An architecture for supporting Development and Execution of Context-Aware Component applications

Nabiha Belhanafi Behlouli
GET / INT, CNRS Samovar 5157
9 rue Charles Fourier, 91011 Evry, France
Nabiha.Belhanafi@int-evry.fr

Chantal Taconet
GET / INT, CNRS Samovar 5157
9 rue Charles Fourier, 91011 Evry, France
Chantal.Taconet@int-evry.fr

Guy Bernard
GET / INT, CNRS Samovar 5157
9 rue Charles Fourier, 91011 Evry, France
Guy.Bernard@int-evry.fr

Abstract

Context-awareness is rarely considered in component-based middleware. In this paper, we present CAMidO, a Context-Aware Middleware based on an Ontology meta-model. CAMidO, enables designers to create component-based context-aware applications. It provides an ontology meta-model for context description. The described information are used by the middleware to automatically adapt the application to relevant context-changes. Context-awareness is managed by component containers through non functional properties. CAMidO manages all tasks related to context collection and application adaptation according to application designer description. Thus, it relieves application developers from programming these painful tasks.

1. Introduction

Mobile devices are characterised by dynamism in their environment. Applications running on top of these device types have to benefit from detecting environment changes in order to adapt their behaviour accordingly. But creating context-aware applications necessitates context management. It implies interaction with sensors for context collection, detection of context changes, and making decisions on appropriate adaptation actions. Without the support of a platform to automate context management, programming this is an additional task for the application developers.

Middleware plays an important role in hiding complexity of distributed applications development. Component-based middleware, such as the CORBA Component Model [15], EJB [8] and Fractal [13], enables developers to describe non-functional properties instead of programming them. Thus application developers focus on application functional code programming, while non-functional properties are managed automatically by component containers [19]. Containers provide a component execution environment and relieve developers from number of painful tasks. To ease development of context-aware applications, it becomes essential that middleware enables applications to support frequent changes of environment through context non functional properties and containers.

A lot of research has been undertaken to ease the development of context-aware applications by proposing either a meta-model for context description or by adding context awareness mechanisms to different kinds of middleware. But only few of them consider component-based middleware.

In this paper, we describe CAMidO, a Context-Aware Middleware based on an Ontology meta-model. CAMidO enables both context management and component-based context-aware application development and execution. With CAMidO, we apply component non functional properties to context-awareness. The idea is to ease the development of context-aware applications by providing a context meta-model for context description. This meta-model completed with context management entities, enables the middleware to collect context, and to react to context changes during the application component execution. Adaptation is carried out by component containers with controllers. These controllers are automatically generated by CAMidO during the compilation step.

The provided meta-model is based on an ontological description. It exploits the ontology’s characteristics to dis-
tribute context information and to reason about the collected context data. It enables both the middleware to interact with new sensors added in the ontology at runtime, and the application designer to add new adaptation rules and interpretation policies.

With CAMidO, creating context-aware applications consists in describing context and adaptation actions. The middleware uses the ontology to interact with sensors, detect relevant context changes and apply the appropriate adaptation actions. During execution, the component/container paradigm [19] enables the middleware to manage context-awareness.

This paper is organized as follows: Section 2 describes the CAMidO meta-model. Section 3 gives a description of the CAMidO compiler. Section 4 presents the CAMidO architecture. Section 5 provides a description of the adaptation mechanisms and the container architecture. The CAMidO implementation and first experimental results are presented in Section 6. Finally, Section 7 discusses some related works and Section 8 concludes the paper.

2 The CAMidO Meta-model

In order to facilitate the development of context-aware applications, a context-aware middleware infrastructure has to provide a model for describing contexts and any piece of information related to context awareness and application adaptation. Although it is necessary, it is often missing in context-aware middleware. The CAMidO middleware provides such a meta-model based on an ontological description.

Ontology is used to describe all information concerning context awareness of an application. CAMidO benefits from the ontological characteristics for modeling context, distributing these information between components of an application, and deducing adaptation policies according to the described relations between context information.

To illustrate the use of this meta-model, we give in this section the example of an e-commerce shopping application for mobile users. The idea is to provide a mobile user the possibility to visit the nearest department store and order the goods he wants to buy. Payment can be made before arriving at the store.

