Autonomic Service-Oriented Context for Pervasive Applications

Colin Aygalinc, Gerbert-Gaillard Eva, German Vega, Philippe Lalanda, Stéphanie Chollet

Autonomic service-oriented context for pervasive applications

Colin Aygalinc, Eva Gerbert-Gaillard, German Vega, Philippe Lalanda

Grenoble University, Laboratoire LIG
38058 Grenoble, France
firstname.lastname@imag.fr

Stephanie Chollet

Grenoble University, Laboratoire LCIS
26900 Valence, France
stephanie.chollet@lcis.grenoble-inp.fr

***Abstract*—Pervasive computing promotes environments where smart, communication-enabled devices cooperate to provide services to people. Due to their inherent complexity, many pervasive applications are built on top of service-oriented platforms, providing a set of facilities simplifying their development and execution. In this paper, we present such a platform, iCasa, extended with an autonomic, service-oriented context module. This module is programmed with a domain-specific service-oriented language built on top of iPOJO, the Apache service-oriented component model. It is validated on smart home applications developed with the Orange Labs.**

Keywords—pervasive computing; context; service-oriented components.

I. INTRODUCTION

Pervasive computing envisions environments where smart devices are blended into everyday objects in order to provide added-value services to people [1] [2] [3] [4]. These devices are communication-enabled and can cooperate with each other in order to deliver advanced applications. Such applications have to meet very stringent requirements: they are expected to be available anytime and anywhere while remaining invisible and non-obstructive.

We believe that advanced software engineering principles and tools are needed to support the production and execution of pervasive applications. A common approach is to introduce an execution platform providing a development model and a set of technical services that can be used by the applications (also called functional services). Precisely, the platform is responsible for dealing with many non-functional properties, like security or dynamic device integration persistence, on behalf of the applications.

Several such platforms have thus been proposed in the pervasive community. For instance, the Gaia platform [5] manages resources present in a pervasive space, called an active space. Gaia provides a set of services allowing application programmers to access the available devices or resources more easily. After this initial work, many platforms have been developed on top of the OSGi framework, which promotes the development of modular and dynamic applications. Indeed, pervasive platforms are usually based on the principles of service-oriented computing [6] where loosely coupled services enable the creation of flexible dynamic applications. An application is built from several services that can be distributed on different devices.

Many platforms include the notion of context, which purpose is to provide timely information about the computing and user environments. This allows applications to react to changes, like user behavior or availability of computational resources for instance. In the early days of pervasive computing, context was essentially limited to location-awareness. Since then, context has evolved towards more elaborated models, and has become a representation of any information that can be used to characterize an entity that is relevant to the interaction between a user and an application [7].

As we will see in more details in section 2, context and applications are generally kept separated. This architectural approach makes applications development and execution easier. It allows application developers to focus on implementation of business services, while delegating the complexity of context management to the pervasive platform. Context management is indeed a complex activity involving several complicated tasks, ranging from raw data acquisition (dealing with heterogeneous, dynamic and sometimes unreliable sensors) to knowledge representation and reasoning.

Many techniques and approaches have been proposed to deal with context management in pervasive settings [8]. Most of them are based on data-centric approaches where contextual information is kept in a knowledge base (database, ontology) and explicitly retrieved by applications. Such approach is well suited to knowledge-intensive applications with no hard deadline to be met. We are targeting applications executed on Internet gateways, at the edge of the network, with real-time constraints.

In this paper, we propose a novel approach based on autonomic computing and service-oriented computing, targeting a finer integration between context and applications. Precisely, we have developed a domain-specific service-oriented language allowing the straightforward definition of autonomic context modules. This language builds upon the iPOJO component model and is integrated in the iCasa platform [9]. It is validated on several applications developed with the Orange Labs.

The structure of this paper is the following. The coming section provides background about our context and requirements. Section 3 presents the overall approach defended in this paper and section 4 provides details about it. Section 5 is about its implementation. Finally, related work is developed in section 6.

II. BACKGROUND AND REQUIREMENTS

A. *Context management in pervasive computing*

Context is traditionally presented as a synchronized description of concepts and relationships pertaining to the execution environment. Precisely, contextual information [10] comes from the computing environment (memory, network, etc.), the user environment (location, needs, preference, etc.) and the physical environment (temperature, noise, etc.). It can be the description of a fact, a physical object, a physical value, or even an event (discrete or continuous [11]).

There are several architectural approaches to build context-aware applications [12]. A popular solution is to use a context management infrastructure. Error-prone tasks like information gathering (context acquisition), information modeling and processing (through inferences or mediation operations), information storing and presentation lie outside the application boundaries. This pattern where context management and applications are clearly separated improves code readability, debugging and evolution. It also allows context sharing between pervasive applications.

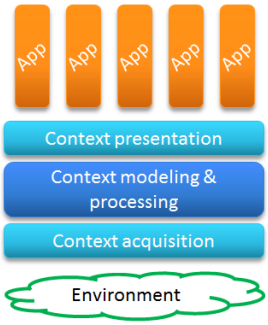

Figure 1. Separating context and applications.

We subscribe to this latter approach where context and applications are developed and extended independently (see Figure 1). However, we readily acknowledge that building a context management infrastructure remains a daunting task, and that multiple design trade-off decisions have to be made regarding data access, synchronization mechanisms, knowledge representation, reasoning, or presentation.

There are indeed many existing available context modeling and processing approaches [13] [14]. They mainly differ in the model used to represent contextual information and in the supported inferring techniques. However, most of them share a data-centric approach, where all context information is collected from sensors, kept in a storage facility, and made available to applications through queries.

Data-centric approaches are better suited to applications requiring complex knowledge representation or reasoning facilities. This is for instance the case of knowledge-intensive applications run in cloud infrastructure. As explained here after, we target applications deployed at the edge of the network, sometimes referred as fog computing [15], in order to perform immediate, added-value services, rather than long term, offline data processing.

B. *Context management in smart homes*

Our research deals with pervasive applications in intelligent environments like smart homes, smart buildings or smart manufacturing (industry 4.0). These applications are now widely distributed, from the sensors up to cloud facilities. Some code is executed at the edge of the network, in an Internet gateway for instance, while other code is run in computing farms. Depending on the code location, various forms of context are needed with different formalisms, different real-time constraints, and different interaction patterns.

To illustrate these various needs, let us consider health care applications in smart homes. More precisely, let us focus on the actimetrics application that we have been investigating for years with the Orange Labs [16]. Two major functions can be distinguished for actimetrics: the first one is about early diagnosis of degenerative diseases like Alzheimer while the second one is concerned with real-time supervision of people at home.

The first function, identification of degenerative diseases, is concerned with long term evolutions spanning several months. It requires complex time-series and event correlation analysis and is based on a rich, slowly evolving context that is explicitly accessed and browsed by the analysis algorithms. This is typically a data-centric context. The second function is about real-time supervision. It deals, for instance, with fall detection or automated alerting in case of unusual events like prolonged inactivity or irregular sleep hours. This second function may use the same environmental data as the first one, but it is much less demanding in terms of knowledge representation and reasoning. On the other hand, it has to deal with stringent real-time constraints: new information should be made available very rapidly to the application. Also, interactions are bi-directional and must meet requirements of low latency and simplicity. Moreover, due to their location, they have to support all the burden of dynamism induced by the physical environment while limiting end-user administration tasks, a property known as zero administration. Here, most existing context-management frameworks would impose an unwanted overhead.

From these features, we identified the following requirements for a context management middleware adapted to applications (or code) located at the network edge:

- **maintainability**: zero administration requires a modularized and maintainable middleware. Most approaches neglect the bi-directional interaction (acquisition/action) aspect although it involves highly error-prone technical code from low-level synchronization to high-level mediation or enrichment operations.

- **dynamicity resilience**: the middleware should support dynamic discovery and opportunistic use of new context sources and elements.
- **autonomic management**: the middleware must support dynamic configuration at runtime because application requirements, and so on context, evolve over time. For example, a service not used by application must be discarded and stopped, or synchronization frequency must be adjusted to avoid resource waste. However, managing this aspect manually is cumbersome and inefficient. We thought that this aspect should be managed in an autonomic way.

We believe that a service-centric context management middleware can meet these requirements. A Service-Oriented Architecture eases application development by the composition of a modularized set of services. It infuses naturally in the application interesting characteristics like loose-coupling, resiliency to dynamism with late-binding, substitutability and location transparency [17].

By presenting Context as a Service to the application, we benefit from all these properties and limit the context to the application needs. In addition, leaving implementation details of context services to the middleware layer frees the application developer from technical, high-error prone development issues related to context. But software engineering and tooling are still needed to ensure a maintainable and manageable context middleware.

III. APPROACH

A. *Overview*

Our approach is based on a clear separation between application and context and the use of service-oriented computing. As illustrated by Figure 2, the context management system publishes context services that can be used by the applications. Formally, context services form the contract between the context management layer and the applications. They are implemented as a graph of context entities, which perform data acquisition, context modeling and processing and, in the end, service publication.

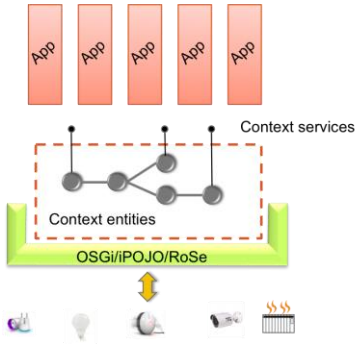

Figure 2. Context as a service.

Context entities are implemented with iPOJO, the Apache service-oriented component model [17]. The integration with the environments is based on RoSe [18], an open source communication middleware that is able to dynamically import and export services [19].

However, we have experienced that this approach remains hard to put in place. The implementation, provisioning and management of such context are very complex. It is highly error-prone, especially because of the need to deal with the dynamicity and heterogeneity of context sources and applications, and to ensure synchronization.

To circumvent these limitations, we have defined and implemented a service-oriented domain-specific language (DSL) for context definition on top of iPOJO. Our purpose is to provide assistance to developers for application and context development. The resulting context is autonomic in the sense that it can deal on its own with highly dynamic pervasive environments.

Precisely, our context management system can dynamically adapt the way it collects information and the way it provides contextual services to the client applications. This approach builds on a service-oriented component model to implement a flexible and maintainable representation of contextual information in order to increase integration and interoperability with pervasive applications. At design time, it allows the straightforward definition of contextual entities with a high level Java-based language. At runtime, it supports the dynamic construction of synchronized and observable entities published as services that can be used by applications. Our approach also comes with architectural and process guidelines to ease the work of designers and developers.

We have also enhanced the autonomic capabilities with an explicit autonomic manager, as illustrated by Figure 3. The purpose of this manager, using the DSL, is to create the context entities when needed by an application. Thus, only relevant concepts are built and maintained by the platform.

Context management is then structured into two modules (as shown in Figure 3):

- **context module** is a representation of contextual information, where entities (relevant persons, places, and objects) are modeled as components that implement context services.
- **context manager** is in charge of building and updating at runtime the context module, based on high-level goals and the current situation.

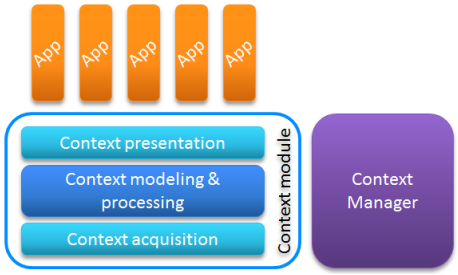

Figure 3. Context modules architecture.

We believe that this approach is well-adapted to reactive applications (or applications parts) implemented near pervasive

resources. The integration between applications and resources is straightforward: the needed services are directly proposed by the context to the applications. In addition, when some data is changed, provided services are re-evaluated and events are sent to applications so that they can call the service again.

B. *Two-phase approach*

Our approach naturally leads to a two-phase context development process. First, domain engineers define a context that can be used by a set of applications that are expected to run on a same platform. Second, applications developers use this context in order to simplify their code and concentrate on business logic.

The first step, usually called Domain Engineering, involves several actors with different skills as illustrated by Figure 4.The tasks to be performed are the following:

- Application and platform developers identify and determine the necessary concepts to be included in the context, in conformance with platform capabilities. Context services are specified in a Java based description language, presented in the next section, and are directly used by applications.
- Platform developers implement context services defined in the first step and additional ones useful for mediation or processing. Component development is simplified by using a Domain Specific Language, presented in the next section, with facilities to express data source's synchronization.

Platform developers implement context manager components in charge of service context provisioning and configuration. The proposed component DSL also includes facilities for this task.

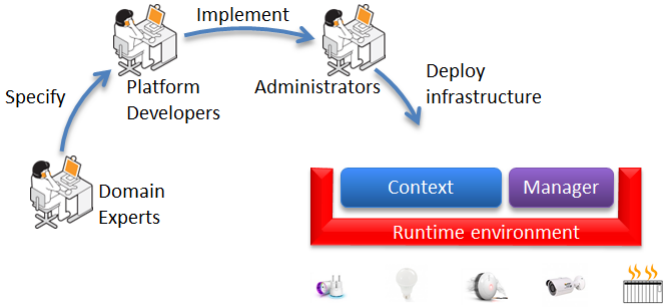

Figure 4. Context development and deployment.

Context components are deployed and executed on top of a pervasive runtime platform that handles their lifecycle. Specifically, our context management system has been integrated in a pervasive platform named iCasa [20] that is able to host multiple components and applications written with iPOJO. The iCasa/iPOJO runtime has been extended to handle all the non-functional aspects introduced by our component DSL, while retaining its service-oriented properties. The iCasa runtime has also been extended to offer appropriate probes and touchpoints to enable autonomic management of the context module, in accordance with the application needs.

Application developers use context services defined in the first step of context development to simplify their code and concentrate on business logic (Figure 5). At runtime, context representation depends on deployed applications and available resources.

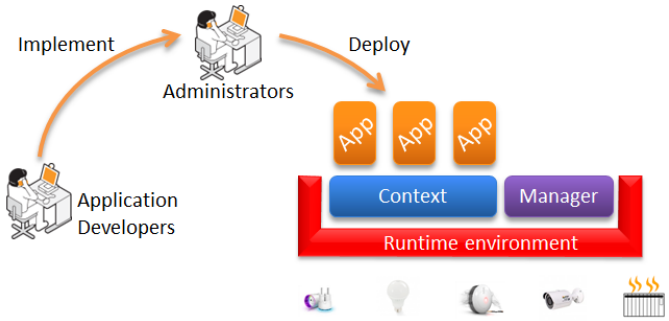

Figure 5. Application development and deployment.

IV. DSL FOR CONTEXT

A. *Overview*

Context is a representation of the surrounding environment. As such, it can be regarded as an explicit model [21] of the environment used by the pervasive applications. We use a service-centric approach to explicit this model. Context services must be described, implemented and dynamically provided at runtime. The following steps are necessary:

- Context service aims to provide a service description.
- Context entity aims to provide an implementation of context services.
- Context provisioning aims to provide a simple way to dynamically instantiate context entities.

For each of these steps, we provide a support through a Domain-Specific Language, defined by the meta-model in Figure 6. The concrete syntax is based on Java annotations and hides the dependency between concepts and the selected SOCM. The current implementation is based on iPOJO, the Apache OSGi service-oriented component model [17]. We provide build time processing to map our concepts to iPOJO concepts. Other implementations of our approach can be done, as long as the selected SOCM supports late-binding and dynamic instantiation.

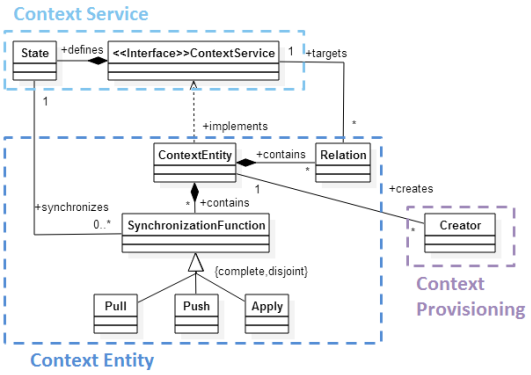

Figure 6. DSL meta-model.

B. Context Service

Application development relies only on service description to reduce coupling between context implementation and application business logic code. The service description must contain enough semantic and information to be used without ambiguity by the application. As depicted in Figure 7, we extend the OSGi service definition, which relies on a simple Java interface, with our DSL. It allows to clearly identify which interface will be exposed as a context service; `@ContextService`; and enhanced the service definition by state properties definition; `@State`; that will be valued and exposed at runtime by the service implementations.

At runtime, context service use in application is greatly simplified by iPOJO injection mechanism. Application components specify their context service dependencies with specific goals. Goals can be cardinality, filtering or ranking strategies. According to these goals, iPOJO autonomically allows opportunistic use of new context services and dynamicity resilience by dynamically injecting service implementation. Additionally, simple event mechanism is handled by calling application callback each time state properties are modified.

```
11 public @ContextService interface BinaryLight implements
12    GenericDevice{
13
14    public final static @State String POWER_STATUS="status";
15
16    public void getPowerStatus();
17
18    public void setPowerStatus(boolean state);
19
20    public void turnOn();
21
22    public void turnOff();
23
24 }
```

Figure 7. Java code for a context entity service.

C. Context Entity

In order to support a reliable, maintainable implementation of previous described context service and keep it as simple as possible, our approach is based on a domain-specific specialization of the iPOJO component model. We extend iPOJO meta-model, with specific concepts required to deal with context issues.

The concrete language used in iPOJO to define components is based on plain Java classes marked with annotations (see Figure 8) to declare non-functional aspects handled by the platform. We have enhanced the iPOJO design time tools with specific annotations required to declare specialized context concepts. For example, annotation `@ContextEntity` allows specifying a context entity that can implement context services.

Concretely, each state property declared in the implemented context service specification must be referenced, through `@State.Field`, in the implementation class as simple Java attribute to be easily manipulated during the service implementation. Each state property is synchronized through dedicated function:

- Functions to retrieve data from an external entity: `@State.Pull`/`@State.Push`
- Function to influence an external entity: `@State.Apply`.

The synchronization process is bi-directional in order to enable application to act on the context. These annotations enable to specify goals, like the frequency to call a `@State.Pull` function if it is periodic.

In our approach, a relation represents a link between two context entities, a pointer from an instance to another. It is implemented by a service dependency. This pointer is particular: it contains semantic information on its source, target and nature. It can be used to enhance context with semantic information. As a service dependency it can be used for synchronization process. Goal can be specified like cardinality or ranking.

```
10 @ContextEntity(services = {BinaryLight.class,…})
11 public class ZigbeeBinaryLightImpl implements BinaryLight{
12
13    @ContextEntity.State.Field(service =
14    BinaryLight.class,state = BinaryLight. POWER_STATUS)
15    public boolean powerStatus;
16
…     /**Service Implementation relying only on state field**/
20    public void setPowerStatus(boolean status){
21              powerStatus = status; }
22
20    public boolean getPowerStatus(){
21              return powerStatus; }
21
…     /**Specific Zigbee synchronization for powerStatus**/
26    @ContextEntity.State.Pull(… frequency = 10
27    ,unit=TimeUnit.SECONDS)
28    Supplier<Boolean> getPowerStatusFromDevice = () -> {
…
30    };
31
32    @ContextEntity.State.Push(…)
33    public boolean getPowerStatus(){
…
36    }
37    @ContextEntity.State.Apply(…)
38    Supplier<Boolean> getPowerStatusFromDevice = () -> {
…
45    };
…     /**Injected Relation field**/
58    @ContextEntity.Relation.Field(value = "isIn",…)
59    @Require (optional = true,…)
60    private Zone injectedZone;
61 }
```

Figure 8. Java code for a context entity service implementation.

At runtime, context entities are wrapped as iPOJO components. Component containers can be extended by iPOJO modules, called handers. The framework provides a number of off-the-shelf handlers in charge of global aspects of the platform, like service publishing or dependency injection. More importantly, it is possible to develop new iPOJO handlers to take charge at runtime of particular concerns.

We implemented two new iPOJO handlers, as show in Figure 9, dealing with specific context concerns. The handlers are described as follow:

- *Synchronization Handler*: It deals with the state synchronization of entity components. It keeps the state properties up-to-date by managing the synchronization functions. Different strategies can be specified to do so. For example, the handler can periodically call *pull* functions or just wait for *push* callbacks to keep the state up-to-date. Additionally, the handler is in charge of publishing state properties as service properties. This publication has two main interests: it allows

processing of more advanced filters and state updates can be reported to the application without the burden of an Observer pattern, by relying on iPOJO notification mechanism.

- *Relation Handler*: The relation handler is in charge of the dynamic service binding of relations.

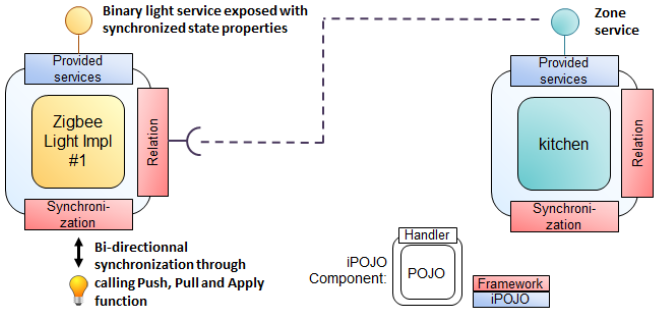

Figure 9. Service-oriented component view of context model.

D. Context Provisioning

Context service provisioning is guided by the discovery of external events, e.g. a device joining the network or a user interaction with a web dashboard. Approaches like RoSe [18] or MUSIC [22] provide pattern to modularized and maintain the discovery of external event at runtime but, no support is provided for dynamic instantiation of context service provider. iPOJO runtime supports this behavior but its establishment remains highly technical, tightly coupled to the iPOJO model and de facto become less feasible for developers. Our middleware provides autonomic facilities regarding this issue, without cluttering the discovery code. As depicted in Figure 10, discovery code emits now instantiation requests (previously it was direct instantiation) and the middleware choose to process or stock the requests according to the application contextual service requirements.

```
13    @Creator.Field Creator.Entity<ZigbeeBinaryLightImpl>
14 zigbeeBinaryLightCreator
15
16    public void catchZigbeeDiscoveryEvent(Map<Parameter>
17 param){
18    String id = …;
…
20    zigbeeBinaryLightCreator.create(id,param);
21 }
```

Figure 10. Java code for a context service dynamic provisionning.

E. Autonomic Execution

As depicted in previous section, each step of context development is probed with autonomic touchpoints. We will see how our context management middleware can benefit from this and provide autonomic features.

First, we assume that applications are developed following the iPOJO model. Each application can be composed of several components and relies on context services. Regarding to dynamicity, iPOJO naturally infuses autonomic behavior in the component's container. Therefore, application can benefit of late-binding and dynamic service substitutability. This level of adaption is specified within iPOJO annotation.

Secondly, when a context-aware application is deployed and executed, our context manager knows its context service dependency. Based on this knowledge, adaptation can be acted. Hence, it is possible to dynamically realize the following changes:

- Enable or disable context entity provisioning;
- Modify specific synchronization parameters;
- Replace context providers.

This autonomic behavior allows managing fault tolerance by switching of context provider, if a new one is available. All of this adaption logic is hard coded in our context manager. We are currently investigating integration with dynamic deployment [23], to provide fine grained context management.

V. Evaluation

The iCasa environment [24] is made of three tools: an Integrated Development Environment based on an eclipse plug-in; an execution platform based on OSGi and RoSe; a smart home simulator to quickly test pervasive applications.

The execution platform is running on a home gateway, which hosts several applications and offers dynamic deployment facilities. Applications belonging to domains like safety, comfort or health care have been developed. For example, iCasa hosts part of a home care application called actimetrics which measures and analyses the motor activity of elderly. Its purpose is to track behavioral changes to early diagnosis degenerative diseases like Alzheimer [16].

We provide an evaluation from a software engineering point of view, encompassing several metrics link to design time activity. We choose the following metrics: number of lines of code, cyclomatic complexity (this metric gives indication on maintainability, reliability and testability) and technical debt (evaluation of the effort needed to fix all issues). All this metrics are computed and provided by an open source quality management platform, SonarQube [25]. We run our evaluation on two different projects: we entirely restructured the context of the iCasa execution platform and its associated simulator; and we refactored an application on the top on the restructured platform. Graphics on Figure 11 and Figure 12 present the chosen measurement comparisons, respectively for the iCasa execution platform and the application.

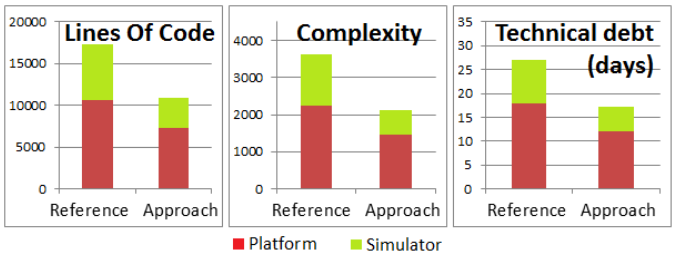

Figure 11. Evaluation on iCasa platform and simulator.

The first evaluation concerns the iCasa platform and its associated simulator. In the reference, contextual information in iCasa execution platform and simulator was computed in an *ad hoc* way. It was heterogeneously developed according developer's will and spread into the platform. It was difficult to extend the context or make it evolve since there wasn't any

consistency. For the evaluation, we redesigned the context by applying our approach and we compared the two versions (Figure 11). Functionalities provided by the reference and our approach are:

- A set of abstraction for device, location and user and their implementations;
- A web interface acting as a dashboard;
- A script language allowing to dynamically instantiate simulated device, location and user.

Thanks to the code provided by the handler and the simple event mechanism, the number of line of code decrease. By clearly identifying synchronization functions and limiting their number, cyclomatic complexity have been reduced. We also noticed that the restructured implementation presents a high percentage of duplicated lines (approach 7%, reference 3%) due to iPOJO technical limitations: it doesn't support inheritance. The number of lines could therefore be reduced more.

Our approach notably improves context development. It offers non-functional technical facilities. The context is modularized, extensible, and autonomic. The whole software is more consistent, testable and maintainable.

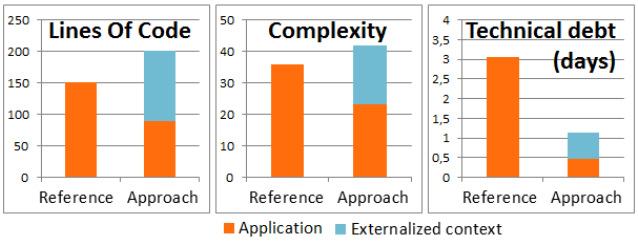

Figure 12. Evaluation on Light Follow Me application.

The second evaluation compares two versions of Light Follow Me application build upon the reference and the restructured platform (Figure 12). Light Follow Me turns the lights off and on depending on the presence of a user for each room of a house. This application is simple yet it encompasses all requirements presented before. It is a typical home pervasive application that doesn't need complex reasoning algorithm, facing the dynamism of the environment (light and sensor can appear/disappear) and directly influencing the user environment through switching on/off the light. In the reference implementation, the application manually processes information like presence per zone by directly reasoning over the sensors and their location. In our approach we choose to externalize the processing of this information with a dedicated presence per zone context service (blue part on Figure 12). This presence per zone service can be shared between applications and evolves independently regarding to the business code of these applications.

The externalization of the presence service produces an overhead in terms of line of code and complexity. This is due to the fact that all the logic of provisioning the service must implemented in our solution. However, this overhead can be shared by many applications. So if we analyze only the application business code (orange part on Figure 12), it is approximately divided by 2 and *de facto* become easy to test, maintain and evolve.

We can summarize that externalizing the context adds an additional development task and the resulting architecture is more complicated, but this cost can be mutualized and shared among several applications. Moreover, new applications can be develop on top of more abstract services, easing their implementation.

VI. Related Work

Developing context-aware applications is one of the hot research topics for the last decade. Naturally many software architectures emerged to reach this goal. We compare our proposed architecture to the existing ones: The Context Toolkit [7], COSMOS [26] SOCAM [27] and another SOC-based context model [28]. Many more architectures are available, but these ones are representative of the global trend.

The Context Toolkit [7] promotes code-reuse through the composition of distinct artifact called widgets to build the context. These widgets are used to hide the complexity of sensors and abstract context information in a suitable way to fit applications need. These reusable blocks are explicitly linked at design time, each block deciding which blocks to use. Our approach is similar in the sense that we divide the context in individual small pieces. The key differences with our work are that we delegate the composition at runtime with more variability expressed at design time thanks to SOCM properties. Moreover our entity relation like model offers more flexibility to design complex context.

COSMOS, COntext entitieS coMpositiOn and Sharing [26], is a component based context middleware. Each pieces of context is reified as a component called Context Node organized in a hierarchical structure. This approach and ours address the separation of concerns by offering several built-in mechanisms like push/pull notifications and reduce the developer's work. However, the strictly hierarchical approach of COSMOS context makes it difficult to model with horizontal relations. Moreover, component specifications are strictly defined at design time, so runtime extensibility proposed by our system of relation will be hard to achieve.

The Service-Oriented Context Aware Middleware (SOCAM) [27] is an ontology-based context middleware. SOCAM architectures rely on several components: Context Providers (extracting context from internal and external data sources, and converting them in ontological instance), Context Interpreter (reasoning engine performing inference to extract high-level context and store it in knowledge base), Context-aware Mobile Service (application that consume context), and Service Locating Service (a registry where providers and interpreters are registered, where other components can search specific providers or interpreters to fit their needs). SOCAM envisions a highly structured context model with ontology in order to benefit from all the powerful processing tools induced by this approach, like reasoning engine. So it generates a programming model through a query language and rules, contrary to our programming model that relies on Java service specifications that we consider more adapted to develop added-value services.

In [28], a work dealing with proactive adaptation and context management based on a SOCM architecture is presented. It underlines the fact that context interactivity is not just about providing the most powerful modeling and reasoning engine. Indeed, applications also can deal with context in a proactive manner, with the ability to change the context through actuators. Our approach, in this sense, is very similar because previous works say little about how to influence context. However, to achieve this goal, a specific query language that generates a cost on the learning curve is provided, whereas we prefer a traditional Java programming model.

VII. CONCLUSION

In this paper we presented a comprehensive approach to build and run Context as a Service. We provide a simple way to address service description, implementation and provisioning in a modular, maintainable and autonomic way. This solution can be integrated in an enriched execution platform as demonstrated. Our work focuses on providing tools to build and execute a runtime autonomic model of context. This model is probed with autonomic touchpoint, synchronized with external entities and can be enriched dynamically by new relations or services. Applications developed using SOC paradigm upon this model can also dynamically add new elements in the context in order to better fit their needs.

As limitations, we notice that our runtime implementation currently maintains a possibly large, in-memory, representation of context. Additional performance optimizations are required to cope with the needs of realistic applications, in terms of memory scalability and footprint.

Further perspectives of our work include extensions to handle more complex synchronization scenarios and provide off-the-shelf processing components like aggregation processing. Moreover, as depicted in the background section, our work focuses on integrating context in the network edge like home pervasive gateway, but as explained many applications can be divided in two parts. On one hand, these applications performed immediate action on the user environment and are located in the network edge. On the other hand, they handle complex reasoning on large historical set of data and are in most case execute in the cloud. It will be very interesting to investigate how our context model can cohabitate with a more “data-centric” context management located in the cloud.

REFERENCES

[1] M. Weiser, “The computer for the 21st century,” in Scientific American, vol. 265, pp. 94-104, 1991.

[2] M. Satyanarayanan, “Pervasive computing: vision and challenges,” in Personal Communications, IEEE, vol. 8, pp. 10–17, 2001.

[3] F. Mattern, “The vision and technical foundations of ubiquitous computing,” in Upgrade European Online Magazine, pp. 5-8, 2001.

[4] L. Atzori, A. Iera and G. Morabito, “The Internet of things: a survey,” in Computer Networks, vol. 54, pp. 2787-2805, October 2010.

[5] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell and K. Nahrstedt, "A middleware infrastructure for active spaces," in IEEE Pervasive Computing, vol. 1, no. 4, pp. 74-83, Oct.-Dec. 2002.

[6] M.P. Papazouglou and D. Georgakopoulos, “Service-oriented computing,” in Communications of the ACM, vol. 46, issue 10, 2003.

[7] A.K. Dey, “Understanding and using context,” in Personal and ubiquitous computing, vol. 5, pp. 4-7, 2001.

[8] S. Lee, J. Chang and S. G. Lee, “Survey and trend analysis of context-aware systems,” in Information-An International Interdisciplinary Journal, vol. 14, pp. 527-548, 2011.

[9] iCasa platform and simulator releases available at http://adeleresearchgroup.github.io/iCasa.

[10] C. Perera, A. Zaslavsky, P. Christen and D. Georgakopoulos, “Context aware computing for the internet of things: a survey,” in Communications Surveys & Tutorials, vol. 16, pp. 414-454, 2014.

[11] S. Ahn and D. Kim. “Proactive context-aware sensor networks,”.Proc. Wireless Sensor Networks Workshop (EWSN 2006), Springer LNCS vol. 3868, pp. 38-53, 2006.

[12] P. Hu, M. Portmann, R. Robinson, and J. Indulska. “Context-aware routing in wireless mesh networks,” Proc. ACM international Conference on Context-awareness for self-managing systems (CASEMANS '08), ACM, pp. 16-23, 2008.

[13] K. Henricksen, J. Indulska and A. Rakotonirainy. “Modeling of context information for pervasive computing applications,” Proc. Pervasive Computing: Conference (Pervasive 2002), Springer LNCS vol. 2414, pp. 167-180, 2002.

[14] D. Preuveneers and Y. Berbers. “Adaptive context management using a component-based approach,” Proc. IFIP Distributed Applications and Interoperable Systems (DAIS 2005), Springer LNCS vol. 3543, pp. 14-26, 2005.

[15] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. “Fog computing and its role in the internet of things;” Proc. Workshop on Mobile cloud computing (MCC 12), ACM, pp. 13-16. 2012.

[16] P. Lalanda, S. Chollet, C. Aygalinc and E. Gerbert-Gaillard, “Service-based architecture and frameworks for pervasive health application,” in Emerging Technologies & Factory Automation, pp. 1-8, 2015.

[17] C. Escoffier, R. S. Hall and P. Lalanda, “iPOJO: an extensible service-oriented component framework,” in Service Computing, pp. 474-481, 2007.

[18] J. Bardin, P. Lalanda and C. Escoffier, “Towards an automatic integration of heterogeneous services and devices,” Proc. IEEE Service Computing Conference, IEEE, pp. 171-178, 2010.

[19] RoSe framework source code available at https://github.com/AdeleResearchGroup/ROSE.

[20] C. Escoffier, S. Chollet and P. Lalanda, "Lessons learned in building pervasive platforms," Proc. IEEE Consumer Communications and Networking Conference (CCNC), IEEE, pp. 7-12, 2014.

[21] E. Seidewitz, “What models mean,” in IEEE software, vol. 20, issue 5, pp. 26-32, 2003.

[22] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli and U. Scholz. “MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments,” In Software Engineering for Self-Adaptive Systems, Springer LNCS vol. 5525, pp. 164-182, 2009

[23] O. Günalp, C. Escoffier and P. Lalanda, "Rondo: A tool suite for continuous deployment in dynamic environments," Proc. IEEE Services Computing Conference (SCC), IEEE, pp. 720-727, 2015.

[24] Pervasive Computing in Practice teaching website (featuring iCasa environment): http://self-star.imag.fr.

[25] SonarQube ; an open platform to manage code quality; website: http://www.sonarqube.org.

[26] D. Conan, R. Rouvoy and L. Seinturier, “Scalable processing of context information with COSMOS,” Proc. Distributed Applications and Interoperable Systems (DAIS 2007), Springer LNCS vol. 4531 pp. 210-224, 2007.

[27] T. Gu, H. K. Pung and D. Q. Zhang, “A service-oriented middleware for building context-aware services,” in Journal of Network and computer applications, vol. 28,pp. 1-18, 2005.

[28] S. VanSyckel, G. Schiele and C. Becker, “Extending context management for proactive adaptation in pervasive environments,” in Ubiquitous Information Technologies and Applications, pp. 823-831, 2013.