Objective-COP: Objective Context Oriented Programming

Basel Magableh
Department of Computer Science and Statistics
Trinity College Dublin, Ireland
Email:magablb@cs.tcd.ie

Stephen Barrett
Department of Computer Science and Statistics
Trinity College Dublin, Ireland
Email:stephen.barrett@tcd.ie

***Abstract*—This paper introduces a context-oriented component-based application MDA (COCA-MDA) that modularizes the application's context-dependent behaviour 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 composable units of behaviour. This architecture provides software developers with a clear modular structure, which reflects a clear separation of concerns between the context provider and context consumer. In addition, this paper demonstrates the possibility of implementing context-oriented programming with Objective-C. The COCA-middleware enables the context-aware application to modify its behaviour dynamically, based on the execution context conditions.**

***Index Terms*—Adaptable middleware, Context-oriented component, Self-adaptive application, Objective-COP, Context Oriented Programming, .**

I. Introduction

The main goal of model-driven architecture (MDA) is to improve software quality by using a model as a means of producing high-quality systems with decreased development costs. In a similar way, MDA enables the developer to represent software solutions through a model and to evaluate the model instead of the base code. The component model design (COCA-components) has been proposed in previous work in [15]. A COCA-component refers to any subpart of the software system that is associated with a specific context condition. A COCA-component is a unit of behaviour contractually specified by interfaces and explicit dependences. The use of Architecture Description Language (COCA-ADL) provides code mobility for the same application into various implementation platforms. The COCA-ADL syntax and design have been proposed in previous work [16].

A self-adaptive application modifies its own structure and behaviour in response to changes in its operating environment [21]. The information that describes the computation environment is called context information. Software applications that use context information to adapt their behaviour are called context-aware applications. Context awareness is a key technology for enabling applications to be sensitive about the social and physical environments in which they operate. Many precise definitions of context can be found in the literature [6], [9], [24].

In line with the definition proposed by Hirschfeld et al. [13], context is considered as any information that is computationally accessible and upon which behavioural variations depend. Context information is provided by a context source [4] or provider [22]. These providers are software or hardware entities distributed in the computation environment. Normally, a runtime infrastructure provides context monitoring, gathering, and reasoning among this context information. This includes a context-binding mechanism, which refers to the association between the application architecture's units and the context information [3].

The key challenge for constructing context-aware applications is the formulation and development of systems and applications that are capable of managing (i.e. configuring, adapting, optimizing, protecting, and healing) themselves. This requires a platform support, including a model-based approach, component-based system, and middleware, that is able to reason about various heterogeneous context conditions.

Heterogeneous cross-cutting concerns refer to multiple points that extend the application behaviour by adding multiple extensions. Each extension is individually implemented by a distinct piece of code, which affects exactly one join point. This implies having multiple code fragments that are applied in different program parts [2].

Hochmuller et al. refer to the technological and non-functional properties of software architecture as extra-functionalities [14]. In other works, they refer to concerns in the architecture quality attributes that may be tangled, i.e. several concerns are observed in the same component, or scattered, i.e. the same concern is observed in more than one component. Broens et al. [3] refer to these properties as context requirements. This work refers to them as extra-functionalities that extend the application behaviour in response to context changes. Hirschfeld et al. refer to these behaviours as context-dependent behaviours [13]. This research focuses on addressing these concerns as context-oriented components (COCA-components). The COCA-components are managed by adaptable middleware, which modifies the application behaviours.

The rest of the article is structured as follows. Section II provides a comparative analysis of the related works. The COCA-MDA phases are described in Section III. Section

IV provides an overall description of the COCA-middleware. Section V demonstrates a case study designed using the COCA-MDA and implemented with the COCA-middleware.

II. Related Works

Compositional adaptation techniques can be used to achieve dynamic context-driven adaptations [17]. Adaptation is divided into internal and external approaches, based on the separation of the adaptation mechanism from the application logic. The internal approach intertwines the application and the adaptation logic. As a result, 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 behaviour [23]. This approach usually involves the extension of existing programming languages, either at the syntax level or by providing complementary external mechanisms. Context-oriented programming (COP) has been proposed as an internal adaptation mechanism [11], [13]. In this approach, the whole set of sensors, effectors, and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability.

The external approach uses an external adaptation engine to perform the adaptation processes. In this approach, the self-adapting software system consists of an adaptation engine and adaptable software. The external engine implements the adaptation logic, mostly with the aid of middleware [7], [10], [19], a policy engine [1], or other application-independent mechanisms. However, a complex software system requires a mixed approach between internal and external adaptations, which provides a composition of elements in an appropriate architecture. The middleware flexibility in adopting a suitable adaptation approach provides a lead to achieving the adaptation results with less cost and several levels of granularity [23].

