary operators* for collecting raw data, *memory operators*, such as averagers, translation operators, *data mergers*, *abstract or inference operators*, such as additioners or thresholds operators. The only programming is in the context operators. If a developer has at her disposal a sufficiently large library of context operators well targeted to her business, there should be no programming at all, but only declarative composition of context nodes.

2.2 Architecture of a context node

Each context node extends the abstract composite ContextNode depicted in Figure 1. The interfaces Pull and Push are the interfaces for the observation and the notification, respectively. The abstract composite ContextNode contains at least an operator (primitive abstract component ContextOperator) as well as the message and activity managers. The message manager is in charge of handling the observation and notification reports which are sent and received by the component on the Pull and Push interfaces. The activity manager provides the support for dealing with active components. Finally, nodes are equipped with attributes which characterise their behaviour with respect to the properties defined in the previous section. As illustrated, 9 different attributes are defined: nodes which are observers can be active (isActiveObserver $= true$) with a period (periodObserver), passive (isActiveObserver $= false$), or can be limited to just one observation (observeOnlyOnce $= true$); in addition, observer nodes can be blocking (observerThrough $= false$) or pass-through ($= true$). Note that the same set of properties can be defined for notifier nodes. Finally, the nodeName attribute holds the name of the context node.

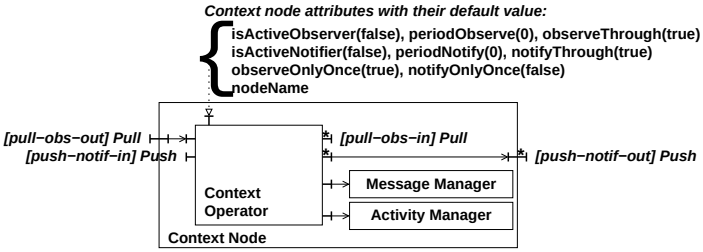

Figure 1: Core architecture of a Context Node component.

Context nodes are then classified into two categories: leaves and other nodes. Leaves of the hierarchy are ContextNodes extended to contain one or several components that receive context information from an external entity. This external entity may be the operating system or another framework, being 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 which are not leaves are extended to contain one or several other context nodes. For example, a context node may compute the battery charge state of a terminal by encapsulating two other context nodes, the first one computing the battery charge state and the second one computing the battery time left.

3 Motivating scenario

To illustrate the objectives of the COSMOS framework, we consider the scenario of a family shopping in a mall with a mobile device[1]. This application allows them to share information, to consult product prices, to download discount tickets, to be notified of advertisements, to access additional information and comments about a product, or to find the location of a product or a shop in the mall. The parents want their children to remain in the mall, with their devices connected as far as possible, so that everybody knows the location of the other family members. Nevertheless, children can disconnect for some periods of time in order to save their battery. While walking in the mall, the eldest girl sees an advertisement indicating that a dressing store proposes a RFID tag-based service for helping choose clothes.

All these features are based on different network technologies, such as Bluetooth or WiFi, and require the application to adapt itself depending on network connectivity and context information availability. As depicted in Figure 2, each adaptation situation (in the upper part of the picture) is isolated in a context tree with the possibility of sharing sub-trees between policies. Each adaptation situation relates to a particular

[1]This scenario is a use case of the French project “Cappuccino” (`http://www.cappucino.fr`).

functionality and focuses on a precise set of context information. For example, the WiFi download enabled situation is associated to the functionality supporting the download of a discount ticket, it allows the application to know when the functionality is available. The detection is performed by monitoring the quality of the WiFi link. The WiFi browsing enabled situation is built upon the previous one and allows the application to enable and to configure a browsing facility to access comments about a product or to find its location in the mall. The Bluetooth observation enabled situation is associated to the possibility to consult product prices and references in the vicinity of the client. This functionality is enabled when the device battery life expectancy is high enough. The Bluetooth availability situation is combined with user preferences to infer the Bluetooth notification enabled situation. This last situation leads to the configuration of the user application to be notified by the mall infrastructure of product advertisements. Finally, the Group membership service uses the disconnection and failure detectors to make the distinction between disconnections and failures, and to be informed about the location of the other members. This last definition illustrates the definition of sharing between hierarchies by reusing the Disconnection detector context node provided by the hierarchy WiFi browsing enabled.

