Context-oriented Software Development

Basel Magableh
Trinity College Dublin, Ireland, Ireland
Email: magablb@cs.tcd.ie

Stephen Barrett
Trinity College Dublin, Ireland, Ireland
Email: Stephen.barrett@tcd.ie

Abstract— Context-oriented programming is an emerging technique that enables dynamic behaviour variation based on context changes. In COP, context can be handled directly at the code level by enriching the business logic of the application with code fragments responsible for performing context manipulation, thus providing the application code with the required adaptive behavior. Unfortunately, the whole set of sensors, effectors, and adaptation processes is mixed with the application code, which often leads to poor scalability and maintainability. In addition, the developers have to surround all probable behavior inside the source code. As an outcome, the anticipated adjustment is restricted to the amount of code stubs on hand offered by the creators. Context-driven adaptation requires dynamic composition of context-dependent parts. This can be achieved trough the support of a component model that encapsulates the context-dependent functionality and decouples them from the application's core-functionality. The complexity behind modeling the context-dependent functionality lies in the fact that they can occur separately or in any combination, and cannot be encapsulated because of their impact across all the software modules. Before encapsulating crosscutting context-dependent functionality into a software module, the developers must first identify them in the requirements documents. This requires a formal development paradigm for analyzing the context-dependent functionality; and a component model, which modularizes their concerns. COCA-MDA is proposed in this article as model driven architecture for constructing self-adaptive application from a context oriented component model.

***Index Terms*— Adaptable middleware, Context oriented component, Self-adaptive application, Object.**

I. INTRODUCTION

There is a growing demand for developing applications with aspects such as context awareness and self-adaptive behaviors. Self-adaptive software evaluates its own behavior and changes its behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible. Traditionally, self-adaptability is needed to handle complexity, robustness of unanticipated conditions, changing of priorities and polices governing the objective goals, and changing conditions in the contextual environment. Hirschfeld et al. [1] considered context to be any information that is computationally accessible and upon which behavioral variations depend. A context-dependent application adjusts its behavior according to context conditions arising during execution. The techniques that enable applications to handle the contextual application are generally known as "context-handling" techniques. Context handling is of vital importance for developers and for self-adaptive architecture since it provides dynamic behavioral adaptation and makes it possible to produce more useful computational services for end users in the mobile computing environment [2]. The mobile computing environment is heterogeneous and dynamic. Everything from devices used, resources available, network bandwidths, to user context, can change drastically at runtime [3]. This presents the application developers with the challenge of tailoring behavioral variations to each specific user and context. With the capacity to move and the desire to be socially collaborative, mobile computing users might benefit from the self-adaptability and the context-awareness features that are supported by self-adaptive applications.

This article focuses on describing a development paradigm for Context-Oriented Programming, which enables self-adaptability features in this emerging class of applications. The development methodology Context Oriented Component-based Application Model-Driven Architecture (COCA-MDA) modularizes the application's context-dependent behavior into context-oriented components. The components separate the application's functional concerns from the extra-functional concerns. The application is organized into two casually connected layers: the base layer, which provides the application's core structure, and the meta-layer, where the COCA-components are located, and which provides compassable units of behavior. The component model design (COCA-components) has been proposed in previous work in [4]. A COCA-component refers to any subpart of the software system that encapsulates a specific context-dependent functionality in a unit of behavior contractually specified by interfaces and explicit dependences. The result from this methodology is a component-based application described by an architecture description language, COCA-ADL. COCA-ADL is used to bridge the gap between the

doi:10.4304/jetwi.4.2.172-180

software models in the platform-independent model of the MDA and the software architecture runtime model. Such employment of the ADL decouples the application's architecture design from the target platform implementation.

The rest of the article is structured as follows. Section II discusses behavioral variability support in Context-Oriented Programming and aspects. Section III describes the rationale for providing a development paradigm for context-oriented Programming. Section IV describes the COCA-component model. Section V describes the COCA-ADL elements. The COCA-MDA phases are described in Section VII. Section VIII demonstrates a case study designed using the COCA-MDA and implemented with the COCA-middleware.

II. Variability Management with Context-oriented Programming and Aspects

Compositional adaptation enables an application to adopt a new structure/behavior for anticipating concerns that were unforeseen during the original design and construction. Normally, compositional adaptation can be achieved using the separation of concerns techniques, computational reflection, component-based design, and adaptive middleware [5]. The separation of concerns enables the software developers to separate the functional behavior and the crosscutting concerns of the self-adaptive applications. The functional behavior refers to the business logic of an application [5]. Context-driven behavioral variations are heterogeneous crosscutting concerns and a set of collaborated aspects that extend the application behavior in several parts of the program and have an impact across the whole system. Such behavior is called crosscutting concerns. Crosscutting concerns are properties or areas of interest such as quality of service, energy consumption, location awareness, users' preferences, and security. This work considers the functional behavior of an application as the base-component that provides the user with context-free functionality. On the other hand, context-dependent behavior variations are considered as crosscutting concerns that span the software modules in several places.