The application consists of four components: A Basket component named (B), a Data Base component (DB) sensitive to bandwidth, a User Interface component (UI) and a Local View component (LV) installed on the user laptop and sensitive to the network connexion state. The LV component allows the application to run even if the terminal is disconnected. The DB component contains information about a proximity store goods, their picture, their characteristics and their prices. When the user is near a department, he can ask the DB component for a department. This component provides goods textual information if low bandwidth is detected, or all the goods description (text, image and video) otherwise. The LV component is a cache component which contains some selected department description provided by the DB (according to the available disk space), if the UI component sends any request to the DB component and there is no network connexion between them, these requests are redirected to the LV component. When the connexion is restored, the LV component forward all the non treated requests to the DB component.

The CAMidO meta-model is written in the OWL language [2]. It allows the description of all the following elements: context, sensors from which data are collected, interpretation rules and adaptation policies. As shown in Figure 1, the proposed meta-model is divided into three levels, an ontology is associated with each level.

The first ontology is associated with the middleware level, it contains information about sensors with which the middleware can interact. This information may be used by the application running on top of the CAMidO middleware, it is created and updated by the middleware maintenance agent to allow CAMidO to interact with new sensors.

Figure 2 describes a bandwidth sensor used by the e-commerce application. The sensor class name and package are described in line 2, and line 3 respectively.

```xml
<Sensor rdf:ID="bandwidthSensor">
  <SensorName>bandwidthSensor</SensorName>
  <SensorReturnedType>Integer</SensorReturnedType>
</Sensor>
```

Figure 2. Location sensor description
The second ontology is associated with the context level, it gathers information about the context to which all context-aware applications may be sensitive. It contains two types of data, direct data and indirect ones. The direct data represents information acquired by the mean of physical or software sensors, these sensors are described in the middleware level of the ontology. The designer has to specify for each direct context, the sensor from which data are collected. The indirect data represents high level of context information deduced from both direct and indirect data. The application designer has to describe how each indirect context is deduced from other context types in the application level of the ontology.

Figure 3 shows the description of the bandwidth and the networkConnexion contexts to which the e-commerce application is sensitive. Bandwidth context is a direct context captured from the bandwidthSensor sensor, while the networkConnexion context is an indirect context deduced using the ConnexionState interpretation method described in Figure 4.

The third ontology is associated with the application level, it contains application specific information. This information is used by the CAMidO compiler (cf. Section 3) to generate the appropriate source code to manage context and apply adaptation policies. This ontology is divided into four classes: The Component class, the CAService class, the RelevantContext class and the Policy&Rule class. These classes enable the description of the following entities.

- Context interpretation rules for indirect context description. These rules are described by creating instances of the HowDeduce property, this property enables the designer to describe the method used to interpret an indirect context using other direct and indirect contexts. This method consists either in calling a java method using reflexion mechanism, or in using a logic comparison between direct or other indirect context values. Figure 4 illustrates the interpretation method used by CAMidO to deduce the ConnexionState context of the e-commerce application. As described in lines 3, 4, and 5, the networkConnexion is deduced by comparing the bandwidth context value with a context value equals to 1.

- Relevant contexts to which the application is sensitive, by creating instances of the RelevantContext class. A relevant context is a simple or a complex combination of different context information values which necessitates application reaction when these values are detected. Figure 5 describes a relevant context to which the LV component of the e-commerce application is sensitive.

- Context-aware components belonging to the application and the relevant context they are sensitive to, by creating instances of the Component class, and binding them with the described relevant context using the AwareOf property, as shown in Figure 6 for the LV component.

- Application adaptation reactions when a relevant context is detected, by describing adaptation methods as instances of the Policy&Rule class and binding them with the appropriate relevant context. The e-commerce adaptation policies description are given in Section 5.1.

Information described in this meta-model is used by the CAMidO middleware for several purposes. It enables it to interact with sensors in order to collect context information, to manage the collected data, to analyse and to iterate it in order to detect any relevant context changes and adapt the application accordingly. All these tasks are automated thanks to
to the compiler included in CAMidO. Indeed, this compiler generates the appropriate source code to deal with context management and application adaptation from the information described in the CAMidO meta-model.

3 The CAMidO Compiler

Application specific context modeling is the first step in the process of context-aware application creation. As illustrated in Figure 7, the described data are used by the CAMidO compiler to generate both inference rule files and controllers source code, this figure shows development steps of a context-aware application on top of the CCM platform to which CAMidO context awareness facilities have been added.

CAMidO generates rules written in the first order logic. Some of these rules are deduced from existing relations between context information, the other ones are first order logic representation of the designer rules description. Those rules are used by the CAMidO entities described in Section 4, to interpret high level context and filter collected and interpreted data to detect relevant changes.

