# Automated generation of pervasive systems architectures: a detailed empirical evaluation

Mostafa A. Hamza* and Sherif G. Aly

Department of Computer Science and Engineering,
The American University in Cairo,
Egypt
Email: hamza@aucegypt.edu
Email: sgamal@aucegypt.edu
*Corresponding author

Maged Elaasar

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Canada
Email: melaasar@gmail.com

**Abstract:** The importance of having mature software development methodologies and tools for the increasingly popular pervasive systems cannot be understated. Focusing on system architectures, we previously conducted a thorough review of over 50 state of the art architectures related to pervasive systems. From the review, we elicited a set of major features that should be supported in pervasive systems, along with best practice architectures for designing such features. We then detailed a methodology, through which designers of new pervasive systems can select a set of desired features and generate a baseline architecture for their system. In this article, we evaluate our methodology with an empirical study that compares generated architectures with ones designed by subject matter experts with sufficient experience in the domain. We used different evaluation suites and measurement techniques in our comparisons. Results show that our automatically generated architectures are very comparable with, and in many cases of higher quality than, the architectures designed by subject matter experts.

**Keywords:** pervasive systems; component-based architecture; evaluation metrics; software product lines; reference architecture; empirical study; automated generation.

**Reference** to this paper should be made as follows: Hamza, M.A., Aly, S.G. and Elaasar, M. (2015) 'Automated generation of pervasive systems architectures: a detailed empirical evaluation', *Int. J. Software Engineering, Technology and Applications*, Vol. 1 No. 1 pp.64–89.

**Biographical notes:** Mostafa A. Hamza is a PhD student in the Department of Computer Science of Calgary University since January 2014. Before starting his PhD journey, he worked as a Life Cycle Manager since 2012 at IBM for different regions within Central Eastern Europe, Middle East and Africa (CEE&MEA). At IBM, he also worked as Enhanced Technical Support Team Leader and Delivery Project Manager for Middle East and North Africa from 2010 until 2013 and Software Support Engineer for AIX, TSM and DB2 from

2009 until 2011. He received his MSc and BS degrees in Computer Science from the American University in Cairo, Egypt, in 2012 and 2007. His research interests include software engineering, software product line architectures, pervasive systems and mobile computing.

Sherif G. Aly is Professor of Computer Science and Engineering and Associate Dean for Graduate Studies and Research at The American University in Cairo. Prior to that, he worked as a senior member of technical staff for general dynamics in Egypt, as a Research Scientist at Telcordia Technologies in New Jersey, and as a Guest Researcher for the Advanced Network Technologies Division of the National Institute of Standards and Technology in Maryland. He is an IEEE senior member and is a recipient of numerous awards. His main research interests involve context awareness, cloud computing, and social networks.

Maged Elaasar is Consulting Software Engineer, Computer Scientist and Adjunct Professor. During his 17-year career in the software industry, he has consulted many companies around the world. He has also become a known leader in the field of applied software engineering; in particular model-driven engineering, where he holds several patents. He has also managed development teams and technically lead software products, like Rational Software Architect, that are in widespread use today. Additionally, he has been an active contributor to open-source projects at Eclipse, like GMF, and a chair of open-standards at OMG, like UML and Diagram Definition. He received his PhD in Electrical and Computer Engineering from Carleton University in 2012, his MSc of Computer Science from Carleton University in 2003, and his BSc of Computer Science from the American University in Cairo in 1996. His main research interests include software engineering, model-driven engineering, semantic web, and big data.

## 1 Introduction

New breeds of pervasive systems are highly characterised by their extreme innovation and ability to sense and react to the surrounding environments in a way that relieves users from interacting with software using unnecessary jargon. According to Mark Weiser, "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" (Weiser, 1991). Such systems are characterised by their significant heterogeneity and the presence of highly mobile devices with limited resources (Yared and Défago, 2003). Examples of resources could be battery lifetime, processing power and storage allocation. They are also characterised by context awareness, intelligent interaction and invisibility as indicated in Ferscha (2003).

Such complex systems continuously challenge software developers with complications of design and implementation. Unfortunately, such challenge is not met with a proportional advancement in the software engineering methodologies that are used to develop these kinds of applications. In fact, much of the utilised development activities for this purpose are based on the reuse of methodologies that have been used in remotely similar types of systems. Eventually, and due to this lack of maturity in development methodologies, emerging applications are not scalable, very expensive, difficult to maintain and of questionable quality. Software development methodologies for this

domain have yet to reach a maturity level that facilitates the development of such systems. Software product line engineering (SPLE) is an example of such methodologies.

Utilising software product line (SPL) concepts in the domain of pervasive systems can ultimately increase the productivity of pervasive systems development and may imminently result in a shorter time-to-market. SPL is divided into domain engineering, application engineering and variability and commonality management as described in Pohl et al. (2005), Hunt (2006) and Bragança and Machado (2005). Domain engineering is used for building the core assets to be used in the product line, while application engineering is used for building the final applications utilising the previously developed assets. Variability and commonality management, on the other hand, are used for configuring the SPL and adding or enhancing the core assets. Frank and others described in more details the SPL life cycle in Van der Liden et al. (2007). Many software engineering approaches have been coupled with SPL approaches to design a reference architecture. Examples of such approaches are aspect oriented programming (Young, 2005), feature-oriented model driven development (Gonzalez, 2007), model driven architectures (Machado et al., 2005), component-based architectures (Heineman and Councill, 2001) and others.

Investigation of related work in this domain led us to the conclusion that a rather modest effort was put into the development of product lines for pervasive systems. In an effort to contribute to this area, we attempted at first to specify a reference architecture to be used in developing pervasive systems (Hamza and Aly, 2010). However, initial efforts made in this area demonstrated that a reference architecture of this kind is significantly large and complex. As an alternative, the researchers herein adopted a bottom up approach in which architects can select from a set of desired features for a given pervasive system, and ultimately, generate a system architecture comparable in quality to those created by subject matter experts (Hamza et al., 2011).

In this article, we present an empirical evaluation of our methodology of generating architectures for pervasive systems. The objective of the evaluation is to assess whether or not our methodology generates architectures comparable to those created by subject matter experts. Two architecture evaluation frameworks were used; namely Narasimhan and Hendradjaya's (2007) evaluation suite, and Zayaraz and Thambidurai's (2008) measurement techniques. The evaluation was carried in an experiment that involved designing architectures for three different pervasive systems. Results show that our generated architectures are comparable to, and in a lot of cases of higher quality than manually defined ones.

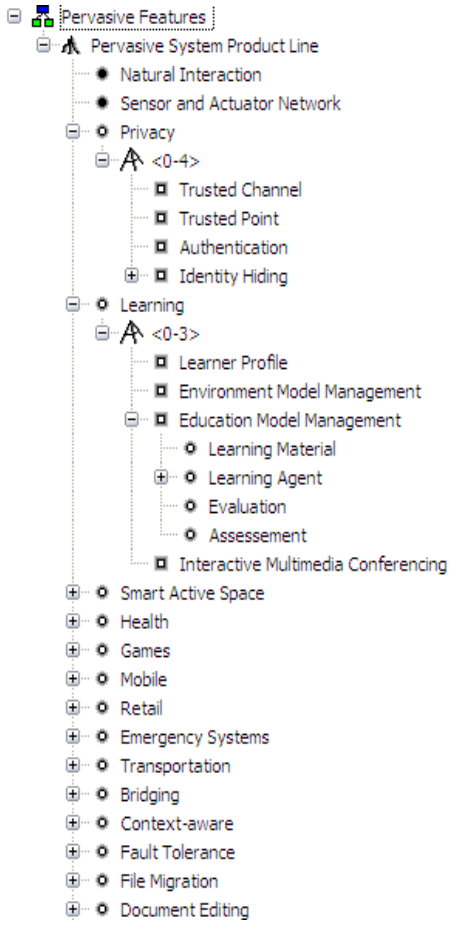
The remainder of this article is structured as follows: in Section 2, we discuss the pervasive systems categorisation and feature models created. In Section 3, we highlight the related work in this area; an overview of our architecture generation methodology is given in Section 4; Section 5 details the case studies used for generating the architectures of the pervasive systems. Section 7 shows the design of our architecture evaluation experiment; the results of the experiment are discussed in Section 8; Section 9 outlines future works; finally, the conclusion is given in Section 10.

