CA3M: A runtime model and a middleware for dynamic context management

Chantal Taconet, Zakia Kazi-Aoul, Mehdi Zaier, and Denis Conan

Institut Télécom; Télécom SudParis; CNRS UMR SAMOVAR
9 Rue Charles Fourier, 91011, Évry Cedex, FRANCE
{chantal.taconet,zakia.kazi-aoul,
mehdi.zaier,denis.conan}@it-sudparis.eu

Abstract. In ubiquitous environments, context-aware applications need to monitor their execution context. They use middleware services such as context managers for this purpose. The space of monitorable entities is huge and each context-aware application has specific monitoring requirements which can change at runtime as a result of new opportunities or constraints due to context variations. The issues dealt with in this paper are *1)* to guide context-aware application designers in the specification of the monitoring of distributed context sources, and *2)* to allow the adaptation of context management capabilities by dynamically taking into account new context data collectors not foreseen during the development process. The solution we present, CA3M, follows the model-driven engineering approach for answering the previous questions: *1)* designers specialised into context management specify context-awareness concerns into models that conform to a context-awareness meta-model, and *2)* these context-awareness models are present at runtime and may be updated to cater with new application requirements. This paper presents the whole chain from the context-awareness model definition to the dynamic instantiation of context data collectors following modifications of context-awareness models at runtime.

Key words: ubiquity, context-awareness, meta-modelling, model at runtime.

1 Introduction

Nowadays, the use of mobile devices (*e.g.*, smart phones, PDA) is getting more and more popular. Mobile devices are not only used as simple phones as the range of distributed applications developed for mobile devices increases drastically. These trends indicate that more and more users depend on ubiquitous applications for their daily life.

Ubiquitous applications must gain in capabilities to adapt themselves and also to manage their autonomy. The term *context-aware application* appeared in 1994 [23]. Since then, many models that describe applications contexts have been proposed, many context-aware middlewares and services have been designed, and many context-aware applications have been implemented. However, about

15 years later, some improvements in software engineering are still necessary to enable ubiquitous context-aware applications to be easy to design, to implement and to reconfigure.

Context-aware applications need both *1)* to collect high-level observations meaningfull to them and *2)* to identify situations under which they need adaptations. High-level observations may be computed from different distributed sources such as operating systems, user profiles and environment sensors. Context managers are services in charge to compute those high-level observations [2, 10, 11, 21]. In this paper, we want to enable applications to dynamically make use of new context managers coming from different frameworks in open environments. The design of context-aware applications for ubiquitous environments has seen the advent of a new stakeholder in the application design process that we call the "context-awareness designer" in the sequel of the paper. Context-awareness designers need the support of model engineering to manage the wide diversity of context data in ubiquitous environments. Several context modelling approaches such as context profiles and context ontologies have been surveyed in [25]. Specifying application context-awareness with Model Driven Engineering (MDE) enables designers to draw links between the context models and the application models. Context-Awareness Domain Specific Models (DSM) further the MDE approach and promote context-awareness models to express the dynamic variability due to context changes [4, 20]. However, context-awareness models still lack the ability to define the contracts which link context-aware applications to context managers. Thus, the approach promoted by the paper is to use MDE for guiding context-aware application designers in the specification of the monitoring of distributed context sources.

MDE is often used statically to automate code generation. Recent research works propose to reify models at runtime [3]. This is the direction we follow in this paper in order to take into account new ubiquitous environments with new context sources requirements not foreseen during the development process. Models at runtime contribute to maintain consistency across design time decisions and runtime adaptations by enabling models to be updated at runtime.

We propose CA3M, a Context-Aware Middleware based on a context-awareness Meta-Model for the following goals. Firstly, CA3M guides context-aware application designers in the specification of the monitoring of distributed context sources. Secondly, CA3M enables applications to adapt their behaviours by dynamically taking into account new context data sources not foreseen during the development process. In our approach, in addition to classical application models such as UML models, a context-awareness model is built by the context-awareness designer . This model may be updated during the application lifetime —*i.e.*, design, deployment, runtime. At runtime, the CA3M middleware dynamically constructs bridges between context-aware applications and context managers to reify the monitoring elements of the context-awareness model of the application.

The outline of the paper is as follows. In Section 2, we motivate and give the objectives of CA3M through an illustrative scenario. Then, we present an overview of CA3M in Section 3. In Section 4, we introduce the CA3M meta-model and show the corresponding context-awareness models for the motivating scenario. We detail the implementation and provide experimental evaluation considerations in Sections 5 and 6. Then, we compare our contribution with regard to related work on context-awareness modelling and context management middleware in Section 7. Finally, we conclude and present some perspectives of our work in Section 8.

2 Motivation and objectives

We begin this section by introducing the terminology used in the sequel of the paper. Then, we present an illustrative scenario in Section 2.2 and finally we bring out our objectives in Section 2.3.

2.1 Terminology

We present here terminology adopted for CA3M and especially in the CA3M meta-model. Some of the following concepts such as entity and situation were already present in Dey's context definition [?].We have chosen simple definitions easy to manipulate by a context-awareness designer.

