Towards a Model-Driven Approach for Ontology-Based Context-Aware
Application Development: A Case Study

Nektarios Georgalas\textsuperscript{1}, Shumao Ou\textsuperscript{1,2}, Manooch Azmoodeh\textsuperscript{1} and Kun Yang\textsuperscript{2}

\textsuperscript{1}BT Group, UK
\{nektarios.georgalas, shumao.ou, manooch.azmoodeh\}@bt.com,
\textsuperscript{2}University of Essex, UK
\{smou, kunyang\}@essex.ac.uk

Abstract

Context-awareness is an essential characteristic for pervasive applications to enhance their flexibility and adaptability to changing conditions and dynamic environments. Using ontologies to model context information and reason about context at a semantic level has attracted a lot of interest in the research community. However, adopting ontologies in the development process of pervasive services introduces additional burden by the further work required for ontology specification and management. This paper studies the use of OMG’s Model Driven Architecture in the development process of pervasive services with intent on reducing this burden. In our previous work, we proposed a Context Ontology Model (COM) and a Model Driven Integration Architecture (MDIA) for ontology-based context information modelling and Context-Aware Application (CAA) development. In this paper, we present a model transformation mechanism for the generation of the CAA logic and present a case study on a context-aware pervasive service scenario that explains in deep detail how the proposed approach works in practice.

1. Introduction

In order to flexibly adapt to changing conditions and dynamic environments, pervasive services need to become more aware of the context they operate in. This setting raises a number of challenging issues, such as representation and semantic reasoning of context information in pervasive computing environments. Earlier research work focused on context information gathering and integration aiming to achieve reusability for higher level pervasive applications [2] [1]. Other work studied the modelling of information from variant context sources in a platform independent way in order to support context management and interoperation [9] [3]. More recently, ontology has emerged as a new approach to context modelling. Ontologies can model context at a semantic level establishing a common understanding of terms and meaning and enabling context sharing, reasoning and reusing in pervasive computing environments [9] [10] [11] [15].

Languages such as OWL [13] can be used to specify ontologies in a machine-interpretable way. An ontology includes definitions of commonly understood vocabularies and of logic statements that specify what each term in the vocabularies mean and how they relate to each other. An ontology is semantically independent to reader and context. It is, therefore, useful in bridging terminology differences and thus enhances interoperability. The concepts and logic expressed by ontology are commonly accepted and can be communicated between human users and computer programs from different vendors. These features make ontology just the right mechanism for modelling context information in support of Context-Aware Application (CAA) development for pervasive computing environments tackling the issue of heterogeneity introduced by diverse device technology, multiplicity of vendors developing CAAs and variant operating systems CAAs run on.

The use of ontologies to model context augments the development process of pervasive services with additional burden introduced by the work required for ontology specification and management. Therefore, to make the use of ontologies practically viable, development approaches need to be applied capable of reducing this burden. These cannot be ad-hoc and proprietary approaches but rather ones that allow for rigorous/precise modelling of context ontology and for rapid development of ontology-based context-aware applications. To this end, we have been investigating the use of Model Driven Architecture (MDA) [12], the emerging standard by the Object Management Group (OMG) for software systems design and development in order to evaluate benefits this approach can contribute to this area.
MDA aims at providing clear separation between technology-neutral and technology-specific concerns involved in the different stages of a system’s development process. This is achieved by separating specification artefacts at different levels of abstraction with a clear distinction between models independent of technical detail, namely, Platform Independent Models (PIMs) and models that include detail of the implementation technology, namely, Platform Specific Models (PSMs). MDA consists of a set of standards, namely, MOF, OCL, XMI and QVT that enable the definition of Domain Specific Languages (DSLs) used to specify a system’s structure and behaviour. DSLs are represented as meta-models based on the Meta-Object Facility (MOF) and can be precisely defined using the Object Constraint Language (OCL). OCL allows the definition of constraints over meta-models as well as actual models for a specific system. With sequential transformations by defining mappings among various DSLs, using the Query View Transform (QVT) standard, implementations of the system can be produced for particular platforms. MOF expressed model data can be freely transferred between compliant tools using OMG’s XML Metadata Interchange (XMI) technology.