## 2 Pervasive systems categorisation and feature model

Pervasive systems are featured by ubiquitous access, context awareness, intelligent interaction and natural interaction. Ubiquitous access refers to sensors and actuators in

the pervasive environment. Context awareness answers the questions of where objects are located, with whom they are interacting and what are the intentions of the surroundings. Intelligent and natural interactions are the ability for the pervasive environment to adapt to the surrounding people's actions and vice versa. A categorisation for extracting the key features and components from pervasive architectures is surveyed in Hamza (2011). The architectures collected are used to build a well-structured categorised reference architecture to be used for designing pervasive systems. The division of the categories is based on the pervasive systems' usage, operating environment and the domain they best fit in. The categorisation is divided into general, bridging, privacy and security, fault tolerance, context-awareness and domain specific architectures as shown in Figure 1.

**Figure 1** Pervasive categorisation (see online version for colours)

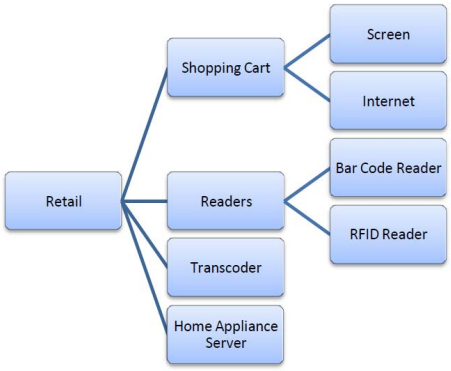



*Source:* Hamza et al. (2011)

Utilising SPLs' reference architecture in building pervasive systems will help in benefiting from commonalities and variability management of components. In Hamza (2011), the most essential features with their commonalities and variabilities are captured from the pervasive architectures mentioned earlier. Natural interaction and sensor and

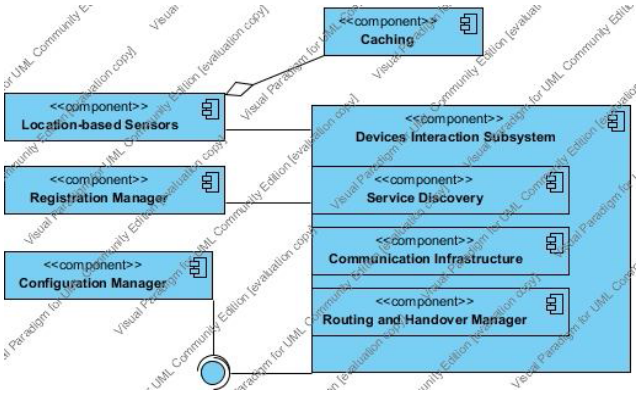
actuator networks are the commonalities that must exist in every pervasive system as shown in Figure 1. The rest could be variabilities that can be chosen or not in a pervasive system. For example, Figure 2 shows retail systems' features in pervasive systems. Retails systems can include a shopping cart that is equipped with either a screen to display different messages and notifications to the user, or internet-access to provide the user a quick way to check online reviews for a certain product. Also, a retail system can include bar code or RFID readers. In some retail pervasive systems, there could be dependency on transcoders as a way of communication with a store's back-end system. Finally, a home appliance server located at a buyer's home that is responsible for checking the availability of the items that the home ran out of, can order them according to the user's preferences and pre-defined settings. A detailed study about the features categorisation is presented in Hamza and Aly (2010).

**Figure 2** Retail features (see online version for colours)




*Source:* Hamza and Aly (2010)

**Figure 3** Sensor network component diagram (see online version for colours)




*Source:* Hamza et al. (2011)

Each feature is then represented by a component or a set of components that provide the functionality of this feature. All those components are collected together in a reference architecture. An example of components for sensors networks from the feature model is shown in Figure 3.

In the next section, we will discuss some the techniques and measurements that we came across and can be applied in our study.

## 3 Related works

We conducted research to find the best suitable evaluation methodologies for the generated architectures. For evaluating software architectures in general, a set of metrics (coupling, cohesion, complexity, size, reusability and others) has been defined by both research and industry. We came across different equations for these metrics in http://grahamberrisford.com/, Allen et al. (2007), Barnard (1998), Briand et al. (1996), Hitz and Montazeri (1995), Liu and Wang (2005), and Pressman (2001). In Briand et al. (1996), for example, coupling which measures the relationship of dependency between two interacting modules is calculated as:

$$C(S,S') = i + \frac{n}{n+1}$$

where $i$ is a number corresponding to the worst coupling type, and n the number of interconnections between $S$ and $S'$, global variables and formal parameters, respectively.

Cohesion evaluates the tightness between the linked features composing a system or module. Interconnected relations are considered cohesive. Three cohesive measurements, NRCI, PRCI and ORCI, are introduced in Briand et al. (1996). They represent Neutral, Pessimistic and Optimistic Ratio of Cohesive Interaction respectively. Using these three measurements help to estimate the non-visible interactions of a module or a software part at the design phase. The following equations are used to measure cohesion:

$$NRCI(sp) = \frac{knownInteractions(sp)}{\#MaxInteractions(sp) - \#UnknownInteractions(sp)}$$

$$PRCI(sp) = \frac{knownInteractions(sp)}{\#MaxInteractions(sp)}$$

$$ORCI(sp) = \frac{knownInteractions(sp) + \#UnknownInteractions(sp)}{\#MaxInteractions(sp)}$$

where *#MaxInteractions*(*sp*) is the maximum number of possible intra-module interactions between the features exported by each module of the software part (*sp*).

System complexity is affected by the dependency relationships between different elements as defined in Briand et al. (1996). It is measured by converting the components and their elements into a graph then using the following formula:

$$v(G) = |R| - |E| + 2p$$

where $G$ represents the graph, $E$ is the number of edges, $R$ is the binary relation between two elements ($E \times E$) and $p$ is the number of connected components of $G$.

Another approach for measuring complexity is configuration complexity (http://grahamberrisford.com/15%20Scale%20and%20Change/Can%20we%20measure%20architecture.htm). Configuration complexity can be applied to any component dependency diagram, entity-relationship model, box-line diagram, or node-arc structure. It is defined by the following formula:

$$Configuration\ Complexity = R/C$$

where $R$ is the number of relationships and $C$ is the number of components.

The size metric is defined in Allen et al. (2007) as the sum for all the sizes of all the disjoint components or nodes in a system using the following equation:

$$Size(S) = \sum_{e \in E}^{n} Size\left(m_e\right)$$

where $n$ is the number of elements, $e$ is the element that belongs to the component $E$, and $m$ is the module inside the component.

However, two evaluation methods that were most suitable to our work were Narasimhan and Hendradjaya's (2007) evaluation suite, and Zayaraz and Thambidurai's (2008) measurement techniques. Those two evaluation methods were comprehensive and the equation parameters can be readily extracted from the component-based diagrams. Narasimhan and Hendradjaya (2007) presented a suite for measuring the integration of software components. The metrics involved are complexity, criticality, triangular and dynamic metrics. In our evaluation, we discarded dynamic metrics because they are designed to test applications during runtime. Zayaraz and Thambidurai (2008) presented a technique for quantifying and measuring software quality. The technique is built on top of COSMIC Full Function Points (CFFP) and ISO 9126 quality standards. They have incorporated both CFFP and ISO 9126 quality standards at the architectural level. The notation they used is presented in Table 1 and will be used in equations throughout the paper. Details about the metrics we used will be discussed in more details in the coming section.

**Table 1** Zayaraz and Thambidurai's notation

| *Parameter* | *Notation* |
|---|---|
| Entry | E |
| Exit | X |
| Read | R |
| Write | W |
| Number of components | N |
| Layer | L |

In the next section, we discuss previous research that we have done for generating pervasive system architectures including the different milestones the research has passed through.