Figure 8 describes a generated rule from the e-commerce model. This rule is made of two parts, the conditions part and the action part. The action NewIndirectContextVal is a built-in class implemented on the CAMidO platform. This action consists in updating the ontology with the new interpreted context value.

For each context-aware component, the CAMidO compiler generates adaptation source code associated to the controllers described in Section 5.2. These controllers are added as hooks to the container architecture to manage component adaptation.

Two execution modes are provided by CAMidO the adaptation mode and the reconfiguration mode. The adaptation mode enables the middleware to consider context awareness and application adaptation, while the reconfiguration mode allows it to manage context awareness, application adaptation, and its reconfiguration at runtime. The difference between these two modes consists in the frequency of the ontology access. In the first mode, the ontology is accessed during the compilation step only, while in the reconfiguration mode the ontology is accessed also during the application runtime. So the designer can update relevant context to which each component is sensitive, adaptation policies and interpretation rules at runtime.

In the reconfiguration mode, the CAMidO compiler does not generate inference rules, it rather generates generic controllers which accesses the ontology to apply the appropriate adaptation. Inference rules are created by the CAMidO entities (ContextAnalyser and ContextInterpreter) at runtime rather than at compilation time as in the adaptation mode.

All descriptions given in the continuation of this paper concern the adaptation mode, the reconfiguration mode is out of this paper scope.

4 CAMidO Architecture

Component-based middleware facilitates the development of distributed applications, this is achieved by separating functional code from non-functional one. Functional code of an application is handled by components, while non-functional one is handled by containers. Containers act as interfaces between components and middleware, and manage all communications and calls to middleware services.

Mechanisms to deal with context and adaptation is still missing in component based middleware. So, creating context-aware applications on top of those middleware is painful. The application developer has to code the interaction of the application with sensors for context collection, context interpretation to deduce high level of context information, context analyses to detect relevant ones to which the application is sensitive, and component adaptation when a relevant context is detected.

The CAMidO middleware is an extension of a component-based middleware to which context-awareness
facilities have been added. As every existing context-aware middleware, CAMidO enables the interaction with the underlying execution environment to communicate with sensors and collect context information, the interpretation and analyses of those information for relevant context detection, and application adaptation. But among the existing context-aware middleware, CAMidO architecture distinguishes itself by the ontology meta-model it contains. The CAMidO meta-model enables the middleware to activate and deactivate sensors during application runtime according to their needs thanks to reflection mechanisms. It enables the middleware to reason about context and to consider all new adaptation rules added during the application execution.

As shown in Figure 9, CAMidO is made up of six blocks, which work together to achieve context management. During the application deployment, each sensitive component, subscribes to the ContextAnalysers. This registration is made by the associated ComponentAdapter. ContextAnalysers is responsible for filtering collected contexts. Context values are captured by the CollectionManager from sensors. The CollectionManager uses the first and the second level of the ontology to activate agents for context collection.

After application deployment, the following process is executed: (1) each agent collects context information, (2) the CollectionManager sends the collected values to the ContextAnalysers and the ContextInterpreters, (3) the ContextInterpreters deduces high level context information using the InferenceComponent and interpretation rules (cf. Section 2, Section 3), it transfers interpreted data to the ContextAnalysers, (4) the ContextAnalysers filters sensed and interpreted data to detect relevant context changes thanks to the inferenceComponent. If necessary it notifies the subscribed component. New context values are saved in the ContextRepository.

The described entities use the ontology in order to manage context. The CollectionManager uses the first and the second level of the ontology to activate agents for context collection, while the ContextAnalysers, the ContextInterpreters and the InferenceComponent use the second and the third level of the ontology to interpret high level contexts and filter them to detect relevant changes. The InferenceComponent uses also rule files generated by the CAMidOCompiler in order to reason about context, interpret it, and filter it. Adaptation is carried out by each component container according to the process described in Section 5.2 using the third level of the ontology.

5 Adaptation Mechanisms in CAMidO

Adding context management services to a component-based middleware for context collection, interpretation, and analyses is not sufficient to facilitate context-aware application creation. CAMidO has also the role of managing application adaptation in order to relieve the developer from this painful task.

In this section, we define the adaptation types handled by CAMidO, then we describe the adaptation management mechanism and the container architecture.

5.1 Adaptation Types in CAMidO

Adaptation mechanism is the core of any context-aware middleware, it includes all strategies and policies used by the middleware in order to react to context changes.