We have practically applied MDA in a number of case-studies that demonstrated the advantages the approach offers in the development process of systems and services [3] [4] [5]. In [3] and [4] we discussed the use of MDA for context-aware pervasive service modelling, provisioning and composition. In [5] we presented how MDA was used for the design, development and integration of telecommunications Operations Support Systems (OSS) and the benefits gained in terms of improved quality and lower development costs.

The above experiences lead to the conclusion that MDA can be a beneficial paradigm for capturing context ontologies as a number of advantages are to be gained. Modelling ontologies as PIMs can be a one-off activity as these PIMs (models of roles, devices and tasks) can be re-used in the development of other CAAs. Heterogeneity is also catered for since ontology and CAA PIMs can be transformed into implementations suitable for the platforms and devices at hand. MDA can facilitate the rapid generation of ontology-based CAAs with significant reduction in development time and costs.

Through a detailed case study on a pervasive application scenario, this paper presents a model driven approach for CAA development. The core of the approach is a model driven integration architecture (MDIA). MDIA integrates DSLs for application development. Inside the MDIA, a context ontology model (COM), consisting of ULCOM and ESCOM, is utilized to model ontology-based context information.

The rest of the paper is organized as follows. The next section discusses some related work. Section 3 describes our approach comprising the context ontology model, MDIA and the integrated model transformation mechanism. In Section 4, we present the SmartSpeaker case study which shows in detail the application of the approach in practice. Finally, Section 5 discusses lessons learned and future work.

2. Related Work

Related research has dealt with the issue of ontology-based context modelling and reasoning in a number of perspectives. Wang et al proposed an OWL-encoded ontology (CONON) for modelling and reasoning about context in pervasive computing environments [9]. Chen et al proposed an architecture called Context Broker Architecture (CoBra) that uses OWL to define ontology in intelligent environments [10]. Furthermore, they proposed a Standard Ontology for Ubiquitous and Pervasive Computing Applications (SOUPA) [15]. Henricksen et al proposed a hybrid approach for context modelling, reasoning and interoperation between object-oriented context models and ontology-based context models [11]. All the above referenced research illustrated some advantages of handling context at a semantic level by using different solutions. However, no evidence was found of any solutions trying to model ontology in the context of MDA.

Other research work focuses on ontology-based CAA development. Biegel and Cahill proposed a framework to develop CAAs based on their sentient object model. They focus on fusing data from disparate sensors to ease context-aware application development by simple coding [16]. McFadden et al proposed a model driven approach to develop CAA based on their object-oriented Context Modelling Language (CML) [17]. These practices are aiming to reduce the development effort or to automate the CAA development process using, though, rather specific or proprietary mechanisms.

In our work, a pure MDA-based approach is applied for context ontology modelling that is based on well-recognised OMG standards, such as MOF, OCL and XMI and on OMG’s newer effort regarding ontology modelling, the Ontology Definition Meta-Model (ODM) [8]. Based on the ontology models and other domain specific models, we propose a model driven integration architecture for automatic CAA development.
3. Approach: COM and MDIA

This section presents our approach for ontology-based CAA development. We firstly describe our ontology modelling architecture which shows how ontologies are captured using the four abstract layers adopted by MDA. We, then, describe the Context Ontology Model (COM) which consists of the Upper-Level Context Ontology Model (ULCOM) and the Extended Specific Context Ontology Model (ESCOM) and is used to model context information. Finally, we present MDIA and the integrated model transformation mechanism. For our modelling work, we used an MDA toolkit, namely, XMF [7] by XACTIUM.

