Managing Ubiquitous Applications
Software Architecture Patterns for a Context-Processing Middleware Framework

Romain Rouvoy • University of Oslo
Denis Conan • Institut TELECOM SudParis
Lionel Seinturier • University of Lille 1

COSMOS is a component-based framework for managing context information in ubiquitous context-aware applications. COSMOS decomposes context observation policies into fine-grained units called context nodes.

Ubiquitous applications are characterized by variations in their execution context. Their correct operation requires some continual adaptations based on the observation of this context. The design and implementation of these observation policies is thus the cornerstone of any ubiquitous application. COSMOS (Context Entities Composition and Sharing) is a component-based framework for managing context information in ubiquitous context-aware applications. It supports the design and development of such applications, which react to changes in their execution environment. Examples of such context changes are the appearance or disappearance of hardware or software resources, and modifications in user preferences. Because the context information that such applications require is very diverse, COSMOS relies on component-based software engineering principles to ensure the integration of context information. In particular, the framework combines the concepts of software components and architectural design patterns to define its architecture’s foundations. Software components efficiently encapsulate the context information diversity, and architectural design patterns outline the context management policies.

In an earlier work, we introduced context management policies in COSMOS to identify contextual situations for which we expected the application to react. COSMOS describes context management policies as hierarchies of context nodes using a dedicated composition language. Here, we present the mapping of the composition language constructions to architectural design patterns used in COSMOS. Although the use of a domain-specific language leverages the definition of context management policies, components and design patterns support the dynamic reconfiguration and evolution of the context management policies, once specified and deployed.

Components are increasingly gaining more attention for designing and implementing middleware platforms. However, researchers haven’t investigated the identification of architectural design patterns in a similar way to object-oriented middleware. Thus, we illustrate how to reuse well-known design patterns and apply them at the architectural level to offer better control over the COSMOS architecture. In particular, we use these design patterns to separate the various extra-functional concerns (memory footprint, resource consumption, instance management, and so on) involved in a context management policy from the business concerns of context management. A mobile computing scenario illustrates the benefits of COSMOS.

Foundations of COSMOS
Here, we explain the basis for our proposed approach by discussing the properties of the basic building units of a context policy’s composition and describing the core microarchitecture of these building units.
Context node concepts and properties

The basic structuring concept of COSMOS is the context node. A context node is context information modeled by a software component. COSMOS organizes context nodes into hierarchies to form context management policies. The relationships between context nodes are sharing and encapsulation. The sharing of a context node—and, by implication, of a partial or complete hierarchy—corresponds to the sharing of part or all of a context policy.

Context nodes at a hierarchy’s leaves (the bottom-most elements, with no descendants) encapsulate raw context data obtained from collectors (such as operating system probes, sensors near the device, user preferences in profiles, and remote devices). Context nodes should provide all the inputs necessary for reasoning about the execution context, which is why COSMOS considers user preferences as context data. A context node’s role is thus to isolate the inference of high-level context information from lower architectural layers responsible for collecting context data.

Context nodes also possess properties that define their behavior with respect to the context management policy.

Active or passive. Each context node can be active or passive. An active node is associated with a thread of control. Typical examples of active nodes include a node in charge of the centralization of several types of context information, a node responsible for the periodic computation of higher-level context information, and a node to provide the latter information to upper nodes.

Observation and notification. Communication into a context node’s hierarchy can be top-down or bottom-up. The former corresponds to observations that a parent node triggers, whereas the latter corresponds to notifications that context nodes send to their parents.

Pass-through or blocking. A context node that receives data transmitted by a notification or an observation can be pass-through or blocking. Pass-through nodes propagate observations and notifications. Blocking nodes stop the traversal. For observations, COSMOS transmits the most up-to-date context information without polling child nodes. For notifications, COSMOS uses context data to update the node’s state, but it doesn’t notify parent nodes.

COSMOS provides the developer with predefined generic context operators, organized following a typology: elementary operators for collecting raw data; memory operators, such as averaging and translation operators; data mergers; and abstract or inference operators, such as adders or thresholds operators. The only programming is in the context operators. If a sufficiently large library of context operators is available and well targeted to a developer’s business, no programming should be necessary—only declarative composition of context nodes.

Context node architecture

Each context node extends the abstract composite ContextNode, as Figure 1 shows. Pull and Push are the interfaces for observation and notification, respectively. ContextNode contains at least an operator (primitive abstract component ContextOperator) as well as the message and activity managers. The message manager handles the observation and notification reports that the component sends and receives through the Pull and Push interfaces. The activity manager provides support for dealing with active components.