An *entity* is an element representing a physical or logical phenomenon (person, concept, etc.) which can be treated as an independent unit or a member of a particular category, and to which "observables" may be associated. A mobile device is an entity. An *observable* is an abstraction which defines something to watch over (observe). The battery level of a mobile device is an observable. Some observables may be computed from other observables, they are called *interpreted observables*. An *observation* represents the state of an observable at a given time. It is obtained from a context-manager named *collector* in the sequel of the paper. An *adaptation situation* is an observable which allows to track down a change of state in the space of the information of context. This change of state may require a reaction in the system. Such a reaction is called an *adaptation*. A mobile device battery state is an example of observable which may take a finite number of values (*e.g.*, LowBattery, AlmostLowBattery or NormalBattery) and each state may lead to a different application behaviour. The link between an observable and an application is defined through a context-awareness contract. *A context-awareness contract* may define for example the quality of context required by the application as well as the mode of communication between the application and the collector —*i.e.*, observation or notification.

2.2 Mobile-chat application scenario

We illustrate the dynamicity of an application context-awareness with a mobile-chat application. This scenario brings into play distributed context sources. It shows how the application behaviour is adapted according to context changes and highlights the necessity of context-awareness modification at runtime.

*Eric is a student in physics travelling from Paris to Geneva by train. He has arrived in Paris train station and must wait a couple of minutes before going on board. He decides to discuss with a friend of him named Susan. He launches the mobile-chat application. The basic version of this application is already downloaded in his mobile. This version may be extended at runtime by several plugins if necessary. Eric uses the WiFi connection offered in the train station. The "outdoor mode" allows Eric and Susan to use the voice option. On Susan demand, her location is shown on Eric's mobile as well as her distance from Eric. For this demand, the plugin "monitoring a peer location" is added at runtime by the mobile chat application already running on Eric's mobile. New observables (*e.g.*, Eric location, Susan location, and Eric and Susan distance) are added as well for this new plugin.*

Thanks to the good network connection between Eric and Susan, they can add the video option to the mobile-chat application. After a few minutes, another friend, named Rob, joins the conversation. Rob is also in an "outdoor mode" and has a good network link quality.

Eric knows that if the battery level goes down, the video option will be disabled for everyone, so he plugs in his mobile. Then, he wants to share an mp3 file with his two friends. During the file transfer, the link quality of the connection of Susan goes down. The file is then put into the pipe queue and will be transferred later. The video option is switched off for Susan but kept "on" between Eric and Rob.

*Now, it's time for Eric to go on board. As soon as he enters the train, both his cell phone and his laptop switch to the "indoor mode" in order not to disturb the other passengers with a voice communication. Eric sits down and plugs in his laptop. The conversation with his friend was not over. However, the "indoor mode" forced the application to move from a voice-based communication to a text-based communication. In addition, the WiFi connection is automatically replaced by a 3G connection. In this situation —*i.e.*, Eric using a 3G connection—, the video option cannot be used between Eric and his peers.*

This scenario justifies the context-awareness of the mobile-chat application in order to cope with different user profiles and preferences, different terminal capabilities, and different elements of the environment in a distributed setting. The context-awareness designer may define the environment which should be taken into account before and during the execution of the mobile-chat application. Context-awareness on Eric's mobile depends not only on Eric's context, but also on Susan's and Rob's ones (*e.g.*, the kind of the connection on both sides may be used to evaluate the quality of the link between Eric and his peers). Furthermore, the context-awareness model has to be updated at runtime. Indeed, new entities and new observables have to be added as new friends are integrated into the chat. The context-awareness model may change as well when new plugins are added to the basic version of the mobile-chat. For example, the geolocation plugin needs the monitoring of other elements such as the location of a peer and the distance between two peers.

2.3 Objectives

The previous scenario illustrates three main objectives handled in the design of CA3M.

Firstly, the scenario puts the stress on distributed monitoring. For instance, the basic ubiquitous mobile-chat application instantiated on Eric'c mobile terminal needs to monitor both Eric's and Susan's contexts. The decision concerning the video option depends on distributed context data. Therefore, context-awareness designers should be able to model distributed observations.

Secondly, a context-awareness model can specify the initial kind of observables to monitor (*e.g.*, link quality). However, at design time, the models can indicate neither on which computer they should be observed nor how many instances should be taken into account (*e.g.*, the number of persons involved in the chat is unknown). New plugins may also modify the monitoring requirements and lead to model updates. More generally, the adaptation may need runtime reconfigurations because of new execution conditions (which include the availability of new context sources). Subsequently, it is necessary to enable the context-awareness model to evolve during runtime.

Thirdly, for a given observable, several context sources may be utilised. The sources come from several providers, and may have different application programming interfaces (API). For instance, the user location may be measured by the GPS (Global Positioning System) of the user's mobile device or may be obtained by the nearest GPS found. It should not be the role of the context-awareness designer to choose the concrete collector to be used. The concrete collector is unknown at design time and is chosen afterwards and even at runtime. The middleware should be able to provide a meaningfull observation to the application for several collectors with different APIs.

3 CA3M overview

CA3M is a framework for both the design and the execution of context-aware applications. We briefly describe these two parts in Sections 3.1 and 3.2.

3.1 Design overview

Figure 1 depicts the CA3M context-awareness design process with, from left to right, the stakeholders, the activities, and the resulting artefacts.

The figure distinguishes roughly two kinds of activities: *(1) context specification and design* and *(2) application design*. The *context specification and design* comprises the design of collectors and the specification of contexts. It produces two kinds of artefacts: implementations and models. The presentation of this modelling task is out of the scope of the paper and can be found in [26]. Of course, the APIs of the collectors vary and this task should benefit from standardisation actions. Examples of proposed standards for modelling collectors are SensorML [19] for sensors and CIM [12] for operating system resources.

In this paper, we focus on the lower part of Figure 1, that is on application design. We divide this activity into two large-grain tasks to promote a new stakeholder: the context-awareness designer. The application designer produces the application model and classes. The context-awareness designer produces context-awareness models as explained in Section 4. In summary, we apply the principle of separation of concerns twice: *1)* separation of context data providers from context data users (context designers and context-awareness designers), and *2)* separation of application concerns from context-awareness concerns when designing and executing the application. In addition, context-awareness models are built at design time in order to be manipulated at runtime.

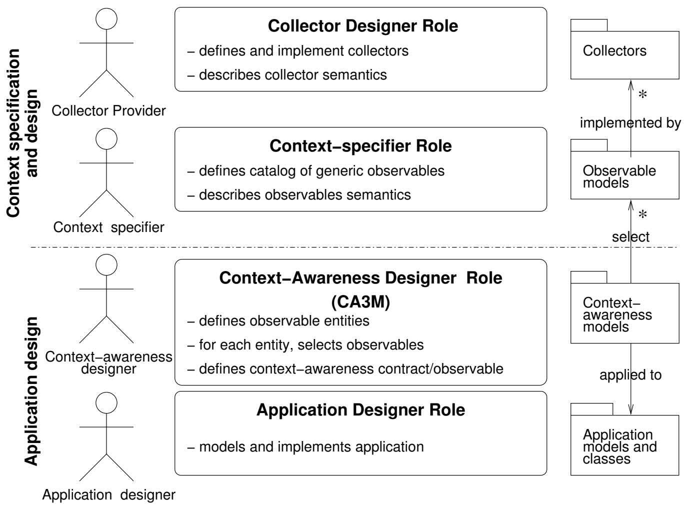
Context specification and design
Application design
Collector Provider
Context specifier
Context−awareness designer
Application designer
Collector Designer Role
– defines and implement collectors
– describes collector semantics
Context−specifier Role
– defines catalog of generic observables
– describes observables semantics
Context−Awareness Designer Role (CA3M)
– defines observable entities
– for each entity, selects observables
– defines context−awareness contract/observable
Application Designer Role
– models and implements application
Collectors
implemented by
Observable models
select
Context−awareness models
applied to
Application models and classes
*
*

Fig. 1. Separation of design tasks for producing context-aware applications

Context-awareness models have to conform to a specific meta-model presented in Section 4.1. This meta-model introduces meta-classes such as entities, observables, collectors, and context-awareness contracts. An instance of the context-awareness model is created at runtime which may be updated at runtime for example as new plugins are added to the mobile-chat application.

3.2 Runtime overview

Figure 2 shows the runtime architecture of CA3M. The architecture is divided into the context-aware application, CA3M, the distributed collectors, and the distributed context sources. A context-aware application accesses context management mechanisms through CA3M. CA3M drives the monitoring of the environment according to the context-awareness model. The distributed collectors provide context management and context interpretation. They collect data from distributed context sources, and they interpret and compute new high-level observations. The observations may be obtained from entities at different levels of the architecture: the system level, the network level, the environment level, but also the software level including observations of the context-aware application itself. CA3M corresponds to knowledge manager (the "K") of the K-MAPE autonomic computing loop presented in [14]. CA3M model manager is in the knowledge part of the loop, the collector is in the monitoring and analysing part of the loop, the CAController is in the planning part, and the application is of course the execution part.

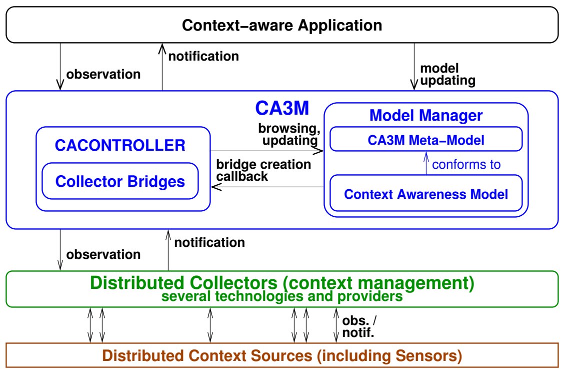

Fig. 2. Runtime architecture of CA3M

In this architecture, at any level, the interactions may be either top-down, or bottom-up. The top-down interactions correspond to the observation mode: the upper level synchronously requesting an observation. In the observation mode, since the upper level initiates the exchange, it controls the interruption of the application service. The drawback of this mode is that the upper level must know when an interaction is relevant *w.r.t.* its current context execution, that is when there is a significant probability of a meaningful context change happening.

The bottom-up interaction is called the notification mode. In this latter case, the contract defines when the lower level notifies the upper level: periodically or when the observation goes past a given threshold from the last notified value, or even at any change of the observation value. For instance, a notification may be sent to the application if the battery state changes (*e.g.*, from "NormalBattery" to "AlmostLowBattery"). In the notification mode, the application is less impacted by the monitoring of its environment than in the observation mode, provided that the application is able to express its contract. The context-awareness contract includes application operations to be called on notification. In conclusion, following [2], we decide to provide the two modes of interaction.

The collector bridge is defined following the bridge pattern [13]. The objective of a bridge is to decouple an abstraction from its implementation so that the two can vary independently. We use the bridge pattern for several reasons. Firstly, there may be plenty of collector implementations for a given observable. Secondly, collector may have slightly different APIs and we do need to decouple the application code from the collector implementation interfaces. Finally, we want runtime binding to the collector implementations and we wish to hide some tricky parts of the collector interfaces to the application programmer. The interfaces of the collector bridge abstraction is presented in Section 5.1.

CA3M comprises the CAController and the Model Manager. The CAController is in charge of binding to concrete collectors, creating collector bridges, and notifying the application when necessary as defined by the contracts. The model manager loads the application context-awareness model. The model manager handles query requests about the structure of the model as well as update requests to deal with runtime modifications of the model.

There are two kinds of interactions between the CAController and the Model Manager. Firstly, the CAController may use query operations for browsing the model. This kind of interaction is necessary after the initial loading of the model to create a bridge for each observable foreseen at the design phase. Secondly, in the case of model change, such as the addition or removal of an observable, the model manager triggers callbacks to the CAController in order to create or remove a collector bridge.

4 Context-awareness modelling

We present in this section a generic and extensible way to model the context-awareness of any application using the MDE approach. We describe the context awareness meta-model in Section 4.1. We illustrate our solution by modelling the context-awareness of the mobile-chat application in Section 4.2.

As shown in Figure 1, the context-awareness meta-model depends on the observable meta-model. This enables us (i) to share observable models between several context-aware applications and (ii) to exploit several observable models coming from different providers. The *observable meta-model* defines the observable and the interpreted observable concepts allowing them to be independent from applications. Thus, each observable model is then a catalog of pre-defined observables at the disposal of context-awareness designers. A context-awareness designer selects observables from one or several observable models which are relevant for an application and links the observables to entities. The concepts manipulated by the context-awareness designers are defined in the *context-awareness meta-model* which is detailed in the rest of this section.

4.1 CA3M context-awareness meta-model

Figure 3 describes the CA3M context-awareness meta-model. The ContextAwareSystem meta-class is the entry point of this meta-model. The left part of the meta-model defines the entities, their observables, the links between entities, the interpreted observables and the adaptation situations. The right part

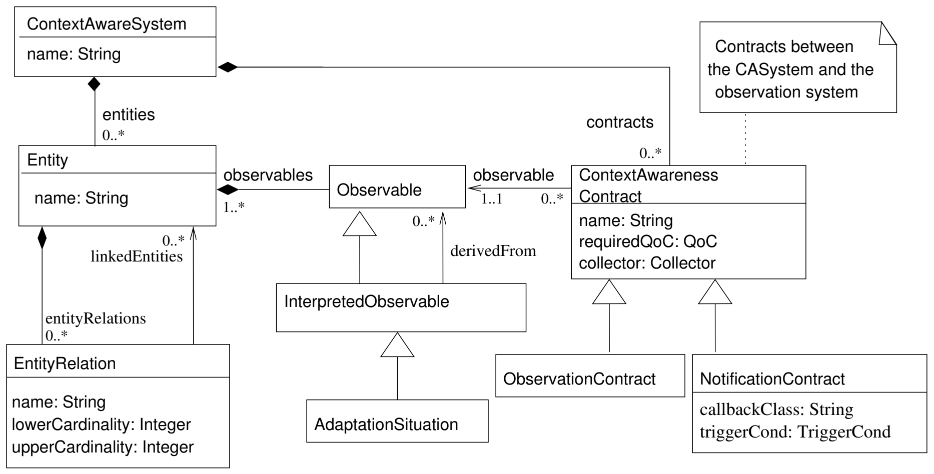

Fig. 3. The context-awareness meta-model

of the meta-model defines the context-awareness contracts. A context-awareness contract may be specialised for different context-awareness control mechanisms. A context-awareness contract is associated to an observable. For a notification contract, a context-awareness contract defines the events which trigger notifications and the class in the application model to be called in case of notification.

Entity represents a logical or physical element to be observed, *e.g.* a device. It allows a context-aware system to differentiate several distributed observables from different physical or virtual entities, *e.g.* the bit rate of two devices. An entity may be linked to another entity through the EntityRelation meta-class. An entity is linked to several Observable. An InterpretedObservable is linked to several source observables through the derivedFrom association, *e.g.* the battery state is derived from the battery level and the battery plugged observables. The type of the observables necessary to compute an interpreted observable is provided by the observable meta-model, not described in this paper (see [26]). When an interpreted observable is added to the model, the type of the source observables (defined by the derivedFrom association) are verified with the source types defined in the observable model. As type of the sources are defined in the observable model, source observables may be omitted from the context-awareness model. For example, the battery state, the battery level and the battery plugged observables are at evidence linked to the same entity (the user device). The battery level and the battery plugged observables are shown in dotted line in Figure 4 to show that they may be omitted from the context-awareness model. An AdaptationSituation is a kind of observable which has the characteristic to take a finite number of domain values and which is used to identify adaptation situations.

4.2 Mobile-chat context-awareness model

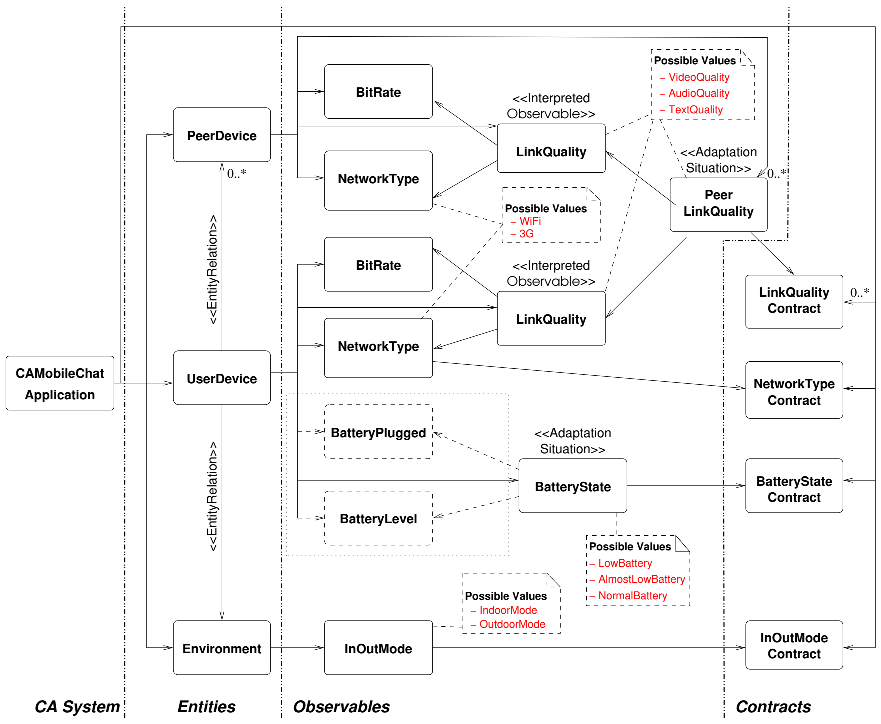

Fig. 4. The context-awareness model for the mobile-chat application

Figure 4 models the context-awareness of the basic mobile-chat application executing on Eric's mobile. This model conforms to the context-awareness meta-model presented in Figure 3. UserDevice, Environment and PeerDevice are the entities.

On each device, the observables BitRate and NetworkType are inputs to compute the interpreted observable LinkQuality. The context-awareness designer models the adaptation situation PeerLinkQuality computed from the interpreted observables (i) link quality of UserDevice and (ii) link quality of PeerDevice. For example, the adaptation situation value `VideoQuality` means that the video option is authorised in accordance with the link quality between the two parties. Several adaptation situations PeerLinkquality are instantiated during the execution (*e.g.*, one for Eric-Susan's link).

The possible values of the interpreted observable BatteryState (computed from its source values) can be `NormalBattery`, `AlmostLowBattery` and `LowBattery` $(NormalBattery = BatteryPlugged \vee BatteryLevel > 20\%$;

$AlmostLowBattery = \neg BatteryPlugged \wedge BatteryLevel < 20\% \wedge BatteryLevel > 10\%$; $LowBattery = \neg BatteryPlugged \wedge BatteryLevel < 10\%$). The observable InOut-Mode is associated to the entity Environment. According to Eric's scenario, it takes two possible values: `IndoorMode` and `OutdoorMode`.

Several context-awareness contracts are represented in this figure. LinkQualityContract, BatteryStateAdaptContract, NetworkTypeContract and InOutMode-Contract are notification contracts. The class of the application model refered by each contract has to implement a notification interface for the callbacks.

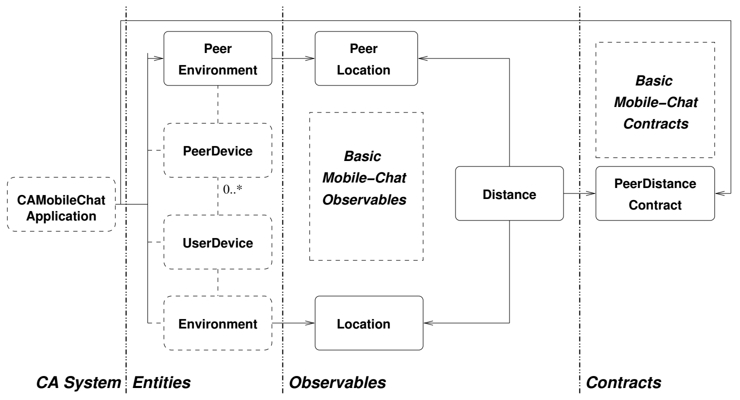

Fig. 5. Model extensions for the plugin location

Figure 5 shows the elements added to the model when the *monitoring a peer location* plugin is added to the application. Note that the initial model instance has already changed during the execution when Rob is added to the discussion. Here, another PeerDevice entity is instantiated and the link Eric-Rob is created on the fly.

5 CA3M prototype implementation

In this section, we present the CA3M implementation. We describe the CA3M class diagram in Section 5.1. Then, we explain modelling implementation choices in Section 5.2. As COSMOS is the context manager chosen for our evaluation, we describe COSMOS collector bridge in Section 5.3.

5.1 CA3M class diagram

Figure 6 depicts the CA3M UML class diagram representing the interfaces between a context-aware application and CA3M. We explain how to use CA3M in notification and in observation modes, and how to modify the application model to add or remove entities and observables.

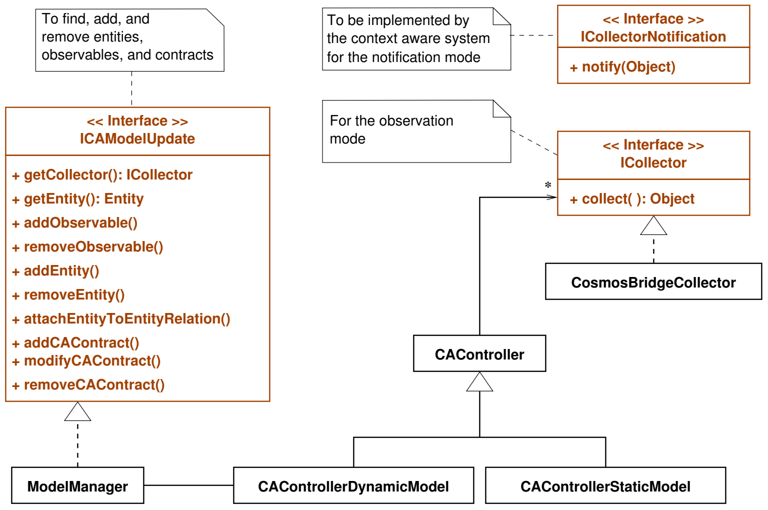

Fig. 6. CA3M related application interfaces

The CAController may use either a static model (class CAControllerStatic-Model) or a dynamic model (class CAControllerDynamicModel). In this article, so far, we focused on the dynamic model which allows CA3M to modify the model at runtime through the interface ICAModelUpdate. But if the mobile device, because of memory or library constraints, cannot afford the dynamic one, the static one may be used instead. In this latter case, CAControllerStaticModel is produced statically by transformation for a given model and the model cannot be updated at runtime.

In the mobile-chat scenario, we present the dynamic case. Each time a new peer user connects to the chat, a new PeerDevice entity and its associated observables are added to the model. Next, CA3M automatically creates a new bridge for each new observable, each bridge providing the interface ICollector for the observation mode. For the notification mode, the class defined in the notification contract should provide the interface ICollectorNotification.

5.2 Modelling implementation choices

The most popular meta-modelling languages for defining DSMs are MOF (*Meta-Object-Facility*) [16], ECORE from *Eclipse Modelling Framework* (EMF) [6] and UML Profile [17]. Designing the context-awareness model as a UML profile was not possible because with UML profiles we could not define associations between profile meta-classes. Between MOF and EMF, we have chosen EMF because of the availability of many EMF tools.

For the static CAController, we use ECORE models for transformation purpose to generate CAControllerStaticModel classes. When possible, the dynamic

CAController is used instead. At runtime, the model manager loads an application context-awareness model, accessed and updated through the EMF generated API. Through this API, new entities, observables, and contracts may be added to the model at runtime. Thanks to an EMF adapter, insertions of observables trigger the creation of bridge collectors.

5.3 CA3M bridge illustrated with COSMOS collector bridge

CA3M architecture allows the CAController to be interfaced with several context management frameworks. The constraints on the collector framework are the following ones. The collector framework should provide notification and observation modes and be able to compute high-level observations from distributed observations. At least one collector bridge has to be implemented per collector framework to wrap collector API with CA3M API. The bridge class has to be designed to enable the bridge to work with any observation class. Several bridges may be implemented according to the kind of binding from the bridge to the collector. We design two bindings: one for the connection to an external collector and the other one for an instantiation of the collector into CA3M.

For our evaluation, we have interfaced CA3M with the COSMOS framework [9]. COSMOS offers tools to collect, interpret and process context data. We have chosen COSMOS because it provides developers with the ability to define new finely tuned interpreted observables. The basic structuring concept of COSMOS is the *context node*. The architecture of a context node is component-based. For instance, the component (or context node) BatteryState is the composition of the context nodes BatteryPlugged and BatteryLevel with a context operator realising the logical and comparison operations mentioned previously. Observation and notification modes are provided through Pull and Push interfaces, respectively.

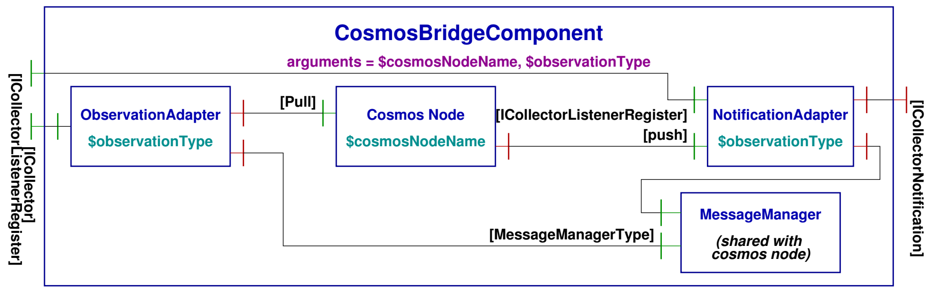

Fig. 7. The COSMOS Bridge component

Figure 7 presents the COSMOS bridge component. The bridge hides COSMOS specific interfaces and COSMOS internals using the FRACTAL component model [5] and Dream [15] message management for context reporting. It includes adapters to transform COSMOS specific message chunks to application observation types. The bridge provides CA3M interfaces for collecting observations and registering notification handlers. It forwards notifications to all the registered ICollectorNotification interfaces.

6 Experimental evaluation

6.1 Performance evaluation of the prototype

We present the overhead incurred by using CA3M compared to a direct usage of COSMOS collectors by the application. We have conducted performance measurements on a laptop PC with the following software and hardware configuration: 2.8GHz processor, 512MB of RAM, GNU/Linux Fedora 9, Java Virtual Machine Sun JDK 1.6, COSMOS 0.1.5, Fractal implementation Julia 2.5.2. Each test was run 1000 times. A garbage collection and a warm-up phase occurred before each measure.

The Table 1 presents the average measures with a 95% confidence interval. The table includes the following overhead measurements. The two first lines presents the time necessary for the initial model loading. Model loading concerns reading the EMF model file and reifying the concepts in EMF objects. The third line gives the overhead time for the collector instantiation, the overhead comes from the bridge instantiation. The fourth line presents the overhead time for an observation. The overhead comes from the transformation of an observation from a Cosmos `Chunk` to an application Java object. Lines five and six give the overhead concerning the memory usage after all bridge instantiations (the memory usage is given by the difference between `totalMemory` et `freeMemory` in the JVM). The CA3M overhead follows from model management (`emf` library), and bridge management (especially the reflexivity necessary to interconnect a COSMOS bridge to any kind of context node).

1-	model load (1 entity, 1 observable, 1 contract)	$723ms \pm 4$
2-	model load (100 entities, 100 observables, 100 contracts)	$785ms \pm 5$
3-	instantiation overhead (1 node)	$141ms \pm 8$
4-	observation overhead	$0.439ms \pm 0,003$
5-	memory overhead (1 entity, 1 observable, 1 contract)	$0,56MB \pm 0,05$
6-	memory overhead (100 entities, 100 observables, 100 contracts)	$1,26MB \pm 0,07$

Table 1. CA3M overhead measurements

6.2 CA3M experimentation

We evaluate the context-awareness meta-model with the mobile-chat application and also with a mobile commerce application. For these applications, we are able to share common ubiquitous observables (*e.g.*, BatteryState) defined in an ubiquitous observable model. In a second step, we define each application context-awareness models. In the two applications, we are able to reconfigure application context-awareness both statically and at runtime. The complete chain from context-awareness modelling to context collector bridge creation, observation and notification is tested. We validate the whole process with the COSMOS context manager. Through these experimentations, the following criteria are validated. The observable model is reusable through different applications. CA3M enable designers to define interpreted observables computed from distributed source observables. A context-awareness model may be modified at runtime. And a collector bridge may be instantiated at runtime enabling runtime collector binding.

7 Related Work

Since CA3M links context-management frameworks, context-awareness models and context-aware middleware, the related work section deals with these three kinds of research works. Many context management framework [10, 11] have been designed without context modelling. Due to the variety of contexts to be collected and analysed, we argue that context management needs the support of abstract context modelling.

Several research works address context-awareness with the MDE approach. ContextUML [24] was one of the first DSM for context-awareness. It presents a context-awareness meta-model for Web services. ContextUML models associate context constraints with filtering operations which should be applied to input or output messages of a Web service. CAPPUCINE [20] describes an MDE approach for dynamically producing product lines according to context information. Those models put the stress on adaptation mechanisms rather than context modelling. Those models are used for transformation purpose. CA3M, presented in this paper, also defines a DSM for adding a context-awareness concern in the application modelling stage. CA3M presents two main differences with the above research works. Firstly, CA3M modelling concepts enable application designers to express complex situations computed from distributed context observations. Secondly, the context-awareness model is not only used for transformation purpose, but is also available at runtime. Therefore, the application or the middleware are able to add new observables at runtime. In addition, Reichle & *al.* [22] proposes ontology-based context model and model framework, and the ontology model is present at runtime. Context model interactions are performed using CQL (Context Query Language). Some concepts present in this model such as entity and observables are similar to the CA3M concepts. Other ones such as context-awareness contracts do not exist and the framework does not propose the notification mode. Finally, [8, 18] consider context-awareness with model at runtime for application adaptation purpose but not for monitoring adaptation.

Several middleware solutions have been proposed for managing context-awareness. CARISMA [7] proposes to define context-aware profiles for tuning the behaviour of middleware services according to context information. These profiles are available at runtime. Applications may modify these profiles through a reflexive API. Profiles for context management are not addressed by CARISMA. CA3M offers also a reflexive API, not only to modify application behaviours, but also to add new observables which will be collected by new context collectors. Context-Toolkit [11] allows application developers to attach application handlers to context widgets. Context Toolkit triggers call-back to these handlers when context values change. The links between the handlers and the collectors are programmed rather than modelled in notification contracts as this is the case in CA3M. RCSM [27] offers CA-IDL, a language to define context situations and to specify the operations to trigger in these situations. Adaptation situations are evaluated by RCSM which triggers reactive operations at runtime. RCSM limitations come from the limited number of context information available in the CA-IDL grammar. The triggering of operations and the context monitoring are defined statically at compilation time.

Another issue of this article is to be able to be connected to several context management frameworks with different APIs. This may be handled statically by model transformation such as proposed in several context-awareness MDE works such as [1]. But this approach does not enable the middleware to choose the collectors at runtime. In MUSIC [21], the context-management framework includes an observation and a notification mode and the framework may be compared to the CA3M CAController. As in CA3M, they consider the dynamically insertion and removal of so called context plugins in the middleware. When a new collector is connected by the framework, the collector framework provides an archive which includes a wrapper between the collector and the framework. As a consequence, one wrapper has to be provided for each collector. In CA3M, we provide instead a bridge for all the collectors of the same family (*e.g.*, one bridge for all the COSMOS collectors). Thanks to bridges, CA3M may be connected to various kinds of context management frameworks more easily.

8 Conclusion and perspectives

In this article, we have presented the CA3M middleware. Our contributions are the following ones. CA3M provides tools to easily reconfigure the context-awareness of ubiquitous applications. Firstly, we have defined a context-awareness meta-model which defines concepts chosen for a new stakeholder that we call the context-awareness designer. This meta-model defines entities, observables, and context-awareness contracts. It allows designers to model distributed observables. We validate this meta-model through the definition of models for several ubiquitous applications. Secondly, we provide a middleware which, based upon an ubiquitous application context-awareness model, is able to connect to different context-manager collectors. CA3M offers two kinds of interactions between context managers and applications: the observation mode and the notification mode. We consider the design of bridges for binding to various context managers and we have validated and evaluated the approach building the bridge component for the COSMOS context manager. In summary, the main contribution of our proposition is to enable to update application context-awareness models at runtime. The advantage is to allow autonomous context-awareness.

This work may be extended in several directions. The MDE approach may be used not only to connect applications to collectors but also to produce high-level collectors from existing lower-level collectors. In addition, we have developped a bridge for the COSMOS context manager and we plan to validate our approach with additional context managers. We also believe that the model at runtime approach can benefit to a better choosing of collectors at runtime using a collector discovery service. Last but not least, we plan to extend CA3M to deal with adaptation mechanisms for changing the behaviour or the structure of applications. We intend to add various context-awareness adaptation contracts in the model and adaptation mechanisms in the middleware.

References

1. D. Ayed, D. Delanote, and Y. Berbers. *Computer Science*, volume 4635/2007 of *LNCS*, chapter MDD Approach for the Development of Context-Aware Applications, pages 15–28. Springer, Berlin / Heidelberg, 2007.
2. M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context Aware Systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.
3. N. Bencomo, G. Blair, R. France, F. Munoz, and C. Jeanneret, editors. *3rd Workshop on Models @ runtime*, Toulouse, France, September 2008.
4. N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008)*, Limerick, Ireland,, September 2008.
5. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software—Practice and Experience*, 36(11):1257–1284, Sept. 2006.
6. F. Budinsky, E. Merks, and D. Steinberg. *Eclipse Modeling Framework 2.0*. Addison Wesley, March 2008.
7. L. Capra, G. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting Reflection in Mobile Computing Middleware. *Mobile Computing and Communications Review*, 1(2), 2003.
8. C. Cetina, P. Giner, J. Fons, and V. Pelechano. A Model-Driven Approach for Developing Self-Adaptive Pervasive Systems. In *Models@runtime'08*, pages 97–106, Toulouse, France, September 2008.
9. D. Conan, R. Rouvoy, and L. Seinturier. Scalable Processing of Context Information with COSMOS. In Springer-Verlag, editor, *DAIS'2007*, volume 4531 of *LNCS*, pages 210–224, Paphos, Cyprus, june 2007.
10. J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. Context is Key. *CACM*, 48(3):49–53, Mar. 2005.
11. A. Dey, D. Salber, and G. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Special issue on context-aware computing in the Human-Computer Interaction Journal*, 16(2–4):97–166, 2001.

12. Distributed Management Task Force. Common information model (cim): Infrastructure specification, version 2.3 final. OpenGIS Implementation Specification, Oct. 2005.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley Professional Computing Series. Addison Wesley Professional, October 1993.
14. J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), Jan. 2003.
15. M. Leclercq, V. Quema, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), Sept. 2005.
16. Object Management Group. Meta Object Facility (MOF) Core Specification Version 2.0. OMG document formal/06-01-01, January 2006.
17. Object Management Group. UML 2.0 Superstructure Specification v2.1.1. OMG documents formal/2007-02-05, February 2007.
18. A. Occello, A. Dery-Pinna, and M. Riveill. A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service. In *Models@runtime'08*, pages 67–76, Toulouse, France, September 2008.
19. Open Geospatial Consortium. Opengis sensor model language (sensorml): Implementation specification, version 1.0.0. OpenGIS Implementation Specification, July 2007.
20. C. Parra, X. Blanc, and L. Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In *13th International Software Product Line Conference (SPLC)*, San Francisco, CA, USA, August 2009.
21. N. Paspallis, R. Rouvoy, P. Barone, G. Papadopoulos, F. Eliassen, and A. Mamelli. A Pluggable and Reconfigurable Architecture for a Context-aware Enabling Middleware System. In *Proc. 10th*, volume 5331, pages 553–570, Monterrey, Mexico, Nov. 2008.
22. R. Reichle, M. Wagner, M. Khan, K. Geihs, J. Lorenzo, M. Valla, C. Fra, N. Paspallis, and G. Papadopoulos. A Comprehensive Context Modeling Framework for Pervasive Computing Systems. volume 5053 of *Lecture Notes in Computer Science*, pages 281–295, Oslo, Norway, june 2008.
23. B. Schilit and M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, 1994.
24. Q. Sheng and B. Benatallah. ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services. In *ICMB, Sydney, Australia.*, pages 206–212, July 11-13 2005.
25. T. Strang and C. Linnhoff-Popien. A context modeling survey. In *UbiComp Workshop on Advanced Context Modelling, Reasoning and Management*, pages 34–41, Nottingham/England., September 2004.
26. C. Taconet and Z. Kazi-Azoul. Context-Awareness and Model Driven Engineering: Illustration by an e-commerce application scenario. In *CMMSE Workshop on Context Modeling and Management for Smart Environments*, pages 864–869, London, UK, November 2008.
27. S. S. Yau and F. Karim. An Adaptive Middleware for Context-Sensitive Communications for Real-Time Applications in Ubiquitous Computing Environments. *Real-Time Systems*, pages 29–61, 2004.