3.1. Ontology Modelling Architecture

MDA is based on four layers of abstraction, M0 through M3. M0 contains application run-time data; M1 contains application models designed for a specific problem/application domain; M2 contains meta-models that capture DSLs used in the application designs of M1; M3 hosts the Meta-Object Facility (MOF), which is a language to specify DSLs.

![Fig. 1 Ontology Modelling in MDA Four-layer Architecture](image)

Fig. 1 shows how the ontology models and meta-models are positioned around the above four layers. M2 hosts the MOF-based Ontology Definition Meta-model (ODM) and the UML profile for Ontology. Domain Ontology Models are situated on M1 and are instances of ODM representing models of domain-specific ontologies. An example of a domain ontology model is the Context Ontology Model (COM) introduced in next section. M0 contains models that are instances of M1 domain-specific ontologies.

3.2. Context Ontology Model (COM)

An ontology of context represents knowledge about the context domain and comprises definitions of a set of context entities, the entity attributes, the functions the entities provide, the relationships between context entities, the instances of context entities and the axioms used for context reasoning.

We have defined COM that describes context for pervasive services. COM consists of two parts, namely, the Upper-Level Context Ontology Model (ULCOM) and the Extended Specific Context Ontology Model (ESCOM). This two-layer ontology modelling is inspired from [9]. ULCOM is static, i.e., its entities are not expected to change as frequently. It is a high-level ontology which captures general features of basic contextual entities, thus, we represent it as a meta-model. ESCOM is more dynamic because it can be extended with more entities as they are discovered in different contexts. It is a collection of ontology terms which define the details of general concepts and their features in each problem domain, thus, we represent it as an instance of ULCOM on M1 level. This is an improvement introduced to the way COM was presented in our previous work, where readers can find more information about COM (ESCOM and ULCOM) contents [6].

3.3. Model Driven Integration Architecture (MDIA)

This section presents our MDA-based approach for Context-Aware Application (CAA) development. Context ontology alone is useful but not sufficient to entirely support CAA development as it only captures knowledge about the CAA context. For CAA it is necessary to further specify models describing the application logic, the graphical user interfaces (GUI), the application data and the way the CAA integrates with other systems and services. Therefore, alongside COM, more meta-models have been developed to facilitate the rapid generation of CAAs.

Fig. 2 gives an overview of our Model Driven Integration Architecture (MDIA) for CAA development. At the meta-model layer there are three categories of artefacts: CAA integration related meta-models, implementation languages meta-models and mappings between the meta-models.

The CAA integration related meta-models category includes the following six packages:

- **ComponentMetaModel** defines a language to model functional interfaces of existing functional components (such as ontology
reasoning components in our application domain, or inventory components in OSS systems [5]). Using this language we can model at the M1 level ontology handling functionality of Commercial-Off-The-Shelf’ (COTS) components (or libraries) which we can then integrate into the models of CAAs.

- **ProcessMetaModel** represents a language that can be used on M1 to specify application logic in the form of a process. The meta-model defines elements of a UML activity diagram.

- **ULCOM** is used to define context ontology data in our architecture. The ULCOM model is able to be mapped into OWL metamodels and finally be transformed to XML code.

- **GUIMetaModel** defines basic elements of a language to describe a graphical user interface, such as window, label and textbox and an event-based model describing the dynamic way GUI elements can trigger logic associated with them.

- **DataMetaModel** describes a language for the specification of application-related data on M1. This meta-model is based on the UML class diagram.

- **IntegrationMetaModel** is key as it defines the way all previous meta-models associate and integrate. It serves as the glue that brings all necessary elements together in order to compose a CAA. More specifically, this meta-model defines how (1) a flow of process activities integrates different components by invoking certain operations on each component to deliver an activity; (2) GUIs integrate with processes by events GUI elements generate and trigger process activities or entire processes representing the logic behind these elements; (3) data integrates with both components and processes that consume and produce information of different types.