Context-oriented programming is an emerging technique that enables context-dependent adaptation and dynamic behavior variations [6, 7]. In COP, context can be handled directly at the code level by enriching the business logic of the application with code fragments responsible for performing context manipulation, thus providing the application code with the required adaptive behavior [8]. Unfortunately, the whole set of sensors, effectors, and adaptation processes is mixed with the application code, which often leads to poor scalability and maintainability [9]. In general, the proposed COP approaches support fine-grained adaptation among the variant behaviour that were introduced at the compile time, A special compiler is needed for performing the context handling operation. To best of our knowledge, COP does not support dynamic composition of software modules and have no support for introducing new behaviour/or adjusting the application structure to anticipate the context changes. In addition, the developers have to surround all probable behavior inside the source code. As an outcome, the anticipated adjustment is restricted to the amount of code stubs on hand offered by the creators. On the other hand, it is impractical to forecast all likely behaviors and program them at the source code.

For a more complex context-aware system, the same context information would be triggered in different parts of an application and would trigger the invocation of additional behavior. In this way, context handling becomes a concern that spans several application units, essentially crosscutting into the main application execution. A programming paradigm aiming at handling such crosscutting concerns (referred to as aspects) is aspect-oriented programming (AOP) [10]. In contrast to COP, Using the AOP paradigm, context information can be handled through aspects that interrupt the main application execution. In order to achieve self-adaptation to context in a manner similar to COP, the context-dependent behavioral variations must be addressed. Unfortunately, the aspect-oriented development methodology can be used to handle homogeneous behavioral variations where the same piece of code can be invoked in several software modules [11, 12], and it does not support adaptation of aspects to context in what is called context-driven adaptation [9]. Moreover, static AOP is classified as a compositional adaptation performed in compile time [5]; anticipating context changes at runtime is not an option, especially with the presence of unforeseen changes. Another approach supported by AOP is called dynamic weaving for aspects [13]; this injects the code in the program execution whenever a new behavior is needed. However, existing AOP languages tend to add a substantial overhead in both execution time and code size, which restricts their practicality for small devices with limited resources [14].

III. Rationale

Context changes are the causes of adaptation. A context-driven adaptation requires the self-adaptive software to anticipate its context-dependent variations. The context-dependent variation can be classified into actor-dependent, system-dependent, and environment-dependent behavior variations. The complexity behind modeling these behavior variations lies in the fact that they can occur separately or in any combination, and cannot be encapsulated because of their impact across all the software modules. Context-dependent variations can be seen as collaboration of individual features (aspects) expressed in requirements, design, and implementation, and are sufficient to qualify as heterogeneous crosscutting concerns in the sense that different code fragments are applied to different program parts. Before encapsulating crosscutting context-dependent behaviors into a software module, the developers must first identify them in the requirements documents. This is difficult to achieve

because, by their nature, context-dependent behaviors are tangled with other behaviors, and are likely to be included in multiple parts (scattered) of the software modules. Using intuition or even domain knowledge is not necessarily sufficient for identifying the context-dependent parts of self-adaptive applications. This requires a formal procedure for analyzing them in the software requirements and separating their concerns. Moreover, a formal procedure for modeling these variations is needed. Such analysis and modeling procedures can reduce the complexity in modeling self-adaptive applications. In this sense, a formal development methodology can facilitate the development process and provide new modularization of a self-adaptive software system in order to isolate the context-dependent from the context-free functionalities. Such a methodology, it is argued, can decompose the software system into several behavioral parts that can be used dynamically to modify the application behavior based on the execution context.

Behavioral decomposition of a context-aware application can provide a flexible mechanism for modularizing the application into several units of behavior. Because each behavior realizes a specific context condition at runtime, such a mechanism requires separation of the concerns of context handling from the concern of the application business logic. In addition, separation of the application's context-dependent and context-independent parts can support a behavioral modularization of the application, which simplifies the selection of the appropriate parts to be invoked in the execution whenever a specific context condition is captured. The adaptive software operates through a series of substates (modes). The substates are represented by j, and j might represent a known or unknown conditional state. Examples of known states in the generic form include detecting context changes in a reactive or proactive manner.

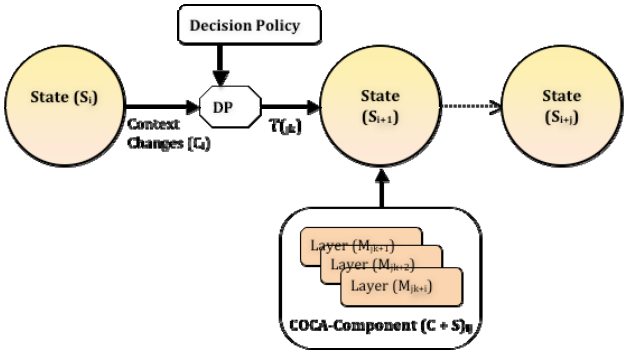

Figure 1: Behavioral Decomposition Model

In the presence of uncertainty and unforeseen context changes, the self-adaptive application might be notified about an unknown condition prior to the software design. Such adaptation is reflected in a series of context-system states. $(C+S)_{ji}$ denotes the i^{th} combination of context-dependent behavior, which is related to the decision points j by the notion mode M_{jk}. In this way, the development methodology decomposes the software into a set of context-driven and context-free states. At runtime, the middleware transforms the self-adaptive software form $state_i$ into $state_{i+1}$, considering a specific context condition t_{jk}, as shown in Figure 1. This enables the developer to clearly decide which part of the architecture should respond to the context changes t_{jk}, and provides the middleware with sufficient information to consider a subset of the architecture during the adaptation; this enhances the adaptation process, impact, and cost and reduces the computation overhead from implementing this class of applications in mobile devices.

Self-adaptive and context-aware applications can be seen as the collaboration of individual behavioral variations expressed in requirements, design, and implementation. This article contributes by proposing a model-driven architecture (COCA-MDA) integrated with a behavioral decomposition technique, based on observation of the context information in requirements and modeling. As a result of combining a decomposition mechanism with MDA, a set of behavioral units is produced. Each unit implements several context-dependent functionalities. This requires a component model that encapsulates these code fragments in distinct architecture units and decouples them from the core-functionality components. This is what motivates the research towards proposing a context-oriented component model (COCA-component). Context-driven adaptation requires dynamic composition of context-dependent parts, which enables the developer to add, remove, or reconfigure components within an application at runtime.

Each COCA-component embeds a specific context-dependent functionality $(C+S)_{ji}$, realized by a context-oriented component (COCA-component) model. Each COCA-component realizes several layers that encapsulate a fragment of code related to a specific software mode layer(M_{jk}), as shown in Figure 1. The developers have the ability to provide a decision $policy_{(jk)}$ for each decision point ($_j$) whenever a specific context-related condition is captured. Hereafter, the COCA-components are dynamically managed by COCA-middleware and their internal parts to modify the application behavior. The COCA-middleware performs context monitoring, dynamic decision-making, and adaptation, based on policy evaluation. The decision policy framework is maintained in modeling and runtime time.

IV. CONTEXT-ORIENTED COMPONENT MODEL (COCA-COMPONENT)

The COCA-component model was proposed in [21], based on the concept of a primitive component introduced by Khattak et al. in [17] and Context-Oriented Programming (COP) [13]. COP provides several features that fit the requirements of a context-aware application, such as behavioral composition, dynamic layers activation, and scoping. This component model dynamically composes adaptable context-dependent applications based on a specific context-dependent

functionality. The authors developed the component model by designing components as compositions of behaviors, embedding decision points in the component at design time to determine the component behaviors, and supporting reconfiguration of decision policies at runtime to adapt behaviors.

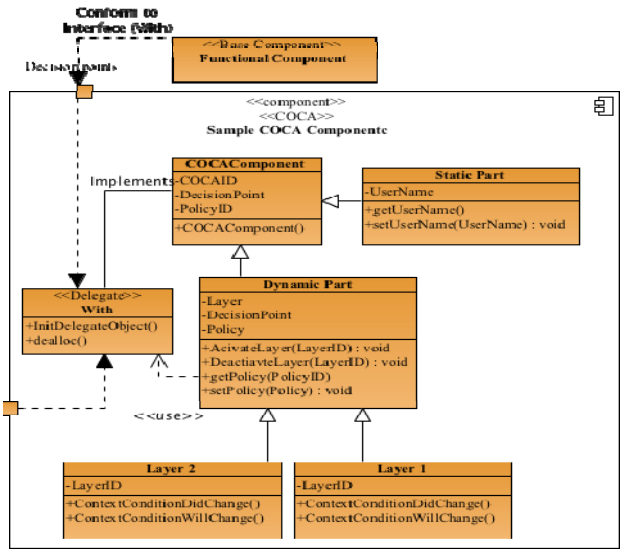

Figure 2: COCA-component Conceptual Diagram

The COCA-component has three major parts: a static part, a dynamic part, and ports. The component itself provides information about its implementation to the middleware. The COCA-component has the following attributes: ID, name, context entity, creation time, location, and remote variable. The Boolean attribute remote indicates whether or not the components are located on the distributed environment. The decision policy and decision points are attributes with getter and setter methods. These methods are used by the middleware to read the attached PolicyID and manipulate the application behavior by manipulating the decision policy.