Finally, nodes come with attributes that characterize their behavior with respect to the properties defined earlier. COSMOS lets users choose nine different attributes: four for observer nodes, four for notifier nodes, and the nodeName attribute, which holds the context node’s name. Observer nodes can be

• active (isActiveObserver = true) or passive (isActiveObserver = false),
• with (periodObserver) or without (periodObserver = 0) a period,
• limited to only one observation (observeOnlyOnce = true) or allowing more than one observation (observeOnlyOnce = false), and
• pass-through (observerThrough = true) or blocking (observerThrough = false).

Likewise, notifier nodes can be

• active (isActiveNotifier = true) or passive (isActiveNotifier = false),
• with (periodNotifier) or without (periodNotifier = 0) a period,
• limited to only one notification (notifyOnlyOnce = true) or allowing more than one notification (notifyOnlyOnce = false), and
• pass-through (notifyThrough = true) or blocking (notifyThrough = false).

Figure 1 lists the default values for these nine attributes.

COSMOS classifies context nodes into two categories: leaves and other nodes. Leaves in a hierarchy are ContextNodes extended to contain one or several components that receive context information from an external entity. This external entity can be the operating system or another framework, built with COSMOS or not, and component oriented or not. For example, a Wi-Fi resource manager could obtain the corresponding context information directly from the operating system (through system calls) or could encapsulate a (legacy) framework dedicated to the reification of system resources. Graph nodes that aren’t leaves are extended to contain one or several other context nodes. For example, a context node could compute a terminal’s battery charge state by encapsulating two other context nodes—the first one computing the battery charge state, and the second one computing the battery time left.

Mobile computing scenario

To illustrate the COSMOS framework’s objectives, we consider a family shopping in a mall with a mobile device. (This scenario is a use case of the French “Cappuccino” project, www.cappucino.fr.) This application lets the family share information, consult product prices, download discount tickets, receive advertisements, access information and comments about products, and find a product or a shop’s location in the mall. The parents want their children to remain in the mall, with their devices on as much as possible, so that each family member knows the others’ locations. But they can turn off their devices once in a while to save batteries. While walking in the mall, the eldest girl sees an advertisement indicating that a dress store proposes an RFID-tag-based service for helping to choose clothes.
All these features are based on different network technologies, such as Bluetooth or Wi-Fi, and require the application to adapt itself depending on network connectivity and context information availability. As Figure 2 shows, each adaptation situation (upper part of the figure) is isolated in a context tree, with the possibility of sharing subtrees between policies. Each adaptation situation relates to a particular functionality and focuses on a precise set of context information.

For example, WiFi download enabled relates to the functionality that supports downloading a discount ticket; it lets the application know when that functionality is available. COSMOS detects availability by monitoring the Wi-Fi link’s quality. WiFi browsing enabled builds on the previous adaptation situation and lets the application enable and configure a browsing facility to access comments about a product or to find its location in the mall. Bluetooth observation enabled lets users consult product prices and product references nearby. COSMOS enables this functionality when the device battery life expectancy is high enough. Bluetooth availability combines with user preferences to infer the Bluetooth notification enabled situation. This last situation configures the user application so that the mall infrastructure will notify it of product advertisements. Finally, Group membership service uses the disconnection and failure detectors to distinguish between disconnections and failures, and to inform the user about other members’ locations. Group membership service illustrates the definition of sharing between hierarchies by reusing the Disconnection detector context node provided by the hierarchy WiFi browsing enabled.

To describe the context management policy for implementing this mobile computing scenario, COSMOS provides a declarative language dedicated to the composition of context nodes. The core of this language is as follows, using the extended Backus-Naur form (EBNF):

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

Figure 2. Modular context information management. This is the hierarchy of context nodes to implement the mobile computing scenario.

IEEE Distributed Systems Online (vol. 9, no. 6), art. no. 0806-mds2008060001
The composition language isolates the descriptions of functional concerns from extra-functional ones. Two constructions, sensor and processor, reify functional concerns. Users can configure sensors with the \texttt{AO}, \texttt{AN}, \texttt{BO}, \texttt{BN}, or \texttt{OO} properties to activate the \textit{active observation}, \textit{active notification}, \textit{blocking observation}, \textit{blocking notification}, and \textit{only once} mechanisms, as well as additional attributes specific to the resource manager (such as \texttt{resourceName=>eth1}). Similarly, processors use the same configuration mechanisms but must describe context dependencies.