Each of the above CAA integration related meta-models generates a specific profile, i.e. a set of precisely defined modelling artefacts (stereotypes), which can be used for the technology-neutral specification of CAAs at the M1 layer. In Section 4.3, a number of these stereotypes are presented and used in the context of the case study.

In order to enable the generation of technology-specific CAA implementations, it is important to introduce another category of meta-models, namely, implementation languages meta-models. In this category, just for the purpose of our case study, we defined **J2MEMetaModel**, **CSharpMetaModel** and **XMLMetaModel**, which constitute specifications of the respective languages’ syntax, including grammars, expressions, statements and programming structures (classes, operations, variables etc).

What is still missing before MDIA is completely enabled to automatically generate CAA implementations is specifying precise transformations of technology-neutral to technology-specific meta-models. More specifically, we define two types of mappings in the architecture:
• Mappings between integration and implementation language meta-models, namely, IntegrationToJ2ME, Integration2CSharp, and Integration2XML. These are used to generate CAA implementations.
• Mappings between ULCOM to ontology language OWLMetaModel and further to implementation language meta-models, such as XMLMetaModel. These are used to generate technology-specific representations of ontological artefacts in the specified implementation languages.

3.4. Integrated Model Transformation

This sub-section discusses the integrated model transformation mechanism which specifies precise transformations of multiple technology-neutral meta-models into a single technology-specific meta-model range.

Fig. 3 illustrates the steps of the integrated model transformation mechanism for transforming between a group of CAA related metamodels and J2MEModel.

Step 1 generates a Model/View/Controller (M/V/C) application framework.

Step 2 transforms data element into platform-specific representation, i.e., Java format in this case. This step initializes the Model part of the M/V/C structure by declaring and initializing the data elements.

Step 3 deals with the View of the CAA. All the GUI components are defined and initialized.

Step 4 generates the Controller by transforming application processes. The system firstly looks for the start node in a process. It then follows the control flow to examine all defined actions. An action will be examined by listing its input/output data items. If an end node is encountered, the process is ended.

Step 5 is the most important. It generates the integration logic by checking <<IsDeliveredBy>> and <<KicksOff>> stereotyped connections which relate, respectively, process actions to component operations and GUI event-triggered actions to process actions. Navigating through the models over these connections, this step brings together the component, process and GUI elements in order to eventually produce executable output.

The system will loop between Step 4 and 5 until all defined integration models are examined.

4. Case Study: SmartSpeaker

In this section we present a case-study aiming to practically demonstrate how MDIA enables the automatic generation of CAs for different mobile devices. The CAA at hand is SmartSpeaker, a simple context-aware pervasive application. Applying MDIA, two SmartSpeaker implementations are generated aimed for deployment on a PDA and on a smartphone.

4.1. Scenario

George is in his lounge listening to a live digitally broadcast radio program on a mobile device (such as a PDA or a smartphone). He moves to his garden while he continues listening to the program. Due to the noisy environment of the garden, the device adjusts the volume automatically to an appropriate audible level. To achieve this, the device runs a CAA, namely, SmartSpeaker, which senses the change of George’s location and smartly turns up the speaker volume. When George returns to his lounge, SmartSpeaker again adjusts the volume automatically back to the previous lower level.
4.2. Solution to the Scenario

Fig. 4 illustrates the architecture of SmartSpeaker. There are three independent components deployed on the mobile device, namely, LocationSensor, SmartVolumeController, and OSMediaController. Each component supports one or more capabilities, i.e., interfaces, which contain specific operations. In particular, the LocationSensor understands the current location of the device and through its LocationSensing interface can pass this information to a requester. The SmartVolumeController is responsible for adjusting the volume of a media controller based on default levels per location pre-configured by the user. It provides the SVCConfiguration interface with operations to configure/maintain default location-volume pairs and the SVCTuner interface which passes back to a requester the default volume level for a particular location. Finally, the OSMediaController is the media controller provided by the device’s operating system, and through its MCAudioTuner interface one can adjust its volume to an indicated level.