Adaptation policies allow application designers to describe how the application has to react when relevant context changes are detected. In CAMidO, each context-aware component has associated relevant contexts and adaptation policies. These policies are described by the application designer using the ontology application level (cf. Section 2). This description is used by the CAMidO compiler to generate the appropriate controllers to adapt this component. An example of the e-commerce adaptation policies description is given at the end of this section.

CAMidO manages two adaptation types, the reactive adaptation and the proactive adaptation.

The reactive adaptation, consists in triggering the appropriate adaptation operation when a relevant context is detected. The associated operation defined in the ontology (cf. Section 2) is activated whatever the application is performing at that moment. This operation can be provided by the component invoking the request, by another one, or implemented in an external package.
The proactive adaptation concerns component invocation behaviour. It consists in switching from one invoked operation to another one. Proactivity consists in anticipating this redirection by preparing a switch request when a relevant context is detected, and associating it to the appropriate operation. This will have an effect only when the operation is invoked. Two kinds of proactive adaptation are considered in CAMidO: the client-side adaptation managed by the Client Request Adaptor (CRA), aims to redirect the client operation request to another operation (having the same parameters) provided by the same component or by another one as illustrated in Figure 10, it could be from O1 to O2 or from O1 to O3. While server-side adaptation managed by the Server Request Adaptor (CRA), as illustrated in Figure 11, enables a component to provide different operation in term of behaviour and name, according to context values.

The adaptation types, described above, are managed by controllers added to the component containers. Their behaviour is driven by the context properties defined in the ontology.

In the e-commerce application illustrated in Figure 12, the UI component can ask the DB component for a department, this latter provides goods textual information if low bandwidth is detected, or all the goods’ description (text, image and video) otherwise. If there is no connexion between these components, all the UI requests are redirected by the Client Request Adaptor (CRA) to the LV component. When the connexion is restored, the LV component forward all the non treated requests to the DB component using the reactive adaptation described in Figure 13. This adaptation consists in invoking the forwardRequest action provided by the LV component. As described in line 7 and 8 of this figure, the forwardRequest action does not necessitate any parameters, and does not return any value.

The UI component is sensitive about connexion mode. As illustrated in Figure 12, and described in Figure 14, when the UI component asks for a request and there is no network connexion between this component and the DB, this request is redirected to the LV component.

As shown in Figure 15, the CAMidO compiler gener-
ates an adaptation rule associated to the adaptation description shown in Figure 13. CAMidO manages this rule as an Event/Condition/Action, the event is implemented on the CAMidO platform (ContextAnalyser), it consists on a new context value detection. When this event happens the condition which represents a relevant context evaluation, is evaluated by the InferenceComponent. The Notification action is called if the relevant context is detected. It consists in notifying the subscribed component for this relevant context.

\[\text{RelevantContext0:} (?x0 \text{http://etna.int-evry.fr/\{\text{belhanaf/ContextAwareness\#networkConnexion\}val \?x1 \text{equal}(\?x1, \text{\textquote{\textquote{Connected}}})}) \text{\textarrow{\rightarrow}Notification(react, relevantConnexion)}\]

**Figure 15.** An adaptation rule generated by the CAMidO compiler

5.2 Adaptation Management and Container Architecture

In component-based middleware, non functional properties are managed by the component container, it controls component interactions through entities called controllers. We benefit from the component/container paradigm [19] in order to add adaptation management as non-functional properties. Then we add controllers to the container architecture for this purpose.

**Figure 16.** CAMidO Container Architecture

As illustrated in Figure 16, several controllers have been added to the container architecture: The AdaptationDetector (AD), the ReactivAdaptor and the ProactivAdaptor. The AdaptationDetector is used to subscribe to the ContextAnalyser for relevant contexts in order to be notified when a relevant context change occurs, while the ReactivAdaptor and the ProactivAdaptor allows the container to apply the appropriate adaptation rules when the AdaptationDetector is notified for relevant context detection.

The ReactivAdaptor controller applies reactive adaptation when it is notified by the AD. The ProactivAdaptor controller prepares proactive adaptation requests when it is notified by the AD, it is used also to add pre-processing to component invocations.

For component invocation interception, we use Portable Interceptors [16] in order to be independent of any platform. As illustrated in Figure 16, the pre-request controllers CAClientInterceptor and CAServerInterceptor, are used to intercept component invocations. The CAClientInterceptor intercepts the component outgoing requests and uses the CAMidOProxy in order to transfer those requests to the ProactivAdaptor in order to add the appropriate pre-processing. The pre-processing consists of redirecting an operation request to another operation if necessary. The CAServerInterceptor intercepts the component incoming requests and redirects them to the ProactivAdaptor in order to add the appropriate pre-processing too.