The second part of the language supports the specification of extra-functional concerns, such as resource consumption. The task and thread constructions specify the organization of activities into executable-task hierarchies and the mapping of tasks to threads. Finally, the reporting construction supports the grouping of context report managers to reduce the policies’ memory footprints. COSMOS defines groups using XPath requests (www.w3.org/TR/xpath20), which apply to a context hierarchy’s nodes.

Using this composition language, we describe context policies \texttt{WiFi Download Enabled}, \texttt{WiFi Browsing Enabled}, and \texttt{Group Membership Service} (right side of Figure 2) as follows:

```verbatim
//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 = AverageCO(WiFiMgr.extract("bit-rate-chunk","is-variable-chunk"));
processor WiFiConnectivity = ConnectivityDetectorCO[BO,AO,AN](AverageWiFiQuality,AverageWiFiBitRate);

processor WiFiAdjustedBitRate =
  AdjustedBitRateCO(WiFiConnectivity,AverageWiFiBitRate);
processor WiFiStabilization =
  DecisionStabilizationCO[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(WiFiStabilization);
processor WiFiBrowsingEnabled = IsEnabledCO(WiFiDownloadEnabled);
processor GroupMembershipService =
  GroupMembershipCO[BO,BN,AO,AN](DisconnectionDetector,FailureDetector);
```
// Concern: Thread management

```java
//Thread
thread WiFiTasks = WiFiConnectivity,WiFiAdjustedBitRate,WiFiStabilization;
thread Communication = WiFiStabilization[30000],WiFiTasks[5000],WiFiMgr[1000];
thread Group = GroupMembershipService[10000],FailureDetector[3000];
```

// Concern: Memory management

```java
//Group
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 for the definition of context management policies. For example, the language can prevent deadlocks in the policies, such as 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, it's possible to extend the extra-functional part of this language to address other cross-cutting concerns, such as distribution (mapping of context nodes to physical machines).

**COSMOS pattern-oriented architecture**

Here, we present how COSMOS maps context policies to context node hierarchies. In particular, Erich Gamma and his colleagues (the "Gang of Four") identified four design patterns for building an extensible architecture. By supporting these design patterns at design time and at runtime, COSMOS exhibits an architecture closely related to its conceptual model, thus facilitating the dynamic reconfiguration of context policies.

**Factory method: Building context information reports**

Each hierarchy node operates a specific treatment on the context information provided either by child nodes or, in the case of leaves, by encapsulated primitive components. At each hierarchical level, COSMOS must dynamically create context information reports on the basis of reports retrieved from child nodes. To handle the management of report instances, the context nodes apply a component-oriented version of the factory method design pattern. This pattern is for creating objects without specifying the exact class of objects to create.

COSMOS defines a context node's skeleton as the assembly of a context operator (extension of a ContextOperator) with, on the one hand, the components for the extra-functional services (the ActivityManager and MessageManager components) and, on the other hand, the child nodes or the component that reifies a system resource (ResourceManager in the example of Figure 3). Using this skeleton, COSMOS leverages a context node’s definition and can easily override this definition 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 can store a message of any type. However, other implementations of this component let the framework implement various kinds of operations, such as mathematical operations, Boolean operations, and fuzzy rules.
Figure 3. Architecture of a context node that implements the factory method design pattern.8

Figure 3 depicts a context hierarchy’s leaf—a sensor—that uses the ForwarderCO operator as an implementation of the factory method design pattern. For example, COSMOS describes the sensor reifying the Wi-Fi manager as follows:

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

COSMOS translates the WiFiMgr sensor into the following Fractal ADL (architecture description language) code excerpt,9 which reflects the factory method design pattern:

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

In particular, the context operator is automatically configured with the ForwarderCO component. When loaded by COSMOS, the sensor description is reified as the software architecture shown in Figure 3.

**Composite: Supporting context node hierarchies**

The organization of context information into hierarchies leads to the confinement of different subtrees to ease their composition. The composite design pattern realizes this confinement.8 This pattern homogenizes the architecture such that an element consists of several subelements—themselves composites, except for the recursion’s leaves. Furthermore, hierarchies built with COSMOS exploit the composition of nodes to infer higher-level context information. Therefore, this type of hierarchical structure motivates the use of composites to isolate the subtrees at each hierarchical level. The composite design pattern thus automatically provides a solution for dealing with the nodes’ complexity and dependencies, simplifying the composition at every hierarchical level.

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

An example of the composite design pattern is the processor definition WiFiAdjustedBitRate, which COSMOS transforms into the following Fractal ADL code excerpt:

```xml
<definition name="WiFiAdjustedBitRate" extends="ContextNode(notifyThrough=>false)">
  <component name="cn-1" definition="WiFiConnectivity"/>
  <component name="cn-2" definition="AverageWiFiBitRate"/>
</definition>
```
This excerpt depicts a definition that builds the WiFiAdjustedBitRate context node, which comprises the WiFiConnectivity and AverageWiFiBitRate nodes. The composite design pattern doesn’t preclude the sharing of components at several hierarchical levels when several processors require the same processor.

As Figure 4 shows, the most nested components in the hierarchies are the reified sensors, whereas the other components are the processors that infer context information from the former nodes.

![Figure 4. Architecture of a context node that implements the composite design pattern.](image)

**Flyweight: Reducing context nodes’ memory footprints**

Several context nodes can share the system resources reified in a hierarchy’s leaf nodes because these leaf nodes can contain a large amount of elementary context data. In Figure 4, three context nodes share the WiFiMgr context node. In fact, WiFiMgr reifies more than 30 context elements, potentially shared by more than 30 context nodes. This is precisely the purpose of the *flyweight* design pattern: to efficiently share several fine-grained elements. Applying a component-oriented version of this design pattern lets context nodes in COSMOS efficiently share any child node of the hierarchy.

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

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

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

```xml
<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>
```

Figure 5 shows the resulting component-oriented version of the flyweight design pattern. Enforcing sharing of COSMOS hierarchies considerably reduces the COSMOS policies' memory footprints.

Figure 5. Architecture of a context node that implements the flyweight design pattern.8

**Singleton: Controlling the resources that context nodes consume**

The singleton design pattern restricts instantiations of a class to one object.8 This is useful when exactly one object is needed to coordinate actions across the system. But this design pattern is generalized to systems that operate more efficiently when only one or a few objects exist. In COSMOS, this design pattern centralizes the fine-grained control of system resources consumed by context operators. For example, users or developers might want to execute all observation and notification tasks in only one thread, separate observations from notifications, or partition context nodes of the graph into as many threads as needed. Therefore, they must be able to express the sharing of a component (in this case, ActivityManager) by every node of the graph. For example, the context nodes associated with the GroupMembershipService and FailureDetector processors can share the ActivityManager component, which is responsible for scheduling activity tasks.

```java
processor FailureDetector = FailureDetectorCO(PeriodicHeartbeat);
processor GroupMembershipService = GroupMembershipCO[BO,BN,AC,AN] (DisconnectionDetector, FailureDetector);
```
The following ADL code excerpt shows the resulting architecture for the `GroupMembershipService` component. In particular, COSMOS shares `ActivityManager` by specifying that the `FailureDetector` child node must employ the instances of `ActivityManager` contained in the current definition (`GroupMembershipService`), thus using the path `./am`.