The SmartSpeaker CAA orchestrates data flows and interactions among the above components so that the OSMediaController’s volume is automatically adjusted to pre-configured levels when the mobile device changes location. SmartSpeaker: (1) gets the current location from LocationSensor; (2) gets from SmartVolumeController the pre-configured volume level that maps to the current location; (3) changes the OSMediaController volume to the pre-configured level. Additionally, SmartSpeaker provides to the device owner facilities for configuring SmartVolumeController with new location/volume level pairs or for changing the volume level of existing locations. SmartSpeaker is controlled through a GUI.

4.3. M1 Models

This section provides an analytical account of the modelling pursued towards the implementation of SmartSpeaker. More specifically, we illustrate GUI, Application Logic, Component and Data models and their integration making full and exclusive use of the MDIA meta-models. These models are platform independent and are built using stereotypes representing meta-model entities and associations defined in the MDIA packages of section 3.3.

Modelling of Components

The LocationSensor, SmartVoiceController and OSMediaController components used by SmartSpeaker are modelled using stereotypes of the MDIA Component Metamodel profile.

Modelling of Application Logic

SmartSpeaker’s application logic is mainly responsible for managing interactions and data flows among the aforementioned components. Therefore, we model the application logic as a process that consists of a series of actions using stereotypes of the MDIA Process Metamodel profile.

The process, shown in Fig. 8, begins at a start node followed by <<Action>> nodes that represent actions GetLocationFromSensor, AdjustVolumeOnSVC and TuneVolumeOnMC and finishes at a stop node. Execution of these actions triggers invocation of certain component operations,

1 Each stereotype is included in <<...>>

2 Operations GetVolumeByLocation and AdjustVolumeLevel although identical from an implementation viewpoint, they are conceptually different in their use. The former operation is used by GUI elements in order to show the correspondent volume level for a selected location. The latter operation is used to retrieve the volume level corresponding to a current location detected by LocationSensor so that SmartSpeaker can appropriately adjust OSMediaController’s volume.
Fig. 5 Model of LocationSensor

Fig. 7 Model of OSMediaController

Fig. 6 Model of SmartVolumeController
which eventually deliver the result described in section 4.2 and Fig. 4.

Through its GUI, SmartSpeaker also facilitates configuration of SmartVolumeController with new location/volume level pairs (process NewLocationVolumePairConfiguration), change of already set volume levels per location (process VolumeLevelOnLocationChange) and retrieval of SmartVolumeController’s current configuration, i.e. the already preset location/volume level pairs (process VolumeLevelByLocationRetrieval and RetrieveAllLocationsRetrieval). Apparently, each of these functionalities represents application logic that is modelled as a simple one-action process, as indicated in the brackets, using the MDIA ProcessMetaModel profile.

Modelling of Data

A common concern when modelling process and component is data that is input to or output from process actions and component operations. Fig. 9 shows the data types used by SmartSpeaker’s components (see Fig. 5, 6, 7) and application logic (see Fig. 8). These data types, except for allLocations, represent Context Ontology terms specified in the ESCOM package of MDIA’s M1 layer. Since ESCOM instantiates ULCOM, these ontology terms use stereotypes of the ULCOM profile (<<location>> and <<volumeLevel>> represent ULCOM terms). allLocations is not an ontology term, but rather a separate application-specific data item which, conceptually, belongs to the DataModel package of MDIA’s M1 layer.

Modelling of GUI

End-users interact with SmartSpeaker through an application GUI. The GUI is modelled using stereotypes of the MDIA GUIMetaModel profile. Fig. 10 shows part of the GUI model indicating the GUI’s structure, i.e. the GUI elements it consists of. More specifically: (1) a label and textbox are used to indicate the device’s current location and volume level respectively; (2) a label and textbox are used to define a new location and volume level pair that is then saved in SmartVoiceController by interacting with <<GUILabel>>Add; and (3) a label and textbox are used to present the default volume level.
for a selected location; this default volume level can be modified by interacting with <<GUIElement>> Change.

The MDIA GUI MetaModel specifies that GUIElement generates GUIEvent which triggers Logic, where Logic represents application logic modelled as an entire process or some action within a process. This pattern is applied here in order to show that some behaviour will be triggered when interacting with SmartSpeaker GUI elements Add and Change. In the next section we show how the exact behaviour to be triggered is specified.

### Modelling the Integration

In this step we show how processes, i.e. SmartSpeaker application logic models, associate with component operations and how the SmartSpeaker GUI Elements associate with the defined processes. To model this integration, stereotypes of the MDIA IntegrationMetaModel profile are used.

Fig. 11 shows how the three process actions of Fig. 8 integrate with component operations of Fig. 5, 6 and 7. For instance,

\[
\text{GetLocationFromSensor} \searrow \text{GetCurrentLocation} \searrow \text{GetCurrentLocationFromSensor}
\]

GetCurrentLocation of component LocationSensor.

Similarly, each SmartSpeaker GUI Element of Fig. 10 integrates with application logic in two steps. First, as part of the GUI modelling work we defined the type of event a GUI element generates and the type of behaviour this event triggers. Second, as part of the GUI-to-process integration work, we specify which process or process action, i.e. application logic model, the event triggered behaviour maps to. For instance, <<GUIElement>> Add: (1) <<generates>> <<GUIEvent>> OnClick which <<triggers>> <<Logic>> AddLogic and (2) <<Logic>> AddLogic <<kicks off>> NewLocationVolumePairConfiguration. The latter is a one-action process specifying the action node

\[
\text{SetNewLocationVolumePair} \searrow \text{SetNewLocationVolumePair}
\]

which integrates with SmartVolumeController operation Configure as follows:

\[
\text{SetNewLocationVolumePair} \searrow \text{SetNewLocationVolumePair}
\]

4.4. Ontology-Based Location Information

Different location sensors may use different formats to represent location. In our case study we used a Bluetooth and a GPS-based location sensor. The former represents location using a String that captures the location’s name, e.g. “Garden”. The latter uses co-ordinates (longitude and latitude) to represent a location. In order to integrate the different formats we use the “class mapping” technique presented in [18]. Class mapping facilitates interoperation between heterogeneous ontologies by defining how the semantics of entities of two or more ontologies map to each other. For the purpose of our case-study, we implement class mapping rules using xOCL, the version of OCL supported by XACTIUM’s XMF toolkit. Fig 12 assumes that MDIA ESCOM package contains two subpackages one for GPS and one for Bluetooth, both containing ontology entity <<Location>> Garden. According to the rule if GPS coordinates enter a particular value range this signifies that GPS Garden maps to Bluetooth Garden.

\[
\text{LocationMapping}
\]

SmartSpeaker::ESCOM::GPS::Garden[
latitude = lat,
longitude = long]
when
\[
lat >= 51.8815 && lat <= 51.8816
&& long >= 0.9334 && long <= 0.9335
\]
do
SmartSpeaker::ESCOM::Bluetooth::Garden[]
end

Fig. 12 Location Class Mapping
public class SmartSpeaker extends MIDlet {
    public SmartSpeaker() {
        super();
    }
    protected void startApp() throws MIDletStateChangeException {
        Display display = Display.getDisplay(this);
        Form SmartSpeakerMainFrame = new Form("SmartSpeaker");
        SmartSpeakerMainFrame.append(new StringItem(" Current Location",""));
        SmartSpeakerMainFrame.append(new StringItem(" Current Volume",""));
        display.setCurrent(SmartSpeakerMainFrame);
    }
}