At the component deployment step, each AD subscribes to the ContextAnalyser for all its associated component relevant contexts. During application runtime, the ContextAnalyser filters and analyses the collected data, when a relevant context is detected, the appropriate AD is notified. If this notification concerns a reactive adaptation, an invocation request is created and triggered by the ReactivAdaptor using the DynamicRequestCreator controller. Otherwise, if it concerns a client or a server side proactive adaptation, a request for adapting the incoming or outgoing calls of the service to be adapted is created and saved. When the Portable Interceptors intercept component requests, the call is forwarded to the ProactivAdaptor controller which checks if an adaptation request associated with the invoked operation exists. If an adaptation is needed, the ProactivAdaptor provides the appropriate operation instead of the invoked one if it concerns a server side adaptation, or it changes or forwards the request to the appropriate component if it concerns a client side adaptation, as described in the adaptation policy.

In order to access to the intercepted component request information, the CAClientInterceptor implements the sendRequest method and adds the appropriate treatment which consists in redirecting the request to the ProactivAdaptor. Redirecting a request to another target is done by throwing a CORBA LOCATION_FORWARD exception. In CORBA, request description is accessible via the ClientRequestInfo object [3]. This object implements the RequestInfo interface and stores information about client side request parameters, but accessing the attribute of these interfaces is a strictly platform dependent issue [12]. As proposed in [1], we used a proxy server called CAMidOProxy
to overcome this kind of platform limitations. This proxy is able to read the request content and call the appropriate proactivAdaptor to check if an adaptation is needed.

In the case of a server-side proactive adaptation, the CAServerInterceptor implements the receive-request method in order to add the appropriate treatment. This treatment consists in changing the name of the operation field in the intercepted request by the name of the service needed for this adaptation.

The adaptation mechanisms added to the container architecture use information provided by the designer descriptions according to the CAMidO ontological meta-model. Indeed, designer description is used during the application compilation by the CAMidO compiler to generate container controllers (AdaptationDetector, ReactivAdaptor, ProactivAdaptor).

6 Implementation and Evaluation

A CAMidO prototype has been developed on top of the OpenCCM platform [14]. We have chosen OpenCCM because it is a component-based middleware which manages non functional properties using containers.

The Ontological meta-model provided by CAMidO is described using the OWL language [2], it is an ontology based language for defining and instantiating web ontologies. The size of ontology data set is measured in the number of the OWL classes without considering properties. The middleware ontology level is made up of six classes, the context ontology level is made up of six classes whereas the application ontology level contains forty nine classes.


The context collection, interpretation and treatment mechanisms are written with the Java Language. The Jena toolkit was used to execute the inference engine for relevant context detection and interpretation.

We have evaluated the performance and the cost generated by the context management without considering the proactive adaptation. The experimentations have been conducted on windows 2000 laptop powered by 1,3 GHz with 256 Mo of RAM. At first we measured the impact of the ontology RDF triplets number and the collected context number for context analyses and interpretation performance. Figure 17 and 18 illustrate that the runtime performance depends on the ontology RDF triplets number. Indeed the InferenceComponent used by both the ContextAnalyser and ContextInterpreter parses all the ontology in order to interpret data and to check for relevant context values. Figure 19 shows the cost generated by context management by measuring the number of a component treated requests. In one second, the component installed on top of CAMidO has treated 234 requests whereas the same component has treated 344 requests when it is installed directly on top of OpenCCM. This cost is explained by both the reactive adaptation actions which occurred during the application execution, and the computation needed by the contextAnalyser and the contextInterpreter in order to interpret context and analyse the collected data. In order to reduce the impact of the interpretation and analyses cost for component request treatment, it is more efficient to distribute these entities and to install them in a remote host.

![Figure 17. Analyses performance in CAMidO](image1)

![Figure 18. Interpretation performance in CAMidO](image2)

![Figure 19. Runtime performance using CAMidO](image3)
7 Related Work

There is substantial research works on context-aware middleware, but few of them consider component based middleware. This section compares some context-aware middleware solutions with the CAMidO middleware.

The RCSM middleware [22] is an object-based middleware which enables development of context-aware applications. As CAMidO, it provides a language for context description and uses it to generate adaptation source code. The CA-IDL language used by RCSM enables an application designer to describe context to which its application is sensitive, but unlike CAMidO, sensors from which data are collected can not be chosen nor modified, because it is fixed by the CA-IDL grammar. CAMidO differs from the RCSM middleware in the facts that it interpret high level context, it manages reactive and proactive adaptations, and it uses an ontology which enables CAMidO to reason about context using an inference component.

