Context-Oriented Component-based Software Development

Basel Magableh

A Thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

February 2012

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Basel Magableh

Dated: February 13, 2012

Acknowledgements

First and foremost, I thank my supervisor Stephen Barrett for his guidance, and support over the years. I very much appreciate his motivation, feedback, and endless patience during my Ph.D. studies. I would like to thank Dr. Stefan Weber for his continuous support and guidance. A very special thank you to Professor Khalid Al- Begain, for his decent advices and guidance.

I express my gratitude to the women who supports me through the past four years, who gave me love and great company to produce this volume of work. To Hebah my lovely wife, the one who suffers the most from being very far away from home. This thesis would not have been possible unless your hard support my dear Hebah. This work is totally yours.

I will dedicate this work to you my dear son Sanad. Because of you my lovely lad this work was started and finished. I lift my family and home land to improve my career to assure you will have a better future. To the man who always motivates me and provides me with decent advices to keep fighting and to add very small footprint in this world. To my father Ismail, who was always advising me to be creative and do my best. To the women who always keep filling my heart with forgiveness and faith to be a better human in this world. To my mother, to that hard worker that spent her life minding my needs worrying about my wishes.

To all past and present members in the Distributed Systems Group, thank you for making it such a great and lively research environment to be in. A big thank you goes especially to my room mates of Lloyd 008 for lively discussions (not only) about work, and for being great colleagues.

A very special thank you to all my friends for encouraging me, each in their own way. Each one of you deserves a big thank you. My regards should go especially to my journey fellows: Dr. Nearchos Paspallis, Dr. Melanie Bouroche, Dr. Serena Fritsch, Dr. Mazeiar Salehie, Dr. Mark gleeson, Dr. Asad Salkham and Dr. Pierpaolo Dondio. A big thank you also to my dear friends: Mr. Khaled Smadi, Mr. Nidal AL-Buriuite and Mr. Azez Al-Rawashdeh.

Finally, I would like to thank my father-in-law Omar. Thank you to all my dear brothers, Dr. Amer, Dr. Mohammad, Mr. Ala'a, Eng. Mo'ad, Mr. Bashar, Mr. Ahmad and Mr. Ibrahim for your support and encouragements.

Basel Magableh

University of Dublin, Trinity College

February 2012

Abstract

Software in distributed and mobile computing environments needs to cope with variability, diversity of computing platforms and operates in different execution environments. Mobile computing environments are heterogeneous and dynamic. Everything from the devices used and resources available to network bandwidth and user context can change at runtime. This presents the software developers with the challenge of tailoring behavioural variations of the software to specific user needs and adapt to context changes. The design and the development of context-dependent and self-adaptable applications in mobile computing environments cannot rely on classical software-development methodologies, which assume that the software execution environment is known a priori at design time, and the application environment can be anticipated. Supporting the development and execution of self-adaptive software systems raises numerous challenges, from development processes, design space and development tools, to the adaptation mechanism that ensures adaptability and dependability of the self-adaptive software that is targeted. This thesis explores how far we can support the engineering of self-adaptive applications using generic development paradigm provided by non-specialized language frameworks, and not being limited to a specific platform or mechanism. This gives the software developers the flexibility to construct a self-adaptive application using an object-oriented programming language and deploy it on several platforms.

The thesis is that the software developers must considered the context information and context-dependent behaviour in the analysis, design and implementation of self-adaptive software. In particular, software needs to consider the composition of its components in conjunction with the contextual changes. In order to overcome the problem and the challenges of engineering self-adaptive software, this thesis contributes to the knowledge by presenting Context-Oriented Software Development (COSD), a generic development paradigm for the construction of self-adaptive software from context-oriented components, which enables dynamic composition of the context-dependent behaviours and provides the software with capabilities of self-adaptability and dependability in mobile computing environments. Our model is based on a decomposition strategy of self-adaptive software based on context, which provides a flexible mechanism for modularising the software into several composable units of behaviour and decouples the context-dependent from context-free parts. The context-oriented component model encapsulates the implementation of the context-dependent parts in distinct architecture units, which enables the software to adjust its functionality and/or behaviour dynamically. This differs from the majority of existing works, which seek to embed awareness of context in the functional implementation of applications. The Context-Oriented Software is developed using a Context-Oriented Component-based Application Model Driven Architecture (COCA-MDA). Afterwards, the context-oriented software is manipulated at runtime by COCA-middleware, which performs a runtime behavioural composition of the context-dependent functionality based on the operational context. The evaluation of context-oriented software in comparison to existing work shows that context-oriented software development is better suited for implementing context-dependent and self-adaptive applications. In addition, the evaluation of COCA-middleware in terms of the modifiability and performance quality attributes, shows better performance in performing the adaptation with less impact on the allocated resources. This thesis shows that COCA-MDA has reduced the development effort in modelling the Platform Independent Model (PIM) and Platform Specific Model (PSM), as it reduces the amount of configurations and maintenance needed to transform the PIM into PSM. In addition, COCA-MDA produced a component-based architecture described by an Architecture Description language (ADL), which reduces the effort needed to implement the architecture in different platforms.

Publications Related to this Ph.D.

1. Magableh, B. and Barrett, S. (2011). Context Oriented Software Development. Journal of Emerging Technologies in Web Intelligence (JETWI), 3(4), pages 206–216.

2. Magableh, B. and Barrett, S. (2011). Adaptive Context Oriented Component-Based Application Middleware (COCA-Middleware). In Proceedings of the 8th International Conference of Ubiquitous Intelligence and Computing, (UIC 2011), pages 137–151, Banff, Canada.

3. Magableh, B. and Barrett, S. (2011). Model-Driven Productivity Evaluation for Self-adaptive Context-Oriented Software Development. In Proceedings of the 5th International Conference and Exhibition on Next Generation Mobile Applications, Services, and Technologies, (NGMAST 11), pages 158–167, Cardiff, Wales, United Kingdom.

4. Magableh, B. and Barrett, S. (2011). Self-adaptive application for indoor wayfinding for individuals with cognitive impairments. In Proceedings of the 24th International Symposium on Computer-Based Medical Systems, (CBMS 11), pages 1–6, Bristol, United Kingdom.

5. Magableh, B. and Barrett, S. (2011). Objective-COP: Objective Context Oriented Programming. In Proceedings of the first International Conference on Information and Communication Systems, (ICICS 2011), pages 45–49, Irbid, Jordan.

6. Magableh, B. and Barrett, S. (2010). PrimitiveC-ADL: Primitive Component Architecture Description Language. In Proceedings of the 7th International Conference on Informatics and Systems, (INFOS 2010), pages 103–118, Cairo. Egypt.

7. Magableh, B. and Barrett, S. (2009). PCOMs: A Component Model for Building Context-Dependent Applications. In Proceedings of the First International Conference on Adaptive and Self-adaptive Systems and Applications, (Adaptive 09), pages 44–48, Athens, Greece.

Contents

List of Tables

List of Figures

Listings

Glossary

Notation	**Description**
advice	Describes a class of functions which modify other functions when the latter are run; it is a certain function, method or procedure that is to be applied at a given join point of a program.
aspect	An aspect of a program is a feature linked to many other parts of the program, but which is not related to the program's primary function. An aspect crosscuts the program's core concerns, therefore violating its separation of concerns that tries to encapsulate unrelated functions.
context	Any information that is computationally accessible and upon which behavioural variations depend.
context-aware application	refer to a class of software systems that are able to monitor and detect context changes in the environment where they operate.
context-awareness	The software system is aware of its context, which is its operational environment.

Notation	**Description**
context-dependent	A context-dependent application adjusts its behaviour according to context conditions arising during execution.
crosscutting	Properties or areas of interest such as quality of service, energy consumption, location awareness, users' preferences, and security.
desmet	A methodology for evaluating software engineering methods and tools.
heterogeneous	A set of collaborated aspects (code fragments), that extend the application behaviour in several parts of the program and have an impact across the whole software system.
homogeneous	Applying the same code, that extend the application behaviour in several parts of the program.
joinpoint	A point in the control flow of a program. In aspect-oriented programming a set of join points is described as a pointcut. A join point is a specification of when, in the corresponding main program, the aspect code should be executed..

Notation	**Description**
pointcut	Is a set of join points. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed. This allows a programmer to describe where and when additional code should be executed in addition to an already defined behavior..
self-* properties	The autonomic properties of a software, which includes (self-organising, self-healing, self-optimising and self- protecting).
self-adaptive	A self-adaptive application modifies its own structure and behaviour in response to changes in its operating environment.
self-healing	The capability of discovering, diagnosing and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent a failure.
self-optimising	The capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end response time, throughput, utilisation and workload are examples of important concerns related to this property.

Notation	**Description**
self-organising	The capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing/decomposing software entities.
self-protecting	The capability of detecting security breaches, anticipating problems and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or mitigate their effects.
separation of concerns	Is the process of separating a computer program into distinct features that overlap in functionality as little as possible..

Acronyms

Notation	Description
A-MUSE	Architectural Modeling for Service Enabling in Freeband.
ADL	Architecture Description language.
AOP	Aspect-Oriented Programming.
AOSD	Aspect Oriented Software Development.
ASLOC	Adapted source lines of code.
AspectJ	Aspect-oriented Java extension.
ATAM	Architecture Trade-off Analysis Method.
ATL	Atlas Transformation Language.
CAMEL	Context Awareness ModEling Language.
CASA	Contract-based Adaptive Software Architecture.
CAUCE	Context-aware Applications for Ubiquitous Computing Environments.
CBSD	Component-based Software Development.
CCA	Component Collaboration Architecture.
CCTV	Closed-Circuit TeleVision.
CIM	Computation Independent Model.
CIV	Computation Independent View.

Notation	**Description**
COCA-ADL	Context-Oriented Component-based Applications Architecture Description Language.
COCA-component	Context-Oriented Component model.
COCA-MDA	Context-Oriented Component-based Applications Model-Driven Architecture.
COCA-middleware	Context-Oriented Component-based Applications Middleware.
COCOMO II	Constructive Cost Model II.
COP	Context-Oriented Programming.
COSD	Context Oriented Software Development.
DAOP	Dynamic Aspect Oriented Programming.
DP	Decision Point.
DPL	Decision PoLicy.
DSL	Domain Specific Language.
ECA	Enterprise Collaboration Architecture.
ECORE	Eclipse CORE meta-model.
EDOC	Enterprise Distributed Object Computing.
EMF	Eclipse Modelling Framework.
JCOOL	Java COntext Oriented Language.
JCOP	Java Context-Oriented Programming.
MADAM	Mobility and ADaptation enAbling Middleware.
MDA	Model Driven Architecture.
MDD	Model Driven Development.

Notation	Description
MOF	Meta Object Facility.
MOSEL	MOdeling Specification and Evaluation Language.
MUSIC	Mobile USers In Ubiquitous Computing.
OMG	Object Management Group.
OOP	Object Oriented Programming.
OSGI	Open Services Gateway initiative framework.
PIM	Platform Independent Model.
PIV	Platform Independent View.
PM	Person-Months.
POI	Places Of Interest.
PSM	Platform Specific Model.
PSV	Platform Specific View.
QoS	Quality of Services.
QR-code	Quick Response Code.
RFID	Radio Frequency IDentification.
SLOC	Source Lines Of Code.
TDEV	Time to Develop.
U-MUSIC	Unanticipated dynamic-adaptation for Mobile USers In Ubiquitous Computing.
UFP	Unadjusted Function Points.

Notation	**Description**
UML	Unified Modelling Language.
VML	Virtual Machine Layer.

Chapter 1

Introduction

Context-dependent applications refer to a class of software systems that are able to monitor and detect context changes in an environment where they operate. They can autonomously modify their own structure and behaviour in response to context changes [Oreizy et al., 1999]. Software in distributed and mobile computing environments needs to cope with variability as software systems are deployed on an increasingly large diversity of computing platforms and operate in different execution environments. Mobility induces context changes to the computational environment and therefore, changes to the availability of resources, and continuously evolving requirements require software systems to be able to adapt to context changes [Inverardi and Tivoli, 2009]. Moreover, because of the software pervasiveness, and in order to make adaptation effective and successful, adaptation processes must be considered in conjunction with dependability and reliability by providing dynamic verification and validation mechanism, which validates the adaptation output with the adaptation goals, objectives, and architecture quality attributes [Cheng et al., 2008, de Lemos et al., 2011].

This thesis contributes to the knowledge by presenting Context Oriented Software Development (COSD), a generic development paradigm for the construction of self-adaptive software from context-oriented components, which enables a runtime composition of context-dependent behaviours and provides the software with capabilities of self-adaptability and dependability in mobile computing environment.

The thesis is that the software developers must considered the context information and context-dependent behaviour in the analysis, design and implementation of self-adaptive software. In particular, software needs to consider the composition of its components in conjunction with the contextual changes, which provides context-driven adaptation and self-adaptability.

This chapter is organized as follows: Section 1.1 highlight the needs for dynamic adaptation of context-aware applications. Section 1.2 provides an overview of the research problems and motivations of engineering self-adaptive software. The research objectives are discussed in Section 1.3. The outline of the solution is presented in Section 1.4. The scope of this research is demonstrated in Section 1.5. Finally, the thesis structure is shown in Section 1.6.

1.1 Background

A self-adaptive and context-dependent software system operating in a highly dynamic world must adjust its behaviour automatically in response to changing environments or requirements, while shifting the human role from operational to strategic. Humans define adaptation goals and new applications or domain requirements, and the system performs all necessary adaptations autonomously at runtime. Throughout the system's life-cycle, including adaptation periods, the system needs to be available and provides functionality to users or other systems, while keeping acceptable levels of Quality of Services (QoS) [Al-Begain, 2004]. However, several researchers have emphasized the need for a new development paradigm that overcomes the complexity, mobility, and variability of this class of applications [Inverardi, 2007, Inverardi and Tivoli, 2009, Baresi and Ghezzi, 2010, Blair et al., 2009, Amoui et al., 2011, de Lemos et al., 2011]. The authors argue that we have to reconceptualize the whole software engineering process for modern software systems, and particularly for the case of self-adaptive systems [de Lemos et al., 2011]. This is only achieved if software is designed to be dynamic and offers adaptability and variability in conjunction with mobility and heterogeneity of the computational environments in which it operates.

In the last few years, Model Driven Development (MDD), Component-based Software Development (CBSD), and context-oriented software have become interesting alternatives for the design and construction of self-adaptive software systems. In general, the ultimate goal of these technologies is to be able to reduce development costs and effort, while improving the modularity, flexibility, adaptability, and reliability of software systems [Clemente et al., 2011]. An analysis of these technologies shows them all to include the principle of the separation of concerns, and their further integration is a key factor to obtaining high-quality and self-adaptable software systems. Each technology identifies different concerns and deals with them separately in order to specify the design and build applications, and, at the same time, provides dynamic behavioural variations autonomously.

The Object Management Group (OMG) [Kleppe et al., 2003] proposes the Model Driven Architecture (MDA), a set of standards that provides a practical implementation of MDD approach. MDA provides a set of guidelines that focus on the explicit separation of platform-independent from platform-specific concerns. In MDA, there are three different views for the software: the computation-independent view (CIV), the Platform Independent View (PIV), and the Platform Specific View (PSV). The Computation Independent View (CIV) focuses on both the environment and the requirements of the system and hides the details of the software structure and processing. The PIV focuses on the operation of the system and hides details that are dependent on the deployment platform. The PSV combines the CIV and PIV, with an additional focus on the details of a specific platform [Kleppe et al., 2003].

CBSD targets the construction of large, high-quality, evolvable software systems in a timely and affordable manner by assembling independent and reusable software modules known as components [Clemente et al., 2011]. A software component is a unit of composition with contractually specified interfaces and explicit context dependences [Szyperski, 2002].

CBSD emphasizes on the separation of concerns among the software modules that encapsulate a set of related functionalities, or, in the case of Service-oriented Architecture [Papazoglou et al., 2007], a component is converted into a service and subsequently inherits further characteristics beyond those of an ordinary component definition.

Context-Oriented Programming (COP) has emerged as a dynamic fine-grained behavioural adaptation, which uses a programming-level techniques for performing the context handling [Gassanenko, 1998, Keays and Rakotonirainy, 2003]. COP has dedicated support for defining and composing variations to basic program behaviour. A variation, which is defined within a layer, can be deactivated/activated for the dynamic extent of a code block. 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 behaviours. 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) [Kiczales et al., 1997]. Dynamic Aspect Oriented Programming (DAOP) has emerged to enforce separation of concerns and support runtime adaptations through weaving code blocks in the application execution [Popovici et al., 2002].

1.2 Runtime Context-Dependent Behaviour Variability Management

Mobile computing environments are heterogeneous and dynamic. Everything from the devices used and resources available to network bandwidths and user context can change drastically at runtime [Belaramani et al., 2003]. This presents the software developers with the challenge of tailoring behavioural variations both to specific user needs and to the context information. The design and development of context-dependent and self-adaptable software applications in mobile computing environments cannot rely on classical software-development methodologies, which assume that the software execution environment is known a priori at design time and that the application environment can be anticipated at the development time [Inverardi and Tivoli, 2009]. Supporting the development and execution of software systems raises numerous challenges, from development processes, design space, and tools for the system's thorough development, to the adaptation mechanisms that ensure adaptability and dependability of the targeted self-adaptive systems [de Lemos et al., 2011]. However, these challenges, taken in isolation, are not new in the software domain. Several approaches attack this domain by providing a model-centric, middleware-centric, or programming-level technique that enables the software to be more context-aware and self-adaptive [Kapitsaki et al., 2009]. This research observes the following challenges in engineering self-adaptive software systems.

1.2.1 Upfront autonomic design.

Engineering a self-adaptive system from scratch requires the support of dynamic and behavioural views of the software design; the software developers should focus on 1) the parts of the application that need to adapt accordingly as a response to a specific context change; and 2) each component must be able to perform the adaptation autonomously. This presents a challenge for the software developers because self-adaptive applications, by their nature, can be seen as the collaboration of individual context-dependent behaviour variations. Context-dependent variations can be seen as a collaboration of individual features spanning the software modules in several places [Hirschfeld et al., 2008], and they are sufficient to qualify as heterogeneous crosscutting in the sense that different code fragments are applied to different program parts [Apel et al., 2006]. Before encapsulating crosscutting context-dependent behaviours into a software module, the developers must first identify the behaviours in the software requirements. This is difficult to achieve because, by their nature, context-dependent behaviours are entangled with other behaviours, and are likely to be included in multiple parts (scattered) of the software modules [Lincke et al., 2011]. Using intuition or even domain knowledge is not necessarily sufficient for identifying their volatile behaviour; instead, a formal analysis procedure is needed for the software requirements and a separation of their individual concerns [Carton et al., 2007].

1.2.2 Runtime variability management.

Mobile computing infrastructures make it possible for mobile users to run software services on heterogeneous and resource-constrained platforms. Heterogeneity and device limitedness create a challenge for the development and deployment of mobile services that are able to adapt to context changes and are able to ensure that users experience the 'best' quality of services possible, according to their needs and specific contexts of use. Thus, it is desirable that self-adaptive software is able to reconfigure and reoptimize itself by recomposing components or services dynamically, according to the operational context [Salehie and Tahvildari, 2009,Salvaneschi et al., 2011]. The assumptions made by the COP approaches proposed in [Gassanenko, 1998,Costanza, 2005,Costanza et al., 2006,Hirschfeld et al., 2008,Salvaneschi et al., 2011], i.e. that the developer knows all the possible software adaptations in advance and designs the application accordingly, is not sufficient to fulfil this need. In addition, in COP and context-aware aspects [Tanter et al., 2006], the context model and the adaptation logic are explicitly hard-coded in the application's business code [Lincke et al., 2011]; this often leads to poor scalability and maintainability [Kapitsaki et al., 2009].

1.2.3 Unanticipated adaptation.

Mobility induces changes in the computational environment and therefore changes in the availability of the allocated resources and services; this often requires the adaptation engine to decide which behaviour must be deactivated/activated in the presence of unforeseen context changes. Unforeseen refers to context changes that cannot be predicted at the design time. Adaptation must be considered in conjunction with dependability, i.e. no matter what adaptation is performed, the software system must continue to guarantee a certain level of quality of services and meet the user's performance needs. Anticipated adaptation is defined by Keeney [Keeney, 2004] as an adaptation behaviour that is foreseen by the developers at development time. On the other hand, semi-anticipated is defined as an adaptation behaviour that can be partially foreseen by the developers during the development process, then a middleware is used for executing an adaptation plan, which specifies application variability models [Rouvoy et al., 2008b]. The assumption is that the developers can provide a preliminary adaptation plan which is able to reason about a certain environmental condition. An unanticipated adaptation is very different and is defined as an adaptation behaviour which incorporates components, possibly from a number of different developers to adjust the application behaviour at runtime [Keeney, 2004, Khan, 2010, Ding et al., 2009]. An unanticipated adaptation requires the software to consider a trade-off between the adaptation goal and the quality attributes of the architecture. To increase the anticipatory capability of a self-adaptive system, code units need to be loaded dynamically i.e. a late binding of a particular component implementation or services. However, the capability of software to perform late binding requires a clear modularization of the self-adaptive software system; this leads us to the next challenge in engineering self-adaptive software using modularity techniques [Cheng et al., 2008, de Lemos et al., 2011].

1.2.4 Modularization.

In the classical view of object-oriented software development, the modular structure for software systems has rested on several assumptions. These assumptions may no longer characterize the challenge of constructing self-adaptive software systems that are to be executed in mobile computing environments [Harrison, 2011]. The most important assumptions in object-oriented development methodologies are that the decision to use or reuse a particular component/object is made at the time the software is developed.

However, the development of a variety of modern self-adaptive software architectures such as mobile/ubiquitous computing, and component-based and context-oriented software has emphasized on deferring these decisions about component selection until runtime. This might increase the software capabilities in terms of variability, adaptability, and maintainability, and increase the anticipatory level of the software by loading a particular component/service that can handle unforeseen context changes dynamically. In addition, modularity can reduce the development effort and increase software comprehensibility [Munnelly et al., 2007]. However, this is not always the case. DAOP supports dynamic weaving of aspects to modify the software behaviour based on the modularity mechanism, supported by the AOP paradigm. Using the AOP paradigm, context information can be handled through aspects that interrupt the main application execution.

The idea behind AOP is to implement crosscutting concerns as aspects whereas the core features are implemented as components. Using pointcuts and advice, an aspect weaver glues aspects and components together. Pointcuts specify the join points of aspects and components, whereas advice define which code is applied to these points. However, designing context-dependent behaviour using aspect oriented programming paradigm requires a platform support for activating aspects driven by the context states. Such implementation requires the AOP platform to evaluate each joinpoint in conjunction with the associated context state and the passive context values. This means that the AOP framework needs to keep track of past context conditions and their associated states [Tanter et al., 2006]. Evaluating each joinpoint with the passive and active context many times leads to poor performance and consumes the allocated resources. Unfortunately, the existing AOP languages tend to add a substantial overhead in both execution time and code size, because all pointcuts must be registered by the aspect framework and parsed each time the advice methods are executed. This restricts their practicality for small devices with limited resources [Hundt et al., 2010]. In addition, it is infeasible for an AOP-based framework to support dynamic de-/activation of collaborative context-dependent behaviours that entangle with each other, as stated by Mezini et al. [Mezini and Ostermann, 2004], Salvaneschi et al. [Salvaneschi et al., 2011], and Lincke et al. [Lincke et al., 2011].

1.2.5 Behavioural composition.

The selection of a particular component at runtime by context-dependent and self-adaptive applications is presumably made based on the active or passive context in addition to the context state and its dependency [Tanter et al., 2006, Lincke et al., 2011], and the possible composition of a context-dependent functionality [Salvaneschi et al., 2011, Hirschfeld et al., 2008], which will exhibit volatile behaviour in the face of context changes. One could ask whether the current software domain techniques have sufficient support or the mechanism for performing such dynamic selection and composition of a software component based on its context-dependent functionality. Behavioural composition and configuration concerns require the software modules to be loosely coupled, and their behaviour variations can be combined and activated autonomously, according to the context changes. Such a challenge has been tackled by a composition strategy that was performed through the development process and depended totally on a static view of the self-adaptive software design. Such a view implies that the developers have to explicitly predict the final composition of the software and possible variations of the application using a programming-level technique such as COP [Schuster et al., 2011] or AOP [Tanter et al., 2006]. In model-based approaches, the composition is produced by proposing application variability models at the design time, then at runtime the middleware uses a utility function to select the best variant to be executed [Geihs et al., 2011]. Alternatively, the composition is shifted one step further to be performed through model-to-model transformation, assuming that a workflow script can provide multiple variations of the application design [Carton et al., 2007].

1.2.6 Platform- or framework-specific adaptation.

The generation of multiple self-adaptive software systems to be implemented on heterogeneous software platforms raises the need for a generic development process that enables the software developer to build a self-adaptive application without being limited to a specific programming framework and/or particular middleware technology. Instead, software developers can use a standard development process and implement the application using an object-oriented programming language that requires a lower level of programmer familiarity and results in less of a gap between the middleware designer and the software developers. Our work, which investigates MDA approaches that target self-adaptive component-based software systems, has found that the real potential behind MDA is not being fully employed either by current MDA-based tools or by the proposed MDA approaches. In addition, MDA-based approaches generate an architecture that is tightly coupled to the middleware or platform technology that they used [Asadi and Ramsin, 2008].

1.3 Objectives and Research Questions

In this thesis, we wish to explore how far we can support the engineering of self-adaptive applications using generic development paradigms provided by non-specialized programming language such as COP, and AOP, and not limited to a specific platform or mechanism. This gives the software developers the flexibility to construct a self-adaptive application using an object-oriented programming language and deploy it on several platforms. Additionally, from the software developer's perspective, it is vital to know the productivity of the development paradigm that might be used in constructing the self-adaptive application.

In addition, it is important to explore the best practices in designing and implementing the adaptation engine (middleware), which is expected to provide adaptability and dependability. Productivity evaluation of the development methodology and the middleware can assist developers in selecting a methodology from those proposed in the literature to achieve adaptability and dependability of the software system. Finally, modularity properties are key determinants of quality in software, so we wish to explore the best decomposition strategy that can facilitate dynamic behavioural composition of the context-dependent variations. Our research has shown that MDD, CBSD, and COP may be appropriately combined to complement each other, improving the development of self-adaptive software systems. However, such combination in the form of COSD is a subject for further investigation and evaluation in terms of software adaptability, variability, and dependability, in conjunction with their expected improvements with respect to software performance and modifiability.

1.3.1 Research questions

To attain these objectives, this thesis addresses the following research question:

Does engineering self-adaptive applications using a decomposition strategy based on the context-dependent functionality support adaptability and variability in several levels of granularity and reduce the development effort from the software developers' point of views? If it does not, which development methodology by its own or combined with others, can be used to develop this class of applications targeting the mobile computing environments and considering the mobility constraints without being limited to a language framework? Which decomposition strategy can facilitate the development of context-aware applications? It is unknown if a better and significant advance in modularity based on context information can break through the complexity of constructing self-adaptive software, and its costs and benefits remain uncovered. What is the efficacy of using the proposed MDA on the development efforts and the developer productivity? Does any model-driven methodologies reduce the development cost and the effort to maintain the application code, or in some cases they will increased the development effort ? What are the pros and cons of implementing the proposed platform in resource constrained environment like mobile devices? What are the optimisation gains from implementing the Context-Oriented Component-based Applications Middleware (COCA-middleware) on comparison to other adaptive middleware architectures?

1.4 Contributions

In order to overcome the problem and the challenges of engineering self-adaptive software, this thesis contributes to the knowledge by presenting COSD methodology. The result of COSD methodology is a component-based architecture described by a Context-Oriented Component-based Applications Architecture Description Language (COCA-ADL). COCA-ADL is a platform-independent model transformed by a tool support into the desired platform-specific model. This provides code mobility for the same application into various deployment platforms.

The context-oriented component model encapsulates the implementation of the context-dependent parts in distinct architectural units, which enables the software to adjust its functionality and/or behaviour dynamically. This differs from the majority of existing work, which seek to embed awareness of context in the functional implementation of applications. The context-oriented software is developed using a Context-Oriented Component-based Applications Model-Driven Architecture (COCA-MDA). Afterwards, the context-oriented software is manipulated at runtime by a COCA-middleware that performs a runtime behavioural composition of the context-dependent functionality based on the operational context. The self-adaptive software dependability is achieved through the COCA-middleware capability in considering its own functionality and the adaptation impact/costs. A dynamic decision-making based on a policy framework is used to evaluate the architecture evolution and verifies the fitness of the adaptation output with the application's objectives, goals and the architecture quality attributes. A case study application is implemented and deployed in an IPhone device as proof of concept. The implementation demonstrates the application's ability to modify its behaviour based on the execution context. The implementation demonstrates how the platform maintains the quality attributes by adapting a self-tuning and self-configuring mechanisms in response to multiple context changes.

The self-adaptive context-oriented component-based software has been evaluated in two distinct phases. The first phase includes pre-implementation evaluation using the Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 2002]. The objective of this evaluation is to consider the COCA-middleware architecture's sensitivity points and the trade-off among the quality attributes, which the COCA-middleware design must maintain. This provides evidence that the proposed middleware architecture does maintain the quality attributes by trade-offs among the quality attributes while achieving the adaptation goals. The second phase evaluates the Context-Oriented Software Development paradigm and the COCA-middleware and the case study in terms of the architecture performance and modifiability. The evaluation compared the performance gain from implementing a case study application using the COSD paradigm and COCA-middleware with several approaches proposed in the literature like Aspect Oriented Software Development (AOSD) [Filman et al., 2004] and Context-Oriented Programming [Appeltauer et al., 2008, Schuster et al., 2011]. In addition, the performance of COCA-middleware implementation was evaluated with other middleware architectures proposed in the literature.

Finally, a state-of-the-art case study was selected to demonstrate the COCA-MDA capability in facilitating the development of a self-adaptive and context-dependent application. The development of the case study demonstrates how the methodology adapts a policy framework for achieving and assuring the adaptation results among unforeseen changes. In addition, it demonstrates how the modularization technique can reduce the complexity of the self-adaptive application design. The Constructive Cost Model II (COCOMO II) [Boehm et al., 2000] is used to evaluate the impact on the development cost of using the COCA-MDA methodology. It shows how COCA-MDA does reduce the required development effort compared to other MDAs. It also demonstrates how COCA-MDA has reduced the software maintenance ratio through the architecture deployment and transformation.

1.5 Scope

This thesis focuses on studying dynamic software composition using model-driven development, component-based software and a dedicate adaptive middleware. On another hand, there are several approaches in the literature that target self-adaptability and dependability of software systems. The model-free adaptation is one of them. In this approach the mechanism does not have a predefined model for the environment and the system itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the system. Such an approach is out of the scope of this thesis, because it focuses more on collaborative reinforcement learning mechanism, which enables groups of reinforcement learning agents to solve system optimisation problems in dynamic and decentralized networks.

In addition to that, context-awareness can be achieved via ontology model and rule-based engine. The ontologies usage exploits the principles of the semantic web to produce ontologies that describe context information and its associations and provide means for reasoning and inference. We have also omitted this category from our study, because ontologies usually model context information specific to a chosen application domain.

Regarding rule-based reasoning approaches, a rule-based system is a combination of a number of rules and a set of activation conditions for these rules. This category has not been analysed further in this thesis, since only few and quite early studies are presented with respective approaches specific to context management for services adaptation [Daniele et al., 2007].

1.6 Road Map

The reminder of this thesis is structured as follows:

Chapter 2 summarises and discusses the current state-of-the-art approaches for self-adaptive software and their limitations, and also identifies critical challenges that arise when engineering self-adaptive systems. Context adaptation approaches and techniques are discussed in this chapter in terms of the modelling approach, self-adaptability requirements, and run time infrastructure.

Chapter 3 proposes a development methodology, that applies to the Model-driven Architecture style. The COCA-MDA methodologys phases and tasks are described in detail and show the process of constructing a case study application.

Chapter 4 describes the implementation of the COCA-middleware and the case study applications.

Chapter 5 provides results of architecture evaluation. The proposed architecture is evaluated using the ATAM method, which has been adapted and used to evaluate the COCA-middleware capability in terms of several quality attributes. In addition, the COCA-middleware is evaluated with other architectures proposed in the literature in terms of its adaptability and modifiability.

Chapter 6 demonstrates the capabilities of COCA-MDA in supporting the development of context-aware applications by describing a state-of-the-art case study and evaluating the development effort involved in adapting the COCA-MDA in constructing the application.

Chapter 7 presents conclusions and outlines possible future works.

1.7 Summary

This chapter highlights several challenges in building self-adaptive software from a conceptual view to the level of runtime infrastructure middleware components. In order to support context-binding mechanisms, a platform solution is needed to legitimately deal with the heterogeneity of context-dependent behaviours. The results of the decomposition mechanism are a clear, modularised, component model, which separates the context-dependent from context-independent concerns.

Chapter 2

Engineering Self-adaptive Software

Supporting the development and execution of self-adaptive software systems raises numerous challenges. These challenges include the development processes for building them, the design space, which describes the design patterns and the best practices of designing their building blocks, i.e. component model or code fragments. The adaptation mechanism, which describes the best adaptation action that can be used under the limited resources of the execution environment.

The proposed approaches in the literature can be classified into model-centric, middleware-centric, and programming-level techniques, as shown in Figure 2.1. The ultimate goal of these approaches was to support adaptability, variability and increase the software quality by managing the context-dependent functionality at the programming level, middleware layer, or architecture model. In addition to that, they were trying to provide an adaptation mechanism, that have less impact on the allocated resources under the mobility constrains of the execution environments.

This chapter summarises and discusses the current state-of-the-art approaches for engineering self-adaptive software systems. Specifically, we intend to focus on the management of the context-dependent concerns, the model-driven development methodologies, component-based software development, and middleware centric-development that support dynamic self-adaptive software. In addition, self-adaptability assurance and verification approaches are discussed, as they tend to support runtime verification and validation mechanisms of the adaptation output.

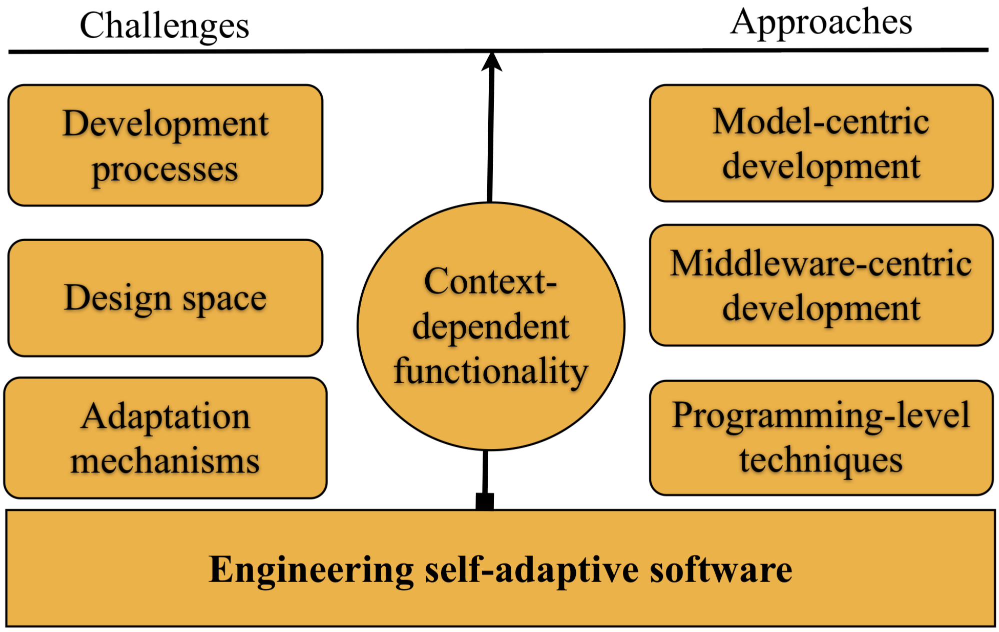

Challenges
Approaches
Development processes
Design space
Adaptation mechanisms
Context-dependent functionality
Model-centric development
Middleware-centric development
Programming-level techniques
Engineering self-adaptive software

Fig. 2.1: Self-adaptive software challenges and approaches

However, self-adaptive software and their self-* properties are defined in Section 2.1. Section 2.2 discusses behavioural variability support in the programming level. Section 2.3 addresses the challenges of modelling self-adaptive software. A comparison of the related work on model-based adaptation is presented in Section 2.4. Section 2.5 focuses on component-based adaptive applications. Section 2.6 addresses the middleware support for self-adaptive software. Self-adaptability assurance and runtime verification is discussed in Section 2.7.

2.1 Self-adaptive Software

There is a growing demand for developing applications with aspects such as context awareness and self-adaptive behaviours. Context awareness [Parashar and Hariri, 2005] means that the system is aware of its context, which is its operational environment. Hirschfeld et al.

[Hirschfeld et al., 2008] considered context to be any information that is computationally accessible and upon which behavioural variations depend. A context-dependent application adjusts its behaviour according to context conditions arising during execution. A self-adaptive application modifies its own structure and behaviour in response to changes in its operating environment [Oreizy et al., 1999].

Self-adaptive software has certain characteristics, known as the autonomic properties or the self-* properties which include self-organising, self-healing, self-optimising and self-protecting [Horn, 2001].

Self-organising is the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing/decomposing software entities [Salehie and Tahvildari, 2009].

Self-healing is the capability of discovering, diagnosing and reacting to disruptions. It can also anticipate potential problems, and accordingly take suitable actions to prevent a failure [Robertson and Laddaga, 2005, Kuwadekar et al., 2010].

Self-optimising, which is also called self-tuning or self-adjusting [Hinchey and Sterritt, 2005], is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end response time, throughput, utilisation and workload are examples of important concerns related to this property. Self-protecting is the capability of detecting security breaches, anticipating problems and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating security problems by taking actions to avoid them or mitigate their effects [Salehie and Tahvildari, 2009]. Self-awareness, self-monitoring, self-situated and context-awareness are the underlying primitive properties of the self-autonomic properties based on the autonomic computing paradigm specified by Horn [Horn, 2001].

Some other properties were also mentioned at this level, such as openness and anticipatory [Parashar and Hariri, 2005], which are optional. Self-awareness [Hinchey and Sterritt, 2005] means that the system is aware of its self states and behaviours. This property is based on self-monitoring, which reflects what is monitored.

Implementing a self-adaptive software system in a resource-poor environment faces a wide rang of variance in platforms' specifications and Quality of Services (QoS) [Kuwadekar et al., 2010]. Mobile devices have different capabilities in terms of CPU, memory, and network bandwidth [Kuwadekar et al., 2008]. Everything from the devices used and resources available to network bandwidths and user context can change drastically at runtime [Belaramani et al., 2003].

In general, architecture-driven dynamic system adaptation has several challenges such as maintaining the correspondence between architectural models and system implementation in order to ensure that architecture-based adaptation is appropriately executed. The second issue is providing the necessary evolution facilities in the implementation infrastructure. An appropriate way to study the challenges is to classify them on the basis of adaptation features that they support and how they manage software variability in the architecture level [Salehie and Tahvildari, 2009]. In the following sections, several approaches that target engineering self-adaptive software system are discussed.

2.2 Variability Management with Context-oriented Programming and Aspects

Compositional adaptation enables an application to adapt its structure or behaviour for anticipating concerns that were unforeseen during its original design and implementation. Normally, compositional adaptation can be achieved using the separation of concerns technique, computational reflection, component-based design, and adaptive middleware [McKinley et al., 2004]. The separation of concerns enables the software developers to separate the functional behaviour and the crosscutting concerns of self-adaptive applications. The functional behaviour refers to the business logic of an application [McKinley et al., 2004]. Context-driven behavioural variations are heterogeneous crosscutting concerns and a set of collaborating aspects that extend the application behaviour in several parts of the program and have an impact across the whole system. Such behaviour 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 behaviour of an application as the base-component that provides the user with context-free functionality. On the other hand, context-dependent behaviour variations are considered as crosscutting concerns that span the software modules in several places.

Context-Oriented Programming (COP) is an emerging technique that enables context-dependent adaptation and dynamic behaviour variations [Gassanenko, 1998,Keays and Rakotonirainy, 2003]. In COP, context can be handled directly at the code level by enriching the business logic of an application with code fragments responsible for performing context manipulation, thus providing the application code with the required adaptive behaviour [Salehie and Tahvildari, 2009].

Costanza et al. [Costanza et al., 2006] proposed the design of context-aware systems following a layered approach. The term " layer " refers to a specific context-dependent functionality, which might include a partial implementation of a class or a set of methods [Costanza et al., 2006], or the whole class is encapsulated inside a layer [Hirschfeld et al., 2008]. Hirschfeld et al. argued that the class-in-layer approach is more effective than the layer-in-class approach for encapsulating the context-dependent functionality, starting from the claim that context-dependent behavioural variations occur separately or in any combination, and in most cases they are collaborating and entangled with each other. A layer can be dynamically activated and composed with other layers, allowing fine-grained control of an application's runtime behaviour [Hirschfeld et al., 2008]. An example for using COP for implementing context-aware applications was proposed by Schuster et al. [Schuster et al., 2011]. Schuster et al. proposed Java Context-Oriented Programming (JCOP), which uses a layered approach for achieving behavioural de-/activation for a prototype mobile application. The application was implemented based on a simple context model, which was implicitly encoded with the application code. Such approach, shows the feasibility to use COP for implementing self-tuning context-aware application for mobile computing environments.

COP supports context handling internally on the source code, which supports behavioural variations using the layered approach mentioned above. The assumption made by the COP prototype is the context information is processed and delivered by the infrastructure and the focus was on providing an add-hoc mechanism of the context-dependent behaviour at the programming level. However, supporting the self-* autonomic properties requires the software systems to support the MAPE-K autonomic computing loop, which includes monitoring, analysing, planning and executing [Salvaneschi et al., 2011]. This requires the context-oriented software to have a runtime infrastructure that supports context monitoring, detection, deciding, and acting processes.

In the JCOP approach proposed by Schuster et al. [Schuster et al., 2011], the developers have to predict all possible behaviour inside the source code. As an outcome, the anticipated adjustment is restricted to the amount of code stubs on hand offered by the creators [Kapitsaki et al., 2009]. In this case, the amount of behavioural variations introduced in the application is limited by the developer's ability to predict the piece of code that might extend the application behaviour [Salvaneschi et al., 2011]. On the other hand, it is impractical to forecast all likely behaviours and program them at the source code. In addition, this method does not separate the adaptation mechanism from the application's business logic, which provides poor scalability and maintainability [Kapitsaki et al., 2009]. Furthermore, the context model is implicitly coupled inside the source code, which requires rewriting and recompiling the code whenever a new context-dependent behaviour or context provider is introduced to the application's platform. To a certain degree, it is very difficult for the developers to decide when and where the context-dependent behavioural variations are needed, particularly in the presence of unanticipated contextual changes that were unforeseen during the original design and construction of the COP software.

For more complex context-aware systems, the same context information would be triggered in different parts of an application and would trigger the invocation of additional behaviour. 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) [Kiczales et al., 1997]. 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 [Gassanenko, 1998, Keays and Rakotonirainy, 2003], the context-dependent behavioural variations and the context monitoring and handling must be addressed in separate aspects. Unfortunately, the aspect-oriented development methodology can be used to handle homogeneous behavioural variations where the same piece of code can be invoked in several software modules [Apel et al., 2006, Mezini and Ostermann, 2004], and it does not support adaptation of aspects to context in what is called context-driven adaptation [Kapitsaki et al., 2009].

Context-aware aspects proposed by Tanter et al. [Tanter et al., 2006], which provides these functionality by designing pointcuts which depend on different contexts, so that advices would only be executed in specific context conditions. Current AOP languages are limited regarding to context condition expression. First, they are not able to consider past context. Second, they are not able to express context-dependencies in aspects. Designing aspects that become active when particular contexts are verified, require the possibility to refer to a context definition in a pointcut construction. This means that the AOP framework should allow the programmers to specify the context condition in the syntax of joinpoint definitions. For example, BeInContext(Context LocationCtx) would allow the programmer to react to context changes based on the user's location.

Another important ability of a framework should be giving an overview about all actual and past activated contexts, so that pointcuts can be designed on the base of this information. This means that the AOP framework needs to keep track of past context conditions and their associated states. This is called context snapshotting [Tanter et al., 2006], and the saved state of one context condition at a given point of time is called context snapshot. A global context snapshot is therefore a snapshot of all context conditions at a given point in time.

Context snapshots are only made at a special point of time, because otherwise it would lead to high memory problem. So the main problem is to define the right points of time to take such context snapshots. The actual current solution is to take snapshots of context conditions only if necessary as stated in the Reflex framework [Tanter, 2006]. However, anticipating context changes at runtime may require new behaviour or functionality to be invoked in the application execution and a runtime composition of several collaborating aspects. Another approach supported by AOP is called Dynamic Aspect Oriented Programming (DAOP) [Popovici et al., 2002].

Dynamic weaving of aspects can be used for adjusting the software behaviour at runtime. However, existing DAOP techniques tend to add a substantial overhead in both execution time and code size, which restricts their practicality for small devices with limited resources [Hundt et al., 2010]. Researchers in the DAOP community keep building self-adaptive software using AOP, and they were relying too much on optimising the performance of AOP frameworks in the Virtual Machine Layer (VML) [Hundt et al., 2010]. The major reason for this poor performance is that the DAOP architectures like PROSE 2 [Popovici et al., 2002] provide an AOP engine running at VML. This engine accepts aspects at runtime, then transforms them into basic entities like joinpoint requests. The joinpoints are activated by registering them to the execution monitor. When the execution reaches one of the activated joinpoint, the execution monitor notifies the DAOP engine, which execute the advice method.

In general, source-code approaches can handle context information directly at the code level. However, in many instances, the delivery of new context-dependent behaviour or introducing a new context provider to the platform require the developers to start a new software engineering process, which includes analysing, designing and implementing the software systems [Salehie and Tahvildari, 2009]. In AOP and COP the whole set of context models and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability [Kapitsaki et al., 2009].

2.3 Modelling Self-adaptive Applications

In recent years, model-driven architecture techniques have been applied by both researchers and developers. A significant number of model-driven architecture approaches were proposed for the construction of context-aware and self-adaptive applications. In most cases, they have adapted the Unified Modelling Language (UML) to represent the software model. The UML model can be used in different domains of interest when supported by a model-to-model or model-to-code transformation, the so-called Model Driven Development (MDD).

The Object Management Group (OMG) presented a set of guidelines Model Driven Architecture (MDA) for building software systems based on the use of the MDD methodology [Kleppe et al., 2003]. MDA focuses primarily on the functionality and behaviour of a distributed application or system across platforms. With MDA, the functionality and behaviour are modelled once and only once. Thus, MDA defines the notions of a Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). CIM describes the software requirements in computational free fashion. A PIM describes the parts of a solution that do not change from one platform to another, and a PSM includes descriptions of parts that are platform dependent [Kleppe et al., 2003].

The Enterprise Collaboration Architecture (ECA) [ECA OMG, 2004], is another standard presented by the OMG. ECA aims to provide a development methodology to simplify the development of the component-based system, using the Enterprise Distributed Object Computing (EDOC), by means of a modelling framework and by conforming to the OMG's MDA. The Component Collaboration Architecture (CCA) describes how to model the structure and the behaviour of components at varying and mixed levels of granularity. The components' structure and behaviour are defined by partitioning the system specification into several viewpoints. The application's architecture is described by a recursive decomposition and assembly of parts that enables its application to several domains. The Entities, Events, and Business Process Model are a set of UML models proposed by ECA to define platform-independent models of component-based software systems [ECA OMG, 2004]. The following sections are describing several MDA-based approaches, that were proposed for the construction of context-aware and self-adaptive software system using a component model and an adaptive middleware.

2.3.1 Model Driven Development and AOP

Carton et al. [Carton et al., 2007] proposed the Theme/UML, a model driven approach supported by aspect-oriented programming in an attempt to model various crosscutting concerns of context-aware applications at an early stage of the software development. Theme/UML provides a systematic means to analyse the requirements' specification in order to identify base and crosscutting concerns, and the relationships between them. However, to the best of our knowledge, there is no similar approach that can help the developers to analyse and understand the context-dependent behaviours in the requirements, design and implementation of the self-adaptive applications.

The Theme/UML approach was based on the use of the Meta Object Facility (MOF) extension and the Eclipse CORE meta-model (ECORE) [Eclipse, 2010]. The MOF meta model for the development of context-aware mobile applications proposed by de Farias et al. [de Farias et al., 2007] was structured according to the core and service views of the software system. This approach provides a contextual model that is independent from the application domain. However, it does not provides high level abstraction of the software models, which express conceptual characteristics of the context-dependent behaviours. From a software developer's perspective, it does not take into account architectural or deployment issues, because it is based on the service-oriented architectures. In addition, it has focused on the model-to-model transformation for generating the software composition. Such approach adds substantial overhead over the development for writing and configuring the MOF scripts. The Theme/UML methodology limits the development of self-adaptive applications to a very specific framework that supports the Aspect-oriented Java extension (AspectJ) and Eclipse Modelling Framework (EMF) [Kiczales et al., 2001]. Extending this paradigm for another platform requires a specific compiler that supports AOP and toolset that follow the EMF.

Plastic is another development approach, which uses the MDD paradigm for developing and deploying adaptable applications, implemented in Java language [Inverardi and Tivoli, 2009]. The Plastic development process focuses on the model verification and validation and service composition of java service stubs. The methodology shows a very interesting feature of runtime model verification and validation mechanism. Unfortunately, the generated software is tightly coupled with the target deployment platform and cannot be used with a standard development process supported by a standard object-oriented language other than the JAVA and AspectJ languages. However, the two paradigm Theme/UML and Plastic face challenges with regard to the model manipulation and management. These challenges arise from problems associated with (1) defining, analysing, and using model transformations, (2) maintaining traceability links between model elements to support model evolution and roundtrip engineering, (3) maintaining consistency among viewpoints, (4) tracking versions, and (5) using models during runtime [France and Rumpe, 2007].

2.3.2 A-MUSE

An MDA-based approach for behaviour modelling and refinement is introduced by Daniele et al. [Daniele et al., 2009]. Daniele et al. proposed the Architectural Modeling for Service Enabling in Freeband (A-MUSE) approach, which focuses on the decomposition of the PIM model into three levels; each level is used to automate a behavioural model transformation process. Daniele et al. [Daniele et al., 2009] applied their approach to a Mobile System Domain Specific Language (DSL) (called M-MUSE). Therefore, the platform independent design phase has been decomposed into the service specification and platform-independent service design steps. The platform-independent service design model should be a refinement of the service specification, which implies correctness and consistency, particularly of behavioural issues, which have to be addressed in the refinement transformation. However, when trying to realize this refinement transformation, a gap between service specification and platform-independent service design was wide, so that correctness and consistency were hard to guarantee in a single refinement transformation. The authors approach this problem by proposing multiple rounds of transformation between the PIM and PSM, which requires the developers to switch simultaneously between the PIM, PSM and the service specifications several times.

2.3.3 CAMEL

Context Awareness ModEling Language (CAMEL) is an MDD-based approach proposed by Sindico and Grassi [Sindico and Grassi, 2009]. The approach uses a domain-specific language called Java COntext Oriented Language (JCOOL), which provides a metamodel for context sensing with the supports of the context model designed using the JCOOL meta model. However, Sindico and Grassi implemented the context binding as the associate relationship between context value and context entity. On the other hand, context-driven adaptation refers to a structure or behaviour elements, which are able to modify the behaviour based on context values. The structural or behavioural insertion is accomplished whenever a context value changes; it uses AOP inter-type deceleration, where the behavioural insertion is accomplished by means of an AOP advice method to inject a specific code into a specific joinpoint.

The CAMEL paradigm provides insufficient details with regard to the underlying component model or the application architecture. The authors used their former domain-specific language to support the COP approach proposed by Hirschfeld et al. [Hirschfeld et al., 2008]. Moreover, CAMEL has no formal MDD methodology that possesses a generic life cycle that a developer can use. Irrespective of these problems, JCOOL is specific to an AOP framework called the Simple Middleware Independent LayEr (SMILE) [Bartolomeo et al., 2008]. SMILE platform used for distributed mobile applications [Bartolomeo et al., 2008]. The model approach in JCOOL supports only ContextJ, which is an extension of the Java language proposed by Appeltauer et al. [Appeltauer et al., 2009]. The CAMEL methodology requires the software to be re-engineered whenever a new context provider is introduced into the context model. The developers must build a complete context model for the new values and maintain the underlying JCOOL DSL and the UML model. The CAMEL methodology has adapted AOP and the EMF to produce a context-oriented software similar to the layered approach proposed by Hirschfeld et al. [Hirschfeld et al., 2008]. This makes CAMEL limited to the EMF tool support and the ContextJ language [Appeltauer et al., 2011]. From our point of view CAMEL tightly coupled the software with modelling language, modelling tool and the target deployment platform.

2.3.4 MUSIC MDD

The Mobile USers In Ubiquitous Computing (MUSIC) development methodology [Floch et al., 2006, Rouvoy et al., 2008a, Rouvoy et al., 2009] adapts a model-driven approach to construct the application variability model. In MUSIC, applications are built using a component framework, with component types as variation points. The MUSIC middleware is used to resolve the variation points, which involves the election of a concrete component as a realization for the component type. The variability model defines the component types involved in the application's architecture and describes their different realizations. This comprises either a description of collaborating component types and rules for a composite realization, or a reference to a concrete component for an atomic realization. To allow the realization of a component type using external services, the variability model also includes a service description, which is used for service discovery.

The software architecture in MUSIC is a pluggable architecture for self-adaptive applications. It proposes middleware featuring a generic and reusable context management system. The architecture supports context variation and resource utilization by separating low-level platform-specific context from higher-level application-specific concerns. The resource utilization is improved through intelligent activation and deactivation of context-related plug-ins based on the needs of the active application. The MUSIC middleware architecture defines multiple components that interact with each other to seamlessly enable self-adaptive behaviour in the deployed applications. These components include context management, adaptation reasoner, and a plug-in life-cycle management based on the Open Services Gateway initiative framework (OSGI) [OSGI framework, 2010].

At runtime, a utility function is used to select the best application variant; this is the so-called 'adaptation plan'. The utility function is defined as the weighted sum of the different objectives based on user preferences and QoS. Realistically, it is impossible for the developer to predict all possible variations of the application when unanticipated conditions could arise. In addition, mobile computing devices have limited resources for evaluating the many application variations at runtime and can consume significant amounts of device resources.

As an outcome, the benefit gained from the adaptation is negated by the overhead required to achieve the adaptation [Salehie and Tahvildari, 2009].

2.3.5 Paspallis MDD

Paspallis [Paspallis, 2009] introduced a middleware-centric development of a context-aware applications with reusable components. Essentially, his work is based on the MUSIC platform [Reichle et al., 2008]. According to Paspallis, an MDA-based context-aware application is built by separating the concerns of the context provider from those of the context consumer. For each context provider, a plug-in or bundle is planned and designed during the design phase. At runtime, a utility function is used to consider the context state and perform decision making. Once the plug-in is selected to be loaded into the application, middleware support performs dynamic runtime loading of the plug-in.

However, it is impossible for the developers to predict all the context providers that might produce context information at runtime. In addition, using this methodology means that the developer is required to design a separate plug-in architecture for each context provider, which is proportional to the available number of context providers. Additionally, this methodology does increase the development effort as each plug-in requires an separate development process.

2.3.6 U-Music MDD

Khan [Khan, 2010] proposed Unanticipated dynamic-adaptation for Mobile USers In Ubiquitous Computing (U-MUSIC) methodology. U-MUSIC adapts a model-driven approach to constructing self-adaptive applications and enabling component model-based, unanticipated adaptation. However, the author has modified the MUSIC methodology to support semi-anticipated adaptation; also called planning-based adaptation, which enables the software to adapt among foreseeable context changes. U-MUSIC enables developers to specify the application variability model, context elements, and data structure. The developers are able to model the component functionalities and quality of service (QoS) properties in an abstract, platform-independent way. In U-MUSIC, dynamic decision-making is supported by the MUSIC middleware mentioned above. However, this approach suffers from a number of drawbacks. First, it is well-known that correct identification of the weight for each goal is a major difficulty for the utility function. Second, the approach hides conflicts between multiple goals in its single, aggregate objective function, rather than exposing the conflicts and reasoning about them. It would be optimistic to assert that the process of code generation from the variability models can become completely automatic or that the developer's role lies only in application design.

2.3.7 CAUCE

Context-aware Applications for Ubiquitous Computing Environments (CAUCE) proposed as a model-driven development approach [Tesoriero et al., 2010]. The authors defined an MDA approach that focuses on three layers of models. The first layer confirms to the computational independent model for capturing the conceptual properties of the applications. The second layer defines three complementary points of view of the software systems. These views include deployment, architecture and communication. The third layer focuses on converting the conceptual representation of the context-aware application into a software representation using a multi model transformation. The Atlas Transformation Language (ATL) is used to interpret the model and convert them into a set of models conforming to the platform independent model. The final model is transformed using the MOF Script language based on the EMF paradigm [Eclipse, 2010]. The CAUCE methodology focuses more on the CIM by splitting this layer into three layers of abstraction, which confirms to the tasks, social and space meta models. The task model focuses on modelling a set of tasks and the relationships among them that any entity in the system is able to perform. The social meta model defines the social environment of the entities in the system and is directly related to the entity task and entity information that identify of the context-aware application behaviour. The space meta model defines the physical environment of the entities in the system. Therefore, this meta model is directly related to the physical conditions, infrastructure and location characteristics of the context-aware applications.

However, the CAUCE methodology provides a complete development process for building context-aware applications. Despite that, CAUCE is limited to specific modelling tool and language, in this case the UML is integrated with EMF. The generated application can only be implemented using Java language as it is supported by the ATL and MOF Script languages. However, it is impossible for the developers to adapt CAUCE for building heterogeneous and distributed mobile applications, which might have multiple deployment platforms and requires variant implementation languages.

2.3.8 ContextUML

Generally, UML profiles and metamodels are used to extend the UML language semantics. ContextUML was one of the first approaches that targeted the modelling of the interaction between context and web service applications [Sheng and Benatallah, 2003]. ContextUML was extended by Prezerakos et al. [Prezerakos et al., 2007], using aspect-oriented programming and service-oriented architecture to fulfil the user's needs. and context awareness. Another UML profile that is similar to ContextUML has been proposed by Grassi and Sindico [Grassi and Sindico, 2007]. However, contextUML used a UML metamodel that extended the regular UML by introducing appropriate artifacts that created a context-aware application. A class diagram is produced, which corresponds to the context class and to specific services. They mitigate the UML relationship and dependency to express the interaction between the context information and the respective services. A means of parameter injection and service manipulation are used to populate specific context-related parameters in the application execution loop.

However, the UML profiles and metamodels lack from several features required for modelling the self-adaptive software system. Ignoring the heterogeneity of the context information, they based their claims on the nature of the context values, which can fluctuate and evolve significantly at runtime. It is not feasible to this study how the behaviour is modelled when multiple context values have changed at the same time.