Fig. 13 Generated Platform Specific Code

public class SmartSpeaker : System.Windows.Forms.Form {
    private System.Windows.Forms.Label CurrentLocation_label;
    private System.Windows.Forms.TextBox CurrentLocation_textBox;
    ...
    public SmartSpeaker() {
        InitializeComponent();
        ...
    }
    private void InitializeComponent() {
        this.CurrentLocation_label = new System.Windows.Forms.Label();
        this.CurrentLocation_textBox = new System.Windows.Forms.TextBox();
        ...
        this.CurrentLocation_label.Size = new System.Drawing.Size(128, 16);
        this.CurrentLocation_label.Text = "Current Location:");
        this.CurrentLocation_textBox.Size = new System.Drawing.Size(80, 22);
        this.CurrentLocation_textBox.Text = "";
        ...
    }
}

Fig. 14 SmartSpeaker in C-Sharp on PDA

Fig. 15 SmartSpeaker in J2ME on Smartphone
4.5. Implementation and Deployment

We applied the integrated model transformation mechanism presented in section 3.4 and generated platform-specific code in C-Sharp and J2ME out of the technology-independent models of application logic and GUI. Fig. 13 lists GUI code snippets for C-Sharp and Java. The C-Sharp outcome is deployed on the HP iPAQ PDA (Fig. 14) and the J2ME outcome on the Nokia Smartphone (Fig. 15).

Due to differences in the C-Sharp and J2ME GUI libraries, some modelled GUI elements appear differently on the PDA and Smartphone devices. For instance, the Add and Change GUI elements of Fig. 10 show as buttons in the PDA GUI and as commands under the “Options” menu of the Smartphone GUI. To tackle this, in Fig. 10 we stereotyped Add and Change as <<GUIElement>>, which in the MDIA GUIMetaModel is a superclass to Button and MenuCommand. Then, out of the GUI PIM of Fig. 10 we generated two PSMs: (1) the GUICSharp PSM, where Add and Change are stereotyped as <<Button>>; and (2) the GUIJ2ME PSM, where Add and Change are stereotyped as <<MenuCommands>> and are grouped together in <<Menu>>Options. When we then run the MDIA IntegrationToCSharp transformation, CSharp GUI code is generated, based on the GUICSharp PSM, showing the GUI elements as buttons. Similarly, the IntegrationToJ2ME transformation runs on the GUIJ2ME PSM and generates the J2ME GUI showing the GUI elements in a menu.

5. Conclusions and Future Work

In this paper we presented the MDIA approach for automatic CAAs generation. We presented an improved version of the architecture from that presented in previous work [6], where the ULCOM part of Context Ontology Model shows as meta-model whilst the ESCOM part as an M1 instantiation of ULCOM. This way ESCOM can be extended and adapted as necessary to fit specific context specification requirements that may vary depending on the needs of the CAA under development. Then, we explicitly applied the approach in practice to develop the SmartSpeaker CAA. In this case-study, we showed how each of the MDIA meta-models are used and the MDA benefits that are to be gained. The main lessons learned are:

- MDIA separates GUI, application logic and component concerns providing flexibility to manipulate, manage and change as necessary the CAA design and generated code. This stems from the MDIA IntegrationMetaModel package, which provides the mechanism to associate GUIs with processes and processes with component capabilities without polluting the individual GUI/Process/Component model packages.
- MDIA provides direct forward traceability from PIM models to code, whilst it can cater for generation of code on different platforms. We showed how we can generate SmartSpeaker implementations in J2ME and C-Sharp tackling also platform idiosyncrasies regarding GUI limitations.
- MDIA caters for modelling of context using ontologies, allowing the generated applications to be fully aware of and to be influenced by the context they operate in. MDIA also assists in overcoming ontology heterogeneity issues. As demonstrated in the case study, heterogeneity introduced by the GPS and Bluetooth sensors was overcome through the definition of appropriate semantic mappings that allowed one common perception of ontology entity location in the CAA design.

We are currently working to expand MDIA using design patterns aiming to expedite and significantly enhance the quality of the CAA development process.

6. References