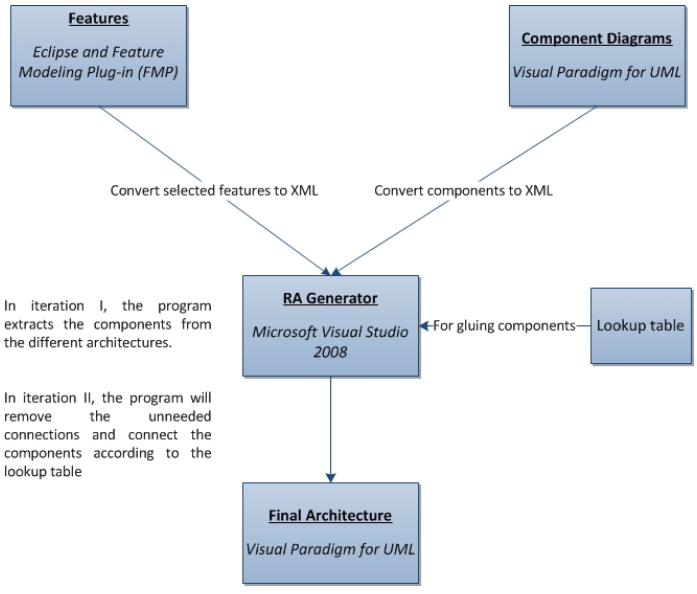
## 4 Overview of architecture generation

In our previous work (Hamza, 2011), we incrementally tackled the problem of defining an architecture for pervasive systems in three phases: first, we collected various architectures of pervasive systems from the literature. Second, we extracted their main features and categorised them. Third, we devised a methodology, by which we can generate pervasive systems' architectures from a set of features selected by designers. (The work we report on herein is considered a fourth phase where we evaluate the generated architectures against similar ones developed by subject matter experts.)

The considered feature-set was gathered from a survey of over 50 state of the art architectures specified in the literature for various kinds of pervasive systems. Examples of such features included but were not limited to: context awareness, privacy, mobility and fault tolerance. For each of the indicated features, we gathered, and sometimes developed, relevant components that could be aggregated together, based on designer's choice, to produce a new system architecture.

The process of generating architectures is shown in Figure 4. The required features of a given system are first selected by a system architect. Each selected feature is then mapped to a best practice micro architecture (consisting of a set of interdependent components). The resulting group of micro architectures is then merged into a large reference architecture. The components in the selected micro architectures are all included in that reference architecture. However, they are not initially glued together. In order to glue them to form a larger architecture representing the entire system, we pre-define a map of dependencies amongst the feature-related micro architectures in a way that supports gluing them together. A dependency is identified if a component reads, writes or uses another component.

**Figure 4** The architecture generation flow (see online version for colours)




*Source:* Hamza (2011)

Implementation wise, the Feature Modeling Plug-in (FMP) in http://gsd.uwaterloo.ca/fmp, developed for Eclipse, was used to visualise and select the features. Visual Paradigm for UML (http://www.visual-paradigm.com/product/vpuml) was used to represent the feature related architectures, which were exported in XML format. The generation process was driven by a C# program that is called RA Generator (Hamza, 2011). It works by extracting the components that map to the selected features from the reference architecture. Then, it enhances the final architecture by removing the unneeded connections and adding other ones between the different components that need to be connected together. The removal algorithm works by deleting the unneeded connections if two components are connected together and only one of them is included in the new system. This is achieved by checking if a component has a loose connection from its end. A manually pre-defined lookup table that contained the components needed to be connected together is used for adding new connections between the components coming from different architectural designs. More details on the feature set and the methodology for generating architectures can be found in Hamza and Aly (2010), Hamza et al. (2011) and Hamza (2011).

Next, we will explain briefly the case studies that were performed to evaluate the generated architectures.

## 5 Case studies

In order to evaluate our approach for generating a pervasive architecture, we compare generated architectures and ones designed by subject matter experts in three case studies. The experts are from industry and academia with experience ranging from three-five years in designing software systems. In this section, we discuss the software systems that are involved in the three case studies. The first system, which will be referred as case 1 from now on, implements context awareness in retails systems. During a shopping experience, a shopper is notified of surrounding shoppers with common interests. Such system could be smart enough to detect if a shopper is alone or not, and accordingly, notifies the shopper with possible common interests with surrounding shoppers. It provides the shopper with different promotions and reviews once he/she places a product in a shopping cart. The user is expected to have network access to check those reviews if needed.

The second system, or case 2, is one for elders and people with health conditions that require monitoring. According to their conditions, patients could be notified of the nearest pharmacy, clinic or hospital according to the criticality of the situation. The physicians monitoring the cases get notified with their health status.

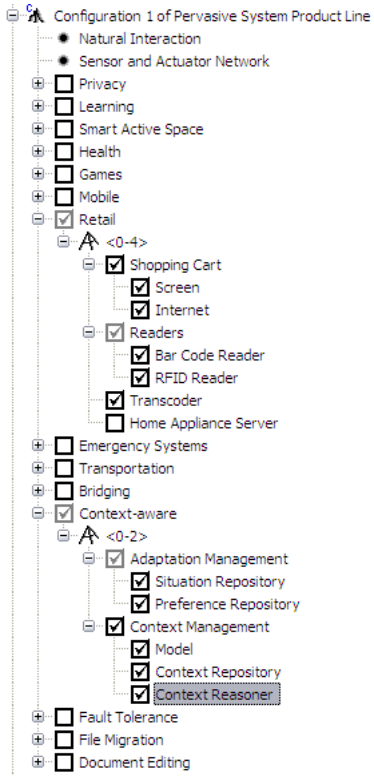
The third system, or case 3, is for transportation. Users are notified with the alternative routes while driving in case of traffic congestions. They register their destination once they get in the vehicle. All such data is collected from surrounding drivers given street capacity in mind; drivers are re-routed with the most efficient path. Users can use their devices for registering their position and their destination. The system integrates with the legacy transportation systems (such as cameras, radars … etc.) for collecting regular updates about traffic status in the streets. Full details of the requirements of those three systems can be found in Hamza (2011, pp.183–188).

In the coming section, we discuss the selected features that were used to generate a reference architecture for case 1 and discuss their main components. We do not discuss cases 2 and 3 with similar details for brevity.

## 6 Feature model

In this section, we discuss the feature model that was designed for Case 1 and how it was translated into components in a generated architecture. The features in the model were chosen based on the system requirements discussed earlier (Section 5). Some of the features are mandatory to be in the system to bring in pervasiveness, which are: Natural Interaction and Sensor Networks (shown in Figure 5 with solid dots). The rest of the features are optional and selected based on requirements. For example, we chose shopping cart, readers (barcode and RFID), transcoder, context management and adaptation management. The selection of those features is based on the subjective assessment of the designer who interprets the requirements. Notice that when we categorised the features, we focused on keeping each one self-contained enough such that it does not overlap with other features in order to minimise subjectivity during its selection as much as possible.

**Figure 5** Selected features for case 1 (see online version for colours)




*Source:* Hamza et al. (2011)

Those selected features are then mapped to components (of their respective micro architectures) that build up the system. The main components are: actor, application tier subsystem, device interaction subsystem, shopping cart and backend subsystem. The actor component is responsible for keeping the identity of a user along with his/her profile, which maps to the natural interaction feature in the feature model. It is connected with the application tier subsystem, which is a link between the actor, the devices interaction subsystem and the shopping cart components. It is responsible for managing the shopping cart and handling the notifications delivered to the actor. Furthermore, the devices interaction subsystem component provides the location services, the tracking subsystems, the communication infrastructure and the context management, which are also used to bring pervasiveness to the system as discussed in the requirements. The shopping cart is a key component in the retail scenario. It could be equipped with internet connection, screen and/or transcoder to enhance the shopping experience. The shopping cart tracks the products put inside it and communicate that with the backend subsystem through the goods tracking subsystem. The selection of components from the reference architecture and how they are connected together is discussed in details in Hamza (2011).

The selected components participating in the system are loosely coupled and can be replaced easily in case they are obsolete or better components become available. The replacement may be made in the reference architecture directly in case it is needed only for one system.

The subsequent section will discuss the setup used for running the experiments and the techniques selected to evaluate the architectures.

## 7 Experiment design

In this section, we describe the design of an experiment we carried to evaluate the quality of our generated architectures for pervasive systems in comparison to ones that were manually created by subject matter experts. In order to evaluate the architectures, we conducted an extensive search in order to find quantifying metrics for evaluating architectures. We were also looking for acceptable ranges for each of these metrics. Unfortunately, no universally acceptable ranges were found due to the heterogeneity of the different architectures. We then resorted to evaluating our generated architectures and the ones designed manually by the subject matter experts and comparing them to each other for each metric. We used two architecture evaluation frameworks in our evaluation; namely Narasimhan and Hendradjaya's (2007) evaluation suite, and Zayaraz and Thambidurai's (2008) measurement techniques (Section 2). The former is a suite devised for measuring the integration of software components from the following perspectives: complexity, criticality, triangular metrics and dynamic metrics. The latter is a technique developed for quantifying and measuring software quality at the architectural level by measuring complexity, coupling, cohesion and other metrics. Moreover, since the architectures were defined using UML component diagrams, we used the SDMetrics tool (http://www.sdmetrics.com) to extract some basic metrics like the number of components, interfaces, associations and other relevant elements from these component diagrams. Table 2 shows all the metrics we used. Table 3 defines the terminology we used in displaying the results with the SDMetrics tool.