The COCA-component handles the implementation of a context-dependent functionality through employing the delegate design pattern [6], so the adaptation manager invokes these components whenever the COCA-component is notified by the context manager. A delegate is a component that is given an opportunity to react to changes in another component or influence the behavior of another component. The basic idea is that two components coordinate to solve a problem. A COCA-component is very general and intended for reuse in a wide variety of contextual situations. The base-component stores a reference to another component, i.e., its delegate, and sends messages to the delegate at critical times. The messages may only inform the delegate that something has happened, giving the delegate an opportunity to do extra processing, or the messages may ask the delegate for critical information that will control what happens. The delegate is typically a unique custom object within the controller subsystem of an application [6].

At this stage, each COCA-component must adopt the COCA-component model design. A sample COCA-component is shown in Figure 2; it is modeled as a control class with the required attributes and operations. Each layer entity must implement two methods that collaborate with the context manager. Two methods inside the layer class, namely ContextEntityDidChanged and ContextEntityWillChanged, are called when the context manager posts the notifications in the form [NotificationCeneter Post:ContextConditionDidChanged]. This triggers the class layer to invoke its method ContextEntityDidchanged, which embeds a subdivision of the COCA-component implementation.

V. COCA-ADL: A Context-Oriented Component-Based Application ADL

The aim of this section is to introduce the architecture description language COCA-ADL. COCA-ADL is an XML-based language used to describe the architecture produced by the development methodology COCA-MDA. COCA-ADL is used to bridge the gap between the application models and the implementation language. Thus, it enables the architecture to be implemented by several programming languages.

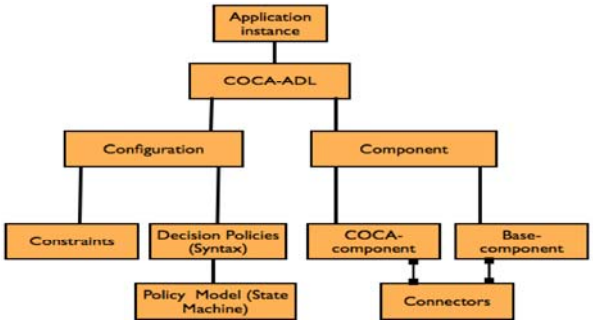

Figure 3: COCA-ADL Elements

COCA-ADL is designed as a three-tier system. The first level consists of the building blocks, i.e., the components, including the COCA-component and base-component. The second refers to connectors, and the third refers to the architecture configuration, which includes a full description of the state-machine models, which describes an activity diagram plus the decision policies' syntax. Figure 3 shows the main elements of COCA-ADL. Each element is associated with an architecture template type. The main features provided by the element types are instantiation, evolution, and inheritance.

VI. COCA-MDA Development Approach

COCA-MDA has adopted the component collaboration architecture (CCA) and the entity model. The CCA details how to model the structure and behavior of the components that comprise a system at varying and mixed levels of granularity. The entity model describes a meta-model that may be used to model entity objects that are a representation of concepts in the application problem domain and define them as composable components [17].

COCA-MDA partitioning the software into three viewpoints: the structure, behavior, and enterprise viewpoints. The structure viewpoint focuses on the core component of the self-adaptive application and hides the context-driven component. The behavior viewpoint focuses on modeling the context-driven behavior of the component, which may be invoked in the application execution at runtime. The enterprise viewpoint focuses on remote components or services, which may be invoked from the distributed environment. The design of a context-aware application according to the COCA-MDA approach was proposed in [18]. The use of COCA-MDA for developing self-adaptive applications for indoor wayfinding for individuals with cognitive impairments was proposed in [19]. Evaluating the COCA-MDA productivity among the development effort was demonstrated in [20]. This article focuses on describing in detail the process of analyzing and classifying the software requirements, and how the software is designed through the platform-independent model and the platform-specific model. Model transformation and code generation were discussed in a previous work [18].

VII. Context-oriented Component-based Application Example

IPetra is a tourist-guide application that helps the client to determine the bravura historical city of Petra, Jordan. IPetra offers a map–client interface maintained by an augmented reality browser (ARB). The browser exhibits many points of interest (POI) inside the physical outlook of the tool's camera. Information related to every POI is exhibited inside the camera overlay outlook. The POIs comprise edifices, tourist services sites, restaurants, hotels, and ATMs in Petra. The AR browser offers an instantaneous live direct physical display inside the portable camera. When the client positions the portable camera in the direction of a building, an explanation confined to a small area related to that edifice is shown to the client.

Constant use of the device's camera, backed by attaining data from many sensors, can consume the tool's resources. This needs the application to adjust its tasks among several contexts to maintain quality of service without disrupting the function's tasks. The function requires frequent updates of client position, network bandwidth, and battery level.

Figure 6 summarizes the modeling tasks, using the associated UML diagrams. The developer starts analysis of an application scenario to capture the requirements. The requirements are combined in one model in the requirements diagram. The requirements diagram is modeled using a use-case diagram that describes the interaction between the software system and the context entity. The use-case is partitioned into two separate views. The core-structure view describes the core functionality of the application. The extra-functionality object diagram describes the COCA-component interaction with the core application classes. The state diagram and the activity diagram are extracted from the behavioral view. Finally, the core structure, the behavioral models, and the context model are transformed into the COCA-ADL model.

A. Computational Independent Model

In the analysis phase, the developers analyze several requirements using separation of concern techniques. The developers focus on separating the functional requirements from the extra-functional requirements as the first stage. They then separate the user and context requirements from each other. There are two subtasks in the analysis phase.

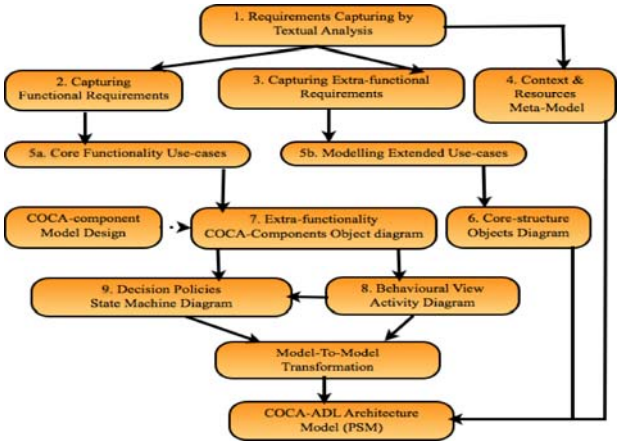

Figure 6: Modelling tasks

1) *Task 1: Requirements capturing by textual analysis*: In this task, the developer identifies the candidate requirements for the illustration scenario using a textual analysis of the application scenario. It is recommended that the developer identifies the candidate actors, use-cases, classes, and activities, as well as capturing the requirements in this task. This can be achieved by creating a table that lists the results of the analysis. This table provides an example of a data dictionary for the proposed scenario.

2) *Task2: Identifying the extra-functional requirements and relating them into the middleware functionality:* The first step in the process is to understand the application's execution environment. The context is classified in the requirements diagram, based on its type, and whether it comes from a context provider or consumer. A context can be generated from a physical or logical source (i.e., available memory), or resources (i.e., battery power level and bandwidth). The representations of sensors and resources on the application that is going to consume them at runtime refers to the context consumers.

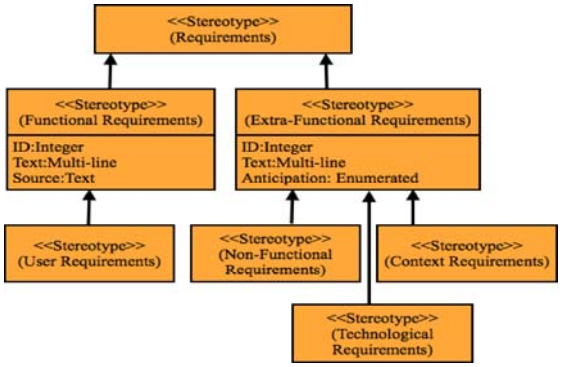

Figure 7: Requirements UML profile

The next level of requirements classification is to classify the requirements based on their anticipation level; this can be foreseeable, foreseen, or unforeseen. This classification allows the developer to model the application behavior as much as possible and to plan for the adaptation actions. However, to facilitate this classification framework, a UML profile is designed to support the requirements analysis and to be used by the software designer, as shown in Figure 7.

As shown in Figure 8, extra-functional requirements are captured during this task, for example, requirement number 3: adapt the location service. IPetra is required to adapt its behavior and increase the battery life. This is achieved by adopting a location service that consumes less power. For example, if the battery level is low, the IPetra application switches off the GPS location services and uses the cell-tower location services. Using an IP-based location reduces the accuracy of the location but saves battery energy. In addition, the application may reduce the number of POIs it displays to the most recent device location. Moreover, the application reduces the frequency of the location updates. On the other hand, if the battery level is high and healthy, IPetra uses the GPS service with more accurate locations. The application starts listening for all events in the monitored region inside Petra city.

3) Task 3: Capturing user requirements: This task focuses on capturing the user's requirements as a subset of the functional requirements, as shown in the UML profile in Figure 7. This task is similar to a classical requirement-engineering process where the developers analyze the main functions of the application that achieve specific goals or objectives.

B. Modelling: Platform Independent Model

In order to be aware of possible resources and context variations and the necessary adaptation actions, a clear analysis of the context environment is the key to building dynamic context-aware applications.

4) Task 4: Resources and context entity model

Resources and context Model refers to generic a overview of the underlying device's resources, sensors, and logical context provider. This diagram is modelling the engagement between the resources and the application under development. It facilitates the developer to understand the relationship between the context providers and their dependency.