2.4 Feature Analysis and Comparative Study of MDA-based Approaches

From the software developer's perspective, it is vital to know the features of the development paradigm which might be used in constructing a self-adaptive application. Feature evaluation of the development methodology can assist the developers in selecting among the proposed methodologies in the literature for achieving adaptability and dependability of the software systems. Improving the development of self-adaptive software systems using model driven approach has attained several research efforts. The target was in general to introduce software with adaptability and variability while focusing on reducing the software complexity and optimising the development effort.

The examination of software system performance, dependability and availability is of greatest importance for tuning software system in conjunction with several architecture quality attributes. Such performance analysis was considered by the MOdeling Specification and Evaluation Language (MOSEL) [Begain et al., 2001]. The system modelling using MOSEL illustrates how easily it can be used for modelling real-life examples from the fields of computer communication and manufacturing systems. However, extending the MOSEL language towards the modelling and performance evaluation of self-adaptive software system can estimate several quality attributes of model-based architecture and provides early results about how efficient is the adaptation action.

Kitchenham et al. in [Kitchenham et al., 2002] proposed the DESMET method, which evaluates software development methodologies using an analytical approach. Asadi et al. have adapted the DESMET method to analyse several MDA-approaches. The authors adapted several evaluation criteria that can be used to compare MDA methodologies based on MDA-related features and MDA-based tool features [Asadi and Ramsin, 2008].

However, Calic et al. [Calic et al., 2008] proposed an evaluation framework to evaluate MDA-based approaches in terms of four major criteria groups, as follows: I) MDA-related features: The degree to which the proposed methodologies are compliant with OMG's MDA specification [OMG, 2010]. II) Quality: Evaluation of the overall quality of the MDA-based approaches including their efficiency, robustness, understandability, ease of implementation, completeness, and ability to produce the expected results [Kitchenham et al., 2002]. III) Usability: Simplicity of use and ease of implementation by the developer, which covers clear information about the impact of the methodology on the development effort [Norman, 2002, Preece et al., 2002]. IV) Productivity: The quality of benefits derived from using the methodology and its impact on the development time, complexity of implementation, code quality, and cost effectiveness [Calic et al., 2008]. Calic at al. [Calic et al., 2008] presents the COPE tool, to evaluate the MDA productivity by automate the coupled evaluation of metamodels and model by recording the coupling history in an history model.

Lewis et al. [Lewis and Wrage, 2005] have evaluated the impact of MDA on the development effort and the learning curve of the MDA-based development tools based on their own experiences. The authors concluded that the real potential behind MDA is not completely employed either by current tools or by the proposed MDA approaches in the literature. In addition, the developers have to modify the generated code such that it is suitable for the target platform. The degree to which the generated code needs modification is affected by the MDA tools used. In the same way, the developer's understanding of the MDA tasks and familiarity with the target platform have direct impacts on MDA productivity.

The Constructive Cost Model II (COCOMO II) [Boehm et al., 2000] emerged as a software cost estimation model which considers the development methodology productivity. The productivity evaluates the quality of benefits derived from using the development methodology, in terms of its impact on the development time, complexity of implementation, code quality, and cost effectiveness [Calic et al., 2008]. COCOMO II allows estimation of the effort, time, and cost required for software development. The main advantage of this model over its counterparts such as the Software LIfe-cycle Management (SLIM) model [Estell, 1976] and the System Evaluation and Estimation of Resources Software Estimation Model (SEER-SEM) [Galorath and Evans, 2006] is that COCOMO II is an open model with various parameters which effect the estimation of the development effort. Moreover, the COCOMO

II model allows estimation of the development effort in Person-Months (PM) and the Time to Develop (TDEV) a software application. A set of inputs such as software scale factors (SF) and 17 effort multipliers is needed. A full description of these parameter is given in the COCOMO II model definition manual, which can be found in [Boehm et al., 2000]. An example of an evaluation of MDA approaches with COCOMO II can be found in [Achilleas, 2010].

In this research, we intend to use an evaluation framework that can test and qualify the ability of MDA-based approaches to produce the expected results [Kitchenham et al., 2002] in terms of dynamic adaptation in general, and self-adaptability, in specific. These features are evaluated in the following sections.

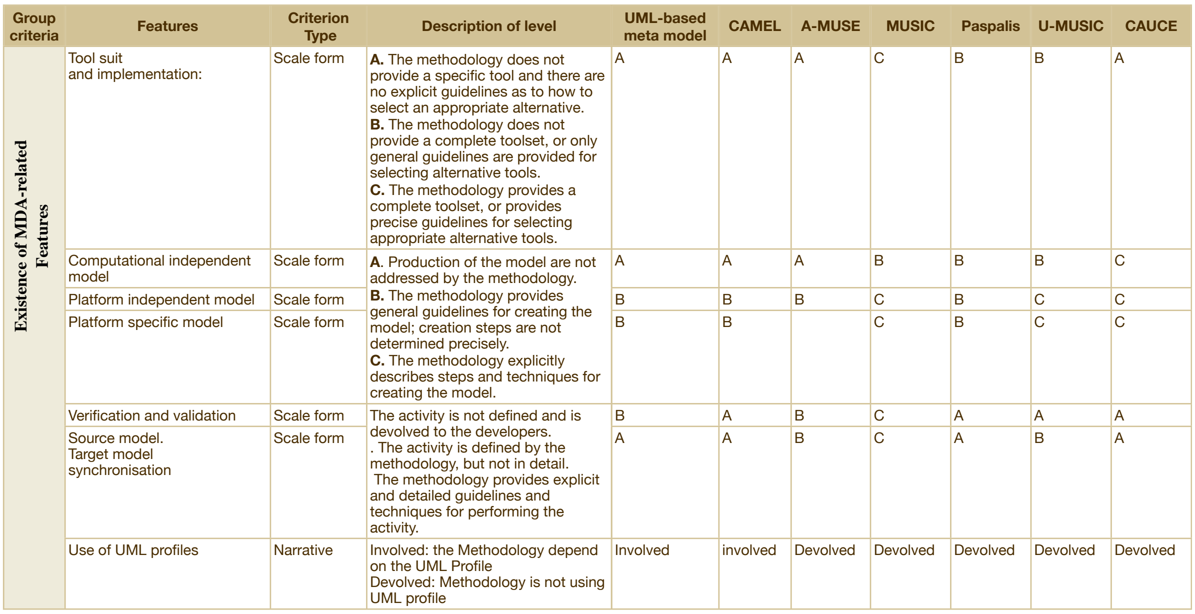
2.4.1 Existence of MDA-related Features

MDA features refers to the degree to which the proposed methodologies are compliant with the OMG's MDA specifications; these specifications can be divided into the support of CIM, PIM, PSM, model validation, and transformation [OMG, 2010]. In terms of MDA features, we adapt the criteria proposed by Asadi and Ramsin [Asadi and Ramsin, 2008], which highlights the methodology's conformance to the original OMG standard, as shown in Table 2.1. Feature analysis can be performed in two ways: scale form and narrative form. The scale form attaches the methodology complaint to a specific feature, which is divided into three ranks, from A to C, as shown in each table. The narrative form captures whether the methodology covers a specific feature based on the level of involvement.

2.4.2 Tool-related feature analysis

The major challenges that developers face when attempting to realize the MDD vision is the selection of a modelling language and modelling tool. Modelling languages challenges arise from concerns associated with providing support for creating and using an appropriate modelling abstraction for analysing and designing the software [France and Rumpe, 2007]. A second challenge posed by Asadi and Ramsin [Asadi and Ramsin, 2008], that each de-

Table 2.1: MDA-related criteria evaluation

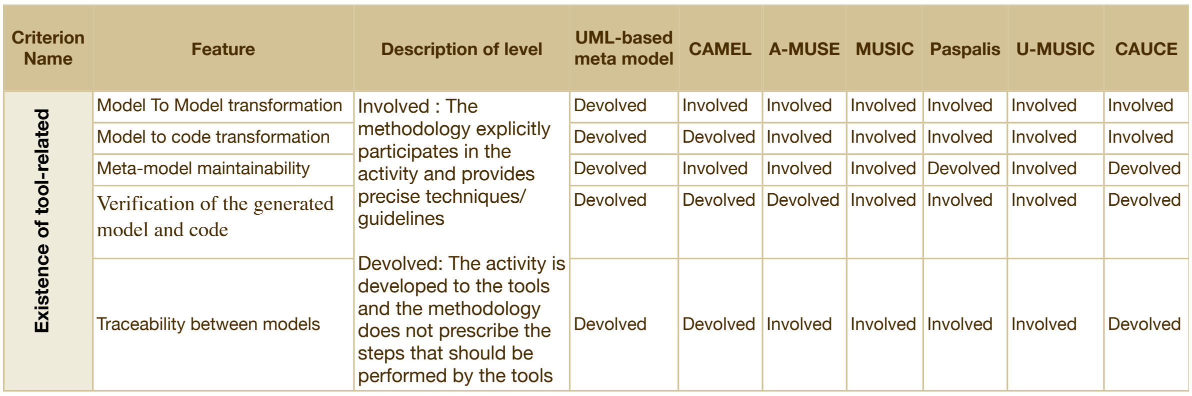


Group criteria	Features	Criterion Type	Description of level	UML-based meta model	CAMEL	A-MUSE	MUSIC	Paspalis	U-MUSIC	CAUCE
Existence of MDA-related Features	Tool suit and implementation:	Scale form	**A.** The methodology does not provide a specific tool and there are no explicit guidelines as to how to select an appropriate alternative. **B.** The methodology does not provide a complete toolset, or only general guidelines are provided for selecting alternative tools. **C.** The methodology provides a complete toolset, or provides precise guidelines for selecting appropriate alternative tools.	A	A	A	C	B	B	A
	Computational independent model	Scale form	**A**. Production of the model are not addressed by the methodology. **B.** The methodology provides general guidelines for creating the model; creation steps are not determined precisely. **C.** The methodology explicitly describes steps and techniques for creating the model.	A	A	A	B	B	B	C
	Platform independent model	Scale form		B	B	B	C	B	C	C
	Platform specific model	Scale form		B	B		C	B	C	C
	Verification and validation	Scale form	The activity is not defined and is devolved to the developers. . The activity is defined by the methodology, but not in detail. The methodology provides explicit and detailed guidelines and techniques for performing the activity.	B	A	B	C	A	A	A
	Source model. Target model synchronisation	Scale form		A	A	B	C	A	B	A
	Use of UML profiles	Narrative	Involved: the Methodology depend on the UML Profile Devolved: Methodology is not using UML profile	Involved	involved	Devolved	Devolved	Devolved	Devolved	Devolved

velopment methodology generates more specific technical details that suit the underlying modelling language or modelling tool they used, as each tool requires a learning curve, and it might have some limitation with regard to the platform and the number of implementation languages they support [Lewis and Wrage, 2005]. This implies that a MDD approach should be decoupled from using a specific tool or modelling language. The developers have to be free on selecting the tool(s) that fits their needs and the software under development.

On the other hand, MDD approaches should focus more on describing standard development processes without relaying on a specific technology or platforms like EMF and ECORE. In terms of the tools the methodology used, the features that highlight the methodology dependency on the modelling languages and tools are shown in Table 2.2.

Table 2.2: MDA tool-related criteria



Criterion Name	Feature	Description of level	UML-based meta model	CAMEL	A-MUSE	MUSIC	Paspalis	U-MUSIC	CAUCE
Existence of tool-related	Model To Model transformation	Involved : The methodology explicitly participates in the activity and provides precise techniques/ guidelines	Devolved	Involved	Involved	Involved	Involved	Involved	Involved
	Model to code transformation		Devolved	Devolved	Involved	Involved	Involved	Involved	Involved
	Meta-model maintainability		Devolved	Involved	Involved	Involved	Devolved	Involved	Devolved
	Verification of the generated model and code		Devolved	Devolved	Devolved	Involved	Involved	Involved	Devolved
	Traceability between models	Devolved: The activity is developed to the tools and the methodology does not prescribe the steps that should be performed by the tools	Devolved	Devolved	Involved	Involved	Involved	Involved	Devolved

2.4.3 Quality of the MDA-based Approaches

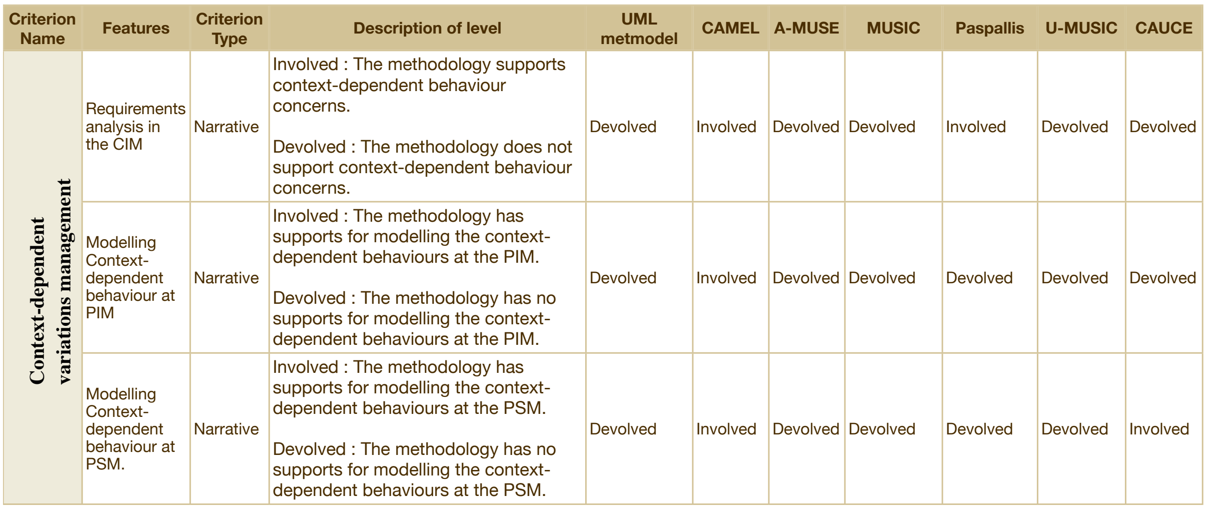
Quality refers to the overall quality of the MDA-based approaches, including their efficiency, robustness, understandability, ease of implementation, completeness, and ability to produce the expected results [Kitchenham et al., 2002]. However, in this research, we have focused on the ability of the MDA-based approaches to provide the expected results that support the adaptability of the generated software, whether these results are derived from the code or the architecture. Moreover, we have split these criteria into four groups: requirements engineering, unanticipated awareness, context model, and modelling context-dependent behavioural variations.

2.4.3.1 Requirements Engineering of Context-dependent Behavioural Variations

Requirements engineering refers to the causes of adaptation, other than the functional behaviour of the self-adaptive system. Whenever the system captures a change in the context, it has to decide whether it needs to adapt. The MDA-based approaches in the related work were evaluated regarding whether they support the modelling of context requirements as a specific feature and whether they support the requirements' engineering in general, as shown in Table 2.3.

In addition, the methodology's ability to analyse and models the context-dependent behaviour variations requires the MDD supports at three levels. The first is the requirement analysis at the computational independent model. The second is the representation of these requirements by means of UML objects at the platform independent model and platform specific model. The third is the representation of the context-dependent behaviour as runtime objects, which are a code representation of these requirements [Bencomo et al., 2010,de Lemos et al., 2011]. However, the evaluation of these criteria is shown in Table 2.3.

Table 2.3: Supporting context-dependent behaviour variations on the analysis, design and implementation

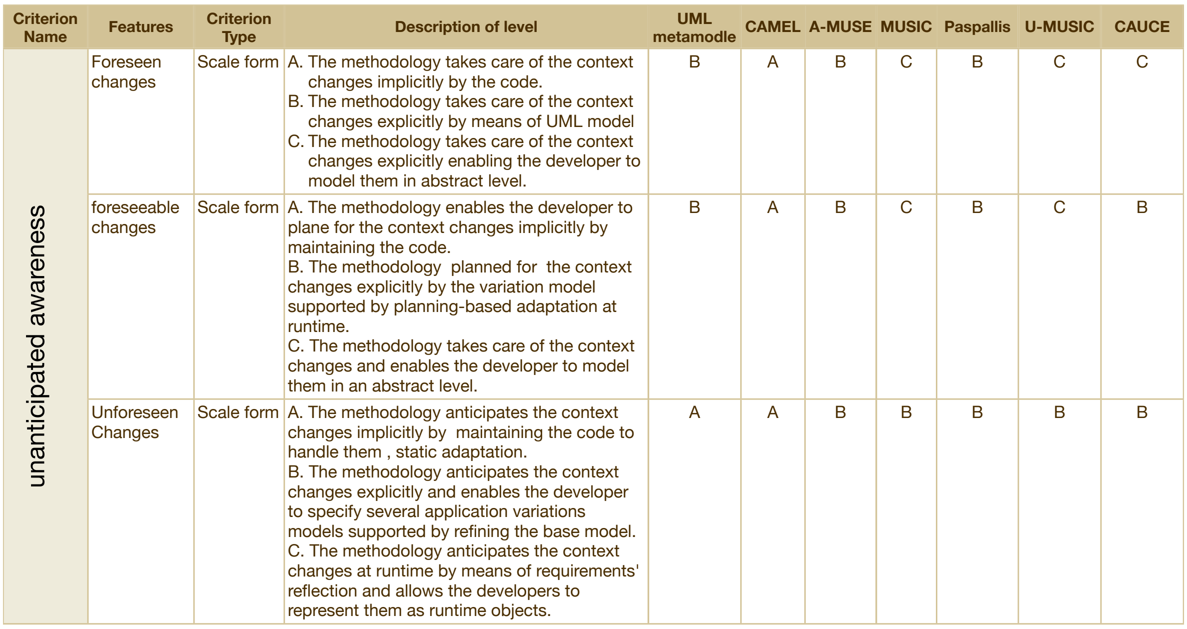


Criterion Name	Features	Criterion Type	Description of level	UML metmodel	CAMEL	A-MUSE	MUSIC	Paspallis	U-MUSIC	CAUCE
Context-dependent variations management	Requirements analysis in the CIM	Narrative	Involved : The methodology supports context-dependent behaviour concerns. Devolved : The methodology does not support context-dependent behaviour concerns.	Devolved	Involved	Devolved	Devolved	Involved	Devolved	Devolved
	Modelling Context-dependent behaviour at PIM	Narrative	Involved : The methodology has supports for modelling the context-dependent behaviours at the PIM. Devolved : The methodology has no supports for modelling the context-dependent behaviours at the PIM.	Devolved	Involved	Devolved	Devolved	Devolved	Devolved	Devolved
	Modelling Context-dependent behaviour at PSM.	Narrative	Involved : The methodology has supports for modelling the context-dependent behaviours at the PSM. Devolved : The methodology has no supports for modelling the context-dependent behaviours at the PSM.	Devolved	Involved	Devolved	Devolved	Devolved	Devolved	Involved

2.4.3.2 Unanticipated Awareness

This feature captures whether a context change can be predicted ahead of time [Cheng et al., 2008]. Anticipation can be classified into three degrees: foreseen, foreseeable, and unforeseen changes. Foreseen refers to the changes that are handled in the implementation code. Foreseeable refer to the context changes that were predicted at the software design. Unforeseen refers to the changes that are not modelled at the design or the implementation stage, but are to be handled at runtime [Laprie, 2004]. The evaluation criteria are shown in Table 2.4 with their related scale form.

Table 2.4: Anticipation of context change-related criteria and evaluation results



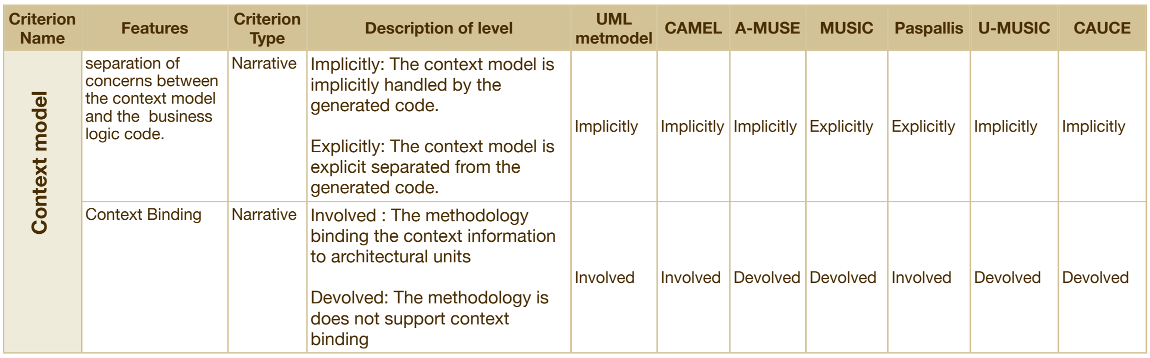
Criterion Name	Features	Criterion Type	Description of level	UML metamodle	CAMEL	A-MUSE	MUSIC	Paspallis	U-MUSIC	CAUCE
unanticipated awareness	Foreseen changes	Scale form	A. The methodology takes care of the context changes implicitly by the code. B. The methodology takes care of the context changes explicitly by means of UML model C. The methodology takes care of the context changes explicitly enabling the developer to model them in abstract level.	B	A	B	C	B	C	C
	foreseeable changes	Scale form	A. The methodology enables the developer to plane for the context changes implicitly by maintaining the code. B. The methodology planned for the context changes explicitly by the variation model supported by planning-based adaptation at runtime. C. The methodology takes care of the context changes and enables the developer to model them in an abstract level.	B	A	B	C	B	C	B
	Unforeseen Changes	Scale form	A. The methodology anticipates the context changes implicitly by maintaining the code to handle them , static adaptation. B. The methodology anticipates the context changes explicitly and enables the developer to specify several application variations models supported by refining the base model. C. The methodology anticipates the context changes at runtime by means of requirements' reflection and allows the developers to represent them as runtime objects.	A	A	B	B	B	B	B

2.4.3.3 Context Model

This captures the ability of the methodology to incorporate the context information using the 'separation of concerns' technique between the context information model and the business logic. The first criterion focuses whether the methodology supports/uses the separation of concerns in the development processes. The second criterion refers to the ability to bind the context source to the context provider, as proposed by Sen and Roman [Sen and Roman, 2003] and Broens et al. [Broens et al., 2007] and Paspallis [Paspallis, 2010]. The binding mechanism enables the developers to map each context cause to the affected architectural units. The binding mechanism also enables the application to determine which part has to manage the context changes, by means of the adaptation mechanism.

The evaluation criteria for the context model are shown in Table 2.5.

Table 2.5: Context model-related criteria and evaluation results



Criterion Name	Features	Criterion Type	Description of level	UML metmodel	CAMEL	A-MUSE	MUSIC	Paspallis	U-MUSIC	CAUCE
Context model	separation of concerns between the context model and the business logic code.	Narrative	Implicitly: The context model is implicitly handled by the generated code. Explicitly: The context model is explicit separated from the generated code.	Implicitly	Implicitly	Implicitly	Explicitly	Explicitly	Implicitly	Implicitly
	Context Binding	Narrative	Involved : The methodology binding the context information to architectural units Devolved: The methodology is does not support context binding	Involved	Involved	Devolved	Devolved	Involved	Devolved	Devolved

2.4.3.4 Modelling Context-dependent Behaviour

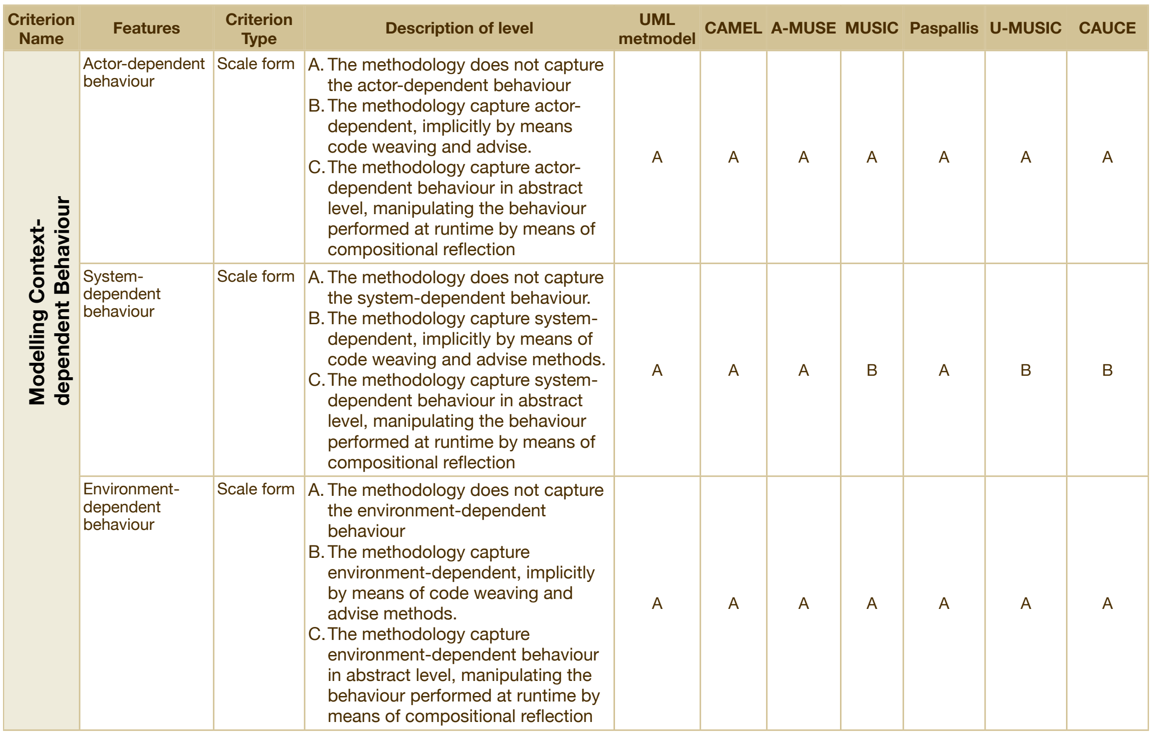
These criteria refer to the ability of the model to capture the impact of context changes on the self-adaptive application's behaviour. However, Hirschfeld et al. [Hirschfeld et al., 2008] classified these changes into three kinds of variations: actor dependent, system dependent, and environment dependent behavioural variations. These behavioural variations requires a separation between their concerns, by separating the context handling from the concern of the application business logic. In addition, a separation between the application-dependent parts from the application-independent parts can support behavioural modularization of the application, thereby simplifying the selection of the appropriate parts to be invoked in the execution, whenever a specific context condition is found. The behavioural modelling criteria are shown in Table 2.6.

2.4.4 Evaluation Results

Based on the analysis results shown in Figures 2.1, 2.2, 2.3, 2.4, 2.5, and 2.6, we find that the discussed methodologies in the related work suffer from several critical failings in terms of their conformance to the OMG's guidelines for MDA methodology [Kleppe et al., 2003].

First, it is well known that correct identification of the weight of each goal is a major

Table 2.6: Modelling context-dependent behaviour variations and evaluation results



Criterion Name	Features	Criterion Type	Description of level	UML metmodel	CAMEL	A-MUSE	MUSIC	Paspallis	U-MUSIC	CAUCE
Modelling Context-dependent Behaviour	Actor-dependent behaviour	Scale form	A. The methodology does not capture the actor-dependent behaviour B. The methodology capture actor-dependent, implicitly by means code weaving and advise. C. The methodology capture actor-dependent behaviour in abstract level, manipulating the behaviour performed at runtime by means of compositional reflection	A	A	A	A	A	A	A
	System-dependent behaviour	Scale form	A. The methodology does not capture the system-dependent behaviour. B. The methodology capture system-dependent, implicitly by means of code weaving and advise methods. C. The methodology capture system-dependent behaviour in abstract level, manipulating the behaviour performed at runtime by means of compositional reflection	A	A	A	B	A	B	B
	Environment-dependent behaviour	Scale form	A. The methodology does not capture the environment-dependent behaviour B. The methodology capture environment-dependent, implicitly by means of code weaving and advise methods. C. The methodology capture environment-dependent behaviour in abstract level, manipulating the behaviour performed at runtime by means of compositional reflection	A	A	A	A	A	A	A

difficulty for the utility functions as shown in the MUSIC, U-MUSIC and Paspallis methodologies.

Second, these approaches hide conflicts among multiple adaptation goals by combining them into a single, aggregate objective function, rather than exposing the conflicts and reasoning about them. On the other hand, it would be optimistic to assert that the process of code generation from models can become completely automatic or that the developer's role lies only in application design, as discussed in the above with regard to CAMEL and A-MUSE.

Third, it is impossible for the developers to predict the possible application variations, that will extend the application behaviour when unanticipated conditions arise, this applied to all methodologies mentioned in the above.

In addition, mobile devices have limited resources for evaluating many application variations at runtime, which might consumes significant amounts of the allocated resources. As a result, the benefits gained from the adaptation are negated by the overhead required to achieve the adaptation. Fourth, the previously mentioned methodologies produce an architecture with a tight coupling between the context provider and the context consumer, which may cause the middleware to notify multiple components about multiple context changes. Finally, all the methodologies seem to generate an architecture that is tightly coupled with the target platform for deployment and the modelling tools they used.

In addition, the developers have to explicitly predict the final composition of the software and the possible variations of the application, whether at the platform independent model or through the model transformation. Moreover, the developers have to modify the generated code to be suitable for deployment on the target platform and to be integrated with the middleware implementation, which is in the best case made a hug gap between the middleware designer and the application developer. In the same way, the developer's understanding of the MDA tasks and familiarity with the target platform have limited the ability to adapt the MDA-approaches in several domains.

2.5 Component-based Software Development

Component-based software engineering focuses on shifting the developers' attention from lines-of-code to coarse-grained components and their interconnection structure. Unlike fine-grained objects, these components typically encompass business functionality uses a separation of concerns technique to modularise the software. Although, context information can drive the changes on the software functionality and behaviour, context-awareness has not considered through the development or deployment of component-based software [Lau, 2006].

The component-based self-adaptive applications require an architecture and framework that support their modelling and implementation. Several adaptation techniques concentrate on providing a mechanism to compose them based on their structure intersection and their dynamic behaviour introspection. This section discusses a few of the most closely related component-based model, which targeting dynamic component-based adaptation and runtime composition of software modules.

2.5.1 LEAD++

Amano and Watanabe [Amano and Watanabe, 1999] proposed a framework based on a description language called LEAD++. LEAD++ creates a software model, called DAS, that structures a dynamic adaptive component-based system. In the DAS model, adaptability is achieved based on the mechanism of the adaptable procedure. Each procedure refers to a variant of methods, which are selected according to the execution context. The adaptation is controlled using an adaptable procedure mechanism, which uses methods for selecting adaptation strategies.

LEAD++ can be considered as an example of parameter-based adaptation. Although it covers the state of the environment as a runtime object, it does not capture the impact of the context state on the application behaviour in either a proactive or reactive manner. Whenever the environment state changes, the strategy objects dispatch control objects; this is not sufficient to select a valid variation from among multiple behaviour variations because each control object considers only one environment state. In general, this framework is not aware of the context information and no context gathering is provided. Context monitoring is not supported by this framework.

2.5.2 FRACTAL

The fractal component model, FRACTAL, was proposed by Bruneton et al. [Bruneton et al., 2002]; it allows a component to be nested at an arbitrary level. A fractal component consists of content and controller parts. Each content part can consist of a finite number of components; each component is controlled using the controller part of the fractal component. In the same way, a component's content can be shared among multiple distinct fractal components. Moreover, components can interact with their environment through operations at identified access points, called interfaces, which might be a server or a client interface. The fractal component can interact using multiple interfaces, which satisfies a client-server architecture style.

A component controller embodies the component behaviour for the associated fractal component. The controller performs operation interception for both the outgoing and incoming operation invocations and their operation results. In general, FRACTAL shows a component model that supports recursive and internal components to satisfy two-way operation invocation. It is clear that context information is not supported at any level. Moreover, the sub-components inside a FRACTAL component are not realized based on their behaviour, instead, they are included by the controller whenever they are visible to the environment and emit operation invocation. This may lead to the exclusion of an important sub-component from the adaptation strategy when it has a direct dependency on other components.

2.5.3 One.world

Grimm [Grimm, 2004] developed an architecture based on the pressing need to cover the dynamicity of context information, information that is constantly increasing in size as users move through the physical space and as users collaborate and share data in distributed systems. One.world emerged to provide an architecture that offers four services. First, a Java virtual machine is used to support multiple heterogeneous user devices that are available in the distributed environment. Second, all data are represented as tuples, which are records with name and (optional) type fields that provide self-description for the application structure. Third, event synchronization is provided for local and remote connections. Fourth, each environment host executes an application and isolates applications from one another.

In general, One.world supports contextual changes, ad-hoc composition, and information sharing among environment hosts. Applications are able to inspect their structure using the self-description property of the data tuple. One.world partially supports contextual information in a distributed environment, but has no context monitoring, detection, or reasoning. Another issue is the degree to which the framework can handle a fluctuating device, i.e. one that might appear or disappear at arbitrary times. Context handling requires a mechanism that supports the physical and logical context and their dynamicity at runtime.

2.5.4 PCOM

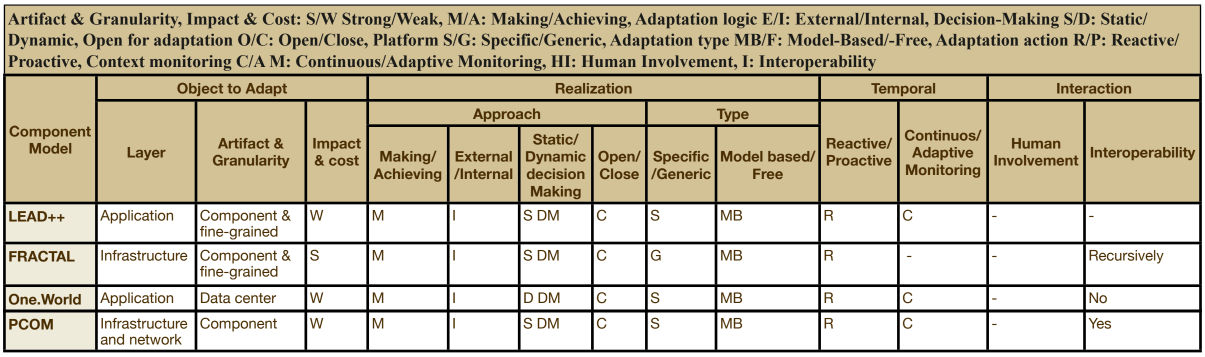
PCOM is a component system presented by Becker et al. [Becker et al., 2004]. It enables the developer to capture the dependencies among components at the abstract level using contracts. As a result, the application architecture is a tree formed by components and their dependencies. The component model supports dynamic adaptation based on context. Applications in PCOM are composed of components that interact with each other. Components are atomic to the distributed environment; however, they can rely on local or remote components. The application tree reflects the dependencies among components. Each component encloses contracts that describe both their provided functionality and their requirements regarding the platform and other components.

Finally, the PCOM component model can be categorized as a programmable component model, in that it relies on the developer for the structuring the software components in a suitable way. The model relies on functional decomposition of the software into service oriented components. In addition, the composition mechanism relies on the user and the developers to decide the kinds of services the application can adapt. In addition, PCOM relies on a known network topology, which, in most cases, limits the application's ability be used on a specific platform.

2.5.5 A Comparative Analysis of Component-based Frameworks

A comparison of the above-mentioned component frameworks is shown in Table 2.7. The table outlines the features supported by each component model in terms of objects which they adapt: realisation approaches, proactive or reactive adaptation, context monitoring, and human interaction. This comparative study is adapted from previous work conducted by Salehie and Tahvildari [Salehie and Tahvildari, 2009]. The findings related to this comparative analysis based on each component framework, can be analysed column-wise in the table. The values corresponding to different columns in the table are as follows:

Table 2.7: Component model comparative analysis



Artifact & Granularity, Impact & Cost: S/W Strong/Weak, M/A: Making/Achieving, Adaptation logic E/I: External/Internal, Decision-Making S/D: Static/ Dynamic, Open for adaptation O/C: Open/Close, Platform S/G: Specific/Generic, Adaptation type MB/F: Model-Based/-Free, Adaptation action R/P: Reactive/ Proactive, Context monitoring C/A M: Continuous/Adaptive Monitoring, HI: Human Involvement, I: Interoperability

Component Model	Object to Adapt			Realization						Temporal		Interaction	
				Approach				Type					
	Layer	Artifact & Granularity	Impact & cost	Making/ Achieving	External /Internal	Static/ Dynamic decision Making	Open/ Close	Specific /Generic	Model based/ Free	Reactive/ Proactive	Continuos/ Adaptive Monitoring	Human Involvement	Interoperability
LEAD++	Application	Component & fine-grained	W	M	I	S DM	C	S	MB	R	C	-	-
FRACTAL	Infrastructure	Component & fine-grained	S	M	I	S DM	C	G	MB	R	-	-	Recursively
One.World	Application	Data center	W	M	I	D DM	C	S	MB	R	C	-	No
PCOM	Infrastructure and network	Component	W	M	I	S DM	C	S	MB	R	C	-	Yes

- Application layer: This feature captures which layer of the software system can be changed during the adaptation action. The adaptation action can be applied to the application/middleware layer or distributed in the infrastructure nodes (decentralized). As shown in Table 2.7, in LEAD++ and One.World, the adaptation is achieved in the application layer. In FRACTAL and PCOM, the adaptation is performed with support of the infrastructure and decentralized on the network nods.

- Artifacts & granularity: This feature captures at which level of granularity, the artifact can be changed. The adaptation can change the modules or the architectural units; and the way they are composed. The artifact & granularity column in the table shows that LEAD++ and FRACTAL support both fine- and coarse-grained adaptations. The One.World and PCOM frameworks can adapt the whole component or the data centre of the application.

- Impact & cost: The adaptation impact and cost describes the scope of the after effects, while cost refers to the execution time, resources required and complexity of the adaptation actions. The adaptation actions can be categorised into Weak (W) and Strong (S) classes. Weak adaptation involves modifying parameters (parameter adaptation) or performing low-cost/limited-impact actions, whereas strong adaptation deals with high-cost/extensive-impact actions such as replacing components with those that improve system quality [McKinley et al., 2004]. In these terms, the weak adaptation may include changing parameters or other actions with local impact and low cost.

On the other hand, strong adaptation may changes, adds, removes or substitutes system artifacts. LEAD++ and One.World have limited impact on the architecture components and their interconnections. FRACTAL have unlimited impact/cost on the architecture. The four frameworks compose the components at the design time and recomposed them at runtime based on the adaptation action.

- Making/Achieving: Self-adaptability can be introduced into software systems on the development phase (M: Making) or the adaptation is achieved at runtime through dynamic composition (A: Achieving). In the four frameworks, the adaptation is achieved at runtime.

- External/Internal Adaptation: The adaptation can be divided into two categories with respect to the separation between the adaptation mechanisms from the application logic.

  The internal approaches mixed the application and the adaptation logic. This approach is based on programming language features such as conditional expressions, parametrisation and exceptions [Oreizy et al., 1999, Floch et al., 2006].

  A complex self-adaptive software system requires a mixed approach between internal and external adaptation, which provides a composition of elements in an appropriate architecture, and an infrastructure support for interoperability. Unfortunately, the four frameworks in this study support an internal adaptation only, which limited their flexibility and scalability to adapt context changes.

- Static/Dynamic decision making: This feature captures how the decision process can be constructed and modified. Static refers to the frameworks that use a hard-coded decision and its modification requires recompiling and redeploying the software system or some of its components.

In dynamic decision-making, policies, rules or QoS are externally defined and managed, so that they can be changed during runtime. One.World framework is the only framework that supports dynamic decision-making. The others support static decision making, which limited their adaptability and dependability.

- Open/Close: A closed-loop software system has only a fixed number of adaptive actions, and no new behaviours and alternatives can be introduced during runtime.

  On the other hand, in open-loop software system, the self-adaptive software can be extended, and consequently, new alternatives can be added, and even new adaptable entities can be introduced into the adaptation mechanism. The four frameworks are classified as closed-adaptive systems, because they use a closed execution loop supported by a feedback from the computational environment, which limits their adaptability to a fixed number of adaptation actions.

- Specific/Generic: Some of the component frameworks address only specific domain/application. However, generic framework can be configured by customising the component model for different domains. The four frameworks are designed to suit only a specific domain and they can not be configured to address different domains. In this context, PCOM targets a distributed environment that has various heterogeneous devices. FRACTAL and LEAD++ target context-aware applications in embedded systems and has no support for self-adaptability. One.world supports context-awareness in the virtual machine layer of Java applications.

- Model based/Free: In model-free adaptation, the mechanism does not have a predefined model for the environment and the system itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the software structure and/or behaviour dynamically. On the other hand, in model-based adaptation, the mechanism utilizes an architecture model of the software system and its context information for adjusting the behaviour/functionality dynamically using compositional adaptation techniques. The four frameworks were designed as a model-based framework which can support a specific computational environment.

- Reactive/Proactive: This feature captures the anticipatory property of the self-adaptive software. In the reactive mode, the system responds when a change has already happened, while in the proactive mode, the system predicts the changes and suitable adaptation actions before it happened. This issue impacts the detecting and the deciding processes. The frameworks support a reactive response for upcoming events, they sense the environment and then propose reaction in response to the incoming events. In this analysis, the four frameworks support reactive adaptation.

- Continuous/Adaptive Context Monitoring: This feature captures, whether the context monitoring process is continuously collecting and processing data, or the software system adapts the process of context monitoring based on the state of the environment and the allocated resources. This decision affects the process of context monitoring and detecting in terms of cost and the amount of context changes they sense and analyses. The LEAD++, One.World, and PCOM frameworks continuously monitor the environment. FRACTAL has no support for monitoring the context.

- Human Involvement: The four frameworks do not offer the end user with a facility for controlling the adaptation strategy and deciding which adaptation output might suit the user needs.

- Interoperability: Self-adaptive software often consists of elements, modules, and subsystems. Interoperability is always a concern in distributed software systems for maintaining the behaviour across all constituent elements and subsystems. In self-adaptive software, the elements need to be coordinated with each other to achieve the desired self-* properties and to fulfil the expected adaptation objectives. Component interoperability is not supported in the LEAD++ and One.World frameworks, but it is recursively integrated in FRACTAL component framework, and it is totally supported by PCOM framework.

2.6 Middleware-centric Development

The middleware approach uses an external adaptation engine to carry out 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, primarily with the aid of middleware [Floch et al., 2006, Mukhija and Glinz, 2005, Chusho et al., 2000], a policy engine [Anthony et al., 2009], or other application-independent mechanisms. The middleware flexibility when adapting a suitable adaptation approach helps to achieve the adaptation results at a lower cost and at several levels of granularity [Salehie and Tahvildari, 2009].

The middleware proposed in the literature [Capra, 2003, Dowling and Cahill, 2001, Anthony et al., 2008b, Mukhija and Glinz, 2005] does not consider the effects of context monitoring on the device's resources. The second aspect of context detection is the need to detect changes and notify the interested context consumer about them. These enhancements of context monitoring and detection can improve the efficiency of adaptation processes.

Another aspect that the middleware must consider is the decision process. Decision making determines which parts need to be changed and how to change them to achieve the best output. Some middleware such as Mobility and ADaptation enAbling Middleware (MADAM) [Mukhija and Glinz, 2005], Contract-based Adaptive Software Architecture (CASA) [Noble et al., 1997], and QUO [Loyall et al., 1998] has used static decision-making, i.e. the decision process is hard-coded and its modification requires recompiling and redeploying the whole system or some of its components. Dynamic decision-making is externally defined and managed; therefore, it can be changed at runtime to create or adjust the behaviour of either functional or behavioural adaptations.

Anthony et al. [Anthony et al., 2009] used policies for maintaining and evaluating the adaptation results at runtime, a policy framework that is managed by a policy manager, which verifies the policies among contradictions and evaluates the architecture evolution among constraints specified by the policy.

The middleware must be able to switch autonomously between weak adaptation and strong adaptation types. In some cases, adaptation can be achieved by modifying parameters or performing low-cost/limited-impact actions, whereas, in other cases, adaptation requires either replacing components with those that improve system quality or performing high-cost/extensive-impact actions [McKinley et al., 2004]. The cost in this classification refers to how much time and resources the adaptation action requires. The adaptation costs have a direct impact on architecture quality attributes; therefore, the middleware has to consider the trade-off between these attributes during an adaptation.

2.6.1 DySCAS

Anthony et al. [Anthony et al., 2009] proposed DySCAS, which is dynamically self-configuring middleware for automotive control systems. The middleware facilitates context-aware dynamic reconfiguration. The framework focuses on the context-aware logic and the selection of context information used in dynamic decision-making and is runtime changeable. This yields a highly flexible system in which the functionalities of its applications are not restrained by the (limited) design-time vision. The DySCAS component model for runtime configuration is based on embedded Decision Points (DPs) in the software component into which policies are loaded at runtime. Thus, the configuration logic can be distributed throughout the middleware and application components wherever deferred logic or runtime context-sensitive configuration is required. The logic modules are loaded into DPs in the form of policies written in AGILE policy grammar [Anthony et al., 2009]. The decision points encapsulate a runtime supervisor in which the policy operates. The wrapper detects and handles any problems that arise during policy evaluation. during the policy loading procedure, the context requirements are identified for a specific policy.

However, the DySCAS model is better suited for the design of embedded systems than it is to mobile computing environment, where the context model can dynamically evolve at runtime. In addition, it has no support for an abstract model of the policy or the component, which requires the developer to focus on the policy language syntax and in the implementation code. Furthermore, DySCAS adaptation relies only on policy evaluation and has no support for dynamic decision-making that considers the interoperability between the architecture components. However, DYSCAS can supports policy evolution using a policy engine, but it failed to support completely the architecture evolution.

2.6.2 CASA

Mukhija and Glinz [Mukhija and Glinz, 2005] proposed CASA that provides a framework for enabling the development and operation of adaptive applications by separating adaptation concerns from business concern. The focus is on providing a runtime infrastructure to meet a broad range of applications. Additionally, CASA supports adaptation at several levels, ranging from lower services to the application code. The adaptation policy for specific applications is defined using a contract that facilitates changes in the adaptation policy at runtime. In CASA, adaptation techniques address a dynamic change in lower-level services, dynamic weaving and unweaving of aspects, dynamic recomposition of application components, and dynamic changes in application attributes. Additionally, the application might use any combination of the above techniques based on the adaptation needs. The CASA runtime system (CRS) is responsible for monitoring the execution environment and triggering the adaptation process for the affected application. During the adaptation, the user has direct control over the process, which is intended to serve as a user-transparent mechanism.

CASA uses the Odyssey framework [Noble et al., 1997] for monitoring the network resources and relies on the PROSE runtime framework [Nicoara and Alonso, 2005] for aspect weaving. In addition, it does handle homogeneous functional crosscutting concerns that can be weaved into the application. Context-awareness and self-adaptability is not supported in CASA. Furthermore, aspectual decomposition of functional concerns does not suit the context-driven adaptation that is needed by a context-aware application.

2.6.3 Kinesthetics eXtreme (KX)

Kinesthetics eXtreme is a mobile, agent-based infrastructure for runtime monitoring and reconfiguration of component-based distributed systems [Valetto et al., 2001]. The architecture aims to handle global situations, perhaps involving heterogeneous components obtained from multiple sources, where it would be difficult if not impossible to retrofit self-assurance. The occurrence of a context-related condition within the target system is detected and reported by the monitoring part of the meta-architecture. The process engine is notified about the condition and may dynamically instantiate, initialize, and finally dispatch one or more software agents, called worklets, to perform the adaptation process. Each worklet contains multiple mobile code snippets, called worklet junctions, to actuate the required adaptation of the target system. The data structures of a junction can be initialized with data, typically incoming from the task definition, process context, and information contained in the events, which represents the triggering condition. Another part is called a worklet jacket; it allows scripting of certain aspects of the worket behaviour in the course of its route. The process engine requests junctions for the dynamic adaptation task at hand from a worklet factory, which has access to a categorized semantic catalogue of junction classes and instantiates them on its behalf.

Their work was driven by adding autonomic properties to legacy systems, i.e. existing systems that were not designed with autonomic properties in mind. Indeed, it is sometimes not possible to modify these systems, thus requiring both the addition of autonomic properties that are completely decoupled, and autonomic monitoring sensors to be installed, on top of the existing system APIs and monitoring functionality. Their work is more focussed on the collection and processing of monitoring data from legacy systems and the execution of adaptation and repairs, rather than algorithms and policies for adaptation planning. This makes KX unsuitable for the implementation of self-adaptive applications that require an intensive operation over the context environment.

2.6.4 MADAM

Mobility and ADaptation enAbling Middleware (MADAM) was created to build adaptive applications for mobile devices using architecture models [Mikalsen et al., 2006, Floch et al., 2006]. MADAM claims to facilitate adaptive application development by separating the context provider concern from the context consumer concern. MADAM provides a framework for developers to design component-based applications and support their execution in a distributed environment. MADAM requires an application to be divided into a hierarchical set of components. The architecture model presented at runtime allows generic middleware components to reason about and control the adaptation strategy.

In the MADAM middleware, context changes are detected using context providers. It reasons about the changes and makes decisions about what adaptation is to be performed based on the application variation model, and implements the adaptation choices. The configuration manager reconfigures the component-based application to put the decided adaptations into effect. To support adaptation, the architecture must encode variation and selection criteria so that the middleware can automate the derivation of a variant for a specific context at runtime. The component framework describes the composition of the component types. The application variability is achieved by plugging in different component implementations. Each component's externally observable behaviour conforms to its type. The component can be atomic or composite, which is built as a component framework itself. In this way, the application is assembled from a recursive structure of components frameworks.

The MADAM middleware can support planning-based adaptation by instantiating a plug-in architecture that fulfils the utility function evaluations. The middleware is responsible for constructing and analysing several variability models at runtime, which adds an extreme overhead to a mobile device with limited resources. It is very expensive to evaluate the context state then executing a preloaded plug-in, especially when context changes occur frequently.

Mobile USers In Ubiquitous Computing (MUSIC) middleware [Rouvoy et al., 2008a, Geihs et al., 2011] is an extension of the MADAM component-based planning framework that optimizes the overall utility of applications when context-related conditions occur. The planning-based adaptation of MADAM employs dynamic configuration of component frameworks. In MUSIC, the planning extends further, to support seamless configuration of component frameworks based on both local and remote components and services. Thus, both components and services are plugged in interchangeably to provide functionalities defined by the component framework.

In addition to what MADAM has achieved, MUSIC also models and realises service bindings and extended situational contexts as part of the context dependencies based on a service-oriented approach. This means that if some appropriate service is detected at runtime in the execution environment, it can automatically be integrated and can replace another software component. This flexible dynamic reconfiguration may apply to service components at both the platform and the application levels [Rouvoy et al., 2008a, Geihs et al., 2011].

The decision-making process in MUSIC is similar to MADAM middleware, both use a utility function to evaluate all the reasoning dimensions used by the adaptation reasoner to select and deploy the component implementation, thereby providing the best utility. Mobile devices have limited resources to devote to the evaluation of many application variations at runtime and can consume significant amounts of device resources. As result, the benefit gained from the adaptation is negated by the overhead required to achieve the adaptation. Furthermore, the generated architecture can allow fine-grained adaptation or coarse-grained adaptation, where, in some cases, both types of adaptation action are required to anticipate the context changes. Finally, this form of the decision-making process does not allow the user to interfere with the decision-making process; the user has no control over the adaptation effect.

2.6.5 CARISMA

Capra [Capra, 2003] proposes Context-aware Reflective Middleware System for Mobile Applications (CARISMA). CARISMA proposed as a mobile computing middleware that enhances the construction of adaptive and context-aware applications. The middleware enables the developer to describe context information in name/value pairs by means of XML, which is defined as a policy. Policy conflict is handled at runtime using a microeconomic approach that relies on a particular type of sealed-bid auction. As a result, the application is allowed to dynamically inspect the middleware behaviour and dynamically change its behaviour by means of a meta-interface that enables runtime modification of the internal representation that was previously made explicit. CARISMA exploits the use of computational reflection to achieve dynamic adaptation to context changes.

The CARISMA middleware is responsible for maintaining a valid representation of the execution context by directly interacting with the underlying network operating system. CARISMA uses aspects weaving of functional concerns, which does not suit the context-driven behaviour. As described before, this requires context-driven aspects supported by context handling aspects in the platform. In addition, it supports parameter-based adaptation using internal adaptation approach. Furthermore, CARISMA is specific to applications in which context changes are foreseen and planned for their anticipation at the design time of the software.

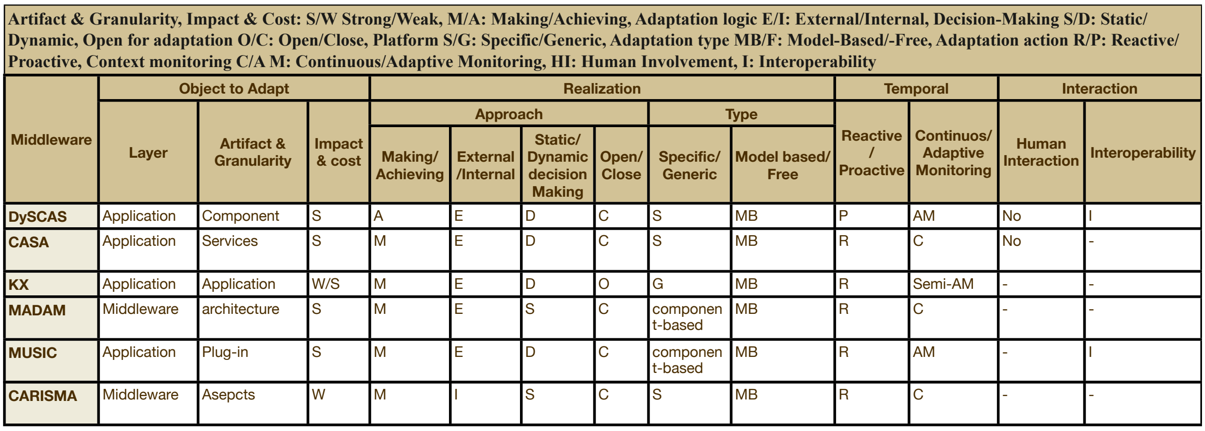
2.6.6 Middleware Comparative Study

A comparison of the above-mentioned middleware is shown in Table 2.8. The table outlines the features supported by different middleware architectures, including the adapted objects, the adaptation realization, the temporal feature, which refers to the middleware response among context changes, and the human interaction in controlling the adaptation process.

Each column in the table can be described as follows:

- Application layer: This feature captures which layer of the software system can be changed during the adaptation action. The adaptation action can be applied to the application/middleware layer or distributed in the infrastructure nodes (decentralized). As shown in Table 2.8, in CARISMA and MADAM, the adaptation is achieved in the application layer. In other middleware architectures, the adaptation is performed with the support of the infrastructure and decentralized on the network nods.

Table 2.8: Middleware comparative study



Artifact & Granularity, Impact & Cost: S/W Strong/Weak, M/A: Making/Achieving, Adaptation logic E/I: External/Internal, Decision-Making S/D: Static/Dynamic, Open for adaptation O/C: Open/Close, Platform S/G: Specific/Generic, Adaptation type MB/F: Model-Based/-Free, Adaptation action R/P: Reactive/Proactive, Context monitoring C/A M: Continuous/Adaptive Monitoring, HI: Human Involvement, I: Interoperability

Middleware	Object to Adapt			Realization						Temporal		Interaction	
	Layer	Artifact & Granularity	Impact & cost	Approach				Type		Reactive / Proactive	Continuos/ Adaptive Monitoring	Human Interaction	Interoperability
				Making/ Achieving	External /Internal	Static/ Dynamic decision Making	Open/ Close	Specific/ Generic	Model based/ Free				
DySCAS	Application	Component	S	A	E	D	C	S	MB	P	AM	No	I
CASA	Application	Services	S	M	E	D	C	S	MB	R	C	No	-
KX	Application	Application	W/S	M	E	D	O	G	MB	R	Semi-AM	-	-
MADAM	Middleware	architecture	S	M	E	S	C	componen t-based	MB	R	C	-	-
MUSIC	Application	Plug-in	S	M	E	D	C	componen t-based	MB	R	AM	-	I
CARISMA	Middleware	Asepcts	W	M	I	S	C	S	MB	R	C	-	-

- Artifacts and granularity: This feature captures at which level of granularity the artifact can be changed. The adaptation can change the modules or the architectural units; and the way they are composed. The artifact and granularity column in the table shows that DySCAS support adaptation at component level. The CASA middleware supports adaptation for services found in the distributed environment. The KX supports adaptation at the application level. The MUSIC midleware uses a plug-in architecture for performing components' composition. The CARISMA middleware uses aspects weaving for performing the adaptation actions.

- Impact & cost: The adaptation impact describes the scope of the after effects, while cost refers to the execution time, resources required and complexity of the adaptation actions. The adaptation actions can be categorised into Weak (W) and Strong (S) classes. In these terms, weak adaptation may include changing parameters or other actions with local impact and low cost. On the other hand, strong adaptation may changes, adds, removes or substitutes software artifacts. DySCAS, CASA, MADAM and MUSIC middleware architectures support strong adaptation action with high impact over the allocated resources. The KX middleware has the ability to switch between strong and weak adaptation actions. The CARISMA middleware supports weak adaptation only with low impact over the application code, because it uses aspects weaving.

- Making/Achieving: Self-adaptability can be introduced into software systems on the development phase (M: Making) or the adaptation is achieved at runtime through dynamic composition (A: Achieving). The DySCAS is the only middleware that achieves the adaptation at runtime, because it uses decision policies for controlling the adaptation action and it provides support for policies evolution.

- External/Internal Adaptation: The adaptation can be divided into two categories with respect to the separation between the adaptation mechanisms and the application logic. The internal approach encodes the adaptation action in the application logic. This approach is based on programming language techniques such as conditional expressions, parametrisation and exceptions [Oreizy et al., 1999, Floch et al., 2006]. In the internal 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 approaches use an external adaptation engine, which provides the adaptation actions. In this approach, the self-adaptive software system consists of an adaptation engine and adaptable software. The external engine implements adaptation logic, mostly with the aid of middleware [Chusho et al., 2000, Mukhija and Glinz, 2005], a policy engine [Anthony et al., 2009], or other application-independent mechanisms. The DySCAS, CASA, KX, MADAM, and MUSIC have a dedicated software component for performing the adaptation actions, and it is separated from the application business code. In CARISMA, the adaptation mechanism is mixed with the application business logic, because it uses an AOP framework.

- Static/Dynamic decision making: This feature captures how the decision process can be achieved and modified. Static refers to the middleware that uses a hard-coded decision and its modification requires recompiling and redeploying the software or some of its components. In dynamic decision-making, policies, rules or QoS are externally defined and managed, so that they can be changed during runtime. Both MADAM and CARISMA support static decision making, which limited their adaptability and dependability properties. The DySCAS, CASA, KX and MUSIC support dynamic decision making. The DySCAS uses a policy framework for architecture evolution. Both KX and CASA middleware use rule-based engine to control the adaptation action. The MUSIC middleware uses a utility function to calculate the best adaptation plan.

- Open/Close: A closed-loop software system has only a fixed number of adaptive actions, and no new behaviours and alternatives can be introduced during runtime. On the other hand, in open-loop software system, the self-adaptive software can be extended, and consequently, new alternatives can be added, and even new adaptable entities can be introduced into the adaptation mechanism. The DySCAS, CASA, MADAM, MUSIC, and CARISMA are classified as closed-adaptive systems, because they use a closed execution loop with pre-defined number of adaptation actions. The KX middleware is open to its environment and it can includes new behaviours or services without pre-knowledge about their implementation.

- Specific/Generic: Some middleware architectures address only a specific domain/application. However, generic middleware architecture can be customised by configuring the decision making process, and adaptation processes for different domains. The DySCAS, CASA, and CARISMA target context-aware applications in embedded systems and has no support for self-adaptability. The MADAM and MUSIC middleware target component-based software for mobile and ubiquitous computing.

- Model based/Free: In model-free adaptation, the mechanism does not have a predefined model for the environment and the software itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the software structure and/or behaviour. On the other hand, in model-based adaptation, the mechanism utilizes a model of the system and its context. The middleware architectures discussed in this study were designed as a model-based architecture, which makes them support a specific environment only.

- Reactive/Proactive: This feature captures the anticipatory property of middleware. In the reactive mode, the middleware responds when a change has already happened, while in the proactive mode, the middleware predicts the changes and provides suitable adaptation actions before it happened. This issue impacts the detecting and deciding processes. The middleware support a reactive response for upcoming events, they sense the environment and then propose reaction in response to them. In this analysis, we find that the DySCAS middleware can support a reactive adaptation among the others.

- Continuous/Adaptive Context Monitoring: This feature captures whether the context monitoring process (and consequently sensing) is continually collecting and processing context information, or it is being adaptable in the sense that it monitors a few selected context changes. This decision affects the cost of the context monitoring and detection process. The CASA, MADAM, and CARISMA middleware architectures continuously monitor the environment. DySCAS and MUSIC have the ability to adapt the monitoring process and utilize the allocated resources.

- Human Involvement: The middleware architectures do not offer the end user a facility for controlling the adaptation strategy and deciding which adaptation output might suits user needs. The four middleware architectures discussed in this analysis do not allow the users to control in the adaptation actions.

- Interoperability: Adaptive middleware often consists of elements, modules, and subsystems. Interoperability is always a concern in distributed complex systems for maintaining the behaviour across all elements and subsystems. In self-adaptive software, the elements need to be coordinated with each other to achieve the desired self-* properties and to fulfil the expected adaptation objectives. Interoperability is not supported in CASA, KX, MADAM and CARISMA middleware architectures, but it is recursively supported in the DySCAS and MUSIC middleware.

2.7 Self-adaptability Assurance and Verification

The goal of self-assurance is to provide evidence that the set of functional and extra-functional properties are satisfied during a system's operation. In general, verification and validation methods depend on stable descriptions of software models and properties. Methods can include performance through runtime evaluation of constraints [Garlan et al., 2004], trade-off of quality attributes [Yang et al., 2009], or policy syntax grammar [Anthony et al., 2008a]

Anthony et al. proposed a policy definition language (AGILE) that uses simple expressive syntax and semantics [Anthony et al., 2009]. The language structure and components are policy suite, policy, rule, action, and return values. The inputs to the policy are internal and external variables. The former refers to internal values between polices, whereas the later refers to variables that are passed from environmental and contextual conditions to the policy at decision points. However, the policy refers to a sequence of rules that implements the self-management behaviour of the software. A rule refers to a statement that can be evaluated as either true or false. An action is taken by the architecture whenever a true or false value is found.

Self-adaptive systems have multiple context-sensitive parts (highly context dependent). Whenever a change in the context is detected, the software has to decide the need for adaptation based on three primary factors: (1) the extra functionalities that the software is providing, (2) the quality attributes that are affected by the change and their related self-* properties, and (3) the user's perspectives and/or the application goals. All of these factors prompt the need for verification activities to provide continual assessment. Deciding the fitness of the adaptation results may require the verification activities to be embedded in the adaptation mechanism.

However, in the literature, there are two classes of models in terms of adaptation assurance. The first class comprises static decision-making models based on the architecture constraints found either in the RAINBOW framework [Garlan et al., 2004] or in later RAINBOW efforts, which evaluate the effectiveness of deploying RAINBOW in specific platform.

The study conducted by Cheng et al. [Cheng et al., 2009] evaluated the effectiveness at maintaining the quality attribute, runtime overheads for the adaptation, and the engineering effort involved in deploying the RAINBOW framework.

In general, RAINBOW attempts to evaluate the adaptation results among predefined architecture constraints that are not sufficient to decide the fitness of the adaptation at runtime (a time at which goals, requirements, and quality attributes evolve and are dynamic). The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns [Garlan et al., 2004]. The RAINBOW framework uses a closed control loop that monitors an executing system's runtime properties and evaluates the model for constraint violation. Whenever this kind of problem is considered, a module-level adaptation is performed.

Another approach to statical middleware system verification can be found in [Keung et al., 2010]. Keung et al. used a statical procedure that performs a sensitivity analysis over quality attributes, identifies and removes influential data points, estimates system stability, and evaluates system load capacity. Such approach is not sufficient to evaluate a self-adaptive system, where quality attributes may have several trade-offs between them in a short time.

Yang et al. [Yang et al., 2009] referred to a dynamic decision model that uses the trade-off of quality attributes to validate and verify the adaptation method at runtime. The method measures the quality attributes at runtime and makes a trade-off change dynamically. The goal of this approach is to guarantee the quality attributes of the target system. Yang et al. extended the traditional quality attributes process, which consists of (1) the estimation of the Quality-of-Service (QOS) using a mathematical model, simulation, or experience-based reasoning; (2) architecture analysis to identify conflict among different QOS; (3) a trade-off solution proposed by the architect for every conflict identified; and (4) a reconstruction of the architecture with the applied solution.

2.8 Summary

With the current state-of-the-art, it is possible to design a system that could adapt its behaviour. However, any adaptation would either have to be pre-defined at design time, or would have to be a reflective response to some monitored parameters, perhaps by using the available techniques. An effective pre-defined response would be dependent on the requirements analyst anticipating and enumerating all the possible environmental states and the corresponding behaviour required. A drawback of the reflective response is that the relationship between the adaptation and the objective goal would be at best implicit, making the verification of goal satisfaction hard or even impossible. The design and the development of context-dependent and self-adaptable software applications in the mobile computing environment cannot rely on the classical software development methodologies, which assume that the software execution environment is known a priori at design time and the application environment can be statically anticipated.

Programming-level adaptation approaches handle context information directly at the code level. Which implies that the context models and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability. Self-adaptability requires both an anticipation method that enables the application to reason about unforeseen context changes and a reasonable mechanism of the adaptation action, which considers the allocated resources and the quality attributes.The MDD-based approaches proposed in the literature suffer from a number of drawbacks. First, it would be optimistic to assert that the process of model transformation and code generation from the software models can become completely automatic and that the developer's role lies only in application design. Second, it is impossible for the developer to predict all possible variations of the application when unanticipated conditions will arise. In addition, mobile devices have limited resources for evaluating many application variations at runtime and can consume significant amounts of device resources. As result, the benefit gained from the adaptation is negated by the overhead required to achieve the adaptation. Third, each development methodology generates more specific technical details that suit the underlying implementation language or modelling tool they used. These challenges motivate this study to explore the possibility for engineering self-adaptive software using a standard and generic development methodology. In addition, the adaptation cost must be sustainable and affordable in conjunction with mobility constraints of the mobile computing environment.

Chapter 3

Context-Oriented Software Development

In this thesis, we explore the level of support of the engineering of self-adaptive applications using a general and standard development paradigm provided by non-specialized language framework like Context-Oriented Programming (COP) [Gassanenko, 1998], Aspect-Oriented Programming (AOP) [Kiczales et al., 1997] and Dynamic Aspect Oriented Programming (DAOP) [Popovici et al., 2002] and not limited to a specific platform or technique, which gives the software designer the flexibility to construct self-adaptive application using a standard programming language and be deployed in several platforms. In addition to this, we will explore design practices that can be used to implement the middleware architecture without relying on a specific framework for performing behavioural activation and dynamic code loading. To address this issues, this chapter focuses on describing the context-oriented software development paradigm. The result of this software methodology is a component-based architecture described by a Context-Oriented Component-based Applications Architecture Description Language (COCA-ADL), that is a platform-independent model transformed by a tool support into the desired platform-specific model. This provides code mobility for the same application into various deployment platforms.

The rest of the chapter is structured as follows. Section 3.1 describes the rationale for providing a development paradigm for context-oriented software. Section 3.2 describes the Context-Oriented Component model (COCA-component). Section 3.3 describes the COCA-ADL elements. Section 3.4 provides an overall description of the Context-Oriented Component-based Applications Middleware (COCA-middleware). The model-driven development methodology, the Context-Oriented Component-based Applications Model-Driven Architecture (COCA-MDA) is described in Section 3.5. Section 3.6 demonstrates a case study designed using the COCA-MDA.

3.1 Rationale

A context-driven adaptation requires self-adaptive software to anticipate its context-dependent variations. A context-dependent variation can be classified into actor-dependent, system-dependent, and environment-dependent behaviour variations. The complexity behind modeling these behaviour variations lies in the fact that they can occur separately or in any combination, and cannot be encapsulated because of their impact across all software modules. Context-dependent variations can be seen as collaboration of individual features expressed in requirements, design, and implementation, and are sufficient to qualify as heterogeneous crosscutting concerns. Heterogeneous crosscutting concerns refers to a set of collaborating aspects, code fragments, that extend the application behaviour in several parts of the program and have an impact across the whole software system, in the sense that different code fragments are applied to different program parts. Before encapsulating crosscutting context-dependent behaviours 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 behaviours are tangled with other behaviours, and are likely to be included in multiple parts 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 analysing them in the software requirements and separating their concerns. Moreover, a formal procedure for modelling these variations is needed. Such analysis and modelling procedures can reduce the complexity in modelling 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 behavioural parts that can be used dynamically to modify the application behaviour based on the execution context.

Behavioural decomposition of a context-aware application can provide a flexible mechanism for modularizing the application into several units of behaviour. Because each behaviour realizes a specific context-dependent functionality, the development methodology 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 behavioural 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. The substates are represented by j, and j might represent a known or unknown conditional state k. Examples of known states in the generic form include detecting context changes in a reactive or proactive manner, so the developers are able to specify decision policy (k), which controls the adaptation in the associated state *(S_i)*. Each decision policy (k) is attached to a decision point *DPj*, which controls the transformation *T(jk)* of the self-adaptive software form $state_i$ into $state_{i+1}$, when the application receives context changes *(Ci)* from the computational environment, as shown in Figure 3.1.

In the presence of uncertainty and unforeseen context changes, a 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 behaviour, which is related to the *Decision Point (DP)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

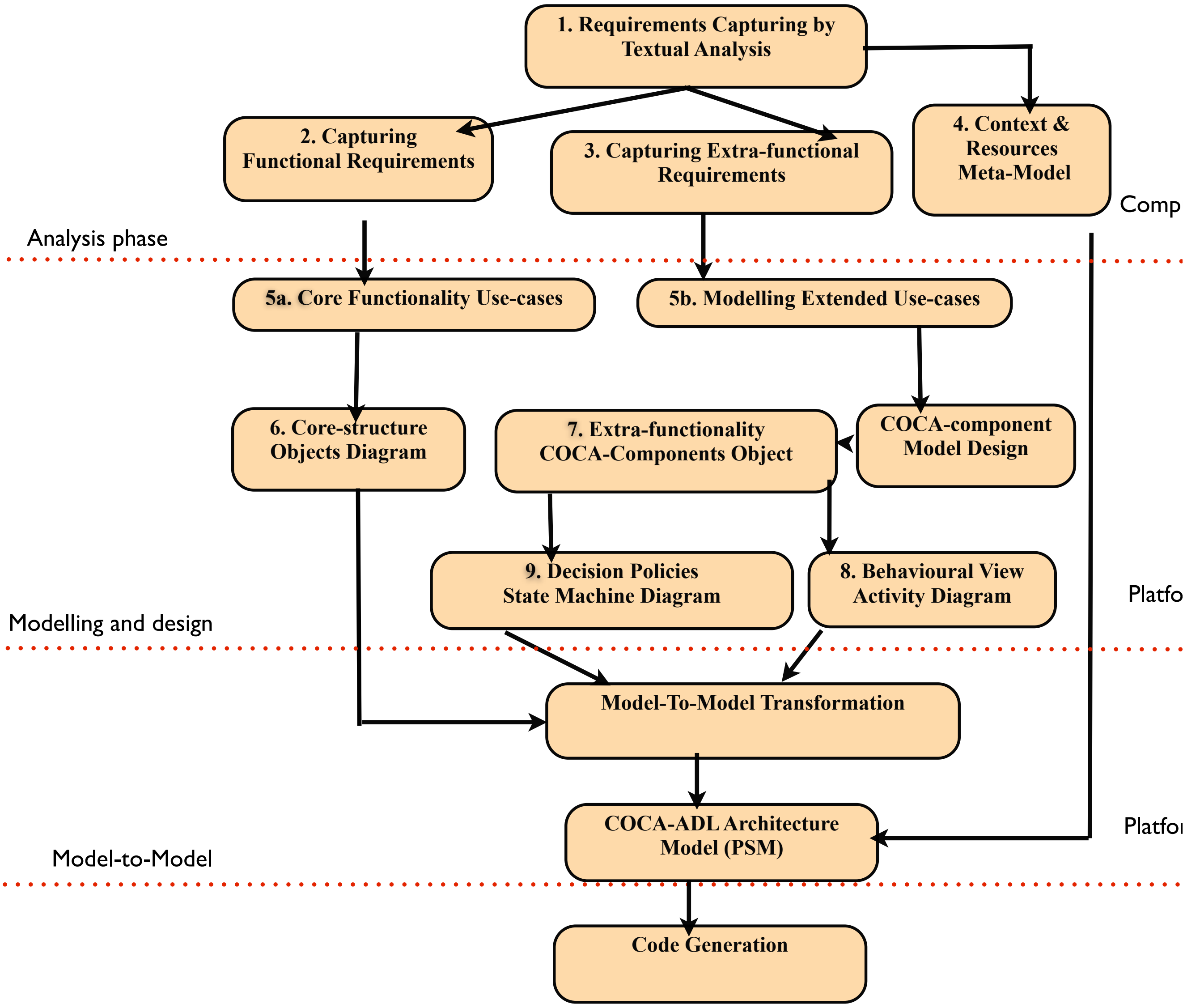

1. Requirements Capturing by Textual Analysis
2. Capturing Functional Requirements
3. Capturing Extra-functional Requirements
4. Context & Resources Meta-Model
Comp
Analysis phase
5a. Core Functionality Use-cases
5b. Modelling Extended Use-cases
6. Core-structure Objects Diagram
7. Extra-functionality COCA-Components Object
COCA-component Model Design
9. Decision Policies State Machine Diagram
8. Behavioural View Activity Diagram
Platfo
Modelling and design
Model-To-Model Transformation
COCA-ADL Architecture Model (PSM)
Platfor
Model-to-Model
Code Generation

Fig. 3.1: Behavioural Decomposition Model

software form $state_i$ into $state_{i+1}$, considering a specific context condition t_{jk}, as shown in Figure 3.1. 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.

Context-driven adaptation requires dynamic composition of context-dependent parts, which enables the middleware to add, remove, or reconfigure components within an application at runtime. Each component embeds a specific context-dependent functionality $(C + S)_{ji}$, realized by a COCA-component. 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 3.1. The developers have the option to provide a decision policy (k) for each DPj for a specific context-related condition. Hereafter, the COCA-components are dynamically managed by COCA-middleware and their internal parts to modify the application behaviour. The COCA-middleware performs context monitoring, dynamic decision-making, and adaptation, based on policy evaluation.

Model-driven approaches provide a mechanism for designing a self-adaptive software using an abstract model and facilitate development by means of code generation. As a result of combining a decomposition mechanism with COCA-MDA, a set of behavioural units are produced. Each unit implements several context-dependent functionalities. This requires a component model which encapsulates these code fragments in distinct architecture units. Each component embeds a specific context-dependent functionality realized by a COCA-component. Each COCA-component realizes several layers. Each layer encapsulates a fragment of code produced by the COCA-MDA. Hereafter, the COCA-components are dynamically managed by COCA-middleware and their internal parts to modify the application behaviour.

3.2 Context-oriented Component Model (COCA-component)

The COCA-component model was proposed by Magableh and Barrett [Magableh and Barrett, 2009], based on the concept of a primitive component introduced by Khattak and Barrett [Khattak and Barrett, 2009] and COP [Hirschfeld et al., 2008]. COP provides several features that fit the requirements of a context-aware application, such as behavioural compo-

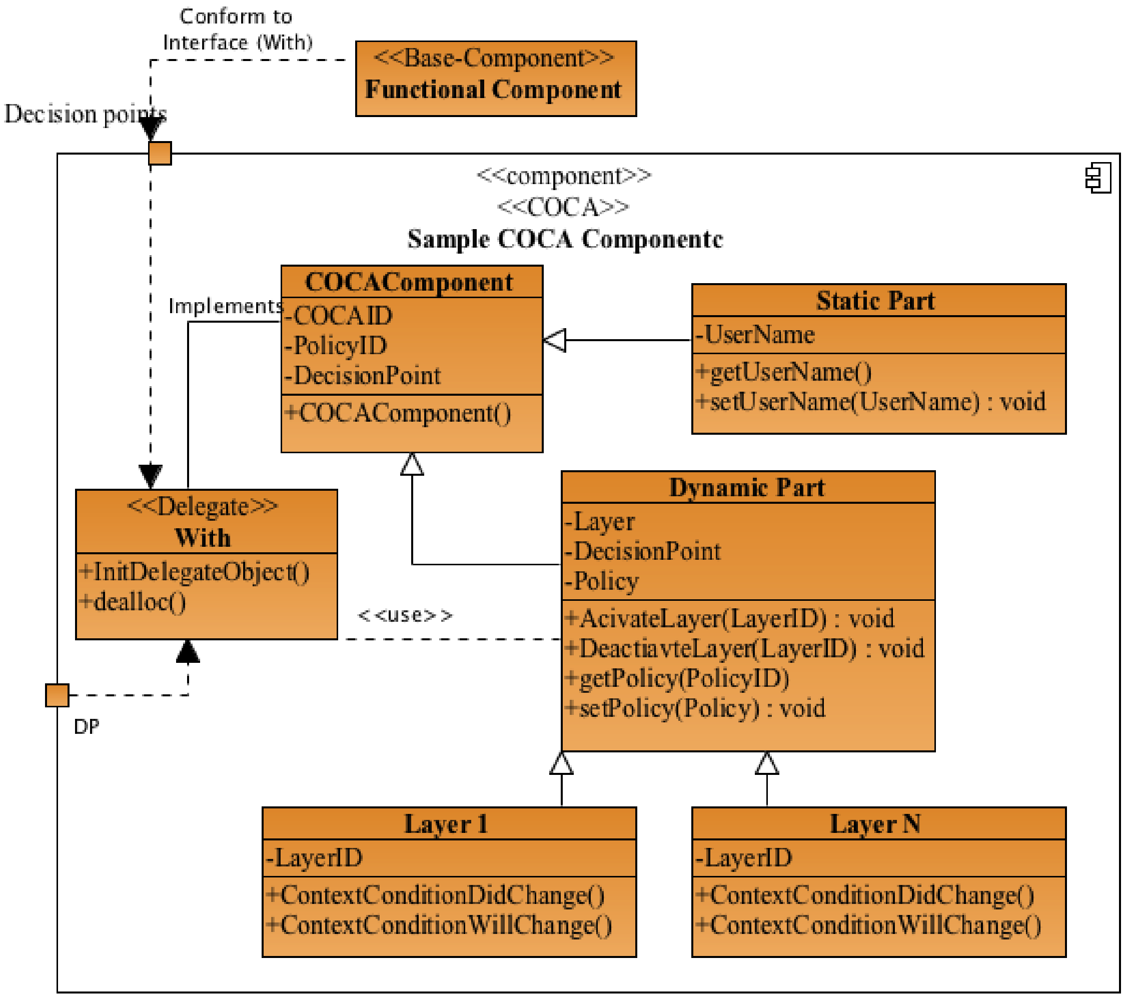


Fig. 3.2: COCA-component Conceptual Diagram

sition, dynamic layer activation, and scoping. This component model dynamically composes adaptable context-dependent applications based on a specific context-dependent functionality. The developers build the application model by designing components as compositions of behaviours, embedding DP in the component at design time to determine the component behaviours, and supporting reconfiguration of Decision PoLicys (DPLs) at runtime to adapt behaviours.

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, DPs, policy id and a protocol methods. These methods are used by the middleware to read the attached PolicyID and manipulate the application behaviour by manipulating the DPL.

The COCA-component handles the implementation of a context-dependent functionality through employing the delegate design pattern [Buck and Yacktman, 2010]. Using the delegate pattern allows the adaptation manager to invoke a subdivision of COCA-component sub-layer implementation when a specific condition is found in the execution. A delegate is a component that is given an opportunity to react to changes in another component or influence the behaviour of another component. The basic idea is that two components coordinate to solve a problem. A COCA-component is 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 the same time. The messages may only inform the delegate that something has happened, giving the delegate an opportunity to de/activate a layer implementation, 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 [Buck and Yacktman, 2010].

At this stage, each COCA-component must adapt the COCA-component model design. A sample COCA-component is shown in Figure 3.2. Each COCA-component is modelled 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 *ContextConditionDidChange* and *ContextConditionWillChange*, are called when the context manager posts the notifications in the form *[NotificationCenter Post:ContextConditionDidChange]*. This triggers the class layer to invoke its method *ContextConditionDidChange*, which embeds a subdivision of the COCA-component implementation.

3.3 COCA-ADL: A Context-oriented Component-based Application ADL

The aim of this section is to introduce the architecture description language COCA-ADL [Magableh and Barrett, 2010]. 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 design and the runtime model of the application. Thus, it enables the architecture to be implemented by several programming languages.

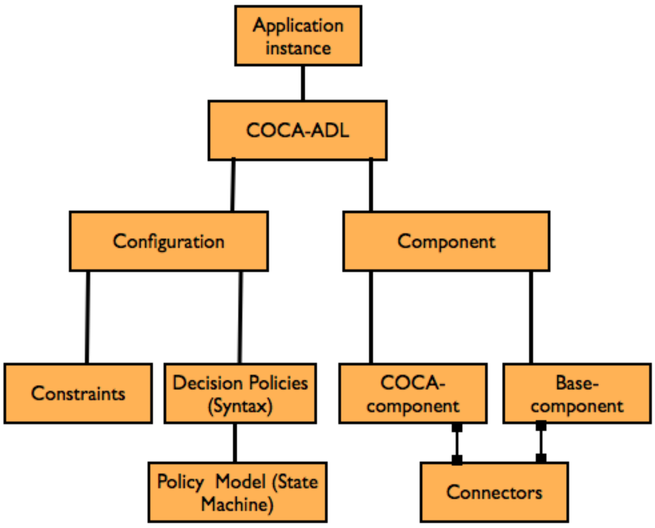


Fig. 3.3: COCA-ADL Elements

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 architecture configurations, which describes using an activity diagram plus the DPLs' syntax and architecture constraints. Figure 3.3 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.

Each element is inherited from a complex entity type, for example, a component inherited from the COCA-component model. The component model is described in the meta-model in Figure 3.2. The component element is used to define the application base-component as well as the COCA-component. The connectors are used to connect components through the interfaces. The configuration element describes the external composition, which is achieved through the connectors and the internal composition, which is used to describe the realization of component's sub-layers through the delegate interface.

3.4 Overview of the COCA-middleware

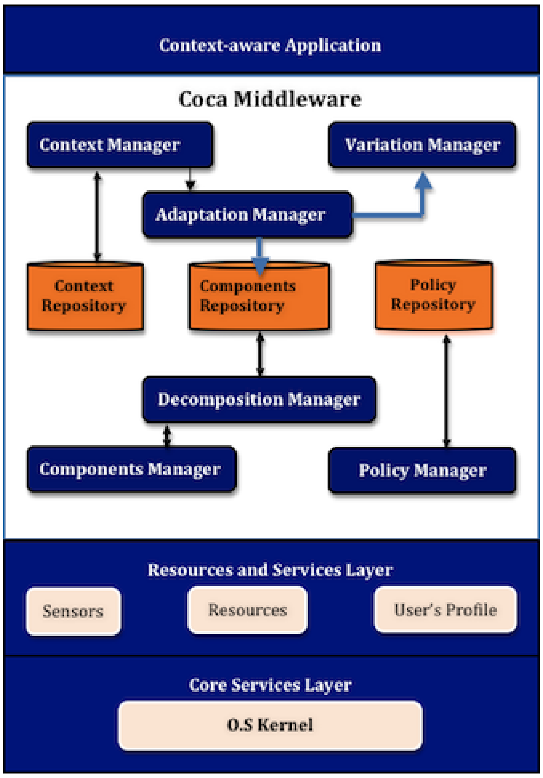


Fig. 3.4: COCA-platform Architecture.

The COCA-platform offers a context-aware middleware environment for adjusting the application's behaviour dynamically [Magableh and Barrett, 2011]. Figure 3.4 shows the COCA-middleware architecture. The platform is layered into four major layers. Each layer provides an abstraction of the underlying technology. Each layer is 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. The COCA-middleware subcomponents are shown in Figure 3.4. The OS sensor retrieves information about the OS. Function calls are used to retrieve information about CPU, memory, and disk space [Magableh and Barrett, 2009].

3.4.1 Context Manager

The first component of the COCA-middleware is the context manager, as shown in Figure 3.4. The context manager gathers and detects context information from the sensors. If the context is changed, the context manager notifies the adaptation manager and the observer COCA-component about the changes. Each COCA-component is designed to be an observer for one or more context entities. This type of interaction is called context binding.

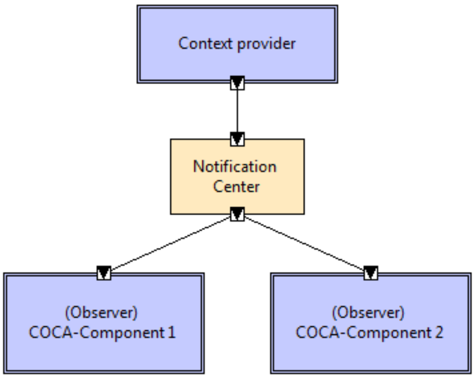


Fig. 3.5: Observer Design Pattern

The observer pattern reduces the tight coupling between the context provider, e.g. context entity, and the context consumer, e.g. COCA-component. In addition, it enables the middleware to identify which COCA-component has to be manipulated in response to context changes. Figure 3.5 demonstrates the observer design pattern with one context entity and two observers. At runtime, the COCA-component registers itself as an observer for the context entity by sending a registration request to the notification centre. The context-change event is sent to the notification centre queue instead of the COCA-component, then the notification centre broadcasts the context changes, and only the registered component receives the notification. COCA-components 1 and 2 have registered as an observer for the context entity. Whenever the context changes, COCA-components 1 and 2 are notified by the notification centre. In this way, the adaptation manager can identify COCA-components 1 and 2 to be included in the adaptation action, which embeds a subdivision of their implementation by de/activating the associated sub-layer.

Supporting context-binding mechanisms with observer pattern provides a clear separation between the context provider and consumer. In addition, it modularizes the application component based on context. This makes identifying which component must respond to a specific context condition an easy task in the design phase. To achieve this integration, the developers have to consider the following two aspects in the application design: how to notify the adaptation manager about context changes, and how the component manager can identify the parts of the architecture that have to respond to these changes.

3.4.2 Component Manager

The component manager performs three major functions in the middleware: It searches for a COCA-component in the component repository, adds components from the repository, and provides COCA-component instantiation. The intercession operation is achieved by the component manager by adding a component, or a component sub-layer, to the application structure. To add a component, the adaptation manager asks the component manager to instantiate a specific component. The component manager performs several inspections of the application components through the operation time of the software.

3.4.3 Policy Manager

The COCA-MDA provides the developers with the ability to specify the adaptation goals, actions, and causes associated with several context conditions using a policy-based framework. For each COCA-component, the developers can embed one or more DPLs that specify the architecture properties. The DPL is described by a state-machine model based on a set of internal and external variables and conditional rules. The rules determine the true action or else an action based on the variable values. The action part of the state diagrams usually involves invoking one or more of the component's layers. A single layer is activated if a specific context condition is found, or deactivated if the condition is not found [Anthony et al., 2009]. The policy manager uses the DPL objects to store policies in the policy repository. The DPL is stored in the policy repository, which conforms to the Associative Storage design pattern [Buck and Yacktman, 2010]. This pattern organizes the policies into data and keys; this reduces the computation overhead from processing them at runtime.

3.4.4 Adaptation Manager

The adaptation manager starts the adaptation process after receiving the notifications that identified the context changes and the COCA-components that observed the notification. The first function of the adaptation manager is to produce the composition plan. The composition plan recursively describes the composite components and the connections between them by describing several connectors and interfaces. To construct a composition plan, the following information is needed by the adaptation manager. 1) A component graph: The component graph is generated by the decomposition manager after parsing the COCA-ADL XML file. At development time, the application's models are transformed into a COCA-ADL XML file. At runtime, the decomposition manager reads the COCA-ADL XML file, then adds the architecture instances, including the components, connectors, and configuration, to the application graph. 2) Decision PoLicy (DPL): The DPL rules determine the true action or the else action based on the values of its variables. 3) Runtime structure style: When several context conditions are found at the same time, the DPL proposes a runtime instance of a design pattern, which may imply combination of multiple components or their internal parts to fulfil the execution context. The structure style describes a structure modification combining a set of the component's layers.

The adaptation process starts the adaptation action, including two types of composition mechanism: internal composition and external composition. In internal composition, the adaptation manager switches a component's layers on or off, based on the composition plan, using the delegation and decorator patterns. The decorator pattern can be used to extend, decorate, the functionality of a certain object at run-time, independently of other instances of the same component. In internal composition, the adaptation manager introspects the application's graph. The component sub-layers are activated by redirecting the COCA-component delegate to the desired layer.

In external composition, the adaptation manger adds or replaces components from the application structure, based on the composition plan. The decomposition component builds the application graph by reading the COCA-ADL. In external composition, a COCA-component is loaded into the application. This require the adaptation manager to confirm to the bundle pattern [Buck and Yacktman, 2010]. The bundle pattern achieves the following goals: 1) Keep executable code and related resources together even when there are multiple versions and multiple files involved in the underlying storage. 2) Implement a flexible plug-in mechanism that enables dynamic loading of executable code and resources. In addition, the invocation design pattern is used to provide a means of capturing runtime messages so that they can be stored, rerouted, or treated and manipulated according to the context state, and allows new messages to be constructed and sent at runtime without requiring code recompilation process [Buck and Yacktman, 2010]. For example, when a component receives a message, a method implementation is usually invoked to handle the message. However, this is not always the case. As an illustration, if a component does not implement a particular method, then there is no method, which can be invoked and a runtime exception is raised instead. Because of the Invocation design pattern, it is possible for a message to be delayed, rerouted to other components, or even ignored at runtime without re-compiling the application's code.

In addition, the adaptor design pattern lets components work together, even if they have incompatible interfaces [Buck and Yacktman, 2010]. Assume a base-component needs to communicate with a COCA-component, but its interfaces make that unachievable. To solve this problem, the COCA-component applies to the delegate pattern by defining a protocol, which is essentially a series of method declarations unassociated with the component. The base-component then adapts the protocol and confirms this by implementing one or more of the protocol's methods. The protocol may have mandatory or optional methods. The base-component can then send a message to the protocol interface. At this stage, the adaptation manager can verify whether the COCA-component is responding to the message, before invoking the message call by adapting the chain of responsibility pattern. This pattern verifies whether the component can respond to the method call using the responder pattern, which avoids coupling between the sender of a request and its receiver by giving more than one COCA-component sub-layers a chance to handle the request.

3.4.5 Verification Manager

As long as the COCA-middleware is aware of the architecture configuration, which is supported by the COCA-ADL configuration element. The COCA-middleware can anticipate the associated configuration with specific context changes. In each DP, the COCA-middleware transforms the software from $state_i$ into $state_{i+1}$, considering the properties of the self-adaptive software. These properties include the following: 1) The set of DPLs attached to the COCA-components that participate in the adaptation; 2) the architecture configuration elements in the COCA-ADL, which include the description of the DPLs and the behavioural model of the architecture, and the external and internal variables specified in the DPLs embedded in the COCA-components that are evolving through the adaptation process; and 3) the adaptation goals, actions, rules, and causes specified by the DPLs.

3.5 COCA-MDA Development Approach

The COCA-MDA follows the principles of Object Management Group (OMG) model-driven architecture. In Model Driven Architecture (MDA), there are three different viewpoints of the software: the Computation Independent View (CIV), the Platform Independent View (PIV), and the 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 [Miller and Mukerji, 2003].

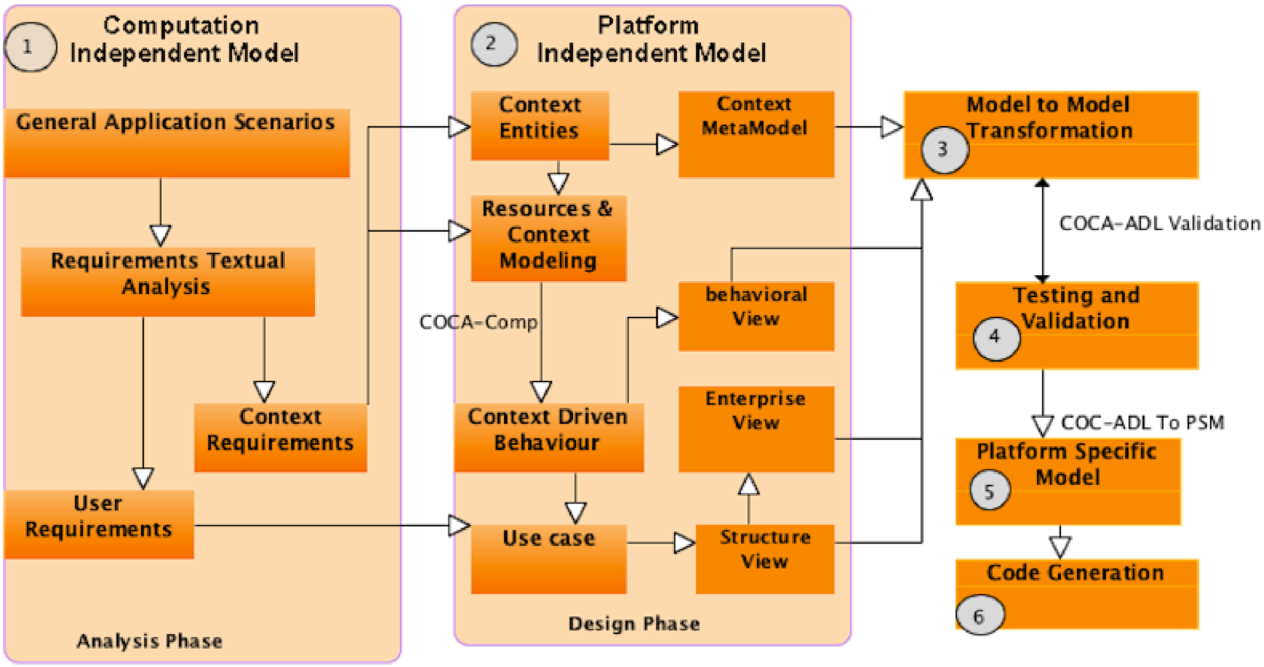


Fig. 3.6: Context-oriented component-based application model-driven architecture (COCA-MDA)

In Enterprise Collaboration Architecture (ECA) [ECA OMG, 2004]. the 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 components, that can be applied 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 adapted 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 [ECA OMG, 2004].

The design of a context-oriented component-based application according to the COCA-MDA generally involves the six phases shown in Figure 3.6. COCA-MDA 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.

Modelling self-adaptive software using COCA-MDA can be summarised as shown in Figure 3.7. The figure summarizes the modelling tasks using the associated UML diagrams. The developer can start the analysis of an application scenario to capture the requirements.

Analysis: The requirements of the system are modelled in a computation-independent model (CIM), thus describing the situation in which the application will be used and predicting the exact behaviour of the application as a result of runtime context changes. In this phase, the developers perform a separation of concerns between the functional and contextual requirements.

Modelling and design: The PIV focuses on the operation of a system while hiding the details necessary for use of a particular platform. In this phase, the requirements diagram is combined into a use-case model. The use-cases describe the interactions between the software system and the actors. 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 that describes the application's behavioural view.

The use-case diagram is split into two distinct class diagrams. The first diagram describes the basic application components that are executed regardless of the execution context. The core structure is integrated with the extra-functional class model in the final architecture model. The extra-functionality class diagram provides a detailed view of the application COCA-component and the COCA-middleware. In addition, these diagrams model the desired behaviour that can be used to anticipate context changes.

COCA-MDA has adapted CCA [ECA OMG, 2004] at the Platform Independent Model (PIM) phase by partitioning the software into two models: The structure model and the behavioural model. The structure model focuses on the core components of the self-adaptive application and hides the context-dependent components. The behavioural model focuses on modelling the context-driven behaviour of the component, which may be invoked in the application execution at runtime, which may satisfy the execution context. 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 time. Using the activity diagram, the developers can extract numerous decision polices.

Model-to-model transformation: The platform-independent model and behavioural model are translated into COCA-ADL. This phase includes model-to-model transformation and model verification for the application's structure and behaviour views. The COCA-ADL is implemented by extending the xADL schema, which is an extensible XML language. ArchStudio is a modelling tool, that helps the developers to model the architecture using three grouped models: activity diagram, state diagram, and structure diagram [Dashofy et al., 2007].

Testing and validating: This process used to test the model and verifies its fitness for the application goals and objectives.

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 Platform Specific Model (PSM) may provides 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. Alternatively, it may act as a PIM used to further refine the PSM so that it can be directly implemented.

3.6 Context-Oriented Component-based Application Example

I-TrinityTour is a tourist guide application that helps the user to explore the historical campus of Trinity College Dublin, Ireland (TCD). I-TrinityTour offers a map–client interface maintained by an Augmented Reality Browser (ARB). The browser exhibits many Places Of Interests (POIs) inside the physical outlook of the tool's camera. Information related to every POIs is exhibited inside the camera overlay outlook. The POIs comprise edifices, tourist services sites, restaurants, hotels, and ATMs in Trinity college. 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 requires the application to self-tune its behaviour among several contexts to maintain quality of services without disrupting the function's tasks. To demonstrate the I-TrinityTour ability to self-adjusting its behaviour among the battery usage, and self-configuring its structure for adapting a service with better quality, the following scenarios are proposed:

A1: I-TrinityTour is required to adapt its behaviour and increase the battery life. This is achieved by adapting a location service that consumes less power. For example, if the battery level is low, the I-TrinityTour application switches off the GPS location services and uses the cell-tower location services. Using the cell-tower for updating the 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, I-TrinityTour uses the GPS service with more accurate locations. The application starts listening for all events in the monitored region inside Trinity College.

A2: The user takes photographs for the locations inside Trinity College, using the cameras flash because of the low quality of light. At the same time, the devices battery is drained and needs recharging. In such a situation, I-TrinityTour reconfigures itself so that it stops taking photographs and using the cameras flash. The ability to use the flash depends on the level of light inside the room when the application senses a low level of light. This will enable the flash automatically, unless the battery is drained. To reason about these conditions, the application could search for photographs taken by any nearby devices. In addition to the previous set of contexts, the devices storage capacity has to be monitored by the application. While performing the adaptation action, the remaining battery power is insufficient for selecting from the available photographs in the distributed environment, so the application adapts a sorting component, which is able to select photographs based on their quality, picture tags. Once the photographs are stored in users device, he/she can upload a few of them to one of the social networking sites.

A3: The application must be able to guide the user towards the place of interest. The route directions can be delivered to the user in several output formats: video, still image, and voice command. The application should change the direction output while also considering the device resources.

Figure 3.7 summarizes the modelling tasks, using the associated UML diagrams. The developer starts the analysis of an application scenario to capture the requirements. The requirements are combined in one model in the requirements diagram. The requirements diagram is modelled 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 behavioural view. Finally, the core structure, the behavioural models, and the context model are transformed into the COCA-ADL model.

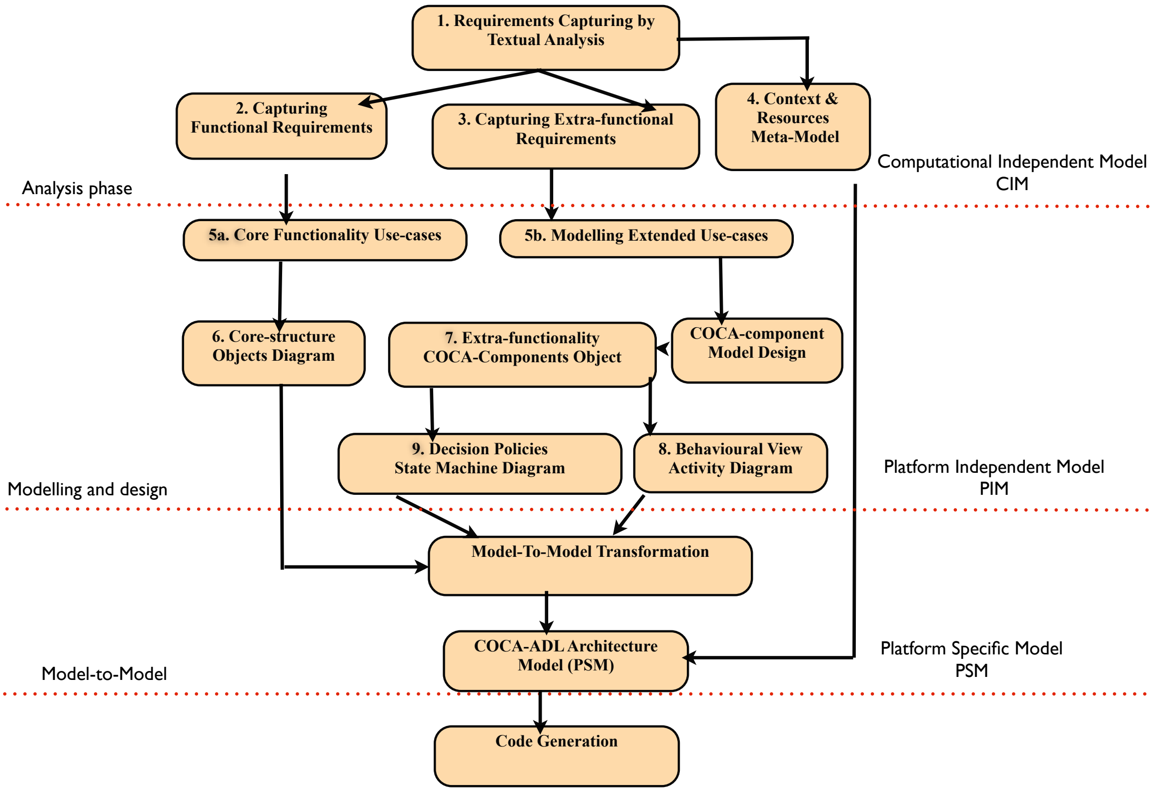

1. Requirements Capturing by Textual Analysis
2. Capturing Functional Requirements
3. Capturing Extra-functional Requirements
4. Context & Resources Meta-Model
Computational Independent Model CIM
Analysis phase
5a. Core Functionality Use-cases
5b. Modelling Extended Use-cases
6. Core-structure Objects Diagram
7. Extra-functionality COCA-Components Object
COCA-component Model Design
9. Decision Policies State Machine Diagram
8. Behavioural View Activity Diagram
Platform Independent Model PIM
Modelling and design
Model-To-Model Transformation
COCA-ADL Architecture Model (PSM)
Platform Specific Model PSM
Model-to-Model
Code Generation

Fig. 3.7: Modelling tasks

3.6.1 Analysis: Capturing Context and User Requirements

In the analysis phase, the developers analyse several requirements using separation of concerns technique, which focus on separating the functional requirements from the extra-functional requirements as the first stage. Extra-functional refers to contextual requirements [Broens et al., 2007] that extend the application behaviour when a specific condition found in the execution context [Hochmuller, 1999, Geihs et al., 2006, Paspallis, 2009]. Extra-functional includes the context, non-functional and technological requirements. There are two subtasks in the analysis phase.

3.6.1.1 Task 1: Requirements capturing by textual analysis

In this task, the developers identify the candidate requirements for the application using a textual analysis of the application scenario. In this task, the developers identify 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.

3.6.1.2 Task 2: Identifying the middleware functionality:

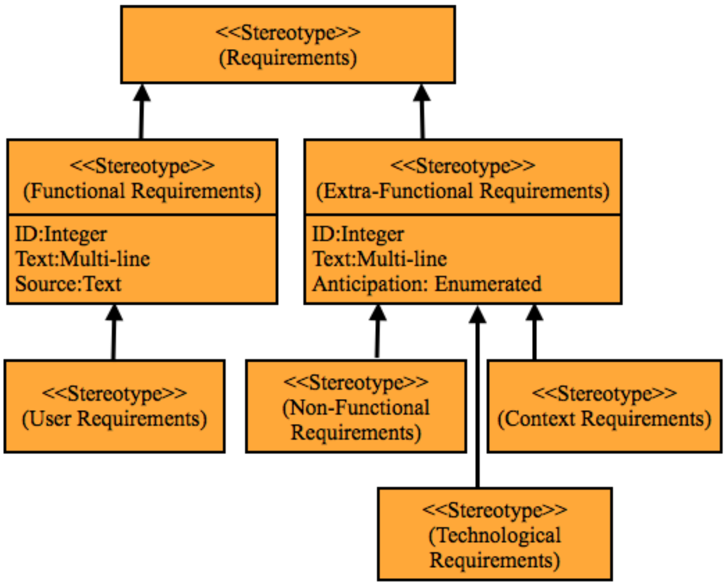
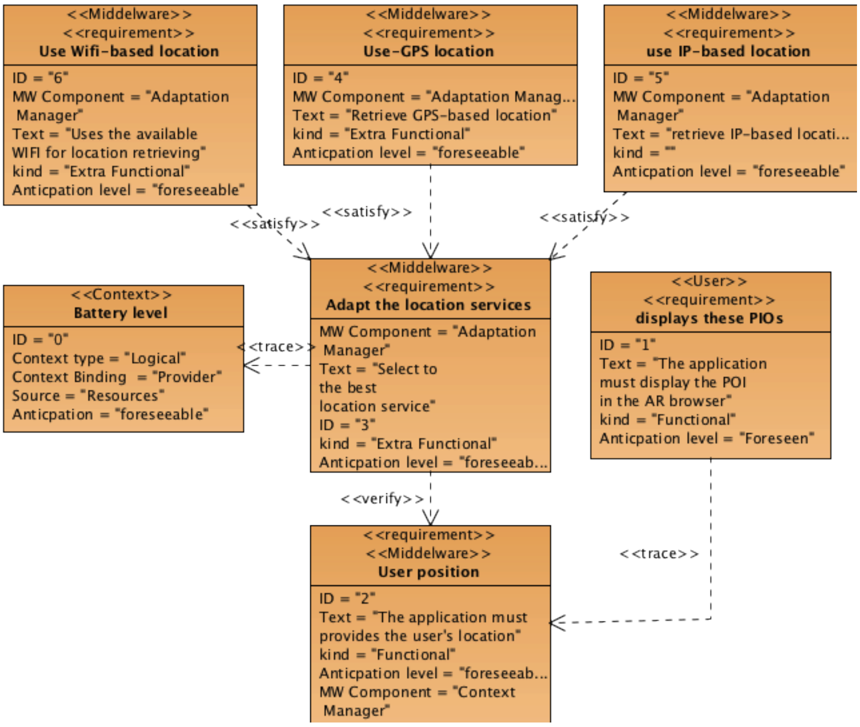


Fig. 3.8: Requirements UML profile

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 provided by a physical or logical source, i.e. available memory, or resources, i.e. battery power level and bandwidth. Context consumer refers to a software component, which provides a representation of context information to the application. Context consumer refers to the parts of the application, which handles, manipulates, or analyses the context information.

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 behaviour 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 3.8.



<<Middelware>>
<<requirement>>
Use Wifi-based location
ID = "6"
MW Component = "Adaptation Manager"
Text = "Uses the available WIFI for location retrieving"
kind = "Extra Functional"
Anticpation level = "foreseeable"
<<Middelware>>
<<requirement>>
Use-GPS location
ID = "4"
MW Component = "Adaptation Manag...
Text = "Retrieve GPS-based location"
kind = "Extra Functional"
Anticpation level = "foreseeable"
<<Middelware>>
<<requirement>>
use IP-based location
ID = "5"
MW Component = "Adaptation Manager"
Text = "retrieve IP-based locati...
kind = ""
Anticpation level = "foreseeable"
<<satisfy>>
<<satisfy>>
<<satisfy>>
<<Context>>
Battery level
ID = "0"
Context type = "Logical"
Context Binding = "Provider"
Source = "Resources"
Anticpation = "foreseeable"
<<trace>>
<<Middelware>>
<<requirement>>
Adapt the location services
MW Component = "Adaptation Manager"
Text = "Select to the best location service"
ID = "3"
kind = "Extra Functional"
Anticpation level = "foreseeab...
<<User>>
<<requirement>>
displays these PIOs
ID = "1"
Text = "The application must display the POI in the AR browser"
kind = "Functional"
Anticpation level = "Foreseen"
<<verify>>
<<trace>>
<<requirement>>
<<Middelware>>
User position
ID = "2"
Text = "The application must provides the user's location"
kind = "Functional"
Anticpation level = "foreseeab...
MW Component = "Context Manager"

Fig. 3.9: Functional and extra-functional partial requirements diagram

As shown in Figure 3.9, extra-functional requirements are captured during this task, for example, requirement number 3: adapt the location service. I-TrinityTour is required to adapt its behaviour and increase the battery life. This is achieved by adapting a location service that consumes less power. For example, if the battery level is low, the I-TrinityTour 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, I-TrinityTour uses the GPS service with more accurate locations. The application starts listening for all events in the monitored region inside Trinity College.

In order to meet the middleware functionality that extends the application behaviour based on the battery level. The software designer associates a trace relationship with the context requirement “Battery level”. Requirements tracing was proposed by Jarke et al. [Jarke, 1998], to help the developers to align software system evolution with changing stakeholders needs. However, The “Battery level” requirement needs a middleware functionality to manage its context changes and take the adaptation actions that satisfy them. For example, displaying the POIs in the camera browser is a functional requirement that drives the extra-functional requirement number 4: use GPS-based location. This requirement is classified as a foreseeable anticipation level. The middleware traces the battery level and the bandwidth connectivity. If the bandwidth speed is greater than a specific limit, a WIFI-based location is used to save the battery energy, otherwise the middleware will decide to use the GPS-based location if the battery is not drained.

3.6.1.3 Task 3: Capturing user requirements

This task is combined with the previous requirements diagram. This task focuses on capturing the user’s requirements as a subset of the functional requirements, as shown in the UML profile in Figure 3.8. This task is similar to a classical requirement-engineering process where the developers analyse the main functions of the application that achieve specific goals or objectives.

3.6.2 Modelling: Platform-Independent Model

In order to be aware of possible context variations and the necessary adaptation actions, a clear analysis of the context environment is the key to building dynamic context-aware applications. As result of the analysis phase, the developers should be able to provide a design for the context entities and their dependences, and separate the model of the application core-structure from the context-driven behavioural variations.

3.6.2.1 Task 4: Resources and context entity model

Resources and context model refer to a generic overview of the underlying devices resources, sensors, and logical context provider. Such a diagram models the engagement between the resources and the application under development. It helps the developer to understand the relationship between them and their dependences. Figure 3.10 shows the general resources and context for a mobile device. All resources, physical sensors, and logical context providers are modelled. In addition, the developers identify provisional assumptions about their expected conditions or values.

The dependencies between context entities are also specified in this model. This task helps developers to understand the relationship between each context provider, such as an accelerometer, and the context representation, which is the speed value. In the same way, memory is a device resource providing context, but the available memory is a context entity representing the context value. The context-entity representation can be modelled using the context model shown in Figure 3.11.

3.6.2.2 Task 5: Use-cases

The requirements diagram in Figure 3.9 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

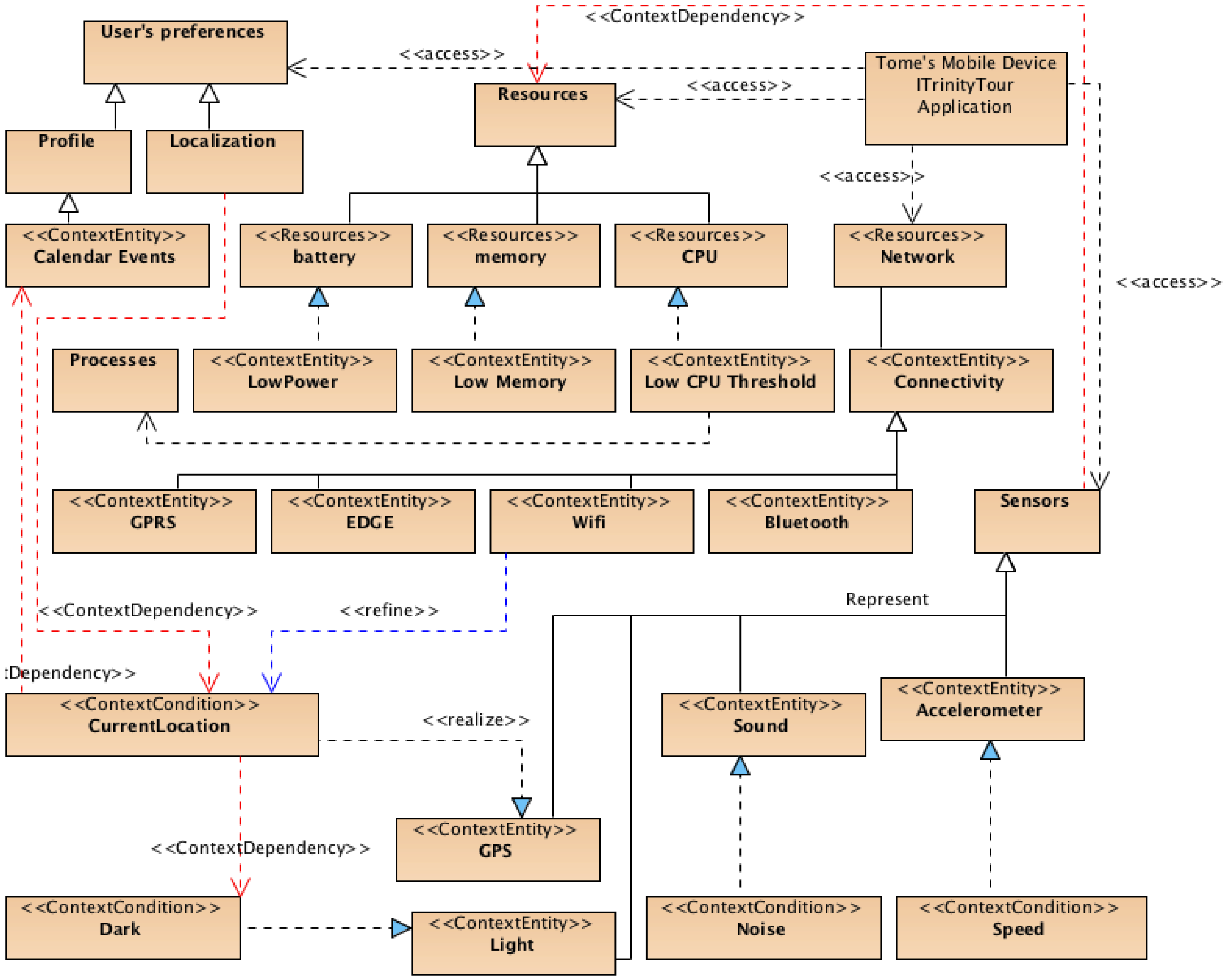


Fig. 3.10: Resources and context entities model

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 modelled separately, using a class diagram that describes the application core-structure or the base-component model, as shown in the following task.

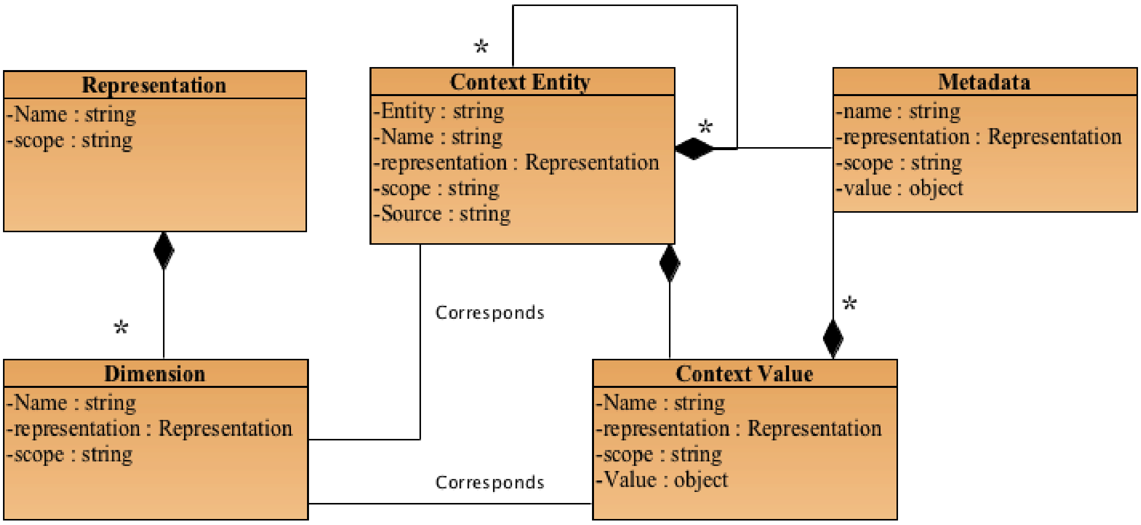


Fig. 3.11: Context meta-model

The use case "enables the stream-receiving functionality" is a use case extended by the context, and it extends other use cases. In the same way, the use case "Stop the camera flashes" is extended by the context change battery energy. Such an interaction is called an environment-dependent variation. The developers scan and search the use-case model for the use case that are extended by the context changes, then each use case is modelled separately in a sequence diagram. The sequence diagram shows the lifeline and the message interaction among the application, middleware, and the context environment, while achieving the desired functionality of the use case. The three use-cases, "Enables the stream-receiving functionality", "Searches for pictures", and "Stop the camera flashes", are identified as an extra-functionality managed by the middleware.

3.6.2.3 Task 6: Modelling 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 modelled independently from the variations in the context

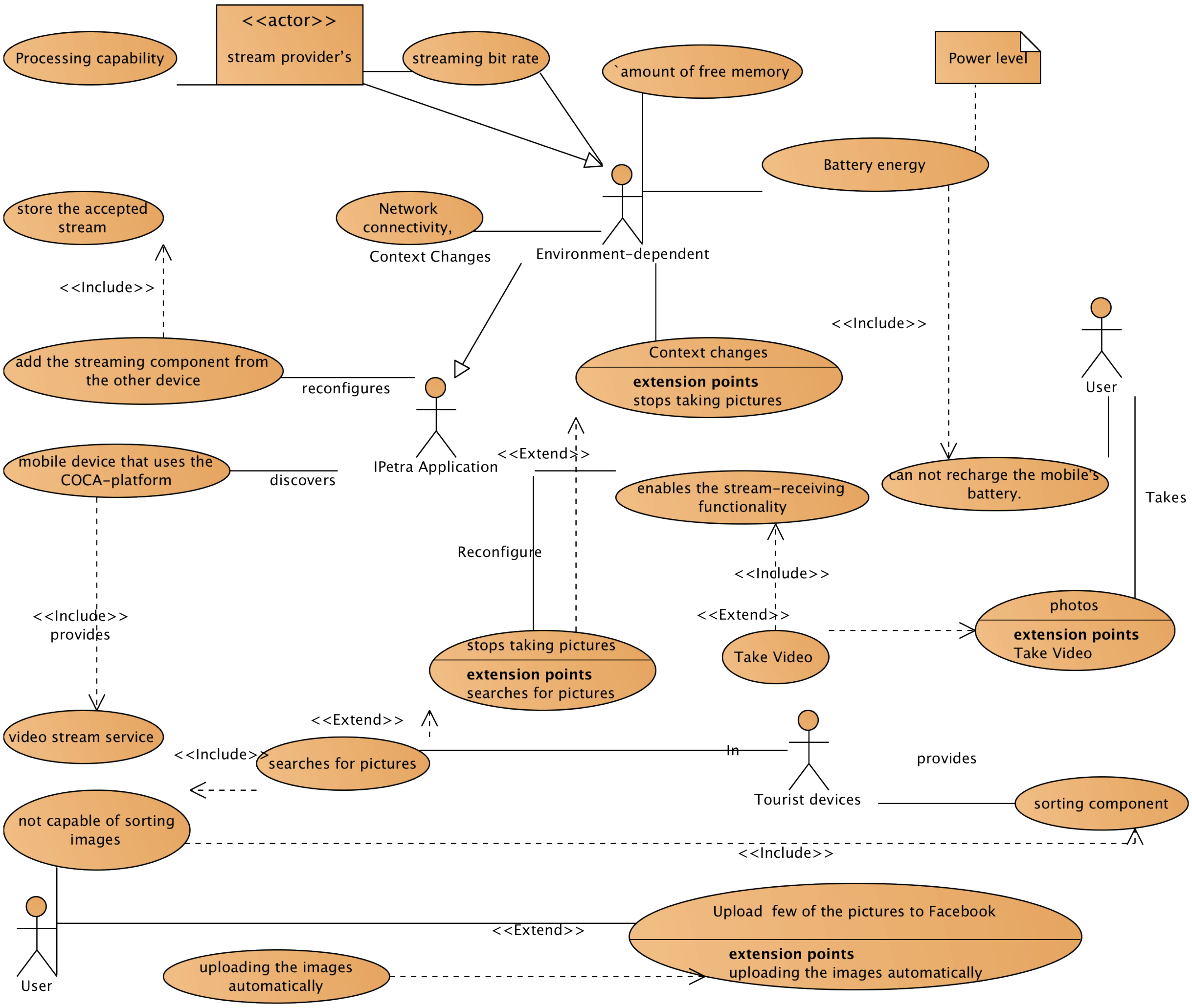


Fig. 3.12: Use case model

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 components. These classes provide the core functions for the user during his tour inside Trinity College. Figure 3.13 shows the core-structure class-model without any interaction with the context environment or the middleware.

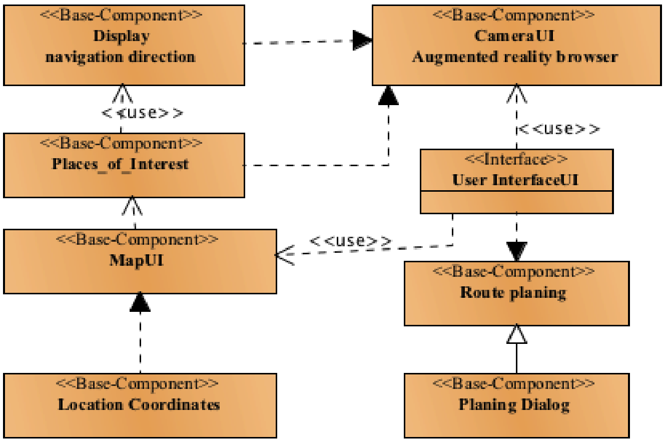


Fig. 3.13: I-TrinityTour Core-Classes structure

3.6.2.4 Task 7: Identifying application behavioural variations (behavioural model)

In this phase, the developers specify how the application can adapt to context conditions to achieve a specific goal or objectives. 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 sub-divisions, 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 how the component manager can identify the parts of the architecture that have to respond to these changes. These aspects can be achieved by adapting the notification design pattern in modeling the relation between the context entity and the behavioural component. Hereafter, these extra-functionalities are called the COCA-components. Each component must be designed on the basis of the component model described in Section 3.2.

The I-TrinityTour application is modularized into several COCA-components. Each component models one extra-functionality such as the $LocationCOCA - component$ in Figure 3.14. 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 modelled as shown in Figure 3.14.

Figure 3.14 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.

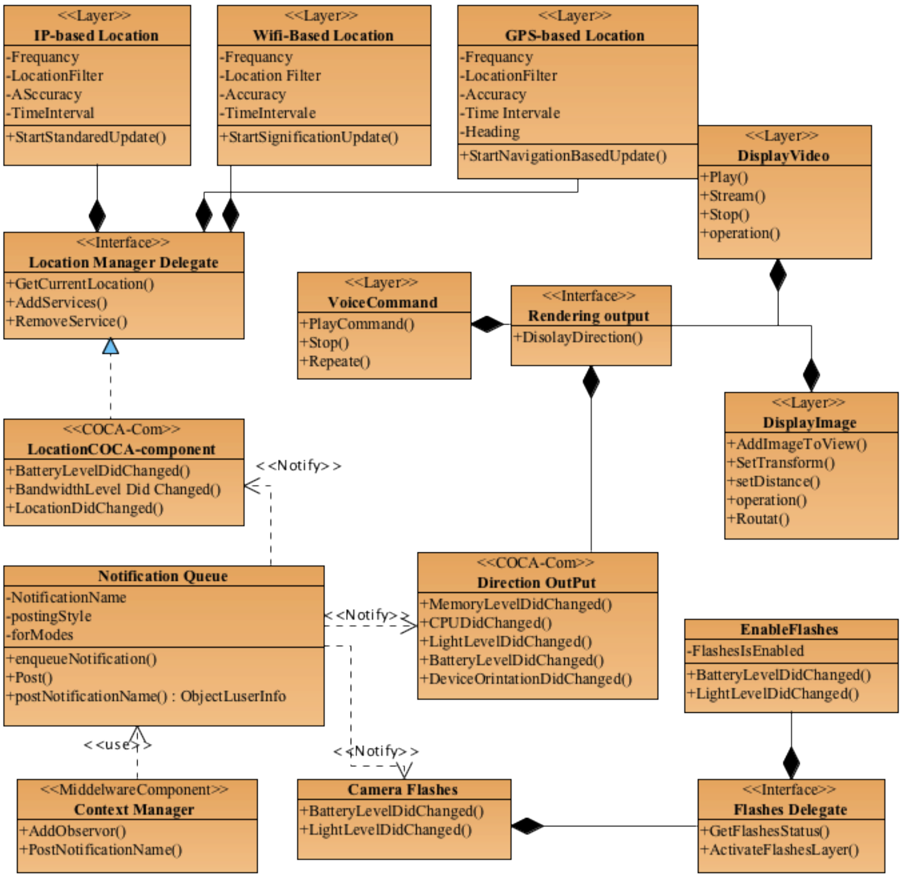


Fig. 3.14: Extra-functionality Object Diagram of the Context Oriented Components

Invoking different variations of the COCA-component requires identification of the application architecture, behaviour, and the DPLs in use. As mentioned before, these DPLs play an important role for the middleware functionality, which use them in handling, the architecture evolution, and the adaptation action. The model in Figure 3.14 helps the developer to extract the DPLs and the DPs from the interactions between the context entities and the COCA-components.

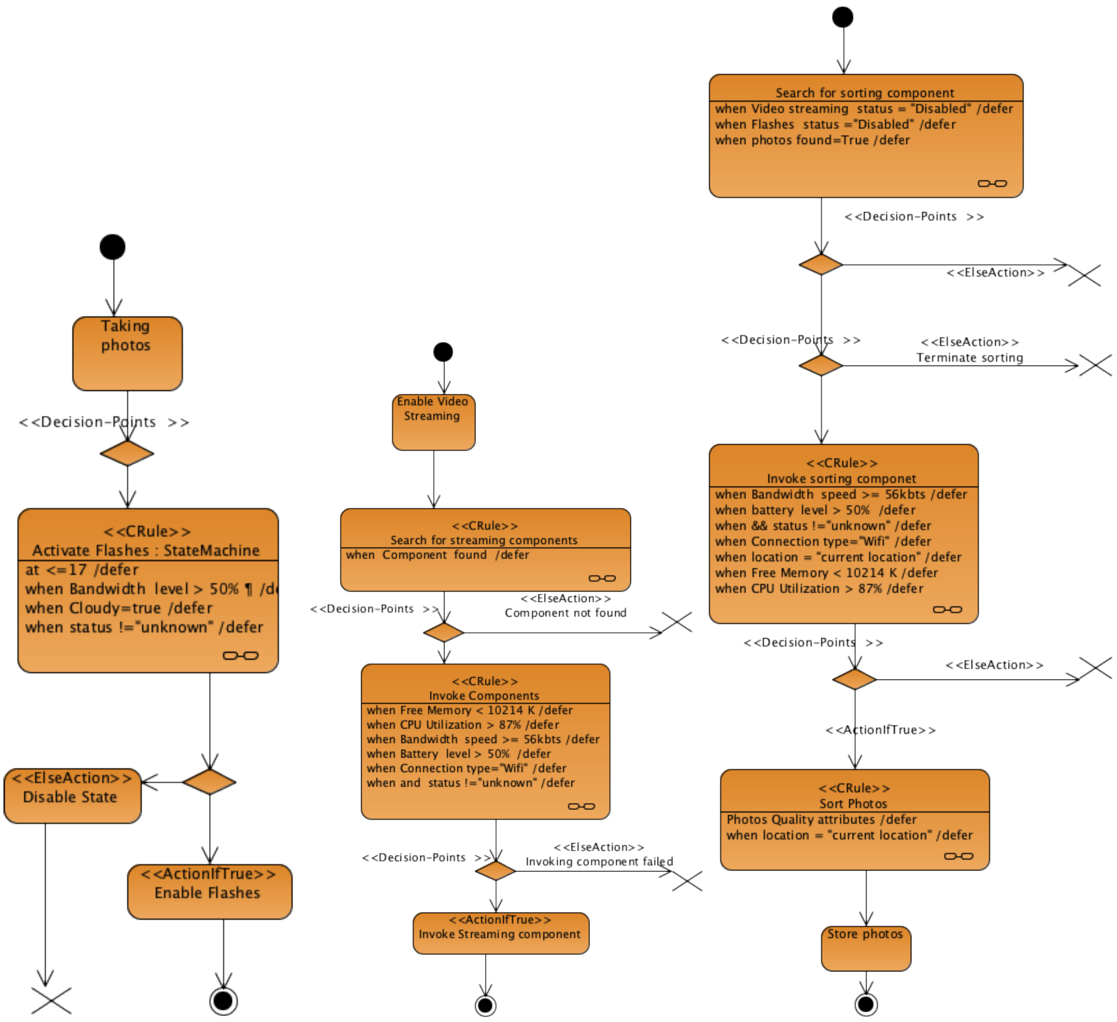
To reduce the development effort and increase the understanding of the policy framework, a UML profile for specifying a DPL was provided. The UML profile adapts the policy-definition language introduced by Anthony et al. [Anthony et al., 2009]. These DPLs are modelled using a state diagram; each state diagram is controlled by a class entity from the previous objects diagram. We can extract the following policies from the behavioural model.

1. Policy 1: Use flashes if the location is dark or the time is after 5 pm and if the battery power level is > 50. This policy is modelled using the UML activity diagram shown in Figure 3.15a.

2. Polciy 2: If the user enables video streaming, search for components in nearby devices if connectivity uses wifi, bandwidth speed > 56 kbit, battery power 50, no low memory, and CPU not busy. This policy is demonstrated in Figure 3.15b.

3. Policy 3: Search for photographs taken by nearby devices. If sorting is not available locally, search for sorting component. Use sort component to select the photograph based on its quality, size, and location. Store policy in local devices if connectivity uses wifi, bandwidth speed > 56 kbit, battery power > 50, no low memory, and CPU is not busy. This policy is demonstrated in Figure 3.15c.

The application behavioural model is used to demonstrate the DPs in the execution that might be reached whenever internal or external variables are found. This DP requires several parameter inputs to make the correct choice at this time. Using the behavioural model of the application, 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 3.1 and the state model shown in 3.15a.


Taking photos
<<Decision-Points >>
<<CRule>>
Activate Flashes : StateMachine
at <=17 /defer
when Bandwidth level > 50% ¶ /d
when Cloudy=true /defer
when status !="unknown" /defer
<<ElseAction>>
Disable State
<<ActionIfTrue>>
Enable Flashes
Enable Video Streaming
<<CRule>>
Search for streaming components
when Component found /defer
<<Decision-Points >>
<<ElseAction>>
Component not found
<<CRule>>
Invoke Components
when Free Memory < 10214 K /defer
when CPU Utilization > 87% /defer
when Bandwidth speed >= 56kbts /defer
when Battery level > 50% /defer
when Connection type="Wifi" /defer
when and status !="unknown" /defer
<<Decision-Points >>
<<ElseAction>>
Invoking component failed
<<ActionIfTrue>>
Invoke Streaming component
Search for sorting component
when Video streaming status = "Disabled" /defer
when Flashes status ="Disabled" /defer
when photos found=True /defer
<<Decision-Points >>
<<ElseAction>>
<<Decision-Points >>
<<ElseAction>>
Terminate sorting
<<CRule>>
Invoke sorting componet
when Bandwidth speed >= 56kbts /defer
when battery level > 50% /defer
when && status !="unknown" /defer
when Connection type="Wifi" /defer
when location = "current location" /defer
when Free Memory < 10214 K /defer
when CPU Utilization > 87% /defer
<<Decision-Points >>
<<ElseAction>>
<<ActionIfTrue>>
<<CRule>>
Sort Photos
Photos Quality attributes /defer
when location = "current location" /defer
Store photos

(a) Policy 1 (b) Policy 2 (c) Policy 3

Fig. 3.15: Decision policies

3.6.3 Modelling the Platform Specific Model

The three diagrams modelled in the previous tasks are transformed into COCA-ADL, as shown in Figure 3.16; these models are used as input for ArchStudio proposed by Dashofy et al. [Dashofy et al., 2007] as architecture modelling tool. The ArchStudio takes the following inputs: 1) The behavioural model includes the COCA-component object diagram, the

Listing 3.1: Decision Policy Example

```
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();
```

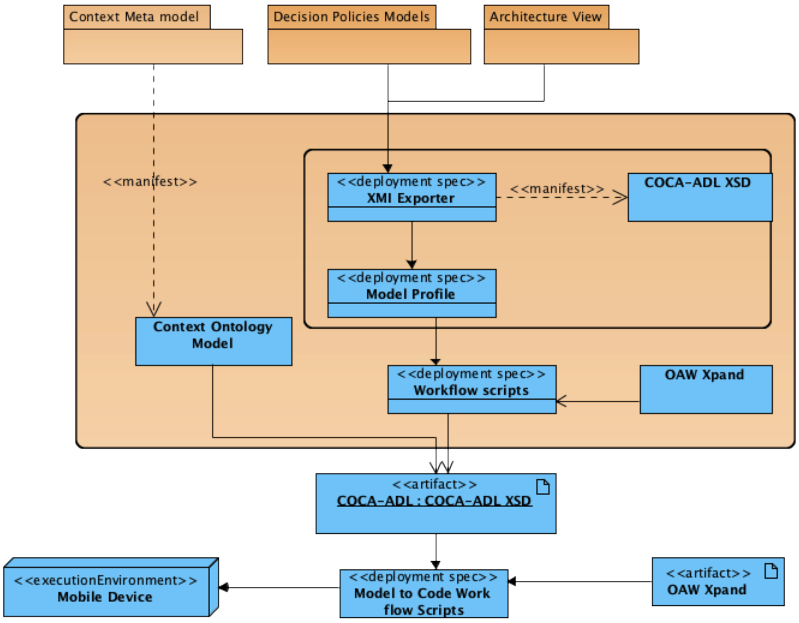


Fig. 3.16: Overall model

behavioural view, and the state diagram of the three policies, Figure 3.15 exemplifying the DPLs extracted from the previous phase. 2) The core structure of the application is shown in Figure 3.13. 3) The context meta-model includes the context model itself and the context and resource models, that describes the available context entities and their representation on the target platform.

The first task in the transformation phase is transferring the models produced in the design and model phases into the architecture description language. The ADL can be transformed into multiple platforms using XMI Exporter supported by a COCA-ADL XSLT stylesheet. A general description of the transformation process is shown in Figure 3.16. The final architecture of the application is shown in Figure 3.17. This task is bidirectional. The designer checks the transformation result by validating the generated Architecture Description language (ADL) among the original models.

3.6.4 Code Generation

The final step is generating the code from the ADL into the target implementation language. Model-to-code transformation is supported by several tools such as Visual Paradigm [Visual Paradigm, 2010], Lattice [Lattice Business Software, 2010], and the Enterprise Architecture [SPARX Enterprise Architecture, 2010]. The ADL-XML code can be transferred to Java and Objective-C using the Visual Paradigm tool [Visual Paradigm, 2010] to generate an Objective-C code. In the next chapter the implementation of the case study is illustrated and the code is discussed.

3.7 Summary

This chapter 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. Specifically, a model-driven architecture is used to demonstrate a generic and standard approach to building context-dependent and self-adaptive applications by adapting a model-driven architecture (COCA-MDA). COCA-MDA enables developers to modularize applications based on their context-dependent behaviours, enables developers to separate context-dependent functionalities from the application's generic functionality, and enables the developers to support the context-driven adaptation.

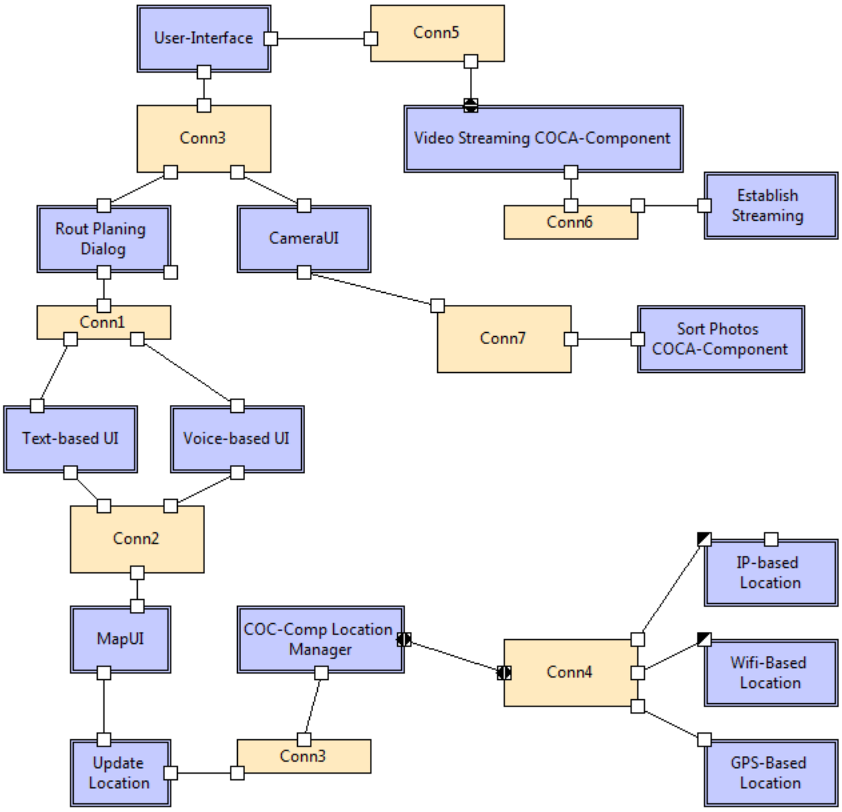


Fig. 3.17: I-TrinityTour architecture

Chapter 4

Implementation

The main objective of this chapter is to describe the implementation of the COCA-middleware using one of a standard object-oriented programming languages. Objective-C has been selected to implement the COCA-component framework and the COCA-middleware. The proposed I-TrinityTour application is implemented in an IPhone device. The application has been evaluated with respect to achieving self-configuring and self-tuning properties.

The implementation of the platform is divided into two major branches. The first is implementation of the COCA-middleware in a mobile platform, as described in Section 4.1. The second branch, is implementing the case study I-TrinityTour, as described in Section 4.2

4.1 The COCA-platform Implementation

The final step of the COCA-MDA is to generate the code and the COCA-ADL XML file. The runtime functions start once these two inputs are in place. The major component in the COCA-middleware is the adaptation manager. On a broader scale, the adaptation manager defers as many decisions as it can from compile time and link time to runtime. Whenever possible, it performs actions dynamically and executes the compiled code. Handling the COCA-component framework, the composition plan, the application singleton, and the way in which several components interact with the runtime system is the focus of the following sections. Figure 4.1 shows a class diagram for the COCA-middleware.

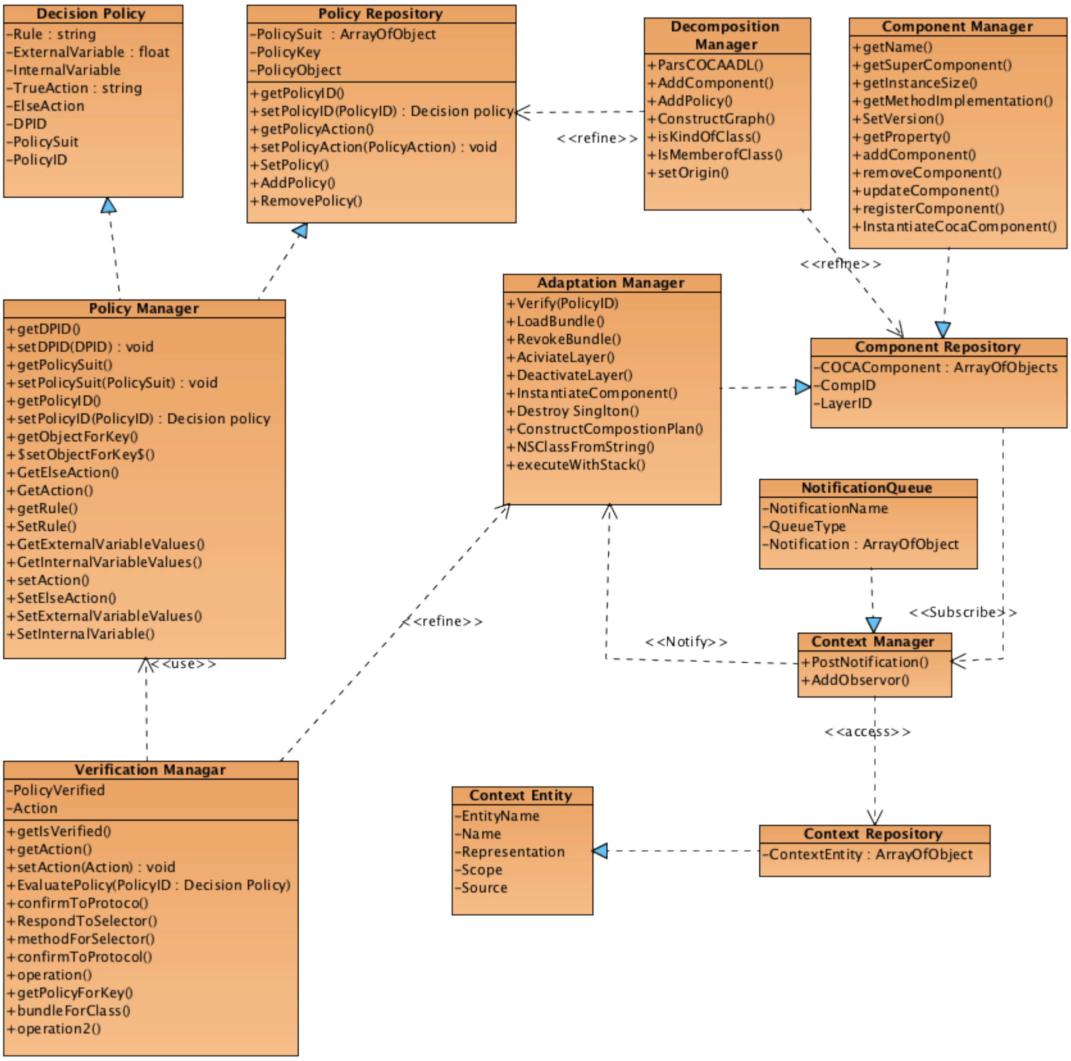


Fig. 4.1: COCA-middleware

The COCA runtime platform is a dynamic shared library with a public interface, consisting of a set of functions and a data structure in the header file, located within the framework in the "COCA.h" file. Many of these functions allow the application to perform the adaptation actions through the adaptation manager.

4.1.1 The COCA Runtime Platform

The COCA runtime platform starts once it has the compiled code for the application base-components and the COCA-component framework plus the COCA-ADL XML file. When the application is launched, the COCA-middleware components are executed first. The adaptation manager then calls the decomposition manager to build the application composition graph and the inheritance tree. The decomposition manager parses the COCA-ADL XML file for the component elements. The decomposition manager adds the components to the graph and the component repository. Each graph node has a component dispatch table. This table has entries which associate method selectors with the component-specific addresses of the methods they identify. In the same way, the decision policies are attached to the associated COCA-component and added to the policy repository. Figure 4.2 shows a decomposition mechanism. Each component and its subdivisions are added to a graph in the buffer.

4.1.2 Adaptation Manager Runtime Functions

Once the decomposition is finished, the adaptation manager asks the context manager for the context state. At the same time, the adaptation manager runs the component instance for the base-component type. Each base-component is a subclass from the super-class. The adaptation manager checks each class by parsing the graph. In each node, the adaptation manager performs the following operations: 1) creating the application singleton, 2) adding the base-components to the singleton, and 3) constructing the primary composition plan.

Afterwards, the context manager notifies the adaptation manager about the context state. Based on the context state, the adaptation manager reads the description of the component from the dispatch table. It confirms whether the component has the right objects, and methods which suit the context state. This is accomplished by asking the object to identify its classes using $isKindOfClass$ and $isMemberOfClass$. This verifies an object's position in the inheritance tree. The COCA-middleware design confirms to the delegation design pattern. This requires each COCA-component to define a protocol or a formal interface, as in JAVA language. During the composition, the adaptation manager identifies whether

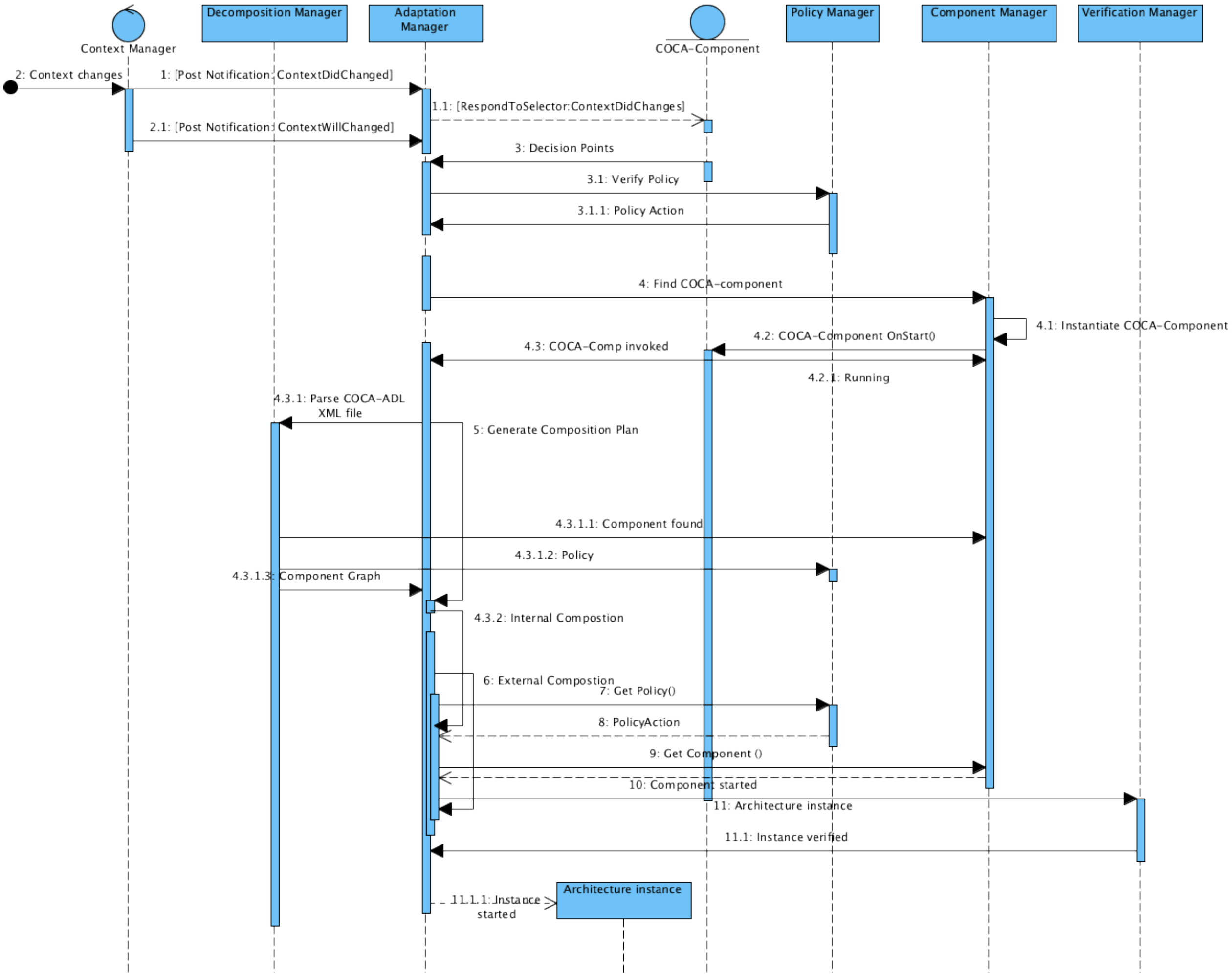


Fig. 4.2: Application Adaptation Runtime Model

a specific component does conform to a protocol by calling $conformsToProtocol$:, which indicates whether an object claims to implement the methods in a specific protocol, then the operation $RespondToSelector$: is performed, which indicates whether an object can accept a particular message. After that, the adaptation performs $methodForSelector$:, which provides the address of a method's implementation. These methods enable the adaptation manager to to introspect the application structure. The class diagram in Figure 4.1 shows the relation between the adaptation manager class and the other COCA-middleware components.

Potentially, the COCA-ADL provides a description of the components, connectors, and configuration. Parsing the XML file to construct the graph has some drawbacks with respect to device performance. A reasonable approach to parsing the XML file with less impact on the quality attributes is the use of the Flyweight design pattern and the NSXMLParser [Buck and Yacktman, 2010].

The Flyweight pattern minimizes the amount of memory and/or processor overheads required to use objects [Buck and Yacktman, 2010]. The Flyweight pattern enables instance sharing, to reduce the number of instances needed, while preserving the advantages of using objects. Classes which implement the Flyweight pattern are called 'flyweights'. Flyweights encapsulate non-object data so that the data can be used in contexts where objects are required. Flyweights reduce storage requirements when a large number of instances are needed. Flyweights act as stand-ins for other objects [Buck and Yacktman, 2010].

In addition to Flyweight, another pattern which can be used during implementation is the Associative Storage pattern. The most important feature of this pattern is the efficient storage of arbitrary data associated with objects; this promotes flexibility by delaying the selection of which data to access until runtime.

The NSMutableDictionary is used to implement the composition plan and the decision policies [Apple IPhone Operating System IOS, 2011]. The use of NSXMLParser implements an event-driven approach with a delegate object implementing methods for handling each of the 'events' the parser encounters during its single pass over the XML data [Apple IPhone Operating System IOS, 2011]. Events most commonly of interest are the beginning and ending of ADL elements and attribute data within elements. The NSXMLParser reads the XML elements, then uses NSMutableDictionary to store them in the dictionary. The *setObject:ForKey:* method is used to create new associations in the dictionary. When keys and values are added and removed from a mutable dictonary, the memory allocated for storing objects grows and shrinks automatically. If *setObject:ForKey:* is called with a key which is already in the dictionary, the object associated with that key is replaced by the new object. Each key is stored at most once [Apple IPhone Operating System IOS, 2011].

After completing the composition plan, the adaptation manager implements the dynamic creation pattern to load and execute the application's components. Once the composition plan is completed, the adaptation manager introspects the application's structure. The component sublayers are activated by redirecting the COCA-component delegate to the desired layer. In some cases, a COCA-component is loaded into the application. This requires the adaptation manager to employ the bundle design pattern [Buck and Yacktman, 2010]. The bundle pattern achieves the following goals: 1) keeping executable code and related resources together, even when there are multiple versions and multiple files involved in the underlying storage; and 2) implementing a flexible plug-in mechanism which enables dynamic loading of executable code and resources.

In addition, the Invocation design pattern is used to provide a means of capturing runtime messages so that they can be stored, rerouted, or treated and manipulated as objects, and allows new messages to be constructed and sent at runtime without requiring a compiler. When an object receives a message, a method is usually invoked to handle the message. However, this is not always the case. For example, if an object does not implement a particular method, then there is no method which can be invoked, and a runtime exception is raised instead. Because of the Invocation design pattern, it is possible for a message to be delayed, rerouted to other receivers, or even ignored. In some cases, the adaptation manager uses the mechanism of forwarding to surrogate objects [Buck and Yacktman, 2010].

The $forwardInvocation$: method is used to give a default response to the message, or to avoid the error in some other way. For example, suppose that the adaptation manager receives a message call for a method $SalarySummation$. First the verification manager verifies whether the receiver object can respond to this message using $respondToSelector$ [Apple IPhone Operating System IOS, 2011]. When the object cannot respond to the message because it does not have a method matching the selector message, the COCA runtime system informs the object by sending it a $forwardInvocation$: message. Every object inherits the method $forwardInvocation$: from the super-class COCA-component.

However, the object version of the method simply invokes $doesNotRecognizeSelector$. In this case, the adaptation manager forwards to other objects. First, the adaptation manager determines where the message should go and sends the message with its original arguments. The message can be sent with the *invokeWithTarget* : method, as shown in the code listing 4.2. If the invocation has failed in the desired sublayer, the method forwards the invocation to the COCA-component, which forwards it into its sublayers until one of the sublayers responds to it. If the sublayers do not respond to it, the adaptation manager introspects the component graph and reconstructs the composition plan. Once an object found in the distributed environment in a remote component.

The adaptation manager performs NSInvocation by obtaining the method signature and the selector [Apple IPhone Operating System IOS, 2011]. The code example illustrated in the listing 4.1 shows a dynamic method invocation. When a message is sent to an object which does not implement it, the actual implementation assumes that the stack frame for the arguments of the method already exists. All further changes to the method's arguments using NSInvocation's methods are performed on that stack frame. After the method invocation returns, you can access the return value and possibly change it, using the getReturnValue: and setReturnValue: methods.

Listing 4.1: Forward invocation to other objects

```
BOOL flag = YES;
    int anInt = 1234;
    float aFloat = 12345.0;
    double aDouble = 98765.0;
    id invocation = [[NSInvocation new] autorelease];
    [invocation setSelector:
                @selector(setFlag:intValue:floatValue:doubleValue:)];
    [invocation setTarget:object];
    [invocation setArgument:&flag atIndex:2];
    [invocation setArgument:&anInt atIndex:3];
    [invocation setArgument:&aFloat atIndex:4];
     [invocation invoke];
```

Listing 4.2: Forward invocation to other objects

```

−(void) forwardInvocation:(NSInvocation *) anInvocation
{
if ([COCAcomponentLayer respondToSelector:[anInvocation selector]])
[anInvocation invokeWithTarget:COCAcomponent];
else
[super forwardInvocation:anInvocation];


}
```

4.1.3 Context Manager

Once the composition plan is finished and the application singleton is constructed, the adaptation manager executes the first application instance. At this stage, the context manager monitors and detects context changes and the adaptation action is started as discussed in Chapter 3. After running the COCA-component in the application instance, each COCA-component registers itself with the context manager as an observer of two or more notifications using the notification centre. If any context condition is changed, the adaptation manager is notified to find the associated COCA-component by identifying the notification observers. Each component registers itself as an observer for one or more notifications using the code in the listing 4.3. After the registration is accomplished, the context manager notfies the

Listing 4.3: COCA-component registers itself as observer for a specific context condition

```
[[NSContextManagerCenter defaultCenter]
addObserver:Self
selector:@selector(ContextConditionDidChange:)
name:NSCOCAComponenetDidChangeSelectionNotification
object:NSCocaComponent;
```

adaptation manager and the COCA-component about the context change using the code in the listing 4.4

Listing 4.4: Context manager notification sent to COCA-component

```
// Register for battery level and state change notifications.
[[NSNotificationCenter defaultCenter] postNotification:NotificationName
Selector:ContextDidChanged
Object:COCA−component
Controller:AdaptationManager];
```

The methods *addObserver* and *removeObserver* are used at runtime by the adaptation manager to register new objects. In addition, the context manager has the ability to post distributed context information. This is accomplished using the *NSDistributedContextManager* default notification queue class. The *NSDistributedContextManager* is a subclass of *NSContextManager*, so the notifications are posted to the default queue in the same way as they are posted to a regular context manager notification queue.

4.1.4 COCA-components Framework

A general overview of the COCA-component is shown in Figure 3.2. The benefit of using the subclassing method is that the class properties are inherited by the sublayer. This simplifies modifications of the class sublayer implementation and provides an easy mechanism for the adaptation manager to use the Invocation design pattern. In some cases, when the adaptation manager sends a message call to an object, the component manager implements the method as shown in Figure 4.1.

When a message is sent to an object, the messaging function follows the object's *isa* pointer to the class structure, where it looks up the method selector in the dispatch table. If it cannot find the selector there, *objc_msgSend* follows the pointer to the super-class and tries to find the selector in its dispatch table. Successive failures cause *objc_msgSend* to climb the class hierarchy until it reaches the COCA-component class. Once it locates the selector, the function calls the method entered in the table and passes it the receiving object's data structure. In this way, methods are dynamically bound to messages.

Sending a message to an object which does not handle that message is an error. However, before announcing the error, the runtime system gives the receiving object a second chance to handle the message [Buck and Yacktman, 2010]. In this case, the adaptation manager sends a $forwardInvocation$ message with an NSInvocation object as its sole argument [Apple IPhone Operating System IOS, 2011]. The NSInvocation object encapsulates the original message and the arguments which were passed with it. This technique is used for preserving the states of messages, arguments, and return values. Invocation can be used to completely decouple the sender of a message from the receiver. The sender and receiver can be in different processes or separated by time. This is used whenever the adaptation manager executes a proactive adaptation. A delayed Invocation message is sent to the object in the component manager. When the time comes for the method to be executed, the adaptation manager invokes the objects in the application instance.

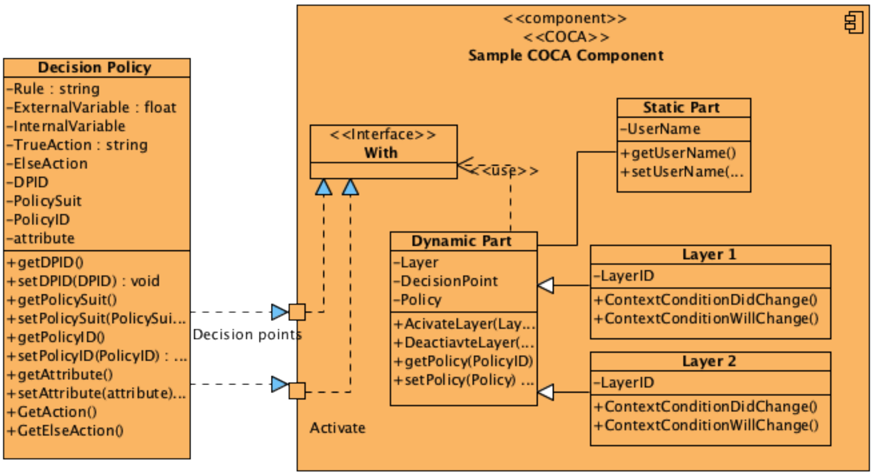


Fig. 4.3: COCA-component conceptual diagram

The intercession operation is achieved by the component manager by adding a component, or a component sublayer, to the application structure. To add a component, the adaptation manager asks the component manager to instantiate a specific component. The component manager performs several inspections of the component, e.g. *getName:, getSuperComponent:, getInstanceSize:, getMethodImplementation:, getSubLayers:, setVersion:, getProperty:, addComponent:, removeComponent:, updateComponent:, and registerComponent:*. The component manager performs the method *init* with the component instance and, in the same way, it performs *dealloce* : to destroy the instance from the memory. Once the component instance is running, the component manager can work with these instances through the methods in 4.5:

Listing 4.5: Component framework methods

```
-(COCAcomp *) Comp_copy:(COCAcomp *);
-(id)_setLayerActive:(COCAcomp *) CompLayer;
-(id) Comp_getLayerMethod:(COCAcomp *) CompLayer;
-(NSString *) Comp_getName:(COCAcomp *) CompLayer;
```

In addition, the component manager manipulates the COCA-component instance lifecycle. Each component passes through several stages, as shown in Figure 4.4. A component is in the running state in three events: OnStart, OnResumed, or OnRestart. When a running component is paused, its status changes to pause, then it is stopped, and then killed. When a component is killed, it is immediately destroyed. When a stopped component is killed and then destroyed, it is deallocated from the memory.

4.1.5 Policy Manager

Figure 4.1 demonstrates a class diagram for the COCA-middleware, showing its components and data stores. The policy manager uses the decision policy objects to store policies in the policy repository. The policy dictionary stores each policy in loosely coupled objects, which are accessed through the method *getObjectForKey*. Once the object is retrieved, the policy manager obtains the policy actions, attributes, rules, external variables, and internal variables. Then it passes them back into the verification manager. The verification manager

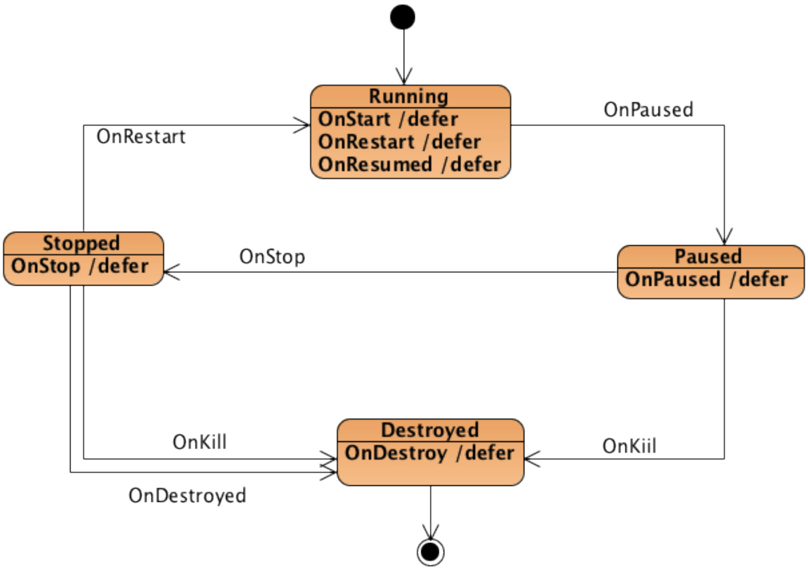


Fig. 4.4: Components life-cycle

evaluates the current value for the variables among the predefined values in the policy syntax.

Whenever the application execution reaches decision points and/or the context manager has notified the adaptation manager of a context change, the decision points must be executed to advise the application of pre- or post-actions among specific notifications. The manager implements the necessary methods to manipulate the decision policy syntax. The policies are stored in an array of objects. Each object is accessed through the method *getPolicyForKey*. The key refers to the policy ID which is attached to every COCA-component. In the same way, the policy manager is used to upgrade the policy by calling the methods *setObjectForKey*, *SetPolicySuit*, *SetRule()*, *setAction*, and *setElseAction*.

The decision policy is stored in the policy repository, which conforms to the Associative Storage design pattern. A binary representation for each policy is stored in the NSMutableDictionary data structure, where the key value is used to access the desired policy. The method *addPolicy* is used to add a new policy to the repository. In the same way, policies can be removed using the method *RemovePolicy*. Once the policy is updated by the policy manager, the method *setPolicy* is used to update the policy syntax in the repository. The policy syntax is retrieved through the method *getPolicyForKey().* This implementation of the Associative Storage design pattern reduces the computation overhead for retrieving the policies and evaluating them.

4.1.6 Verification Manager

A context-aware application is self-configurable if it is able to adapt autonomously to changing environmental conditions or internal status by altering its structures, behaviours, and data to meet its functionality and quality requirements. From a middleware perspective, such a feature relies on the following key characteristics: A) The middleware's ability to monitor and define its internal status and external conditions e.g. application modes, CPU and memory use, and attachment of external devices; B) its built-in knowledge of configuration variability and related policies/rules for deciding and planning changes; and C) its ability to perform dynamic configuration changes without violating the constraints relating to overall system functionality, performance, and dependability.

The Flyweight pattern is used to guarantee that the verification process does not affect the quality attributes. This is accomplished by adapting the feature of instantiation in the Flyweight. Moreover, if there are many external and internal variables in the decision policy, all these variables will be instantiated once and share this instance to multiple values. In addition, the Flyweight acts as a temporary place holder for other more heavyweight objects.

Figure 4.5 shows a sequence diagram of the self-assurance verification. The verification manager evaluates the policies by calling the *verifyPolicy:(NSInteger) PolicyID* method. This method asks the policy manager to retrieve the stored policy by its key. The policy manager searchs the array of objects for the specific policy ID. The evaluation result, which contains the proposed action to be performed, is passed back to the adaptation manager. Afterwards, the adaptation manager locates the desired component and/or sublayers which need to be executed. Then it asks the verification manager to verify them using the methods *methodForSelector, confirmToProtocl, RespondToSelector.* Once the verification has been accomplished, the verification manager sets the boolean variable *PolicyVerified* to be true.

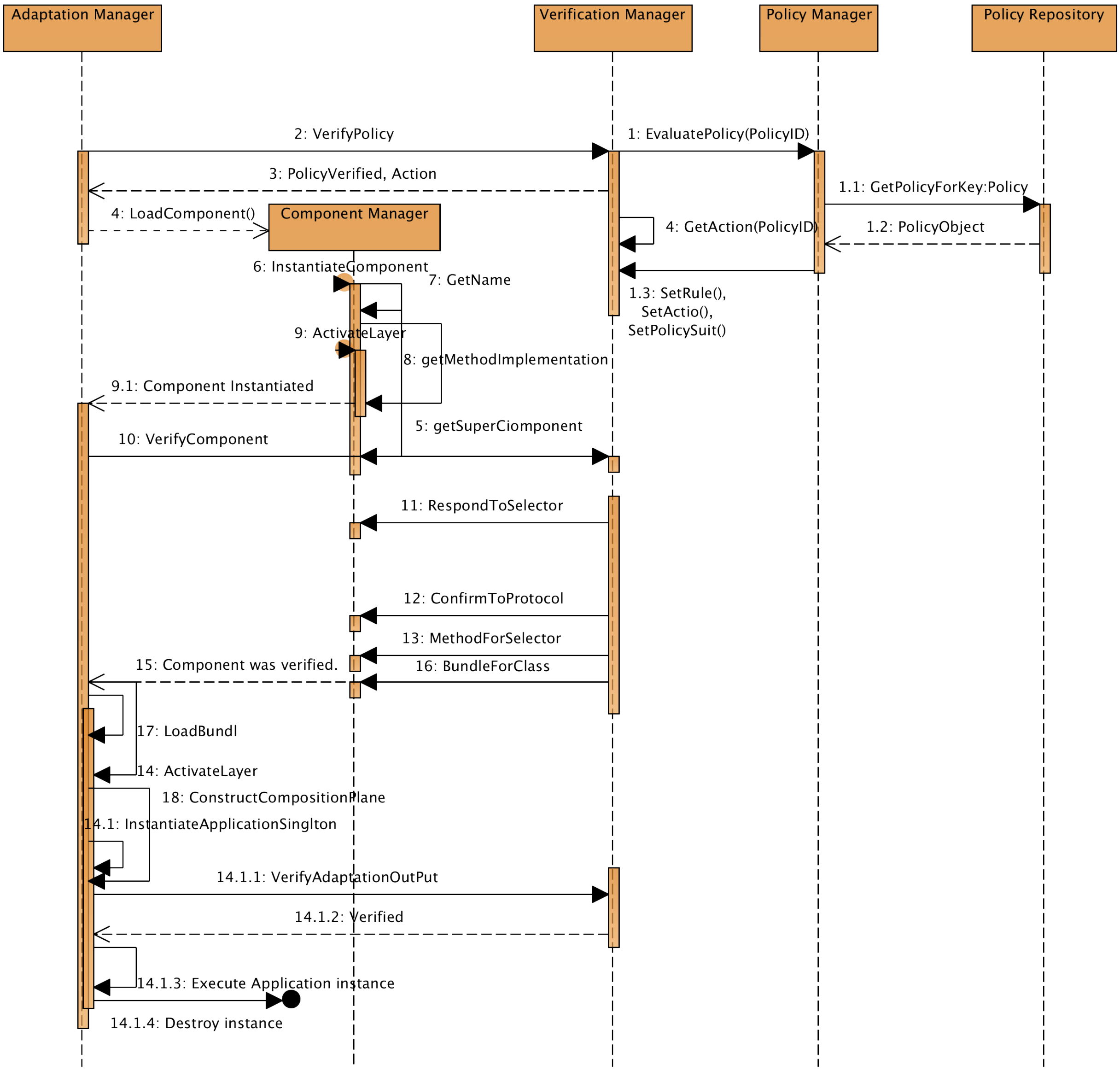


Fig. 4.5: Adaptation self-assurance verification

The motivation behind the use of selectors is to postpone specifying the message which will be sent to an object until runtime. This reduces coupling between objects by limiting the information which message senders need about the message sent. In the same way, whenever the adaptation manager needs to verify the implementation of a COCA-component or a subdivision, the selector will be used to determine whether the object has the proposed method. The verification manager can load bundles/components by executing the method *bundle = [NSBundle bundleWithPath:theBundlePath];*. However, after loading the bundle which contains a COCA-component, the verification manager testifies to its ability to respond to the desired method call using the code in the listing 4.6.

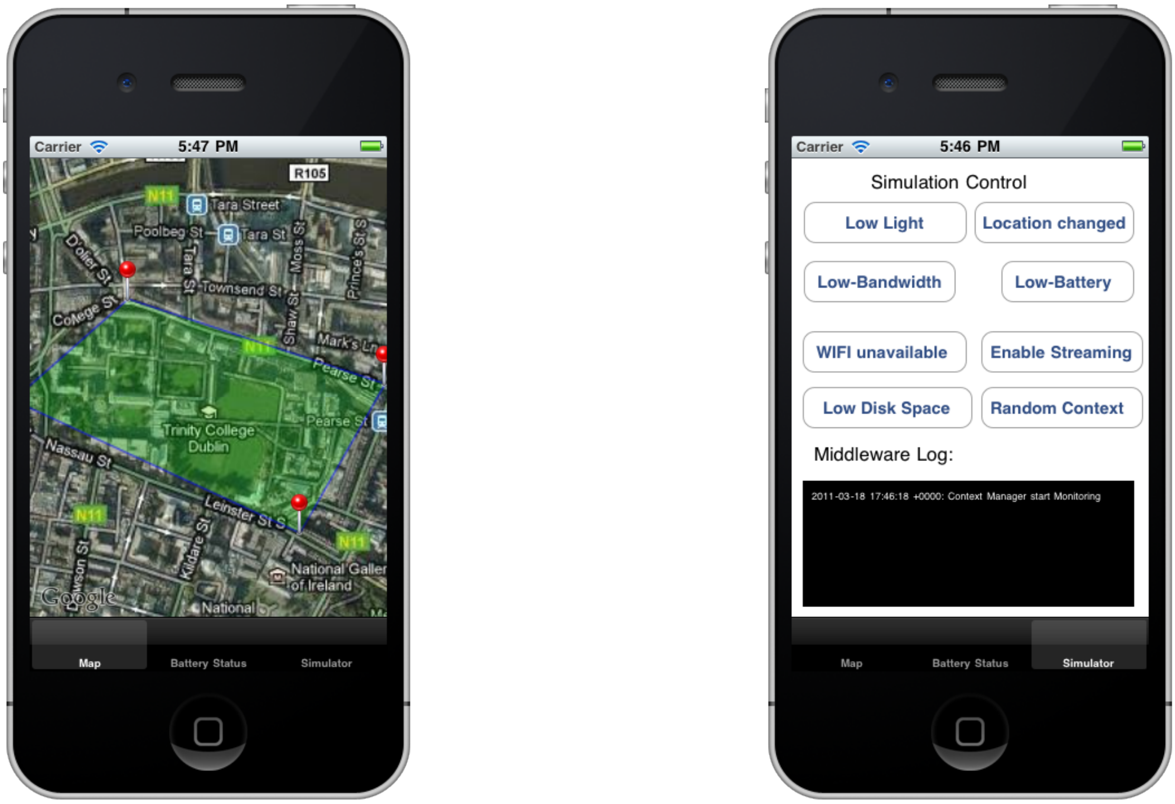
Listing 4.6: Retreving bundle information

```
NSBundle *bundle = [NSBundle bundleWithPath:componentRepository];
NSDictionary *infoDictionary = [bundle infoDictionary];
[NSBundle bundleForClass:[NSString COCA-componentClass]];
[COCA-componentClass respondToSelector:ContextConditionDidChanged];
[COCA-componentClass respondToSelector:ContextConditionWillChanged];
```

4.2 I-TrinityTour Case Study Implementation

The case study has been selected to demonstrate the capability of the COCA-platform to perform external and internal compositions to maintain an architecture quality attribute. This section focuses on demonstrating the variations in application behaviour, based on context changes at runtime. The final Objective-C code for the architecture was generated by the Visual Paradigm tool [Visual Paradigm, 2010]. However, the application is implemented on an IPhone mobile device, as shown in Figure 4.6, using the IOS SDK 4.3 [Apple IPhone Operating System IOS, 2011].

To demonstrate variation in the application behaviour, a simulator was included with the I-TrinityTour implementation. The simulator, shown in Figure 4.6b, is used to allow the user to simulate specific context changes, which are used to test the application's ability to adapt the desired behaviour. As shown in the figure, the user may select any of the buttons to generate the same context condition as that which triggers the adaptation. For example, when the user presses low battery, a notification is posted into the notification centre in the



```
2011-03-18 17:46:18 +0000: Context Manager start Monitoring
```

(a) Map UI for Trinity College Region

(b) Simulator UI

Fig. 4.6: I-Trinity application running on IPhone device

context manager. The context manager notifies the adaptation manager. In such a case, the adaptation manager activates the desired adaptation action. The middleware console shows the output log from the simulator.

MapUI in Figure 4.6a displays the region which is monitored by the application. This demonstrates the context-monitoring process performed by the context manager. The implementation demonstrates the application's ability to adapt itself with respect to video streaming, sorting components, camera flashes, and location services.

Achieving the self-adaptive property 'Self-tuning' can be demonstrated by implementing the COCA-component 'location Update'. The location manager COCA-component was

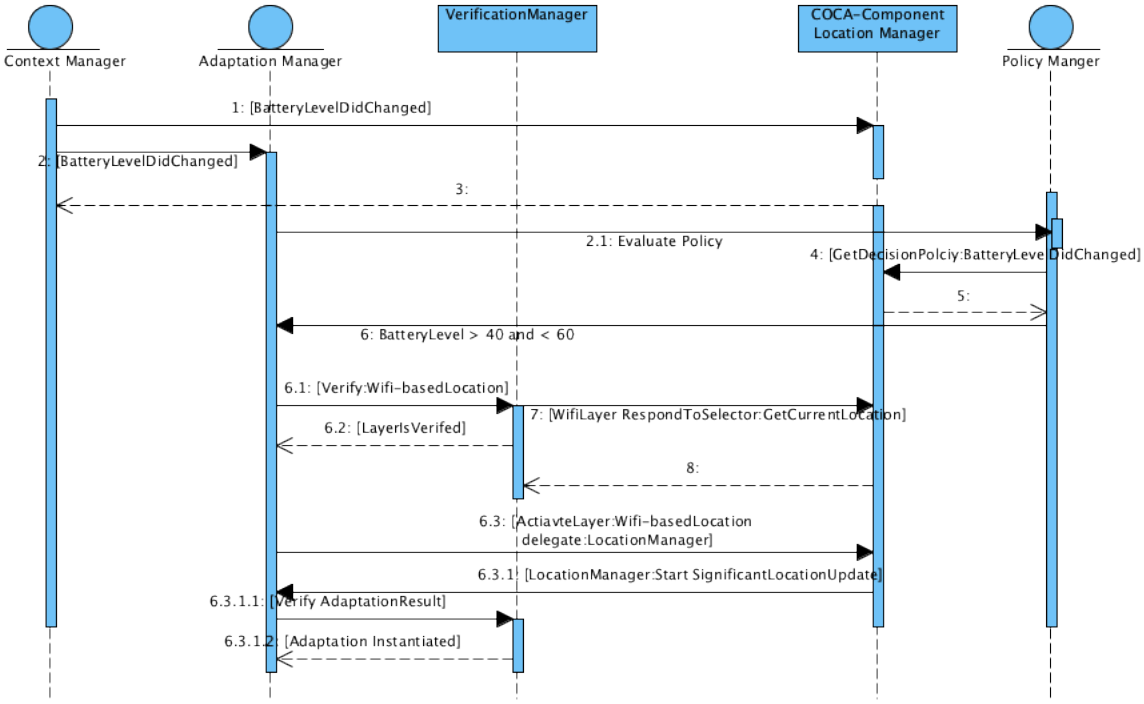

VerificationManager
COCA-Component Location Manager
Context Manager
Adaptation Manager
Policy Manger
1: [BatteryLevelDidChanged]
2: [BatteryLevelDidChanged]
3:
2.1: Evaluate Policy
4: [GetDecisionPolciy:BatteryLevelDidChanged]
5:
6: BatteryLevel > 40 and < 60
6.1: [Verify:Wifi-basedLocation]
7: [WifiLayer RespondToSelector:GetCurrentLocation]
6.2: [LayerIsVerifed]
8:
6.3: [ActiavteLayer:Wifi-basedLocation delegate:LocationManager]
6.3.1: [LocationManager:Start SignificantLocationUpdate]
6.3.1.1: [Verify AdaptationResult]
6.3.1.2: [Adaptation Instantiated]

Fig. 4.7: COCA-component Location-update sublayer activation

proposed as an extra functionality which demonstrates the application's ability to adapt a location service, based on the battery level. Such an adaptation action can extend the durability of the battery and demonstrates the application's ability to use battery resources efficiently. Figure 4.7 shows a sequence diagram which describes the middleware operation for achieving this kind of adaptation. As shown in the figure, the context manager posts the notification BatteryLevelDidChanged. This notifies the adaptation manager and the COCA-component location manager about the context changes. The adaptation manager decides, based on the battery level, which location service to use. If the battery level is less than 30% the adaptation manager activates the IP-based layer. If the battery level is between 60% and 40%, the Wifi-based location is updated. If the battery level is greater than 60%, the GPS-based adaptation is activated.