To describe this policy, COSMOS provides a declarative language dedicated to the composition of context nodes. The core of this language is described below using the *Extended Backus-Naur Form* (EBNF):

```
Sensor ::= "sensor" SensorId "=" ComponentId [ Properties ] ";"
SensorId ::= Identifier
ComponentId ::= Identifier
Properties ::= "[" Property { "," Property } "]"
Property ::= "AO" | "AN" | "BO" | "BN" | "OO" | Identifier "=>" Value

Processor ::= "processor" ProcessorId "=" ComponentId [ Properties ] Dependencies ";"
ProcessorId ::= Identifier
Dependencies ::= "(" Dependency { "," Dependency } ")"
Dependency ::= ( SensorId | ProcessorId ) [ ".extract(" Chunks ")" ]
Chunks ::= ChunkId { "," ChunkId }
ChunkId ::= String

Task ::= "task" TaskId "=" NodeId { "," NodeId } ";"
TaskId ::= Identifier
NodeId ::= SensorId | ProcessorId
Thread ::= "thread" ThreadId "=" ThreadDef { "," ThreadDef } ";"
ThreadId ::= Identifier
ThreadDef ::= ( TaskId | NodeId ) "[" PositiveNumber "]"

Reporting ::= "reporting" ReportingId "=" ReportingDef { "," ReportingDef } ";"
ReportingId ::= Identifier
ReportingDef ::= NodeId [ "/" Xpath ]
```

The composition language we defined isolates the descriptions of functional concerns from extra-functional ones. Functional concerns are reified by two constructions **`sensor`** and **`processor`**. In addition to the component descriptor, sensors can be configured with the AO, AN, BO, BN, and OO properties to activate *active observation*, *active notification*, *blocking observation*, *blocking notification*, and *only once* mechanisms (cf. Section 2) as well as additional attributes specific to the resource manager (*e.g.*, `resourceName=>eth1`). Similarly, processors use the same configuration mechanisms, but have to describe context dependencies.

The specification of extra-functional concerns, such as resource consumption, is supported by the second part of the language. Constructions **`task`** and **`thread`** are used to specify *i)* the organisation of activities into hierarchies of tasks to execute and *ii)* the mapping of tasks to threads. Finally, the construction **`reporting`** supports the grouping of context report managers to reduce the memory footprint of the policies. Groups are defined using XPath requests[2] that apply to the nodes of the context hierarchy.

Using this composition language, the context policies `WiFi Download Enabled`, `WiFi Browsing Enabled`, and `Group Membership Service` depicted on the right-side of Figure 2 can be described as follows:

```
//Bottom: Data gathering
sensor WiFiMgr=WirelessInterfaceRM[BO,AO,resourceName=>eth1];
sensor HeartbeatMgr=PeriodicHeartbeatRM[BO,AO];

//Middle: Data interpretation
processor AverageWiFiQuality=AverageCO(WiFiMgr.extract("link-quality-chunk"));
processor AverageWiFiBitRate=AverageIfCO(WiFiMgr.extract("bit-rate-chunk","is-variable-chunk"));
```

[2] `http://www.w3.org/TR/xpath20`

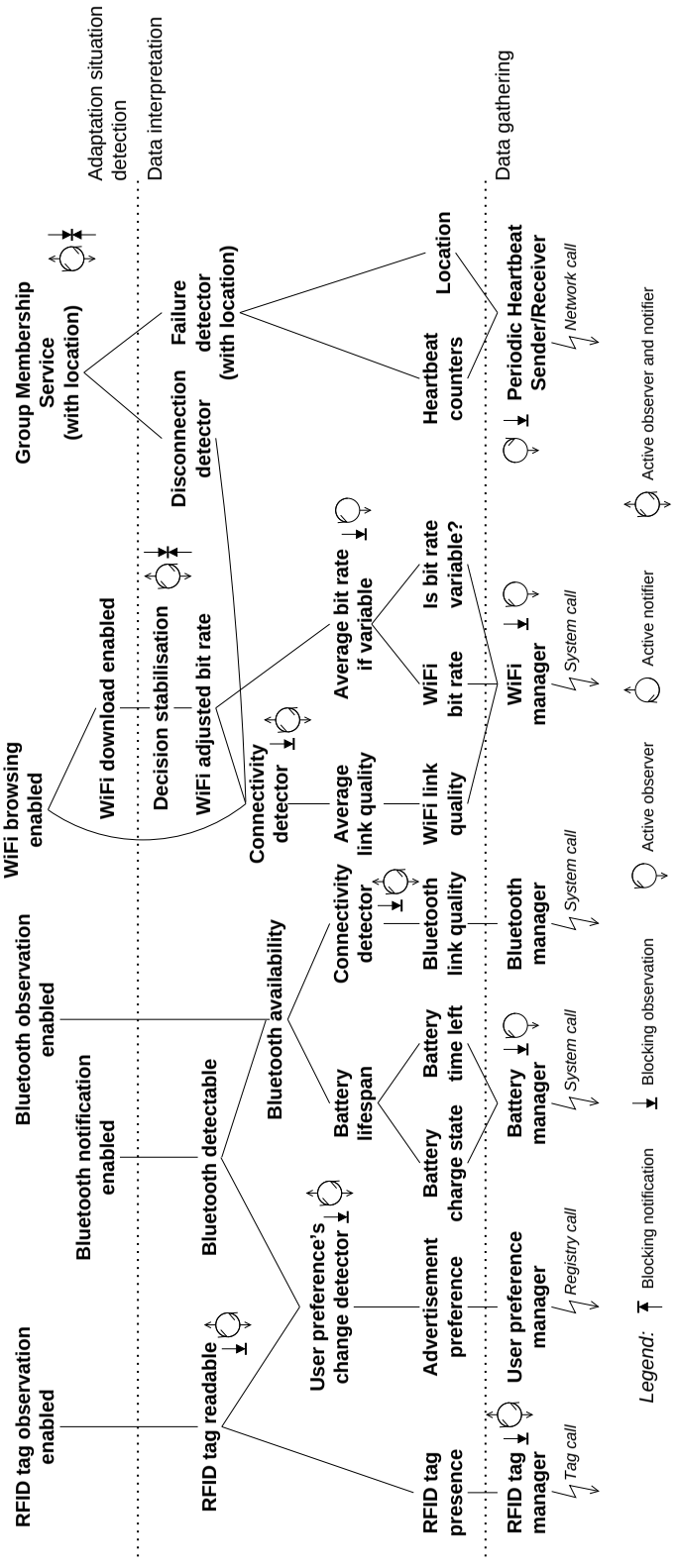

Figure 2: Modular context information management.

```
processor WiFiConnectivity=ConnectivityDetectorCO[BO,AO,AN](AverageWiFiQuality,AverageWiFiBitRate);

processor WiFiAdjustedBitRate=AdjustedBitRateCO(WiFiConnectivity,AverageWiFiBitRate);
processor WiFiStabilisation=DecisionStabilisationCO[BO,BN,AO,AN](WiFiAdjustedBitRate);
processor DisconnectionDetector=ConnectivityDetectorCO(WiFiConnectivity);
processor FailureDetector=FailureDetectorCO(HeartbeatMgr.extract("hb-counters-chunk","location-chunk"));

//Top: Adaptation situation detection
processor WiFiDownloadEnabled=IsEnabledCO(WiFiStabilisation);
processor WiFiBrowsingEnabled=IsEnabledCO(WiFiDownloadEnabled);
processor GroupMembershipService=GroupMembershipCO[BO,BN,AO,AN](DisconnectionDetector,FailureDetector);

//Concern: Thread management
task WiFiTasks=WiFiConnectivity,WiFiAdjustedBitRate,WiFiStabilisation;
thread Communication=WiFiStabilisation[30000],WiFiTasks[5000],WiFiMgr[1000];
thread Group=GroupMembershipService[10000],FailureDetector[3000];

//Concern: Memory management
reporting Communication=WiFiBrowsingEnabled/descendant-or-self::*;
reporting Group=GroupMembershipService,DisconnectionDetector,FailureDetector/descendant-or-self::*;
```

While leveraging the definition of context management policies, this language also provides various verifications dedicated to the definition of context management policies. For example, the language can prevent deadlocks in the policies —*e.g.*, observations (top-down flows of down-calls) and notifications (bottom-up flows of up-calls) that potentially traverse the same path of context nodes. Furthermore, the extra-functional part of this language can be extended to address other cross-cutting concerns, such as distribution (mapping of context nodes to physical machines).

In the remaining of this paper, the language is used as the basis for building the component-based architecture implementing the context policy. In particular, we illustrate how the constructions of this language can be mapped to design patterns in the architecture, which are latter reflected at runtime to support the dynamic reconfiguration of the context policies.

4 Pattern-oriented architecture of COSMOS

In this section, we present how COSMOS maps context policies to context node hierarchies. In particular, we describe the use of four design patterns, originally identified by the *Gang of Four* [8], for building an extensible architecture. By supporting these design patterns at design-time and at run-time, COSMOS exhibits an architecture closely related to its conceptual model, thus facilitating the dynamic reconfiguration of context policies. The remaining of this section introduces the mapping of COSMOS composition language construction to the design patterns *Factory method* (cf. Section 4.1), *Composite* (cf. Section 4.2), *Flyweight* (cf. Section 4.3), and *Singleton* (cf. Section 4.4).

4.1 “Factory method”: Building the context information reports

Each node of the hierarchy operates a specific treatment on the context information provided either by child nodes or by encapsulated primitive components in the case of leaves. At each level of the hierarchy, context information reports need to be dynamically created based on reports retrieved from child nodes. To handle the management of report instances, the context nodes apply a component-oriented version of the design pattern “Factory method” [8]. The factory method is a creational pattern that deals with the creation of objects without specifying the exact class of the objects that will be created.

In COSMOS, the skeleton of a context node is defined as the assembly of a context operator (extension of a ContextOperator) with, on the one hand, the components for the technical services (components ActivityManager and MessageManager), and with, on the other hand, the child nodes or the component that reifies a system resource (ResourceManager in the example of Figure 3). Thanks to this skeleton, the definition of a context node is leveraged, and it can be easily overridden to support a particular type of context information. The context operator ForwarderCO (derived from the abstract factory ContextOperator) is a generic implementation of the factory method and is able to store a message of whatever type. However, other implementations of this component allows the framework to implement various kind of operations, such as mathematical operations, boolean operations, or fuzzy rules.

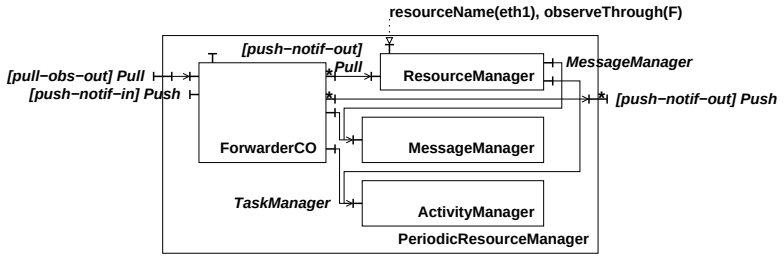

Figure 3: Illustration of the design pattern “Factory method”.

Figure 2 depicts a leaf of a context hierarchy —*i.e.*, a sensor— that uses the operator ForwarderCO as implementation of the factory method. For example, the sensor reifying the WiFi manager is described as follows:

```
sensor WiFiMgr = WirelessInterfaceRM[BO,AO,resourceName=>eth1];
```

The sensor `WiFiMgr` is translated into the following Fractal ADL [11] code excerpt, that reflects the design pattern “Factory method”. In particular, the context operator is automatically configured with the component ForwarderCO.

```
<definition name="WiFiMgr" extends="PeriodicResourceManager(nodeName=>WiFiMgr)">
    <component name="co" definition="ForwarderCO(observeThrough=>false,isActiveObserver=>true)"/>
    <component name="rm" definition="WirelessInterfaceRM(resourceName=>eth1)"/>
</definition>
```

When loaded by COSMOS, the sensor description is reified as the software architecture depicted in Figure 3.

4.2 “Composite”: Supporting the hierarchies of context nodes

The organisation of context information into hierarchies leads to the confinement of the different sub-trees in order to ease their composition. This confinement is realized with the design pattern “Composite” [8], which makes possible the homogenisation of the architecture in which an element is composed of several sub-elements themselves composites, except the leaves of the recursion. Furthermore, hierarchies built with COSMOS exploit the composition of nodes for inferring higher-level context information. Therefore, this type of hierarchical structure motivates the use of composites in order to isolate at each level of the hierarchy the sub-trees. Thanks to the composite, the complexity and the dependencies of nodes are automatically solved: This simplifies the composition at every level of the hierarchies.

```
processor WiFiAdjustedBitRate = AdjustedBitRate[BN](WiFiConnectivity, AverageWiFiBitRate);
```

An example is the definition of the processor `WiFiAdjustedBitRate`, which is transformed to the FRACTAL-ADL code excerpt that follows. It depicts a definition that builds the context node `AdjustedBitRate` composed of a node `ConnectivityDetector` and a node `AverageBitRate`. Note that the design pattern “Composite” does not preclude the sharing of components at several levels of a hierarchy (when a processor is required by several processors):

```
<definition name="WiFiAdjustedBitRate" extends="ContextNode(notifyThrough=>false)">
    <component name="cn-1" definition="WiFiConnectivity"/>
    <component name="cn-2" definition="AverageWiFiBitRate"/>
    [...]
</definition>
```

In this hierarchy, as illustrated in Figure 4, the most nested components in the hierarchies are the reified sensors, while the other components are the processors that infer context information from the former nodes.

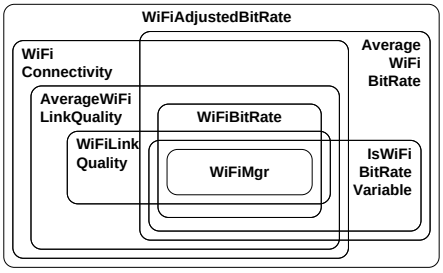

Figure 4: Illustration of the design pattern “Composite”.

4.3 “Flyweight”: Reducing the memory footprint of context nodes

The system resources reified in the leaf nodes of the hierarchy can be shared by numerous context nodes since the former nodes may contain many elementary context information. In the example of Figure 4, the context node WiFiMgr is shared by three context nodes. In fact, WiFiMgr context node reifies more than 30 context elements, thus being potentially shared by more than 30 context nodes. This is precisely the purpose of the design pattern “Flyweight” [8]: To efficiently share numerous fine-grained elements. By applying a component-oriented version of this design pattern, context nodes in COSMOS can efficiently share any child node of the hierarchy.

Another example of such a situation is the inference of the failure detector from information retrieved by the periodic heartbeat. In the description below, the processor `FailureDetector` extracts the context information `hb-counters-chunk` and `location-chunk` provided by the `HeartbeatMgr` context node.

```
sensor HeartbeatMgr=PeriodicHeartbeatRM[BO,AO];
[...]
processor FailureDetector=FailureDetectorCO(HeartbeatMgr.extract("hb-counters-chunk","location-chunk"));
```

This definition is translated into the ADL code excerpt that follows. The ADL definition shows the sharing of the component `HeartbeatMgr` by the context nodes `PeriodicHeartbeatCounters` and `PeriodicHeartbeatLocation`. These nodes are already defined in other ADL files not shown here. In this ADL code excerpt, the sharing is performed by recursively re-opening the component `PeriodicHeartbeatLocation` and by naming the “target” component of `PeriodicHeartbeatCounters` with a path (`./cn-1/cn/rm`). This definition states that the instance of the component `HeartbeatMgr` contained in the component `PeriodicHeartbeatLocation` (`./cn-2/cn/rm`) is the same as the one contained in the component `PeriodicHeartbeatCounters` (`./cn-1/cn/rm`):

```
<definition name="FailureDetector" extends="ContextNode">
    <component name="co" definition="FailureDetectorCO(resourceName=>FailureDetector)"/>
    <component name="cn-1" definition="PeriodicHeartbeatCounters"/>
    <component name="cn-2" definition="PeriodicHeartbeatLocation">
        <component name="cn" definition="HeartbeatMgr">
            <component name="rm" definition="./cn-1/cn/rm"/>
        </component>
    </component>
    [...]
</definition>
```

The resulting component-oriented version of the design pattern “Flyweight” is illustrated in Figure 5. By enforcing sharing of COSMOS hierarchies, the memory footprint of the COSMOS policy is considerably reduced.

4.4 “Singleton”: Controlling the resources consumed by context nodes

The design pattern “Singleton” is used to restrict instantiation of a class to one object [8]. This is useful when exactly one object is needed to coordinate actions across the system. But, sometimes it is generalised

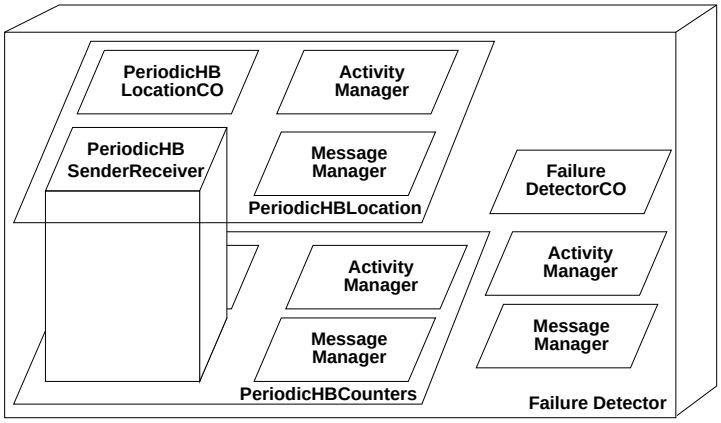

Figure 5: Illustration of the design pattern “Flyweight”.

to systems that operate more efficiently when only one or a few objects exist. In COSMOS, it consists in centralising the fine-grained control of system resources consumed by context operators. For instance, the end-user or the developer may want to execute all the observation and notification tasks in only one thread, or, on the contrary, to separate the observations from the notifications, or to partition context nodes of the graph into as many threads as needed (cf. Section 3). Therefore, one must be able to express the sharing of a component, here the component ActivityManager, by every nodes of the graph. For example, we expect that the context nodes associated to the processors `GroupMembershipService` and the processor `FailureDetector` share the component `ActivityManager`, which is responsible for scheduling activity tasks.

```
processor FailureDetector = FailureDetectorCO(PeriodicHeartbeat);
processor GroupMembershipService = GroupMembershipCO[BO,BN,AO,AN](DisconnectionDetector,FailureDetector);
[...]
thread GroupThread = GroupMembershipService[10000],FailureDetector[3000];
```

The following ADL code excerpt shows the resulting architecture for the component `GroupMembershipService`. In particular, the component `ActivityManager` is shared by specifying that the child node `FailureDetector` uses the instances of the component `ActivityManager` contained in the current definition (`GroupMembershipService`), thus using the path “`./am`”.

```
<definition name="GroupMembershipService" extends="ContextNode(isActiveObserver=>true,
                  isActiveNotifier=>true,observerThrough=>false,notifyThrough=>false)">
    <component name="co" definition="MembershipCO(periodObserve=>10000,periodNotify=>10000)"/>
    <component name="mm" definition="MessageManager"/>
    <component name="am" definition="ActivityManager"/>
    <component name="cn-1" definition="DisconnectionDetector">
    <component name="cn-2" definition="FailureDetector(periodObserve=>3000,periodNotify=>3000)">
        <component name="am" definition="./am"/>
    </component>
</definition>
```

Similarly, this design pattern is also used to share the component `MessageManager` when the context policy uses the construction **`reporting`** to group context report managers.

5 Evaluation

In this section, we report the benefits of using COSMOS as a framework for composing context management policies. This evaluation is based on preliminary experiences we operated with COSMOS. In particular, we observed that COSMOS provides a comprehensive approach for describing the realization of context policies. By providing a uniform abstraction of context information, the context node, COSMOS supports the composition of context information from low-level sensors to high-level policies. At the lowest level, context nodes can reify hardware capabilities (*e.g.*, CPU, battery, network), software resource (*e.g.*,

user preferences, e-mail history, personal agenda), or embedded sensors (*e.g.*, position, temperature, blood pressure). At an higher level, context nodes can reuse or develop composition operators (*e.g.*, mathematics, comparison, fuzzy) to infer advanced context information. As context policies are reflected themselves as context nodes, they can be reused in different contexts. For example, the scenario we present in this article reuses context policies we described in [3].

The reflection of design patterns in the architecture supports a canonical architecture for describing extensible context nodes. This base architecture acts as a backbone, which is refined and configured using the COSMOS composition language. In other words, design patterns we identified reflect the variation points of the architecture, which are fixed by the COSMOS descriptions. Context policies we developed and illustrated in this article demonstrate this extensible architecture. Furthermore, preliminary experiments show that the runtime performances are not affected by our approach compared to existing context management frameworks [3].

Up to now, COSMOS and its reference implementation, based on the FRACTAL component model, is used for the design and the development of context policies for ubiquitous computing scenarios[3]. Furthermore, COSMOS has been selected by the IST MUSIC project[4] in order to develop context operators for synthesising social relationships among collocated mobile users. Finally, the COSMOS abstract model is also investigated in combination to wireless sensor networks for reifying context information in next generation health-care applications[5].

6 Related Work

In this section, we compare COSMOS with middleware frameworks of the literature.

Context Toolkit is one of the first research works on context management that was based on event programming and widget concepts introduced by GUI (*Graphical User Interfaces*) [7]. Following the philosophy of the framework, 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.

Gaia Context Service consists of context providers offering low-level or high-level context information [13]. Context information is modelled using first order logic and boolean algebra. Gaia services and applications are programmed in a high-level scripting language (LuaOrb), which implements language bindings to object broker technologies such as CORBA and COM. The context providers are then either large-grained objects or developers must program the composition of context providers.

MoCA Context Service architecture transposes the ontology-based approach to an object-oriented one [6]. For instance, the source of a context information is described by an attribute rather than being described in the architecture; the type manager and the repository are the only accesses to context information, whatever the abstraction level and the use case. Context data are typed objects and described using an XML-based model. The authors propose to partition the context information space into views for improving the performance.

The Contextor builds the context manager as a network of contextors [5]. The Contextor defines an *ad hoc* component model, but the component model is implicit and the network of contextors is not configurable. The Flyweight design pattern may be applied to build a hierarchy of Java classes for contextors, but there is no design pattern at the architectural level. To limit the number of activities, the authors plan to use the Composite design pattern. The sharing of context nodes is not addressed.

[3] French project CAPPUCINO: `http://www.cappucino.fr`
[4] European project MUSIC: `http://www.ist-music.eu`
[5] Norwegian project SWISNET: `http://www.ifi.uio.no/forskning/grupper/nd/projects/swisnet`

CARISMA provides application developers with an application profile abstract syntax [2]. Profiles are passed down to the middleware and applications can change their profiles during execution. In a profile, the behaviour of the middleware with respect to an application specifies the context dependent adaptation situations. The specification gives no hints on the architecture of the context manager being responsible for detecting these situations.