5) Task 5: use-cases

The requirements diagram in Figure 8 represents the main inputs for this task. Each requirement is incorporated into a use-case, and the developers identify the actor of the requirement. An actor could be a user, system, or environment. The use-cases are classified into two distinct classes, i.e., the core functionality and extended use-cases, by the context conditions. The first step is to identify the interaction between the actor and the software functions to satisfy the user requirement in a context-free fashion. For example, the displaying POIs functionality in the figure is context independent in the sense that the application must provide it, regardless of the context conditions. All these use-cases are modeled separately, using a class diagram that describes the application core-structure or the base-component model, as shown in the following task.

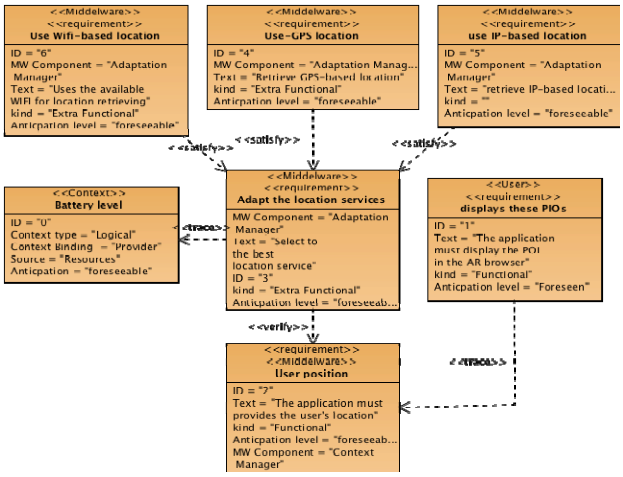

Figure 8: Functional and extra-functional partial requirements diagram

6) Task 6: modeling the application core-structure

In this task, a classical class diagram models the components that provide the application's core functions. These functions are identified from the use-case diagram in the previous task. However, the class diagram is modeled independently from the variations in the context information. For this scenario, some classes, such as "Displaying POIs", "Route-planningUI", "CameraUI", "MapUI", and "User Interface", are classified to be on the application core. These classes provide the core functions for the user during his tour of Petra city. Figure 9 shows the core-structure class-model without any interaction with the context environment or the middleware.

7) Task 7: identifying application-variant behavior (behavior view):

In this phase, the developers specify how the application can adapt to context conditions to achieve a specific goal or objectives. After specifying the core elements of the application in the previous task, the behavioral view is identified in this task. This task identifies when and where an extra-functionality can be invoked in the application execution. This means the developer has to analyze the components involved, their communication, and possible variations in their subdivisions, where each

division realizes a specific implementation of that COCA-component.

To achieve this integration, the developers have to consider two aspects of the context-manager design: how to notify the adaptation manager about the context changes, and the how the component manager can identify the parts of the architecture that have to respond to these changes. These aspects can be achieved by adopting the notification design pattern in modeling the relation between the context entity and the behavioral component. Hereafter, these extra-functionalities are called the COCA-components. Each component must be designed on the basis of the component model described in Figure 2.

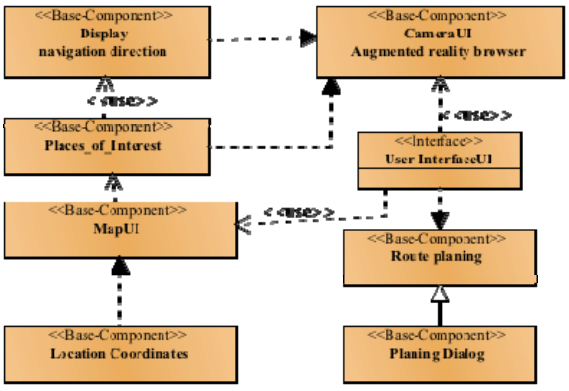

Figure 9: I-Petra Core-Classes structure

The IPetra application is modularized into several COCA-components. Each component models one extra-functionality such as the LocationCOCA−component in Figure 10. The COCA-component sublayers implement several context-dependent functionalities that use the location service. Each layer is activated by the middleware, based on context changes. After applying the observer design pattern and the COCA-component model to the use-cases, the class diagram for the middleware functionality “Update Location” can be modeled as shown in Figure 10. Figure 10 shows a COCA-component modelled to anticipate the ’direction output’. The COCA-component implements a delegate objects and sub layers; each layer implements a specific context-dependent function. The COCA-middleware uses this delegate object to redirect the execution among the sub layers, based on the context condition.

Invoking different variations of the COCA-component requires identification of the application architecture, behavior, and the decision policies in use. As mentioned before, these decision policies play an important role in the middleware functions, which use them in handling, the architecture evolution, and the adaptation action. The model in Figure 10 helps the developer to extract the decision policies and the decision points from the interactions between the context entities and the COCA-components. Decision policies are used by the middleware to select suitable adaptation actions among specific context conditions.