Implementing the self-adaptive property 'Self-configuring' can be demonstrated by imple-

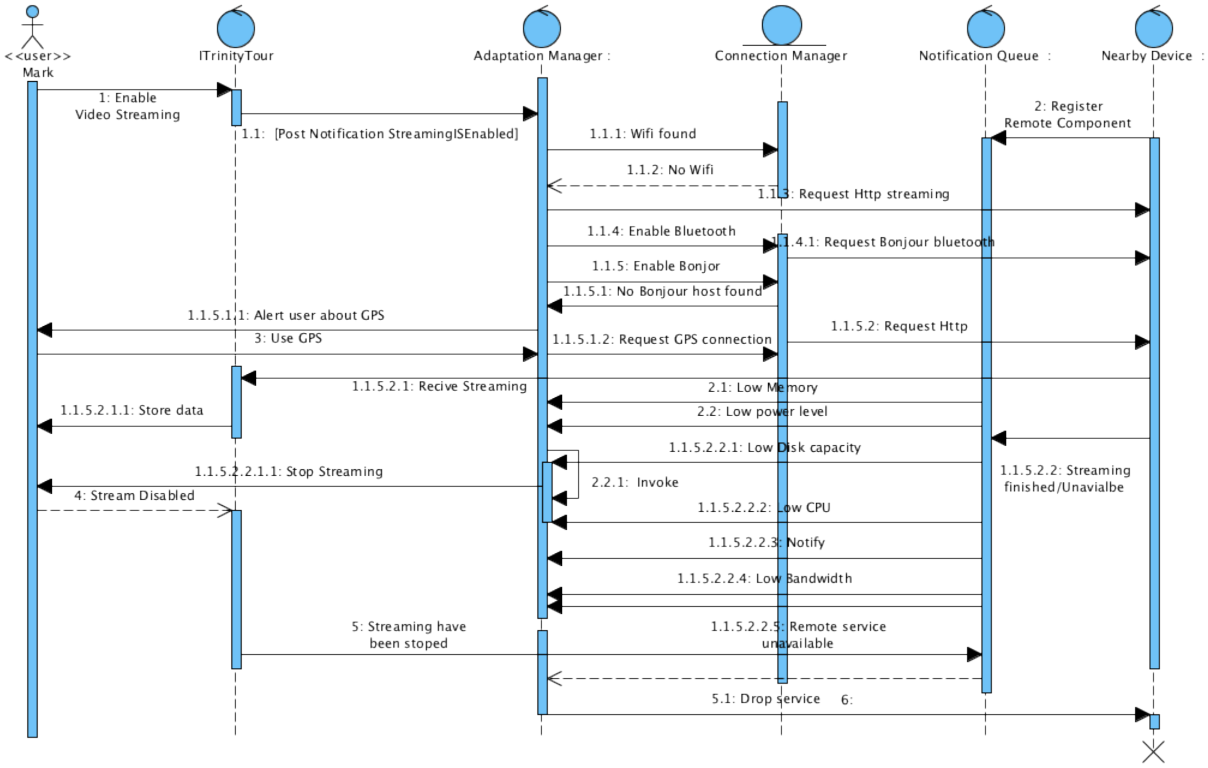


Fig. 4.8: Sequence Diagram for Streaming Component Adaptations

menting the COCA-component 'Video Streaming'. The implementation demonstrates how the application can invoke a streaming component dynamically at runtime. The sequence diagram in Figure 4.8 demonstrates the possible reasoning for adapting the streaming component. Figure 4.9 shows a streaming component invoked in the execution. The user enables the video streaming feature. The middleware searches for the streaming host in a nearby device. Before starting the streaming services, the middleware evaluates the current context state. If a Wifi connection is available, http streaming is started. If a Wifi connection is not available, the middleware enables the service discovery protocol e.g. Bonjour and the Bluetooth connection.

Once the Bonjour service is activated, and no devices found to provide the streaming service. The application alerts the user about how expensive it is to perform the streaming through the GPRS connection. At this stage, the user can decide if he wants to proceed with streaming or not. Such an adaptation for a network connection supports the desired functionality and provides the user with low-cost streaming. The streaming component is invoked in the application using the external composition mechanism.

A bundle is a collection of executable code and related resources such as images, sounds, strings, and localization information. In general, bundles are able to store multiple versions of each resource so that a developer or user can use one set of executable code with different resource versions, based on user preferences and context conditions. The most obvious benefit of using the bundle pattern is a mechanism for organizing and dynamically loading executable code and resources.

To demonstrate the power of the NSBundle in Objective-C, assume that the application will need to invoke a component ‘VideoStreaming’. The adaptation manager gets the name of the component and its URL through the service discovery. Afterwards, the adaptation manager executes the code in 4.7 to execute the component.

Listing 4.7: Loading COCA-component from remote url

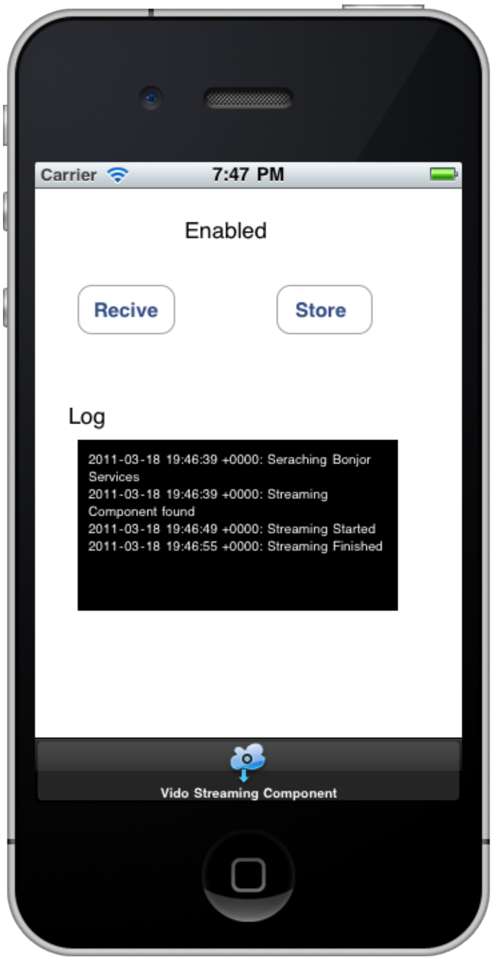
```
NSBundle *bundle= nil;
bundle = [NSBundle bundleWithURL:RemoteUrl]; //alternavilly to load the bundle using the identifier
NSBundle* myBundle = //in the Application ADL the following code might be used
 [NSBundle bundleWithIdentifier:@‘dsg.tcd.VideoStreaming’];
```

The adaptation manager must be sure about which bundle it might run to execute the desired functions. If the application is looking for a bundle which has a method for invoking a component for sorting photosgraphs, the code in 4.8 can be used to accomplish this task.

Listing 4.8: Verifying COCA-component for a specific class

```
NSBundle* myBundle = [NSBundle bundleForClass:@‘VideoStreaming’];
```

The second COCA-component which is invoked is the photograph-sorting component.



```
2011-03-18 19:46:39 +0000: Seraching Bonjor
Services
2011-03-18 19:46:39 +0000: Streaming
Component found
2011-03-18 19:46:49 +0000: Streaming Started
2011-03-18 19:46:55 +0000: Streaming Finished
```

Fig. 4.9: Streaming component invoked in the execution

This component is adapted according to the sequence diagram shown in Figure 4.10. The same sequence diagram demonstrates how the application simultaneously adapts to the light level to activate camera flashes. Adaptation to the light level is achieved through internal composition by activating the associated layer. On the other hand, adapting the photograph-sorting component is achieved through external compositions which add, remove, or update a component instance from the architecture.

While the user is taking photographs, the context manager posts the notification *BatteryLevelDidChangedNotification* to the adaptation manager; the adaptation manager re-

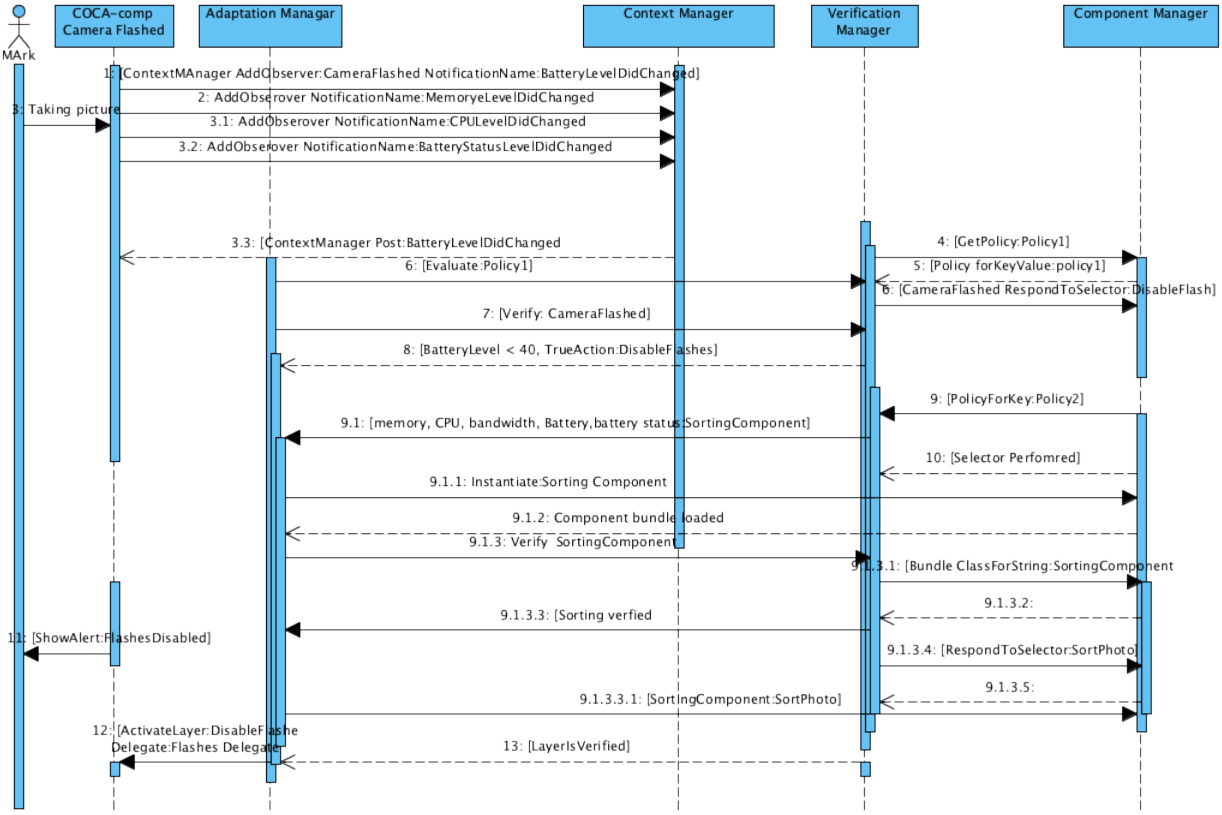
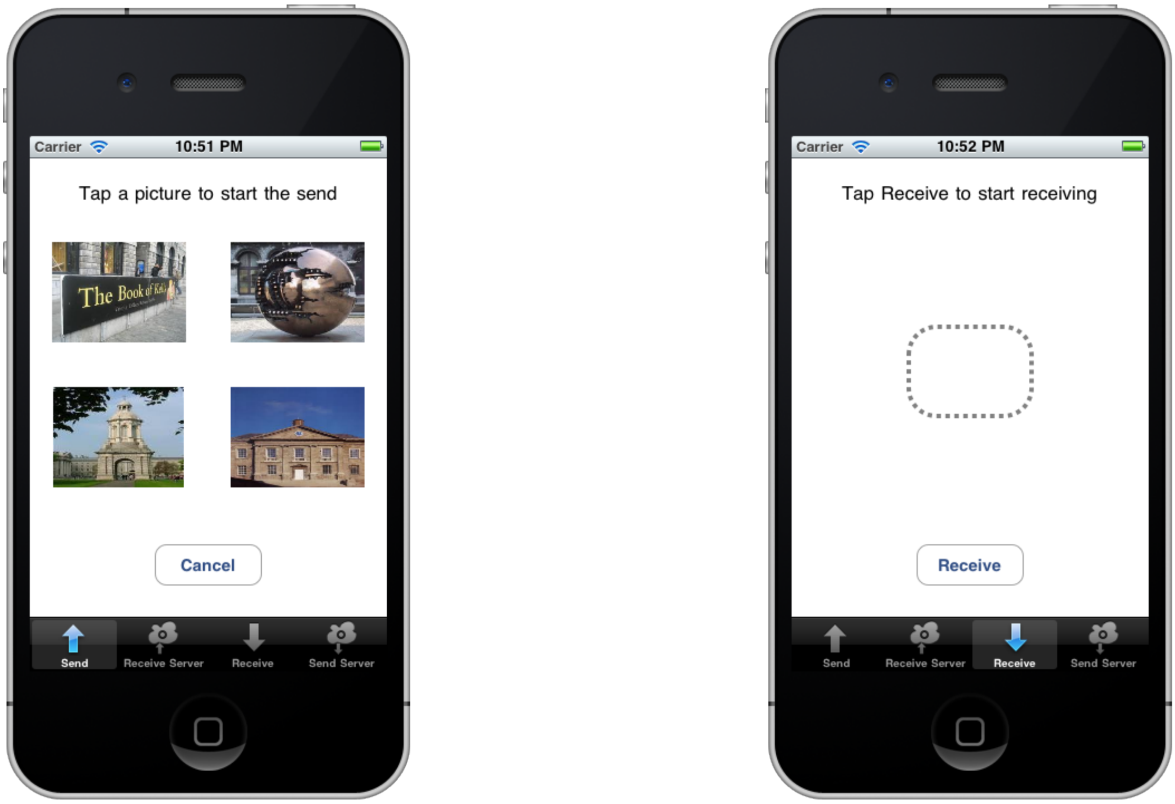

MArk
COCA-comp Camera Flashed
Adaptation Managar
Context Manager
Verification Manager
Component Manager
1: [ContextMAnager AddObserver:CameraFlashed NotificationName:BatteryLevelDidChanged]
2: AddObserover NotificationName:MemoryeLevelDidChanged
3: Taking picture
3.1: AddObserover NotificationName:CPULevelDidChanged
3.2: AddObserover NotificationName:BatteryStatusLevelDidChanged
3.3: [ContextManager Post:BatteryLevelDidChanged
4: [GetPolicy:Policy1]
6: [Evaluate:Policy1]
5: [Policy forKeyValue:policy1]
6: [CameraFlashed RespondToSelector:DisableFlash]
7: [Verify: CameraFlashed]
8: [BatteryLevel < 40, TrueAction:DisableFlashes]
9: [PolicyForKey:Policy2]
9.1: [memory, CPU, bandwidth, Battery,battery status:SortingComponent]
10: [Selector Perfomred]
9.1.1: Instantiate:Sorting Component
9.1.2: Component bundle loaded
9.1.3: Verify SortingComponent
9.1.3.1: [Bundle ClassForString:SortingComponent
9.1.3.2:
9.1.3.3: [Sorting verfied
11: [ShowAlert:FlashesDisabled]
9.1.3.4: [RespondToSelector:SortPhoto]
9.1.3.5:
9.1.3.3.1: [SortingComponent:SortPhoto]
12: [ActivateLayer:DisableFlashe Delegate:Flashes Delegate
13: [LayerIsVerified]

Fig. 4.10: Adapting Sorting Component and Camera flashes using external and internal adaptations

tains the notification to the COCA-component Camera Flashed. The adaptation manager calls the policy manager to evaluate policy 1. This policy is demonstrated in Figure 3.15a. However, the policy manager informs the adaptation manager to deactivate the flashes, then the COCA-component 'camera flashed' alerts the user about disabling the flashes. The sequence diagram for disabling the flashes is shown in Figure 4.10. The adaptation manager asks the verification manager to verify the 'camera flashed' COCA-component, and whether it responds to the selector 'DisableFlash'. The verification manager returns a successful verification of the COCA-component 'camera flashed'. The adaptation manager then delegates the disable flash message to the flash delegate, which disables the flash by calling the disable flash layer. The policy manager requests the policies from the COCA-component; there appears to be a dependence between policy 1 and policy 2. In this case, the policy manager evaluates policy 2 and notifies the adaptation manager about the required actions of policy 2. The true action of policy 2 specifies adapting a photograph-sorting component when several context conditions exist. Policy 2 is demonstrated in 3.15b, Chapter 3.


Tap a picture to start the send

Tap Receive to start receiving

(a) Sending User Interface

(b) Receiver User Interface

Fig. 4.11: Two devices communicating using the Bonjour service

In the same way, the policy manager informs the adaptation manager to invoke the COCA-component, sorting component. The adaptation manager asks the component manager to instantiate and load the component bundle. Then the verification manager verifies whether the component bundle is responding to the class ‘PhotoSorting’ using the method *bundleForClass*.

Then the class testifies as to whether it responds to the selector *SortingCOCA-Component RespondToSelector:SortingPhotos.* Once the verification has been passed successfully, the adaptation manager passes the delegate message to the sorting component bundle. Then the adaptation manager launches the bundle. The photograph-sorting component is used to sort the component and send or receive the photosgraphs. Figure 4.11 shows two devices communicating using the Bonjour service. The first device, A, is used to select the picture to send. The second device enables the user to receive photographs from the network stream.

4.3 Summary

This chapter showed the COCA implementation on IPhone devices. The COCA-middleware and the proposed case study, the I-TrinityTour application, were implemented and evaluated. The application successfully achieved self-configuring and self-tuning properties without. The COCA-middleware preserved the device resources during the adaptation. The use of a notification pattern for context binding reduced the computation overhead in context monitoring, and reduced the traffic from notifying a high volume of context changes. The I-TrinityTour application has proved its ability to tune itself and conserve energy. This implementation of the Associative Storage design pattern reduced the computation overhead of retrieving policies and evaluating them. The verification manager played a major role in assuring and verifying that the adaptation output does not intertwine the architecture attributes. The invocation and delegation patterns support introspection and intercession at the level of the architecture.

Chapter 5

COCA-Middleware Architecture Evaluation

Software architecture evaluation has become a familiar practice in the software engineering community for developing quality software. Architectural evaluation reduces software development effort and cost and enhances the quality of software by verifying the addressability of quality requirements and identifying potential risks. Several methods and techniques have been used to evaluate software architecture with respect to desired quality attributes such as maintainability, usability, and performance [Kazman et al., 2002].

This chapter presents an evaluation of the COCA-middleware design based on its capability to maintain several quality attributes and achieve the self-* properties of the self-adaptive software systems. Surveys of the state of the art are used to select an appropriate method for evaluating the COCA-middleware. The best-suited method described in the literature seems to be the Architecture Trade-off Analysis Method (ATAM) presented by Bass et al. [Bass et al., 2003]. ATAM can be executed without additional training for the evaluation team and provides a principled approach for evaluating the fitness of software architecture with respect to multiple competing quality attributes.

This chapter is organized as follows. Section 5.1 addresses the evaluation objectives. The self-adaptive application quality attributes are characterized in section 5.2. The evaluation of the COCA architecture with ATAM is discussed in Section 5.3. Section 5.4 describes the lessons learned from the ATAM evaluation and the manner in which the evaluation results are reflected in the architecture implementation. Finally, the Context-Oriented Software is evaluated in terms of energy utilisation and adaptation time compared to other approaches proposed in the literature, the evaluation results are discussed in Section 5.5. The COCA-middleware performance and adaptability is evaluated in Section 5.7.

5.1 COCA-middleware Evaluation Objectives

The challenges of implementing adaptable COCA-middleware present a set of dimensions that captures the system's reaction to context changes. These dimensions are related to the adaptation process itself. The objectives of this evaluation experiment are to verify the middleware's ability to fulfil the requirements associated with the self-adaptation action, the level of autonomy of self-adaptation, the manner in which self-adaptation is controlled, the impact of self-adaptation in terms of space and time, the extent to which self-adaptation is responsive, and the manner in which self-adaptation reacts to change. In addition, this evaluation considers the middleware's ability to perform dynamic decision making among decision policies and the COCA-middleware's ability to assure and verify the adaptation output among the quality attributes in terms of software dependability. The evaluation objectives can be summarized as follows:

- **O1:** The COCA-middleware's ability to perform context monitoring and detection, while, at the same time, the middleware is utilizing the device's resources in an effective manner.
  - **A1.1:** The COCA-middleware's ability to deliver context change notifications to the interested architecture elements.
  - **A1.2:** The COCA-middleware's ability to evaluate its own functionality. Context monitoring and detecting are not overwhelming the device performance.

- **O2:** The COCA-middleware's ability to verify or select the adaptation results dynamically (by dynamic decision making). The fitness of the COCA-middleware to consider the system contribution to the context environment and the interoperability of its components. Evaluating the COCA-middleware's ability to handle a trade off analysis among several quality attributes and the self-* properties. The attributes to evaluate this objective can be summarized as follows:
  - **A2.1:** The COCA-middleware can construct the composition plan without overwhelming the device resources.
  - **A2.2:** The COCA-middleware's ability to perform introspection and intercession of the architecture through the operations of adaptation action, adaptation assurance, and interoperability of its components.
  - **A2.3:** The ability to resolve conflict among several adaptation actions. This refers to the ability to perform dynamic decision policy mismatches and sub-layer manipulations.
- **O3:** The ability to achieve coarse-grained and fine-grained adaptation.
  - **A3.1:** The ability to add, remove, or update components' instances dynamically.
  - **A3.2:** The ability to activate or deactivate components' sub-layers dynamically.
- **O4:** Evaluating the middleware's ability to anticipate unforeseen change dynamically.
  - **A4.1:** Dynamic decision making.
  - **A4.2:** Adaptation assurance and verification.
- **O5:** Evaluating the architecture's performance and maintainability.
  - **A5.1:** Achieving the autonomic property "self-tuning", by utilising the allocated resources and the quality attributes of the architecture.
  - **A5.2:** Achieving the autonomic property "self-configuring", by optimising the adaptation process with less re-configuration and adaptation time.

5.2 Self-adaptive Application Quality Attribute Characterization

The quality attributes of a self-adaptive system are determined by its architecture. On the other hand, the architectural decisions have profound impact on the achievement or non-achievement of quality attributes; therefore, they are the focus of architecture evaluation. However, a prerequisite to the evaluation is to have a clear statement of the quality attribute requirements, which are motivated by key goals, and a specification of the architecture, including a clear articulation of the architectural design decisions. These quality attribute requirements and the architecture documentation are often incomplete, vague, or ambiguous. Therefore, by necessity, two of the major goals of architecture evaluation are to elicit both a precise statement of the quality attributes and a precise statement of the architectural design decisions. Moreover, a major goal of architecture evaluation is to evaluate the architecture design decisions so as to determine whether they address the quality attributes [Kazman et al., 2002].

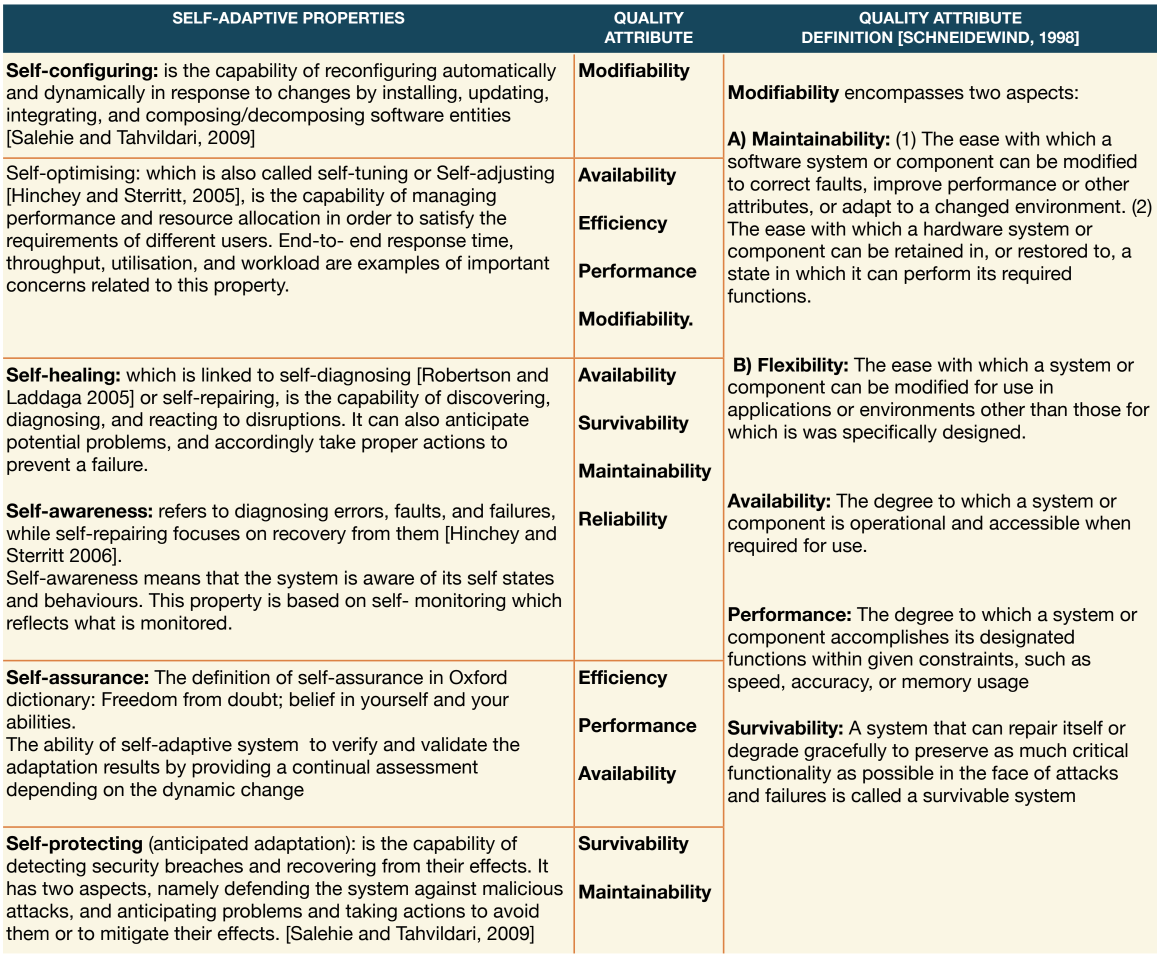
However, clarifying the relationship between architecture and quality attributes greatly extends the design and analysis process. Kazman et al. [Kazman et al., 2002] have proposed a characterization framework for the quality attribute based on three categories:

- External stimuli, i.e., the events that cause the architecture to respond to changes: These events need to be expressed in terms that are measurable or observable.

- Response: These measurable/observable quantities are described in the responses section of the attribute characterization.

- Architectural decision: These are the aspects of an architecture that have a direct impact on achieving attribute responses.

Table 5.1: Relating quality attributes with self-* properties



SELF-ADAPTIVE PROPERTIES	QUALITY ATTRIBUTE	QUALITY ATTRIBUTE DEFINITION [SCHNEIDEWIND, 1998]
Self-configuring: is the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing/decomposing software entities [Salehie and Tahvildari, 2009]	**Modifiability**	**Modifiability** encompasses two aspects: **A) Maintainability:** (1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. (2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions. **B) Flexibility:** The ease with which a system or component can be modified for use in applications or environments other than those for which is was specifically designed. **Availability:** The degree to which a system or component is operational and accessible when required for use. **Performance:** The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage **Survivability:** A system that can repair itself or degrade gracefully to preserve as much critical functionality as possible in the face of attacks and failures is called a survivable system
Self-optimising: which is also called self-tuning or Self-adjusting [Hinchey and Sterritt, 2005], is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to- end response time, throughput, utilisation, and workload are examples of important concerns related to this property.	**Availability** **Efficiency** **Performance** **Modifiability.**	
Self-healing: which is linked to self-diagnosing [Robertson and Laddaga 2005] or self-repairing, is the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent a failure. **Self-awareness:** refers to diagnosing errors, faults, and failures, while self-repairing focuses on recovery from them [Hinchey and Sterritt 2006]. Self-awareness means that the system is aware of its self states and behaviours. This property is based on self- monitoring which reflects what is monitored.	**Availability** **Survivability** **Maintainability** **Reliability**	
Self-assurance: The definition of self-assurance in Oxford dictionary: Freedom from doubt; belief in yourself and your abilities. The ability of self-adaptive system to verify and validate the adaptation results by providing a continual assessment depending on the dynamic change	**Efficiency** **Performance** **Availability**	
Self-protecting (anticipated adaptation): is the capability of detecting security breaches and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or to mitigate their effects. [Salehie and Tahvildari, 2009]	**Survivability** **Maintainability**	

5.2.1 Relating Quality Attributes with Self-adaptive Properties

To define the quality attributes of a self-adaptive system, a relation between the self-* properties of self-adaptiveness and the quality attributes must be created. Salehie and Tahvildari [Salehie and Tahvildari, 2005] discuss the potential links between these properties and the quality attributes. The link can help to define and elucidate self-* properties and to utilize the existing body of knowledge and quality factors, metrics, and requirements in developing and operating self-adaptive systems. In this research, we describe the line between the self-* property and the quality attributes as shown in Table 5.1.

The Table 5.1 shows the relation of self-* property along with its definition with the associated quality attribute (and its definition), as shown in the third column, that has been proposed by the IEEE standard [Schneidewind, 1998].

Self-configuration is related to modifiability, which can be classified into maintainability and flexibility, as shown in the definition column. Self-optimization is related to availability, efficiency, performance, and modifiability, as can be concluded from their relation to the application's ability to optimize its resources. Self-optimization has a strong correlation with efficiency. As minimizing response time is often one of the primary system requirements, it too impacts functionality. On the other hand, self-protecting has a strong correlation with reliability and can also be linked to functionality.

In addition, self-awareness refers to the system's ability to monitor its own structure and behaviour and can be related to availability and reliability. Context awareness refers to the system's ability to monitor its context and can be linked to availability and reliability. Self-awareness and self-healing are related to availability, survivability, maintainability, and reliability. In self-healing, the main objective is to maximize the availability, survivability, maintainability, and reliability of the system [Ganek and Corbi, 2003].

Self-assurance refers to the application ability to verify and validate the adaptation results by providing a continual assessment of the dynamic changes. This concept has been introduced by Cheng at al. [Cheng et al., 2008], which addresses the adaptation assurance challenge. In the same way, it is necessary for the application to trust its own behaviour, achieved using the available assurance mechanism, as described in section 2.7 of chapter 2. We have adapted the definition of the Oxford English Dictionary of self-assurance: 'Freedom from doubt; belief in yourself and your abilities'. Self-assurance is related to efficiency, responsiveness, and availability. Finally, Self-protection refers to the quality attribute survivability and maintainability. Survivability can be defined as 'A system that can repair itself or degrade gracefully to preserve as much critical functionality as possible in the face of attacks and failures is called a survivable system' [Schneidewind, 1998].

5.3 Evaluation of COCA-middleware with ATAM

The ATAM [Kazman et al., 2002] has evolved as a structured way of analysing the trade-offs in software architecture. ATAM provides a principled approach for evaluating the fitness of software architecture with respect to multiple competing quality attributes. ATAM consists of nine activities. Some of these activities are executed in parallel. The major three activities are present the ATAM, identify the architectural approaches, and present the ATAM evaluation results. The concept of ATAM is presented to the stakeholders and the ATAM steps are outlined, in brief, as shown in the following:

5.3.1 ATAM Evaluation Steps

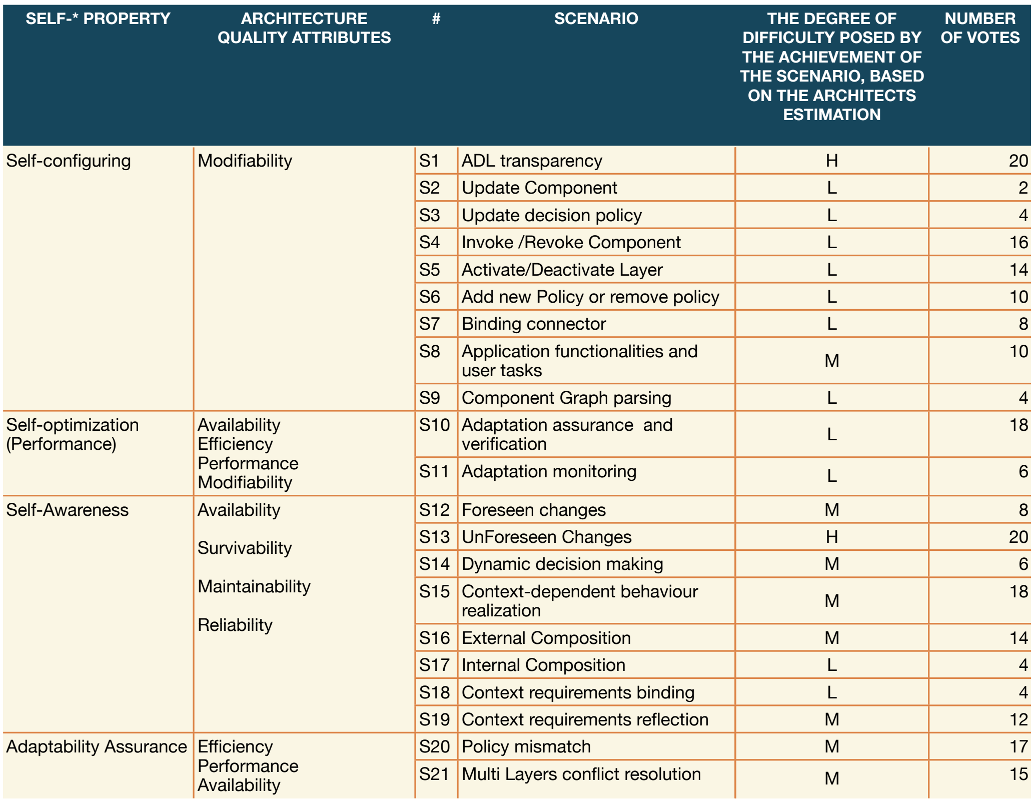
1. Present Business Drivers - Everyone in this step presents and evaluates the business drivers for the system in question, including presenting its important functional requirements, constraints, business, goals, context, and major architectural drivers, including defining the major quality attribute goals that shape the architecture. These goals have been discussed in Section 5.1. In what follows, we have summarized the quality attribute, which measures the self-adaptability of the architecture.

2. Present the Architecture - The architect presents the high-level architecture to the team at an 'appropriate level of detail'. The architecture was presented in-depth in chapter 3. The focus of this section is to evaluate the architecture using ATAM. Technical constraints such as those from OS, hardware, or middleware prescribed for use on other systems with which the system must interact are overcome using architectural approaches to meet quality attribute requirements.

3. Identify architectural approaches - Different architectural approaches to the system are presented by the evaluation team i.e. stockholders, end users, developers, and architects, and are discussed. These architectural approaches define the important structures of the software system and describe the ways in which it can grow, respond to changes, specifically foreseen and unforeseen changes, achieve the level of self-configuring, and

integrate with other services distributed in the environment. In general, the architectural approach can be identified by the self-* properties of the self-adaptive system. These properties are manifold and substantial; they cannot be adapted to capture specific quality attributes in the architecture, based on the division of these properties into more specific features, as shown in the Table 5.2.

Table 5.2: Architecture approach/quality attribute



SELF-* PROPERTY	ARCHITECTURE QUALITY ATTRIBUTES	#	SCENARIO	THE DEGREE OF DIFFICULTY POSED BY THE ACHIEVEMENT OF THE SCENARIO, BASED ON THE ARCHITECTS ESTIMATION	NUMBER OF VOTES
Self-configuring	Modifiability	S1	ADL transparency	H	20
		S2	Update Component	L	2
		S3	Update decision policy	L	4
		S4	Invoke /Revoke Component	L	16
		S5	Activate/Deactivate Layer	L	14
		S6	Add new Policy or remove policy	L	10
		S7	Binding connector	L	8
		S8	Application functionalities and user tasks	M	10
		S9	Component Graph parsing	L	4
Self-optimization (Performance)	Availability Efficiency Performance Modifiability	S10	Adaptation assurance and verification	L	18
		S11	Adaptation monitoring	L	6
Self-Awareness	Availability Survivability Maintainability Reliability	S12	Foreseen changes	M	8
		S13	UnForeseen Changes	H	20
		S14	Dynamic decision making	M	6
		S15	Context-dependent behaviour realization	M	18
		S16	External Composition	M	14
		S17	Internal Composition	L	4
		S18	Context requirements binding	L	4
		S19	Context requirements reflection	M	12
Adaptability Assurance	Efficiency Performance Availability	S20	Policy mismatch	M	17
		S21	Multi Layers conflict resolution	M	15

The table outline the self-* property, the architecture quality attribute, the scenario tile and the description of the proposed scenario. However, the main focus of this research is the self-tuning properties of the software system: self-optimising and self-configuring. In addition, the architecture's ability to perform component composition and configuration based on context-related conditions implies fine-grained tuning and coarse-grained tuning based on the context.

4. Generate quality attribute utility tree - Define the core business and technical requirements of the system, and map them to an appropriate architectural property. Present a scenario for this given requirement. This step identifies, prioritizes, and refines the system's most important quality attribute goal. This is a crucial step in that it guides the reminder of the analysis. The output of the utility tree generation step is a prioritization of specific quality attribute requirements, realized as scenarios. The utility tree makes the quality attribute requirements concrete, thus forcing the architecture to define the relevant quality requirements precisely.

   The quality attribute scenarios at the leaf node of the utility tree are now specific enough to be prioritized relative to each other and be analysed. The prioritization may be on a scale from 0 to 10 or may use relative ranking such as high (H), medium (M), and low (L). The utility tree is prioritized along two dimensions (1) by the importance of each scenario to the success of the system and (2) by the degree of difficulty posed by the achievement of the scenario, based on the architecture's estimation. The scenarios that are marked with (H,H) are prime candidates for the analysis. Thereafter, the scenarios marked with (M,H) and (H,M) are analysed. The scenarios labelled (L) are not likely to be examined, as they have little importance and have low expected difficulty. The utility tree is shown in Figure 5.1

5. Analyse architectural approaches - Analyse each scenario, giving priority ratings. The architecture is then evaluated against each scenario. However, this evaluation focuses on the self-adaptive ability in the unanticipated condition, self-configuration of the application structure (based on the current context in both layers of granularity), and self-optimization of its behaviour (in response to context changes and decision policies rules violation). The analysis concluded that several quality attributes should be included in the evaluation, as shown in Table 5.3.

   Architectural introspection and intercession are important attributes of any self-adaptive system. They must be included in the analysis. However, the property of self-optimization

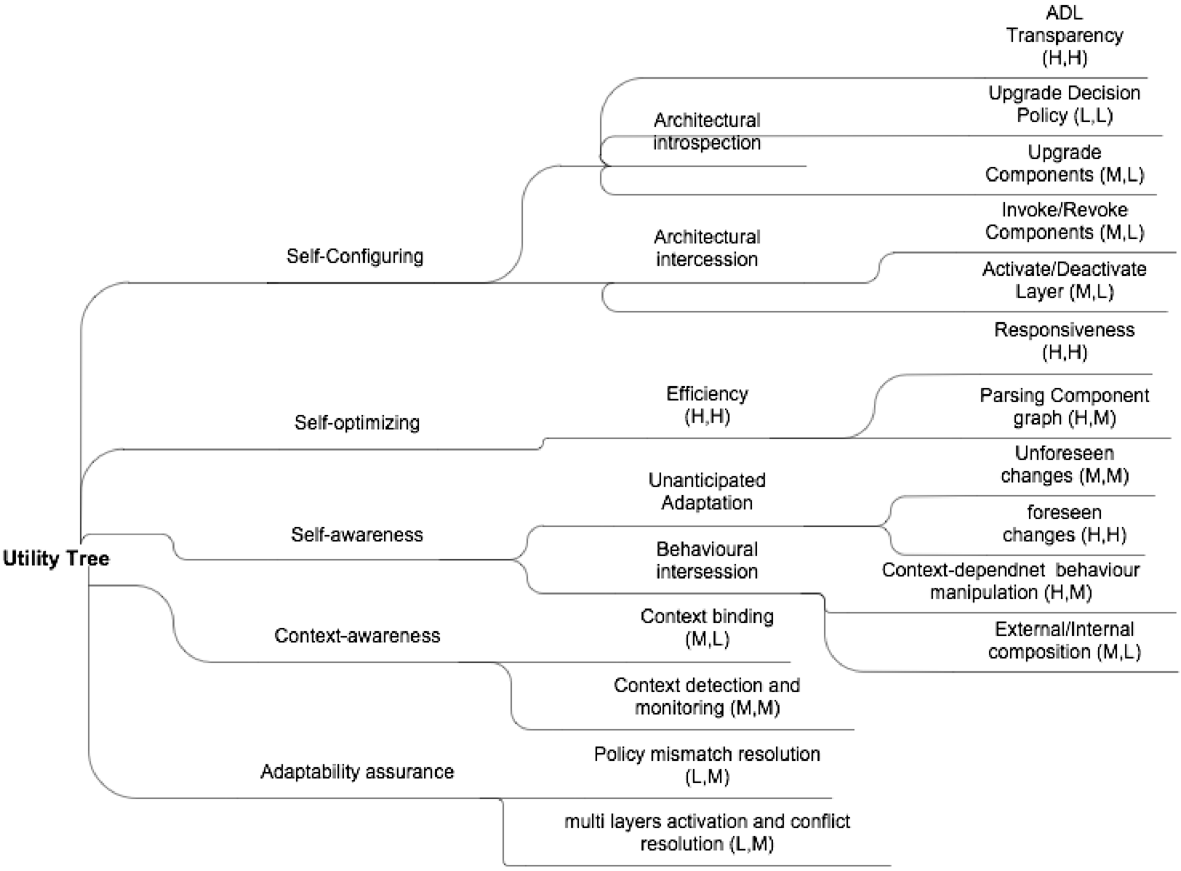
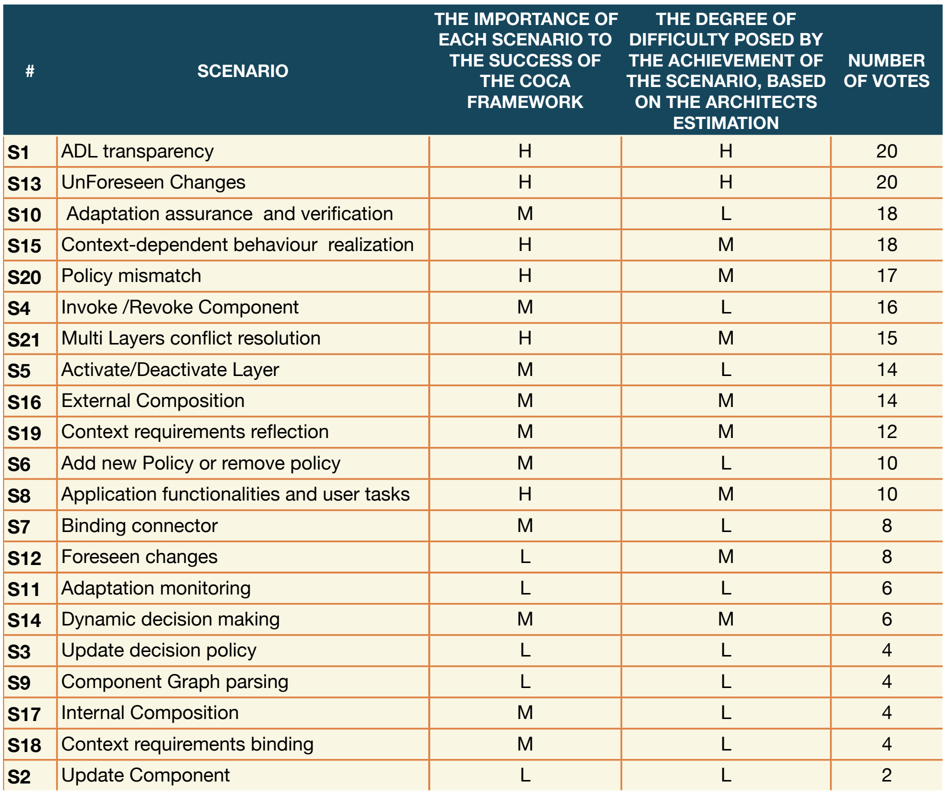

Utility Tree
Self-Configuring
Self-optimizing
Self-awareness
Context-awareness
Adaptability assurance
Architectural introspection
Architectural intercession
Efficiency (H,H)
Unanticipated Adaptation
Behavioural intersession
Context binding (M,L)
Context detection and monitoring (M,M)
Policy mismatch resolution (L,M)
multi layers activation and conflict resolution (L,M)
ADL Transparency (H,H)
Upgrade Decision Policy (L,L)
Upgrade Components (M,L)
Invoke/Revoke Components (M,L)
Activate/Deactivate Layer (M,L)
Responsiveness (H,H)
Parsing Component graph (H,M)
Unforeseen changes (M,M)
foreseen changes (H,H)
Context-dependnet behaviour manipulation (H,M)
External/Internal composition (M,L)

Fig. 5.1: Quality attribute utility tree

can be characterized by the responsiveness and efficiency of the system. Self-awareness can be addressed using behavioural introspection and intercession and the ability to achieve unanticipated adaptation dynamically at runtime, which includes dynamic decision-making, context monitoring and continuous adaptation. Self-adaptation assurance can be evaluated in terms of the ability to reason about policy mismatch and multi-layer conflict.

6. Brainstorm and prioritize scenarios - The proposed scenarios are presented to the larger stakeholder group and expand on them. Table 5.3 shows the number of votes made by

Table 5.3: Scenario ratings and number of votes



#	SCENARIO	THE IMPORTANCE OF EACH SCENARIO TO THE SUCCESS OF THE COCA FRAMEWORK	THE DEGREE OF DIFFICULTY POSED BY THE ACHIEVEMENT OF THE SCENARIO, BASED ON THE ARCHITECTS ESTIMATION	NUMBER OF VOTES
S1	ADL transparency	H	H	20
S13	UnForeseen Changes	H	H	20
S10	Adaptation assurance and verification	M	L	18
S15	Context-dependent behaviour realization	H	M	18
S20	Policy mismatch	H	M	17
S4	Invoke /Revoke Component	M	L	16
S21	Multi Layers conflict resolution	H	M	15
S5	Activate/Deactivate Layer	M	L	14
S16	External Composition	M	M	14
S19	Context requirements reflection	M	M	12
S6	Add new Policy or remove policy	M	L	10
S8	Application functionalities and user tasks	H	M	10
S7	Binding connector	M	L	8
S12	Foreseen changes	L	M	8
S11	Adaptation monitoring	L	L	6
S14	Dynamic decision making	M	M	6
S3	Update decision policy	L	L	4
S9	Component Graph parsing	L	L	4
S17	Internal Composition	M	L	4
S18	Context requirements binding	M	L	4
S2	Update Component	L	L	2

the evaluation team for each scenario and their importance for the evaluation process. Scenarios S1, S5, S8, S13, S14, S15, S20, and S21 have been marked with higher importance and have gained the highest number of votes, as shown in the Table. The next group of scenarios, i.e. S4, S6, S10, S16, S19, S20, and S2, come next in importance. The remaining scenarios marked with less priority, have been ignored during the evaluation because of their importance rank.

7. Present results - The scenarios that have the highest importance (S1, S5, S8, S13, S14, S15, S20, and S21) are shown in the following sections.

5.3.2 ATAM Evaluation Results

The evaluation team presents the ATAM analysis for each scenario and illustrates the results in the tables by performing the following steps:

Step 1: Identifying the quality attributes. For each scenario the evaluation team identified the related quality attributes to the scenario based on the utility tree illustrated in Figure 5.1 and the architecture approach illustrated in Table 5.2.

Step 2: Identifying the environment conditions. This field describes the execution environment conditions and proposes a set of context conditions.

Step 3: Identifying the environment stimulus. This field identifies the events that cause the architecture to respond to changes. These events need to be expressed in terms that are concrete (measurable or observable).

Step 4: Identifying the architecture response. This field identifies measurable/observable quantities, which addresses how the architecture responds to the environment stimulus.

Step 5: Identifying the architecture decision This field identifies the aspects of the architecture that have a direct impact on achieving attribute responses. The evaluation team proposes the action that might be taken by the architecture for reasoning about the architecture 's stimulus.

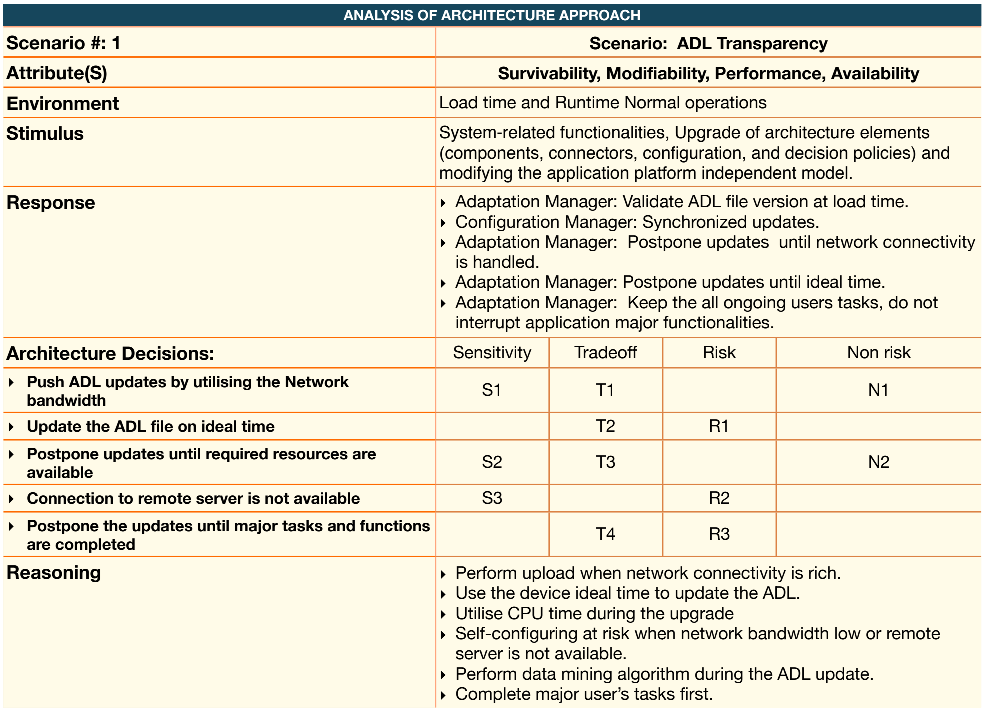
Step 5: Identifying the sensitivity points. The evaluation team starts identifying the sensitivity points, related decisions, and quality attributes related to each scenario. A sensitivity point is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute [Kazman et al., 2002]. Sensitivity points tell a designer or analysts where to pay more attention when trying to understand the achievement of a quality goal.

Step 6: Identifying the tradeoff points. After identifying the sensitivity points, the evaluation team considers the tradeoff points between the quality attributes. A tradeoff point is a property that affects more than one attribute and it is a sensitivity point for more than one attribute.

Step 7: Identifying the risks and non-risks points The risk points are identified on the basis that when the decision taken might have a risk over the quality attributes.

Step 8: Identifying the reasoning mechanism. The reasoning provides valid mechanisms that can be used to reason about sensitive points and risk points identified in previous steps.

Table 5.4: Evaluation results of scenario 1. ADL transparency

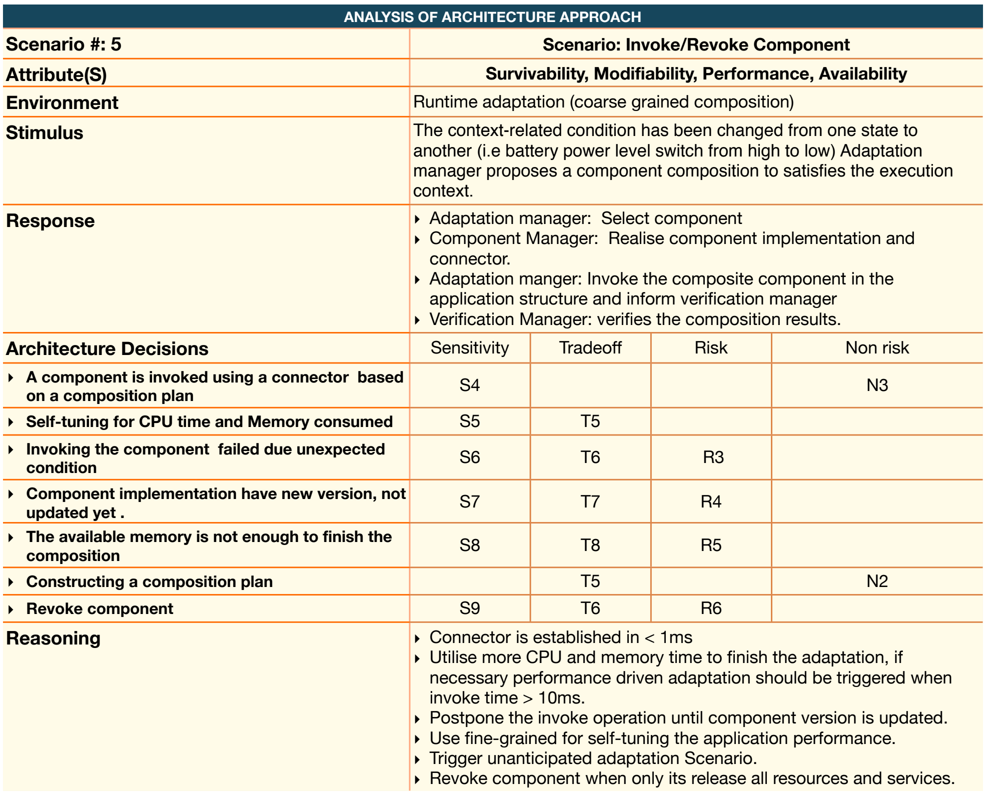


ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #: 1	**Scenario: ADL Transparency**			
Attribute(S)	**Survivability, Modifiability, Performance, Availability**			
Environment	Load time and Runtime Normal operations			
Stimulus	System-related functionalities, Upgrade of architecture elements (components, connectors, configuration, and decision policies) and modifying the application platform independent model.			
Response	‣ Adaptation Manager: Validate ADL file version at load time. ‣ Configuration Manager: Synchronized updates. ‣ Adaptation Manager: Postpone updates until network connectivity is handled. ‣ Adaptation Manager: Postpone updates until ideal time. ‣ Adaptation Manager: Keep the all ongoing users tasks, do not interrupt application major functionalities.			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
‣ **Push ADL updates by utilising the Network bandwidth**	S1	T1		N1
‣ **Update the ADL file on ideal time**		T2	R1	
‣ **Postpone updates until required resources are available**	S2	T3		N2
‣ **Connection to remote server is not available**	S3		R2	
‣ **Postpone the updates until major tasks and functions are completed**		T4	R3	
Reasoning	‣ Perform upload when network connectivity is rich. ‣ Use the device ideal time to update the ADL. ‣ Utilise CPU time during the upgrade ‣ Self-configuring at risk when network bandwidth low or remote server is not available. ‣ Perform data mining algorithm during the ADL update. ‣ Complete major user's tasks first.			

Table 5.4 shows the results of the evaluation of scenario 1, referring to the transparency of architecture description language ADL. This refers to the ability of the architecture to generate and update the ADL based on the abstract model provided in the MDA phase, i.e. the platform independent model (PIM). The synchronization process between the PIM and the ADL must be performed in a manner that is separate from the application functionality and without interrupting user tasks.

The results of an evaluation of scenario number 5 are shown in Table 5.5. These re-

Table 5.5: Evaluation results of scenario 5. Components composition

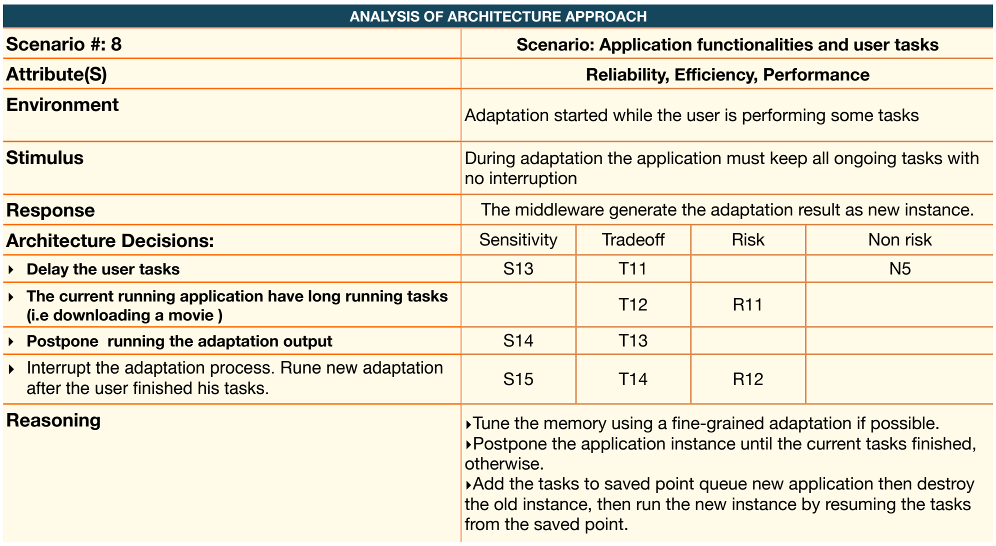


ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #: 5	**Scenario: Invoke/Revoke Component**			
Attribute(S)	**Survivability, Modifiability, Performance, Availability**			
Environment	Runtime adaptation (coarse grained composition)			
Stimulus	The context-related condition has been changed from one state to another (i.e battery power level switch from high to low) Adaptation manager proposes a component composition to satisfies the execution context.			
Response	‣ Adaptation manager: Select component ‣ Component Manager: Realise component implementation and connector. ‣ Adaptation manger: Invoke the composite component in the application structure and inform verification manager ‣ Verification Manager: verifies the composition results.			
Architecture Decisions	Sensitivity	Tradeoff	Risk	Non risk
‣ **A component is invoked using a connector based on a composition plan**	S4			N3
‣ **Self-tuning for CPU time and Memory consumed**	S5	T5		
‣ **Invoking the component failed due unexpected condition**	S6	T6	R3	
‣ **Component implementation have new version, not updated yet .**	S7	T7	R4	
‣ **The available memory is not enough to finish the composition**	S8	T8	R5	
‣ **Constructing a composition plan**		T5		N2
‣ **Revoke component**	S9	T6	R6	
Reasoning	‣ Connector is established in < 1ms ‣ Utilise more CPU and memory time to finish the adaptation, if necessary performance driven adaptation should be triggered when invoke time > 10ms. ‣ Postpone the invoke operation until component version is updated. ‣ Use fine-grained for self-tuning the application performance. ‣ Trigger unanticipated adaptation Scenario. ‣ Revoke component when only its release all resources and services.			

sults reveal that a component invoked based on a composition is a performance-related attribute. There is a trade-off between reaching the quality attribute self-configuration and self-optimization that requires the application to adapt a self-tuning mechanism when the associated architecture decision has risk points R3, R4, R5 or R6. The non-risk points N3 and N2 show the architecture ability with no risk on the device performance.

The evaluation results of scenario number 8 are shown in Table 5.6. The result from the evaluation reveals that during adaptation, application functionalities and users tasks should not be interpreted. Invoking components based on a composition plan is considered as a trade-off between performance and modifiability. The sensitivity of this kind of architecture decision, S6, refers to the consumption of memory, which might, therefore, affect the functionality, responsiveness, and the effectiveness of the application. The same decision has a trade-off point between the efficiency of the adaptation and the responsiveness of the architecture, T8 in the table. The risk R11 incurred from adapting such a decision can lead to a risk of postponing the user's tasks for long period of time until the adaptation is completed; the users' tasks can be estimated to finish before the adaptation is accomplished or the application adaptation is interrupted to allow the user to finish. In the reasoning field, a description of a valid mechanism to handle such a risk is provided; this mechanism must be reflected in the COCA platform design, as shown in the following section.

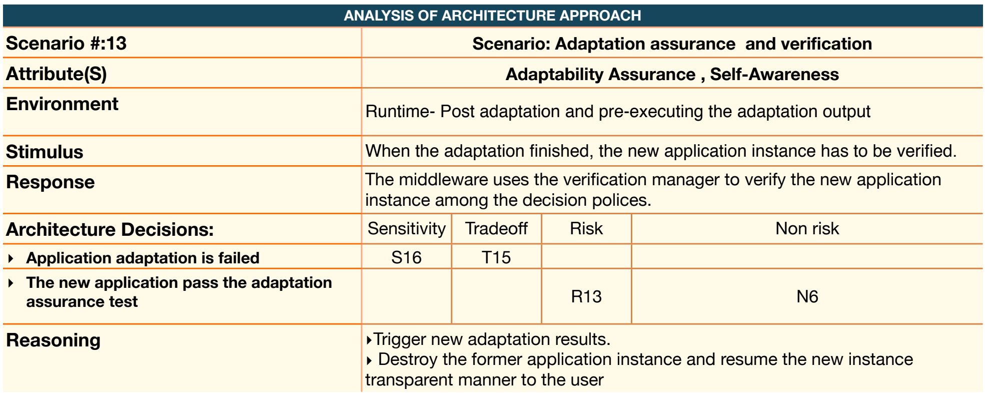
Table 5.6: Evaluation results of scenario 8. User tasks



ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #: 8	**Scenario: Application functionalities and user tasks**			
Attribute(S)	**Reliability, Efficiency, Performance**			
Environment	Adaptation started while the user is performing some tasks			
Stimulus	During adaptation the application must keep all ongoing tasks with no interruption			
Response	The middleware generate the adaptation result as new instance.			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
▸ **Delay the user tasks**	S13	T11		N5
▸ **The current running application have long running tasks (i.e downloading a movie)**		T12	R11	
▸ **Postpone running the adaptation output**	S14	T13		
▸ Interrupt the adaptation process. Rune new adaptation after the user finished his tasks.	S15	T14	R12	
Reasoning	▸Tune the memory using a fine-grained adaptation if possible. ▸Postpone the application instance until the current tasks finished, otherwise. ▸Add the tasks to saved point queue new application then destroy the old instance, then run the new instance by resuming the tasks from the saved point.			

The evaluation results of scenario number 13 are shown in Table 5.7. This scenario captures the architecture's ability to provide adaptation assurance, thus satisfying the unanticipated adaptation quality attribute. The analysis has shown that realization of the context-dependent behaviour, that is, to combine the related policies, has a sensitive point in self-awareness, with no risk to the quality attribute. On the other hand, a trade-off between responsiveness/efficiency and unanticipated adaptation/self-awareness is detected by risk points R12, R13, R14, and R15. In other words, to reach this level of self-adaptability and self-awareness, the application must trade performance and efficiency, by paying less attention to them while paying higher attention to the self-adaptability. This does not mean that the performance and responsiveness are neglected, rather, they will be affected during unanticipated adaptation. The reasoning field in the Table provides valid procedures that must be adapted when facing such risk points. In some cases, the application might prompt the user control the decision policy when the self-assurance process failed to verify the fitness of the policy among the application goals and objectives. On other cases, the adaptation manager requests, the context state from the context manager to verify the fitness of the taken decision among them.

Table 5.7: Evaluation results of scenario 13. Adaptation assurance and verification

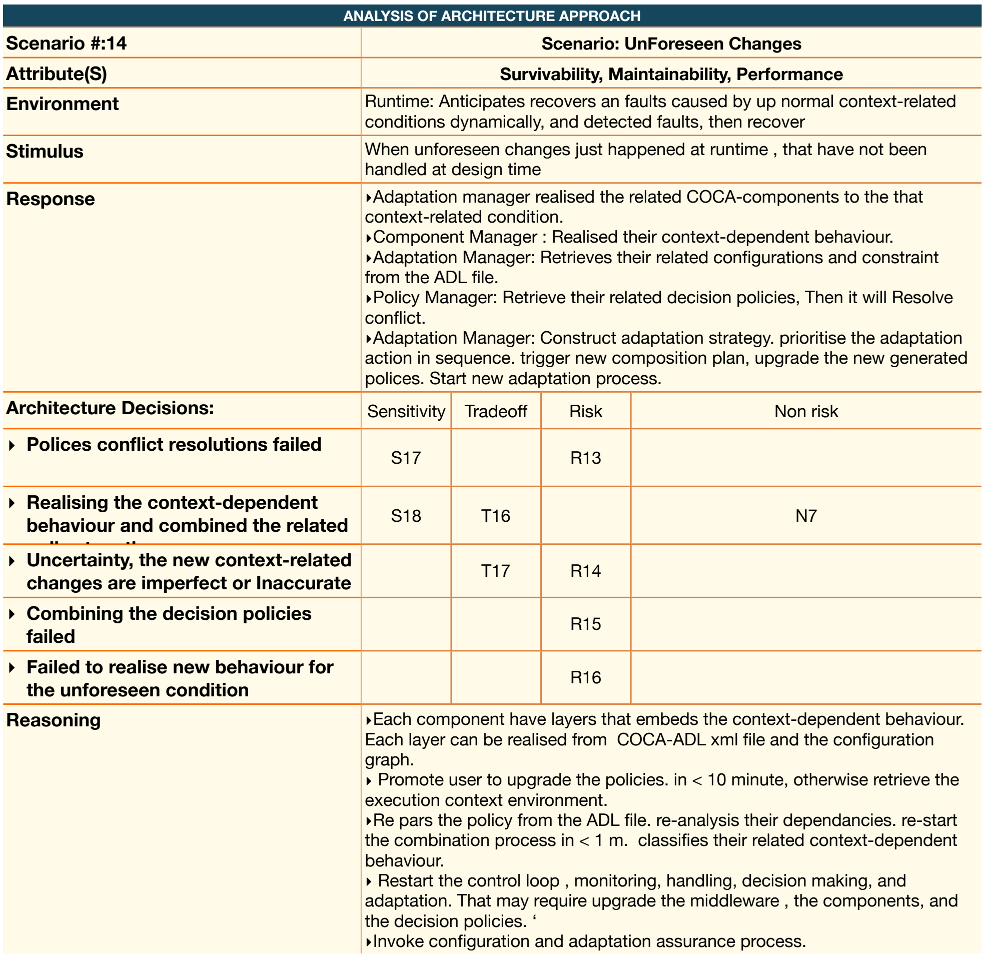


ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #:13	**Scenario: Adaptation assurance and verification**			
Attribute(S)	**Adaptability Assurance , Self-Awareness**			
Environment	Runtime- Post adaptation and pre-executing the adaptation output			
Stimulus	When the adaptation finished, the new application instance has to be verified.			
Response	The middleware uses the verification manager to verify the new application instance among the decision polices.			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
‣ **Application adaptation is failed**	S16	T15		
‣ **The new application pass the adaptation assurance test**			R13	N6
Reasoning	‣Trigger new adaptation results. ‣ Destroy the former application instance and resume the new instance transparent manner to the user			

The analysis in Table 5.8 shows that dynamic decision making has a sensitivity point to self-awareness because of continuous context monitoring, a result which has a direct impact on the performance and responsiveness (T12). The risk from making architecture decisions of a filer on achieving a decision implies the whole process of adaptation failed. As a reasoning mechanism, the architecture proposes an action of promoting the user to upgrade the related decision policy as the first action. If it is not valid, the middleware retrieves the context condition and restarts the decision making. During this process, all policies are upgraded and then evaluated; conflicts are removed within the specified time.

The analysis shown in Table 5.9 indicates that the context-dependent behaviour of combining the related policies has a sensitive point to the self-awareness with no risk to the quality attribute. On the other hand, a trade-off between responsiveness/efficiency and unan-

Table 5.8: Evaluation results of scenario 14. Unforeseen changes

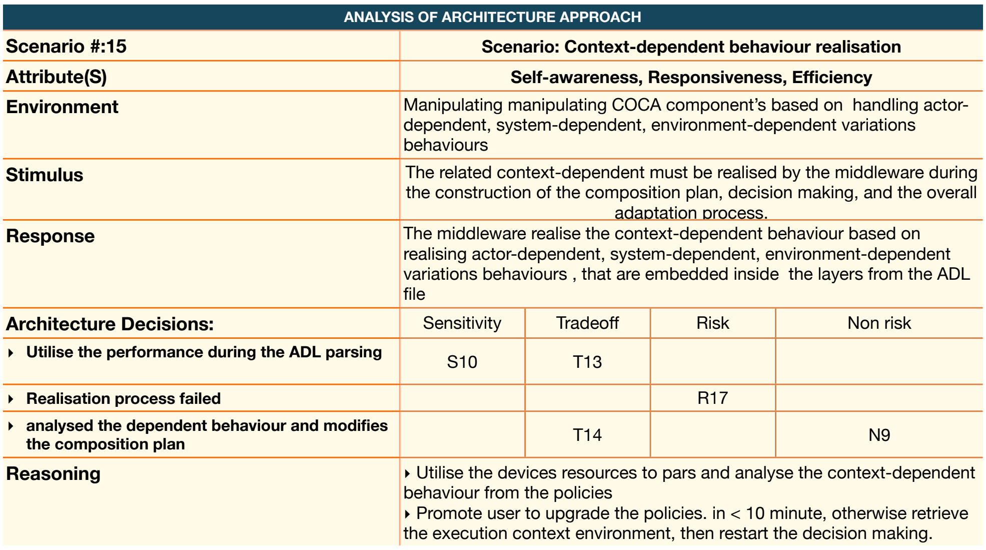


ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #:14	**Scenario: UnForeseen Changes**			
Attribute(S)	**Survivability, Maintainability, Performance**			
Environment	Runtime: Anticipates recovers an faults caused by up normal context-related conditions dynamically, and detected faults, then recover			
Stimulus	When unforeseen changes just happened at runtime , that have not been handled at design time			
Response	▸Adaptation manager realised the related COCA-components to the that context-related condition. ▸Component Manager : Realised their context-dependent behaviour. ▸Adaptation Manager: Retrieves their related configurations and constraint from the ADL file. ▸Policy Manager: Retrieve their related decision policies, Then it will Resolve conflict. ▸Adaptation Manager: Construct adaptation strategy. prioritise the adaptation action in sequence. trigger new composition plan, upgrade the new generated polices. Start new adaptation process.			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
▸ **Polices conflict resolutions failed**	S17		R13	
▸ **Realising the context-dependent behaviour and combined the related**	S18	T16		N7
▸ **Uncertainty, the new context-related changes are imperfect or Inaccurate**		T17	R14	
▸ **Combining the decision policies failed**			R15	
▸ **Failed to realise new behaviour for the unforeseen condition**			R16	
Reasoning	▸Each component have layers that embeds the context-dependent behaviour. Each layer can be realised from COCA-ADL xml file and the configuration graph. ▸ Promote user to upgrade the policies. in < 10 minute, otherwise retrieve the execution context environment. ▸Re pars the policy from the ADL file. re-analysis their dependancies. re-start the combination process in < 1 m. classifies their related context-dependent behaviour. ▸ Restart the control loop , monitoring, handling, decision making, and adaptation. That may require upgrade the middleware , the components, and the decision policies. ‘ ▸Invoke configuration and adaptation assurance process.			

ticipated adaptation/self-awareness is detected by risk point R17. In different words, to reach this level of realization, the application must trade the performance and efficiency, by paying less attention to them while paying higher attention to the self-adaptability. This does not mean the performance and responsiveness are neglected; however, they will be affected during unanticipated adaptation.

The analysis shown in Table 5.10 indicates that performance of policy conflict resolution has a sensitive point within the adaptation assurance of the quality attribute with a trade-off between self-awareness and performance. On the other hand, the trade-off between having a

Table 5.9: Evaluation results of scenario 15. Unanticipated adaptation



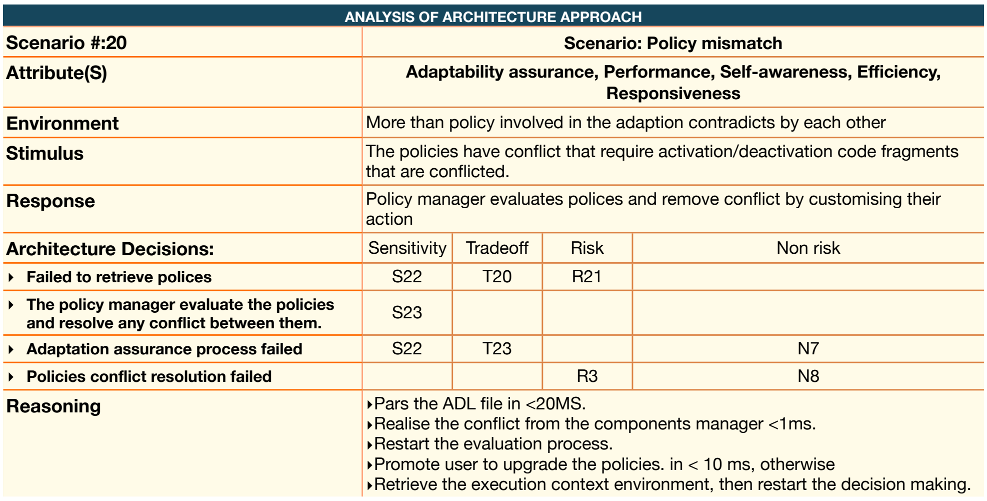
ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #:15	**Scenario: Context-dependent behaviour realisation**			
Attribute(S)	**Self-awareness, Responsiveness, Efficiency**			
Environment	Manipulating manipulating COCA component's based on handling actor-dependent, system-dependent, environment-dependent variations behaviours			
Stimulus	The related context-dependent must be realised by the middleware during the construction of the composition plan, decision making, and the overall adaptation process.			
Response	The middleware realise the context-dependent behaviour based on realising actor-dependent, system-dependent, environment-dependent variations behaviours , that are embedded inside the layers from the ADL file			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
▸ **Utilise the performance during the ADL parsing**	S10	T13		
▸ **Realisation process failed**			R17	
▸ **analysed the dependent behaviour and modifies the composition plan**		T14		N9
Reasoning	▸ Utilise the devices resources to pars and analyse the context-dependent behaviour from the policies ▸ Promote user to upgrade the policies. in < 10 minute, otherwise retrieve the execution context environment, then restart the decision making.			

sensitive point in the adaptation-assurance quality attribute and performance/responsiveness is T23. In other words, there is a risk of not achieving the adaptation due to a failure in the policy mismatch mechanism (high risk in the adaptation action). As a precipitation procedure, the architects propose a reasoning mechanism that includes prompting the user to upgrade the policies in $<$ 10 millisecond, otherwise will it will retrieve the execution context environment and thereafter restart the decision making.

The analysis shown in Table 5.11 indicates that performing layers conflict with resolution has a sensitive point to the adaptation assurance quality attribute with a trade-off between self-awareness and performance. The impact of policy retrieving has a tradeoff among self-awareness, application efficiency, and responsiveness.

Another captured sensitive point is S20, which related to the policy manager and that might consume more CPU time during the policies' evaluation process. This point has a trade-off among adaptability assurance, performance, and responsiveness, called T23. In the same point, a trade-off point has been captured in scenario 20. For more information, please refer to Table 5.10. In the same way, R3 and R21 have been captured in scenarios 5 and 21.

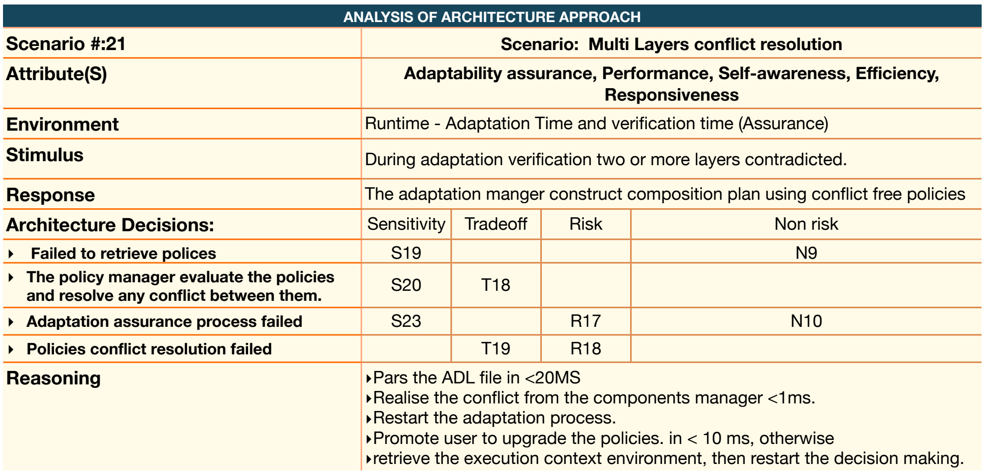
Table 5.10: Evaluation results of scenario 20. Policy mismatch



ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #:20	**Scenario: Policy mismatch**			
Attribute(S)	**Adaptability assurance, Performance, Self-awareness, Efficiency, Responsiveness**			
Environment	More than policy involved in the adaption contradicts by each other			
Stimulus	The policies have conflict that require activation/deactivation code fragments that are conflicted.			
Response	Policy manager evaluates polices and remove conflict by customising their action			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
‣ **Failed to retrieve polices**	S22	T20	R21	
‣ **The policy manager evaluate the policies and resolve any conflict between them.**	S23			
‣ **Adaptation assurance process failed**	S22	T23		N7
‣ **Policies conflict resolution failed**			R3	N8
Reasoning	‣Pars the ADL file in <20MS. ‣Realise the conflict from the components manager <1ms. ‣Restart the evaluation process. ‣Promote user to upgrade the policies. in < 10 ms, otherwise ‣Retrieve the execution context environment, then restart the decision making.			

This provides a clear correlation between the same risk points that are generated by many different scenarios.

Table 5.11: Evaluation results of scenario 21. Conflict resolution



ANALYSIS OF ARCHITECTURE APPROACH				
Scenario #:21	**Scenario: Multi Layers conflict resolution**			
Attribute(S)	**Adaptability assurance, Performance, Self-awareness, Efficiency, Responsiveness**			
Environment	Runtime - Adaptation Time and verification time (Assurance)			
Stimulus	During adaptation verification two or more layers contradicted.			
Response	The adaptation manger construct composition plan using conflict free policies			
Architecture Decisions:	Sensitivity	Tradeoff	Risk	Non risk
‣ **Failed to retrieve polices**	S19			N9
‣ **The policy manager evaluate the policies and resolve any conflict between them.**	S20	T18		
‣ **Adaptation assurance process failed**	S23		R17	N10
‣ **Policies conflict resolution failed**		T19	R18	
Reasoning	‣Pars the ADL file in <20MS ‣Realise the conflict from the components manager <1ms. ‣Restart the adaptation process. ‣Promote user to upgrade the policies. in < 10 ms, otherwise ‣retrieve the execution context environment, then restart the decision making.			

5.4 Lessons Learned from ATAM Evaluation

1. The evaluation verifies the COCA middleware's ability to perform dynamic assurance and verification of the adaptation output. This is achieved by parsing the policies from the ADL XML file, the polices located in the configuration element of the ADL. However, the COCA middleware subscribes to the context manager to be notified about contextual changes that are related to the device resources. The policy manager evaluates the polices among the recently violated architecture properties. This is achieved by adapting the notification pattern for the context manager. This achieves attributes A1.2, A2.2, A4.1, and A4.2 of objectives 1, 2, and 4.

2. The COCA middleware can parse and upgrade the architecture ADL in a transparent manner. However, to enhance the architecture's survivability, modifiability, performance, and availability, it is recommended to adapt the Flyweight design pattern [Buck and Yacktman, 2010] for implementing the adaptation manager, specifically, the functions of parsing and constructing the composition plan. The Flyweight pattern minimizes the amount of memory and/or processor overhead required to use objects at runtime. This achieves attribute A2.1 of objective 2.

3. The COCA middleware can invoke or revoke a specific component dynamically without affecting the device performance; however, then the component manager must testify the component's ability to provide/require specific methods, services, and resources. This is achieved by adapting the bundle design pattern with the support of the verification manager. The verification manager is recommended to adapt the 'responder chain' design pattern and adapt the perform selector and delayed perform pattern that implements responses to the selector method [Buck and Yacktman, 2010]. However, dynamic loading of component code is achieved by adapting the dynamic creation pattern, which enables the COCA-middleware to create new instances of COCA-components, that did not exist at the time the application was complied. This achieves attribute A3.1 of objective 3.

4. The interoperability of the application components and their sub-layers is a major feature of COCA middleware. The evaluation shows the architecture's ability to deactivate or activate sub-layer implementation when driven by a context change. However, this feature is supported by adapting two design patterns when implementing the adaptation manager. The invocation and delegation patterns can support introspection and intercession at the level of the COCA component. Use of invocation patterns is a technique for preserving the state of messages, arguments, and return values. Invocations can be used to completely decouple the sender of a message from the receiver. The sender and receiver can be in different processes or be separated by time. The delegate is an object that is given an opportunity to react to changes in another object or influence the behaviour of another object [Buck and Yacktman, 2010]. This achieves attributes A3.1 and A3.2 of objective 3.

5. The policy manager can perform decision policy mismatches; however, it has some drawbacks in terms of device performance. Therefore, it is recommended that the decision policies adapt an associative storage design pattern that organizes the polices into data and keys so that data can be quickly and easily accessed using the corresponding key. In addition, the policy manager must adapt the archiving and unarchiving design pattern that preserves polices objects, including any interrelationships or dependencies among the archived policy. Archived policies are stored in a binary data, which tends to be fast to read and write from a memory or a disk and fast to transmit over a network. This reduces the overhead of continuous evaluation of the decision policies through the adaptation action. This achieves attributes A2.3, A4.1, and A4.2 of objectives 2 and 4.

6. The evaluation showed that the architecture can anticipate unforeseen changes dynamically. This is achieved by integrating policy mismatch resolution with architecture introspection, as well as instantiating a composition plan. Then, the adaptation manager verifies the fitness of the composition plan from among the unforeseen changes. This achieves attributes A4.1 and A4.2 of Objective 4.

7. Performance and modifiability do, in fact, trade-off with each other. This requires the evaluation of the COCA-middleware and a case study application of their ability to perform self-tuning to maintain architecture performance and self-configuration, and to maintain the modifiability property. This evaluation is performed in the following section.

5.5 Context-oriented Software Evaluation

This section focuses on evaluating the performance and modifiability quality attributes of context-oriented software, including the COCA-middleware and the case study implementation of the I-TrinityTour application. Aspect Oriented Software Development (AOSD) [Filman et al., 2004] and Context Oriented Software Development (COSD) are the alternatives for the design and construction of self-adaptive software. Their ultimate goal is to support the adaptability and variability of software systems, and to be able to reduce development cost and effort, while improving the software modularity and complexity. This motivates us to evaluate these technologies with respect to their ability to support software adaptability (modifiability) and the performance gain from using these technologies to implement the case study application in a mobile computing environment. This thesis claimed that COSD is better suited to dynamic context-driven adaptation in the mobile computing domain. To this end, an evaluation of the two major paradigms (AOSD and COSD) is required to find out which one is better suited to developing self-adaptive applications.

5.5.1 Metrics

In the first experiment, the case study application I-TrinityTour was implemented as a real iPhone application using the COSD and AOSD paradigms. The second experiments compared the performance of the COCA-middleware implementation with several frameworks and middleware architectures including Java Context-Oriented Programming (JCOP) [Schuster et al., 2011], Java COntext Oriented Language (JCOOL) [Sindico and Grassi, 2009], Mobile

USers In Ubiquitous Computing (MUSIC) [Geihs et al., 2011], and Mobility and ADaptation enAbling Middleware (MADAM) [Mikalsen et al., 2006], as described in section 5.7. Then the following questions were analysed. First, how expensive is it to perform context monitoring? Secondly, what is the effect on the allocated resources after the software has performed context detection, particularly when multiple heterogeneous events are detected? Thirdly, what is the performance gain of activating and executing multiple and collaborated aspects in comparison with Context-oriented Component (COCA-component) composition, in response to multiple context change events arriving at the same time. The following experiments focus on evaluating each paradigm implementation of the I-TrinityTour to support adaptability and dependability, based on the following criteria.

1. **Battery Usage.** This criterion evaluates the device's battery durability while running the I-TrinityTour application and performing the adaptation processes, including context monitoring, detecting, decision-making, and adaptation. The Instruments tool provides a relative energy usage on a scale of 0 to 20. These values explain how expensive it is with respect to the battery life to run a specific process over the execution time.

2. **CPU Activity.** This criterion analyses the CPU activities and CPU time required for performing the adaptation processes, including context monitoring, detecting, decision-making, and adaptation. This includes the time required for components/aspects composition in response to multiple and heterogeneous context change events.

3. **Real Memory Allocation.** This criterion measures the amount of memory allocated by the application during the execution of particular functionalities, including context monitoring, context detection, and adaptation.

4. **Sleep/Wake.** This parameter captures the I-TrinityTour application's ability to adjust its activity while the device is running in sleep mode. Normally, if the application keeps running in the background and performs some kind of operation, for example, updating the current location while the device is in sleep mode, the allocated resources

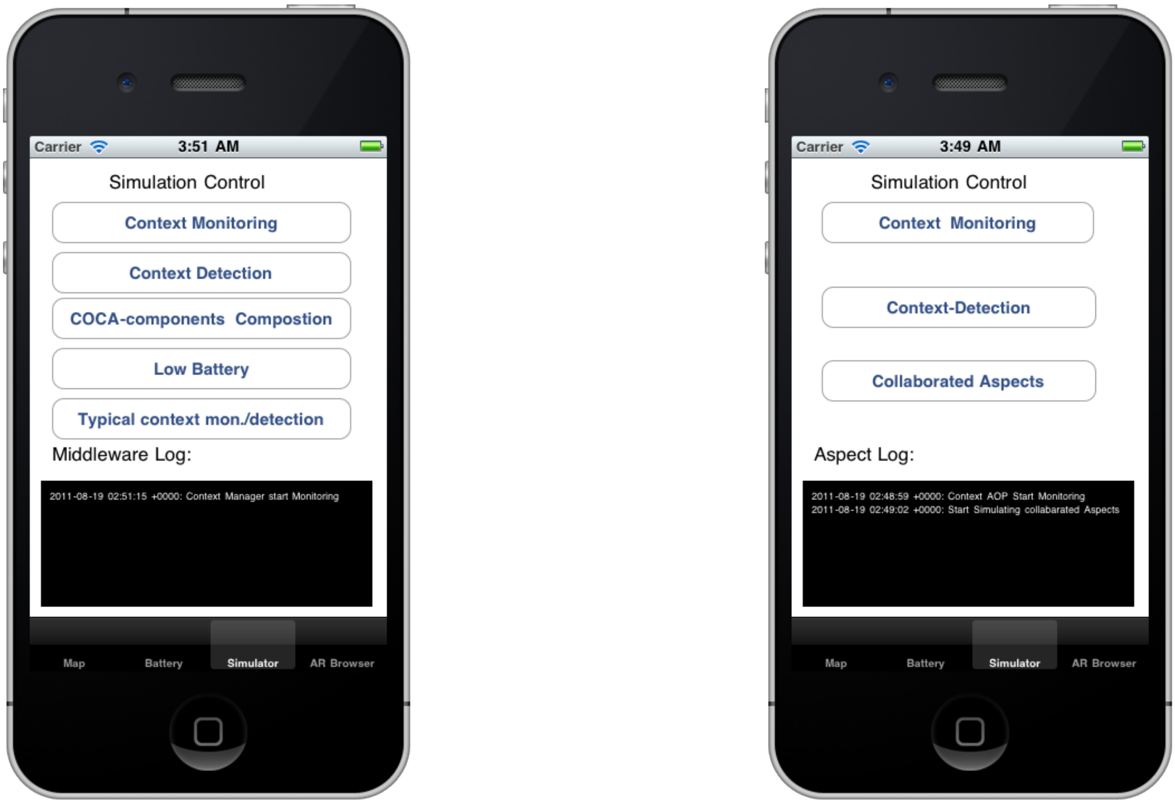
are intensively degraded. This feature evaluates the architecture's ability to adjust the application behaviour, while considering the interoperability between the middleware functionality and the allocated resources.

5. **Adaptation/reconfiguration time.** This criterion captures the required time for the application to adapt its structure and behaviour by adding, removing, or updating components/services. The adaptation time was measured from the start to the end of the adaptation action. The reconfiguration time measures the time required by the middleware to load and execute the plug-in (bundle) implementation.

5.5.2 Hardware and Software Configuration

The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing each adaptation process separately. These values were measured using the energy diagnostics and activity-monitoring tools, which analyses the running application on the iPhone device [Apple IPhone Operating System IOS, 2011]. The Energy Diagnostic tool was used to measure the battery while the device was not connected to an external power supply; after the experiment was finished, the data were imported from the iPhone and then analysed. For measuring the CPU activity, CPU time and the real allocated memory. The I-TrinityTour application was executed on the same IPhone devices for each paradigms/frameworks implementation in separate. The instrument tool was executed on Macbook pro, which traces the data from the IPhone device using the activity monitoring tool. The IPhone was connected wirelessly with the activity monitoring tool. This allows the tool to capture more accurate data for the energy usage and the CPU activity with respect to each process under evaluation. To testify variation in the application behaviour, a simulator was included with each I-TrinityTour implementations. The simulator, UI in Figure 5.2, is used to allow the user to simulate specific context changes, which are used to test the application's ability to adapt the desired behaviour. For each adaptation process, the experiment was established as follows:

Context monitoring. The CPU activity, CPU time, real memory allocation, and energy



```
2011-08-19 02:51:15 +0000: Context Manager start Monitoring
```

```
2011-08-19 02:48:59 +0000: Context AOP Start Monitoring
2011-08-19 02:49:02 +0000: Start Simulating collabarated Aspects
```

(a) COCA-ITrinityTour SimulatorUI

(b) DAOP-ITrinityTour SimulatorUI

Fig. 5.2: Simulator Control User Interface

usage are measured for performing context monitoring at two different time intervals. First, when the application ”did Finish Launching With Options” and the UI views did loaded. Second, the simulator interface shown in Figure 5.2 was used to trigger the context monitoring process, this operation excluded any events related to the application load time. In addition, it allows us to estimate the time required to process 10 contextual events enqueued at the same time. The simulator generates these events generally in First In First Out (FIFO) order. This experiment was executed 200 times, then the variance and standard deviation were calculated for the above criteria.

Context detection. The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing context detection with the aid of the simulator (see figure 5.2).When the context detection button was pressed, the simulator generates and en-queue multiple events, which are executed in First In First Out (FIFO) order. This allows the experiments to evaluate the required time to process these context events, and the time required to perform the reasoning action (decision making) by the application. This experiment was executed 200 times, then the variance and standard deviation were calculated for the above criteria.

Adaptation time/re-configuration time. The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing the adaptation/re-configuration in two modes. In the first mood, the application was executed for the same period of time (five hours). Then, the adaptation time was measured once the adaptation action was started until finished, the CPU time was taken from the activity monitoring tool. In the second mood, the simulator was used to generate multiple context events, that measures the application response with regrade to low battery context. This experiment was executed 200 times, then the variance and standard deviation for each value were calculated. This allows us to identify the positive error as described in the following experiments' evaluation.

5.6 COSD Vs. AOSD Experiments

The assumption made by the AOSD communities is that dynamic aspect weaving can be used to adjust the software behaviour dynamically, regardless of the complexity involved in implementing Aspect-Oriented Programming (AOP) applications. Existing Dynamic AOP techniques tend to add a substantial overhead in both execution time and code size [Hundt et al., 2010].

The I-TrinityTour implementation was re-engineered to be integrated with the Objective-C AOP framework [AspectCOCA, 2011]. As a result, several aspects were implemented which implement context monitoring and detecting. In addition, the context-dependent behaviours for the location service, battery level, and the camera flashes were implemented. These anticipation levels were described in the previous chapter. However, for the location service, there are three nested aspects implemented to provide behavioural variation of the battery level. These aspects are the GPS-based, WiFi-based, and IP-based location services. In COSD, these aspects are implemented using three COCA-components, as demonstrated in Chapter 4.

5.6.1 Experiment 1: Context Monitoring and Sensing

This experiment evaluates the processes of context monitoring and environment sensing, based on the above criteria. Specifically, it evaluates how the software uses the allocated resources such as battery, CPU, and memory. In the DAOP approach, context monitoring is handled using separate aspects which span the application's main execution. In COSD, this is handled using the context manager, as explained in Chapter 3.

Designing aspects which become active when particular contexts are verified requires the possibility of referring to a context definition in a pointcut construction. This means that joinpoints such as BeInContext(Context BatteryLowCTX) should be provided by the framework. In addition, the aspects composition in a framework like Reflex [Tanter, 2006] needs to keep track of past context conditions and their associated states, which require more CPU activity and memory allocation to perform the context monitoring functionality. This adds so much overhead to the advice execution, because the AOP framework must perform context snapshots through the monitoring and sensing process. The problem behind this is the context snapshot is taken every time the context is changed. This makes the platform storing and processing the context history for multiple events at multiple times. In addition, the AOP framework must transform the context changes into basic entities like joinpoint request. The joinpoints are activated by registering them to the execution monitor. When the execution reaches one of the activated joinpoints, the execution monitor notifies the DAOP engine, which executes the advice method. This implies, that the AOP framework will evaluate each joinpoint with regard to the passive and active context through the context detection and decision-making processes as shown in the next experiment.

As a result, battery energy is consumed faster in comparison to the case using COCA-middleware, which implements a dedicated context manager supported by a context repository; the repository stores the past context information. In addition, each COCA-component registers its interest on a specific context change. This makes the context manager sense the environment for a particular set of context information.

The experiment on battery usage is shown in Figure 5.3. This shows that the COCA-component uses less battery energy than the DOAP implementation uses. The COCA-middleware optimizes the context-monitoring process by storing and processing the context information which was considered by the COCA-component registration. Such enhancement of the context monitoring preserves the battery energy by 11.5% of the total energy usage. This value is supported by the evaluation results of the CPU activity shown in Figure 5.4. For context monitoring, the DOAP-ITrinityTour requires more activity to be executed in comparison to COCA-ITrinity; this consumes more battery energy. With regard to memory allocation, the DOAP-ITrinity allocated more real memory to execute and perform the context snapshot i.e. storing and processing the past context than is needed by COCA-ITrinityTour, as shown in Figure 5.5. This figure shows how expensive it is to allocate and process the context snapshot with regard to the DOAP-ITrinity application.

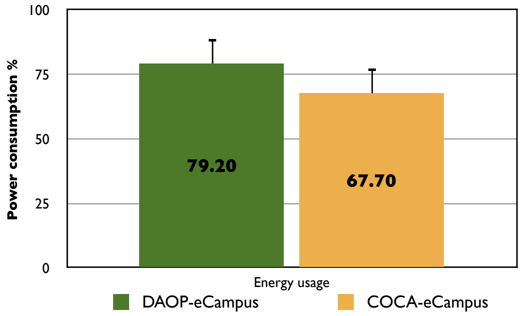


Fig. 5.3: Context Monitoring battery usage

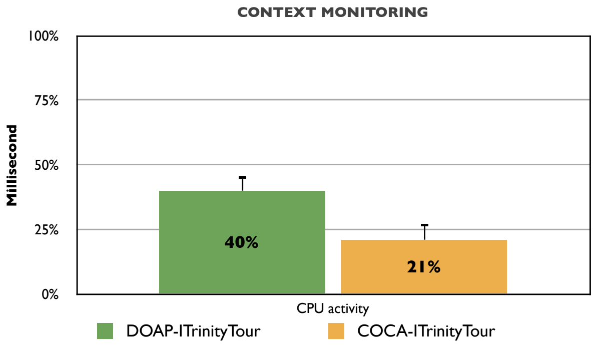


Fig. 5.4: Context Monitoring CPU Activity

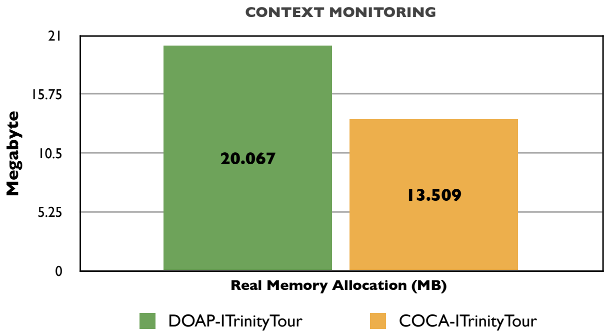


Fig. 5.5: Real Memory Allocation (MB)

5.6.2 Experiment 2: Context Detection

For the context detection process, both implementations were evaluated based on the above criteria. The evaluation results for energy usage are shown in Figure 5.6. The evaluation results show that DOAP-ITrinity consumes more energy to notify the application components about multiple context changes which were detected in short frequency. This requires more CPU activity to process the context changes and evaluate them with the passive context values stored in the joinpoints. The CPU activity for both applications is demonstrated in Figure 5.7. In addition, the DOAP application requires more memory for allocating the aspect contexts and notifying them because each aspect must be allocated and executed. The AOP framework then notifies the aspects about the context changes. Later, the decision is left to the aspect methods implementation to decide whether to adapt or not. Such implementation of the context detection process using DOAP intensively consumes the allocated resources to notify multiple aspects about multiple events. In some cases, the aspect implementation was independent of the execution context, but it was executed and notified. The real memory allocation for the context detection process is shown in Figure 5.8.

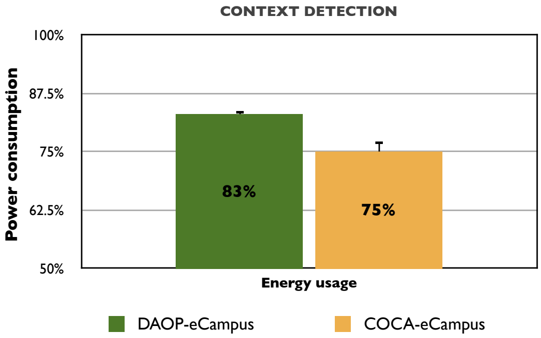


Fig. 5.6: Context Detection Battery Usage

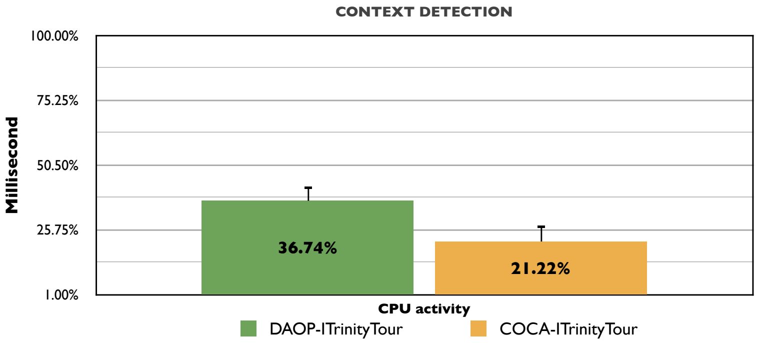


Fig. 5.7: Context Detection CPU Activity

5.6.3 Experiment 3: Collaborated Aspect Activation

It is claimed that in AOSD, dynamic aspect weaving can inject tangle-free code in the program execution; as explained before, context-dependent behaviours are collaborated aspects entangled with each other. It is claimed that in COSD, COCA-components can be acti-

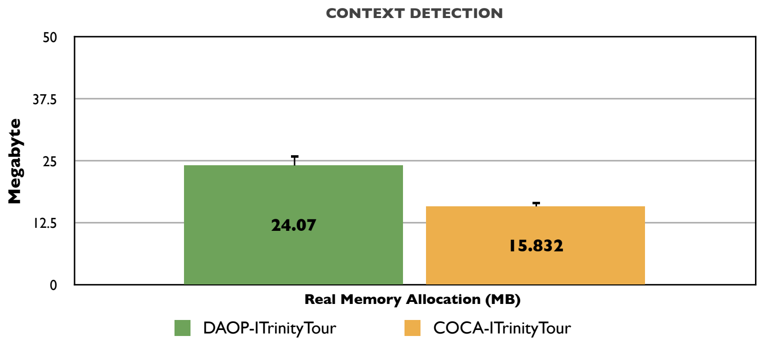


Fig. 5.8: Context Detection Real Memory Allocation (MB)

vated dynamically to adjust the application behaviour, with affordable costs, during the adaptation. Designing context-dependent behaviour using an aspect-oriented programming paradigm requires platform support for activating aspects driven by the context state; such an implementation requires the AOP platform to evaluate each joinpoint in conjunction with the associated context state and the passive context values. In addition, once the decision has been made, the AOP platform must search for the associated method implementation which implements the required context-dependent behaviour. Moreover, from our own experience, it is very complex to decide which aspect should be woven first, because of the implicit dependence among the aspect implementations. For example, the platform should decide when the battery level is low, and which aspects must be activated. On the other hand, when activating the location aspect, the platform must consider the battery level before deciding which location service to use; such processes provide cyclic dependence among the aspects implementations and lead to unguaranteed adaptation outputs.

Figure 5.9 shows the battery usage when multiple contextual aspects are activated and executed compared with the composition of multiple COCA-components. The figure shows that the DAOP-ITrinityTour consumes more energy to perform the adaptation as it requires more energy to process the context state in each joinpoint. In addition, it requires the AOP framework to resolve the dependence between several aspects before and after the advice methods execution. The CPU activity is shown in Figure 5.10 and the real memory allocation for performing the activation and execution is shown in Figure 5.11.

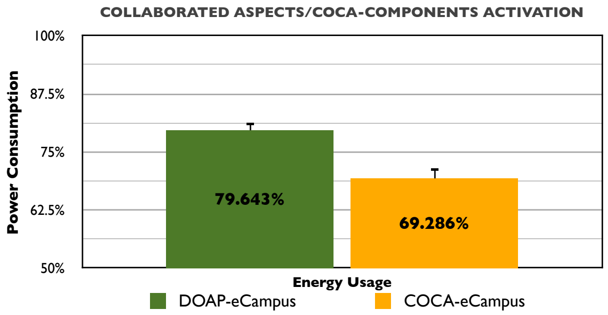


Fig. 5.9: Activating Collaborated Aspects/COCA-components Battery Usage

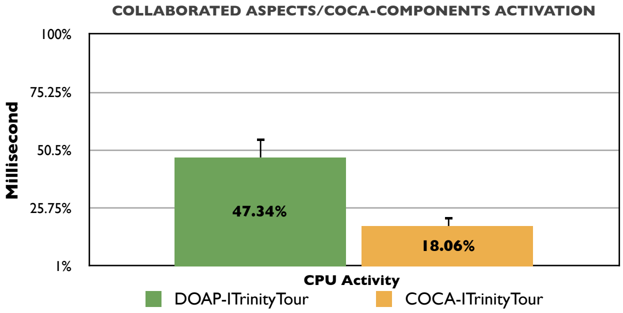


Fig. 5.10: Activating Collaborated Aspects/COCA-components CPU Activity

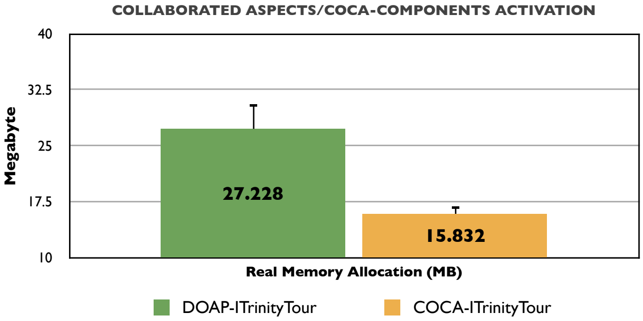


Fig. 5.11: Activating Collaborated Aspects/COCA-components Real Memory Allocation (MB)

The aspects composition needs to keep track of past context conditions and their associated states; more CPU activity and memory allocation are needed to perform this functionality. This experiment describes how each platform responds to multiple events detected at the same time. The adaptation/reconfiguration time for composing aspects/components is shown in Figure 5.12. The values were taken every 2 min from the Instruments tool while executing the application for 30 min continuously. As shown in Figure 5.12, the COCA-ITrinityTour requires less CPU time for composing the components, but DOAP requires more time for activating and executing the contextual aspects. The evaluation of aspects activation and execution shows an increased adaptation time because each aspect requires more memory allocation and CPU time to resolve the execution context with the context snapshot. On the other hand, the COCA-middleware requires more adaptation time for loading and executing the bundle implementation, but it can switch between weak/strong adaptation actions based on the execution context and the allocated resources. As shown in the figure, COCA-components composition requires less adaptation/reconfiguration, based on the adaptation mechanism. Such variations in the adaptation time provided by COCA-middleware can make use of the adaptation process and increase the device durability. The adaptation time in DAOP, as shown in the figure, may increase over the execution time, which leads to poor performance and lower efficiency.

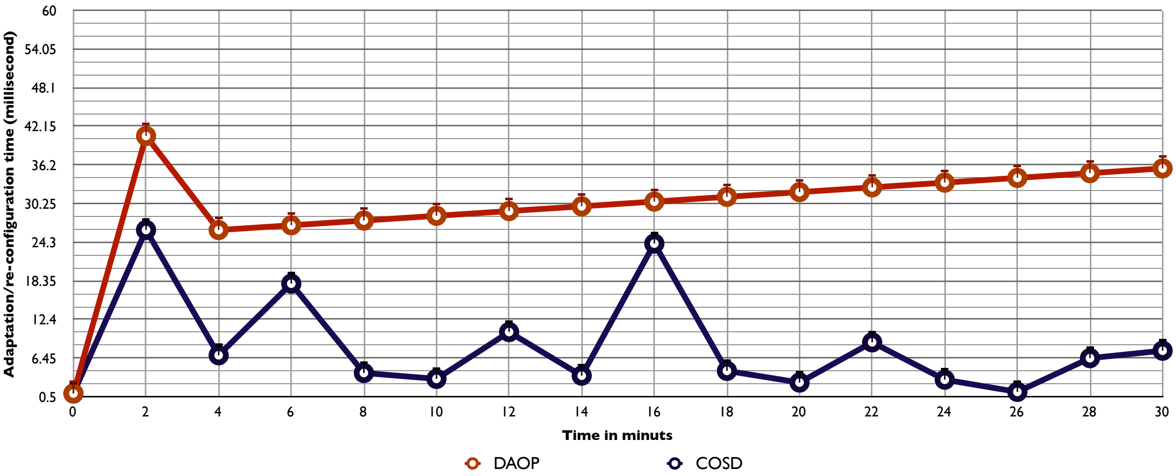
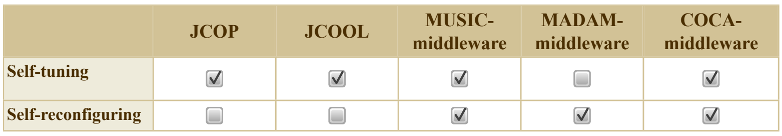


Fig. 5.12: Aspects/COCA-components Composition

5.7 COCA-middleware Evaluation

The case study was implemented with COCA-middleware and other approaches proposed in the literature. These approaches include the context-oriented programming paradigm targeting mobile devices, called JCOP [Appeltauer et al., 2008, Schuster et al., 2011], JCOOL, supported by CAMEL methodology, which used aspect-oriented programming and middleware for context-dependent behaviours de/activation [Sindico et al., 2008, Sindico and Grassi, 2009], MUSIC-middleware [Rouvoy et al., 2008a, Geihs et al., 2011], and MADAM-middleware [Mikalsen et al., 2006], which was fully implemented by Paspallis [Paspallis, 2009]. The implementation of the I-TrinityTour in JCOP followed the COP approach [Appeltauer et al., 2008, Schuster et al., 2011]. The implementation in JCOOL was accomplished with the aid of the aspect-oriented programming framework for Objective-C [AspectCOCA, 2011]. Both MUSIC-middleware and MADAM-middleware functionalities were implemented using the MUSIC development paradigm proposed by Rouvoy et al. [Rouvoy et al., 2008a].



	JCOP	JCOOL	MUSIC-middleware	MADAM-middleware	COCA-middleware
Self-tuning	☑	☑	☑	□	☑
Self-reconfiguring	□	□	☑	☑	☑

Fig. 5.13: Autonomic Properties Support

The objective of this experiment is to evaluate the proposed solutions in supporting the autonomic properties, self-tuning and self-configuring, of the self-adaptive I-TrinityTour application. The support of these properties in the above-mentioned solutions can be summarized as shown in Table 5.13. JCOP and JCOOL support only fine-grained adaptations using ad-hoc programming-level techniques; this is not able to change the application structure. MUSIC-middleware supports both autonomic properties using parametric tuning and plug-in architecture. The plug-in implementation used for adding or removing services/components. MADAM-middleware supports only self-configuring as it models a separate plug-in architecture for each context provider. Table 5.13 summarizes the adaptability support in each platform.

5.7.1 Experiment 4: Self-tuning Evaluation

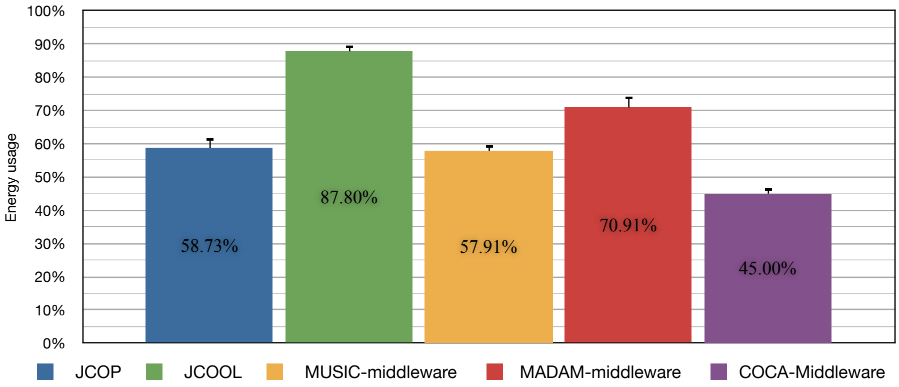


Fig. 5.14: Energy Usage for I-TrinityTour Application

Figure 5.14 shows the experimental results for energy usage analysis for the I-TrinityTour running on the five platforms: JCOP, JCOOL, MUSIC-middleware, MADAM-middleware, and COCA-middleware. The experiment shows that the COCA implementation of the I-TrinityTour application used 15% less battery energy than the MUSIC implementation used. The I-TrinityTour implementation with JCOOL consumes more energy during the adaptation processes because it does not consider the battery level or status during the adaptation action. In JCOOL, the context values are only considered for evaluating the context-dependence in each joinpoint implementation. In the same way, the MADAM-middleware drained the battery faster because each location service was implemented in a distinct plug-in (bundle) architecture, which requires more processing time for loading the bundle implementation. In contrast, when the same application was adapted by the COCA-middleware, the application was able to adapt its behaviour and use less energy because the COCA-middleware adapts to the location service by redirecting the delegate object to activate the required service implementation. In JCOP, the application was able to adjust its behaviour and adapt the required location service; unfortunately, the context monitoring was being relied on the mobile device operating system to deliver and detect the context information; this leads to faster consumption of the battery resource as a result of trying to process too many context events at the same time. The MUSIC-middleware performs better with respect to battery consumption because MUSIC-middleware uses a fine-tuning mechanism for manipulating components implementation. However, the application implemented using MUSIC-middleware consumes more energy than that using COCA-middleware because MUSIC-middleware calculates the fitness of the application variant using a utility function every time the context state changes. Such verification at runtime requires more CPU time and memory allocation, which, as a result, consumes more energy.

Figure 5.15 shows the experimental results for CPU activities analysed for the I-TrinityTour application in the five platforms previously mentioned. The evaluation considered adaptation processes including context monitoring, detecting, decision-making, and adaptation. As shown in Figure 5.15, context monitoring requires much more CPU activity in the JCOP and JCOOL platforms as they have no dedicated context-monitoring process, and they rely on the infrastructure to deliver the context information. The MUSIC and MADAM architectures come second with regard to context monitoring as they both implement a dedicated context manager which is able to process and filter the context information. Unfortunately, JCOP, JCOOL, MUSIC, and MADAM do not consider the effect of a contentious and unbalanced monitoring process for the context environment. This implies notifying the application several times about multiple context events, which requires more of the CPU time to process and handle these events. On the other hand, adapting the observer pattern allows the context manager in the COCA-middleware to notify the interested components about the context changes when needed, so that the context-monitoring time drops from 57 ms in MUSIC-middleware to 33 ms in COCA-middleware. For the same reason, the context detection process drops from 72 ms to 46 ms in COCA-middleware.

The decision-making in JCOP and JCOOL require less CPU activity as both platforms use static decision-making. The JCOP and JCOOL approaches assume that the developers can predict when and where the context-dependent behaviour is needed in the application source code. For the same reason, the MADAM-middleware requires less time for decision-making as it uses predefined rules supported by a rule engine to control the adaptation action. In the MUSIC-middleware, the decision-making process is performed at runtime with the aid of a utility function; this requires more computation activities for analysing the architecture's constraints, adaptation goals, predefined rules, and the quality of services, plus the user's preferences. Afterwards, the application's variations model (adaptation plan) is selected based on the utility function results. In COCA-middleware, the decision-making process is performed at runtime with the aid of the policy and verification manager, which both consider the decision policies in conjunction with the available resources and the quality-of-services (QOS). In general, the dynamic decision-making process requires more effort from the CPU than the static approach does, as shown in Figure 5.15.

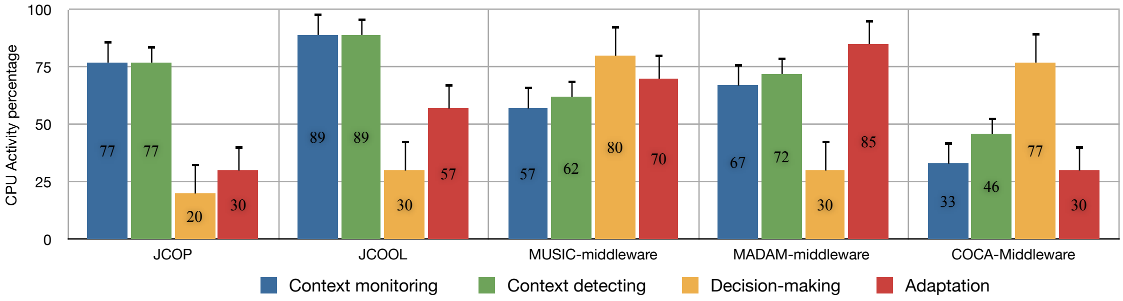


Fig. 5.15: Energy Usage for I-TrinityTour Application

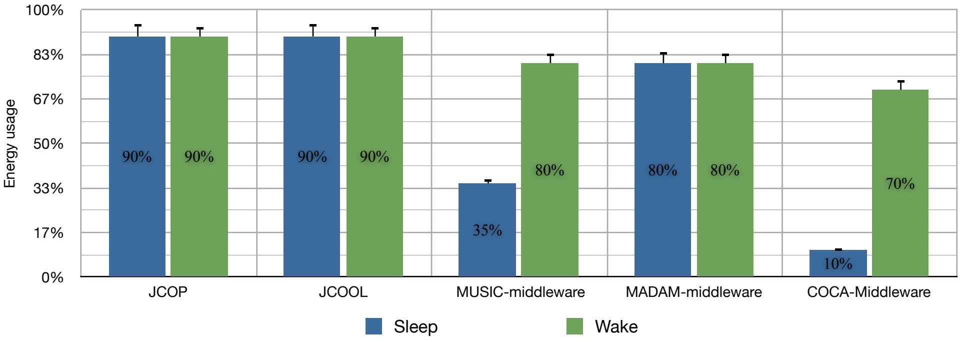


Fig. 5.16: Using the Allocated Resources in Sleep/Wake Mode

Figure 5.16 shows how the five platforms use the allocated resources while the device is in Sleep/Wake mode. Such an evaluation reflects the middleware's ability to adjust its own functionality as long as the application is running in the background. At the same time, it shows how the COCA-middleware wakens the application to notify the user about events in the monitored region. The other approaches have not considered the trade-off between the adaptation process and the allocated resources. The result of this is that the I-TrinityTour implementations on JCOP, JCOOL, and MADAM were consuming battery life even when the application was in sleep mode or running in the background. The applications keep updating the current location of the device. Such an action was unnecessary, as the device was in sleep mode, so they consumed 90% of the battery life after executing the application for 5 h. With the COCA and MUSIC-middleware, the application was executed for the same period of time, and consumed less battery energy.

5.7.2 Experiment 5: Self-reconfiguring Evaluation

As mentioned before, self-reconfiguring is the capability of the software to adapt and behave autonomously in response to context changes. To evaluate this attribute, the I-TrinityTour application has been implemented in three distinct versions, using the MUSIC-middleware, MADAM-middleware, and COCA-middleware. The JCOP and JCOOL platforms were excluded from this experiment as they did not support architecture reconfiguration.

The adaptation/configuration time for adapting the anticipation scenarios were mentioned before. The scenarios include adding a suitable location service according to the battery level, adding a sorting component, and adapting to a video-streaming service. Figure 5.17 shows the evaluation results for the three architectures. The MADAM-middleware requires more time to perform the adaptation because three bundles are loaded and executed. The total adaptation time was 210 ms. The MUSIC-middleware required less time for reconfiguring the software as it adapted the location service using parametric tuning rather than loading a complete bundle for it. The COCA-middleware comes third in the analysis as it can switch autonomously between several location services by de/activating the associated layers. In addition, it verifies whether the plug-in can provide the necessary services before physically executing its implementation. In addition to this, according to the context state, the COCA-middleware performs a runtime composition of the software components so that only the needed components are executed. In MUSIC, the adaptation takes longer as it evaluates multiple application variations, then one variant is selected and executed. Loading a precompiled code from the component repository after instantiation is accomplished in a shorter time than it takes to load the whole bundle implementation at once in the COCA-middleware. This is illustrated in Figure 5.18, which shows the memory allocations for the three platforms. 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 the bundle implementation in the MUSIC- and MADAM-middlewares.

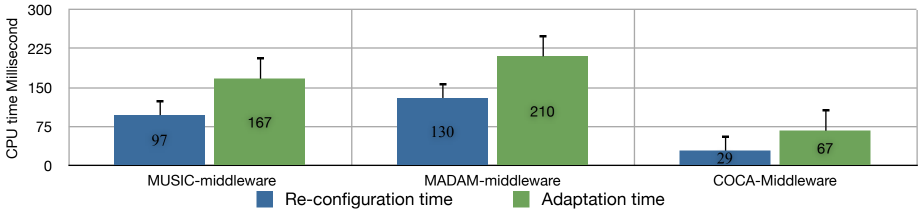


Fig. 5.17: Adaptation/Reconfiguration Time (ms)

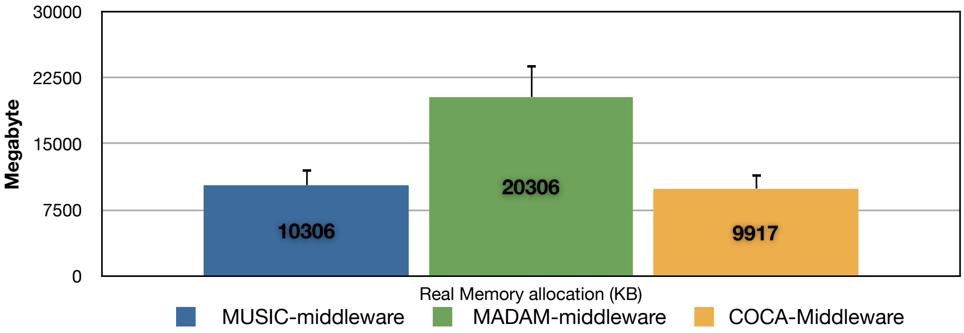


Fig. 5.18: Memory Allocation for I-TrinityTour

5.8 Summary

Any architecture analysis method relies on the active and willing participation of the stakeholders, particularly the architecture team, advance preparation by the key stakeholders, an understanding of architectural design issues and analytic models, a clearly articulated set of quality attribute requirements, and a set of business goals from which they are derived. ATAM is appropriate for medium-sized systems where stakeholders can reach a consensus and the architect can carefully and qualitatively compare each of the scenarios identified. Architecture quality attributes can be related to the self-adaptive system properties. Combining the self-* properties with quality attributes helps the designers to enhanced self-adaptive software engineering.

Evaluating the COCA-middleware through ATAM methods verifies its ability to evaluate its functionality (A1.2 of objective 1), constructing the composition plan (A2.1 of objective 2), perform introspection and intercession of the adaptation action (A2.2 of objective 2), resolve conflict among several decision policies (A2.3 of objective 2), achieving fine-grained and Coarse-grained adaptation (A3.1 and A3.2 of objective 3), and anticipate unforeseen changes (A4.1 and A4.2 of objective 4. In practice, performance and modifiability trade-off with each other as showed in the middleware evaluation. The COCA-middleware achieves self-tuning and self-configuring without degrading the allocated resources. With regards to the adaptation processes including context monitoring, detecting, decision-making and adaptation, COCA-middleware shows better performance compared to other approaches proposed in the literature. Thus, the evaluation of COCA-middleware shows its ability in achieving (A5.1 and A5.2) of objective O5.

The evaluation of the COSD paradigm in comparison to AOSD shows that COSD is better suited to implementing context-dependent and self-adaptive applications. The performance and energy usage in COCA-applications are better than in DAOP-applications. There is no doubt that Aspect-oriented frameworks can be used for developing and implementing self-adaptive applications, but their performance is very poor in comparison to that of COCA-middleware. The COCA-middleware implementation performs better with regard to adaptation processes, including context monitoring, detecting, decision-making, and adaptation. The evaluation results shows that implementing self-adaptive applications with the aid of COCA-middleware can support software adaptability and variability with affordable adaptation costs and less impact on the allocated resources. Programming-level approaches like JCOP and JCOOL tend to support self-tuning of software systems with an acceptable level of performance, but the overall support of adaptability and variability is very limited in comparison with architecture evolution approaches such as MUSIC, MADAM, and COCA-middleware. However, the programming techniques are better suited to small-scale context-dependent applications, and they require extensive modification for supporting context monitoring, context detection, and dynamic decision-making.

Chapter 6

Development Methodology Evaluation

Anticipating context changes using a model-based approach requires a formal procedure for analysing and modelling context-dependent functionalities, and a stable description of the architecture which supports dynamic decision-making and architecture evolution. This chapter demonstrates the capabilities of COCA-MDA in supporting the development of context-aware applications by describing a state-of-the-art case study and evaluating the development effort involved in adapting the COCA-MDA in constructing the application. The benefits gained from using techniques which separate component-based aspects in the modelling methodology and the productivity of COCA-MDA are the core of this chapter.

The evaluation objectives are outlined in Section 6.1. The design of the case study, based on COCA-MDA, is demonstrated in Section 6.2. Section 6.3 describes the implementation and evaluation of the case study application. In Section 6.4, the COCA-MDA is evaluated using Constructive Cost Model II (COCOMO II). The lessons learned from the methodology evaluation are illustrated in Section 6.5.

6.1 Evaluation Objectives

The objective of this chapter is to demonstrate the fitness of COCA-MDA in modelling self-adaptive applications. In general, there are two groups of requirements: first, the MDA-related criteria, which we believe are necessary to increase the cost-effectiveness of developing context-aware applications, and second, the capability to model the application at several anticipation levels using a clear sepration of concerns and producing a component-based model. The evaluation objectives can be summarized as follows.

- **O1:** The capability of the methodology to support a technique for clear separation of concerns through the analysis phase:

  A1: Ensure COCA-MDA is able to support a clear separation between the application functional and extra-functional concerns.

  A2: Separate the context model from the business logic model.

  A3: Separate the context-dependent requirements from the context-independent requirements.

  A4: Enable the developers to identify the extra-functional requirements which are managed by the middleware for behavioural variations and adaptation.

- **O2:** The capability of the methodology to generate a component-based model which modularizes the context-dependent functionality. The methodology can decompose the application into several architectural units to allow developers to decide which part of the architecture should be notified when a specific context condition occurs. The middleware is aware of which parts of the architecture components are affected by the changes.

  A1: Model and modularize actor-dependent, system-dependent, and environment-dependent behaviour variations into architectural components.

  A2: Produce a portable architecture which can be deployed on several platforms without the need for intensive configuration. Ensure ease of deployment on several platforms;

this includes evaluating the effort required to write the configuration code for the new platform compared to that required for others.

- **O3:** Enable the developers to specify a stable description of software models and proprieties. This enables the developers to specify a set of decision policies which describe the architecture evolution for specific changes.

  A1: Enable the developer to embed decision points in the application modules, based on when the adaptation is needed and on which part the new behaviour is extended to. Determine which changes the architecture responds to, thereby extending the application behaviour.

- **O5:** Reduce the development effort and the configuration time.

  - A5.1: Reduce the configuration time, which implies the addition of fewer source lines of code.
  - A5.2: Reduce the effort applied [person-months].
  - A5.3: Reduce the development time [person-months].
  - A5.4: Reduce the number of people required.
  - A5.5: Sizing the software maintenance ratio for the COCA-ADL transformation into the platform-specific model (PSM).

6.2 Self-adaptive Indoor Wayfinding Application for Individuals with Cognitive Impairments

Some of the challenges for individuals with cognitive impairments in wayfinding are remaining oriented, recalling routines, and travelling in unfamiliar areas while relying on limited cognitive capacity. Whereas people without disabilities often use maps or written directions, either as navigation tools or for remaining oriented, the cognitively impaired population is very sensitive to issues of abstraction (e.g. icons on maps or signage), which presents the application designer with the challenge of tailoring navigational information to each specific user and context. With the capacity to move and the desire to be socially included, mentally/cognitively disabled individuals who are independently mobile but have difficulties reaching their intended destination might benefit from the self-adaptive IWayFinder application proposed in this study.

The IWayFinder provides distributed cognition support for indoor navigation to persons with cognitive disabilities. Radio Frequency IDentification (RFID) tags and Quick Response Codes (QR-codes) are placed at decision points such as hallway intersections, exits, elevators, and entrances to stairways. After reading the encoded URL in the QR-codes, the Cisco mobility engine provides the required navigation information and instructs the user [Cisco context-ware software, 2011]. The Cisco mobility infrastructure has the ability to capture and employ contextual information about mobile assets. Contextual information can be collected automatically using the Wi-Fi connectivity of the asset (for example, laptops or Wi-Fi phones) or, for assets that do not have intrinsic wireless, by attaching radio frequency tags or QR-codes to the asset. QR-codes are a specific matrix barcode or two-dimensional code that is readable by dedicated QR barcode readers and camera phones [Parikh, 2005]. The benefit of integrating the application with the Cisco engine is the integration of several assets that provide the contextual information. QR-codes or RFID tags are placed at the DPs such as hallway intersections, exits, doors, elevators, or entrances to stairway identified by the Cisco engine.

A user enters the building and points the mobile phone's camera at any of the QR-codes available at the DP. After reading the encoded URL in the QR-codes, the Cisco engine then provides the required navigation information and instructs the user. To overcome the challenges of image rendering, the proposed self-adaptive application uses an augmented reality browser (ARB) to display the navigation directions. The browser displays the directions on the physical display of the tool's camera. Using the device camera, the system reduces the cognitive load and increases the user's ability to realize the desired route. In addition, the application is able to provide the user with time-based events such as the opening hours of the building, lunch time, closing hours of the offices, location access rights that control the entrance of users to certain locations, and any real time alarm events. Moreover, the infrastructure support allows several persons to monitor and collaborate with the user en route. Assuming the context information is delivered by the Cisco infrastructure, the following anticipation scenarios are proposed:

A1: Self-tuning The application must track the user's path inside the building. When DPs are reached, the application places a marker for each DP the user passed. If the user is unable to locate a decision point in the building, the application must be able to guide the user towards a safe exit. The route directions can be delivered to the user in several output formats: video, still images, and voice commands. The application should change the direction output while also adapting to the device resources and the level of cognitive impairment of the individual.

A2: Self-recovering Assuming that the user is trapped in a lift with no GPRS connection (or in the case of a fire), the fire alarm is raised, the application is notified, and the application adapts the shortest path to the nearest fire exit. In both cases, the application submits the user's current coordinates and an emergency help message to the emergency number, parents, career team, and security staff. The communication is achieved using the available connection, regardless of the resource cost, to alert any nearby devices to the emergent need for help. If no connection is made, the device emits an alarm sound and increases the device volume to maximum. The security staff or fire fighters receive the emergency message and can view the Closed-Circuit TeleVision (CCTV) video to identify the floor on which the user is trapped. When the CCTV system locates the user, full information about the user is displayed, including a personal and health profile. At the same time, the application guides the user to a safe exit using a preloaded path (in case the CCTV camera is disabled and the services engine is off). Fire fighters can use the received message to locate the user within the building.

6.2.1 Modelling the I-WayFinder Application with COCA-MDA

The COCA-MDA is used to decompose the application based on the context-driven behaviour. This case study has several anticipation levels, which must be realized by the COCA-middleware at runtime. The COCA-MDA was described in detail in Chapter 3. This section focusses on highlighting the modelling phase of COCA-MDA. Specifically, it provides a full description of the behavioural decomposition among the extra-functionalities. These extra-functionalities implement the desired behaviour based on their anticipation level.

The first model based on COCA-MDA is shown in the requirements diagram. This diagram is used to classify the requirements based on their type and the anticipation level. For example, Figure 6.1 shows several requirements that were derived by the case study scenarios. The requirements classified into functional, extra-functional, and technological. An example of the functional requirement is shown in the figure with requirements number 5 that provides time-based events to the user. Other examples of functional requirements are ’user current location’, ’display direction’, and ’user facing abstraction’. The extra functional requirements are classified by the tagged value kind in the requirement diagram. For example, ’adapt the direction output’ is an extra functionality because it is driven by the availability of the device resources. In the same way, ’alternative route’, ’user is trapped’, and ’send an emergency message’ are classified as extra functional requirements, as shown in the figure.

The requirements are combined into a use case diagram. The use cases are classified into two major classes. The first class comprises application use cases. The second class refers to any use case that is extended by a contextual condition. Figure 6.2 shows the use case diagram for this case study. The use case ’scan RFID’ is a use case describing a core functionality delivered by the application. The fire alarm use case is a contextually-driven use case that extends the application functionality to send an emergency message and provides a route to the nearest fire exit. The use cases coloured blue refer to contextual use cases that describe context providers. Adapting the user location in the ’symmetric places’ use case requires extending the application behaviour to calculate an alternative route. In this diagram, the parents, career team, security staff, and fire-fighters are classified as actors who are alerted

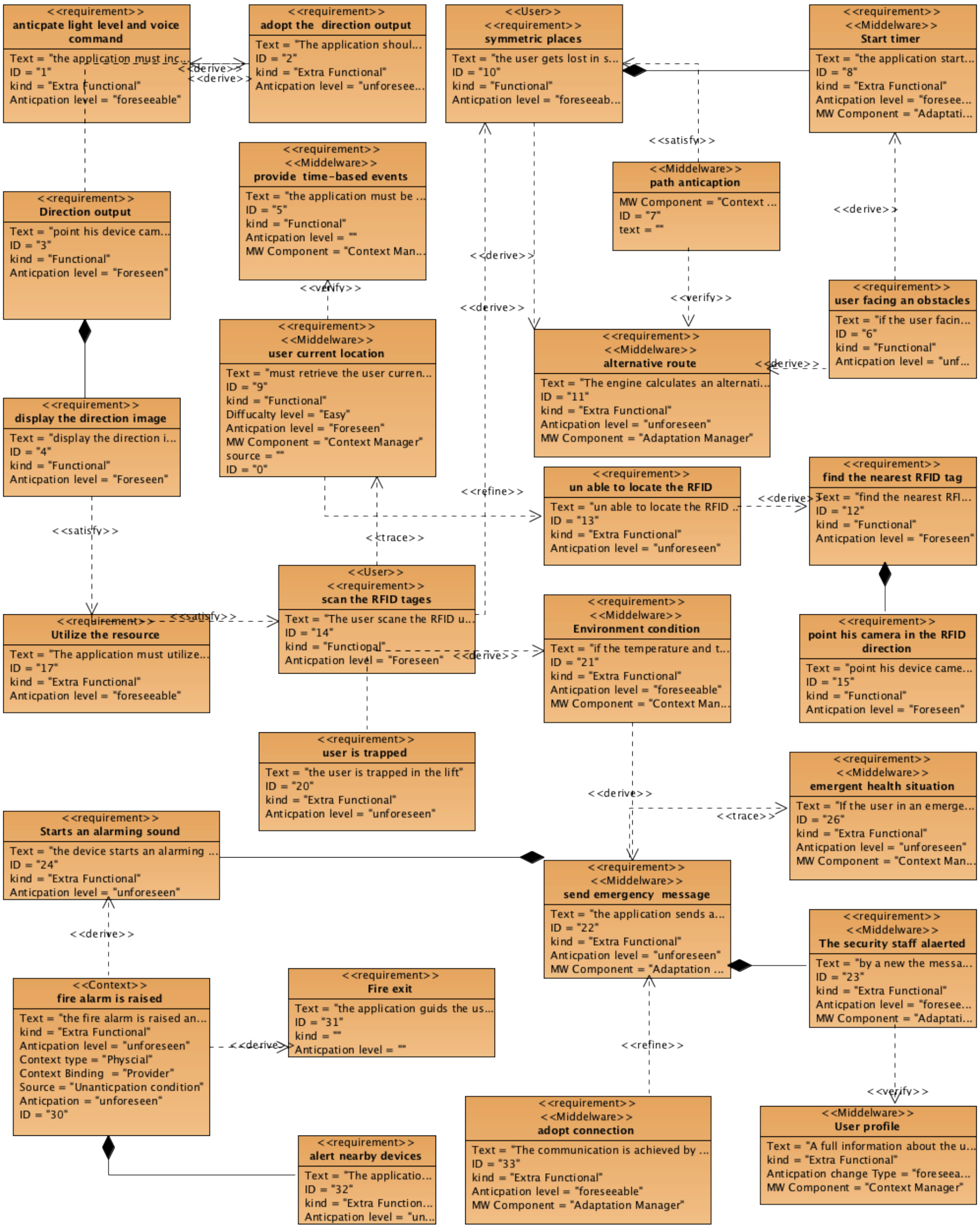

<<requirement>>
anticpate light level and voice command
Text = "the application must inc...
ID = "1"
kind = "Extra Functional"
Anticpation level = "foreseeable"
<<derive>>
<<derive>>
<<requirement>>
adopt the direction output
Text = "The application shoul...
ID = "2"
kind = "Extra Functional"
Anticpation level = "unforesee...
<<User>>
<<requirement>>
symmetric places
Text = "the user gets lost in s...
ID = "10"
kind = "Functional"
Anticpation level = "foreseeab...
<<requirement>>
<<Middleware>>
Start timer
Text = "the application start...
ID = "8"
kind = "Extra Functional"
Anticpation level = "foresee...
MW Component = "Adaptati...
<<satisfy>>
<<requirement>>
<<Middelware>>
provide time-based events
Text = "the application must be ...
ID = "5"
kind = "Functional"
Anticpation level = ""
MW Component = "Context Man...
<<Middelware>>
path anticpation
MW Component = "Context ...
ID = "7"
text = ""
<<derive>>
<<requirement>>
Direction output
Text = "point his device cam...
ID = "3"
kind = "Functional"
Anticpation level = "Foreseen"
<<derive>>
<<verify>>
<<derive>>
<<verify>>
<<requirement>>
user facing an obstacles
Text = "if the user facin...
ID = "6"
kind = "Functional"
Anticpation level = "unf...
<<requirement>>
<<Middelware>>
user current location
Text = "must retrieve the user curren...
ID = "9"
kind = "Functional"
Diffucalty level = "Easy"
Anticpation level = "Foreseen"
MW Component = "Context Manager"
source = ""
ID = "0"
<<requirement>>
<<Middelware>>
alternative route
Text = "The engine calculates an alternati...
ID = "11"
kind = "Extra Functional"
Anticpation level = "unforeseen"
MW Component = "Adaptation Manager"
<<derive>>
<<requirement>>
display the direction image
Text = "display the direction i...
ID = "4"
kind = "Functional"
Anticpation level = "Foreseen"
<<refine>>
<<requirement>>
un able to locate the RFID
Text = "un able to locate the RFID ...
ID = "13"
kind = "Extra Functional"
Anticpation level = "unforeseen"
<<derive>>
<<requirement>>
find the nearest RFID tag
Text = "find the nearest RFI...
ID = "12"
kind = "Functional"
Anticpation level = "Foreseen"
<<satisfy>>
<<trace>>
<<User>>
<<requirement>>
scan the RFID tages
Text = "The user scane the RFID u...
ID = "14"
kind = "Functional"
Anticpation level = "Foreseen"
<<satisfy>>
<<requirement>>
Utilize the resource
Text = "The application must utilize...
ID = "17"
kind = "Extra Functional"
Anticpation level = "foreseeable"
<<derive>>
<<requirement>>
<<Middelware>>
Environment condition
Text = "if the temperature and t...
ID = "21"
kind = "Extra Functional"
Anticpation level = "foreseeable"
MW Component = "Context Man...
<<requirement>>
point his camera in the RFID direction
Text = "point his device came...
ID = "15"
kind = "Functional"
Anticpation level = "Foreseen"
<<requirement>>
user is trapped
Text = "the user is trapped in the lift"
ID = "20"
kind = "Extra Functional"
Anticpation level = "unforeseen"
<<derive>>
<<trace>>
<<requirement>>
<<Middelware>>
emergent health situation
Text = "If the user in an emerge...
ID = "26"
kind = "Extra Functional"
Anticpation level = "unforeseen"
MW Component = "Context Man...
<<requirement>>
Starts an alarming sound
Text = "the device starts an alarming ...
ID = "24"
kind = "Extra Functional"
Anticpation level = "unforeseen"
<<requirement>>
<<Middelware>>
send emergency message
Text = "the application sends a...
ID = "22"
kind = "Extra Functional"
Anticpation level = "unforeseen"
MW Component = "Adaptation ...
<<requirement>>
<<Middelware>>
The security staff alaerted
Text = "by a new the messa...
ID = "23"
kind = "Extra Functional"
Anticpation level = "foresee...
MW Component = "Adaptati...
<<derive>>
<<Context>>
fire alarm is raised
Text = "the fire alarm is raised an...
kind = "Extra Functional"
Anticpation level = "unforeseen"
Context type = "Physcial"
Context Binding = "Provider"
Source = "Unanticpation condition"
Anticpation = "unforeseen"
ID = "30"
<<requirement>>
Fire exit
Text = "the application guids the us...
ID = "31"
kind = ""
Anticpation level = ""
<<derive>>
<<refine>>
<<verify>>
<<requirement>>
<<Middelware>>
adopt connection
Text = "The communication is achieved by ...
ID = "33"
kind = "Extra Functional"
Anticpation level = "foreseeable"
MW Component = "Adaptation Manager"
<<Middelware>>
User profile
Text = "A full information about the u...
kind = "Extra Functional"
Anticpation change Type = "foreseea...
MW Component = "Context Manager"
<<requirement>>
alert nearby devices
Text = "The applicatio...
ID = "32"
kind = "Extra Function...
Anticpation level = "un...

Fig. 6.1: Case study requirements diagram

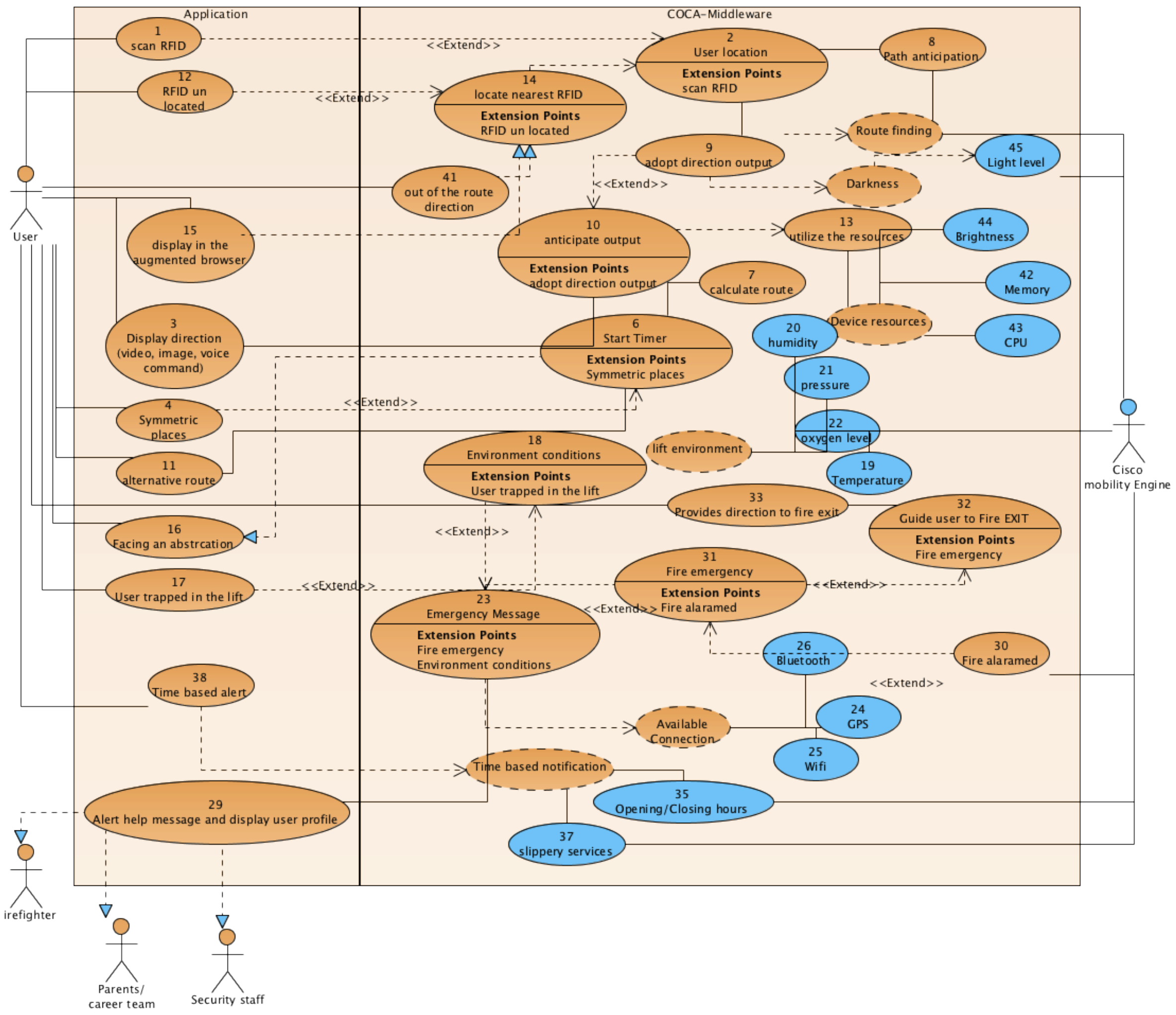


Fig. 6.2: Use case diagram

by the application whenever the context-driven behaviours are adapted. For example, the fire-fighters are notified only when there is a fire in the building. On the other hand, the parents or career team and the building security staff are notified with an emergency message generated by the application.

The use case is split into two distinct class diagrams. The first diagram describes the basic application components that are executed regardless of the execution context. The base components' class diagram is shown in Figure 6.3. This figure describes the relation

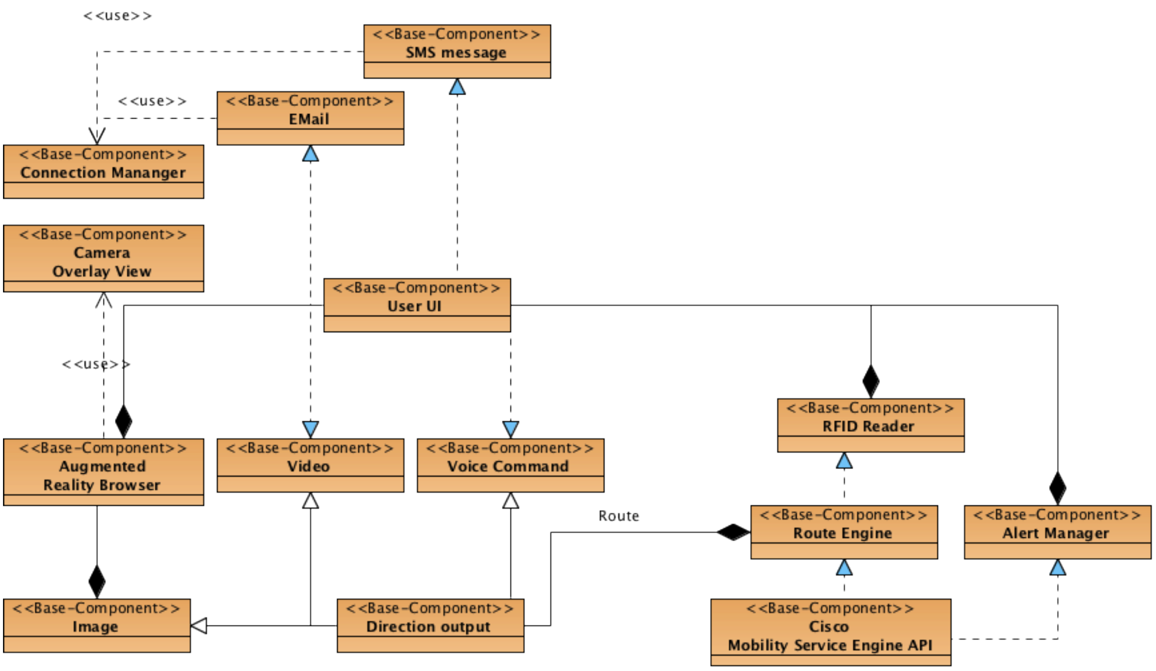

<<use>>
<<Base-Component>> SMS message
<<use>>
<<Base-Component>> EMail
<<Base-Component>> Connection Mananger
<<Base-Component>> Camera Overlay View
<<Base-Component>> User UI
<<use>>
<<Base-Component>> RFID Reader
<<Base-Component>> Augmented Reality Browser
<<Base-Component>> Video
<<Base-Component>> Voice Command
Route
<<Base-Component>> Route Engine
<<Base-Component>> Alert Manager
<<Base-Component>> Image
<<Base-Component>> Direction output
<<Base-Component>> Cisco Mobility Service Engine API

Fig. 6.3: Core structure object diagram

between the application classes and the Cisco engine. For example, the UI is providing the user prompts to scan the RFID tags, displaying the direction outputs, and displaying the direction in the augmented reality browser. In addition, it integrates the application and the Cisco mobility services engine API. This core structure is integrated with the extra-functional class model in the final architecture model.

The extra-functionality class diagram in Figure 6.4 provides a detailed view of the application COCA-component and the COCA-middleware. In addition, this diagram models the desired behavioural layer that can be used to anticipate context changes. The COCA-middleware notification queue is used to notify the COCA-components of context changes. The diagram has four major COCA-components. Each component includes several layers that are executed whenever their implemented behaviour is needed. The COCA-component 'user trapped' implements several selectors executed based on the context state, for example, when the temperature level in the lift exceeds a specific value defined by the decision policy; in this

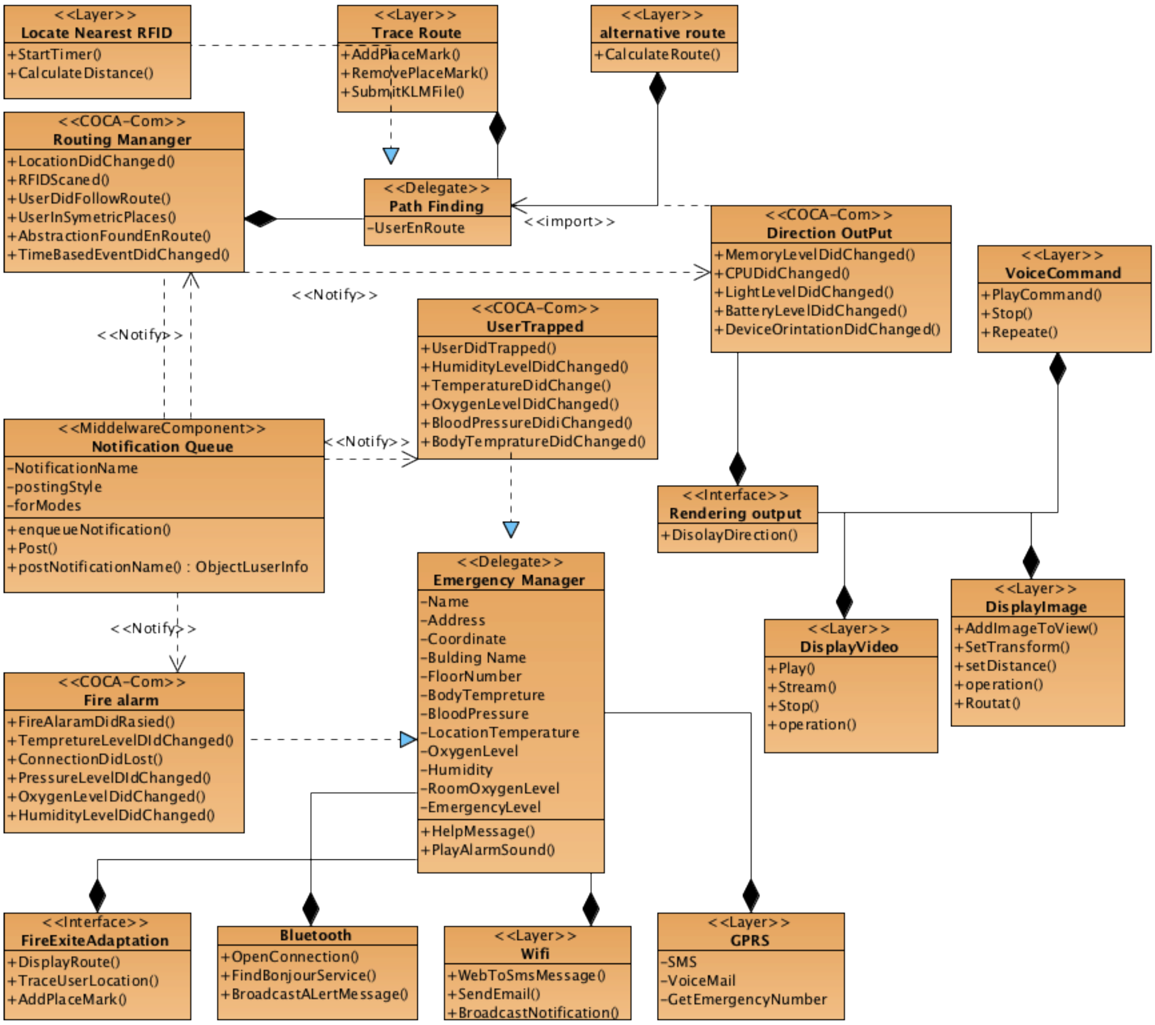


Fig. 6.4: Behavioural Model

case, the selector method in the 'user trapped' COCA-component is executed. However, the method's execution is controlled by the adaptation manager after evaluating the attached DPLs. The other COCA-components are: 'routing manager', 'direction output', and 'fire alarm'. The COCA-component 'fire alarm' realizes the same delegation object, which is the 'emergency message' shown in the Figure. Furthermore, the 'emergency message' includes five distinct layers; each layer implements a specific behaviour. For example, the sub layer

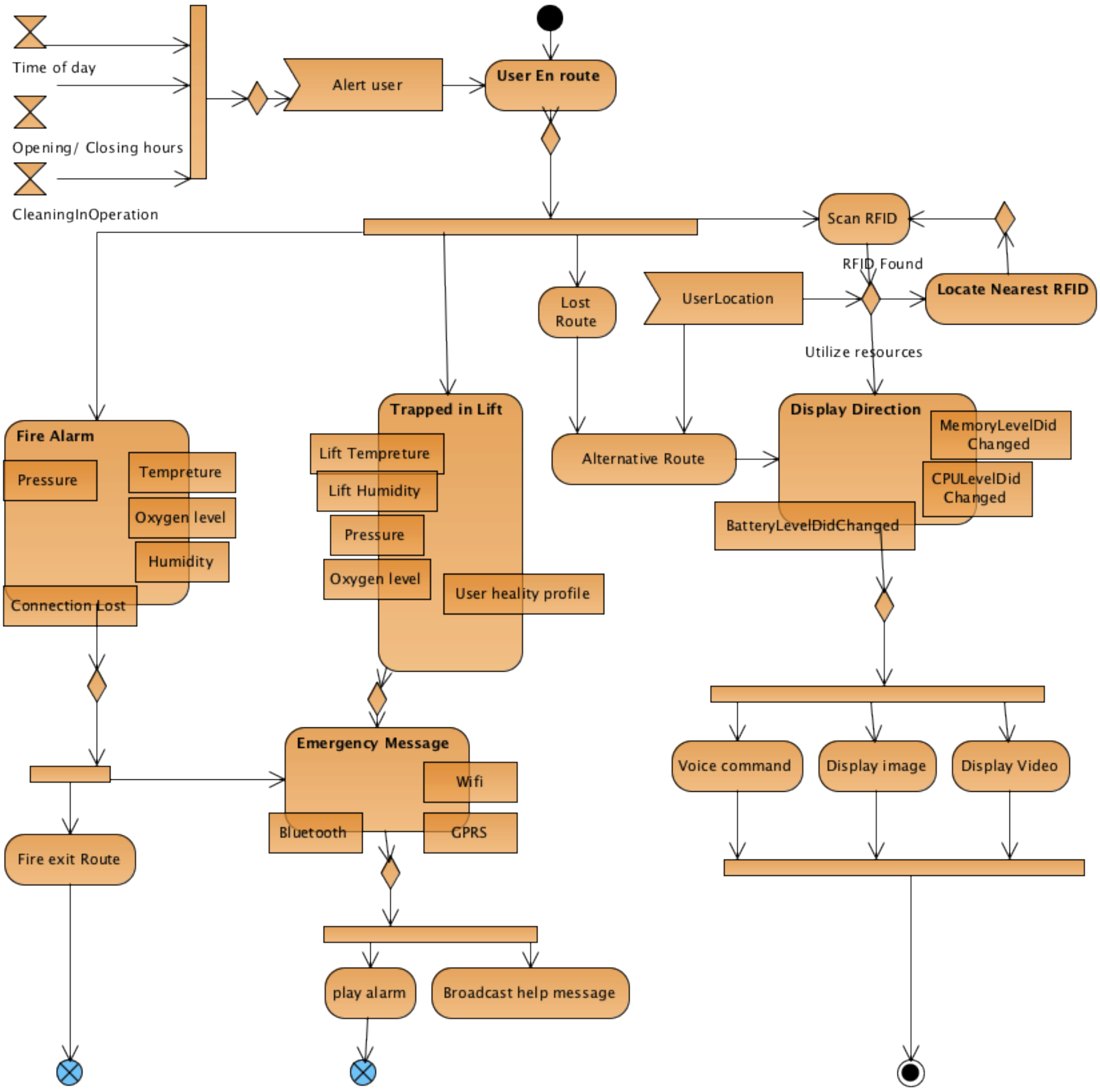


Fig. 6.5: Activity diagram

'fire exit adaptation' is activated in the execution whenever a fire alarm is raised; the application must provide the nearest fire exit to the user. At the same time, the help message is sent after adapting the available connection, regardless of the cost.

The application behavioural model is shown in Figure 6.5. This activity diagram is used to demonstrate the decision points in the execution that might be reached whenever internal or external variables are found. However, the main activities of the application are scanning the RFID, obtaining the route direction from the engine, and displaying the direction image in the augmented reality browser. This describes the basic behaviour of the application. In some cases, the user is trapped in the lift, this abnormal condition requires that the application perform a decision in this situation. This decision point requires several parameter inputs to make the correct choice at this critical time. The activity parameters shown in the figure identify the building temperature, oxygen level, humidity, and pressure. At the same time, this activity is required to adapt to the available condition. Additionally, the connection status must be considered in the decision. From the activity diagram, the developers extract several DPLs that are attached to the correspondent COCA-components.

From the activity diagram, the developers can extract the following DPLs. Each policy must be modelled in a state diagram:

- **Policy 1:** This policy is attached to the 'routing manager' COCA-component in Figure 6.4. The policy syntax can be explained as the code shown in Listing 6.1.

Listing 6.1: Decision policy1

```
If (locationDidChanged && RFIDisScanned )
then {GetRouteDirections(); TraceUserRoute();}
else if (!(RFIDScaned ) && !(UserFollowRoute))
then {LocateNearestRFID(); AlternativeRoute();}
else If (AbstractionDidFound)
then {AlternativeRoute(); StartTimer();TraceUserRoute(); }
else If (userInSymetricPlaces)
then AlternativeRoute();
```

- **Policy 2:** This policy is attached to the 'direction output' COCA-component in Figure 6.4. The policy syntax can be described by the code shown in Listing 6.2.

- **Policy 3:** This policy is attached to the 'user trapped' COCA-component in Figure 6.4. The policy syntax is as shown in Listing 6.3.

Listing 6.2: Decision policy 2

```
If ( direction is Provided && Available memory >= 50
&& CPU throughput <= 89 && light level >= 50
 && BatteryLevel >= 50)
  then {PlayVideo(); displayImage(); VoiceCommand();}
  else If ( BatteryLevel < 50
  || memory level < 50 || CPU >92)
  then {displayImage(); VoiceCommand();}
  else If( Available Memory level < 20
  && CPU > 92 && light level > 88)
   then VoiceCommand();
```

- **Policy 4:** This policy is attached to the 'fire alarm' COCA-component in Figure 6.4. The policy syntax is as shown in Listing 6.4.

The final architecture for this case study is shown in Figure 6.6. The architecture is shown the base-components, COCA-components and the connectors used to extend the application behaviour when a specific method in the interface is called.

Listing 6.3: Decision policy 3

```
If ( UserIsTrapped() && HumidityLevel = 100
&& Body_Temperature > 38 &&
 BloodPressure between 120 and 180
  && LowOxygenLevel
  && RoomTemperature <= −15 and >= 50 )
  then {
if(connection = GPRS)
{HelpMessage(); GetEmergencyNumber();}
else if (connection = WIFI) {WebToSMS();
        SendEmail(); BoradcastNotification();}
else if (connection = bluetooth)
{ OpenBluetooth(); Add_Bonjour_Service();
HelpMessage(); BoradcastAlert(); }
   PlayAlarmSound(); }
```

6.3 IWayFinder Implementation and Evaluation

The IWayFinder application has been implemented in two different versions, with and without the COCA-middleware. The battery life has been measured by running each version on an iPhone 4 device. The 'direction output' COCA-component in Figure 6.7 registers itself with the context manager to be notified when the *BatteryLevelDidChanged, CPULevelDidChanged, MemoryLevelDidChanged, DeviceOrientationDidChanged and/or LightLevelDidChanged.* When the context manager notifies *[PostNotfication:BatteryLevelDidChanged]* to the 'direction output' component, the adaptation manager reads the attached DPL in list 6.2. Based on the policy action, the adaptation manager calls the delegate object *DisplayDirection* which forwards the method invocation to the desired sub layer, based on the battery level. If the battery level < 20%, then the adaptation manager activates the sub layer VoiceCommand to adapt this context condition.

The experiments show that the COCA IWayFinder application saved battery consumption

Listing 6.4: Decision policy 4

```
If ( FireAlaramRasied && HumidityLevel < 10
&& Body_Temperature > 38 &&
 BloodPressure between 120 and 180
  && LowOxygenLevel
  && RoomTemperature >= 40 )
  then {
if(connection = GPRS)
{HelpMessage(); GetEmergencyNumber();}
else if (connection = WIFI) {WebToSMS();
        SendEmail(); BroadcastNotification();}
else if (connection = bluetooth)
{ OpenBluetooth(); Add_Bonjour_Service();
HelpMessage(); BroadcastAlert(); }
   PlayAlarmSound(); FindFireExit(); DisplayRoute(); }
```

by 13%, despite its self-adaptability, as shown in Figure 6.8. One of the expected benefits of using COCA-MDA in structuring the self-adaptive application is the enhancement of the context monitoring and detection processes. The IWayFinder implementation without the COCA-platform consumes more energy during context monitoring, thus draining the battery faster, because context changes are sent to a large subset of components. On the other hand, when the same application adapts the COCA-middleware, the application is able to adapt its behaviour and enhance both context monitoring and detection. The adaptation/configuration time and the context handling time are shown in Figure 6.9.

6.4 Evaluating COCA-MDA with COCOMO II

The IWayFinder application has been selected to determine the development effort using COCA-MDA compared with that using three MDA approaches proposed in the literature: U-MUSIC-MDA proposed by Khan [Khan, 2010], Paspallis's MDA proposed by Paspal-

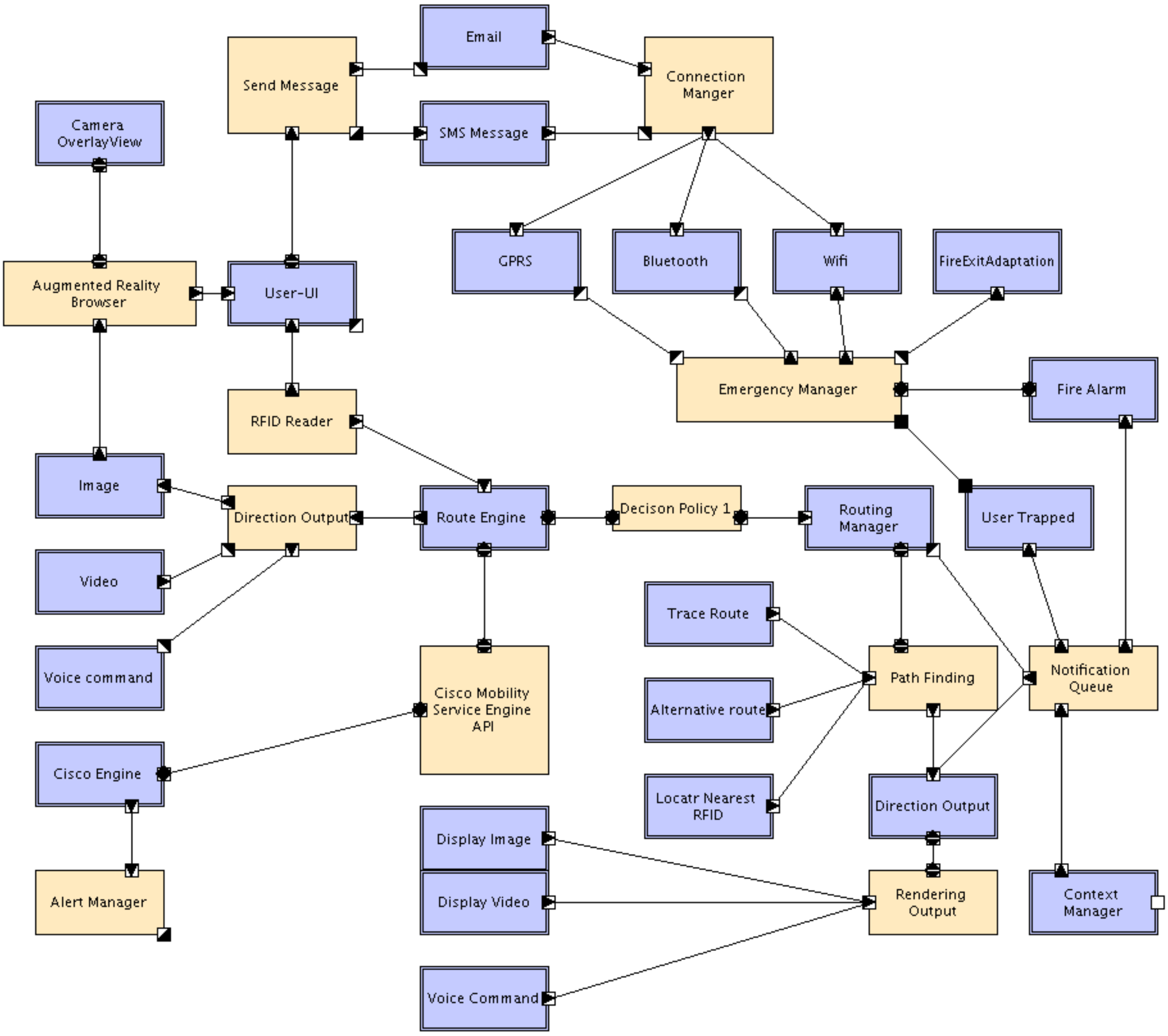


Fig. 6.6: I-WayFinder application architecture

lis [Paspallis, 2009], and MUSIC-MDA proposed by Wagner et al. [Wagner et al., 2011]. The enterprise architecture tool (EA) [SPARX Enterprise Architecture, 2010] was used to develop the IWayFinder application using the four MDAs (COCA, MUSIC, U-MUSIC, and Paspallis's). Each MDA phase was carried out separately. COCOMO II [Boehm et al., 2000] was used to find the development effort in person-months for each MDA. There are two COCOMO II models, i.e. the post-architecture and early design models. The post-architecture

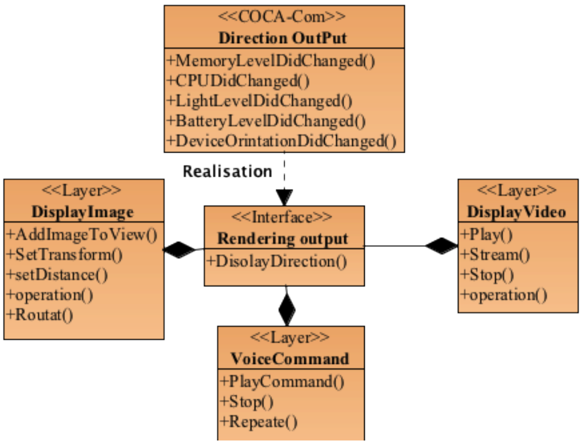


Fig. 6.7: Direction Output Context Oriented Component

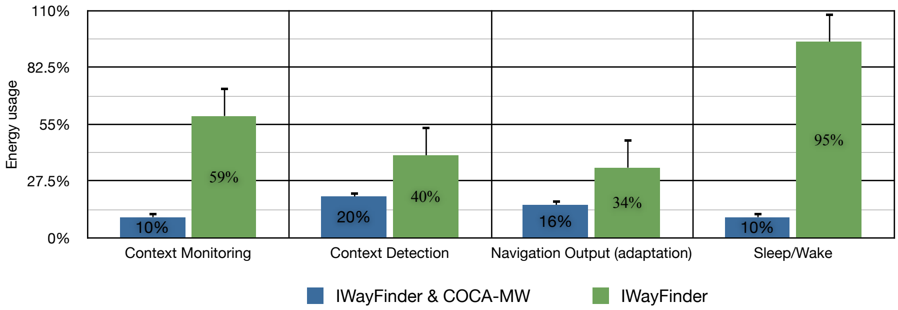


Fig. 6.8: Energy usage for IWayFinder application.

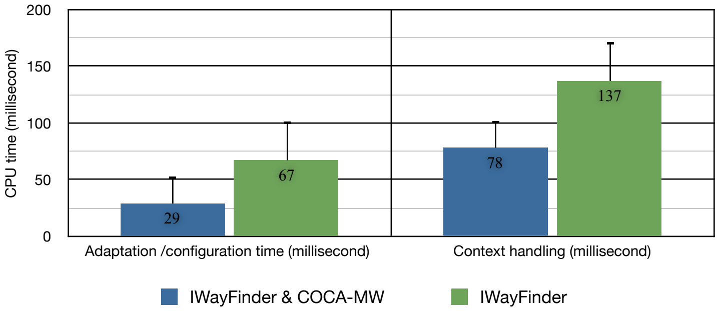


Fig. 6.9: Adaptation time (ms)

model is a detailed model used once the project is ready to develop and sustain a fielded system. The early design model is a high-level model which is used to explore alternative architectures or incremental development strategies [Boehm et al., 2000]. Based on the above, the post-architecture model has been selected to evaluate the four MDAs: COCA, MUSIC, U-MUSIC, and Paspallis's.

Based on the COCOMO II model, the sizing of new and reused code can be estimated via three major methods, as described in Boehm et al. [Boehm et al., 2000]. These methods are counting Source Lines Of Code (SLOC); counting Unadjusted Function Points (UFP); and aggregating new, adapted, and reused code, i.e. Adapted source lines of code (ASLOC). This type of reused code is estimated using the automatically translated code factor; this is considered to be a separate activity from development.

With regard to counting SLOC. The code generated from the MDA tool (EA) is excluded from the estimation. The effort for modelling the architecture can be captured using UFP. In such cases, COCOMO II is capable of relating UFP to SLOC in the implementation language. Starting from the fact that a UML is used to draw the model, the UML is classified on the same scale as a fourth-generation language. The relating process provides greater accuracy during the estimation than is obtained by estimating the generated lines of code using the MDA tool. Based on the above, the final SLOC for a module = the final application SLOC - the generated SLOC. This increases the accuracy of estimating the development effort.

COCOMO II is not only capable of estimating the cost and schedule for a development starting from 'scratch', it is also able to estimate the cost and schedule for products which are built upon already existing code, i.e. reused code. However, the third sizing measure, which aggregates new, adapted, and reused code, is suitable for MDA-based approaches. Starting from this fact, code taken from another source used in another product under development also contributes to the product's effective size. Pre-existing code which is treated as a white-box and is modified for use with a product is called adapted code. The effective size of reused and adapted code is adjusted to be its equivalent in new code. The adjustment on the additional effort it takes to modify the code for inclusion in the product. This method allows us to estimate the development effort during the transformation and deployment phases, phases which all MDA approaches have. When the developer transforms the application from a PIM into a PSM, specific configurations are needed and this can be captured by the percentage of code modified and the percentage of integration modified.

The following equations describe the effort Person-Months (PM) and the Time to Develop (TDEV), taking into consideration the aforementioned inputs, as shown in Equation 6.1. The primary equation in 6.1 denotes the effort in person-months derived from the software size defined in thousands of lines of code (KLOC). The exponent E defines the sum of the scale factors (SF), i.e. the Cartesian product of the effort multipliers (EM) and the constant value A, A value was calibrated from several software projects surveyed in Boehm et al. [Boehm et al., 2000]. The second equation in Equation 6.2 depicts the time required to develop a software, derived from the nominal effort (PM), the sum of SFs, and the constant values calibrated from several software projects evaluated in COCOMO II. The rating scale factors and the effort multipliers used in this work to derive the effort and the time required to develop the IWayFinder application using COCA-MDA.

$$\begin{gathered} PM = A \times (Size)^E \times \prod_{i=1}^{17} EM_i, \\ \text{where}\, E = B + (0.01 \times \sum_{i=1}^{17} SF_i), \\ A = 2.95,\ B = 0.91 \end{gathered} \tag{6.1}$$

$$\begin{gathered} TDEV = C \times (PM)^F, \\ \text{where}\, F = D + 0.2 \times (E - B), \\ C = 3.67,\ D = 0.28\, (\text{COCOMOII.2000}) \end{gathered} \tag{6.2}$$

Thus, counting the SLOC is not adequate for evaluating the development effort in MDA-based methodology. Sizing software maintenance is better for MDA because, after the code is generated, the developer has to maintain the code and add the target platform configuration. This is required in the PSM phase and in the deployment and transformation phases. So, Equation 6.3 is used to calculate the sizing of code maintenance [Boehm et al., 2000]. The initial maintenance size estimate is adjusted with a maintenance adjustment factor (MAF). This relationship can estimate the level of effort, using the Full Time Equivalent Software Personnel FSP_M, given T_M as in annual maintenance estimates, as shown in Equation 6.4, where $T_M = 12$ months, or, given a fixed maintenance staff level, FSP_M, determine the necessary time, T_M, to complete the effort [Boehm et al., 2000]. To estimate the adapted code, the COCOMO II model uses an additional set of equations to calculate the final count for source instructions and related costs and schedule. The equations in 6.3, 6.4, and 6.5 use the following values as parameters.

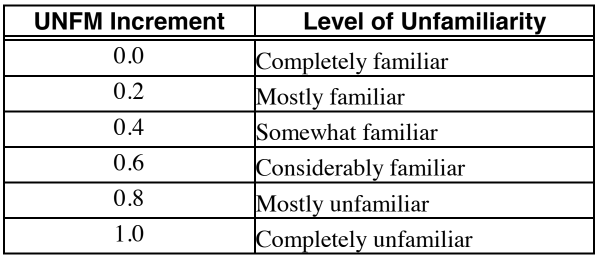
- ASLOC. The number of source lines of code adapted from existing software used in developing the new product.

- Percentage of design modification (DM). The percentage of the adapted software's design which received modifications to fulfil the objectives and environment of the new product.

- Percentage of code modification (CM). The percentage of the adapted software's code which receives modifications to fulfil the objectives and environment of the new product.

- Percentage of integration required for modified software (IM). The percentage of effort needed for integrating and testing of the adapted software in order to combine it into the new product.

- Percentage of reuse effort resulting from software understanding (SU). Percentage of reuse effort resulting from assessment and assimilation (AA); programmer unfamiliarity with software domain (UNFM). Boehm et al. [Boehm et al., 2000] provides a rating scale for programmer unfamiliarity (UNFM) as shown in Table 6.10, .



UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

Fig. 6.10: Rating Scale for Programmer Unfamiliarity (UNFM)

$$MAF = 1 + \left(\frac{SU}{100} \times UNFM \right), \tag{6.3}$$

SU: Software Understanding (zero if DM = 0 and CM = 0),

DM: percentage of design modified,

CM: percentage of code modified,

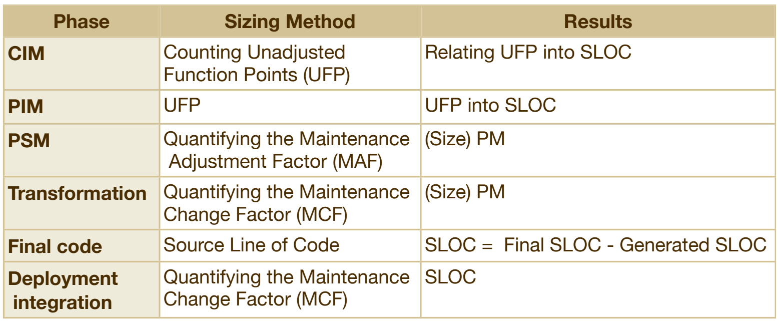
$$UNFM = 0.4$$

$$PM_M = T_M - FSP_M, \tag{6.4}$$

where T = 12 months

$$PM = AX(Size)^B + \left\lceil \frac{ASLOC(\frac{AT}{100})}{ATPROD} \right\rceil \tag{6.5}$$

In general, MDA-based approaches must apply Computation Independent Model (CIM), PIM, PSM, transformation, deployment, and code generation. For each phase in the MDA a sizing method was adapted for estimating the development effort as shown in Table 6.1. However, the code which is directly generated from the MDA tool (EA) is excluded from the development effort, but is used as an input to measure the software maintenance effort. In addition, the middleware code has to be adapted and maintained, or even configured, to suit the new application platform.



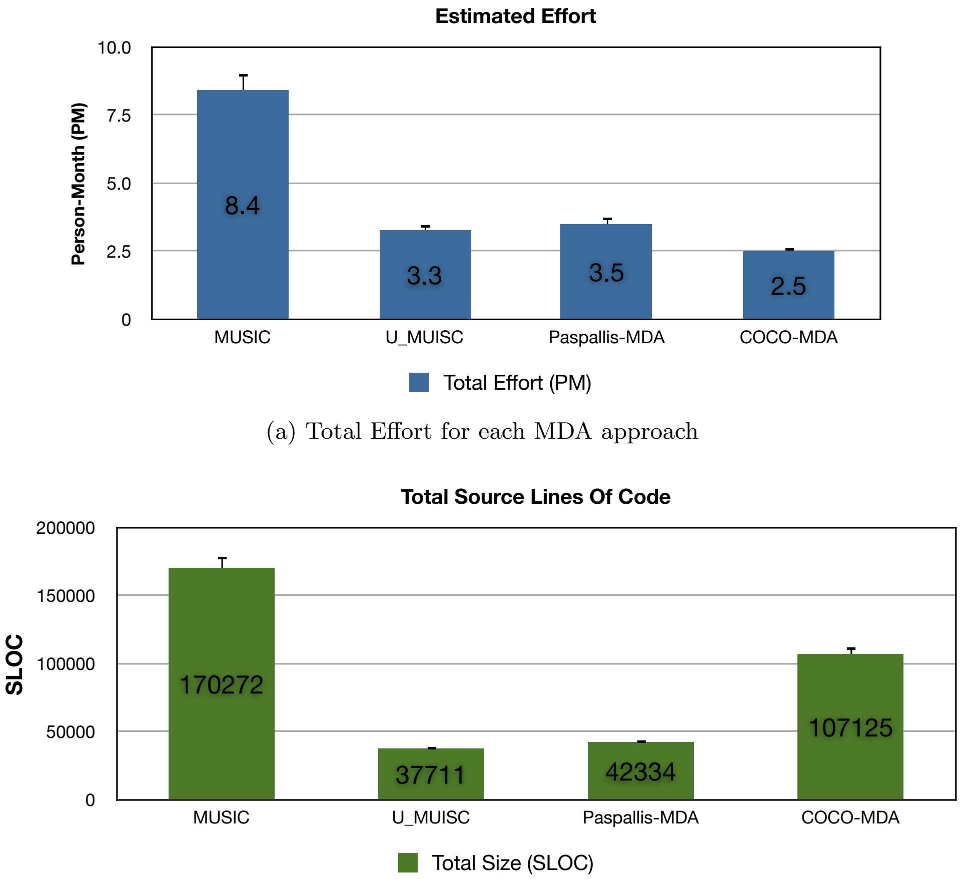
Phase	Sizing Method	Results
CIM	Counting Unadjusted Function Points (UFP)	Relating UFP into SLOC
PIM	UFP	UFP into SLOC
PSM	Quantifying the Maintenance Adjustment Factor (MAF)	(Size) PM
Transformation	Quantifying the Maintenance Change Factor (MCF)	(Size) PM
Final code	Source Line of Code	SLOC = Final SLOC - Generated SLOC
Deployment integration	Quantifying the Maintenance Change Factor (MCF)	SLOC

Table 6.1: MDA phases and Size factors

6.4.1 COCOMO II Evaluation Results

The COCOMO II tool was used to estimate COCOA-MDA, U-MUSIC, MUSIC, and Paspallis's MDA. The evaluations produced the following results for COCA-MDA and the alternative methodologies.

Figure 6.11 provides the estimated efforts for the four MDAs. It also shows the total size (SLOC) for the IWayFinder application after it has been developed in each MDA. The figure shows that COCA-MDA requires less effort in PM, despite the fact that the total SLOC is greater than for Paspallis's MDA. In Paspallis's MDA, each context provider requires a separate plug-in architecture, which requires new software engineering to build the plug-in. The MDA tool does not generate the required code for the plug-in, but leaves the required code to be composed and configured in the deployment stage. This requires more effort to configure and maintain the plug-in architecture. This effort is captured using the UFP analysis, so the total effort for Paspallis's MDA is one of the highest because the ratio of the maintenance adjustment factor is very high. Such facts demonstrate the accuracy obtained using COCOMO II in estimating self-adaptive software development methodology. In addition, the figure shows that the effort in MUSIC is the greatest; the reason for this is a lower ratio of adaptive and reused code in MUSIC compared to that in its extensions U-MUSIC and Paspallis's MDA.

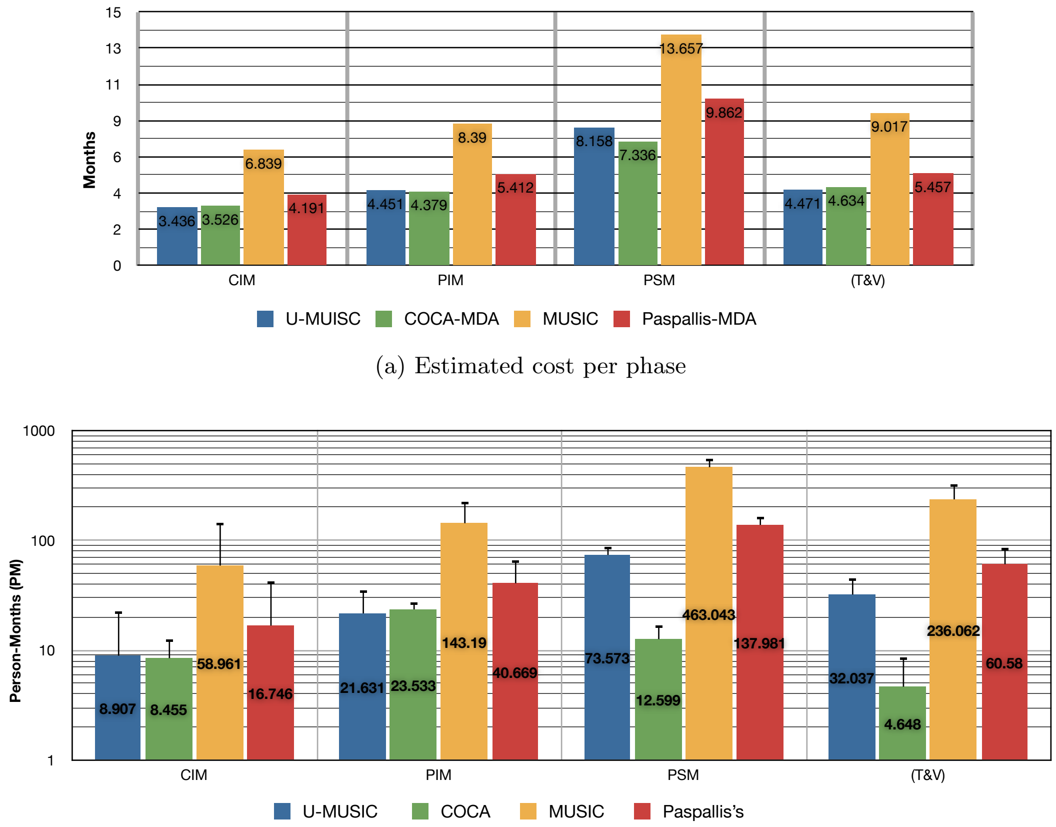


(a) Total Effort for each MDA approach

(b) Total Source Lines Of Code

Fig. 6.11: Total Effort for each MDA approach

Figure 6.12a provides more information for each MDA in terms of the estimated cost per MDA phase. As shown in the figure, the cost of performing the PIM was large for all MDAs. The reason for this is that all MDAs focus more on modelling the application variation model through the PIM. The cost of adapting the PIM in MUSIC is the largest because of the complexity of adapting the MUSIC PIM tasks; this requires the developer to produce more UML models than in the others. For the same stage, Paspallis's MDA comes with less cost. In Paspallis's MDA, the time spent by the developers in building the context-provider plug-ins is greater than the effort required to build the architecture itself. This is why Paspallis's



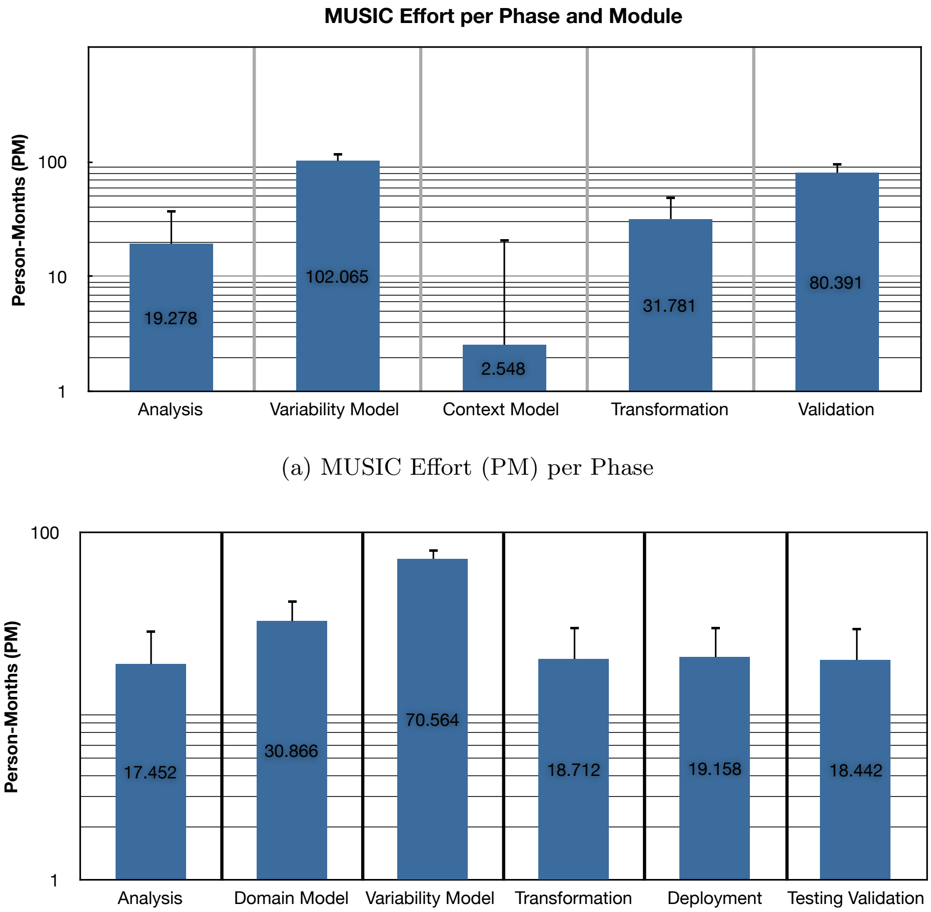
(a) Estimated cost per phase

(b) MDA Cumulative effort in person-months

Fig. 6.12: Cumulative Effort per model/MDA in person-months

MDA comes second, after U-MUSIC, when evaluating the PSM phase.

Figure 6.12b provides the cummulative cost in PM for each MDA phase. As shown in the figure, the cost of performing the PIM was large for all MDAs. COCA-MDA reduced the effort required to generate the PSM during deployment, as shown in Figure 6.12b. On the other hand, Paspallis's MDA increased the effort required for software maintenance in the transformation and deployment phases. Specifically, COCA-MDA and U-MUSIC reduce the effort needed to implement new or reused context provider i.e integrating a new sensor in the platform. This result reflects the benefits gained from employing the COCA-ADL for architecture deployment in several platforms. It is worth mentioning here that the 'labour rate per month' has been given the same value for all the MDAs throughout the evaluation.



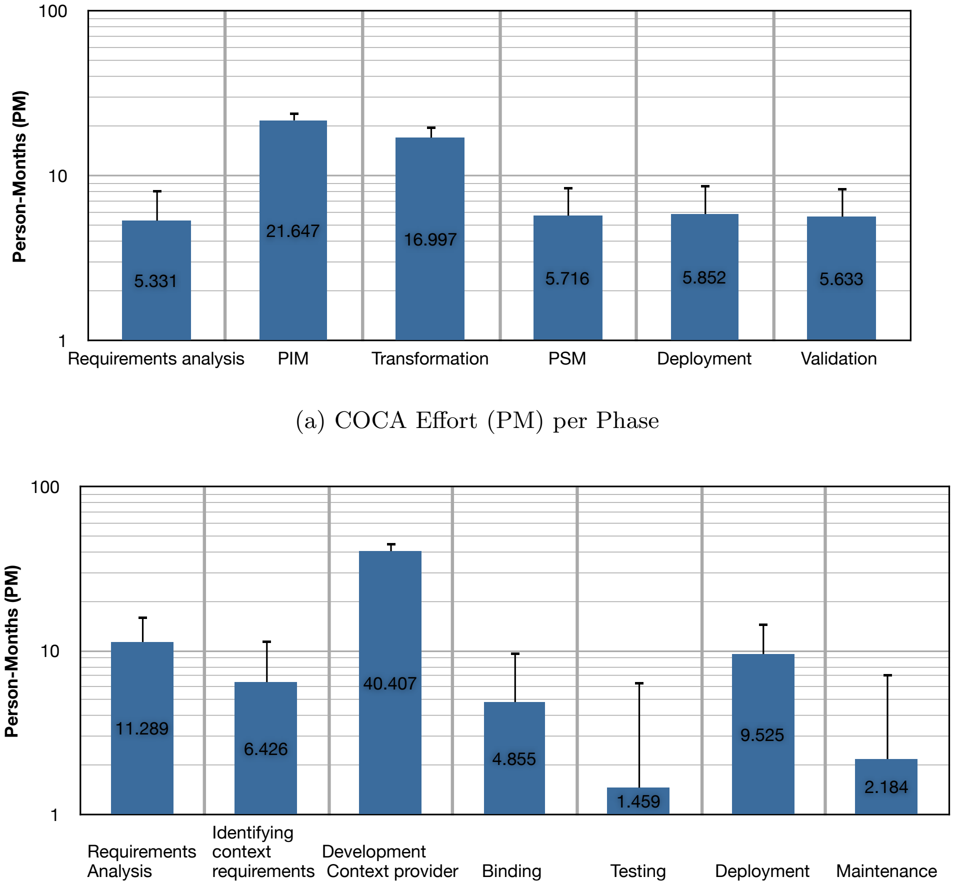
(a) MUSIC Effort (PM) per Phase

(b) U-MUSIC Effort (PM) per Phase

Fig. 6.13: MUSIC-MDA and U-MUSIC-MDA estimated efforts

In order to provides more information about each MDA approach, we have analysed the effort per phase for each MDA. Figure 6.13a shows the estimated effort for each phase for the MUSIC methodology. In tis case, the design of variability models and validation require more effort than in the others, but modelling the context model require less effort. This figure demonstrate that MUSIC requires more effort and provides no cost effectiveness in developing the IWayFinder application.

In the same way, the U-MUSIC evaluation is illustrated in Figure 6.13b. The domain



(a) COCA Effort (PM) per Phase

(b) Paspallis's MDA Effort (PM) per phase

Fig. 6.14: COCA-MDA and Paspallis's MDA estimated efforts

model propsoed by U-MUSIC MDA requires more effort than the variability model does. In U-MUSIC, the domain model requires the developer to split the context model into four models: functionality ontology, service ontology, context and resource model, and context provider. These models require more effort than building a simple context model like MUSIC. These models are collaborated into architecture constraints in the variability model, which uses them as inputs for the utility functions. Such an effort in domain modelling can increase the developers' understanding of the application domain, but it does not really enable them to enhance the architecture design. In our experience, the results from the domain model are not reflected in the architecture variability model; the domain model is only used to obtain information on the architecture constraints which are used as input for the utility function.

Figure 6.14a shows the estimated effort for each phase in Paspallis's MDA methodology. The development of context providers and analysis are the phases which require most effort by the developers. The effort in the deployment and maintenance phases are very high compared to those in the others. Thus, a planning-based adaptation requires more effort in the requirements and the proposed methodology requires more effort in developing the required plug-ins which fit the planned adaptation. Although this methodology does not suit self-adaptive applications when unanticipated conditions are in place, it does increase the development and maintenance efforts.

Figure 6.14a shows the estimated effort for each phase for the COCA-MDA methodology. The figure illustrates that less effort is required to construct the application through the COCA-MDA phases. For example, to model the PIM of the architecture, 21 PM are required in COCA-MDA, but MUSIC requires 102 PM, U-MUSIC requires 70.5 PM, and Paspallis's MDA requires 40.4 PM, assuming that the context providers are not changed at runtime with respect to Paspallis's MDA. The intensive analysis of the application requirements in COCA-MDA simplified the process of modelling the variability model. Instead of modelling several variation models, as in MUSIC and U-MUSIC, the developers model one extra-functionality model and another core structure model. In addition, the methodology modularizes each context-dependent functionality in a separate component model instead of designing a new plug-in from scratch and then configuring it, as in Paspallis's MDA.

Finally, Figure 6.15 shows the required staff per phase in each methodology. The MUSIC methodology requires the most staff to develop the IWayFinder application, and COCA-MDA requires the least. Next to MUSIC comes Paspallis's MDA and then U-MUSIC. This analysis reflects the effort required in 12 months with respect to the ratio of code maintenance and deployment plus the required effort to model the architecture.

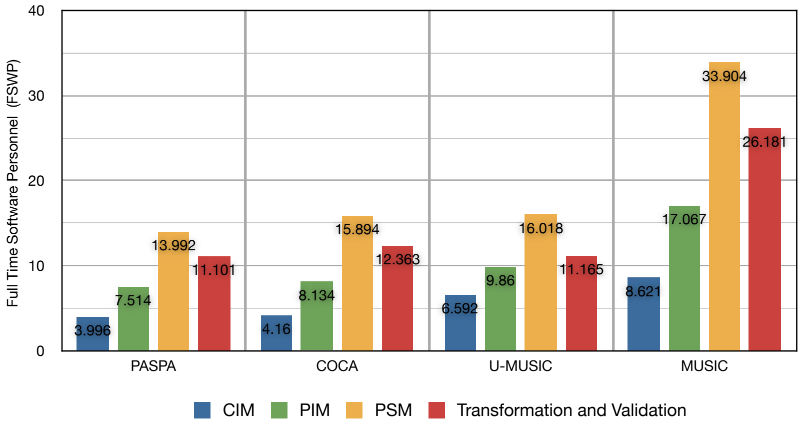


Fig. 6.15: Project personnel for each phase for each MDA approach

6.5 Lessons Learned

COCA-MDA provides the following benefits.

- Intensive analysis of the application requirements simplified the process of modelling the application's behavioural model, so, instead of modelling several variation models as in MUSIC and U-MUSIC, the developer models one behavioural model.

- It enables the architecture to anticipate several behavioural variations, based on the context and the specific needs of individuals with cognitive impairments.

- It enables the application to proactively anticipate or reactively address unforeseen changes through the support of a dynamic decision-making and policy framework. The policy framework is based on a stable description of software models and proprieties.

- It can decompose the application into several architectural units to allow developers to decide which part of the architecture should be notified when a specific context condition occurs.

- Counting the SLOC is not adequate for evaluating the development effort in MDA-based methodology. Sizing software maintenance is better for MDA because, after the code is generated, the developer has to maintain the code and add the target platform configuration.

- Clearly, COCA-MDA has reduced the development effort and increased the architecture's ability to adapt to context changes.
- COCA-MDA decreases the development effort because it uses a clear separation of concerns and employs a decomposition mechanism to produce a context-oriented component model. Using these technique reduces the modelling tasks and combines the MDA phases in a simple way.

6.6 Summary

Self-adaptability requirement, modelling, architecture, implementation, and assurance approaches require a systematic solution which inter-relates all aspects on a single platform. Requirements analysis can provide a great deal of information about the extra-functionalities of the self-adaptive system. In the same way, requirements analysis can facilitate and simplify architecture reflection by providing the information required by the software to manage itself. Moreover, COCA-MDA can reduce the complexity of self-adaptive engineering through mapping requirements to actor-, system-, and environment-dependent behaviours. This study shows how COCA-MDA reduces the required development effort compared to other MDAs. It also demonstrates how COCA-MDA reduces the software maintenance ratio through the architecture deployment and transformation.

Chapter 7

Conclusions

This thesis described COSD as a new way of building context-dependent and self-adaptive applications using a combination of a model-driven architecture which generates an ADL presenting the architecture as a components-based system, and a runtime infrastructure which enables transparent self-adaptation with the underlying context environment.

Specifically, a model-driven architecture was used to demonstrate a new approach to building self-adaptive applications by adapting the COCA-MDA methodology. This methodology enables developers to modularize an application, based on context-dependent behaviours, and to separate context-dependent functionalities from the context-free functionality. It also enables dynamic context-driven adaptation without overwhelming the quality attributes. In addition, developers can design the system to proactively or reactively anticipate context changes by providing a decision policy which triggers the adaptation whenever specific context values cross lower or upper limits. This process is easy to accomplish as long as the middleware is aware of which parts of the architecture are affected by the changes. A predefined policy can provide the middleware with sufficient information to perform the adaptation.

COCA-middleware performs the adaptation processes, including context monitoring and detecting and dynamic decision-making, and maintains the architecture quality attributes during the adaptation. The COCA-middleware uses the operations among the device's resources and considers the interoperability among the architectural components. The middleware verifies the adaptation output among the available resources and the trade-offs among the quality attributes of the architecture. The effects of context monitoring and detecting on the device's resources are enhanced. The COCA-middleware considers a dynamic decision-making process which determines the parts which need to be changed and how to change them to achieve the best output. The COCA-middleware reduces the impact and costs of the adaptation process with respect to the device's resources and achieves the properties of a self-adaptive system, specifically self-configuring and self-optimizing, without affecting the quality attributes of the architecture.

7.1 Achievements

The COSD has successfully modularized the self-adaptive application, based on the context-dependent behaviour. It facilitate the development of this family of applications when context anticipation is in place. The development methodology enables developers to design an application as sets of context-free and context-dependent components. The COCA-component encapsulates a context-dependent behaviour with variant implementations based on context. Finally, the COCA-MDA generates an architecture described by COCA-ADL which is ready to be deployed on several platforms. The finidings of this study can be summarised by the following:

- Modularizing the software based on context-dependent functionality provides software systems with adaptability and variability.

- Supporting context-binding mechanisms with observer patterns was impossible before the introduction of COCA-MDA methodology; COCA-MDA provides clear separation between the context provider and consumer. In addition, it modularizes the application components, based on context. This makes identifying which component must respond to a specific context condition an easy task in the design phase.

- The ATAM evaluation showed that the architecture can anticipate unforeseen changes dynamically by integrating policy mismatch resolution with architecture introspection, as well as instantiating a composition plan. The adaptation manager then verifies the fitness of the composition plan from the unforeseen changes. Performance and modifiability do, in fact, trade-off with each other.

- The dynamic decision is used for tuning the adaptation process, assuring the adaptation, and verifying its results. The COCA-middleware is therefore able to switch autonomously between weak adaptation and strong adaptation types.

- The COCA-middleware can invoke or revoke a specific component dynamically without affecting the device performance; however, the component manager must then verify to the component's ability to provide/require specific methods, services, and resources.

- The interoperability of the application components and their sublayers are a major feature of COCA-middleware. The evaluation shows the architecture's ability to deactivate or activate sublayers implementation when driven by a context change. However, this feature is supported by adapting two design patterns when implementing the adaptation manager.

- The evaluation verifies the ability of the COCA-middleware to perform dynamic assurance and verification of the adaptation output.

- COCA-middleware supports the self-adaptive application assurance by enabling the developer to specify the decision policy through the development methodology and a middleware that maintaining them at runtime. Providing dynamic verification and validation methods rely on a stable description of software models and proprieties in the configuration element in the COCA-ADL.

- The COSD paradigm increases the software-development productivity. This achieved by employing a decomposition strategy and a model driven approach for building self-adaptive software systems.

- The COCA-middleware supports self-adaptability (Self-configuring and self-tuning) and dependability in mobile computing environment.
- The evaluation of the COSD in comparison to AOSD shows that COSD is better suited to implementing context-dependent and self-adaptive applications.
- Programming-level techniques like COP and AOP are not sufficient to build self-adaptive software.
- The performance and energy usage in the software that implements DAOP engines are very poor, because of the continuous evaluation between the passive and active context state in each joinpoint.
- There is no doubt that AOP frameworks can be used for developing and implementing self-adaptive applications, but their performance is very poor in comparison to that of COCA-middleware.
- Programming-level approaches like JCOP and JCOOL tend to support self-tuning of software systems with an acceptable level of performance, but the overall support of adaptability and variability is very limited in comparison with architecture evolution approaches such as MUSIC, MADAM, and COCA-middleware. However, the programming techniques are better suited to small-scale context-dependent applications, and they require extensive modification for supporting context monitoring, context detection, and dynamic decision-making.
- The COCA-middleware implementation performs better with regard to adaptation processes, including context monitoring, detecting, decision-making, and adaptation. The evaluation results shows that implementing self-adaptive applications with the aid of COCA-middleware can support software adaptability and variability with affordable adaptation costs and less impact on the allocated resources.
- Separating the adaptation logic and the context model from the program business logic increases software maintainability and scalability.

- MDA-based approaches are not always increase the development productivity. MDA-based approaches must pay more attention over the implication of PSM configurations.

- Self-adaptive software can be built using a generic development paradigm supported by an adaptation engine that use generic design principles and patterns. As result self-adaptive software can be implemented using a standard Object Oriented Programming (OOP).

7.2 Future work

This section outlines the key areas identified for future work. Future work targets improving the COCA-platform and its integration with more complex context models and applications. The current COCA-component model can facilitate the development of context-aware applications in mobile devices. The COCA-components need improvements to be more generic and customizable to suit a wide range of applications. Potentially, a generic component framework which implements generic context-dependent behaviour can improve application capability and introduce dynamic behaviour adaptations.

The decision policies require more effort with respect to policy mismatch and resolution. This is in line with improving the capabilities of self-assurance and dynamic evaluation of the adaptation output. This thesis proposed a closed adaptation system; improving the current COCA-platform toward an open adaptation system will increase its ability to adapt to unanticipated context changes. Improving dynamic decision-making can increase the ability to achieve several self-adaptability properties such as self-recovering and self-healing. In addition, the introduced architecture needs to be tested with more practical case studies to show its fitness in addressing anticipation and uncertainty in the context model. The architecture evaluation used in this thesis can outline the fitness of the architecture with respect to certain scenarios. However, an intensive architecture evaluation, for example using the adaptability evaluation method (AEM) [Tarvainen, 2007], can inform designers with regard to major modifications to the architecture.

The extension of these processes to runtime involved moving the measurement of quality attributes and performing the trade-off analysis in runtime. This approach sounds promising but further justification is required in terms of the measuring mechanism and the dynamic decision-making evaluation, which are both challenging tasks for self-adaptive applications. One can ask whether such an approach can ensure the accuracy of the data in terms of imperfection and uncertainty. The second concern is how a software makes decisions when several quality attributes change at the same time. Identifying trade-off points is not feasible because of the complexity involved in identifying the dependence among several quality attributes of sensitivity points.

The COCA-MDA requires an improvement supporting requirements reflection and modelling requirements as runtime entities. This can be used to anticipate the evolution of both functional and non-functional requirements. However, the requirements reflection mechanism requires support at the modelling level and the architectural level. Moreover, to achieve such a reflection, the COCA-ADL must support evolution over the operation time of the software system. This thesis proposed a reflective middleware support for self-adaptability; improving the middleware to support a reflection at the meta-level can provide a mechanism for transparency between runtime context changes and the underlying context requirements. Moreover, using COSD to build self-adaptive software might requires further investigation related to the quality of user experience, which measures the end users satisfaction and their experience of the adapted behaviour.

7.3 Summary

This chapter summarized the motivation for the research undertaken and the most significant achievements of the work presented in this thesis. It outlined how this work contributed to the state-of-the art in self-adaptive models, targeting context-aware applications by providing a COCA-platform. The behavioural decomposition of a context-aware application modularizes the application into two casually connected layers of component sets: the base-components and the COCA-components. The COCA-components are managed by a middleware and introduce context-dependent behaviour dynamically, based on the context state. Integrating separation of concerns with a component-based framework through a model-driven architecture support generates context-oriented component-based applications which have the ability to modify their behaviour at runtime by invoking the COCA-components as needed.

Progress toward self-adaptation is realized by COCA-middleware, which ensures the fitness of the adaptation results among the architecture quality attributes. Two case studies have been selected to evaluate the development approach by COCA-MDA, and the generated architecture has been implemented in a mobile device to verify to its ability to achieve the objectives of the proposed design. The implementation, architecture evaluation, and methodology evaluation showed that the proposed tool suite can be used to support dynamic context-dependent behaviours adaptation.

Bibliography

[Achilleas, 2010] Achilleas (2010). *Model-Driven Petri Net based Framework for Pervasive Service Creation*. PhD thesis, University of Essex.

[Al-Begain, 2004] Al-Begain, K. (2004). "Performance Models for 2.5/3G Mobile Systems and Networks". In Calzarossa, M. and Gelenbe, E., editors, *Performance Tools and Applications to Networked Systems*, volume 2965 of *Lecture Notes in Computer Science*, pages 143–167.

[Amano and Watanabe, 1999] Amano, N. and Watanabe, T. (1999). "LEAD++: an object-oriented language based on a reflective model for dynamic software adaptation". In *Proceedings of Technology of Object-Oriented Languages and Systems Conference*, (TOOLS '99), pages 41–50, Washington, DC, USA.

[Amoui et al., 2011] Amoui, M.; Derakhshanmanesh, M.; Ebert, J.; and Tahvildari, L. (2011). "Software Evolution towards Model-Centric Runtime Adaptivity". In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, (CSMR 2011), pages 89 –92, Oldenburg, Germany.

[Anthony et al., 2009] Anthony, R.; Chen, D.; Pelc, M.; Perssonn, M.; and rngren, M. T. (2009). "Context-Aware Adaptation in DySCAS". *Electronic Communications of the EASST*, 19, pp. 15.

[Anthony et al., 2008a] Anthony, R.; Pelc, M.; Ward, P.; and Hawthorne, J. (2008a). "Flexible and Robust Run-Time Configuration for Self-Managing Systems". In *Proceedings of*

IEEE International Conference on Self-Adaptive and Self-Organizing Systems, (SASO '08), pages 491–492, Los Alamitos, CA, USA.

[Anthony et al., 2008b] Anthony, R.; Pelc, M.; Ward, P.; Hawthorne, J.; and Pulnah, K. (2008b). "A Run-Time Configurable Software Architecture for Self-Managing Systems". In *Proceedings of the 2008 International Conference on Autonomic Computing*, (ICAC '08), pages 207–208, Washington, DC, USA.

[Apel et al., 2006] Apel, S.; Leich, T.; and Saake, G. (2006). "Aspectual mixin layers: aspects and features in concert". In *Proceedings of the 28th international conference on Software engineering*, (ICSE '06), pages 122–131, Shanghai, China. ACM.

[Appeltauer et al., 2011] Appeltauer, M.; Hirschfeld, R.; Haupt, M.; and Masuhara, H. (2011). "ContextJ: Context-oriented Programming with Java". *Information and Media Technologies*, 6(2), pp. 399–419.

[Appeltauer et al., 2009] Appeltauer, M.; Hirschfeld, R.; and Masuhara, H. (2009). "Improving the development of context-dependent Java applications with ContextJ". In *Proceedings of the International Workshop on Context-Oriented Programming*, (COP '09), pages 5:1–5:5, Genova, Italy.

[Appeltauer et al., 2008] Appeltauer, M.; Hirschfeld, R.; and Rho, T. (2008). "Dedicated Programming Support for Context-Aware Ubiquitous Applications". In *Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, (UBICOMM '08), pages 38–43, Valencia, Spain.

[Apple IPhone Operating System IOS, 2011] Apple IPhone Operating System IOS (2011). "IOS 4.0 Apple Developer Library". `http://developer.apple.com/library/ios/navigation/`. "[Online; accessed 1-April-2011]".

[Asadi and Ramsin, 2008] Asadi, M. and Ramsin, R. (2008). "MDA-Based Methodologies: An Analytical Survey". In *Proceedings of the Euro Conference on Model Driven Architecture Foundations and Applications*, (ECMDA-FA 2008), pages 419–431, Berlin, Germany.

[AspectCOCA, 2011] AspectCOCA (2011). "Aspect Oriented Programming Framework for Cocoa and Objective-C.". `http://www.cocoadev.com/index.pl?AspectCocoa`. "[Online; accessed 1-June-2011]".

[Baresi and Ghezzi, 2010] Baresi, L. and Ghezzi, C. (2010). "The disappearing boundary between development-time and run-time". In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, (FoSER '10), pages 17–22, Santa Fe, New Mexico, USA.

[Bartolomeo et al., 2008] Bartolomeo, G.; Salsano, S.; Melazzi, N.; and Trubiani, C. (2008). "SMILE- Simple Middleware Independent LayEr for Distributed Mobile Applications". In *Proceedings of the Wireless Communications and Networking Conference*, (WCNC 2008), pages 3039–3044, Las Vegas, USA.

[Bass et al., 2003] Bass, L.; Clements; and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley, 2nd edition.

[Becker et al., 2004] Becker, C.; Handte, M.; Schiele, G.; and Rothermel, K. (2004). "Pcom-a component system for pervasive computing". In *Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, (PerCom '04), pages 67–76, Orlando, Florida.

[Begain et al., 2001] Begain, K.; Bolch, G.; and Herold, H. (2001). *Practical Performance Modeling: Application of the Mosel Language*. Kluwer Academic Publishers, Norwell, MA, USA.

[Belaramani et al., 2003] Belaramani, N. M.; Wang, C.-L.; and Lau, F. C. M. (2003). "Dynamic Component Composition for Functionality Adaptation in Pervasive Environments". In *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, (FTDCS '03), pages 226–232, San Juan, Puerto Rico.

[Bencomo et al., 2010] Bencomo, N.; Whittle, J.; Sawyer, P.; Finkelstein, A.; and Letier, E. (2010). "Requirements reflection: requirements as runtime entities". In *Proceedings* *of the 32nd ACM/IEEE International Conference on Software Engineering, (ICSE '10)*, volume 2 of *LNCS*, pages 199–202, CAPE TOWN, South Africa.

[Blair et al., 2009] Blair, G.; Bencomo, N.; and France, R. (2009). "Models@ run.time". *Journal of IEEE Computer*, 42(10), pp. 22 –27.

[Boehm et al., 2000] Boehm, B. W.; Clark; Horowitz; Brown; Reifer; Chulani; Madachy, R.; and Steece, B. (2000). *Software Cost Estimation with Cocomo II.* Prentice Hall PTR, 1st edition.

[Broens et al., 2007] Broens, T.; Quartel, D.; and Van Sinderen, M. (2007). "Capturing context requirements". In *Proceedings of the 2nd European conference on Smart sensing and context*, (EuroSSC '07), pages 223–238, Kendal, England.

[Bruneton et al., 2002] Bruneton, E.; Coupaye, T.; and Stefani, J. (2002). "Recursive and dynamic software composition with sharing". In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, (WCOP'02), pages 1–26, Malaga, Spain.

[Buck and Yacktman, 2010] Buck, E. and Yacktman, D. (2010). *Cocoa design patterns.* Developer's Library, 2nd edition.

[Calic et al., 2008] Calic, T.; Dascalu, S.; and Egbert, D. (2008). "Tools for MDA Software Development: Evaluation Criteria and Set of Desirable Features". In *Proceedings of the Fifth International Conference on Information Technology*, (ITNG 2008), pages 44–50, Istanbul, Turkey.

[Capra, 2003] Capra, L. (2003). *Reflective mobile middleware for context-aware applications.* PhD thesis, University of London.

[Carton et al., 2007] Carton, A.; Clarke, S.; Senart, A.; and Cahill, V. (2007). "Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing". In *Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing*

Applications, Systems, and Environments, (SEPCASE '07), pages 5–10, Washington, DC, USA.

[Cheng et al., 2008] Cheng, B. H.; Giese, H.; Inverardi, P.; Magee, J.; de Lemos, R.; Andersson, J.; Becker, B.; Bencomo, N.; Brun, Y.; Cukic, B.; Serugendo, G. D. M.; Dustdar, S.; Finkelstein, A.; Gacek, C.; Geihs, K.; Grassi, V.; Karsai, G.; Kienle, H.; Kramer, J.; Litoiu, M.; Malek, S.; Mirandola, R.; Müller, H.; Park, S.; Shaw, M.; Tichy, M.; Tivoli, M.; Weyns, D.; and Whittle, J. (2008). "Software Engineering for Self-Adaptive Systems: A Research Road Map". In *Proceedings of Dagstuhl Seminar, Software Engineering for Self-Adaptive Systems*, (Dagstuhl Seminar '08), pages 1–26, Dagstuhl, Germany.

[Cheng et al., 2009] Cheng, S. W.; Garlan, D.; and Schmerl, B. (2009). "Evaluating the effectiveness of the Rainbow self-adaptive system". In *Proceedings of the ICSE 2009 Workshop on Software Engineering for Adaptive and Self-Managing Systems*, (ICSE '09), pages 132–141, Washington, DC, USA.

[Chusho et al., 2000] Chusho, T.; Ishigure, H.; Konda, N.; and Iwata, T. (2000). "Component-based application development on architecture of a model, UI and components". In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference Conference*, (APSEC '00), pages 349–358, Singapore.

[Cisco context-ware software, 2011] Cisco context-ware software (2011). `http://www.cisco.com`. [Online; accessed 1-April-2011].

[Clemente et al., 2011] Clemente, P. J.; Hernandez, J.; Conejero, J. M.; and Ortiz, G. (2011). "Managing crosscutting concerns in component based systems using a model driven development approach". *The Journal of Systems and Software*, 84, pp. 1032–1053.

[Costanza, 2005] Costanza, P. (2005). "Language constructs for context-oriented programming". In *Proceedings of the Dynamic Languages Symposium*, (DLS'05), pages 1–10, Reno, NV, USA.

[Costanza et al., 2006] Costanza, P.; Hirschfeld, R.; and Meuter, W. D. (2006). "Efficient layer activation for switching context-dependent behavior". In *Proceedings of the 7th Joint Modular Languages Conference*, (JMLC 06), pages 84–103, Oxford, UK.

[Daniele et al., 2007] Daniele, L.; Dockhorn Costa, P.; and Ferreira Pires, L. (2007). "Towards a Rule-Based Approach for Context-Aware Applications". In *Proceedings of the 13th open European summer school and IFIP conference on Dependable and adaptable networks and services, (EUNICE '07)*, volume 4606 of *LNCS*, pages 33–43, Enschede, The Netherlands.

[Daniele et al., 2009] Daniele, L. M.; Ferreira Pires, L.; and Sinderen, M. (2009). "An MDA-Based Approach for Behaviour Modelling of Context-Aware Mobile Applications". In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, (ECMDA-FA '09), pages 206–220, Birmingham, UK.

[Dashofy et al., 2007] Dashofy, E.; Asuncion, H.; Hendrickson, S.; Suryanarayana, G.; Georgas, J.; and Taylor, R. (2007). "ArchStudio 4: An Architecture-Based Meta-Modeling Environment". In *Proceedings of the 29th International Conference on Software Engineering*, (ICSE '07), pages 67–68, Washington, DC, USA.

[de Farias et al., 2007] de Farias, C. R. G.; Leite, M. M.; Calvi, C. Z.; Pessoa, R. M.; and Filho, J. G. P. (2007). "A MOF metamodel for the development of context-aware mobile applications". In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing*, (SAC '07), pages 947–952, Seoul, Korea.

[de Lemos et al., 2011] de Lemos, R.; Giese, H.; Müller, H.; Shaw, M.; Andersson, J.; Baresi, L.; Becker, B.; Bencomo, N.; Brun, Y.; Cikic, B.; Desmarais, R.; Dustdar, S.; Engels, G.; Geihs, K.; Goeschka, K. M.; Gorla, A.; Grassi, V.; Inverardi, P.; Karsai, G.; Kramer, J.; Litoiu, M.; Lopes, A.; Magee, J.; Malek, S.; Mankovskii, S.; Mirandola, R.; Mylopoulos, J.; Nierstrasz, O.; Pezzè, M.; Prehofer, C.; Schäfer, W.; Schlichting, W.; Schmerl, B.; Smith, D. B.; Sousa, J. P.; Tamura, G.; Tahvildari, L.; Villegas, N. M.; Vogel, T.; Weyns, D.;

Wong, K.; and Wuttke, J. (2011). “Software Engineering for Self-Adpaptive Systems: A second Research Roadmap”. In *Proceedings of Dagstuhl Seminar, Software Engineering for Self-Adaptive Systems*, (Dagstuhl Seminar ’11), pages 1–26, Dagstuhl, Germany.

[Ding et al., 2009] Ding, B.; Wang, H.; Shi, D.; and Rao, X. (2009). “Towards Unanticipated Adaptation: An Architecture-Based Approach”. In *Proceedings of the Seventh International Conference on Software Engineering Research, Management and Applications*, (SERA ’09), pages 103–109, Haikou, China.

[Dowling and Cahill, 2001] Dowling, J. and Cahill, V. (2001). “The K-Component Architecture Meta-Model for Self-Adaptive Software”. In *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, (Reflection ’01), pages 81–88, 2001, Kyoto, Japan.

[ECA OMG, 2004] ECA OMG (2004). “Enterprise Collaboration Architecture (ECA) Specification”. `http://www.omg.org/`.

[Eclipse, 2010] Eclipse (2010). “Eclipse Modelling Framework”. `http://www.eclipse.org/modeling/emf/`. [Online; accessed 1-November-2010].

[Estell, 1976] Estell, R. G. (1976). “Software life cycle management”. *International Journal of Management Reviews*, 5, pp. 2–15.

[Filman et al., 2004] Filman, R. E.; Elrad, T.; Clarke, S.; and Akşit, M., editors (2004). *Aspect-Oriented Software Development*. Addison-Wesley.

[Floch et al., 2006] Floch, J.; Hallsteinsen, S.; Stav, E.; Eliassen, F.; Lund, K.; and Gjorven, E. (2006). “Using architecture models for runtime adaptability”. *IEEE software*, 23(2), pp. 62–70.

[France and Rumpe, 2007] France, R. and Rumpe, B. (2007). “Model-driven Development of Complex Software: A Research Roadmap”. In *Proceedings of the Future Of Software Engineering*, (FOSE ’07), pages 37–54, Washington, DC, USA.

[Galorath and Evans, 2006] Galorath, D. D. and Evans, M. W. (2006). *Software Sizing, Estimation, and Risk Management.* Auerbach Publications.

[Ganek and Corbi, 2003] Ganek, A. and Corbi, T. (2003). "The dawning of the autonomic computing era". *IBM Systems Journal*, 42(1), pp. 5–18.

[Garlan et al., 2004] Garlan, D.; Cheng, S.; Huang, A.; Schmerl, B.; and Steenkiste, P. (2004). "Rainbow: Architecture-based self-adaptation with reusable infrastructure". *Computer*, 37(10), pp. 46–54.

[Gassanenko, 1998] Gassanenko, M. (1998). "Context-oriented programming". In *Proceedings of the European Forth Conference*, (EuroFORTH '93), pages 1–14, Marianske Lazne (Marienbad), Czech Republic.

[Geihs et al., 2011] Geihs, K.; Evers, C.; Reichle, R.; Wagner, M.; and Khan, M. U. (2011). "Development support for QoS-aware service-adaptation in ubiquitous computing applications". In *Proceedings of ACM Symposium on Applied Computing*, (SAC '11), pages 197–202, TaiChung, Taiwan.

[Geihs et al., 2006] Geihs, K.; Reichle, R.; Khan, M. U.; Solberg, A.; and Hallsteinsen, S. (2006). "Model-driven development of self-adaptive applications for mobile devices: (research summary)". In *Proceedings of the international workshop on Self-adaptation and Self-managing Systems*, (SEAMS '06), pages 95–95, Shanghai, China.

[Grassi and Sindico, 2007] Grassi, V. and Sindico, A. (2007). "Towards model driven design of service-based context-aware applications". In *Proceedings of the International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE*, (ESSPE '07), pages 69–74, New York, NY, USA.

[Grimm, 2004] Grimm, R. (2004). "One.world: Experiences with a Pervasive Computing Architecture". *IEEE Pervasive Computing*, 3, pp. 22–30.

[Harrison, 2011] Harrison, W. (2011). "Modularity for the changing meaning of changing". In *Proceedings of the tenth international conference on Aspect-oriented software development*, (AOSD '11), pages 301–312, Porto de Galinhas, Brazil.

[Hinchey and Sterritt, 2005] Hinchey, M. and Sterritt, R. (2005). "Self-managing software". *IEEE Computer*, 39(2), pp. 107–109.

[Hirschfeld et al., 2008] Hirschfeld, R.; Costanza, P.; and Nierstrasz, O. (2008). "Context-Oriented Programming". *Journal of Object Technology*, 7(3), pp. 125–151.

[Hochmuller, 1999] Hochmuller, H. (1999). "Towards the Proper Integration of Extra-Functional Requirements". *Australasian Journal of Information Systems*, 6(2), pp. 98–117.

[Horn, 2001] Horn, P. (2001). "Autonomic computing: IBM's Perspective on the State of Information Technology". Technical report, IBM.

[Hundt et al., 2010] Hundt, C.; Stöhr, D.; and Glesner, S. (2010). "Optimizing aspect-oriented mechanisms for embedded applications". In *Proceedings of the 48th international conference on Objects, models, components, patterns*, (TOOLS'10), pages 137–153, Malaga, Spain.

[Inverardi, 2007] Inverardi, P. (2007). "Software of the future is the future of software". In *Proceedings of the 2nd international conference on Trustworthy global computing*, (TGC'06), pages 69–85, Lucca, Italy.

[Inverardi and Tivoli, 2009] Inverardi, P. and Tivoli, M. (2009). "The Future of Software: Adaptation and Dependability". In Lucia, A. and Ferrucci, F., editors, *Software Engineering*, pages 1–31.

[Jarke, 1998] Jarke, M. (1998). "Requirements tracing". *Communications of the ACM*, 41(4), pp. 32–36.

[Kapitsaki et al., 2009] Kapitsaki, G.; Prezerakos, G.; Tselikas, N.; and Venieris, I. (2009). "Context-aware service engineering: A survey". *Journal of Systems and Software*, 82(8), pp. 1285–1297.

[Kazman et al., 2002] Kazman, R.; Klein, M.; and Clements, P. (2002). *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley Professional, 2nd edition.

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

[Keeney, 2004] Keeney, J. (2004). *Completely unanticipated dynamic adaptation of software.* PhD thesis, School of Computer Science and Statistics, Trinity College of Dublin.

[Keung et al., 2010] Keung, J. W.; Liu, Y.; Foster, K.; and Nguyen, T. (2010). "A Statistical Method for Middleware System Architecture Evaluation". In *Proceedings of the Software Engineering Conference*, (ASWEC 10), pages 183–191, Auckland, Australian.

[Khan, 2010] Khan, M. U. (2010). *Unanticipated Dynamic Adaptation of Mobile Applications.* PhD thesis, University of Kassel, Distributed Systems Group, Kassel, Germany.

[Khattak and Barrett, 2009] Khattak, Y. and Barrett, S. (2009). "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)*, (CAMS '09), pages 24–30, Dublin, Ireland.

[Kiczales et al., 2001] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; and Griswold, W. (2001). "An Overview of AspectJ". In *Proceedings of the 15th European Conference on Object-Oriented Programming, (ECOOP 2001)*, volume 2072 of *LNCS*, pages 327–354, Budapest, Hungary.

[Kiczales et al., 1997] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; and Irwin, J. (1997). "Aspect-oriented programming". In *Proceedings of the European Conference of Object-Oriented Programming, (ECOOP '01)*, volume 1241 of *LNCS*, pages 220–242. Budapest, Hungary.

[Kitchenham et al., 2002] Kitchenham, B.; Linkman, S.; and Law, D. (2002). "DESMET: a methodology for evaluating software engineering methods and tools". *Computing & Control Engineering Journal*, 8(3), pp. 120–126.

[Kleppe et al., 2003] Kleppe, A. G.; Warmer, J.; and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing, Boston, MA, USA.

[Kuwadekar et al., 2008] Kuwadekar, A.; Balakrishna, C.; and Al-Begain, K. (2008). "GenX-fone - Design and Implementation of Next-Generation Ubiquitous SIP Client". In *Proceedings of the Second International Conference on Next Generation Mobile Applications, Services and Technologies*, (NGMAST '08), Cardiff, UK.

[Kuwadekar et al., 2010] Kuwadekar, A.; Joshi, A.; and Al-Begain, K. (2010). "Real Time Video Adaptation in Next Generation Networks". In *Proceedings of Fourth International Conference on Next Generation Mobile Applications, Services and Technologies (NGMAST 2010)*, volume 1 of *LNCS*, pages 54 –60, Amman, Jordan.

[Laprie, 2004] Laprie, J. (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable and Secure Computing*, 1, pp. 11–33.

[Lattice Business Software, 2010] Lattice Business Software (2010). "A Template-Based Model-Driven Code Generation engine supports XSLT.". `http://www.latticesoft.com/`. [Online; accessed 1-April-2011].

[Lau, 2006] Lau, K.-K. (2006). "Software component models". In *Proceedings of the 28th international conference on Software engineering*, (ICSE '06), pages 1081–1082, Shanghai, China.

[Lewis and Wrage, 2005] Lewis, G. and Wrage, L. (2005). "Model problems in technologies for interoperability: Model-driven architecture". Technical report, Software Engineering Institute.

[Lincke et al., 2011] Lincke, J.; Appeltauer, M.; Steinert, B.; and Hirschfeld, R. (2011). "An open implementation for context-oriented layer composition in ContextJS". *Science of Computer Programming*, 76, pp. 1194–1209.

[Loyall et al., 1998] Loyall, J. P.; Bakken, D. E.; Schantz, R. E.; Zinky, J. A.; Karr, D. A.; Vanegas, R.; and Anderson, K. R. (1998). "QoS Aspect Languages and Their Runtime Integration". In *Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, (LCR '98), pages 303–318, London, UK.

[Magableh and Barrett, 2009] Magableh, B. and Barrett, S. (2009). "PCOMs: A Component Model for Building Context-Dependent Applications". In *Proceedings of the First International Conference on Adaptive and Self-adaptive Systems and Applications*, (Adaptive '09), pages 44–48, Athens, Greece.

[Magableh and Barrett, 2010] Magableh, B. and Barrett, S. (2010). "PrimitiveC-ADL: Primitive Component Architecture Description Language". In *Proceedings of the 7th International Conference on Informatics and Systems*, (INFOS 2010), pages 103–118, Cairo. Egypt.

[Magableh and Barrett, 2011] Magableh, B. and Barrett, S. (2011). "Adaptive Context Oriented Component-Based Application Middleware (COCA-Middleware)". In *Proceedings of the 8th International Conference of Ubiquitous Intelligence and Computing, (UIC 2011)*, volume 6905 of *Lecture Notes in Computer Science*, pages 137–151, Banff, Canada.

[McKinley et al., 2004] McKinley, P. K.; Sadjadi, S. M.; Kasten, E. P.; and Cheng, B. H. C. (2004). "Composing Adaptive Software". *Journal of Computer*, 37, pp. 56–64.

[Mezini and Ostermann, 2004] Mezini, M. and Ostermann, K. (2004). "Variability management with feature-oriented programming and aspects". In *Proceedings of the 12th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, (SIGSOFT '04), pages 127–136, Newport Beach, CA, USA.

[Mikalsen et al., 2006] Mikalsen, M.; Paspallis, N.; Floch, J.; Stav, E.; Papadopoulos, G.; and Chimaris, A. (2006). "Distributed context management in a mobility and adaptation enabling middleware (madam)". In *Proceedings of the 2006 ACM symposium on Applied computing*, (SAC '06), pages 733–734, Dijon, France.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). "MDA Guide Version 1.0.1". Technical report, Object Management Group (OMG).

[Mukhija and Glinz, 2005] Mukhija, A. and Glinz, M. (2005). "The CASA approach to autonomic applications". In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks*, (ASWN 2005), pages 173–182, Paris, France.

[Munnelly et al., 2007] Munnelly, J.; Fritsch, S.; and Clarke, S. (2007). "An Aspect-Oriented Approach to the Modularisation of Context". In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, (PERCOMW '07), pages 114–124, Seattle, WA, USA.

[Nicoara and Alonso, 2005] Nicoara, A. and Alonso, G. (2005). "Dynamic AOP with prose". In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications*, (ASMEA 2005), pages 125–138, Porto, Portugal.

[Noble et al., 1997] Noble, B. D.; Satyanarayanan, M.; Narayanan, D.; Tilton, J. E.; Flinn, J.; and Walker, K. R. (1997). "Agile application-aware adaptation for mobility". *ACM Operating Systems Review*, 31, pp. 276–287.

[Norman, 2002] Norman, D. A. (2002). *The Design of Everyday Things*. Basic Books, reprint paperback edition.

[OMG, 2010] OMG, O. M. G. (2010). "Software & Systems Process Engineering Meta-Model Specification". Technical report, Object Management Group.

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

[OSGI framework, 2010] OSGI framework (2010). "OSGi - The Dynamic Module System for Java". `http://www.osgi.org/Main/HomePage`. [Online; accessed 1-November-2010].

[Papazoglou et al., 2007] Papazoglou, M.; Traverso, P.; Dustdar, S.; and Leymann, F. (2007). "Service-Oriented Computing: State of the Art and Research Challenges". *IEEE Journal of Computer*, 40(11), pp. 38–45.

[Parashar and Hariri, 2005] Parashar, M. and Hariri, S. (2005). "Autonomic computing: An overview". *Unconventional Programming Paradigms*, pages 257–269.

[Parikh, 2005] Parikh, T. S. (2005). "Using Mobile Phones for Secure, Distributed Document Processing in the Developing World". *IEEE Pervasive Computing*, 4, pp. 74–81.

[Paspallis, 2009] Paspallis, N. (2009). *Middleware-based development of context-aware applications with reusable components.* PhD thesis, University of Cyprus, Department of Computer Science.

[Paspallis, 2010] Paspallis, N. (2010). "Software Engineering Support for the Development of Context-aware, Adaptive Applications for Mobile and Ubiquitous Computing Environments". `http://www.cs.ucy.ac.cy/~paspalli/phd/thesis-proposal.pdf`. "[online accessed 1-December-2010]".

[Popovici et al., 2002] Popovici, A.; Gross, T.; and Alonso, G. (2002). "Dynamic weaving for aspect-oriented programming". In *Proceedings of the 1st international conference on Aspect-oriented software development*, (AOSD '02), pages 141–147, Enschede, The Netherlands.

[Preece et al., 2002] Preece, J.; Rogers, Y.; and Sharp, H., editors (2002). *Interaction Design: Beyond Human-Computer Interaction*. John Wiley and Sons.

[Prezerakos et al., 2007] Prezerakos, G.; Tselikas, N.; and Cortese, G. (2007). "Model-driven composition of context-aware web services using ContextUML and aspects". In *Proceedings of the IEEE International Conference on Web Services*, (ICWS 2007), pages 320–329, Utah, USA.

[Reichle et al., 2008] Reichle, R.; Wagner, M.; Khan, M. U.; Geihs, K.; Lorenzo, J.; Valla, M.; Fra, C.; Paspallis, N.; and Papadopoulos, G. A. (2008). "A comprehensive context modeling framework for pervasive computing systems". In *Proceedings of the 8th international conference on Distributed applications and interoperable systems*, (DAIS '08), pages 281–295, Oslo, Norway.

[Robertson and Laddaga, 2005] Robertson, P. and Laddaga, R. (2005). "Model Based Diagnosis and Contexts in Self Adaptive Software". In Babaoglu, O.; Jelasity, M.; Montresor, A.; Fetzer, C.; Leonardi, S.; van Moorsel, A.; and van Steen, M., editors, *Proceedings of the Conference on Self-star Properties in Complex Information Systems*, volume 3460 of *LNCS*, pages 403–403.

[Rouvoy et al., 2009] Rouvoy, R.; Barone, P.; Ding, Y.; Eliassen, F.; Hallsteinsen, S.; Lorenzo, J.; Mamelli, A.; and Scholz, U. (2009). "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments". In Cheng, B. H.; Lemos, R.; Giese, H.; Inverardi, P.; and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, pages 164–182.

[Rouvoy et al., 2008a] Rouvoy, R.; Beauvois, M.; Lozano, L.; Lorenzo, J.; and Eliassen, F. (2008a). "MUSIC: an autonomous platform supporting self-adaptive mobile applications". In *Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device*, (MobMid '08), pages 6:1–6:6, Leuven, Belgium.

[Rouvoy et al., 2008b] Rouvoy, R.; Eliassen, F.; Floch, J.; Hallsteinsen, S.; and Stav, E. (2008b). "Composing components and services using a planning-based adaptation middleware". In *Proceedings of the 7th international conference on Software composition*, (SC '08), pages 52–67, Budapest, Hungary.

[Salehie and Tahvildari, 2005] Salehie, M. and Tahvildari, L. (2005). "A Policy-Based Decision Making Approach for Orchestrating Autonomic Elements". In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, (STEP '05), pages 173–181, Budapest, Hungary.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). "Self-adaptive software: Landscape and research challenges". *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4, pp. 14:1–14:42.

[Salvaneschi et al., 2011] Salvaneschi, G.; Ghezzi, C.; and Pradella, M. (2011). "Context-Oriented Programming: A Programming Paradigm for Autonomic Systems". Technical report, Cornell University.

[Schneidewind, 1998] Schneidewind, N. (1998). "IEEE Standard for a Software Quality Metrics Methodology". *IEEE standard 1061-1992*, pages 155–165.

[Schuster et al., 2011] Schuster, K.; Appeltaue, M.; and Hirschfeld, R. (2011). "Context-oriented Programming for Mobile Devices: JCop on Android". In *Proceedings of the Workshop on Context-oriented Programming (COP) 2011, co-located with ECOOP 2011*, pages 180–195, Lancaster, UK,.

[Sen and Roman, 2003] Sen, R. and Roman, G. (2003). "Context-Sensitive Binding, Flexible Programming Using Transparent Context Maintenance". Technical report, Department of Computer Science and Engineering Washington University in St. Louis.

[Sheng and Benatallah, 2003] Sheng, Q. and Benatallah, B. (2003). "ContextUML: a UML-based modeling language for model-driven development of context-aware web services".

In *Proceeding of the 4th International Conference on Mobile Business*, (ICMB '05), pages 11–13, Sydney, Australia.

[Sindico et al., 2008] Sindico, A.; Bartolomeo, G.; Grassi, V.; and Salsano, S. (2008). "Design and development of a context oriented language for middleware based applications". In *Proceedings of the 2008 workshop on Next generation aspect oriented middleware*, (NAOMI '08), pages 1–5, Brussels, Belgium.

[Sindico and Grassi, 2009] Sindico, A. and Grassi, V. (2009). "Model driven development of context aware software systems". In *Proceedings of International Workshop on Context-Oriented Programming*, (COP '09), pages 7:1–7:5, Genova, Italy.

[SPARX Enterprise Architecture, 2010] SPARX Enterprise Architecture (2010). "Enterprise Architect 8". `http://www.sparxsystems.com.au/`. [Online; accessed 1-December-2010].

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman, Boston, MA, USA, 2nd edition.

[Tanter, 2006] Tanter, É. (2006). "Aspects of composition in the reflex aop kernel". In *Proceedings of the 5th International Symposium on Software Composition*, (SC 2006), pages 99–114, Vienna, Autriche.

[Tanter et al., 2006] Tanter, É.; Gybels, K.; Denker, M.; and Bergel, A. (2006). "Context-Aware Aspects". In *Proceedings of the 5th International Symposium on Software Composition*, (SC 2006), pages 227–242, Vienna, Autriche.

[Tarvainen, 2007] Tarvainen, P. (2007). "Adaptability Evaluation of Software Architectures; A Case Study". In *Proceedings of the 31st Annual International Computer Software and Applications Conference, (COMPSAC '07)*, volume 02 of *LNCS*, pages 579–586, Beijing, China.

[Tesoriero et al., 2010] Tesoriero, R.; Gallud, J.; Lozano, M.; and Penichet, V. (2010).

"CAUCE: Model-driven Development of Context-aware Applications for Ubiquitous Computing Environments". *Journal of Universal Computer Science*, 16(15), pp. 2111–2138.

[Valetto et al., 2001] Valetto, G.; Kaiser, G. E.; and Kc, G. S. (2001). "A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems". In *Proceedings of the 8th European Workshop on Software Process Technology*, (EWSPT '01), pages 102–116, London, UK.

[Visual Paradigm, 2010] Visual Paradigm (2010). "Visual Paradigm for UML". `http://www.visual-paradigm.com/`. [Online; accessed 1-April-2011].

[Wagner et al., 2011] Wagner, M.; Reichle, R.; Khan, M. U.; and Geihs, K. (2011). "Software Development Method for Adaptive Applications in Ubiquitous Computing Environments". Technical report, IST-MUSIC. [Online; accessed 1-March-2011].

[Yang et al., 2009] Yang, J.; Huang, G.; Zhu, W.; Cui, X.; and Mei, H. (2009). "Quality attribute tradeoff through adaptive architectures at runtime". *The Journal of Systems & Software*, 82(2), pp. 319–332.