```adl
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="am" definition="./am"/>
  </component>
  <component name="cn-2" definition="FailureDetector(periodObserve=>3000, periodNotify=>3000)">
    <component name="am" definition="./am"/>
  </component>
</definition>
```

COSMOS also uses this design pattern to share the `MessageManager` component when the context policy uses the reporting construction to group context report managers.

**Evaluation**

Here, we evaluate the benefits of using COSMOS as a framework for composing context management policies. This evaluation is based on preliminary experiments we conducted with COSMOS. 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 (for example, a CPU, a battery, or a network), software resources (user preferences, an email history, a personal agenda, and so on), or embedded sensors (such as position, temperature, and blood pressure). At a higher level, context nodes can reuse or develop composition operators (such as mathematics, comparisons, and fuzzy logic) to infer advanced context information. The core motivation of COSMOS is to isolate context management policies from applications to enforce their reuse. Because context policies are themselves reflected as context nodes, they can be reused in different contexts. For example, the scenario we present in this article reuses context policies that we described in an earlier work.²

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 the COSMOS composition language refines and configures. In other words, the four design patterns we just identified reflect the architecture’s variation points, which are later configured in the COSMOS descriptions. The context policies we developed demonstrate the architecture’s extensible nature. Furthermore, preliminary experiments show that our approach doesn’t affect runtime performance as much as existing context management frameworks.²

Several existing projects use COSMOS. The French “Cappucino” project (www.cappucino.fr) uses COSMOS and its reference implementation (based on the Fractal component model) for the design and development of context policies for ubiquitous computing scenarios. The European IST (Information Society Technologies) MUSIC project (www.ist-music.eu) uses COSMOS to develop context operators for synthesizing social relationships among collocated mobile users. The Norwegian SWISNET project (www.ifi.uio.no/forskning/grupper/nd/projects/swisnet) is investigating the COSMOS abstract model in...
combination with wireless sensor networks for reifying context information in next-generation health-care applications.

**Related Work**

Here, we compare COSMOS with other middleware frameworks.

**Context Toolkit**

One of the first research works on context management, Context Toolkit is based on event programming and widget concepts introduced by GUIs. Following the philosophy of the framework, users must program interpretation and aggregation functionalities in monolithic blocks: one interpreter and one aggregator per application, independent of the number of widgets and the level of abstraction requested by the application.

**Gaia Context Service**

This framework consists of context providers offering low-level or high-level context information. Gaia models context information 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 (Component Object Model). Then, either the context providers are large-grained objects or developers must program the composition of context providers.

**MoCA Context Service**

This architecture transposes the ontology-based approach to an object-oriented one. For example, the source of context information is described by an attribute rather than in the architecture. The type manager and the repository are the only accesses to context information, regardless of the abstraction level and the use case. MoCA describes context data as typed objects using an XML-based model. This framework’s developers partition the context information space into views to improve performance.

**Draco**

This context manager uses an ontology to model context information and uses components for the architecture. Draco uses a component-oriented approach to dynamically adapt the context manager. Concerning the ontology orientation, defining an optimal deployment is difficult because it’s hard to have an estimate of the processing time for all context management activities and to use an ontology broker on small-scale devices.

**The Contextor**

This framework builds the context manager as a network of contextors. The Contextor defines an ad hoc component model, but the component model is implicit, and the network of contextors isn’t configurable. It’s possible to apply the flyweight design pattern to build a hierarchy of Java classes for contextors, but there’s no design pattern at the architectural level. To limit the number of activities, this framework’s developers plan to use the composite design pattern. The sharing of context nodes is not addressed.

**CARISMA**

This middleware framework provides application developers with an application profile abstract syntax. CARISMA passes profiles down to the middleware, and applications can change their profiles during execution. In a profile, the middleware’s behavior with respect to an application specifies the context-dependent adaptation situations. However, the specification doesn’t describe the context manager’s architecture responsible for detecting those adaptation situations.
PACE
This architecture adds metadata (temporality, quality, and so on) to either context data or relations between such data. This framework’s developers claim that the object and the ontology orientations are the two acceptable alternatives among the myriad modeling methods. With COSMOS, we added component orientation, which removes a limitation of the object orientation. Component orientation provides a more formal specification of the dependencies between context entities, through the use of an ADL.

EgoSpaces
This tuple-space based middleware framework puts forward the concepts of transient tuple space sharing, flexible tuple representation, and declarative view specification. Views are sets of tuples that satisfy some data constraints and are owned by agents that satisfy the agent constraints. Views reside on hosts, which satisfy the host constraints, and these hosts must lie within the boundaries defined by the network constraints. Such a specification language for expressing the distribution of context information doesn’t yet exist in COSMOS.

In 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 runtime. The challenge is to address situations in which the intersection between these policies could be nonempty—that is, COSMOS must be able to dynamically detect and solve conflicts.

References

8. E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

Romain Rouvoy is a postdoctoral researcher in the Department of Informatics at the University of Oslo, Norway. His research interests include the development of self-adaptive applications for mobile users evolving in ubiquitous environments, especially the dynamic adaptation of crosscutting concerns in component-based systems. He received his PhD in computer science from the University of Lille 1. Contact him at rouvoy@ifi.uio.no.

Denis Conan is an associate professor at TELECOM & Management SudParis of Institut TELECOM. His research interests include distributed systems, software components, software architecture, fault tolerance, and ubiquitous environments. He received his PhD in computer science from the University of Paris 6. Contact him at denis.conan@it-sudparis.eu.

Lionel Seinturier is a professor of computer science at the University of Lille 1. His research interests focus on middleware, components, and aspect-oriented software development. He received his PhD in computer science from CNAM (Conservatoire National des Arts et Métiers). Contact him at lionel.seinturier@univ-lille1.fr.

Cite this article: