2011-09-15

Productivity Evaluation of Self-Adaptive Software Model Driven Architecture

Basel Magableh
Dublin Institute of Technology, 453543@dit.ie

Recommended Citation

Magableh, B., Barrett, S.: Productivity Evaluation of Self-Adaptive Software Model Driven Architecture, *International Journal of Information Technology and Web Engineering (IJETWE), 4(2), pages 172-180*. doi:10.4018/jitwe.2011100101

International Journal of Information Technology and Web Engineering

Productivity Evaluation of Self-adaptive Software Model Driven Architecture

Basel Magableh, Stephen Barrett
Distributed Systems Group,
School of Computer Science and Statistics
Trinity College Dublin, Ireland
Emails: magablb@scss.tcd.ie, stephen.barrett@scss.tcd.ie

Abstract

Anticipating context changes using a model-based approach requires a formal procedure for analysing and modelling their context-dependent functionality, and a stable description of the architecture which supports dynamic decision-making and architecture evolution. This article demonstrates the capabilities of the context-oriented component-based application-model-driven architecture (COCA-MDA) to support the development of self-adaptive applications; we describe a state-of-the-art case study and evaluate the development effort involved in adopting the COCA-MDA in constructing the application. An intensive analysis of the application requirements simplified the process of modelling the application's behavioural model; therefore, instead of modelling several variation models, the developers modelled an extra-functionality model. COCA-MDA reduces the development effort because it maintains a clear separation of concerns and employs a decomposition mechanism to produce a context-oriented component model which decouples the applications' core functionality from the context-dependent functionality. Estimating the MDA approach's productivity can help the software developers to select the best MDA-based methodology from the available solutions proposed in the literature. Thus, counting the source line of code is not adequate for evaluating the development effort of the MDA-based methodology. Quantifying the maintenance adjustment factor of the new, adapted, and reused code is a better estimate of the development effort of the MDA approaches.

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, Wang, & Lau, 2003). This presents the software developers with the challenge of tailoring behavioural variations both to each specific user need and to the context information. Context-dependent behavioural variations can be seen as a collaboration of individual features expressed in requirements, design, and implementa-

tion. Before encapsulating the crosscutting context-dependent behaviours into a software module, the developers must first identify them both in the requirements document and in the software model. 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. Using intuition or even domain knowledge is not necessarily sufficient for identifying the behavioural variations; instead, it requires a formal analysis procedure for the software requirements and a separation of their individual concerns. Moreover, a formal procedure for modelling these variations is needed. This kind of analysis and modelling procedure can reduce the complexity in modelling self-adaptive applications and encapsulate the context-dependent part of the distinct architecture module (component).

In this sense, a context oriented component model (COCA-component) (Magableh & Barrett, 2009) is used to encapsulate behavioural variations and decouple them from the application's core functionality. In this way, dynamic component composition is achieved. Additionally, from the software developer's perspective, it is vital to know the productivity of the development paradigm which might be used in constructing the self-adaptive application. Productivity evaluation of model-driven approaches can assist the developers in selecting among the proposed methodologies in the literature which approach dynamic behavioural variations of self-adaptive software. Context-oriented component-based application-model-driven architecture (COCA-MDA) emerged as a development paradigm which facilitates the development of self-adaptive context-oriented software (Magableh & Barrett, 2011b, 2011c).

This article evaluates the development effort involved in adopting COCA-MDA when constructing a self-adaptive application for an indoor wayfinding application (IWayfinder) targeting individuals with cognitive impairments. The development effort of COCA-MDA is compared to other model-driven approaches proposed in the literature.

The remainder of the article is structured as follows. Section provides a comparative analysis of related studies. Section demonstrates a case study of a self-adaptive application. The COCA-MDA phases are described in Section . Section provides a COCA-MDA evaluation using constructive cost model II (COCOMO II). Section summarizes the research findings and describes directions for future work.

Related Work

In the literature, there are several MDA approaches which target the development of self-adaptive applications for mobile computing environments which produce component-based applications; this study borrows from the following methodologies: MUSIC, proposed by (Wagner, Reichle, Khan, & Geihs, 2011); U-MUSIC (Khan, 2010); and Paspallis MDA (Paspallis, 2009).

The MUSIC development methodology (Wagner et al., 2011) adopts a model-driven approach to constructing the application variability model. The applications are built as a component framework with component types as variation points. Middleware is used to resolve the variation points, which involves the election of a concrete component as a realization of the component type. Using this method, a number of application variants can automatically be derived.

The U-MUSIC methodology, proposed by (Khan, 2010), adopts the model-driven approach to constructing self-adaptive applications and enabling dynamic unanticipated adaptation based on a component model. The U-MUSIC system enables the 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 using an abstract, platform-independent model (PIM).

Paspallis (Paspallis, 2009) proposes another MDA-based methodology which considers the context providers for the application. For each context provider, a plug-in is proposed 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 load into the application), middleware support performs dynamic runtime loading of the plug-in. However, it is impossible to consider all the context providers which might produce context information at runtime.

In MUSIC, U-MUSIC, and Paspallis approaches, dynamic decision making is supported by a utility function. The utility function is defined as the weighted sum of the different objectives based on user preferences and QoS. However, this approach suffers from a number of drawbacks. First, it is well known that correct identification of the weight of each goal is a major difficulty. Second, the approach hides conflicts among multiple goals in a single, aggregate objective function, rather than exposing the conflicts and reasoning about them. At runtime, a utility function is used to select the best application variant; this is the so-called 'adaptation plan'. Potentially, it is impossible for the developer to predict all possible variations of the application when unanticipated conditions 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. Because of the above issue, it is impossible to use MUSIC to provide unanticipated adaptation in a self-adaptive application. Moreover, modelling the application using U-MUSIC, MUSIC, and Paspallis's MDA produces an architecture with a tight coupling between the architecture and the target platform.

Lewis et al. (Lewis & 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 supported 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. 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, Dascalu, & Egbert, 2008). COCOMO II allows estimation of the effort, time, and cost required for software development. The main advantage of this model is that COCOMO II is an open model with various parameters which affect the estimation of the development effort. Moreover, the COCOMO II model allows estimation of the software

application development effort in both person-months (PM) and time to develop (TDEV). A set of inputs such as software scale factors (SFs) and 17 effort multipliers is needed. A full description of these parameters 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).

Self-adaptive Indoor Wayfinding Application for Individuals with Cognitive Impairments

IWayFinder provides distributed cognition support for indoor navigation to persons with cognitive disabilities. RFID tags and 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 engine provides the required navigation information and instructs the user. 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. 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 which 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. The IWayFinder application and the COCA-MDA development methodology were fully described in our previous work (Magableh & Barrett, 2011b, 2011c). This article focuses on describing an evaluation of the cost effectiveness when adopting the COCA-MDA (among other MDA approaches) in developing the IWayFinder application. Assuming that the context information is delivered by the Cisco infrastructure, we propose the following anticipation scenarios:

A1: Self-tuning The application must track the user's path inside the building. When decision points (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 considering 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 adopts 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 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 in the building.

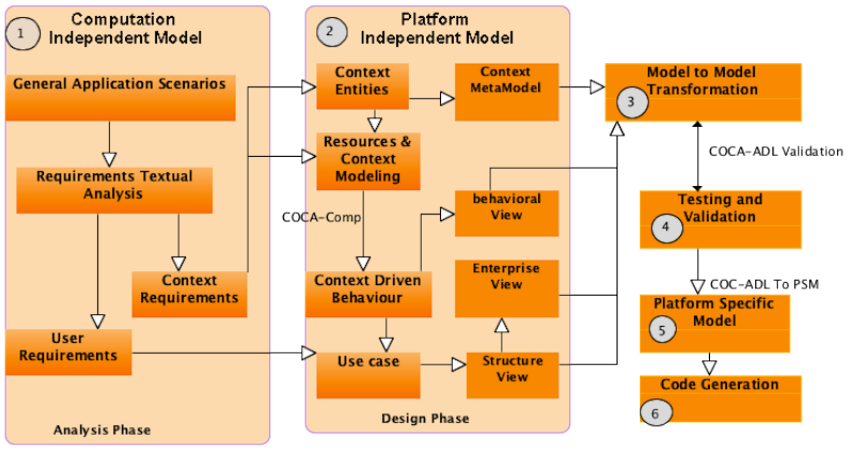


Figure 1. : Context-Oriented Component-Based Application-Model-Driven Architecture (COCA-MDA)

COCA-MDA Development Approach

The COCA-MDA follows the principles of the object management group (OMG) model-driven architecture. The design of a context-aware application according to the COCA-MDA approach generally involves the six phases shown in Figure 1. Modelling IWayfinder using COCA-MDA can be summarised as shown in Figure 2. 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. This phase is accomplished by performing the following three tasks.

*. Task 1. Requirement capturing by textual analysis: In this task, the developer identifies the candidate requirements for the illustration scenario using a textual analysis of the application scenario. It is recommended that the developer identify the candidate actors, use-cases, classes, and activities. This can be achieved by creating a table which lists the results of the analysis.

*. Task 2. Identifying the extra-functional requirements and relating them to the middleware functionality: The requirement is classified in the requirements' diagram, based on its type and whether it comes from a context provider or a consumer. The next level of requirements classification is to classify the requirements based on their anticipation level: 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. For example, displaying the direction output in the camera browser is a functional requirement which drives the extra-functional requirement number 4, 'utilise the resources', which requires a

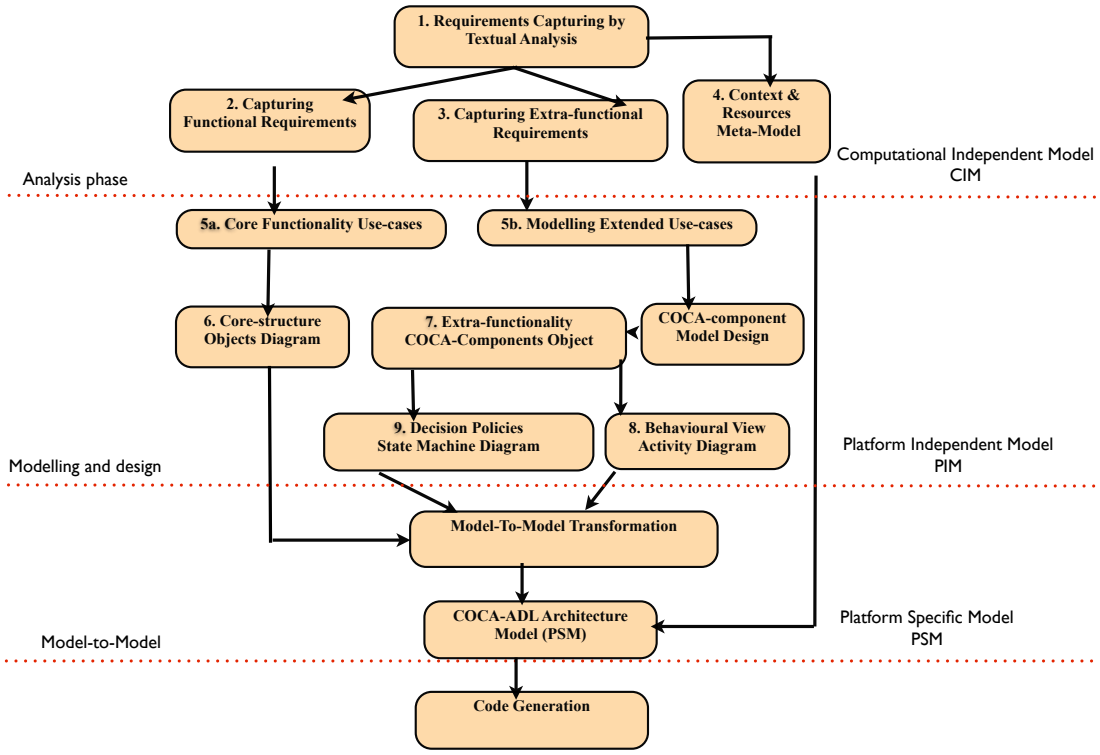


Figure 2. : Modelling Tasks

middleware functionality to manage the context changes and take the adaptation actions which satisfy it. This requirement is classified as the foreseeable anticipation level.

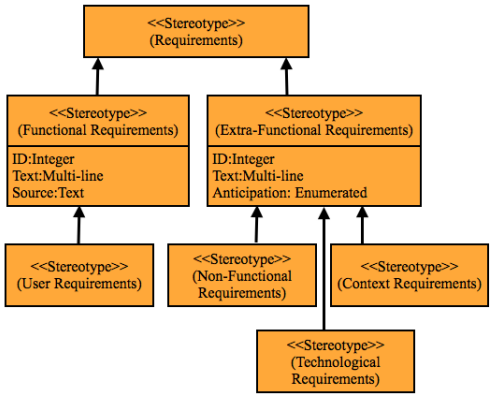


Figure 3. : Requirements UML Profile

*. 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. This task allows the developers to analyse the main functions of the application which achieve specific goals

or objectives. Normally, this kind of requirement is expressed by 'The user must be able to do ...'.

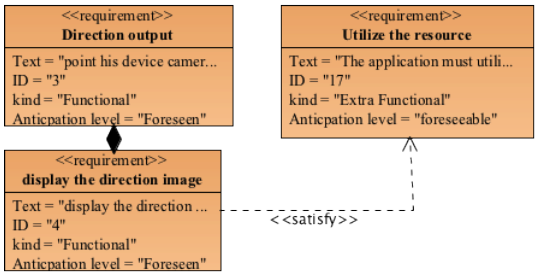


Figure 4. : Partial Requirements Diagram

Modelling and design: COCA-MDA has adopted the component collaboration architecture (CCA) (ECA OMG, 2004) at the PIV phase by partitioning the software into two views: the structure view and the behaviour view. The structure view focuses on the core components of the self-adaptive application and hides the context-driven components. The behaviour viewpoint focuses on modelling the context-driven behaviour of the component, which may be invoked in the application execution at runtime. To achieve this function, the following three tasks are performed.

*. Task 4. Resources and context entity model Resources and context model refers to a generic overview of the underlying device's resources, sensors, and logical context providers. This diagram models the engagement between the resources and the application under development. It facilitates the developer in understanding the relationship between the context providers and their dependency.

*. Task 5. Use cases: 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 actor. The system-dependent and environment-dependent behaviours are modelled as extensions 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 object diagram which describes the application's behavioural view. For example, the 'adapt the direction output' use-case is a contextually driven use-case which extends the application functionality to utilise the devices' resources so as to provide a route to the nearest fire exit.

*. Task 6. Modelling the application core structure: In this task, a classical class diagram models the components which provide the application's core functions. These functions are identified in the use-case diagram in the previous task. However, the class diagram is modelled independently of the variations in the context information. In this scenario, some classes, such as 'Displaying POI's', 'Route-Planning UI, CameraUI, MapUI, and User Interface', are classified to be in the application core. These classes provide the core functions for the user during his tour of Petra. Figure 5 shows the core-structure class model without any interaction with the context environment or the middleware.

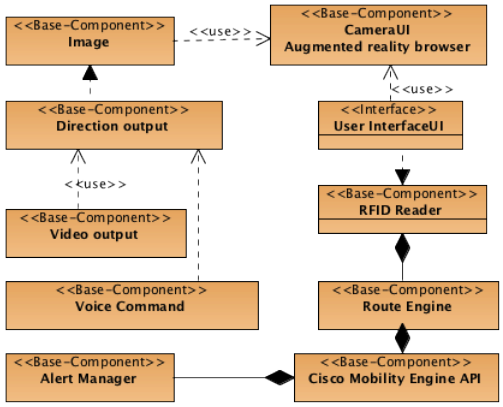


Figure 5. : IWayFinder Core-Classes Structure

Task 7. Identifying Application Variant Behaviour (Behaviour View):

The use-case diagram is split into two distinct object diagrams. The first diagram describes the basic application components which 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, which can be used to anticipate context changes. Figure 6 shows a COCA-component modelled to anticipate the 'direction output'. The COCA-component implements delegate objects and sub layers; each layer implements a specific context-dependent function. The COCA-middleware (Magableh & Barrett, 2009, 2011a), uses this delegate object to redirect the execution among the sub layers based on the context condition.

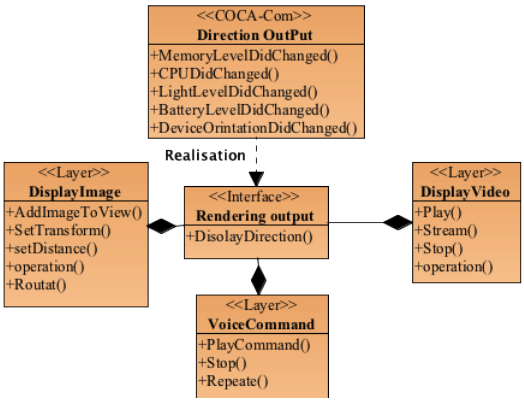


Figure 6. : Direction Output Context Oriented Component

The application behavioural model is used to demonstrate the decision points in the execution which might be reached whenever internal or external variables are found. This decision point requires several parameter inputs to make the correct choice at this critical

time. Using the activity diagram, the developers can extract numerous decision polices. Each policy must be modelled in a state diagram: textbfPolicy: Direction output. This policy is attached to the 'direction output' COCA-component in Figure 6. The policy syntax can be described by the code shown in Listing 1.

Listing 1: 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( BatteryLevel < 20)
   then VoiceCommand();
```

The variant behaviour model is supported by a state-machine model which describes the application decision polices. The three models of the application are used as input for the next phase, model-to-model transformation.

Model-to-model transformation: The platform-independent model and behavioural model are translated into architecture description language (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 (an extensible XML language). ArchStudio is an environment of integrated tools for modelling, visualizing, analysing, and implementing software and systems architectures. The ArchStudio provides Archipelago as the graphical editor used to model the architecture. Archipelago was used to extend the xADL by implementing the COCA-ADL meta-model. The ArchStudio editor enables the developer to model their application using three distinct models: structure, state machine, and activity diagram (Dashofy et al., 2007).

Testing and validating: Tests 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 PSM may provide more or fewer details, depending on its purpose. A PSM will be an implementation if it provides all the information needed to construct a system and to put it into operation. Alternatively, it may act as a PIM used to further refine the PSM so that it can be directly implemented.

Code generation: Model-to-text includes model-to-text transformation deployment and execution verification. The COCA-ADL XMI code is transformed into the implementation language.

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 Paspallis (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 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 SLOC; counting UFP; and aggregating new, adapted, and reused code, i.e. 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 PM and the TDEV, taking into consideration the aforementioned inputs, as shown in Equation 1. The primary equation in 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 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.

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

$$TDEV = C \times (PM)^F, \tag{2}$$
$$\text{where } F = D + 0.2 \times (E - B),$$
$$C = 3.67,\ D = 0.28\,(\text{COCOMOII.2000})$$

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 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 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 3, 4, and 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 7, .

$$MAF = 1 + \left(\frac{SU}{100} \times UNFM\right), \tag{3}$$

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

Figure 7. : Rating Scale for Programmer Unfamiliarity (UNFM)

SU: Software Understanding (zero if DM = 0 and CM = 0),
DM: percentage of design modified,
CM: percentage of code modified,
$UNFM = 0.4$

$$PM_M = T_M - FSP_M, \quad \text{where T = 12 months} \tag{4}$$

$$PM = AX(Size)^B + \left\lceil \frac{ASLOC(\frac{AT}{100})}{ATPROD} \right\rceil \tag{5}$$

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

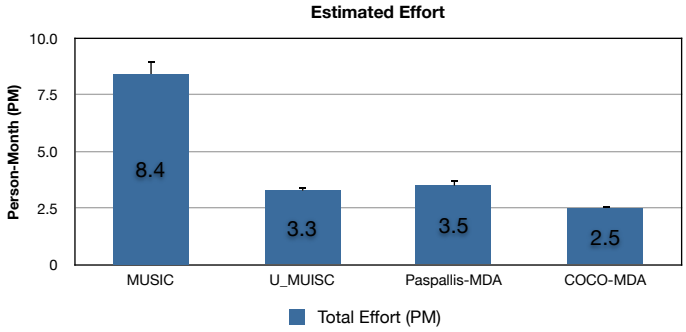
Table 1:: MDA phases and Size factors

In general, MDA-based approaches must apply 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 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.

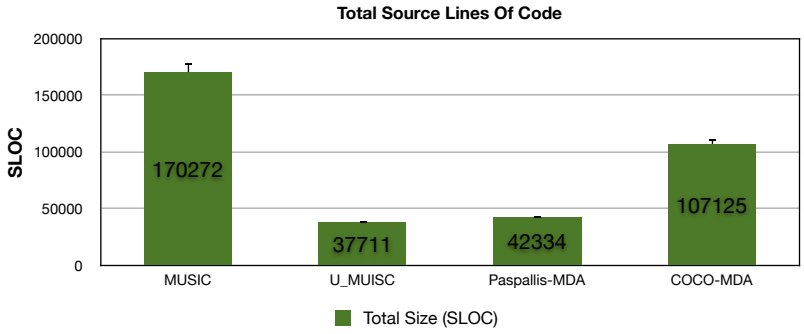
COCOMO II Evaluation Results

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 8 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.



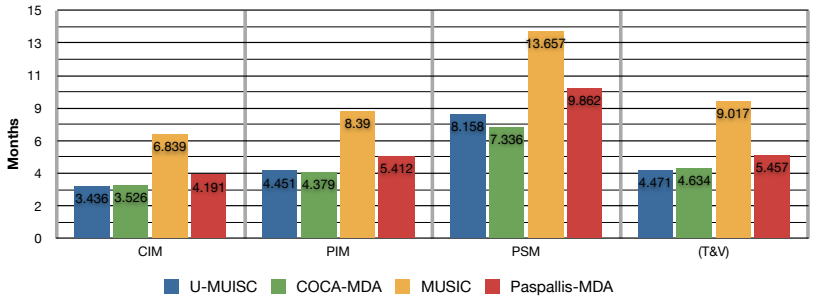
(a) Total Effort for each MDA approach



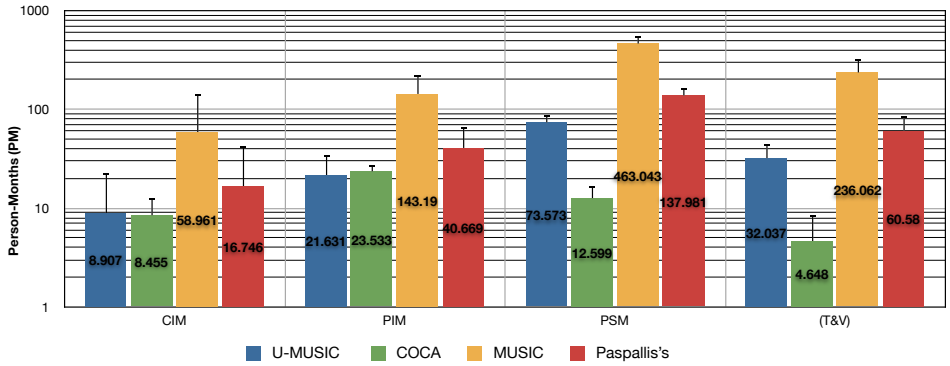
(b) Total Source Lines Of Code

Figure 8. : Total Effort for each MDA approach

Figure 9a provides more information for each MDA in terms of the estimated cost



(a) Estimated cost per phase



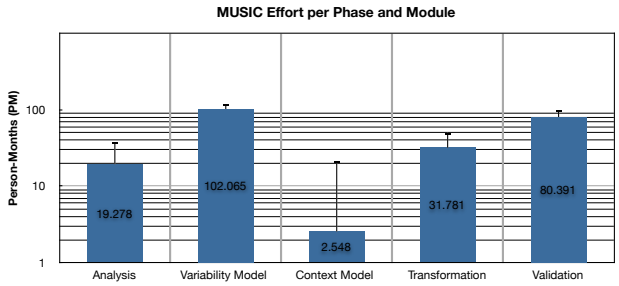
(b) MDA Cumulative effort in person-months

Figure 9. : Cumulative Effort per model/MDA in person-months

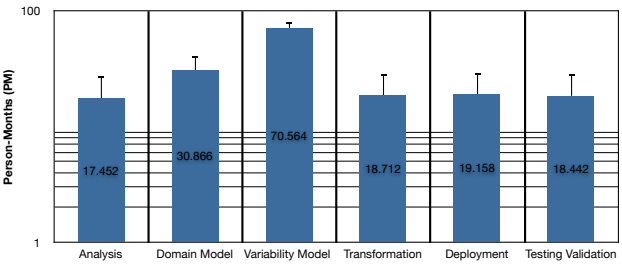
per MDA phase. As shown in the figure, the cost of performing the PIM was large for all MDAs. The reason for this is that all MDAs focus more on modelling the application variation model through the PIM. The cost of adapting the PIM in MUSIC is the largest because of the complexity of adapting the MUSIC 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 MDA comes second, after U-MUSIC, when evaluating the PSM phase.

Figure 9b 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 9b. 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.

In order to provides more information about each MDA approach, we have analysed the effort per phase for each MDA. Figure 10a shows the estimated effort for each phase

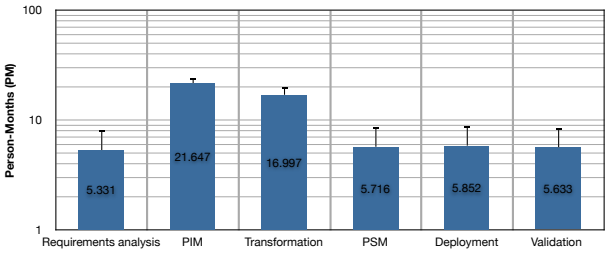


(a) MUSIC Effort (PM) per Phase

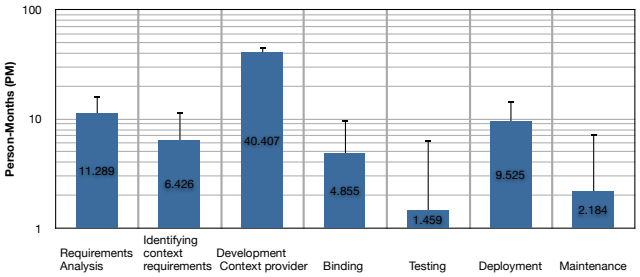


(b) U-MUSIC Effort (PM) per Phase

Figure 10. : MUSIC-MDA and U-MUSIC-MDA estimated efforts



(a) COCA Effort (PM) per Phase



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

Figure 11. : COCA-MDA and Paspallis's MDA estimated efforts

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 10b. The domain 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 11a 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 11a 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.

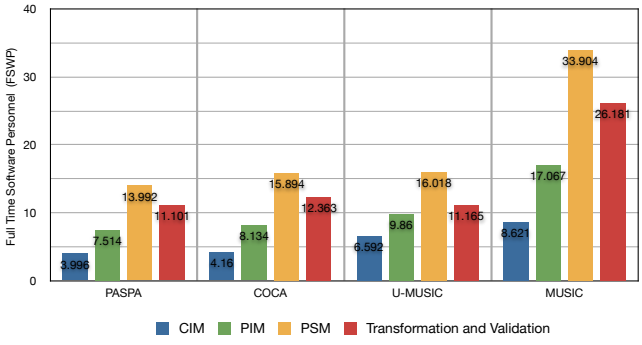


Figure 12. : Project personnel for each phase for each MDA approach

Finally, Figure 12 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.

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.

Conclusions and Future Work

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.

The COCA-MDA requires improvement before it can support requirements reflection and modelling requirements as runtime entities. The requirements reflection mechanism

requires support at the modelling level and architecture level. Requirements reflection can be used to anticipate the evolution of both functional and non-functional requirements.

References

Achilleas. (2010). *Model-driven petri net based framework for pervasive service creation.* Unpublished doctoral dissertation, University of Essex.

Belaramani, N. M., Wang, C.-L., & Lau, F. C. M. (2003, May). Dynamic component composition for functionality adaptation in pervasive environments. In *Proceedings of the the ninth ieee workshop on future trends of distributed computing systems* (pp. 226–232). San Juan, Puerto Rico.

Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, et al. (2000). *Software cost estimation with cocomo ii* (1st ed.). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Calic, T., Dascalu, S., & 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* (pp. 44–50). Istanbul, Turkey.

Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., & Taylor, R. (2007). Archstudio 4: An architecture-based meta-modeling environment. In *Proceedings of the 29th international conference on software engineering* (pp. 67–68).

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

Enterprise collaboration architecture (eca) specification. (2004, Feb). `http://www.omg.org/`.

Khan, M. U. (2010). *Unanticipated dynamic adaptation of mobile applications.* Unpublished doctoral dissertation, University of Kassel, Distributed Systems Group, Kassel, Germany.

Lewis, G., & Wrage, L. (2005). *Model problems in technologies for interoperability: Model-driven architecture* (Tech. Rep.). Software Engineering Institute. url=http://www.sei.cmu.edu/.

Magableh, B., & 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* (pp. 44–48). Athens, Greece.

Magableh, B., & Barrett, S. (2011a, September). Adaptive context oriented component-based application middleware (coca-middleware). In *Proceedings of the 8th international conference of ubiquitous intelligence and computing, (uic 2011)* (Vol. 6905, p. 137-151). Banff, Canada.

Magableh, B., & Barrett, S. (2011b, May). Objective-cop: Objective context oriented programming. In *Proceedings of the first international conference on information and communication systems* (pp. 45–49). Irbid, Jordan.

Magableh, B., & Barrett, S. (2011c, june). Self-adaptive application for indoor wayfinding for individuals with cognitive impairments. In *Proceedings of the 24th international symposium on computer-based medical systems* (p. 1 -6). Bristol, United Kingdom.

Paspallis, N. (2009). *Middleware-based development of context-aware applications with reusable components.* Unpublished doctoral dissertation, University of Cyprus, Department of Computer Science.

Wagner, M., Reichle, R., Khan, M. U., & Geihs, K. (2011, Mar). *Software development method for adaptive applications in ubiquitous computing environments* (Tech. Rep.). IST-MUSIC. `http://www.ist-music.eu/MUSIC/results/music-deliverables/`. ([Online; accessed 1-March-2011])

Basel Magableh received his Ph.D degree in computer science from Trinity College Dublin, Ireland, in 2011. His research focuses in integrating Model Driven Architecture with a component-based system to construct self-adaptive and context-aware software systems. He is a full-time post-

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

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