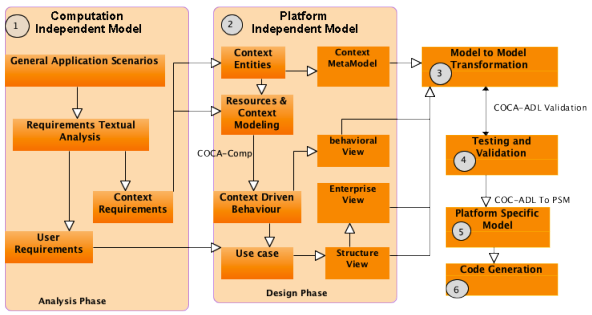

Fig. 1. Context-oriented component-based application model-driven architecture (COCA-MDA) development.

III. Generic Phases of COCA-MDA

The COCA-MDA follows the principles of the object management group (OMG) model-driven architecture. In MDA, there are three different viewpoints to the software: the computation-independent view (CIV), platform-independent view (PIV), and platform-specific view (PSV). The CIV focuses on the environment of the system and the requirements for the system, and hides the details of the software structure and processing. The PIV focuses on the operation of a system and hides the details that are dependent on the deployment platform. The PSV combines the CIV and PIV with an additional focus on the details of the use of a specific platform by a software system [20].

The enterprise collaboration architecture (ECA) [12] is another standard presented by OMG. ECA aims to provide a development methodology to simplify the development of component-based systems by means of a modelling framework and conforming to the OMG-MDA. ECA describes a methodology for building a component-based system at varying and mixed levels of granularity.

Component structure and behaviour are defined by partitioning the system specification into several viewpoints. The application's architecture is described by recursive decomposition and assembly of parts to apply it to several domains. The ECA comprises a set of five models. Each model consists of a set of model elements that represent concepts needed to model specific aspects of the software system. However, COCA-MDA has adopted the component collaboration architecture (CCA) and the entity model. The CCA details how to model the structure and behaviour 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 [12].

However, COCA-MDA adopts the CCA specification at the Platform Independent Model (PIM) phase by partitioning the software into three viewpoints: the structure, behaviour, and enterprise viewpoints. The structure viewpoint focuses on the core component of the self-adaptive application and hides the context-driven component. The behaviour viewpoint focuses on modelling the context-driven behaviour 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 in general involves six main phases, as shown in Fig. 1.

1) Analysis: The requirements for the system are modelled in a computation-independent model describing the situation in which the system will be used. Such a model is sometimes called a domain model or a business model. It may hide much or all of the information about the use of automated data-processing systems. Typically, such a model is independent of how the system is implemented. A CIM is a model of a system that shows the system in the environment in which it will operate, and thus it helps in presenting exactly what the system is expected to do. It is useful not only as an aid to understanding a problem, but also as a mechanism for predicting the exact behaviour of a software system as a

result of runtime changes. The result from this phase is a requirements diagram describing the user and functional requirements besides the extra-functional requirements that describe a context-dependent functionality.

2) Modelling and design: Platform-independent model. The platform-independent viewpoint focuses on the operation of a system while hiding the details necessary for a particular platform. A platform-independent view shows the part of the complete specification that does not change from one platform to another. In this phase, the requirements diagram is incorporated into a use-case model. The use-cases describe the interactions between the software system and the actor. The system-dependent and environment-dependent behaviours are modelled as an extension of the functional use-cases. The functional use-cases are modelled in a class diagram describing the application core functions. The extended use-cases are modelled as another class diagram describing the application behavioural view. The variant behaviour model is supported by a state-machine model that describes the application decision polices. The three models of the application are used as input for the next phase, i.e. model-to-model transformation.
3) Model-to-model transformation: The platform-independent model and behavioural model are transformed into architecture description language (COCA-ADL). This phase includes model-to-model transformation and model verification for the application's structure, behaviour, and enterprise views. The COCA-ADL is implemented by extending the xADL schema, which is an extensible XML language. ArchStudio [8] helps developers to model the architecture using three grouped models: activity diagram, state diagram, and structure diagram.
4) Testing and validating: Tests the model and verifies its fitness for the application goals and objectives.
5) Platform-specific model: The platform-specific model produced by the transformation is a model of the same system specified by the PIM; it also specifies how that system makes use of the chosen platform. A PSM may provide more or fewer details, depending on its purpose. A PSM will be an implementation if it provides all the information needed to construct a system and to put it into operation, or it may act as a PIM that is used for further refinement to a PSM that can be directly implemented.
6) Code generation: Model-to-text includes model-to-text transformation deployment and execution verification. The COCA-ADL XMI code is transformed into the implementation language.

IV. Overview of the COCA-middleware

The COCA-platform offers a context-aware middleware environment for adjusting the application's behaviour dynamically. Fig. 2 shows the COCA-middleware architecture. The platform is layered into four major layers. Each layer provides

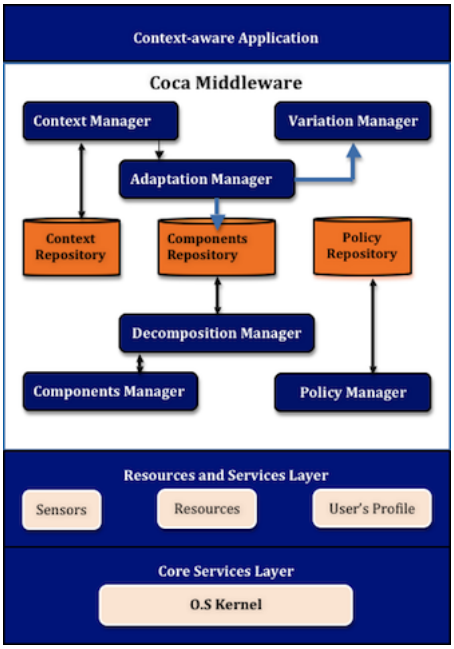

Fig. 2. COCA-platform architecture.

an abstraction of the underlying technology. Each layer is totally platform independent of any given technology. The first layer represents the context-aware application. It provides the user with GUI, functional properties, and non-functional properties.

The second layer in the platform represents the COCA-middleware. It has the middleware components. The middleware has nine components in total. There are six computation components: the context manager, adaptation manager, variation manager, components manager, policy manager, and decomposition manager. These computation components are linked to three data-storage repositories. These repositories are the context repository, components repository, and policy repository.

The third layer in the platform represents the resources and services layer. It provides information about the device's resources, physical sensors, and remote services to the middleware. The context information is retrieved from the OS, resources,and physical sensors. The OS kernel is located in the fourth layer in Fig. 2. The OS sensor retrieves information about the OS. Function calls are used to retrieve information about CPU, memory, and disk space.

V. Context-oriented Component-based Application Example

IPetra is a tourist-guide application that helps the client to explore the magnificent 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

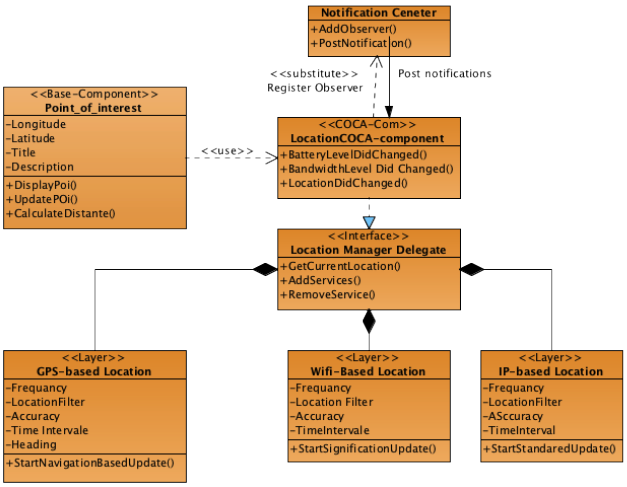

Fig. 3. Location COCA-component with sublayers.

site, 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 with attainment 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 services without disrupting the function's tasks. The function requires frequent updates of client position, network bandwidth, and battery level.

The self-adaptive context-aware IPetra application is designed and implemented using the COCA-component model and COCA-middleware proposed in [15]. Fig. 4 shows the region that IPetra is monitoring, with sample PIOs; the AR browser displays these PIOs inside the camera overlay view.

IPetra is required to adapt its behaviour 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.

The IPetra application is modularized into several COCA-components. Each component models one extra-functionality such as the $LocationCOCA - component$ in Fig. 3. 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.

Listing 1. Loading COCA-component

```
//file: LocationCOCA−component.h
#import <objc/Object.h>
#import "stdio.h"
#import "Location_Manager_Delegate.h"

@class Location_Manager_Delegate;

@interface Update_Location :Object <Location_Manager_Delegate> {
}

−(void) BatteryLevelDidChanged;

−(void) BandwidthLevelDidChanged;

−(void) LocationDidChanged;
@end

@implementation Update_Location
−(void) BatteryLevelDidChanged {
    [AdaptationManager BatteryLevelDidChanged]
}

−(void) BandwidthLevelDidChanged {
    [AdaptationManager BandwidthLevelDidChanged]
}

−(void) LocationDidChanged {
    [AdaptationManager LocationDidChanged]
}
@end
```

Listing 2. Location Manager Delegate

```
//file: Location_Manager_Delegate.h
#import "GPS_based_Location.h"
#import "Update_Location.h"

@class GPS_based_Location;
@class Update_Location;

@protocol Location_Manager_Delegate :GPS_based_Location <
Update_Location>
−(void) GetCurrentLocation;
−(void) AddServices;
−(void) RemoveService;
@end
```

Listing 3. IP-based location Layer

```
 //file: "IP−based Location.h"
#import "Location_Manager_Delegate.h"
@class Location_Manager_Delegate;
@interface IP−based Location : Location_Manager_Delegate {
    private NSFloat frequancy;
    private NSString locationFilter;
    private NSFloat Accuracy;
    private NSInterval timeInterval;
}
−(void) StartStandaredUpdate;
@end

@implementation IP−based Location
− (void)startStandardUpdates
{ // Create the location manager if this object does not // already have one.
    if (nil == locationManager)
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
    locationManager.desiredAccuracy = kCLLocationAccuracyKilometer;
    locationManager.distanceFilter = 1000;
[locationManager startUpdatingLocation];
}@end
```

A partial snapshot for the location manager COCA-component is shown in Listing. 1. The code in listing 2 shows the location manager delegate. The code in listing 3 shows the implementation of a sub layer of COCA-component. This layer implementing a call to an IP-based location update services. The Figure 3 shows the COCA-component and its layers. This component is invoked in the execution whenever "Location-DidChanged" is detected by the context manager. The "Point-of-interest" is a base-component that invokes a current location interface. This interface invokes the "COCA-Component Location Manager", which handles this situation by adopting one of the three dynamic layers. Each layer is implemented as a layer-in-class. The class is simply a subclass of the location manager. The "StandaredUpdates" layer is activated in response to the notification "BatteryLevelDidChanged". This layer implements "StartStandaredUpdates", which uses IP-based location updates. The "Wifi-based location" layer is activated whenever the battery level is notified and a Wifi connection is found. The "GPS-based location" is activated only when the battery level is high.

Fig. 4. IPetra region-monitoring view.

The middleware adopts the notification (observer) design pattern [5], [18]. This reduces the tight coupling between the context provider and consumer. In addition, the COCA-middleware context manager only notifies the component when the context information of interest is changed. This reduces the frequency of context information and provides one-to-one binding between the context provider and consumer; this enables the adaptation manger to dynamically decide which behaviour should be adopted. The location manager

	IPetra & COCA-	IPetra
CPU Activity	33%	59%
Context Monitoring	10%	59%
Battery Consuming	20%	40%
Location update	16%	34%
Sleep/Wake	10%	95%

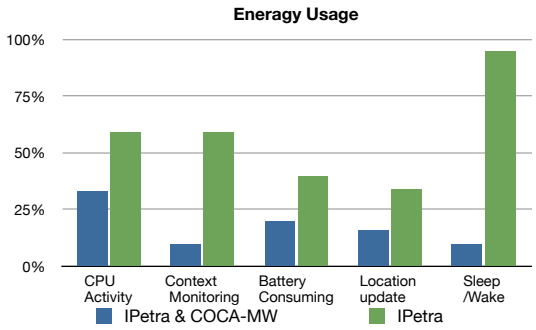

Fig. 5. Energy usage for IPetra application.

COCA-component registers itself with the context manager to be notified only when the battery level is low. When low battery is found at runtime, the context manager notifies $[PostNotficationBatteryLevelDidChanged]$ to the adaptation manager and the location manager to activate its sub-behavioural layer (BatteryLevelLowLayer). As a result, the adaptation manager invokes the layer implementation, and executes and deactivates the "ArBrowserWithHighBatteryLevel" layers.

	Adaptation /configuration time	Context Monitoring
IPetra & COCA-MW	67	137
IPetra	29	78

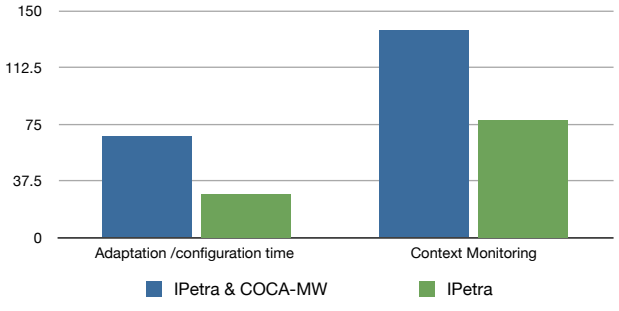

TABLE I
ADAPTATION TIME (MS)

	Memory allocation	Low battery adaptation	High Battery adaptation
IPetra & COCA-MW	9917	9917	9917
IPetra	10306	10306	10306

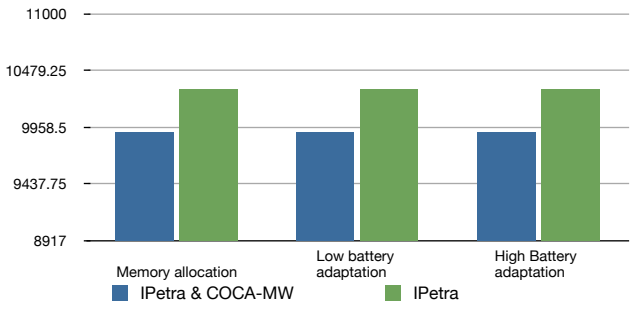

TABLE II
MEMORY ALLOCATION IN KILO BYTE (KB)

The IPetra application has been implemented in two distinct versions, i.e. with and without the COCA-middleware. The battery life has been measured by running each version in an IPhone Device 4. The experiments show that the COCA IPetra application saved the battery-consuming level by 13% in addition to its self-adaptability. Fig. 5 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/configuration time and the context monitoring time for handling low and high battery-levels are shown in Table I. 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. The memory allocation in kilo byte is shown in Table II.

REFERENCES

[1] Anthony, R., Chen, D., Pelc, M., Persson, M.: Context-aware adaptation in dyscas. eceasst.cs.tu-berlin.de (2010)

[2] Apel, S., Batory, D., Rosenmüller, M.: On the structure of crosscutting concerns: Using aspects or collaborations. GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE) (2006)

[3] Broens, T., Quartel, D., Sinderen, M.V.: Capturing context requirements. Smart Sensing and Context pp. 223–238 (2007)

[4] Broens, T.: Dynamic context bindings: infrastructural support for context-aware applications (2008)

[5] Buck, E., Yacktman, D.: Cocoa design patterns. Addidon-Wesley (2010)

[6] Chen, G., Kotz, D.: A survey of context-aware mobile computing research. IEEE Workshop on Mobile Computing (Jan 2000)

[7] Chusho, T., Ishigure, H., Konda, N., Iwata, T.: Component-based application development on architecture of a model, ui and components. apsec p. 349 (2000)

[8] Dashofy, E., Asuncion, H., Hendrickson, S.: Archstudio 4: An architecture-based meta-modeling environment. Software Engineering-Companion (Jan 2007)

[9] Dey, A., Abowd, G., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16(2), 97–166 (2001)

[10] Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE software 23(2), 62–70 (2006)

[11] Gassanenko, M.: Context-oriented programming. EuroForth'98 (1998)

[12] Group, O.M.: Enterprise collaboration architecture specification. Tech. rep., Object Management Group inc (2004)

[13] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3) (Jan 2008)

[14] Hochmüller, E., Dobrovnik, M.: Identifying types of extra-functional requirements in the context of business process support systems. Workshop on Requirements for Business Process Support (REBPS'03), Velden (2003)

[15] Magableh, B., Barrett, S.: Pcoms: A component model for building context-dependent applications. 2009 Computation World: Future Computing pp. 44–48 (Jan 2009)

[16] Magableh, B., Barrett, S.: Primitivec-adl: Primitive component architecture description language. INFOS2010 1, 1–8 (Mar 2010)

[17] McKinley, P., Sadjadi, S., Kasten, E., Cheng, B.: Composing adaptive software. Computer 37(7), 56–64 (2004)

[18] Meyer, B.: Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)

[19] Mukhija, A., Glinz, M.: The casa approach to autonomic applications. Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005) (2005)

[20] The architecture of choice for a changing world. `http://www.omg.org/mda/` (October 2010)

[21] Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. Intelligent Systems and Their Applications, IEEE 14(3), 54–62 (1999)

[22] Paspallis, N., Papadopoulos, G.A.: An approach for developing adaptive, mobile applications with separation of concerns. In: Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01. pp. 299–306. IEEE Computer Society, Washington, DC, USA (2006)

[23] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. Transactions on Autonomous and Adaptive Systems (TAAS 4(2) (May 2009)

[24] Schmidt, A., Beigl, M., Gellersen, H.: There is more to context than location. Computers & Graphics 23(6), 893–901 (1999)