The CARISMA middleware [4] [5] is a reflexive middleware for context-aware application management. The context awareness mechanisms used in CARISMA and CAMidO are quite different. CARISMA uses a reflexive mechanism to apply adaptation while CAMidO uses an adaptation contract and portable interceptors for this purpose. The CARISMA middleware enables application designer to describe context information in name/value pairs using the eXtended Markup Language, without giving it the possibility to specify from which sensors contexts have to be collected, nor to interpret high level context.

The SOCAM middleware [9] aims to enable rapid prototyping of context-aware services using an ontology. As CAMidO, this middleware takes into account context acquisition and interpretation, it offers APIs for context subscription, but there is no mean to describe relevant context nor adaptation policies, so context analyzes and services adaptation have to be developed by the application developer.

K-component [7] is a component based middleware for context-aware application management. It provides the ACDL language (Adaptation Contracts Declarative Language) for application adaptation description. Unlike CAMidO, the K-component middleware does not provide sensors description, the application designer has to develop the context collection in an ad hoc way. When a relevant context is detected, the adaptation is carried out by a configuration manager, this adaptation consists in changing the application architecture using the architectural reflexion [6].

CAMidO belongs to the category of the extended middleware described in [17]. As QuO [18], CIAO [21], and Cygnus [10] which use portable interceptor to manage respectively quality-of-service and load balancing, CAMidO employs CORBA portable interceptors [16] to intercept component requests, and apply the appropriate adaptation to manage context-aware applications.

As illustrated in Table 1, CAMidO distinguishes itself from existing context-aware middleware by the use of both an antological meta-model for context description, context-awareness management using inference mechanism to analyse context, and component/container paradigm to manage application adaptation.

8 Conclusion

Context-awareness helps applications running on mobile devices to respond to changeable environmental conditions. An efficient context representation and a context-aware middleware infrastructure are both necessary to help developers to create such types of applications.

In this paper, we have described CAMidO, a component-based context-aware middleware which provides an ontological meta-model for context description. All described data are used by the CAMidO compiler in order to generate inference rule files, and adaptation source code for application adaptation management.

The generated source codes are added as hooks to the container architecture, they manage component adaptation using portable interceptors to redirect all component invocations.

The described interpretation and adaptation rules, and the generated ones, using the CAMidO mechanisms, can generate ambiguities and contradictions. Indeed, many kinds of conflicts may appear, such as intra- and inter-component conflicts.

The intra-component conflict can appear during the client-side adaptation process, when the same operation has to be switched by more than one other operation. While inter-component conflicts can appear during the server-side adaptation, when a component has to provide more than one adaptation operation to a requested one. Some of these conflicts may be detected during the ontology compilation step, however a mechanism for conflict resolution has to be added to the CAMidO middleware.

Both the inter- and intra-component conflicts can be resolved by adding a priority to each operation. If a conflict occurs, the operation with the highest priority is invoked. However other types of conflicts which necessitate more complex computation, such as intra-application conflict, may appear. Intra-application conflict consists in the presence of contradictory adaptation actions between two applications, at least, running on CAMidO.

As a next step we intend to look further into these conflict types in order to propose a consensus mechanism to resolve them. We also intend to study the distribution of the CAMidO entities and the ping-pong effect consequences which consists in a very frequent changes of context values.
<table>
<thead>
<tr>
<th>Middleware</th>
<th>Type</th>
<th>Context Modeling</th>
<th>Interpretation Management</th>
<th>Analyses Management</th>
<th>Adaptation Management</th>
<th>Sensor description</th>
<th>code generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCSM</td>
<td>Object based</td>
<td>CA-IDL</td>
<td>No</td>
<td>Middleware</td>
<td>Object container</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CARISMA</td>
<td>Reflexive</td>
<td>XML</td>
<td>No</td>
<td>Middleware</td>
<td>Middleware (reflection)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SOCAM</td>
<td>Service based</td>
<td>Ontology</td>
<td>Inference engine</td>
<td>Developer</td>
<td>Developer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>K-Component</td>
<td>Component based</td>
<td>ACDL</td>
<td>No</td>
<td>No</td>
<td>Middleware</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CAMidO</td>
<td>Component based</td>
<td>Ontology</td>
<td>Inference engine</td>
<td>Inference engine</td>
<td>Component container</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Context-Aware Middleware Comparative table

References