**Table 2** All metrics we used in evaluating the generated architectures

| *Metric* | *Definition* |
|---|---|
| Component packing density (CPD) | It measures the packing density of the components in architecture. It is calculated by the ratio between the number of subcomponents related to a component with respect to the number of components |
| Component average interaction density (CAID) | It is used for evaluating the entire components' assembly complexity. It is calculated by the ratio between the component interaction densities to the number of components. |
| $CRIT_{link}$ | It measures if the criticality of a component in terms of the links connected to it. The initial indicator presented in this research is 8 links as a threshold value. |
| $CRIT_{Bridge}$ | Bridge component links are used to connect two or more components or applications. Importance weight should be added to each bridge link by the developer. This weight should reflect the probability for failure. |
| $CRIT_{Size}$ | It measures the size for a component. In order to specify the threshold, you choose the maximum size of a component in the system. |
| $CRIT_{All}$ | Criticality metrics is a summation for metrics $CRIT_{link}$, $CRIT_{Bridge}$ and $CRIT_{Size}$. |
| Coupling | It measures the relationship of dependency between two interacting modules. |
| Cohesion | It evaluates the tightness between the linked features composing a system or module. |
| Complexity | It is used as a metric to evaluate how the system or module is complex. |
| Modifiability | It evaluates to what extent the components could withstand changes without affecting the whole system. |
| Modularity | It evaluates if the system is built on modular basis or not. |
| Reusability | It evaluates if the components in the system can be used in another system without major changes. |

**Table 3** Terminologies used by SDMetric

| *Terminology* | *Definition* |
|---|---|
| Elements | It is the number of components and sub-components in a diagram |
| Interfaces | It is the number of interfaces that the components utilise while communication with each other |
| Associations | It is the number of associations that describe the relationship between two components |
| Deps | It is the number of dependencies in the architecture |

In order to evaluate the quality of the architectures, we executed the following process:

1 We specified a set of requirements for three cases of pervasive systems that address different domains. The requirements of the three cases were of comparable complexity. Detailed description of the cases, along with their requirements, is discussed in Section 5.

2 For each one of those three cases, we used our methodology to automatically generate an architecture through selecting features of interest.

3 We also provided the requirements of the three cases to five subject matter experts (S1-S5). Each of the experts was asked to design an architecture that satisfies each of the three sets of requirements, for a total of 15 manually designed architectures. The subject matter experts were selected with varying years of experience ranging from three to five years in the field of software and systems architecture and with sufficient knowledge in pervasive system development.

4 Using the selected metrics, we then compared the architectures that were automatically generated with those that were manually designed by the subject matter experts.

The next section will show the results of running the experiments highlighting a comparison between the results of the generated architecture and the architectures designed by the subject matter experts.

## 8 Experiment results

In this section, we present the results of carrying the case studies described in Section 5. Recall that the objective of the case studies is to compare the quality of the automatically generated architectures with that of the ones defined manually by subject matter experts involved in the study. We first used the SDMetrics tool to collect basic architecture metrics. The tool took as input a given architecture described in UML and extracted as output the number of elements, interfaces, associations, and dependencies (defined in Table 3). Tables 3, 4, 5, show the output of the SDMetrics tool for the automatically generated architectures and for those created by the experts (S1-S5) for each of the three cases (1) to (3).

**Table 4** SDMetrics diagram output for case 1

| *Case 1* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| Elements | 61 | 23 | 43 | 43 | 21 | 33 |
| Interfaces | 4 | 0 | 0 | 0 | 0 | 0 |
| Associations | 11 | 8 | 20 | 12 | 4 | 6 |
| Deps | 7 | 1 | 0 | 7 | 7 | 2 |

**Table 5** SDMetrics diagram output for case 2

| *Case 2* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| Elements | 47 | 23 | 47 | 39 | 26 | 48 |
| Interfaces | 3 | 0 | 0 | 0 | 0 | 0 |
| Associations | 10 | 9 | 22 | 9 | 6 | 8 |
| Deps | 4 | 0 | 0 | 5 | 8 | 3 |

**Table 6** SDMetrics diagram output for case 3

| *Case 3* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| Elements | 44 | 19 | 37 | 34 | 24 | 44 |
| Interfaces | 2 | 0 | 5 | 0 | 0 | 0 |
| Associations | 8 | 5 | 5 | 9 | 4 | 10 |
| Deps | 4 | 3 | 5 | 4 | 9 | 1 |

The metric values from the SDMetrics tool indicate that the generated architectures contain relatively more elements (components and subcomponents) than those of the experts. The reason behind this relates to the fact that the reference architecture was designed with loosely coupled components such that removing a component would have the least impact on the whole system. However, when the experts were provided with the requirements, they were given general directions to design an efficient system that best addresses the requirements. The number of interfaces reflects how many different components are communicating together. In the generated architectures, we are integrating components from different categories. So, the presence of interfaces was a must to make sure that components can communicate with each other. On the other hand, interfaces were barely used by the experts which justifies that the experts were not concerned much about the reusability aspect of the components as us. We predict that the differences in the number of elements and interfaces will affect the coupling and cohesion metrics. Also, associations and dependencies are close to each other except for S2 in cases 1 and 2, in which we predict that modularity, complexity and criticality metrics will be affected as well.

Next, we used the output of the SDMetrics tool (i.e., the number of elements, interfaces, associations, and dependencies) for each of the three cases as input to the Narasimhan and Hendradjaya's evaluation suite to produce values for six different metrics (CPD, CAID, $CRIT_{link}$, $CRIT_{Bridge}$, $CRIT_{Size}$, $CRIT_{All}$), as shown in Table 7 for case 1. The output of the SDMetrics tool was also fed into the Zayaraz and Thambidurai measurement techniques to produce values for another set of six metrics (coupling, cohesion, complexity, modifiability, modularity, reusability), as shown in Table 8 for case 1 as well. For more information about the equations of the used metrics, the interested reader is referred to Narasimhan and Hendradjaya (2007) and Zayaraz and Thambidurai (2008).

**Table 7** Narasimha, Hendradjaya's evaluation suite for case 1

| *Metric* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| CPD | 0.63 | 0.75 | 0.91 | 0.79 | 1.10 | 0.32 |
| CAID | 0.10 | 0.17 | 0.13 | 0.32 | 0.20 | 0.19 |
| $CRIT_{link}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $CRIT_{Bridge}$ | 4 | 1 | 6 | 4 | 2 | 2 |
| $CRIT_{Size}$ | 0 | 0 | 1 | 0 | 0 | 0 |
| $CRIT_{All}$ | 4 | 1 | 7 | 4 | 2 | 2 |

**Table 8** Zayarazand Thambidurai's measurement techniques for case 1

| *Metric* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| Coupling | 0.31 | 0.25 | 0.07 | 0.5 | 0.8 | 0.26 |
| Cohesion | 0.97 | 0.83 | 0.24 | 0.8 | 0.94 | 0.47 |
| Complexity | $42 \times 10^{-5}$ | $10 \times 10^{-5}$ | $5 \times 10^{-6}$ | $35 \times 10^{-5}$ | $8 \times 10^{-6}$ | $16 \times 10^{-5}$ |
| Modifiability | $2 \times 10^{4}$ | $10 \times 10^{3}$ | $2 \times 10^{6}$ | 402.6 | 125.5 | $6 \times 10^{3}$ |
| Modularity | 12.88 | 1.93 | 3.9 | 4.7 | 2.9 | 4.91 |
| Reusability | 16.08 | 5.93 | 18.5 | 6.9 | 4.3 | 8.78 |

In the remainder of this section, we analyse the results obtained for each metric, making a comparison between our automatically generated architecture and the ones created by the subject matter experts.

## 8.1 Component packing density

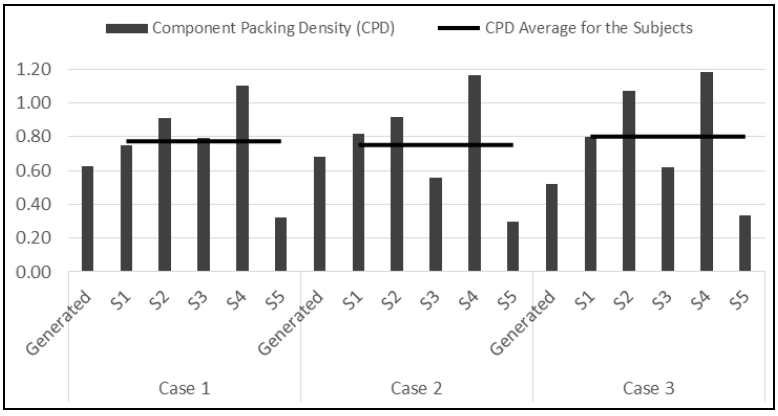
Component packing density (CPD) measures the packing density of the components in the architecture. Therefore, the higher the CPD, the more complex the system is. CPD is directly proportional to the number of interfaces, associations and dependencies between the components and inversely proportional to the number of components. The equation is:

$$CPD_{constituent_{type}} = \frac{\#\ constituent}{\#\ components}$$

where #*constituent* could be: LOC, object/classes, operations, classes and/or modules in the related components, and #*components* is the number of the components.

Figure 6 shows the CPD for cases 1, 2 and 3. The dotted lines display the average for each case between the experts. For case 1, the CPD value for the generated architecture is 0.63 while the average for the experts is 0.77. According to Table 4, although the generated architecture has the highest number of interfaces, associations and dependencies as well as the highest number of components, it still gets lower ratio. The CPD is calculated by the ratio between the number of interfaces, dependencies and associations in the architecture to the number of components. In case 2, the CPD for the generated architecture is 0.68 while the average CPD for the experts is 0.75. Interfaces, associations and dependencies are high but not the highest according to Table 5. This makes the generated architecture more complex than S3 and S5. Finally in case 3, the CPD for the generated architecture is 0.52 while the average for the experts is 0.80. From the above mentioned numbers, we conclude that the three generated architectures have less CPD than the average of the experts.

**Figure 6** CPD for cases 1, 2 and 3



## *8.2 Component average interaction density*

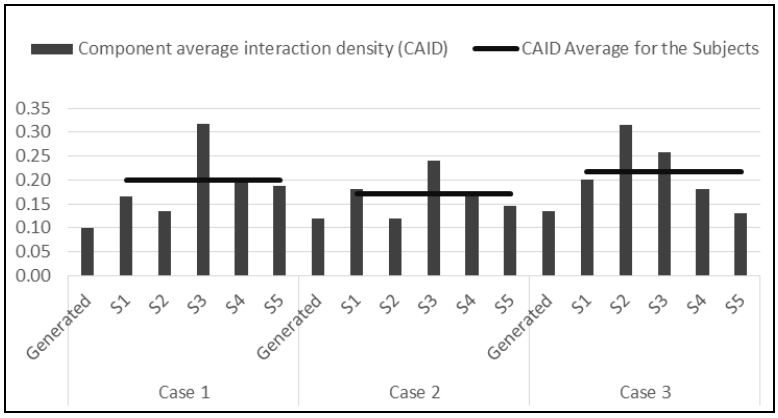
Component average interaction density (CAID) is calculated by the summation of component interaction density (CID) over the number of components. CID is calculated by defining the ratio between the actual numbers of interactions (associations) to the available number of interactions in a component. The actual number of interactions refers to the interfaces provided by a component and used by another one. However, the available number of interactions refers to the total number of interfaces provided by a component, regardless of being utilised or not. Hence, the lower the value of CAID, the fewer the interactions are in the architecture, which means simpler communications in the system. The equation used is:

$$CAID = \frac{\sum_n CID_n}{\#components}$$

where $\sum_n CID_n$ is the summation for all the interaction densities for components 1 to n and #*components* is the number of the existing component in the real system.

Figure 7 shows the CAID calculated for the cases 1, 2 and 3. In case 1, the CAID for the generated architecture has the value of 0.1 while the average is 0.2 for the experts. In case 2, the CAID value is 0.12 for the generated architecture, while the average among the experts is 0.17. For case 3, the CAID is 0.13 for the generated architecture and the average is 0.22 for the experts. From the above results, we conclude that the generated architectures have fewer interactions than the ones designed by the experts due to the small number of interactions between the subcomponents in a component when compared to the architectures defined by the experts. This is because the packaged components defined for the features are well-designed and efficient.

**Figure 7** CAID for cases 1, 2 and 3




## *8.3 Criticality all ($CRIT_{All}$)*

The criticality metric is used to measure the number of critical components in a system. A critical component is one, without which, other components in a system are not able to interact. The more critical components exist in a system, the higher the tendency for its

failure. We present the overall criticality $CRIT_{All}$ metric, which is the sum of the link criticality, bridge criticality, inheritance criticality and size criticality metrics. The equation used for criticality is:

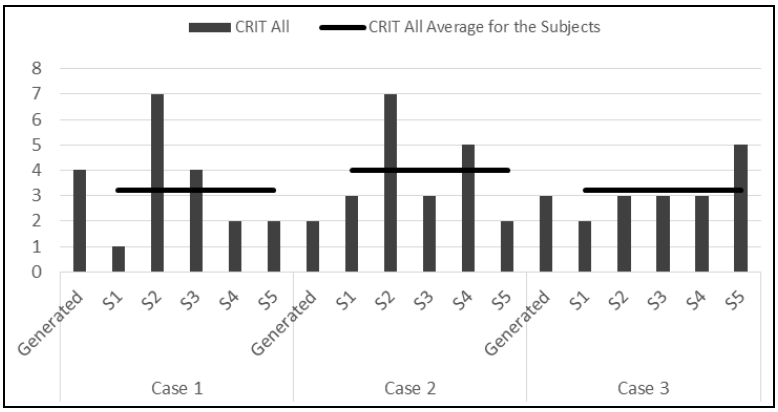
$$CRIT_{all} = CRIT_{link} + CRIT_{bridge} + CRIT_{inheritance} + CRIT_{size}$$

Link criticality refers to a component that has many connections, exceeding a pre-defined threshold. The failure of this component may cause the failure of a major functionality or the whole system. Bridge criticality reflects the existence of components that act as bridges for other components. If a bridge component fails, it can lead to the failure of the whole system or a core functionality. Inheritance criticality counts the number of base components or elements inherited by others. The higher the count, the higher the risk of failure is. Size criticality measures the size of a component including subcomponents, classes, functions...etc. A threshold is set as a maximum value of the size. If the size exceeds that threshold, then there is high risk of component failure. In Figure 8, case 1 scored 4 for $CRIT_{All}$ for the generated system which is higher than the average for the experts which is 3.2. The generated architecture is worse than the average of the experts. This is because bridge criticality for the generated architecture is high, as shown in Table 9. For case 2, the $CRIT_{All}$ is 2 for the generated system while the average is 4. For case 3, the $CRIT_{All}$ is 3, while the average is 3.2. Bridge criticalities in the generated architecture, for case 2 and case 3, are minimum; that's why the overall criticality is low. These cases do not suffer from the same issue of case 1 because the latter has many bridge components in the generated system coming from the reference architecture. Bridge components in this case are needed between the shopping cart and the device interaction subsystem due to the required communication between them.

**Table 9** Narasimhan and Hendradjaya's evaluation suite for case 1

| *Metric* | *Generated* | *S1* | *S2* | *S3* | *S4* | *S5* |
|---|---|---|---|---|---|---|
| $CRIT_{link}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $CRIT_{Bridge}$ | 4 | 1 | 6 | 4 | 2 | 2 |
| $CRIT_{Size}$ | 0 | 0 | 1 | 0 | 0 | 0 |
| $CRIT_{All}$ | 4 | 1 | 7 | 4 | 2 | 2 |

**Figure 8** $CRIT_{All}$ for cases 1, 2 and 3



## *8.4 Coupling*

Coupling measures the interdependencies between components. The lower the coupling, the better the architecture is and vice versa. The equation for coupling is:

$$SC_p = \sum_{i=1}^{L-1} \frac{E_{(i,i+1)} + X_{(i,i+1)} + R_{(i,i+1)} + W_{(i,i+1)}}{2 \times N_i + N_{i+1}}$$

Figure 9 shows a comparison of coupling scores between the generated architecture and the human designed architectures. The dotted line shows the average of the experts. In case 1, the coupling is 0.31 while the average for the experts is 0.36. In case 2, the coupling is 0.21 and the average is 0.30. In case 3, coupling is 0.40 and the average is 0.27. In this last case, our design is deviating from the average by 27.5%. According to our analysis, case 3 has the worst coupling because it has the highest number of layers (more on this below), hence the highest number of interdependencies. Figure 10 shows the number of layers, number of components in each layer and the number of layer interdependencies (e.g., entries, exists, reads and writes) for cases 1, 2 and 3.

**Figure 9** Coupling for cases 1, 2 and 3

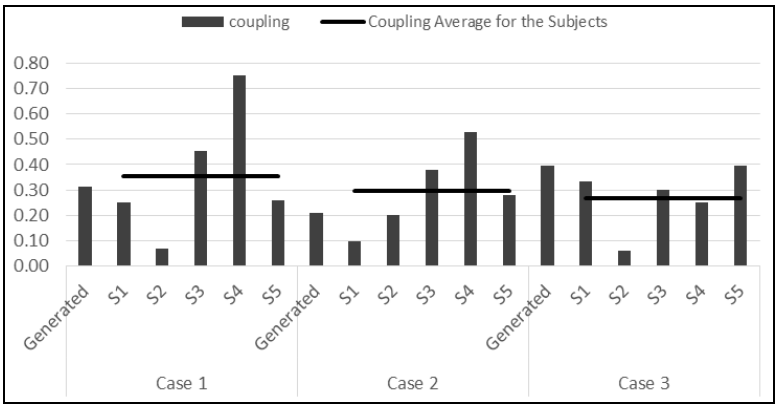
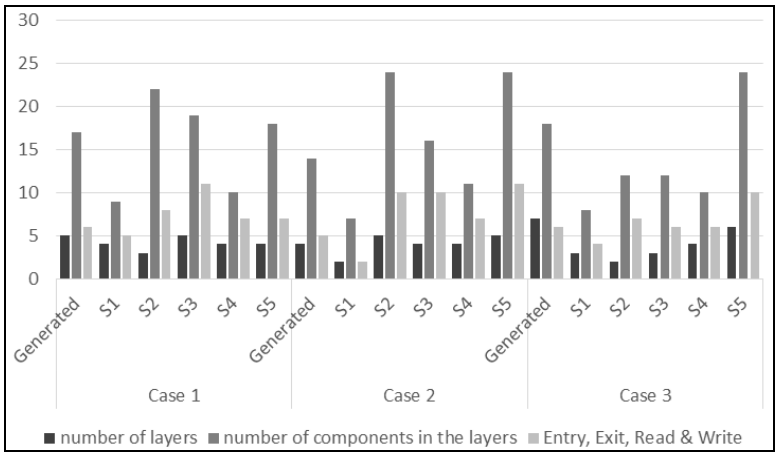



**Figure 10** Coupling/cohesion computational parameters for cases 1, 2 and 3



The research attempted to investigate why coupling of the generated architectures was better in some cases, and worse in others. Coupling is linked to the number of layers (e.g., data access, sensors, actuators and communication components) in the system and the number of components in each layer. The number of layers involved in the architecture is directly proportional to the coupling, while the number of components inside each layer is inversely proportional to the coupling. It is also affected by the number of interactions (entries, exits, reads and writes) between the components. The more the interactions, the higher the coupling in the architecture is. In cases 1 and 2, the coupling is lower than the average for the generated architecture; however, in case 3 the coupling is higher than the average. The reason behind that is that the number of layers in case 3 is the highest causing the coupling to be the worst. This is due to expert 2 designing the system to interact with many levels of legacy systems in the transportation architecture without utilising interfaces and wrappers.

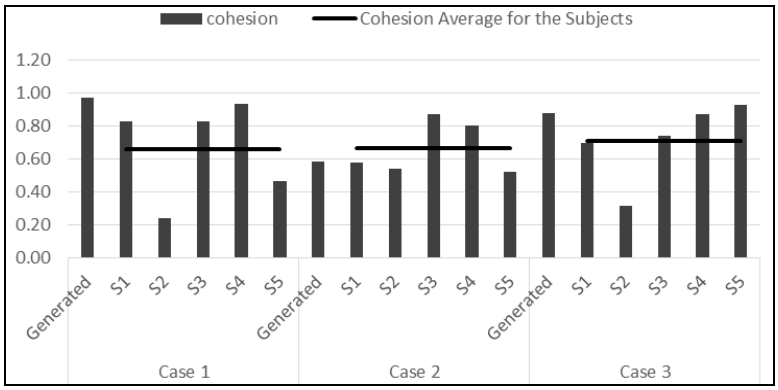
## *8.5 Cohesion*

Cohesion measures the level of strength and unity between different components within a module. The higher the cohesion, the better the architecture is and vice versa. The equation used for cohesion is:

$$SC_o = \sum_{i=1}^{L} \frac{E_i + X_i + R_i + W_i}{N_i^2}$$

Like coupling, cohesion is affected by the number of entries, exits, reads and writes between the components within a layer and the number of components in a layer according to Zayaraz and Thambidurai's (2008) measurement techniques. For cases 1, 2 and 3, Figure 11 shows the cohesion of the generated architectures. For case 1 the cohesion is 0.97, which is the highest cohesion among all the other architectures, while the average for the experts has a value of 0.66. This is due to the generated architectures have a high number of layers with many entries, exists, reads and writes, as shown in Figure 10. For case 2, the cohesion for the generated architecture is 0.58, which is below the average which is 0.60. S3 has the highest cohesion because of the high number of layers with many entries, exists, reads and writes. For case 3, the generated architecture is 0.88. It is higher than the average, which is 0.71.

**Figure 11** Cohesion for cases 1, 2 and 3



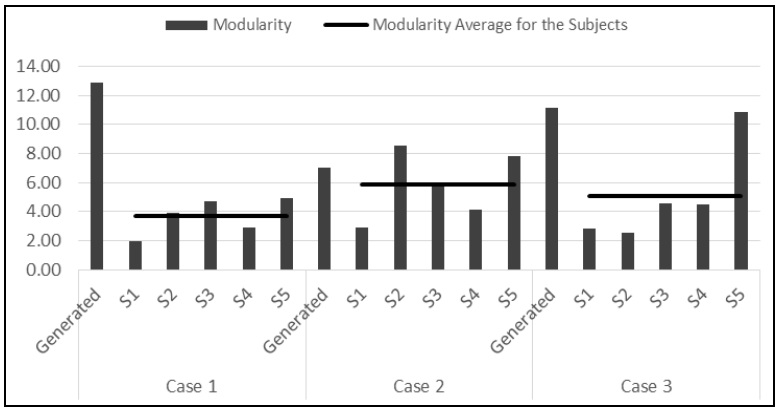
## *8.6 Modularity*

Modularity is used to evaluate if the system is built on modular basis or not. A module is a component or set of components that are self-contained in a system. Changes to a module should not affect the functionality of the rest of the system. It is directly proportional to cohesion and inversely proportional to coupling. The equation used for modularity is:

$$Modularity = \sum_{i=1}^{L-1} \frac{\left( \frac{Co_i + Co_{i+1}}{2} \right)}{Cp_i}$$

The generated architecture showed the highest modularity of 12.9 for case 1 and 11.7 for case 3, as shown in Figure 12. This is because the generated architecture in each of those two cases has high cohesion and low coupling. However, in case 2, the modularity of the generated architecture, while still higher than the experts' average, is low because cohesion is not high. The average scores for the human-designed architectures for case 1, case 2 and case 3 are much lower: 3.67, 5.83 and 5.04, respectively.

**Figure 12** Modularity for cases 1, 2 and 3




## *8.7 Reusability*

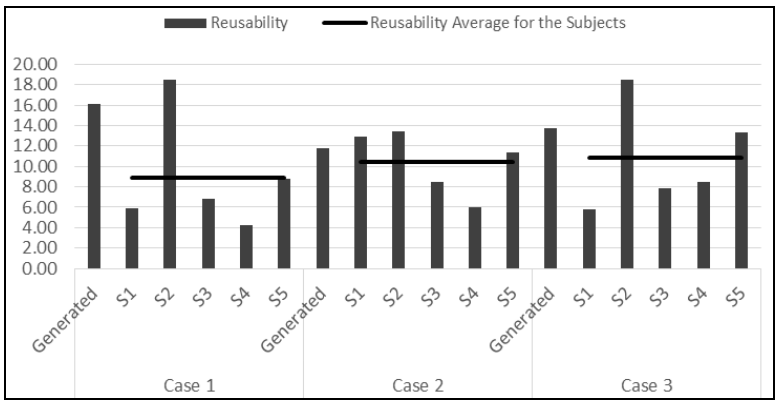
Reusability is used to measure how the components in the system can be used in another one without major changes. It is directly proportional to modularity and inversely proportional to coupling. The equation used is:

$$Reusability = Modularity + \frac{1}{\sum_{i=1}^{L-1} Cp_i}$$

Figure 13 shows the reusability for cases 1, 2 and 3. For case 1, the generated architecture's reusability is 16.08, while the average for the experts' architectures is 8.87. In case 2, reusability is 11.80 for the generated architecture and the average is 10.45. In case 3, the generated architecture scores a reusability of 13.69, while the average is 10.81. We observe that in the three cases, the reusability was high and above average for

the generated architectures. One noticeable anomaly is that S2 scored the highest reusability in all the cases because it achieved almost consistently the lowest coupling.

**Figure 13** Reusability for cases 1, 2 and 3




## 8.8 Complexity

Complexity is affected by the number of entries, exits, reads and writes among the components in a layer and among the layers themselves. It is calculated as the summation of intra-complexity and inter-complexity. The equations used are:

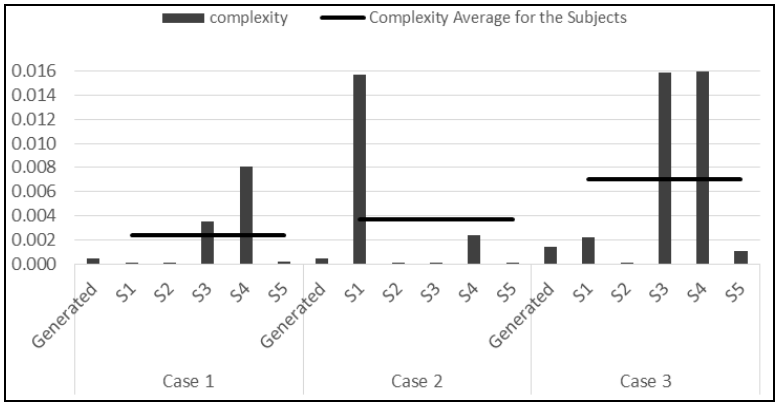
$$Intra^{Cx} = \sum_{i=1}^{L} \frac{\left(E_i \times X_i\right)^2}{\left(N_i^2 \times \left(N_i \times \left(N_i - 1\right)\right)\right)^2}$$

$$Intra^{Cx} = \sum_{i=1}^{L-1} \frac{\left(E_{(i,i+1)} \times X_{(i,i+1)}\right)^2 + \left(R_{(i,i+1)} \times W_{(i,i+1)}\right)^2}{\left(N_i \times N_{i+1}\right)^4}$$

$$SC_x = Intercomplexity + Intracomplexity$$

The equations are discussed in more details in Hamza (2011). Intra-complexity measures the complexity among the components within a layer. Intra-complexity is proportional to the number of entries and exists between the components inside the layer. While inter-complexity measures the complexity among the layers and is directly proportional to the number of entries, exits, reads and writes between the components in a layer with the other external components to that layer. We found the complexity of the architectures low in all three cases, as shown in Figure 14. For case 1, case 2 and case 3, the generated architecture has a complexity of 0.00042, 0.00047 and 0.00145, respectively. On the other hand, the averages for cases 1, 2 and 3 are much higher: 0.00238, 0.00367 and 0.00704, respectively. This is due to the very low inter-complexity for the generated architectures resulting from a small number of entries, exits, reads and writes between the components in a layer and external layers.

**Figure 14** Complexity for cases 1, 2 and 3



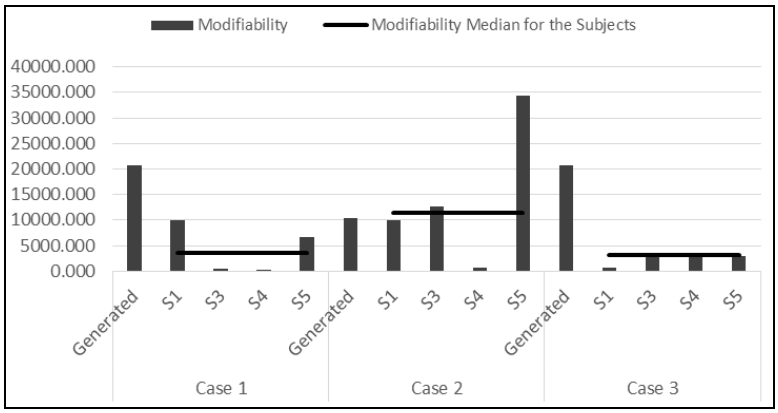

## *8.9 Modifiability*

Modifiability measures how much modifications can be done to the modules and components of a system without affecting the others. Modifiability is inversely proportional to coupling and inter-complexity as given by the equation:

$$Modifiability = \frac{1}{\sum_{i=1}^{L-1} Inter^{Cx_i}} + \frac{1}{\sum_{i=1}^{L-1} Cp_i}$$

Figure 15 shows the modifiability for the cases 1, 2 and 3. The majority of the experts' architectures and the generated architectures show high modifiability for the architectures. For case 1, the generated architecture scored 20,739, for case 2, 10,373, and 20,739 for case 3. In all three cases, S2 scored the highest modifiability due to its lowest coupling and inter-complexity for the architectures. The next highest modifiability is for the generated architecture due to its low inter-complexity. For modifiability, we removed S2 from the graphs because it is biased and considered an outlier due to the big difference between the number ranges.

**Figure 15** Modifiability for cases 1, 2 and 3



## 8.10 General analysis

We calculated the average and the standard deviation (as a percentage) of the results obtained in the three cases, for the generated architectures as well as for all the experts, in order to generalise the analysis. We divided the metrics into two categories, positively monotonic and negatively monotonic metrics. The positively monotonic metrics are those that indicate better results with higher values, while the negatively monotonic metrics indicate better results with lower values. In Figure 16, we show a comparison of the positively monotonic metrics-cohesion, modularity and reusability – between the generated architectures and those designed by the experts. We found that the generated architectures consistently showed better performance. For example, the generated architecture scored 18.8% higher than the experts' for cohesion: 53.2% higher for the modularity and 27.5% higher for reusability. Similarly, Figure 17 shows a comparison of the negatively monotonic metrics-complexity, cohesion, CPD, CAID and $CRIT_{All}$. Again, we see that the generated architectures consistently showed better performance (lower values). For example, the generated architectures scored 0.5% less than the experts' for coupling, 82.18% less for complexity, 21.5% less for CPD, 39.9% less for CAID and 13.5% less for $CRIT_{All}$. We also calculated the standard deviation for the generated architectures and for the experts' architectures, as shown in Table 10. The standard deviations for the generated architectures is much lower than the experts' which implies that the quality of the generated architectures is higher.

We find that the evaluation metrics we chose can predict the quality of the generated architectures by measuring the level of maturity of the reference architecture generated from a feature model. In case the evaluation showed some deficiencies in one or more evaluation aspects, then, this can be an early indication of the kind of problems that could be in the components or the features that brought them in.

The following section will highlight the future work that can be carried out as a further step to enrich this study towards having a complete reference architecture for pervasive systems.

**Figure 16** Positively monotonic metrics

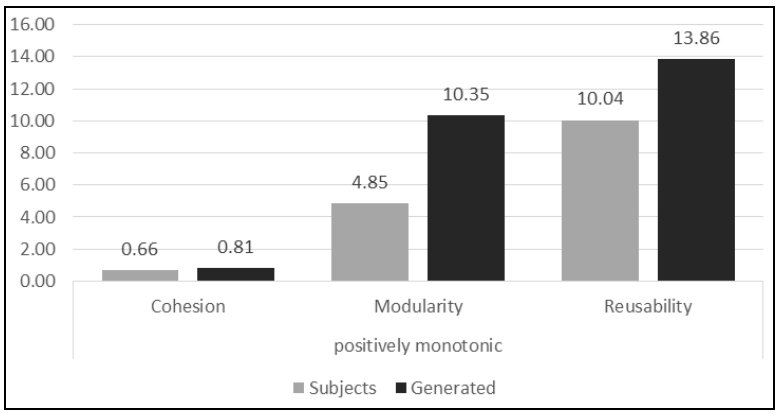


**Figure 17** Negatively monotonic metrics

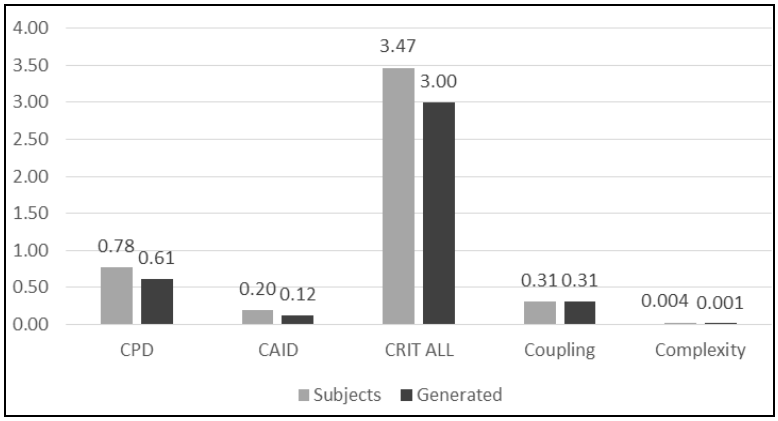



**Table 10** Relative standard deviation % for the generated and the experts

| *Metric* | *Negatively monotonic* | | | | | *Positively monotonic* | | |
|---|---|---|---|---|---|---|---|---|
| | *CPD* | *CAID* | *CRIT all* | *Coupling* | *Complexity* | *Cohesion* | *Modularity* | *Reusability* |
| Generated | 13.5% | 15.2% | 33.3% | 30.7% | 74.7% | 25.1% | 29.2% | 15.5% |
| Experts | 38.6% | 31.4% | 52.1% | 59.1% | 145.0% | 32.9% | 51.1% | 44.6% |

## 9 Future work

The evaluation of generated architectures for pervasive systems presented in this paper has room for improvement. The following are some suggestions that will enhance the evaluation:

- Steps of the evaluation process used in this research were integrated manually. Automating the evaluation process by having a tool accept an architecture as input, getting the metrics results and their averages and standard deviation as output will allow a more efficient way of carrying the evaluation.
- Repeating the analysis on more case studies of other pervasive systems will help in having more mature evaluation model and more statistically relevant results.
- Finding more metrics that can help measure more aspects of the architecture will also help strengthening the evaluation results

## 10 Conclusions

In this paper, we gave a detailed description of a methodology to automatically generate pervasive systems' architectures. By utilising SPL concepts, best-practice micro architectures can be extracted from previously developed pervasive systems and mapped to categorised features. By selecting a subset of these features, a reference architecture for a new pervasive system can be generated by merging the micro architectures of the

selected features. The core of this paper is targeting the evaluation methodology for such generated architectures. We conducted an extensive literature search to find quantifying metrics for evaluating high level architectures. We used Narasimhan and Hendradjaya's evaluation suite and Zayaraz and Thambidurai's measurement techniques to empirically evaluate our generated architectures. The generated architectures were compared to similar ones generated by subject matter experts in the domain. The metrics used for evaluation are coupling, cohesion, complexity, reusability, adaptability, modularity, modifiability, packing density, and average interaction density. Our results demonstrate that our generated architectures were comparable in quality and for the most part better than those generated by subject matter experts.

## References

Allen, E.B., Gottipati, S. and Govindarajan, R. (2007) 'Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach', *Software Quality Journal*, June, Vol. 15, No. 2, pp.179–212.

Barnard, J. (1998) 'A new reusability metric for object-oriented software', *Software Quality Control*, Vol. 7, No. 1, pp.35–50.

Bragança, A. and Machado, R.J. (2005) 'Deriving software product line's architectural requirements from use cases: an experimental approach', *Proceedings of the 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software – MOMPES'05 (Within the 5th IEEE/ACM International Conference on Application of Concurrency to System Design - ACSD 2005)*, June, pp.77–91, Rennes, France, UCS General Publication No. 39, Turku, Finland.

Briand, L., Morasca, S.and Basili, V. (1996) 'Property-based software engineering measurement', *IEEE Transactions on Software Engineering*, Vol. 22, No. 1, pp.68–86.

Enterprise and Solution Architect Certification & Resources, 'Can we measure architecture?', Interview with Anja Fiegler [online] http://grahamberrisford.com/15%20Scale%20and%20Change/Can%20we%20measure%20architecture.htm.

Feature Modeling Plug-in (FMP) *An Eclipse Plug-In for Editing and Configuring Feature Models* [online] http://gsd.uwaterloo.ca/fmp (accessed May 2010).

Ferscha, A. (2003) 'Coordination in pervasive computing environments', *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, June, pp.3–9, IEEE Computer Society, Washington, DC.

Gonzalez, S.T. (2007) *Feature Oriented Model Driven Product Lines*, PhD thesis, March, School of Computer Sciences, University of the Basque Country.

Hamza, M. (2011) *Feature-Based Generation of Pervasive Systems' Architectures Utilizing Software Product Line Concepts*, Masters thesis, December, School of Science and Engineering, The American University in Cairo [online] https://dar.aucegypt.edu/bitstream/handle/10526/2606/Mostafa%20Hamza%20Masters%20Thesis%20-%20Feature-based%20Generation%20of%20Pervasive%20Systems%20Architectures%20Utilizing%20Software%20Product%20Line%20Concepts.pdf?sequence=1 (accessed March 2014).

Hamza, M. and Aly, S.G. (2010) 'A study and categorization of pervasive systems architectures towards specifying a software product line', *Software Engineering Research and Practice (SERP 2010)*, Las Vegas, Nevada, USA, 12–15 July, pp.635–641.

Hamza, M., Aly, S.G. and Hosny, H. (2011) 'An approach for generating architectures for pervasive systems from selected features', *Software Engineering Research and Practice (SERP 2011)*, Las Vegas, Nevada, USA, 18–21 July.

Heineman, G.T. and Councill, W.T. (2001) *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Hitz, M. and Montazeri, B. (1995) 'Measuring coupling and cohesion in object-oriented systems', *Proc. Int'l Symp. Applied Corporate Computing (ISACC '95)*, Monterrey, Mexico, 25–27 October.

Hunt, J.M. (2006) 'Organizing the asset base for product derivation', *10th International Software Product Line Conference*, 21–24 August, pp.65–74, IEEE Computer Society.

Liu, X. and Wang, Q. (2005) 'Study on application of a quantitative evaluation approach for software architecture adaptability', *Fifth International Conference on Quality Software (QSIC'05)*, pp.265–272, QSIC.

Machado, R.J., Fernandes, J.M., Monteiro, P. and Rodrigues, H. (2005) 'Transformation of UML models for service-oriented software architectures', *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005: ECBS'05*, April, pp.173–182.

Narasimhan, V.L. and Hendradjaya, B. (2007) 'Some theoretical considerations for a suite of metrics for the integration of software components', *Information Sciences: an International Journal*, February, Vol. 177, No. 3, pp.844–864.

Pohl, K., Böckle, G. and van der Linden, F.J. (2005) *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, New York, Inc., Secaucus, NJ.

Pressman, R.S. (2001) *Software Engineering: A Practitioner's Approach*, 5th ed., The McGraw-Hill Companies, Inc., New York.

SDMetrics, *The Software Design Metrics tool for UML* [online] http://www.sdmetrics.com (accessed April 2011).

Van der Liden, F., Schmid, K. and Rommes, E. (2007) *Software Product Lines in Action, the Best Industrial Practice in Product Line Engineering*, Springer, Berlin, Heidelberg.

Visual Paradigm for UML [online] http://www.visual-paradigm.com/product/vpuml (accessed August 2011).

Weiser, M. (1991) 'The computer for the 21st century', *Scientific American*, Vol. 265, No. 3, pp.94–104.

Yared, R. and Défago, X. (2003) 'Software architecture for pervasive systems', *Journées Scientifiques Francophones (JSF)*, November, Tokyo, Japan.

Young, T.J. (2005) *Using Aspect J to Build a Software Product Line for Mobile Devices*, MSc dissertation, Univ. of British Columbia.

Zayaraz, G. and Thambidurai, P. (2008) 'COSMIC FFP based quality measurement and ranking framework for software architectures', *Software Quality Professional Journal*, March, Vol. 30, No. 5, pp.1–5, American Society for Quality, USA.