PACE presents an architecture in which meta-data (temporality, quality, etc.) are added either to context data or to relations between them [9]. 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.

EgoSpaces is a tuple-space based middleware framework that puts the concepts of transient tuple space sharing, flexible tuple representation and declarative view specification forward [10]. Views are sets of tuples that satisfy some data constraints, are owned by agents that satisfy the agent constraints, are located on hosts that satisfy the host constraints, and for which these hosts must lie within the boundaries defined by the network constraints. Such a specification language for expressing the distribution of context information does not exist in COSMOS up to now.

7 Conclusion

This article proposes COSMOS: A component- and software architecture-based framework for gathering and processing context information. As a matter of future work, we plan to adopt two directions. First, we believe it could be interesting to complement the domain specific language and the architectural patterns with constructs dedicated to the distribution of context data on different types of networks. A second direction concerns the composition of context policies at run-time, the issue being to be able to address situations in which the intersection between these policies may be non empty, that is dynamically detecting and solving conflicts.

References

[1] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley, 1995.

[2] CAPRA, L., EMMERICH, W., AND MASCOLO, C. CARISMA: Contex-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering 29*, 10 (Oct. 2003), 929–945.

[3] CONAN, D., ROUVOY, R., AND SEINTURIER, L. Scalable Processing of Context Information with COSMOS. In *7th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems* (Paphos, Cyprus, June 5–8 2007), vol. 4531 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 210–224.

[4] COULSON, G., BLAIR, G., GRACE, P., TAÏANI, F., JOOLIA, A., LEE, K., UEYAMA, J., AND SIVAHARAN, T. A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems 26*, 1 (Feb. 2008).

[5] COUTAZ, J., AND REY, G. Foundations for a Theory of Contextors. In *4th International Conference on Computer-Aided Design of User Interfaces* (Valenciennes, France, May 2002), Kluwer, pp. 13–34.

[6] DA ROCHA, R., AND ENDLER, M. Context Management in Heterogeneous, Evolving Ubiquitous Envrionments. *IEEE Distributed Systems Online 7*, 4 (Apr. 2006), 1–13.

[7] DEY, A., SALBER, D., AND ABOWD, G. 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 (2001), 97–166.

[8] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] HENRICKSEN, K., INDULSKA, J., MCFADDEN, T., AND BALASUBRAMANIAM, S. Middleware for Distributed Context-Aware Systems. In *7th International Symposium on Distributed Objects and Applications* (Agia Napa, Cyprus, Nov. 2005), vol. 3760 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 846–863.

[10] JULIEN, C., AND GRUIO-CATALIN, R. EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications. *IEEE Transactions on Software Engineering 32*, 5 (May 2006), 281–298.

[11] LECLERCQ, M., ÖZCAN, A. E., QUÉMA, V., AND STEFANI, J.-B. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *29th International Conference on Software Engineering* (Minneapolis, MN, USA, May 2007), IEEE Computer Society, pp. 209–219.

[12] LECLERCQ, M., QUÉMA, V., AND STEFANI, J.-B. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online 6*, 9 (Sept. 2005).

[13] ROMÁN, M., HESS, C., CERQUEIRA, R., RANGANATHAN, A., CAMPBELL, R., AND NAHRSTEDT, K. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing 1*, 4 (Oct. 2002), 74–83.

[14] SCHMIDT, D., AND BUSCHMANN, F. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. In *25th International Conference on Software Engineering* (May 2003), ACM Press, pp. 694–704.

[15] SZYPERSKI, C. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[16] ZACHARIADIS, S., MASCOLO, C., AND EMMERICH, W. The SATIN Component System—A Metamodel for Engineering Adaptable Mobile Systems. *IEEE Transactions on Software Engineering 32*, 11 (Oct. 2006), 910–927.