The application behavioural model is used to demonstrate the decision points in the execution that might be reached whenever internal or external variables are found. This decision point requires several parameter inputs to make the correct choice at this critical time. Using the activity diagram, the developers can extract numerous decision polices. Each policy must be modelled in a state diagram, for example, the Policy: Camera Flashes is attached to the ’Camera flashes’ COCA-component. The policy syntax can be described by the code shown in listing 1.

The IPetra application has been implemented in two distinct versions, i.e. with and without the COCA-middleware. The Instruments is a tool application for dynamically tracing and profiling IPhone devices. The battery life has been measured by Energy Diagnostics Instruments running on the device [22]. The energy Diagnostic used to measure the battery while the device is not connected to external power supply. The experiments show that the COCA IPetra application saved the battery-consuming level by 13% in addition to its self-tuning adaptability. Fig. 14 shows the experimental results for energy usage. The IPetra implementation without adopting the COCA-platform consumes more energy during context monitoring, draining the battery faster. On the other hand, when the same application adopts the COCA-middleware, the application is able to adapt its behaviour and use less energy. The adaptation time for handling low and high battery-levels are shown in Figure 13. It is worth mentioning here that when the battery level is low, the COCA-middleware allocates less memory because of the size of the COCA-component, which is small compared to its implementation.

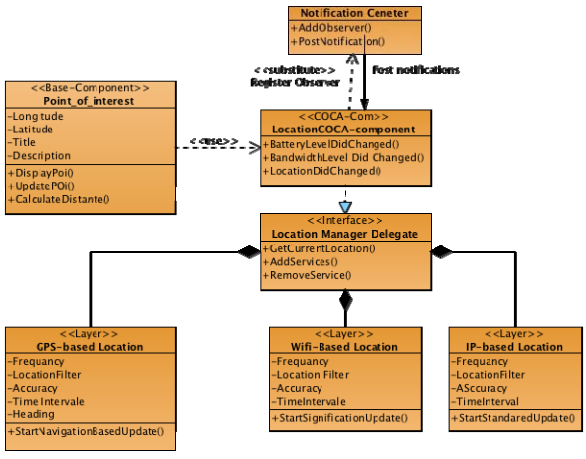

Figure 10: Extra-functionality Object Diagram of the Context Oriented Components

```
If ( direction is Provided && Available memory >=
50
&& CPU throughput <= 89 && light level >= 50
&& BatteryLevel >= 50) then {EnableFlashes();}
Else If ( BatteryLevel < 50 || LightLevel < 50 )
then {DisableFlashes(); SearchForPhotos();}
else If( BatteryLevel < 20)
then DisableFlashes();
```

Listing 1: Adaptation time (ms) and memory allocation (KB)

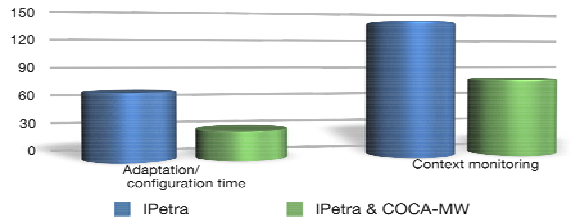

Figure 13: Adaptation time (ms) and memory allocation (KB)

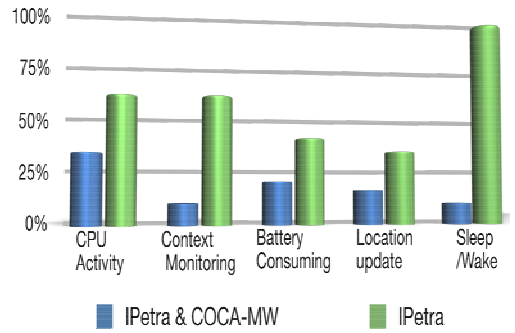

Figure 14: Energy usage for IPetra application.

VIII. Experiments Evaluation

IX. Conclusions and Future Works

This article described a development paradigm for building context-oriented applications using a combination of Model-Driven Architecture that generates an ADL, which presents the architecture as a components-based system, and a runtime infrastructure (middleware) that enables transparent self-adaptation with the underlying context environment.

Specifically, a Model-Driven Architecture is used to demonstrate a new approach to building context-aware and self-adaptive applications by adopting a Model-Driven Architecture (COCA-MDA). COCA-MDA enables developers to modularize applications based on their context-dependent behaviors, enables developers to separate context-dependent functionalities from the application's generic functionality, and enables dynamic context-driven adaptation without overwhelming the quality attributes.

The COCA-MDA needs to be improved with respect to support for both requirement reflection and modeling requirements as runtime entities. The requirement reflection mechanism requires support at the modeling level and at the architecture level. Reflection can be used to anticipate the evolution of both functional and non-functional requirements. The decision policies require more development with respect to policy mismatch and resolution. This is in line with an improvement in terms of self-assurance and dynamic evaluation of the adaptation output.

References

[1] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, March 2008.

[2] A. K. Dey, "Providing architectural support for building context-aware applications," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2000.

[3] N. M. Belaramani, C.-L. Wang, and F. C. M. Lau, "Dynamic component composition for functionality adaptation in pervasive environments," in *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, ser. FTDCS '03. plus 0.5em minus 0.4emIEEE Computer Society, 2003, pp. 226–232.

[4] B. Magableh and S. Barrett, "Pcoms: A component model for building context-dependent applications," in *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, ser. COMPUTATIONWORLD '09. plus 0.5em minus 0.4emWashington, DC, USA: IEEE Computer Society, 2009, pp. 44–48.

[5] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, pp. 56–64, July 2004.

[6] M. Gassanenko, "Context-oriented programming," in *Proceedings of the EuroFORTH'93 conference*, Marianske Lazne (Marienbad), Czech Republic, 15-18 October 1998, pp. 1–14.

[7] R. Keays and A. Rakotonirainy, "Context-oriented programming," in *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, ser. MobiDe '03, San Diego, CA, USA, 2003, pp. 9–16.

[8] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 14:1–14:42, May 2009.

[9] G. Kapitsaki, G. Prezerakos, N. Tselikas, and I. Venieris, "Context-aware service engineering: A survey," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1285–1297, 2009.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 âEuropean Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science. plus 0.5em minus 0.4emSpringer Berlin / Heidelberg, 1997, vol. 1241, pp. 220–242.

[11] S. Apel, T. Leich, and G. Saake, "Aspectual mixin layers: aspects and features in concert," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. plus 0.5em minus 0.4emShanghai, China: ACM, 2006, pp. 122–131.

[12] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 127–136, October 2004.

[13] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *Proceedings of the 1st international conference on Aspect-oriented software*

development, ser. AOSD '02. plus 0.5em minus 0.4emNew York, NY, USA: ACM, 2002, pp. 141–147.

[14] C. Hundt, D. Stöhr, and S. Glesner, "Optimizing aspect-oriented mechanisms for embedded applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6141 LNCS, pp. 137–153, 2010.

[15] Y. Khattak and S. Barrett, "Primitive components: towards more flexible black box aop," in *Proceedings of the 1st International Workshop on Context-Aware Middleware and Services: affiliated with the 4th International Conference on Communication System Software and Middleware (COMSWARE 2009)*, ser. CAMS '09. plus 0.5em minus 0.4emNew York, NY, USA: ACM, 2009, pp. 24–30.

[16] E. Buck and D. Yacktman, *Cocoa design patterns*, 2nd ed. plus 0.5em minus 0.4emDeveloper's Library, 2010.

[17] "Enterprise collaboration architecture (eca) specification," http://www.omg.org/, pp. 1–202, Feb 2004.

[18] B. Magableh and S. Barrett, "Objective-cop: Objective context oriented programming," in *International Conference on Information and Communication Systems*, ser. ICICS 2011, vol. 1, May 2011, pp. 45–49.

[19] ------, "Self-adaptive application for indoor wayfinding for individuals with cognitive impairments," in *The 24th IEEE International Symposium on Computer-Based Medical Systems*, ser. CBMS 2011, vol. In press, Lancaster, UK, June 2011, pp. 45–49.

[20] B. Magableh, "Model-Driven productivity evaluation for self-adaptive Context-Oriented software development," in *5th International Conference and Exhibition on Next Generation Mobile Applications, Services, and Technologies (NGMAST'11)*, vol. In press, Cardiff, Wales, United Kingdom, Sep. 2011.

[21] R. Anthony, D. Chen, M. Pelc, M. Perssonn, and M. Torngren, "Context-aware adaptation in dyscas," in *Proceedings of the Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2009)*, 2009, p. 15.

[22] Ios 4.0 apple developer library. http://developer.apple.com/library/ios/navigation/ (2010), "[Online; accessed 1-April-2011]"

Basel Magableh received his MS degree in computer science from New York Institute of Technology, NY, USA, in 2004. He is currently a Ph.D. candidate at Distributed Systems Group, Trinity College Dublin, Ireland. His research focuses in integrating Model Driven Architecture with a component-based system to construct self-adaptive and context-aware applications.

He is a full-time lecturer in Grafton College of Management Science, Dublin, Ireland. He was member of staff in the National Digital Research Center of Ireland from 2008- 2011.

Stephen Barrett is currently a lecturer at Distributed Systems Group, Trinity College Dublin, Ireland. His research centers on middleware support for adaptive computing. (with particular focus on model driven paradigms) and on large scale applications research (particularly in the context of web search, trust computation and peer and cloud computing) .